HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication

Ajit Mathew *
Virginia Tech
ajitm@vt.edu

Changwoo Min
Virginia Tech
changwoo@vt.edu

ABSTRACT

Increased capacity of main memory has led to the rise of in-memory databases. With disk access eliminated, efficiency of index structures has become critical for performance in these systems. An ideal index structure should exhibit high performance for a wide variety of workloads, be scalable, and efficient in handling large data sets. Unfortunately, our evaluation shows that most state-of-the-art index structures fail to meet these three goals. For an index to be performant with large data sets, it should ideally have time complexity independent of the key set size. To ensure scalability, critical sections should be minimized and synchronization mechanisms carefully designed to reduce cache coherence traffic. Moreover, complex memory hierarchy in servers makes data placement and memory access patterns important for high performance across all workload types.

In this paper, we present HydraList, a new concurrent, scalable, and high performance in-memory index structure for massive multi-core machines. The key insight behind our design of HydraList is that an index structure can be divided into two components (search and data layers) which can be updated independently leading to lower synchronization overhead. By isolating the search layer, we are able to replicate it across NUMA nodes and reduce cache misses and remote memory accesses. As a result, our evaluation shows that HydraList outperforms other index structures especially in a variety of workloads and key types.

PVLDB Reference Format:
Ajit Mathew and Changwoo Min. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. *PVLDB*, 13(9): 1332-1345, 2020.
DOI: https://doi.org/10.14778/3397230.3397232

1. INTRODUCTION

Evolution of hardware technologies has significantly affected the design of today's database systems. Increasing main memory size has led to the rise of in-memory databases [18, 24, 26, 37] and in-memory key-value stores [17, 43]. Since data is in-memory, I/O bottlenecks caused by expensive disk accesses are avoided. Moreover, significant efforts on optimizing query execution has allowed compiled transactions to remove buffer management and latching overheads [28]. This has made performance of index structures critical in modern database systems. A recent study of modern in-memory database systems shows that index lookup can contributes up to 94% of query execution time [29].

For single core machines, performance of an index structure mostly depends on the time complexity of its search operation. Index structures can be broadly classified into four classes based on their search time complexity. Hash indexes are the fastest class, having $O(1)$ lookup, but do not support range queries as they scatter keys randomly. Tree-based indexes (e.g., B-Tree) follow, usually having $O(\log n)$ lookup, where n is the number of keys stored in an index. Many variants of B-Tree have been proposed including B+ tree, k-ary tree [44], PALM [45]. Third, trie-based index structures [32, 36, 40] have $O(k)$ lookup, where k is the length of a key. This index structure type uses a digital key representation instead of hash or plain key comparison. Finally, there exist hybrid indexes that creatively combine the aforementioned index classes such as Masstree [38] (B+ tree with trie) and Wormhole [49] (hash table with B+ tree).

However, time complexity of operations is not the sole factor that determines the performance of an index. With the advent of multicore architectures, most modern databases use concurrent index structures that allow multiple threads to perform operations on the index concurrently and use synchronization mechanisms like locks, memory barriers, and

*The first author is currently at Amazon.

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397232

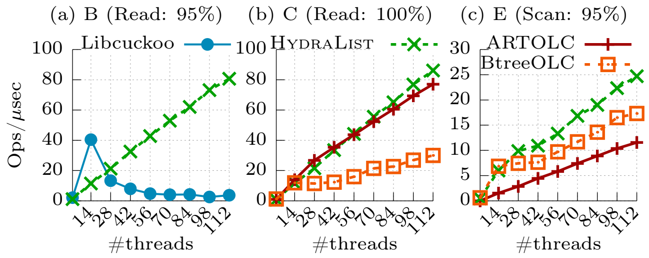

Figure 1: Performance of state-of-the-art indexes for the YCSB workload with 89 million string keys. HydraList consistently performs and scales well regardless of workload type. (a) Libcuckoo [35], an efficient cuckoo hashing implementation shows performance collapse for read mostly workloads because of spinlocks thrashing at high core counts. (b) Throughput of ART [33] with optimistic (ARTOLC) locking is 2.6× higher than B+ tree with the same locking mechanism. (c) B+ tree (BtreeOLC) shows higher scan performance than ART because finding next key in scan is efficient. Refer to Table 1 for workload characteristics.

atomic instructions to coordinate access to shared memory. The underlying synchronization mechanism can greatly affect the performance of index structures to the extent of causing performance collapse at high core counts. For example, Libcuckoo [35] is a known efficient and concurrent implementation of cuckoo-hashing; however, Figure 1(a) shows performance collapse at 14 cores. This is because the lock used to synchronize access to buckets becomes the bottleneck. Moreover, most high performance servers today are non-uniform memory access (NUMA) machines and have complex memory hierarchies. Such machines can have long memory stalls, making traditional assumptions like uniform memory access time inaccurate in data structure design and can reduce performance significantly [46]. *Thus to achieve high performance in modern servers, an index structure should have: 1) efficient time complexity of operations, 2) low synchronization overhead and 3) memory access patterns that are cache efficient and (mostly) NUMA-local.* Figure 1 shows how index structures perform well for some but not all workloads because they violate at least one of the the three aforementioned performance factors. We provide a full analysis in §6.

In this paper, we propose a new in-memory index structure, HydraList, which achieves high performance and scalability. The key idea behind HydraList is to separate and individually optimize an index structure into two components: a *search layer*, which helps locate key-value pairs efficiently, and a *data layer* which stores key-value pairs. The two layers are decoupled which allows asynchronous updates to the search layer reducing the synchronization overhead by making the critical sections in code smaller. Finally, to address memory stalls caused by cross-NUMA accesses, HydraList replicates the search layer across NUMA nodes. HydraList supports insert, update, search, and delete operations as well as range scan queries. All operations in HydraList are strictly serializable. HydraList supports 64-bit integer and string key types.

The contribution of this paper are as follows:

- We introduce, explain, and implement a new design paradigm that provides high performance for a variety of workloads; we do this by breaking down an index structure into two decoupled and individually optimized components: a search layer and data layer
- We show that inconsistency between the two layers can be tolerated and in fact, leveraged to both reduce synchronization and minimize remote memory accesses.
- Using this design approach, we propose HydraList, a new index structure which achieves high performance and scalability on massive multi-core systems.
- We compare HydraList to other state-of-the-art index structures using real world workloads. HydraList outperforms state-of-the-art index structures in insert and scan workloads while maintaining search performance comparable to state-of-the-art.

The remainder of this paper is organized as follows. Section §2 reviews related work. In §3 we present design goals and an overview of HydraList design. In §4 we describe the design of HydraList in detail. We evaluate HydraList in §6 and conclude in §8.

2. RELATED WORK

HydraList is inspired by many previous works on concurrent data structure design and multicore scalability. We have refined the most relevant ones to following principles:

Reduce Synchronization Overhead. The scalability of an index structure heavily depends on the underlying synchronization mechanism being used [6,15,16,19]. Traditional concurrent B-Tree uses lock coupling to reduce the number of locks held while traversing the tree [5]. In lock coupling, a reader, while traversing from root to leaf node, holds lock on a node until it has acquired lock on its child node. Once the lock on child node is acquired, the parent node lock is released and the process is repeated until the leaf node is reached. This approach reduces contention on a root of B-Tree but frequently acquiring and releasing locks does not scale well in multicore systems as it creates large cache coherence traffic [9]. Optimistic synchronization techniques, such as optimistic lock coupling [7,33] and OLFIT [9], have been proposed to reduce the synchronization overhead of frequent lock acquisition. In OLFIT, a version number is associated with every object which is incremented when writer updates the object. Readers (*i.e.*, lookup) check the version number before and after reading a node and retry if the versions do not match. Unlike a typical reader-writer lock which always modifies the lock variable at the start and end of critical section, in optimistic synchronization readers do not update lock variable which reduces cache invalidation traffic and improves scalability. HydraList uses optimistic synchronization to co-ordinate accesses of readers and writers in data layer.

Reduce the Size of Critical Section. Amdahl's law [23] predicts the maximum theoretical speedup when using multiple processors. If the maximum theoretical speedup, which is determined by a sequential portion of a program, is achieved, adding more cores will not yield higher performance. In this case, the only way to achieve higher performance is to reduce the size of critical section (*i.e.*, sequential portion). For example, if 95% of a program can be parallelized, it will reach 12× speedup with 32 cores. But if the serial section in the program is reduced to allow 99% parallelization, the speedup at 32 cores jumps to 24× with the maximum theoretical speedup of 90×. This means reducing the size of critical section is effective to improve performance and scalability. A general technique to achieve this is to delegate non-critical jobs like garbage collection to background threads [34] or to use specialized hardware like vector processing units [29,44,45,51] for faster execution in a critical section. In HydraList we reduce the size of serial section by updating only the data layer inside the critical section and if needed, use a background thread to update the search layer asynchronously. HydraList also uses SIMD instructions to accelerate search of keys in data layer.

Reduce Cache Misses. Previous work has identified that data cache misses are a significant component of database execution time and can be more than 50% of total execution time for certain workloads and configurations [2]. Therefore, many techniques have been proposed to reduce cache misses. Cache sensitive B+ trees stores all child nodes of a given node in contiguous memory [42] to reduce prefetcher-unfriendly pointer chasing and to make memory accesses hardware prefetcher friendly. Masstree uses many techniques like software prefetching, cache optimized fan-out to reduce cache misses [38]. Inspired by these approaches, HydraList uses a slotted doubly-linked list (§4.1) to store key-value pairs in data layer. This design significantly improves scan

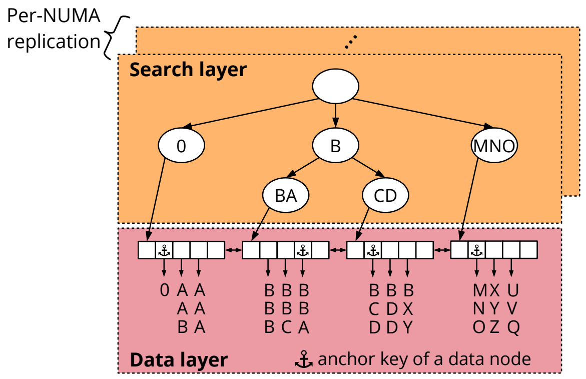

Figure 2: HydraList uses Adaptive Radix Tree (ART) [33] as *search layer* and a slotted doubly linked list as *data layer*. To reduce remote memory accesses, the search layer is replicated per-NUMA node. Nodes in data layer (called a *data node*) are indexed in the search layer using a representative key called *anchor key*. Keys in data node are not stored in sorted order.

performance of HydraList as fewer cache references are required to fetch the next key in the range and memory access pattern is prefetcher friendly. Also, HydraList stores key hash (called fingerprint) instead of keeping keys in sorted order. It reduces cache misses when searching a key because space-efficient fingerprint requires fewer cache references.

Reduce Cross-NUMA Memory Access. To accommodate many cores in a single machine, computer architects have adopted Non-Uniform Memory Architecture (NUMA) wherein cores are clustered into groups called NUMA nodes or simply node and each node share last level cache (LLC) and memory. NUMA architecture allows scaling to large number of cores but a side effect of this design is that cross-NUMA memory/cache-line accesses are more expensive than within a node. Thus, concurrent data structures should carefully handle such NUMA-ness to scale performance in a large multi-core architecture. Our experiments have shown that Wormhole performs worse than ART for read-only workload because of high cross-NUMA traffic, even though Wormhole uses fewer comparisons in case of lookup. A common technique to reduce cross node communication is to replicate shared memory across all NUMA nodes [8, 14]. The challenge with replication is to maintain consistency across all replicas while ensuring minimal synchronization overhead. HydraList solves this using partial replication wherein search layer is replicated among NUMA nodes while data layer is shared across all nodes. This technique reduces cross-NUMA traffic and the cost of replication in terms of memory is not as high as NR algorithm [8].

3. OVERVIEW OF HYDRALIST

3.1 Design Goals

HydraList has three main design goals to be a generic in-memory index for modern multicore systems:

- **Multicore Scalability:** Performance of HydraList should scale with increasing core counts. This is important as the number of cores in servers is rising, making the scalability aspect of index structures critical for database performance.
- **Data Scalability:** HydraList should efficiently index a large number of keys as increasing memory size allows more data to be stored in a database.
- **Versatility:** HydraList should be performant for a wide range of workloads. This eliminates the need for workload specific performance tuning.

No existing index structures achieve all aforementioned design goals. Hash tables have fast lookup but do not support scan operations making them unusable for a wide variety of applications. Skiplists and B-Tree based indexes do not scale well with increasing data sizes as their performance depends on key count. Trie based indexes on the other hand work well with large data sets but perform poorly on scan heavy workloads. Finally, our evaluation shows that most indexes do not perform well with increasing number of cores. We describe how HydraList is able to achieve these design goals in the following.

3.2 Design Overview

To understand how HydraList is able to achieve its design goals, one needs to understand the general design of index structures. Most tree-based index structures can be divided into two layers [1]: a search layer and a data layer. The data layer stores key-value pairs while the search layer stores partial keys (as in trie) or a subset of keys (as in B+ tree) which allows the reader to locate the key in the data layer efficiently. Keys in data layer can be chained (as in B^{link}-Tree [30]) allowing a faster scan or stored independently (as in trie). In these indexes, updates to data layer are synchronously propagated to the search layer if needed. This increases critical section size inhibiting scalability. For example, an insert operation at a leaf node of a B+ tree (update to data layer) can cause a split which in turn can cause a split in the internal nodes of the tree (update to search layer). The insert operation is completed only when the search layer is consistent with the data layer. Therefore to design a high performance index structure, we need to design an efficient search layer, data layer, and a synchronization mechanism that updates both layers efficiently. In the rest of this section, we will explain the design overview of HydraList and motivation behind our design choices.

Search Layer. With increasing memory sizes in modern servers, it is expected that the number of keys stored in a database will also increase. For example, a study of Facebook's KV cache workload reveals that the size of most keys is between 20 and 40 bytes [3]. This means 32 GB of key data can contain from 800 million to 1.6 billion keys which makes indexes whose lookup cost is proportional to `log(number of keys stored)` a poor choice. For example, B+ tree with 100 million keys requires at least 26 comparisons to find a key[2]. On the other hand, the lookup cost of trie-based indexes is proportional to the length of the key. This property makes it possible for tries to perform faster lookups than comparison-based indexes, such as B+ tree, especially for data sets with a large number of keys. However, one downside of trie-based indexes is that long keys can make searches slower. This problem is alleviated by path compression wherein nodes with single child are merged together effectively reducing the lookup cost. Adaptive Radix Tree (ART) [33] is a radix tree based index which supports

[1]Not all tree-based index structures can be decomposed into these two layers. For example a binary search tree stores value in every node.

[2]If the order of B+ tree is S, then the height of tree is $log(100M)/log(S)$ and at each level a binary search is performed with order $log(S)$.

resizing of internal nodes and path compression making it highly space efficient and performant. Therefore, we extend ART as the search layer in HydraList.

Data Layer. The scan performance of B+ trees is good because their design of leaf nodes allows clustering of keys within a range so less pointer chasing is required to process range scans, which otherwise may incur costly cache and TLB misses. Moreover, chaining of adjacent leaf nodes in B+ tree leads to faster discovery of the next leaf node in the range. Tries including ART suffer from poor scan performance because poor clustering properties and lack of chaining of leaf nodes. This forces readers to jump between multiple levels to perform range scans [48]. To alleviate this problem, we use a slotted doubly linked list, called *data list* as the data layer in HydraList (see Figure 2). Every node in the data list (called *data node*) stores multiple key-value pairs and is indexed in the search layer using a unique key (called *anchor key*).

Asynchronous Update. In HydraList, the search and data layers are decoupled, *i.e.*, updates from the data layer to the search layer are not propagated in a synchronous fashion; rather, updates to the search layer are done using background threads. We use operational logging [4], wherein updates to the search layer are enqueued in a per-thread queue. Periodically, all operational logs are merged and the search layer is updated. The search algorithm of HydraList is designed to tolerate transient inconsistency when the data layer is updated but updates to the search layer are pending.

Synchronization. Since the two layers are decoupled, HydraList uses two different synchronization mechanisms. These mechanisms are chosen based on properties of the layers. Because the search layer is only updated by a single background thread and progress of readers is important, we use the Read-Optimized Write Exclusive protocol [33] to synchronize ART in the search layer allowing non-blocking reads. To allow parallelism and lock free traversal of the data layer, we use optimistic version locking to synchronize data nodes. Synchronization is further described in (§4.6).

Replication of Search Layer. To reduce remote memory accesses, we replicate the search layer across NUMA nodes. This is possible while allowing strictly serializable reads because the two layers are decoupled and the search algorithm in HydraList can tolerate inconsistency between them.

4. DESIGN OF HYDRALIST

In this section, we will discuss the design of HydraList. First we will discuss organization of the search and data layers as well as describe the operations of HydraList.

4.1 Data Layer and Search Layer

Figure 4 illustrates the layout of a data node. Every data node is assigned a unique key called the *anchor key* which is the smallest key of the node when it was inserted in the data list. For example, in Figure 2 the anchor key for the second data node is `BBA`. An invariant in the data layer is: *any key stored in a data node should be greater or equal to the anchor key of the current node and should be smaller than the anchor key of the next data node (i.e.,* `node->anchor_key <= key < node->next->anchor_key`*).*

This invariant assigns a key range to every node and thus maps every key to a unique node in the data list. Key-value pairs are stored in contiguous memory locations in a data node but not in sorted fashion as maintaining a sorted order for keys, especially string keys, is expensive. Instead, HydraList stores a 1-byte hash (called *fingerprint*) [41] of every key and maintains a bitmap of every valid key-value slot. A *permutation array* stores the position of keys if the keys were sorted and is maintained to reduce the cost of sorting the key-value array on every scan operation. Data nodes store pointers to next and previous data nodes forming a doubly linked list. Since data nodes store keys in slots, we call this data structure a *slotted doubly linked list.*

The ART-based search layer of HydraList indexes all nodes in a data list by storing anchor keys and pointers of data nodes. A newly created data node might not be immediately indexed as updates to the search layer are propagated asynchronously. However, this does not lead to incorrect results because the search operation in HydraList can tolerate such inconsistencies which is described next.

4.2 Search Operation

This operation begins with traversing the search layer to locate a *jump node*, that specifies where searching in data layer should begin. Second, the data layer is traversed to find the data node (called *target node*) which might contain the key. Finally, the target node is searched for the queried key. Note that in the search layer, a key is divided into 1-byte tokens and path compression is used for cache efficiency. For example in Figure 3, key `BBB` is represented using two nodes, `B` and `BB`. The leaf node of the search layer stores the pointer to the data node of the corresponding anchor key.

4.2.1 Finding a Jump Node in Search Layer

Ideally, the jump node is the target node. However, when they are not the same, which is possible when search layer has pending updates, the jump node should be as close as possible to the target node. By design, anchor keys divide the key space into ranges. Therefore, finding the jump node is equivalent to finding the anchor key in search layer which is a lower or upper bound of the query key (`key`). This can be done in our ART-based search layer using the algorithm described. The pseudocode can be found in appendix (§10).

Consider a query key $<t_1 t_2 ... t_l>$. First a reader will perform a longest prefix matching (LPM) between the query key and the keys in the search layer. If the query matches an anchor key, the search is over. Otherwise the reader has to find the next smaller or larger anchor key. Let p be the length of the longest prefix. Among the children of the last matched node, child node with token L is found such that $LowerBound(t_{p+1}) = L$ Then the reader will walk to the right-most leaf node of the sub-tree rooted at this child , see Figure 3 ❶ - ❷). If a lower bound token does not exist, then the child node with token H is found such that $UpperBound(t_{p+1}) = H$; but, instead of walking to the right most leaf node, the reader searches for the left most leaf node of the sub-tree.

For example, consider the search layer in Figure 3. If the query key is `BAA`, the LPM search will yield node with token `B`. Since no child of this node exists with token smaller than `AA`, node with token `BB` is used. Since it is a leaf node we immediately get the jump node.

4.2.2 Locating a Target Node in Data List

A target node can be found by traversing the data list till a node is found that meets the invariant in §4.1. If a query

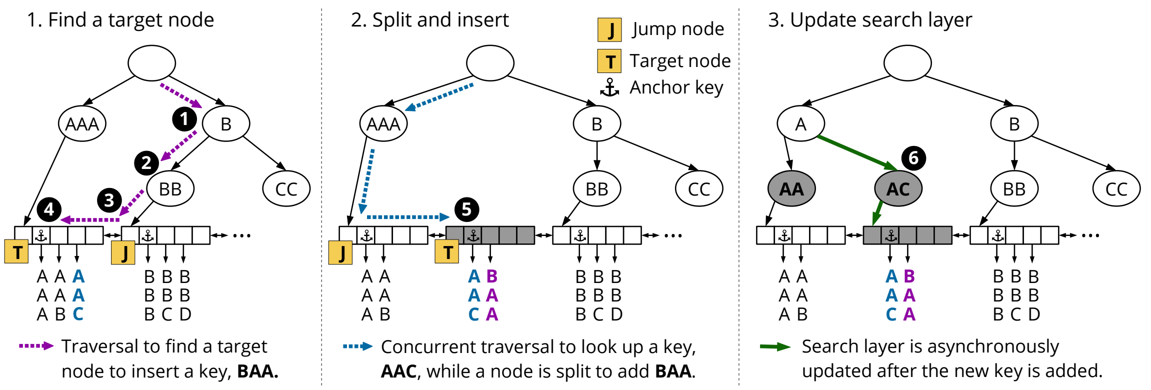
1. Find a target node
2. Split and insert
3. Update search layer
J Jump node
T Target node
⚓ Anchor key
AAA
B
BB
CC
A
AA
AC
A A A
A A A
A B C
B B B
B B B
B C D
A A
A A
A B
A B
A A
C A
Traversal to find a target node to insert a key, BAA.
Concurrent traversal to look up a key, AAC, while a node is split to add BAA.
Search layer is asynchronously updated after the new key is added.

Figure 3: An illustrative example of inserting a key `BAA` in HydraList. 1) First the writer has to find the node where this key should be inserted (*i.e.*, target node). For this, it will perform a longest prefix match search ❶, then, using the smaller child of the node it will perform a walk to the leaf node ❷ and jump to a data node (called a jump node) pointed by the leaf node ❸. Then it will traverse backwards to reach the target node ❹. 2) Since the target node is full, it is split and the key is added to a new node ❺. Note that lookup for a key in a new node will be successful even though it does not have a reference in the search layer. Lookup for `BAA` will follow the path shown in blue until the search layer is updated. 3) Finally the search layer is updated asynchronously and the new node has a direct reference from the search layer (node ❻).

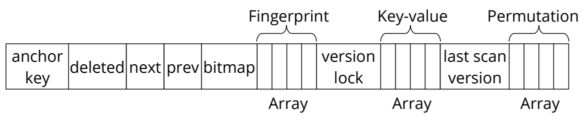

Figure 4: Layout of a data node

key is smaller than the anchor key of current node, then the search proceeds leftward. See step ❸ of Figure 3 where query key `BAA` is smaller than the anchor key `BBB`. If the invariant is not met and a query key is greater than the anchor key, then the search proceeds rightwards. See step ❺ of Figure 3, where a query key `AAC` is greater than the anchor key `AAA`. This traversal is done lock free as the reader only needs to read the anchor key of a node which is never modified once a node has been created.

4.2.3 *Searching for a Key in the Target Node*

On finding the target node, a 1-byte fingerprint of the query key is generated. Only key-value slots that are valid in the bitmap and whose fingerprint matches with the query key fingerprint are probed. Using 1-byte fingerprints, we found that 1.077 probes on average were required to find the correct key which implies collisions are rare. To further improve the performance, we use vector instructions to match multiple fingerprints in the node while using a single instruction.

4.3 Split and Merge of a Data Node

In HydraList, a data node can store 64 key-value pairs. We choose 64 key-value pairs per node as it allows efficient comparison of key fingerprints using SIMD instructions (AVX512). Variable length values can be supported by storing pointers to value fields instead of actual values in the key-value array. Insert operations on a full node causes a split. To split a data node, the writer will first lock the node, find the median key in the key-value array and move keys which are greater or equal to the median key to a new data node. The median key is assigned as the anchor key of the new node. The new node is then inserted into the linked list. Two adjacent nodes are merged when a delete operation causes the total number of keys in the two nodes to be less than half of full key-array capacity. Figure 3 (2), illustrates the split process. The target node (first data node in data layer) of query key `BAA` is full, so the inserting thread creates a new node, finds the median key of the target node and moves all keys greater or equal to the median into the new node. Also it inserts the query key into the new node. It should be noted that only one node is locked during the split process because the invariant in §4.1 ensures the target node is eventually found even when adjacent nodes are concurrently updated. Also, two writers cannot modify the previous pointer field of a successor node because splits are only allowed to create successor nodes and nodes are always merged to the predecessor. The HydraList merge algorithm is similar to split.

4.4 Decoupling Search Layer and Data Layer

The search layer has to be updated whenever an insert or a delete in the data layer causes split/merge of data nodes. These structural modification operations (SMO) are traditionally done in a synchronous fashion *i.e.*, splits/merges are not made visible to other readers/writers until the modification has propagated to the search layer. Performing synchronous updates requires holding locks for multiple nodes, which becomes a performance and scalability bottleneck especially for write-heavy workloads. In HydraList, the search algorithm involves finding an anchor key in the search layer which is closest to the query key (not the exact key), hence a reader can tolerate lazy updates to the search layer as long as the data layer is consistent. This insight allows decoupling of the two layers. Updates to the data layer is done synchronously while updates to the search layer is asynchronous. If the query key is located in a node whose anchor key has not been added to the search layer, then the reader will land at a node further away from the target node. Hence the cost of inconsistency between two layers is longer traversal in the data layer. However, this cost is smaller than the cost of updating both layers synchronously (see §6.3.1).

4.4.1 *Asynchronous Update of Search Layer*

Figure 5 illustrates asynchronous update in HydraList which uses a per-thread operational log and two types of background threads. To propagate updates from the data layer to the search layer, threads that cause a split or merge store information including the anchor key, a pointer to new

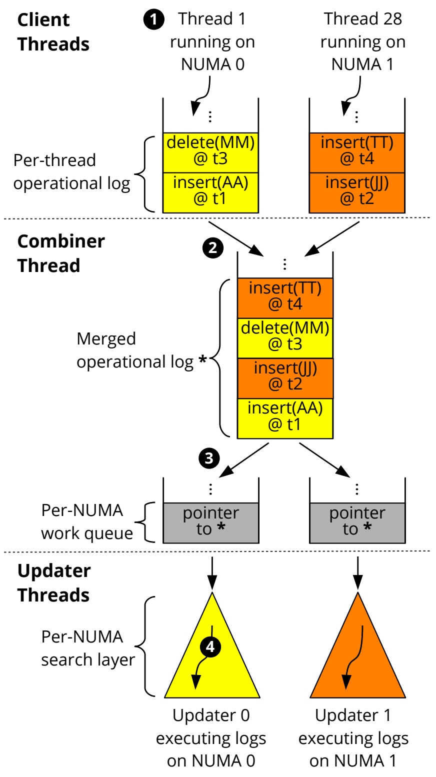

Figure 5: Illustration of how a combiner and updater threads are used to update the search layer asynchronously. ❶ Client threads enqueue operations into their operational log. ❷ A combiner thread periodically reads all operational logs and returns a merged log which has operations sorted according to timestamps. ❸ A merged operational log is broadcast to all updater threads which update the search layer. ❹ A per-NUMA updater thread applies operations to its local search layer.

data node, a current timestamp, and the operation (*i.e.*, insertion or deletion of an anchor key), in its operational log. For example, in Figure 3, after creating the new data node, the thread will enqueue an insert operation into its operational log along with other metadata. Per-thread logging is used instead of a global log as it reduces cross-NUMA traffic and synchronization between enqueuing threads. Timestamping ensures global ordering of events. A special background thread, called a *combiner*, periodically merges operation logs of all threads, sorts them according to timestamp, and adds the pointer of the merged log into the work queue of a second background thread, called *updater*, which updates the search layer asynchronously. Sorting operations according to timestamp is important for consistency of the search layer as operations should be applied in the order in which they were created. In our prototype, we set the combining interval to 200 microseconds. An updater thread dequeues operations from its local work queue and executes the operations against the search layer. Using two different types of background threads allows task level parallelism. It should be noted that only a single thread updates a search layer but HydraList can have multiple search layers (hence multiple updaters) to achieve NUMA-awareness (discussed in §4.5). To generate timestamps, we use a previous work of generating scalable timestamp using hardware counters called ORDO [27]. We cannot directly use `RDTSC` instruction as hardware counters in different sockets have a constant skew between them. ORDO takes the constant skew into account and provides hardware timestamps which can be ordered correctly. Note that no special hardware is required to use the ORDO primitive. Updating search layer asynchronously may form long chains of data nodes in the data layer which do not have anchor keys in the search layer. This could lead to unbounded processing times for reads. However, we did not find this issue in our evaluation. We measured the distance between jump and target nodes for a 100% insert workload and found the distance to be less than 2 node away for 99.5% of inserts. We discuss the detailed results in §6.3.4.

4.4.2 *Reclaiming Merged Operational Logs*

Once a merged log is created, its pointer is broadcast to all updater threads. A merged log is safe to be deleted when all operations in the merged log have been applied to the search layer by the updater threads. These merged logs are named *obsolete logs*. To detect an obsolete log, every merged log is given an monotonically increasing ID by the combiner thread at time of creation. Every updater thread keeps track of the merged logs which it has consumed. The combiner thread will periodically check with all updater threads to free merged logs which have ID smaller than the smallest obsolete log ID among the updater threads.

4.4.3 *Reclaiming a Data Node from Search Layer*

After a merge operation, the data node gets logically deleted from the data layer but the memory cannot be reclaimed as references to that data node still exist from the search layer. To safely reclaim the data node, we use a variation of Epoch Based Memory Reclamation (EBMR) techniques [21,22,39]. Here an epoch is defined as the period between two consecutive cycles of freeing obsolete merged logs. Freeing obsolete merge logs ensures that any deleted data node in the previous epoch has no memory reference from the search layer as structural modifications related to split/merge operation have been propagated to search layer. However, it is possible that a reader can started reading in the previous epoch and still reference the deleted data node. To ensure no reader is currently reading the deleted data node, we wait for all threads to exit the critical section at least once (*i.e.*, passing the second epoch). Then we physically free the memory of all nodes which were deleted in the previous epoch.

4.5 NUMA-Awareness to Search Layer

Since inconsistency between the search and data layers can be tolerated, performance and scalability of HydraList can be further improved by replicating the search layer in every NUMA node. By doing so, every thread will execute only NUMA-local accesses when traversing the search layer. NUMA replication of the search layer is achieved by assigning an updater thread to every NUMA node and making the combiner thread broadcast the merged operational log to work queue of each updater. Therefore in a machine with N NUMA nodes, $N + 1$ background threads are needed, N for replication and 1 thread as combiner.

4.6 Concurrency

The search and data layers have different properties in terms of contention, memory locality, and update frequency. Decoupling the search and data layer allows concurrency control of both layers to be designed independently and optimized for layer specific properties.

4.6.1 *Concurrency in Search Layer*

The search layer is updated by a single thread meaning only reader-writer synchronization is required, and writer-writer synchronization is not essential. Also readers can be prioritized over writers to allow non blocking reads as the search layer can tolerate delayed updates. Keeping these properties in mind, we use Read-Optimized Write Exclusion (ROWEX) [33] protocol to synchronize the search layer in HydraList. In ROWEX, a writer locks a node before modifying it, thus providing exclusion relative to other writers but readers do not acquire any locks. ROWEX ensures reads are safe by only allowing atomic modifications to fields that can be read concurrently. Nodes are replaced or added to ART using atomic compare-and-swap operations. Obsolete nodes are freed using garbage collection.

4.6.2 *Concurrency in Data Layer*

The data Layer in HydraList is shared between all readers and writers. Therefore, the concurrency control of the data layer should ensure maximum parallelism and reduce cache coherence traffic. To achieve this, HydraList uses the Optimistic Version Locking protocol. When a writer wants to modify a node, it first atomically increments the version number of the node. Since version numbers are initialized to zero, an odd version number indicates the node is being updated and any concurrent reader or writer trying to enter the critical section will have to retry. After the node is updated, its version number is incremented again and is unlocked. A reader reads the version number of a node before and after accessing the node. If the version number does not match, then the reader retries. The advantage of this protocol is that it does not cause cache invalidation in case of reads unlike traditional locking schemes like spinlock or reader-writer lock which modify shared cache lines when readers enter critical sections. A drawback of optimistic concurrency is the large number of aborts under high contention which can cause performance collapse. To prevent this, HydraList implements a backoff scheme wherein if a reader or writer performs retries more than a certain threshold, it waits for random period before attempting again. Our evaluation shows that this backoff scheme does not affect performance in low contention case but is effective in preventing performance collapse under high contention (§6.3.1).

4.7 Putting It All Together

Before starting an operation, a thread determines its NUMA node and gets the root of the corresponding NUMA node's search layer. A side effect of using optimistic locking techniques is that readers can read logically deleted data nodes. To avoid this, after entering a critical section, every reader/writer ensures that the current data node is not deleted by checking the deleted field (Figure 4) and ensuring the invariant of the data list defined in §4.1 is not violated.

4.7.1 *lookup(key)*

For lookup, a reader first reads the version number of the node using `readLock()` API which returns 0 if the node is locked making the reader retry. Also, a reader will check the validity of the node. If the node is already deleted, it will retry the lookup operation. If the node is valid, then the reader will find the index of the key in the key-value array using the fingerprint. If the key is found, the value of the key is fetched. The reader checks if the node version is unmodified before returning the value. An updated node version indicates a concurrent write and the reader will rety the lookup.

4.7.2 *insert(key, value)*

The insert operation starts by finding a target node. After finding the target for `key`, the writer tries to lock the data node by incrementing the node version number using atomically. In case of failure, the writer aborts and retries. After locking the data node, it checks again if the target node invariant is met and the node is not deleted, due to a concurrent split or merge. If the node is still valid, the writer checks if the key already exists in the node. To do this, it generates a fingerprint (*i.e.*, 1-byte hash) of the key and checks only those key array slots whose fingerprint matches. Fingerprint matching is accelerated using vector instructions. If such a key does not exist, key-value is inserted in the first empty slot. Finally, the node is unlocked by incrementing the node version again.

4.7.3 *remove(key)*

The remove operation is similar to `insert(k,v)`. After locking the data node and checking its validity, the node tries to find the key in the node. If successful, the key is deleted by resetting its corresponding bit in the bitmap.

4.7.4 *scan(key, range)*

Keys in a data node are not stored in a sorted fashion. This makes sorting keys on every scan operation a performance bottleneck. To reduce the cost of sorting, we use a permutation array which stores the indices of keys in a sorted fashion. The permutation array is computed again only if there has been an update to the node since the last time it was computed. This can be detected by comparing the version number of the node when permutation array was last generated and the most recent version number. If an anchor key of a node is less than the start key, then the starting index of a scan is computed using binary search. Then using the permutation array, the result array is populated. This process is repeated with the next node until the size of the result array is the same as the scan range length.

5. IMPLEMENTATION

We implemented a prototype of HydraList in C++. The prototype comprised of 4500 lines of code (LoC) including 2600 LoC from the concurrent Adaptive Radix Tree library [1]. We added 300 LoC to the library to implement the algorithm to find the jump nodes. We use Intel Advanced Vector Extensions (AVX 512) [25] for comparing key fingerprints to fingerprint arrays. To only require one SIMD instruction for comparison, we allocate 64 keys per node. Also, we accelerate bitmap array operations using x86 assembly instructions. We use a hardware clock (`RDTSC` in x86 architecture) to prevent timestamp allocation from becoming a scalability bottleneck [47, 50]. Our implementation of HydraList supports 64-bit integer and string key types. The maximum size of string keys is limited to 32 bytes. We have discussed techniques to support a variable length key in §7.

6. EVALUATION

In this section, we first begin by introducing our evaluation setup (§6.1). We then show performance results of

HydraList in comparison with other state-of-the-art indexes (§6.2). Finally, we analyze the effectiveness of our design choices (§6.3).

6.1 Experimental Setup

6.1.1 Hardware and Software Platform

We used a Intel Xeon Platinum 8180 server with 112 physical cores. It has four NUMA sockets with 28 physical cores per socket. It has 125 GB of memory. We disabled AutoNUMA [12], which automatically migrates memory from one NUMA domain to another NUMA domain, to avoid performance interference by underlying OS. Also, we enabled huge pages as some index evaluated use huge pages. We used `jemalloc` as memory allocator to prevent memory allocation from becoming a scalability bottleneck. We used gcc 8.3 with `-O3` compiler flag to compile all indexes and benchmarks[3]. All experiments are done on Linux Kernel 5.0.16.

6.1.2 Real-World Workload: YCSB

We used the Yahoo! Cloud Serving Benchmark (YCSB) [11], which is a widely-used key-value store benchmark as it mimics real-world workloads as summarized in Table 1. We used an index benchmarking tool, index-microbench [48], which generates a workload file for YCSB and statically split them across threads. For each workload, we test two key types: integer and string. For integer key, we randomly generate 100 million 8-byte random integers. For string key, we use publicly available 89 million email addresses [20] to mimic the key distribution of real workload. The average length of email addresses is 20 bytes and maximum is limited to 32 bytes. For evaluation, the username and domain name of email addresses are swapped–*e.g.*, `abc@xyz.com` becomes `xyz.com@abc`–which is a common pre-processing for trie-based indexes [32, 38, 48]. The value field in each key type is a 8-byte integer which mimics pointer to record in a real database. For each workload, we first load keys (100 million for integer and 89 million for string) and then run 50 million YCSB transactions on the keys.

6.1.3 In-Memory Index Comparison

We compared HydraList with several state-of-the-art indexes: Adaptive Radix Tree (ART) [33] with optimistic lock coupling (ARTOLC) and read-optimized-write-exclusive synchronization (ARTROWEX), Masstree [38], B+ tree with optimistic lock coupling (BtreeOLC) [10, 31], open source implementation of Bw-Tree (Bw-Tree) [48], Wormhole [49], and Cuckoo hashing (Libcuckoo) [35]. Obviously, Cuckoo hashing does not support scan operation so we did not present Cuckoo hashing results for YCSB workload E, which requires scan operations.

We did not include the evaluation results of skiplists, including a lock-free version [13] and a NUMA-optimized version [14] because they did not perform and scale well with a large number of keys. We found that these skiplist algorithms do not scale because the background thread used to update the skiplist becomes a bottleneck when number of keys increase. This is consistent with findings of previous work [48]. Increasing the number of background threads to improve scalability while maintaining correctness is non-trivial.

[3] For YCSB workload A and workload B, we did not include ARTOLC as it segfaults with -O3 optimization

Table 1: Characteristics of YCSB workloads

Workload	Application	Description
Load	Bulk database insert	100% Insert
A	Session store	Read/Update 50/50
B	Photo tagging	Read/Update 95/5
C	User profile cache	100% Reads
D	User status update	Read (Latest)/Insert 95/5
E	Threaded conversation	Scan/Insert 95/5

6.2 Performance Evaluation

Figure 6 and Figure 7 show the performance and scalability of in-memory index structures with YCSB workload.

6.2.1 Insert Only

Libcuckoo has the highest throughput for both integer and string keys because it randomly scatters the key which lowers contention. HydraList throughput for integer key is 38.5% higher than ARTOLC but is similar to ARTROWEX for string keys. This is because performance improvement dues to reduction in size of critical section in case of integer keys is negated by higher key comparison cost for string keys. Masstree, Wormhole and B+ tree showed performance saturation after 28 threads because reads and writes cross NUMA boundaries.

6.2.2 Workload A

This workload performs large number of reads and updates with skewed access pattern which causes performance saturation or meltdown of most index structures at high core count as updates are serialized. HydraList avoids performance collapse unlike Libcuckoo and ARTOLC as it reduces contention using a backoff scheme. Thus HydraList throughput is 6.4× and 1.75 higher for integer and string keys respectively at 112 threads. High performance shows that HydraList can handle workload skew and does not require load balancing.

6.2.3 Workload B

All indexes show near linear performance scaling as this is a read-mostly workload. For string keys, ARTROWEX, Masstree and HydraList perform best. Masstree and ART performs well because common prefix of string in our workload reduces the number of comparisons. B+ tree and Bw-Tree show lower performance because of large number of keys and high string comparison cost.

6.2.4 Workload C

Libcuckoo shows performance collapse because per-bucket spinlock becomes bottleneck. In case of 112 thread for integer keys, performance profiling of Libcuckoo using Linux `perf` shows 98% of cycles are spent in acquiring spinlock. B+ tree and Bw-Tree show lower performance because of large number of keys which requires more memory access and comparisons. Performance of Wormhole is lower, even though its theoretical asymptotic complexity is better (*i.e.*, $O(log(length\ of\ key))$ because it uses a reader-writer lock to synchronize access to leaf nodes. Reader-writer lock take 15% of CPU cycles for 112 threads and hence becomes a scalability bottleneck. HydraList throughput is about 20% lower than ART in case of integer keys but the performance is comparable in case of string keys.

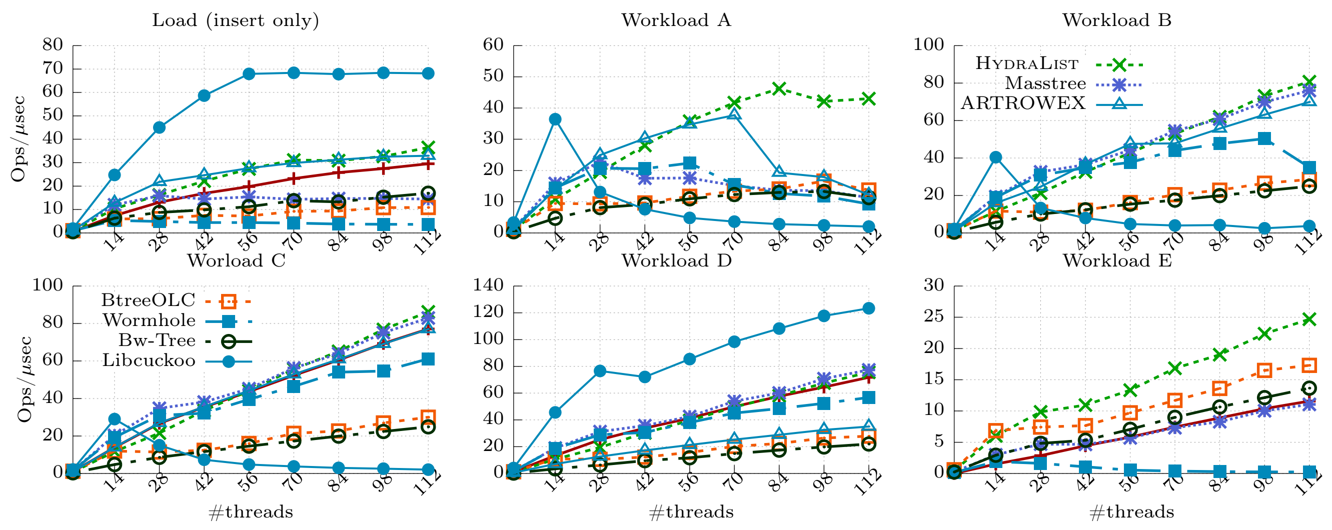

Figure 6: Performance comparison of in-memory indexes for YCSB workload: 50 million operations with 89 million string keys.

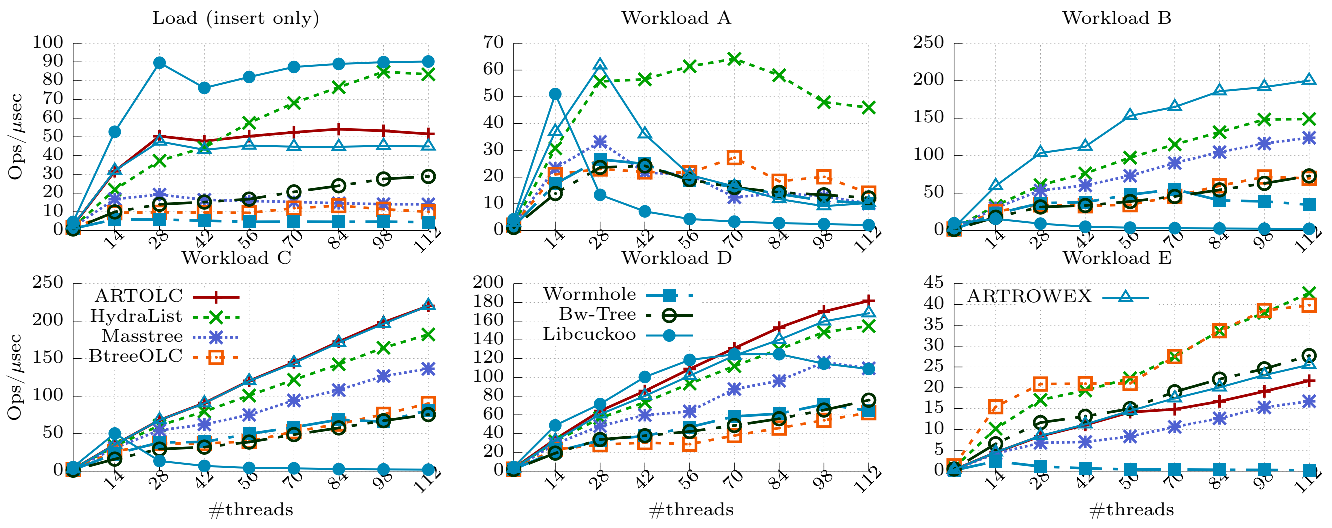

Figure 7: Performance comparison of in-memory indexes for YCSB workload: 50 million operations with 100 million integer keys.

6.2.5 *Workload D*

All indexes show near linear performance scaling as this is read-mostly workload. Performance of ARTOLC and HydraList are comparable for both integer keys and string keys. In this workload, performance of Libcuckoo does not collapse because lower contention on buckets.

6.2.6 *Workload E*

HydraList and B+ tree outperform all index structures in scan workload because slotted nodes structure of the data layer allows readers to read values with next key in the range with fewer cache references as compared to ART or Masstree which have to traverse different levels. Wormhole performs poorly because scanning every node requires sorting of key array. Also Wormhole uses reader-writer lock which generates large cross-NUMA traffic leading to poor performance.

6.2.7 *Delete Workload*

Since YCSB does not support delete, we benchmarked indexes using a synthetic workload comprising of deletes (15%), inserts (15%) and lookup(70%). The results for 89 million string keys and integer keys are summarized in Figure 8.

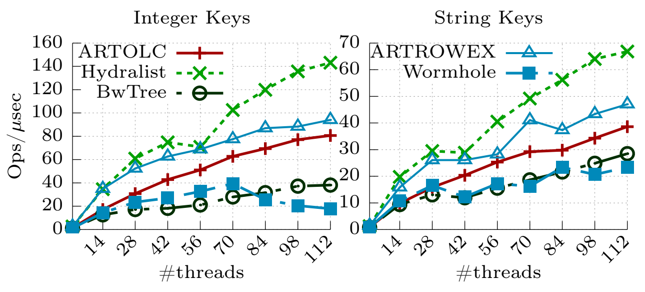

Figure 8: Performance comparison of in-memory indexes with mixed workload of insert(15%), delete(15%) and lookup (70%).

The throughput of HydraList is 1.5× and 1.4× higher than ARTROWEX for integer and string keys respectively for 112 threads.

6.2.8 *Summary*

It should be noted that with exception of few workloads, there is a superior structure or very close competitor to HydraList in terms of performance but these competitors change with different workloads. HydraList consistently

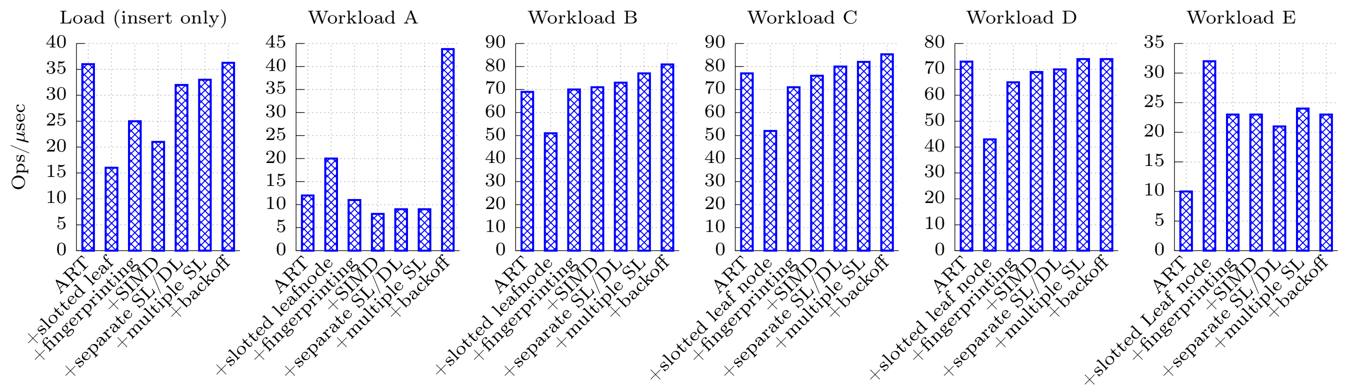

Figure 9: Factor analysis of YCSB Workload for 112 threads with string keys.

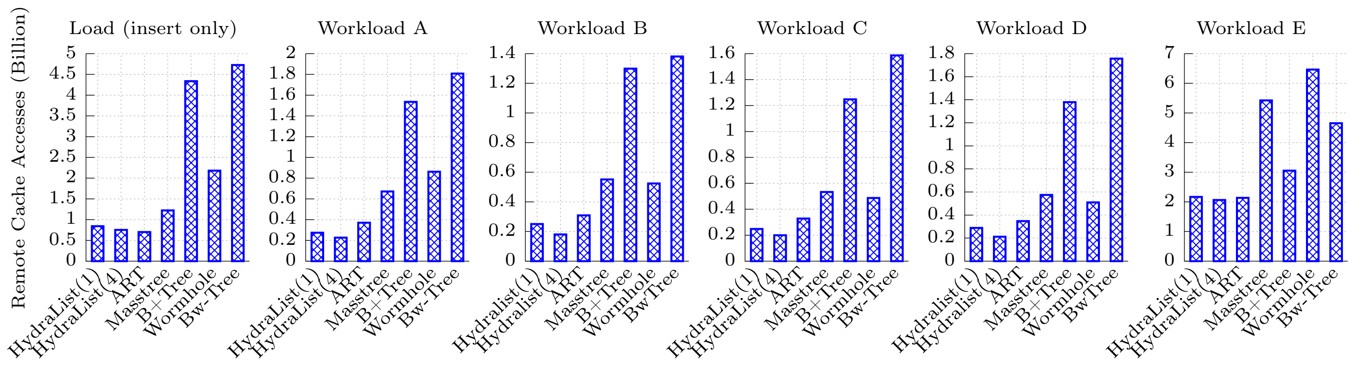

Figure 10: Comparison of remote memory access. HydraList (n), where n is the number of search layer.

performs well for all different workloads and key type, hence achieving its design goal of versatility (§3.1).

6.3 Analysis on Design Choices

In this section, we analyze impact of our design choices on performance and scalability (§6.3.1), impact of search layer replication on NUMA traffic and memory (§6.3.2), HydraList scalability with increasing data size (§6.3.3) and distance between jump node and data node (§6.3.4).

6.3.1 Factor Analysis

We analyze the performance of HydraList by breaking down the performance gap between ART and HydraList. We incrementally added features to ART and benchmarked each increment using YCSB workloads. The throughput of 112 threads for string keys are summarized in Figure 9.

+slotted leaf node. We added multiple key-value slots to the leaf node of ART which improves scan performance by over 3.2×. Keys are stored in a sorted fashion. Binary search is used to search within a node. However, performance in insert-only workloads reduces by 2.4× as inserting into a sorted list requires multiple string comparisons and key shifts. This feature also reduces the performance of workload C and D by 33% and 41% respectively as binary search is not cache efficient.

+fingerprinting. To counteract this deficiency, we use 1-byte fingerprints to search for keys in a node. This improves throughput of insert-only, B, C, and D workloads by reducing amount of string comparisons. This however, reduces performance of scan operations as it adds overhead of array sorting before scanning a node.

+SIMD. Using SIMD instructions allows fingerprint comparison to occur in parallel; this improves performance in workload B, C, and D which are lookup heavy workloads.

+separate SL/DL. Separating search and data layer as well as asynchronous search layer updating improves performance in insert-only case by 52% with no/negligible impact on performance of other workloads. Improvement in performance can be attributed to the removal of updates from the critical path in the search layer.

+multiple SL. Adding a search layer to each NUMA node improves performance further by about 9% in case of insert-only work load. It also improves performance in Workload B, C, and D by 5%, 2.5%, and 5.75% respectively.

+backoff. A thread is forced to retry when it fails to acquire a write lock on a data node. This causes high contention and can lead to performance collapse. We added backoff feature wherein if a thread has retried a certain number of times it will wait before attempting again. Adding backoff improved performance by 4.5× in Workload A which is a high contention workload.

6.3.2 Impact of Search Layer Replication

Cross NUMA Traffic. To measure the remote memory access, we measured the total number of remote cache lines accessed using Intel's Performance Counter Monitor for 50 million transactions of YCSB workload with string keys. As Figure 10 shows, HydraList has the least number of remote memory accesses among the measured indexes for all the workloads. Remote cache line accesses reduce when HydraList uses four search layers because search layer traversals does not access remote memory. B+ tree and BwTree have large number of remote cache line accesses for Workload A and C as their search operation involves accessing more memory locations. However, in case of Workload E, their remote cache line accesses is lower than Masstree because of slotted leaf node structure which leads to smaller

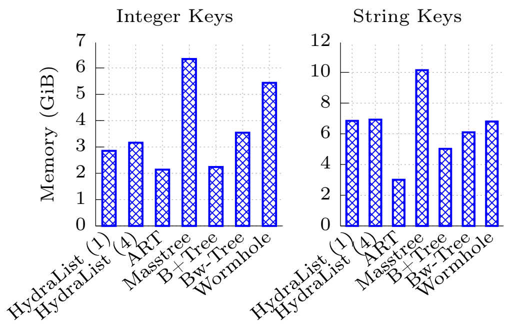

Figure 11: Comparison of memory consumption to store 100 million integer keys or 89 million string keys.

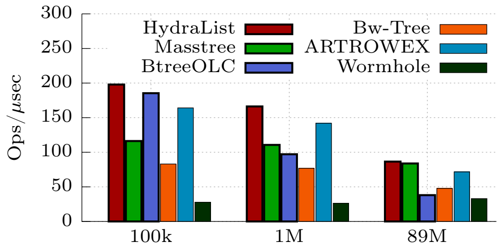

Figure 12: Performance comparison of indexes with varying key set size with 112-thread and string keys.

number of cache misses. NUMA traffic in Workload E is high in case of Wormhole because of reader-writer locking.

Memory Consumption. We measured the peak memory consumption of different indexes after inserting 100 million integer keys and 89 million string keys (see Figure 11). We measured the resident set size (RSS) by reading `/proc/[pid]/statm` in Linux. For this experiment, huge pages were disabled to reduce overhead of fragmentation. ART is the most memory efficient memory among all the indexes because of path compression and adaptive sizing of nodes. The memory overhead of search layer replication is low. In case of integer key with 4-way replication of search layer, only 15% (0.48 GiB) of the total memory consumption (3.16 GiB) is used by search layer. This is significantly lower than node replication algorithm NR [8] where the over head would have been 4x. Data node in HydraList is larger than leaf node of B+ tree because it stores fingerprint, bitmap and permutation array. This leads to higher memory consumption.

6.3.3 Impact of Key Set Size

To understand the performance of index structures with increasing key set size, we measured the throughput of indexes for Workload B with string keys after adding 100K, 1M and 89M keys. Increasing key set size from 100K to 89M reduces the throughput of B+ tree by 5×. In case of HydraList and ARTROWEX, the reduction of throughput is only 2×. Masstree shows the least amount of variation in throughput with increasing size but the performance of Masstree for small key set is less.

6.3.4 Distance between Jump Node and Target Node

We measured the distance between the jump node and the target node when inserting 89 million string keys (*i.e.*, YCSB Load) using 112 threads as this distance could impact the performance of HydraList. This workload is expected to have the maximum distance between jump node and target node because of high number of inserts. Results of average of 10 runs is summarized in Figure 13. 99.5% of inserts had 0 or 1 as distance between jump node and target node while 99.99% of inserts had distance less than 5.

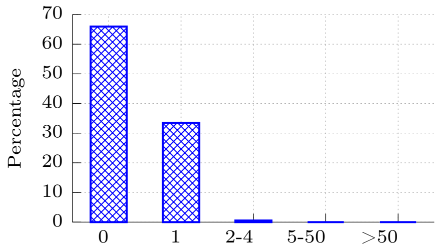

Figure 13: Distance between target node and jump node(x-axis) vs % of insert operation (y-axis).

7. DISCUSSION

Comparison with Wormhole. Wormhole [49] and HydraList differ in their design approaches. Wormhole tries to improve performance by using a very efficient search layer. But in massive multicores, as shown in our evaluation, other factors also contribute to performance. Design of Wormhole and HydraList differ in three key aspects: 1) updates to the search layer, 2) the data layout and 3) concurrency control. First, Wormhole uses a hash table (MetaTrieHT) as its search layer that is always consistent with its data layer (LeafList) leading to higher synchronization overhead and longer critical sections. Second, HydraList supports per-NUMA replication of search layers. As a result, remote cache line accesses in HydraList are significantly fewer (Figure 10). Finally, Wormhole uses RCU to synchronize its search layer which has high memory overhead as it requires two copies of a hash table to be maintained making per-node replication like HydraList unfeasible. Also their reader-writer lock for data layer is non-scalable.

Supporting Variable Length Key. In our prototype, data nodes store a maximum of 64 keys-value pointer pairs and the maximum key length is fixed to 32 bytes. It is possible to implement variable key length by storing keys smaller than 32 bytes in data nodes directly; keys larger than 32 bytes would store partial keys and a pointer to the remainder in the data node.

8. CONCLUSION

We introduce a new index structure, HydraList which is based on the idea that an index structure can be divided into two components, a search layer and a data layer. HydraList decouples these two layers and updates from the data layer to a search layer are propagated asynchronously using background threads. This design removes search layer updates from cirtical path and allows replication of search layers across NUMA nodes which further improves performance. Our evaluation shows that design of HydraList is scalable and versatile over a large variety of workloads.

9. ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035).

10. APPENDIX: PSEUDO-CODE

```
node* getLowerBound(node* curNode, uint8 unmatchedToken) {
  // With each node representing 1 byte of key
  // A node can have 256 possible children
  for (int i = unmatchedToken - 1; i >= 0; i--) {
    if (curNode->getLevelChild(i) != NULL)
      return curNode->getLevelChild(i);
  }
  return NULL;
}
node* findJumpNode(Key_t key, SearchLayer* sl) {
  int level = 0;
  // Longest Prefix Match
  for (int i = 0; i < key.length(); i++) {
    if (sl->checkPrefix(key, level)) level++;
    else break;
  }
  slNode* LPMnode = sl->getNode(level);
  if(LPMnode->type == LEAF_NODE) return sl->getValue(LPMnode);
  // Find child node whose token is lower bound of unmatched token
  slNode* Lnode = getLowerBound(LPMnode, key[level]);
  if (Lnode) {
    // Get the rightmost leaf node of subtree rooted at Lnode
    while(!sl->isLeafNode(Lnode))
      Lnode = getLowerBound(node, 255);
  } else {
    // Find child node whose token is upper bound of unmatched token
    Rnode = getUpperBound(sl->getNode(level), key[level]);
    // Get the leftmost leaf node of subtree rooted at Lnode
    while(!sl->isLeafNode(Rnode))
      node = getUpperBound(Rnode,0);
  }
  return sl->getValue(node);
}
node* findTagetNode(node* jumpNode, Key_t key) {
  node* cur = jumpNode;
  while (1) {
    if (cur->getAnchor() > key) {
      cur = cur->getPrev();
      continue;
    }
    if (cur->getNext()->getAnchor() > key) {
      cur = cur->getNext();
      continue;
    }
    break;
  }
  return cur;
}
bool checkNodeValidity(node* cur, Key_t key,
    node &head, node jumpNode) {
  if (cur->getDeleted()) { // Check if current node is deleted
    if (cur == jumpNode) { // If deleted node is jump node
      // Modify the start node of search in data layer
      head = jumNode->getNext();
    }
    return false;
  }
  //Check if current node meets the key invariant
  if (cur->checkInvariant(key))
    return false;
}
bool insert(node* jumpNode, Key_t key, Val_t val) {
  node* head = jumpNode;
restart:
  node* tagetNode = findTargetNode(head, key);
  if (!cur->writeLock()) // Acquire write lock
    goto restart;
  // Check the node invariant are met
  if (!checkNodeValidity(cur, key, head, jumpNode)){
    cur->writeUnlock();
    goto restart
  }
  uint8_t fingerPrint = getFingerPrint(key);
  int index = cur->findKey(key, fingerPrint);
  if (index > -1) // If key does exits return
    return false;
  bool ret =  cur->insertKey(key, val, fingerPrint);
  cur->writeUnlock();
  return ret;
}
Val_t lookup(node* jumpNode, Key_t key) {
  node* head = jumpNode;
restart:
  node* tagetNode = findTargetNode(head, key);
  version_t readVersion = cur->readLock();
  if (!readVersion) // Node locked, retry
    goto restart;
  if (!checkNodeValidity(cur, key, head, jumpNode))
    goto restart;
  uint8_t fingerPrint = getFingerPrint(key);
  int index = cur->findKey(key, fingerPrint);
  if (index == -1) // Key does not exits
    return nullptr;
  Val_t ret = cur->getValue(index);

  // Check if the version has changed
  if (!cur->readUnlock(readVersion))
    goto restart;
  return ret;
}
bool remove(node* jumpNode, Key_t key, Val_t val) {
  node* head = jumpNode;
restart:
  node* tagetNode = findTargetNode(head, key);
  if (!cur->writeLock()) // Acquire Write lock
    goto restart;
  if (!checkNodeValidity(cur, key, head, jumpNode)) {
    cur->writeUnlock();
    goto restart;
  }

  uint8_t fingerPrint = getFingerPrint(key);
  int index = cur->findKey(key, fingerPrint);
  if (index == -1) // If key does not exits return
    return false;
  bool ret = cur->removeKey(index);
    cur->writeUnlock();
  return ret;
}
vector<Val_t>& scan(node* jumpNode, Key_t key, int range) {
  vector<Val_t> scanVector(range);
  node* head = jumpNode;
restart:
  scanVector.clear();
  int done = 0;
  node* tagetNode = findTargetNode(head, key);
  while (done < range) {
    versiont_t readVersion cur->readLock();
    if (!readVersion) // If node is locked, retry
      goto restart;
    if (done == 0
        && !checkNodeValidity(cur, key, head, jumpNode))
      goto restart;
    int todo = range - done;

    // Check if an update has occurred since last scan
    if (readVersion > cur->getLastScanVersion()) {
      // If yes, regenerate Permutation Array
      cur->generatePermuter();
      lastScanVersion = readVersion;
    }
    uint8_t startIndex = 0;

    // Find index to start the scan
    if (startKey > cur->getAnchorKey())
      startIndex = binSearchLowerBound(startKey);
    for (uint8_t i = startIndex;
   i < cur->numEntries && todo > 0; i++) {
      int index = cur->getPermuter(i);
      scanVector[done++] = cur->getValueArray(index);
      todo--;
    }

    // Recheck read version, if changed retry
    if (!cur->readUnlock(readVersion))
      goto restart;
    cur = cur->getNext();
  }
  return scanVector;
}
```

11. REFERENCES

[1] ART Synchronized. https://github.com/flode/ARTSynchronized.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 266–277, Edinburg, Scotland, Sept. 1999.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[4] S. B. Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. OpLog: a library for scaling update-heavy data structures. *CSAIL Technical Report*, 1(1):1–12, 2013.

[5] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

[6] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[7] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, Banglore, India, Jan. 2010.

[8] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 207–221, Xi'an, China, Apr. 2017. ACM.

[9] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 181–190. Morgan Kaufmann, 2001.

[10] D. Comer. Ubiquitous b-tree. *ACM Computer Surey.*, 11(2):121–137, June 1979.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, Indiana, USA, June 2010. ACM.

[12] J. Corbet. AutoNUMA: the other approach to NUMA scheduling, 2012. https://lwn.net/Articles/488709/.

[13] T. Crain, V. Gramoli, and M. Raynal. No Hot Spot Non-blocking Skip List. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 196–205, 2013.

[14] H. Daly, A. Hassan, M. F. Spear, and R. Palmieri. NUMASK: High Performance Scalable Skip List for NUMA. In *Proceedings of the 32nd International Conference on Distributed Computing (DISC)*, pages 18:1–18:19, New Orleans, USA, Oct. 2018.

[15] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, Farmington, PA, Nov. 2013.

[16] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 631–644, Istanbul, Turkey, Mar. 2015.

[17] Dormando. memcached - a distributed memory object caching system, 2019. https://memcached.org/.

[18] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, Jan. 2012.

[19] P. Fatourou, E. Papavasileiou, and E. Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 275–286, Phoenix, AZ, USA, June 2019.

[20] fonxat. 300 Million Email Database, 2018. https://archive.org/details/300MillionEmailDatabase.

[21] K. Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004. No. UCAM-CL-TR-579.

[22] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[23] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[24] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[25] Intel. Intel AVX-512 Instructions, 2017. https://software.intel.com/en-us/articles/intel-avx-512-instructions.

[26] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[27] S. Kashyap, C. Min, K. Kim, and T. Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, pages 34:1–34:15, Porto, Portugal, Apr. 2018. ACM.

[28] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, Hannover, Germany, Apr. 2011.

[29] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim,

and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 468–479, Davis, CA, USA, Dec. 2013.

[30] P. L. Lehman et al. Efficient locking for concurrent operations on B-trees. pages 650–670, 1981.

[31] V. Leis, M. Haubenschild, and T. Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019.

[32] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 38–49, Brisbane, Australia, Apr. 2013.

[33] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The ART of Practical Synchronization. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 3:1–3:8, San Fransico, USA, June 2016.

[34] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 302–313, Brisbane, Australia, Apr. 2013.

[35] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, pages 27:1–27:14, Amsterdam, The Netherlands, Apr. 2014.

[36] W. Litwin. Trie hashing. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 19–29. ACM, 1981.

[37] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE)*, pages 604–615, Chicago, IL, Mar.–Apr. 2014.

[38] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, pages 183–196, Bern, Switzerland, Apr. 2012.

[39] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-Copy Update. In *Ottawa Linux Symposium*, OLS, 2002.

[40] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[41] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA, June 2016.

[42] J. Rao and K. A. Ross. Making B+- Trees Cache Conscious in Main Memory. In *Proceedings of the 2000 ACM SIGMOD/PODS Conference*, pages 475–486, Dallas, TX, May 2000.

[43] redislabs. Redis, 2019. https://redis.io.

[44] B. Schlegel, R. Gemulla, and W. Lehner. K-ary search on modern processors. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 52–60, Providence, RI, June 2009.

[45] 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. *Proc. VLDB Endow.*, 4(11):795–806, 2011.

[46] N. Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, Mar. 2011.

[47] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, Nov. 2013. ACM.

[48] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*, pages 473–488, Houston, TX, USA, June 2018.

[49] X. Wu, F. Ni, and S. Jiang. Wormhole: A Fast Ordered Index for In-memory Data Management. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pages 18:1–18:16, Dresden, Germany, Apr. 2019.

[50] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[51] S. Zeuch, J.-C. Freytag, and F. Huber. Adapting tree structures for processing with simd instructions. In *Proceedings of the International Conference on Extending Database Technology*, pages 97–108, Athens, Greece, Mar. 2014. OpenProceedings.org.