# Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes


Huanchen Zhang
Carnegie Mellon University
huanche1@cs.cmu.edu

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Michael Kaminsky
Intel Labs
michael.e.kaminsky@intel.com

Lin Ma
Carnegie Mellon University
lin.ma@cs.cmu.edu

Rui Shen
VoltDB
rshen@voltdb.com



## ABSTRACT

Using indexes for query execution is crucial for achieving high performance in modern on-line transaction processing databases. For a main-memory database, however, these indexes consume a large fraction of the total memory available and are thus a major source of storage overhead of in-memory databases. To reduce this overhead, we propose using a two-stage index: The first stage ingests all incoming entries and is kept small for fast read and write operations. The index periodically migrates entries from the first stage to the second, which uses a more compact, read-optimized data structure. Our first contribution is hybrid index, a dual-stage index architecture that achieves both space efficiency and high performance. Our second contribution is Dual-Stage Transformation (DST), a set of guidelines for converting any order-preserving index structure into a hybrid index. Our third contribution is applying DST to four popular order-preserving index structures and evaluating them in both standalone microbenchmarks and a full in-memory DBMS using several transaction processing workloads. Our results show that hybrid indexes provide comparable throughput to the original ones while reducing the memory overhead by up to 70%.


## 1. INTRODUCTION

Main-memory database management systems (DBMSs) target online transaction processing (OLTP) workloads that require high throughput and low latency—performance that is only available if the working set fits in memory. Although the price of DRAM has dropped significantly in recent years, it is neither free nor infinite. Memory is a non-trivial capital cost when purchasing new equipment, and it incurs real operational costs in terms of power consumption. Studies have shown that DRAM can account for up to 40% of the overall power consumed by a server [36].

Improving memory efficiency in a DBMS, therefore, has two benefits. First, for a fixed working set size, reducing the required memory can save money, both in capital and operating expenditures. Second, improving memory efficiency allows the DBMS to keep more data resident in memory. Higher memory efficiency allows for a larger working set, which enables the system to achieve higher performance with the same hardware.

Index data structures consume a large portion of the database's total memory—particularly in OLTP databases, where tuples are relatively small and tables can have many indexes. For example, when running TPC-C on H-Store [33], a state-of-the-art in-memory DBMS, indexes consume around 55% of the total memory. Freeing up that memory can lead to the above-mentioned benefits: lower costs and/or the ability to store additional data. However, simply getting rid of all or part of the indexes is suboptimal because indexes are crucial to query performance.

This paper presents **hybrid index**, a dual-stage index architecture that substantially reduces the per-tuple index space with only modest costs in throughput and latency (and, indeed, is actually faster for some workloads) even when the reclaimed memory is not exploited to improve system performance. Although the idea of using multiple physical data structures to construct a logical entity has been explored before [4, 28, 29, 43, 44, 47], to the best of our knowledge, we are the first to show the benefits of applying it to indexes in main-memory OLTP DBMSs.

The dual-stage architecture maintains a small dynamic "hot" store to absorb writes and a more compact, but read-only store to hold the bulk of index entries. Merge between the stages is triggered periodically and can be performed efficiently. Unlike prior work [20, 35, 39, 47, 53], our design offers low latency and high throughput for the point queries and short-range scans that typify the OLTP workloads used with main-memory databases [34, 50].

Hybrid index leverages the skew typically found in OLTP workloads. This skew manifests itself with respect to item popularity [23, 49]: certain data items are accessed more often than others and thus are more likely to be accessed again in the near future. This observation has been used extensively to move cold data from memory to block-based storage [23, 27, 48], and to store data efficiently by compressing the cold data in a main-memory database [29]. To our knowledge, however, previous work has not exploited skew to shrink the indexes themselves, particularly for purely in-memory structures. Because the indexes can take more than half the memory for an OLTP workload, doing so has the potential to save a considerable amount of still-expensive DRAM without significant performance penalty.

To facilitate building hybrid indexes, we provide **Dual-Stage Transformation** (**DST**), a set of guidelines that can convert any order-preserving index structure to a hybrid index. We applied DST to four order-preserving index structures: B+tree, Masstree [41], Skip List [46], and ART [37]. Using both YCSB-based microbench-



*SIGMOD'16,* June 26-July 01, 2016, San Francisco, CA, USA



http://dx.doi.org/10.1145/2882903.2915222

| | Tuples | Primary Indexes | Secondary Indexes |
|---|---|---|---|
| TPC-C | 42.5% | 33.5% | 24.0% |
| Articles | 64.8% | 22.6% | 12.6% |
| Voter | 45.1% | 54.9% | 0% |

**Table 1:** Percentage of the memory usage for tuples, primary indexes, and secondary indexes in H-Store using the default indexes (DB size $\approx$ 10 GB).

marks and by integrating them into the H-Store OLTP DBMS [33], we find that hybrid indexes meet the goals, achieving performance comparable to the original indexes while reducing index memory consumption by up to 70% (Section 6). Also, by using hybrid indexes, H-Store is able to sustain higher throughput for OLTP workloads for more transactions and with fewer performance interruptions due to memory limitations (Section 7).

The contributions of this paper are as follows. First, we propose hybrid index, a dual-stage index architecture that can improve the memory efficiency of indexes in main-memory OLTP databases. Second, we introduce DST, a set of guidelines that includes a simple dynamic-to-static transformation to help design hybrid indexes from existing index data structures. Third, we applied DST to four index structures and show the effectiveness and generality of our method. Finally, the dual-stage architecture provides the opportunity to use compact/compressed static data structures (e.g., succinct data structures [32]) for database indexing.

## 2. THE CASE FOR HYBRID INDEXES

The overhead of managing disk-resident data has given rise to a new class of OLTP DBMSs that store the entire database in main memory [24, 25, 28, 50]. These systems outperform traditional disk-oriented DBMSs because they eschew the legacy components that manage data stored on slow, block-based storage [30]. Unfortunately, this improved performance is achievable only when the database is smaller than the amount of physical memory available in the system. If the database does not fit in memory, then the operating system will move virtual memory pages out to disk, and memory accesses will cause page faults [48]. Because these page faults are transparent to the DBMS, the threads executing transactions will stall while the page is fetched from disk, degrading the system's throughput and responsiveness. Thus, the DBMS must use memory efficiently to avoid this performance bottleneck.

Indexes are a major factor in the memory footprint of a database. OLTP applications often maintain several indexes per table to ensure that queries execute quickly. This is important in applications that interact with users and other external systems where transactions must complete in milliseconds [50]. These indexes consume a significant fraction of the total memory used by a database. To illustrate this point, Table 1 shows the relative amount of storage used for indexes in several OLTP benchmarks deployed in a main-memory DBMS [3]. We used the DBMS's internal statistics API to collect these measurements after running the workloads on a single node until the database size $\approx$ 10 GB. Indexes consume up to 58% of the total database size for these benchmarks, which is commensurate with our experiences with real-world OLTP systems.

Designing memory-efficient indexes is thus important for improving database performance and reducing costs. Achieving space-efficient indexes is, however, non-trivial because there are trade-offs between function, performance, and space. For example, hash tables are fast and potentially more space-efficient than tree-based data structures, but they do not support range queries, which prevents them from being ubiquitous.

A common way to reduce the size of B+trees is to compress their nodes before they are written to disk using a general-purpose compression algorithm (e.g., LZMA) [8]. This approach reduces the I/O cost of fetching pages from disk, but the nodes must be decompressed once they reach memory so that the system can interpret their contents. To the best of our knowledge, the only compressed main-memory indexes are for OLAP systems, such as bitmap [20] and columnar [35] indexes. These techniques, however, are inappropriate for the write-heavy workload mixtures and small-footprint queries of OLTP applications [50]. As we show in Section 6, compressed indexes perform poorly due to the overhead of decompressing an entire block to access a small number of tuples.

An important aspect of these previous approaches is that the indexes treat all of the data in the underlying table equally. That is, they assume that the application will execute queries that access all of the table's tuples in the same manner, either in terms of frequency (i.e., how many times it will be accessed or modified in the future) or use (i.e., whether it will be used most in point versus range queries). This assumption is incorrect for many OLTP applications. For example, a new tuple is likely to be accessed more often by an application soon after it was added to the database, often through a point query on the index. But as the tuple ages, its access frequency decreases. Later, the only time it is accessed is through summarization or aggregation queries.

One could handle this scenario through multiple partial indexes on the same keys in a table that use different data structures. There are several problems with this approach beyond just the additional cost of maintaining more indexes—foremost is that developers might need to modify their application so that each tuple specifies what index it should be stored in at runtime. This information is necessary because some attributes, such as a tuple's creation timestamp, may not accurately represent how likely it will be accessed in the future. Second, the DBMS's query optimizer might not be able to infer what index to use for a query since a particular tuple's index depends on this identifying value. If a complex query accesses tuples from multiple partial indexes that each has a portion of the table's data, then the system will need to retrieve data from multiple sources for that query operator. This type of query execution is not possible in today's DBMSs, so the system would likely fall back to scanning the table sequentially.

We argue that a better approach is to use a single logical hybrid index that is composed of multiple data structures. This approach gives the system more fine-grained control over data storage without requiring changes to the application. To the rest of the DBMS, a hybrid index looks like any other, supporting a conventional interface and API. Previous work showed the effectiveness of using multiple physical data structures or building blocks to construct a higher-level logical entity. Most notable are log-structured merge-trees [44]. More recent examples include SILT [39], a flash-based key-value store that achieves both memory-efficiency and high-performance by combining three basic stores with different optimization focuses to effectively overcome each others' deficiencies. Anvil [40] offers a toolkit for building database backends out of specialized storage modules, each of which is optimized for different workload objectives. OctopusDB [26] and WiredTiger [17] take a similar approach, but at a table-level granularity.

Applying these ideas to database indexes is a natural fit, especially for in-memory OLTP systems. In these applications, transactions' access patterns vary over time with respect to age and use. Index entries for new tuples go into a fast, write-friendly data structure since they are more likely to be queried again in the near future. Over time, the tuples become colder and their access patterns change, usually from frequent modification to occasional read [23]. Aged tuples thus eventually migrate to a more read-friendly and more compact data structure to save space [47].

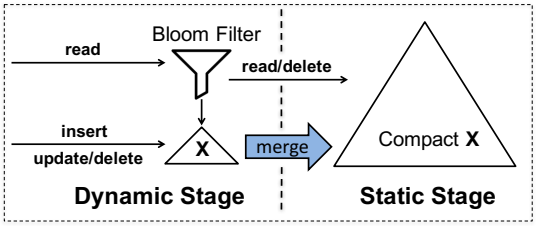



**Figure 1: Dual-Stage Hybrid Index Architecture** – All writes to the index first go into the dynamic stage. As the size of the dynamic stage grows, it periodically merges older entries to the static stage. For a read request, it searches the dynamic stage and the static stage in sequence.

Given this, we present our dual-stage architecture for hybrid indexes in the next section. Section 4 introduces a set of transformation guidelines for designing compact, read-optimized data structures. Section 5 discusses the techniques for migrating data between stages.

# 3. THE DUAL-STAGE ARCHITECTURE

As shown in Figure 1, the dual-stage hybrid index architecture is comprised of two stages: the *dynamic* stage and the *static* stage. New entries are added to the dynamic stage. This stage is kept small so that queries to the most recent entries, which are likely to be accessed and modified in the near future, are fast. As the size of the dynamic stage grows, the index periodically triggers the merge process and migrates aged entries from its dynamic stage to the static stage which uses a more space-efficient data structure to hold the bulk of the index entries. The static stage does not support direct key additions or modifications. It can only incorporate key updates in batches through the merge process.

A hybrid index serves read requests (point queries, range queries) by searching the stages in order. To speed up this process, it maintains a Bloom filter for the keys in the dynamic stage so that most point queries search only one of the stages. Specifically, for a read request, the index first checks the Bloom filter. If the result is positive, it searches the dynamic stage and the static stage (if necessary) in sequence. If the result is negative, the index bypasses the dynamic stage and searches the static stage directly. The space overhead of the Bloom filter is negligible because the dynamic stage only contains a small subset of the index's keys.

A hybrid index handles value updates differently for primary and secondary indexes. To update an entry in a primary index, a hybrid index searches the dynamic stage for the entry. If the target entry is found, the index updates its value in place. Otherwise, the index inserts a new entry into the dynamic stage. This insert effectively overwrites the old value in the static stage because subsequent queries for the key will always find the updated entry in the dynamic stage first. Garbage collection for the old entry is postponed until the next merge. We chose this approach so that recently modified entries are present in the dynamic stage, which speeds up subsequent accesses. For secondary indexes, a hybrid index performs value updates in place even when the entry is in the static stage, which avoids the performance and space overhead of having the same key valid in both stages.

For deletes, a hybrid index first locates the target entry. If the entry is in the dynamic stage, it is removed immediately. If the entry is in the static stage, the index marks it "deleted" and removes it at the next merge. Again, depending on whether it is a unique index or not, the DBMS may have to check both stages for entries.

This dual-stage architecture has two benefits over the traditional single-stage indexes. First, it is space-efficient. The periodically-triggered merge process guarantees that the dynamic stage is much smaller than the static stage, which means that most of the entries are stored in a compact data structure that uses less memory per entry. Second, a hybrid index exploits the typical access patterns in OLTP workloads where tuples are more likely to be accessed and modified soon after they were added to the database (Section 2). New entries are stored in the smaller dynamic stage for fast reads and writes, while older (and therefore unchanging) entries are migrated to the static stage only for occasional look-ups.

More importantly, the dual-stage architecture opens up the possibility of using compact/compressed static data structures that have not yet been accepted for database indexing, especially in OLTP databases. These data structures, including succinct data structures for example, provide excellent compression rates but suffer the drawback of not being able to support dynamic operations such as inserts and updates efficiently. A hybrid index can use these specialized data structures in its static stage—leveraging their compression to gain additional memory efficiency while amortize their performance impact by applying updates in batches. This paper examines several general purpose data structures, but we leave exploration of more esoteric data structures as future work.

To facilitate using the dual-stage architecture to build hybrid indexes, we provide the following Dual-Stage Transformation (DST) guidelines for converting any order-preserving index structure to a corresponding hybrid index:

- **Step 1:** Select an order-preserving index structure (**X**) that supports dynamic operations efficiently for the dynamic stage.
- **Step 2:** Design a compact, read-optimized version of **X** for the static stage.
- **Step 3:** Provide a merge routine that can efficiently migrate entries from **X** to compact **X**.
- **Step 4:** Place **X** and compact **X** in the dual-stage architecture as shown in Figure 1.

We describe the process to accomplish Step 2 of DST in the next section. We note that these steps are a manual process. That is, a DBMS developer would need to convert the index to its static version. Automatically transforming any arbitrary data structure is outside the scope of this paper.

# 4. DYNAMIC-TO-STATIC RULES

As described in Section 3, the static stage is where the index stores older entries in a more memory-efficient format. The cost of this efficiency is that the set of keys in the data structure cannot change. The only way to add new entries is to rebuild the data structure.

The ideal data structure for the static stage must have three properties: First, it must be memory-efficient (i.e., have low space overhead per entry). Second, it must have good read performance for both point queries and range queries. This is particularly important for primary indexes where guaranteeing key uniqueness requires checking the static stage for every insert. Third, the data structure must support merging from the dynamic stage efficiently. This not only means that the merge process is fast, but also that the temporary memory use is low. These three properties have trade-offs, and it is difficult to achieve all three in a single data structure (e.g., more compact data structures are usually slower). Our design, therefore, aims to strike a balance; alternate application-specific two-stage designs that make a different trade-off are possible.

Although any data structure with the above properties can be used in the static stage, using one that inter-operates well with dynamic stage's index simplifies the design and improves merge performance. Therefore, we present the Dynamic-to-Static (D-to-S)

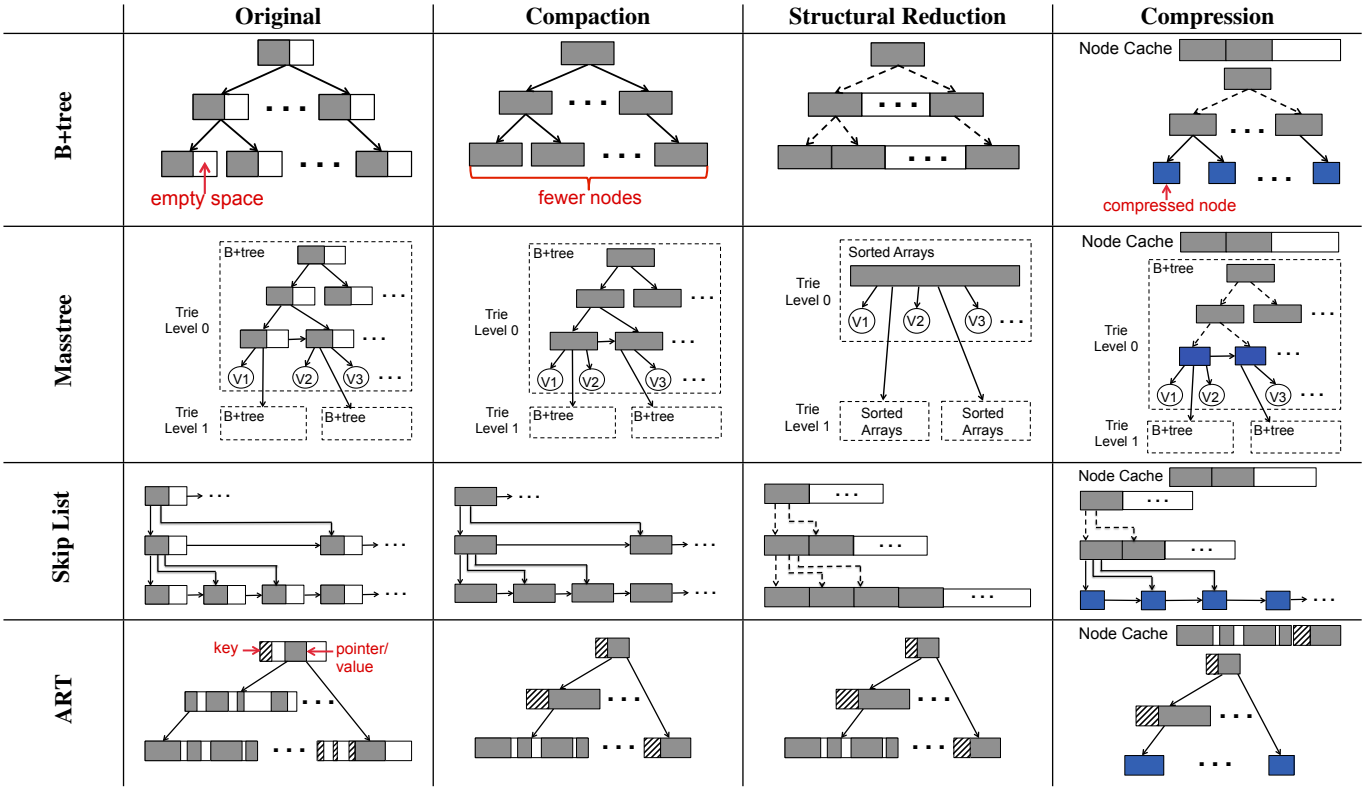



**Figure 2: Examples of Applying the Dynamic-to-Static Rules** – Solid arrows are pointers; dashed arrows indicate that the child node location is calculated rather than stored in the structure itself. The second column shows the starting points: the original dynamic data structures. After applying the Compaction Rule, we get intermediate structures in column three. We then applied the Structural Reduction Rule on those intermediate structures and obtain more compact final structures in column four. Column five shows the results of applying the Compression Rule, which is optional depending on workloads.

rules to help convert a dynamic stage data structure to a smaller, immutable version for use in the static stage.

The crux of the D-to-S process is to exploit the fact that the static stage is read-only and thus extra space allocated for efficient dynamic operations can be removed from the original structure. We observe two major sources of wasted space in dynamic data structures. First, dynamic data structures allocate memory at a coarse granularity to minimize the allocation/reallocation overhead. They usually allocate an entire node or memory block and leave a significant portion of that space empty for future entries. Second, dynamic data structures contain a large number of pointers to support fast modification of the structures. These pointers not only take up space but also slow down certain operations due to pointer-chasing.

Given a dynamic data structure, the D-to-S rules are:

- **Rule #1: Compaction** – Remove duplicated entries and make every allocated memory block 100% full.
- **Rule #2: Structural Reduction** – Remove pointers and structures that are unnecessary for efficient read-only operations.
- **Rule #3: Compression** – Compress parts of the data structure using a general purpose compression algorithm.

A data structure created by applying one or more of these rules is a good data structure for the static stage. First, it is more memory-efficient than the dynamic stage's index. Second, the data structure preserves the "essence" of the original index (i.e., it does not fundamentally change its core design). This is important because applications sometimes choose certain index structures for certain workload patterns. For example, one may want to use a trie-based data structure to efficiently handle variable-length keys that have common prefixes. After applying the D-to-S rules, a static trie is still a trie. Moreover, the similarity between the original and the compact structure enables an efficient merge routine to be implemented and performed without significant space overhead.

In the rest of this section, we describe how to apply the D-to-S rules to create static versions of four different indexes.

## 4.1 Example Data Structures

We briefly introduce the data structures that we applied the D-to-S rules on. Their structures are shown in Figure 2.

**B+tree:** The B+tree is the most common index structure that is used in almost every OLTP DBMS [21]. It is a self-balancing search tree, usually with a large fanout. Although originally designed for disk-oriented databases to minimize disk seeks, B+trees have maintained their prevalence in the main-memory DBMSs (Appendix A). For our analysis, we use the STX B+tree [19] as our baseline implementation. We found in our experiments that a node size of 512 bytes performs best for in-memory operations.

**Masstree:** Masstree [41] is a high-performance key-value store that also supports range queries. Masstree combines B+trees and tries to speed up key searches. The trie design makes the index particularly efficient in terms of both performance and space when handling keys with shared prefixes. In the diagram shown in Figure 2, the dashed rectangles in Masstree represent the individual B+trees that conceptually form a trie. The keys are divided into fixed-length 8-byte keyslices and are stored at each trie level. Unique key suffixes are stored separately in a structure called a *keybag*. Each B+tree leaf node has an associated keybag with a maximum capacity equal to the fanout of the B+tree. A value pointer in a

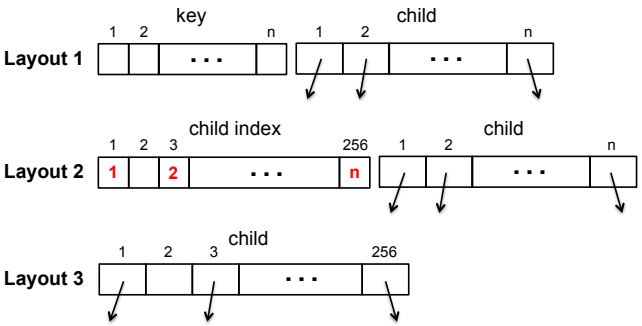



**Figure 3: ART Node Layouts** – Organization of the ART index nodes. In Layout 1, the key and child arrays have the same length and the child pointers are stored at the corresponding key positions. In Layout 2, the current key byte is used to index into the child array, which contains offsets/indexes to the child array. The child array stores the pointers. Layout 3 has a single 256-element array of child pointers as in traditional radix trees [37].

leaf node can point to either a data record (when the corresponding keyslice is uniquely owned by a key), or a lower-level B+tree.

**Skip List:** The Skip List was introduced in 1990 as an alternative to balanced trees [46]. It has recently gained attention as a lock-free index for in-memory DBMSs [45]. The internal structure of the index is a linked hierarchy of subsequences that is designed to "skip" over fewer elements. The algorithms for insertion and deletion are designed to be more simple and potentially faster than equivalent operations in balanced trees. For our analysis, we use an implementation [2] of a variation of Skip List (called a *paged-deterministic Skip List* [42]) that resembles a B+tree.

**Adaptive Radix Tree:** The Adaptive Radix Tree (ART) [37] is a fast and space-efficient data structure designed for in-memory databases. ART is a 256-way radix tree (i.e., each level represents one byte of the key). Unlike traditional radix trees (or tries) where each node is implemented as a fixed-size (256 in this case) array of child pointers, ART uses four node types (`Node4`, `Node16`, `Node48`, and `Node256`) with different layouts and capacities adaptively to achieve better memory efficiency and better cache utilization. Figure 3 illustrates the three node layouts used in ART. `Node4` and `Node16` use the representation in Layout 1 with n=4, 16, respectively. `Node48` uses Layout 2 (n=48), and `Node256` uses Layout 3.

## 4.2 Rule #1: Compaction

This first rule seeks to generate a more efficient layout of an index's entries by minimizing the number of memory blocks allocated. This rule includes two steps. The first is to remove duplicate content. For example, to map multiple values to a single key (for secondary indexes), dynamic data structures often store the same key multiple times with different values. Such key duplication is unnecessary in a static data structure because the number of values associated with each key is fixed. The second step is to fill all allocated memory blocks to 100% capacity. This step may include modifications to the layouts of memory blocks/nodes. Memory allocation is done at a fine granularity to eliminate gaps between entries; furthermore, leaving spacing for future entries is unnecessary since the data structure is static. The resulting data structure thus uses fewer memory blocks/nodes for the same entries.

As shown in Figure 2, a major source of memory waste in a B+tree, Masstree, Skip List, or ART is the empty space in each node. For example, the expected node occupancy of a B+tree is only 69% [54]. We observed similar occupancies in the Masstree and Skip List. For ART, our results show that its node occupancy

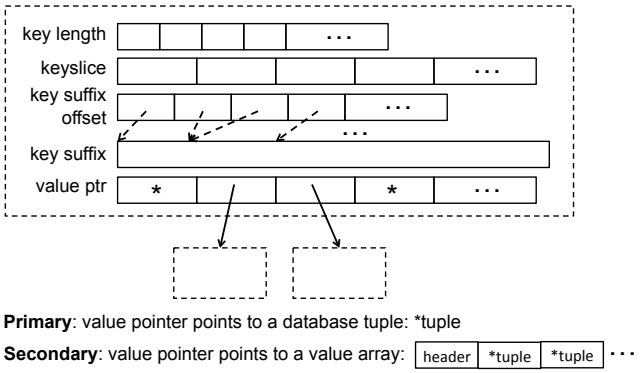



**Figure 4: Compact Masstree** – The internal architecture of the Masstree index after applying the Compaction and Structural Reduction Rules.

is only 51% for 50 million 64-bit random integer keys. This empty space is pre-allocated to ingest incoming entries efficiently without frequent structural modifications (i.e., node splits). For B+tree, Masstree and Skip List, filling every node to 100% occupancy, as shown in Figure 2 (column 3), reduces space consumption by 31% on average without any structural changes to the index itself.

ART's prefix tree structure prevents us from filling the fixed-sized nodes to their full capacity. We instead customize the size of each node to ensure minimum slack space. This is possible because the content of each node is fixed and known when building the static structure. Specifically, let $n$ denote the number of key-value pairs in an ART node ($2 \leq n \leq 256$). We choose the most space-efficient node layout in Figure 3 based on $n$. If $n \leq 227$, Layout 1 with array length $n$ is used; otherwise, Layout 3 is used.

Because of the multi-structure design, compacting Masstree's memory blocks is a more complicated process: both its internal nodes and its dynamically-allocated keybags for suffixes require modification. We found that the original implementation of Masstree allocates memory for the keybags aggressively, which means that it wastes memory. Thus, for this rule, we instead only allocate the minimum memory space to store these suffixes.

For secondary indexes where a key can map to multiple values, the only additional change to the indexes is to remove duplicated entries by storing each key once followed by an array of its associated values.

## 4.3 Rule #2: Structural Reduction

The goal of this next rule is to minimize the overhead inherent in the data structure. This rule includes removing pointers and other elements that are unnecessary for read-only operations. For example, the pointers in a linked list are designed to allow for fast insertion or removal of entries. Thus, removing these pointers and instead using a single array of entries that are stored contiguously in memory saves space and speeds up linear traversal of the index. Similarly, for a tree-based index with fixed node sizes, we can store the nodes contiguously at each level and remove pointers from the parent nodes to their children. Instead, the location of a particular node is calculated based on in-memory offsets. Thus, in exchange for a small CPU overhead to compute the location of nodes at runtime we achieve memory savings. Besides pointers, other redundancies include auxiliary elements that enable functionalities that are unnecessary for static indexes (e.g., transaction metadata).

We applied this rule to our four indexes. The resulting data structures are shown in Figure 2 (column four). We note that after the reduction, the nodes in B+tree, Masstree, and Skip List are stored contiguously in memory. This means that unnecessary pointers are gone (dashed arrows indicate that the child nodes' locations in

memory are calculated rather than stored). For ART, however, because its nodes have different sizes, finding a child node requires a "base + offset" or similar calculation, so the benefit of storing nodes contiguously is not clear. We, therefore, keep ART unchanged here and leave exploring other layouts as future work.

There are additional opportunities for reducing the space overhead with this rule. For example, the internal nodes in the B+tree, Masstree, and Skip List can be removed entirely. This would provide another reduction in space but it would also make point queries slower. Thus, we keep these internal nodes in B+tree and Skip List. For the Masstree, however, it is possible to do this without a significant performance penalty. This is because most of the trie nodes in Masstree are small and do not benefit from a B+tree structure. As a result, our compacted version of Masstree only stores the leaf nodes contiguously as an array in each trie node. To perform a look-up, it uses binary search over this array instead of a B+tree walk to find the appropriate entries. Our results show that performing a binary search is as fast as searching a B+tree in Masstree. We also note that this rule does not affect Masstree's overall trie, a distinguishing feature of Masstree compared to B+trees and Skip Lists.

We also need to deal with Masstree's keybags. In Figure 4, we provide a detailed structure of the compacted Masstree. It concatenates all the key suffixes within a trie node and stores them in a single byte array, along with an auxiliary offset array to mark their start locations. This reduces the structural overhead of maintaining multiple keybags for each trie node.

## 4.4 Rule #3: Compression

The final rule is to compress internal nodes or memory pages used in the index. For this step, we can use any general-purpose compression algorithm. We choose the ones that are designed to have fast decompression methods in exchange for a lower compression rate, such as Snappy [14] or LZ4 [5]. Column five of Figure 2 shows how we apply the Compression Rule to the B+tree, Masstree, Skip List and ART. Only the leaf nodes are compressed so that every point query needs to decompress at most one node. To minimize the cost of an expensive decompress-node operation, we maintain a cache of recently decompressed nodes. The node cache approximates LRU using the CLOCK replacement algorithm.

This rule is not always necessary for hybrid indexes because of its performance overhead. Our results in Sections 6 and 7 show that using compression for in-memory data structures is expensive even with performance optimizations, such as node caching. Furthermore, the compression ratio is not guaranteed because it depends heavily on the workload, especially the key distribution. For many applications, the significant degradation in throughput may not justify the space savings; nevertheless, compression remains an option for environments with significant space constraints.

# 5. MERGE

This section focuses on Step 3 of the Dual-Stage Transformation guidelines: merging tuples from the dynamic stage to the static stage. Although the merge process happens infrequently, it should be fast and efficient on temporary memory usage. Instead of using standard copy-on-write techniques, which would double the space during merging, we choose a more space-efficient merge algorithm that blocks all queries temporarily. There are trade-offs between blocking and non-blocking merge algorithms. Blocking algorithms are faster but hurt tail latency while non-blocking algorithms execute more smoothly but affect more queries because of locking and latching. Implementing non-blocking merge algorithms is out of the scope of this paper, and we defer it to future work.

The results in Section 6.3 show that our merge algorithm takes 60 ms to merge a 10 MB B+tree into a 100 MB Compact B+tree. The merge time increases linearly as the size of the index grows. The space overhead of the merge algorithm, however, is only the size of the largest array in the dynamic stage structure, which is almost negligible compared to the size of the entire dual-stage index. Section 5.1 describes the merge algorithm. Section 5.2 discusses two important runtime questions: (1) what data to merge from one stage to the next; and (2) when to perform this merge.

## 5.1 Merge Algorithm

Even though individual merge algorithms can vary significantly depending on the complexity of the data structure, they all have the same core component. The basic building blocks of a compacted data structure are sorted arrays containing all or part of the index entries. The core component of the merge algorithm is to extend those sorted arrays to include new elements from the dynamic stage. When merging elements from the dynamic stage, we control the temporary space penalty as follows. We allocate a new array adjacent to the original sorted array with just enough space for the new elements from the dynamic stage. The algorithm then performs in-place merge sort on the two consecutive sorted arrays to obtain a single extended array. The temporary space overhead for merging in this way is only the size of the smaller array, and the in-place merge sort completes in linear time.

The steps for merging B+tree and Skip List to their compacted variations are straightforward. They first merge the leaf-node arrays using the algorithm described above, and then rebuild the internal nodes. The internal nodes are constructed based on the merged leaf nodes so that the balancing properties of the structures are maintained. Merging Masstree and ART to their compacted versions, however, are more complicated. When merging two trie nodes, the algorithms (depth-first) recursively create new merging tasks when two child nodes (or leaves/suffixes) require further merging. We present the detailed algorithms in Appendix B.

## 5.2 Merge Strategy

In this section, we discuss two important design decisions: (1) what to merge, and (2) when to merge.

**What to Merge:** On every merge operation, the system must decide which entries to move from the dynamic stage to the static stage. Strategy one, called merge-all, merges the entire set of dynamic stage entries. This choice is based on the observation that many OLTP workloads are insert-intensive with high merge demands. Moving everything to the static stage during a merge makes room for the incoming entries and alleviates the merge pressure as much as possible. An alternative strategy, merge-cold, tracks key popularity and selectively merges the cold entries to the static stage.

The two strategies interpret the role of the dynamic stage differently. Merge-all treats the dynamic stage as a write buffer that absorbs new records, amortizing the cost of bulk insert into the static stage. Merge-cold, however, treats the dynamic stage as a write-back cache that holds the most recently accessed entries. Merge-cold represents a tunable spectrum of design choices depending on how hot and cold are defined, of which merge-all is one extreme.

The advantage of merge-cold is that it creates "shortcuts" for accessing hot entries. However, it makes two trade-offs. First, it typically leads to higher merge frequency because keeping hot entries renders the dynamic stage unable to absorb as many new records before hitting the merge threshold again. The merge itself will also be slower because it must consider the keys' hot/cold sta-

tus. Second, merge-cold imposes additional overhead for tracking an entry's access history during normal operations.

Although merge-cold may work better in some cases, given the insert-intensive workload patterns of OLTP applications, we consider merge-all to be the more general and more suitable approach. We compensate for the disadvantage of merge-all (i.e., some older yet hot tuples reside in the static stage and accessing them requires searching both stages in order) by adding a Bloom filter atop the dynamic stage as described in Section 3.

**When to Merge:** The second design decision is what event triggers the merge process to run. One strategy to use is a ratio-based trigger: merge occurs whenever the size ratio between the dynamic and the static stages reaches a threshold. An alternative strategy is to have a constant trigger that fires whenever the size of the dynamic stage reaches a constant threshold.

The advantage of a ratio-based trigger is that it automatically adjusts the merge frequency according to the index size. This strategy prevents write-intensive workloads from merging too frequently. Although each merge becomes more costly as the index grows, merges happen less often. One can show that the merge overhead over time is constant. The side effect is that the average size of the dynamic stage gets larger over time, resulting in an increasingly longer average search time in the dynamic stage.

A constant trigger works well for read-intensive workloads because it bounds the size of the dynamic stage ensuring fast look-ups. For write-intensive workloads, however, this strategy leads to higher overhead because it keeps a constant merge frequency even though merging becomes more expensive over time. We found that a constant trigger is not suitable for OLTP workloads due to too frequent merges. We perform a sensitivity analysis of the ratio-based merge strategy in Appendix C. Although auto-tuning is another option, it is beyond the scope of this paper.

# 6. MICROBENCHMARK EVALUATION

For our evaluation, we created five hybrid indexes using our DST guidelines proposed in Section 3. We use X to represent either B+tree, Masstree, Skip List, or ART. *Hybrid-Compact* (or simply *Hybrid*) X means that the static stage uses the Compaction and Structural Reduction D-to-S Rules from Section 4. *Hybrid-Compressed* means that the static stage structure is also compressed using Snappy [14] according to the Compression Rule. We implemented Hybrid-Compact for all four data structures. We only implemented Hybrid-Compressed B+tree for this evaluation to verify our conclusion that using general-purpose compression is not a viable solution for improving space-efficiency of indexes in main-memory OLTP databases.

We evaluate hybrid indexes in two steps. In this section, we evaluate the hybrid index as stand-alone key-value data structure using YCSB-based microbenchmarks. We first show the separate impact on performance and space of a hybrid index's building blocks. We then compare each hybrid index to its original structure to show the performance trade-offs made by adopting a hybrid approach for better space efficiency. We did not use an existing DBMS for this section because we did not want to taint our measurement with features that are not relevant to the evaluation.

In Section 7, we evaluate hybrid indexes inside H-Store, a state-of-the-art main-memory OLTP database system. We replace the default STX B+tree with the corresponding transformed hybrid indexes and evaluate the entire DBMS end-to-end.

## 6.1 Experiment Setup & Benchmarks

We used a server with the following configuration in our evaluation:

**CPU:** 2×Intel® Xeon® E5-2680 v2 CPUs @ 2.80 GHz
**DRAM:** 4×32 GB DDR3 RAM
**Cache:** 256 KB L2-cache, 26 MB L3-cache
**Disk:** 500 GB, 7200 RPM, SATA (used only in Section 7)

We used a set of YCSB-based microbenchmarks to mimic OLTP index workloads. The Yahoo! Cloud Serving benchmark (YCSB) approximates typical large-scale cloud services [22]. We used its default workloads **A** (*read/write*, 50/50), **C** (*read only*), and **E** (*scan/insert*, 95/5) with Zipfian distributions, which have skewed access patterns common to OLTP workloads. The initialization phase in each workload was also measured and reported as the *insert-only* workload. For each workload, we tested three key types: 64-bit random integers, 64-bit monotonically increasing integers, and email addresses with an average length of 30 bytes. The random integer keys came directly from YCSB while the email keys were drawn from a large email collection. All values are 64-bit integers to represent tuple pointers. To summarize:

**Workloads:** insert-only, read-only, read/write, scan/insert
**Key Types:** 64-bit random int, 64-bit mono-inc int, email
**Value:** 64-bit integer (tuple pointers)

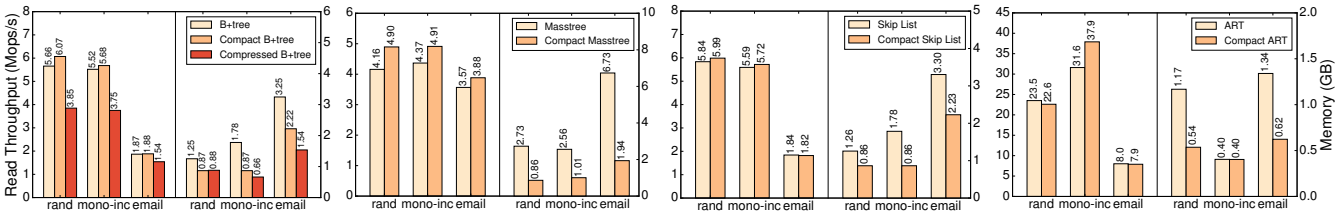
All experiments in this section are single-threaded without any network activity. An initialization phase inserts 50 million entries into the index. Then the measurement phase executes 10 million key-value queries according to the workload. Throughput results in the bar charts are the number of operations divided by the execution time; memory consumption is measured at the end of each trial. All numbers reported are the average of three trials.

## 6.2 Compaction & Compression Evaluation

We analyze hybrid indexes by first evaluating their building blocks. As mentioned in Section 4, the data structure for the static stage must be both space-efficient and fast for read operations. To test this, we compare the data structures created by applying the D-to-S (Compact/Compressed X) rules to the original data structures (X) using the *read-only* workloads described in Section 6.1. Here, X represents either B+tree, Masstree, Skip List, or ART . Note that this is an evaluation of the data structures alone without the runtime merging process that is needed to move data from the dynamic stage to the static stage. We consider the complete hybrid index architecture (with merging) in Section 6.4.

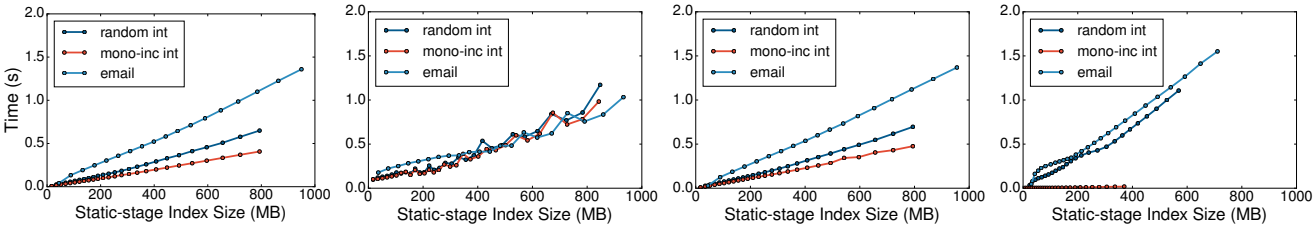
As Figure 5 shows, the read throughput for the compact indexes is up to 20% higher in most cases compared to their original data structures. This is not surprising because these compact versions inherit the core design of their original data structures but achieve a more space-efficient layout with less structural overhead. This results in fewer nodes/levels to visit per look-up and better cache performance. The only compact data structure that performs slightly worse is the Compact ART for random integer (4%) and email keys (1%). This is because unlike the other three compact indexes, Compact ART uses a slightly different organization for its internal nodes that causes a degradation in performance in exchange for a greater space saving (i.e., Layout 1 is slower than Layout 3 for look-ups – see Section 4.2).

Figure 5 also shows that the compact indexes reduce the memory footprint by up to 71% (greater than 30% in all but one case). The savings come from higher data occupancy and less structural waste (e.g., fewer pointers). In particular, the Compact ART is only half the size for random integer and email keys because ART has relatively low node occupancy (54%) compared to B+tree and Skip List (69%) in those cases. For monotonically increasing (mono-inc) integer keys, the original ART is already optimized for space.




**(a)** B+tree **(b)** Masstree **(c)** Skip List **(d)** ART

**Figure 5: Compaction & Compression Evaluation** – Read performance and storage overhead for the compacted and compressed data structures generated by applying the D-to-S rules. Note that the figures have different Y-axis scales (rand=random integer, mono-inc=monotonically increasing integer).




**(a)** B+tree **(b)** Masstree **(c)** Skip List **(d)** ART

**Figure 6: Merge Overhead** – Absolute merge time given the static-stage index size. Dynamic-stage index size = $\frac{1}{10}$ static-stage index size.

| | Instructions | IPC | L1 Misses | L2 Misses |
|---|---|---|---|---|
| B+tree | 4.9B | 0.8 | 262M | 160M |
| Masstree | 5.4B | 0.64 | 200M | 174M |
| Skip List | 4.5B | 0.78 | 277M | 164M |
| ART | 2.1B | 1.5 | 58M | 26M |

**Table 2: Point Query Profiling** – CPU-level profiling measurements for 10M point queries of random 64-bit integer keys for B+tree, Masstree, Skip List, and ART (B=billion, M=million).

The Compact Masstree has the most space savings compared to the others because its internal structures (i.e., B+trees) are completely flattened into sorted arrays.

We also tested the Compression Rule (Section 4.4) on the B+tree. As shown in Figure 5a, although the Compressed B+tree saves additional space for the mono-inc (24%) and email (31%) keys, the throughput decreases from 18–34%. Since the other data structures have the same problems, we choose not to evaluate compressed versions of them and conclude that naïve compression is a poor choice for in-memory OLTP indexes. Thus, for the rest of this paper, when we refer to a "Hybrid" index, we mean one whose static stage-structure is compacted (not compressed, unless explicitly stated).

Overall we see that ART has higher point query performance than the other three index structures. To better understand this, we profiled the 10 million point queries of random 64-bit integer keys for the four original data structures using `PAPI` [10]. Table 2 shows the profiling results for total CPU instructions, instructions per cycle (IPC), L1 cache misses and L2 cache misses. We observe that ART not only requires fewer CPU instructions to perform the same load of point queries, but also uses cache much more efficiently than the other three index structures.

## 6.3 Merge Strategies & Overhead

We next evaluate the merge process that moves data from the dynamic stage to the static stage at runtime. We concluded in Section 5.2 that ratio-based triggers are more suitable for OLTP applications because it automatically adjusts merge frequency according to the index size. We show a sensitivity analysis of the ratio-based merge strategy in Appendix C. Based on the analysis, we choose 10 as the default merge ratio for all hybrid indexes in the subsequent experiments in this paper.

Using the default merge strategy, we next measure the cost of the merge process. We used the *insert-only* workload in this experiment because it generates the highest merge demand. For all four hybrid indexes and all three key types, we recorded the absolute time for every triggered merge operation along with the static-stage index size at the time of the execution to measure the merge speed. Note that the size of the dynamic stage is always $\frac{1}{10}$ of that of the static stage at merge.

Figure 6 illustrates how the merge time changes with the size of the static stage of the indexes. In general, the time to perform a merge increases linearly with the size of the index. Such linear growth is inevitable because of the fundamental limitations of merging sorted arrays. But merging occurs less frequently as the index size increases because it takes longer to accumulate enough new entries to reach the merge ratio threshold again. As such, the amortized cost of merging remains constant over time. We also observe an interesting exception when running hybrid ART using mono-inc integer keys. As Figure 6d shows, the merge time (red line) is much lower than the other key types. This is because the hybrid ART does not store nodes at the same level contiguously in an array in the same manner as the other data structures (see Figure 2). Hence, the merge process for ART with mono-inc integers only needs to create and rebuild a few number of nodes to complete the merge, which is faster than readjusting the entire array.

## 6.4 Hybrid Indexes vs. Originals

Lastly, we compare the hybrid indexes to their corresponding original structures to show the trade-offs of adopting a hybrid approach. We evaluated each of these indexes using the same set of YCSB-based workloads described in Section 6.1 with all three key types. We conducted separate experiments using the data structures as both primary key (i.e., unique) and secondary key (i.e., non-unique) indexes. We present the primary key index evaluation here. Results for secondary key indexes can be found in Appendix E.

Figure 7 shows the throughput and memory consumption for hybrid indexes used as primary key indexes. The main takeaway is that all of the hybrid indexes provide comparable throughputs (faster in some workloads, slower in others) to their original structures while consuming 30–70% less memory. Hybrid-Compressed B+tree achieves up to 30% additional space saving but loses a sig-

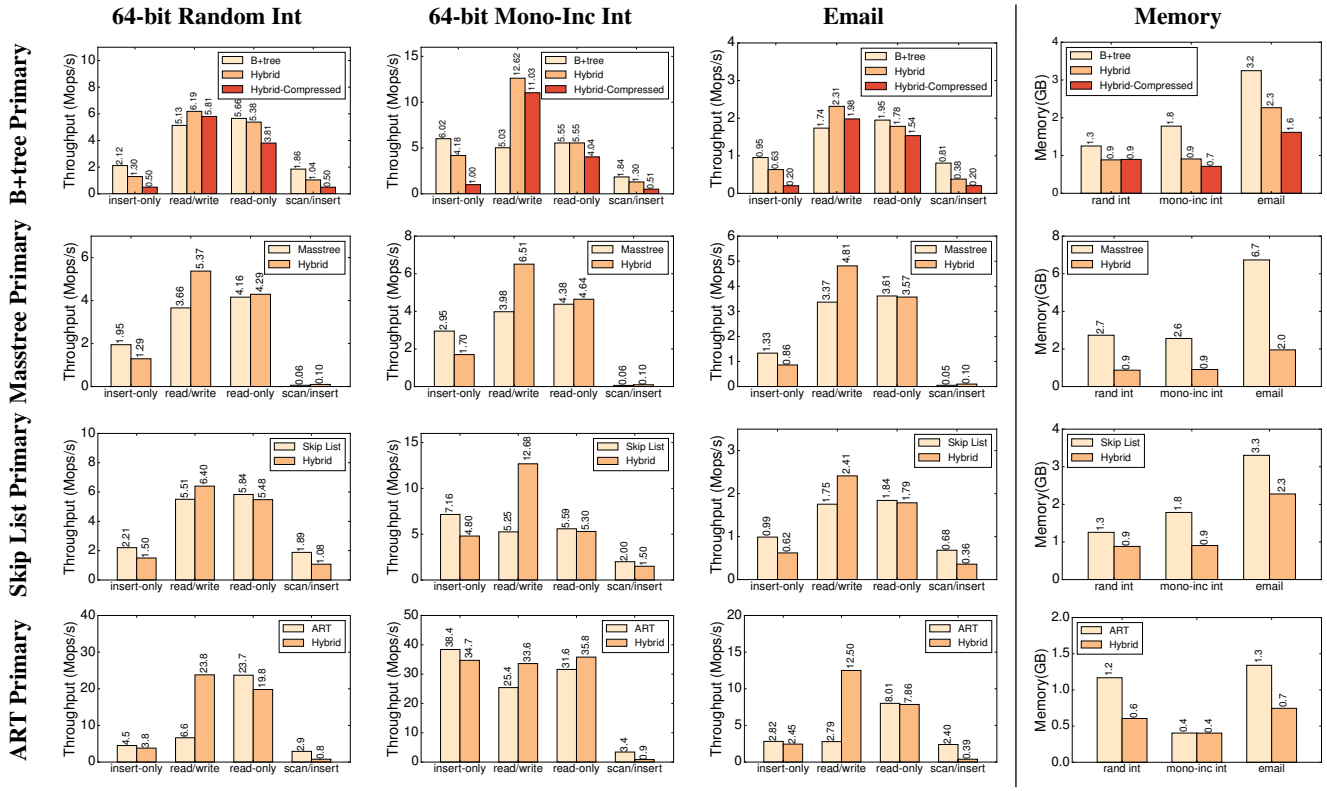



**Figure 7: Hybrid Index vs. Original (Primary Indexes)** – Throughput and memory measurements for the different YCSB workloads and key types when the data structures are used as primary key (i.e., unique) indexes. Note that the figures have different Y-axis scales.

nificant fraction of the throughput. This trade-off might only be acceptable for applications with tight space constraints.

**Insert-only:** One disadvantage of a hybrid index is that it requires periodic merges. As shown in Figure 7, all hybrid indexes are slower than their original structures under the insert-only workloads since they have the highest merge demand. The merging, however, is not the main reason for the performance degradation. Instead, it is because a hybrid index must check both the dynamic and static stages on every insert to verify that a key does not already exist in either location. Such key uniqueness check causes about a 30% insert throughput drop. For the Hybrid-Compressed B+tree, however, merge remains the primary overhead because of the decompression costs.

**Read/Write:** Despite having to check for uniqueness in two locations on inserts, the hybrid indexes' dual-stage architecture is better at handling skewed updates. The results for this workload show that all of the hybrid indexes outperform their original structures for all key types because they store newly updated entries in the smaller (and therefore more cache-friendly) dynamic stage.

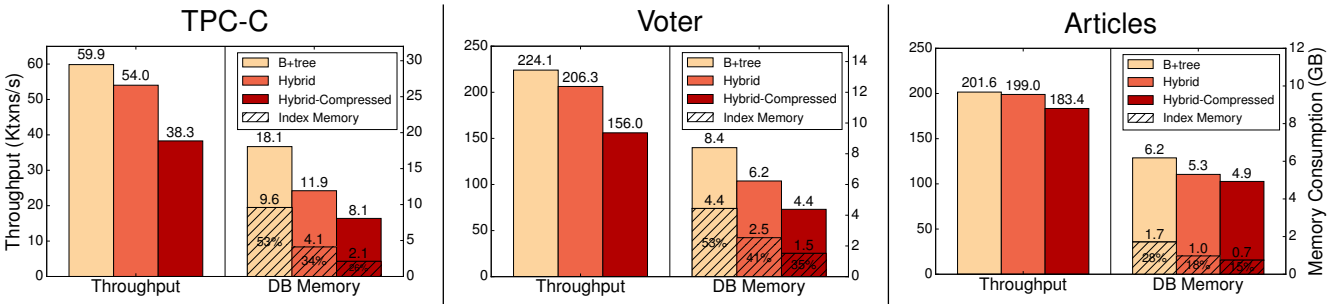
**Read-only:** We already compared the point-query performance for the dynamic and static stage data structures in Section 6.2. When we put these structures together in a single hybrid index, the overall point-query performance is only slightly slower than the static stage alone because a query may have to check both data structures. We, therefore, use a Bloom filter in front of the dynamic stage to ensure that most reads only search one of the stages. We evaluate the impact of this filter in Appendix D.

**Scan/Insert:** This last workload shows that the hybrid indexes have lower throughput for range queries. This is expected because their dual-stage design requires comparing keys from both the dynamic and static stages to determine the "next" entry when advancing the iterator. This comparison operation is particularly inefficient for hybrid ART because the data structure does not store the full keys in the leaf nodes. Therefore, performing full-key comparison requires fetching the keys from the records first. We also note that range query results are less optimized in Masstree because it does not provide the same iterator API that the other index implementations support. We do not believe, however, that there is anything inherent to Masstree's design that would make it significantly better or worse than the other data structures for this workload.

**Memory:** All of the hybrid indexes use significantly less memory than their original data structures. An interesting finding is that although the random and mono-inc integer key data sets are the same size, the B+tree and Skip List use more space to store the mono-inc integer keys. This is because the key insertion pattern of mono-inc integers produces B+tree nodes that are only 50% full (instead of 69% on average). The paged-deterministic Skip List that we used has a similar hierarchical structure as the B+tree and thus has a similar node occupancy. ART, however, uses less space to store mono-inc keys than the random keys because of prefix compression. This also reduces memory for the email keys as well.

# 7. FULL DBMS EVALUATION

This section shows the effects of integrating hybrid indexes into the in-memory H-Store OLTP DBMS [3, 33]. The latest version of H-Store uses B+tree as its default index data structure. We show that switching to hybrid B+tree reduces the DBMS's footprint in memory and enables it to process transactions for longer without having to use secondary storage. We omit the evaluation of the other hybrid data structures because they provide similar benefits.




**(a)** TPC-C Throughput + Memory

**(b)** Voter Throughput + Memory

**(c)** Articles Throughput + Memory

**Figure 8: In-Memory Workloads** – Throughput and memory measurements of the H-Store DBMS using the default B+tree, Hybrid, and Hybrid-Compressed B+tree when running workloads that fit entirely in memory. The system runs for 6 min in each benchmark trial.

## 7.1 H-Store Overview

H-Store is a distributed, row-oriented DBMS that supports serializable execution of transactions over main memory partitions [33]. It is optimized for the efficient execution of workloads that contain transactions invoked as pre-defined stored procedures. Client applications initiate transactions by sending the procedure name and input parameters to any node in the cluster. Each partition is assigned a single-threaded execution engine that is responsible for executing transactions and queries for that partition. A partition is protected by a single lock managed by its coordinator that is granted to transactions one-at-a-time based on the order of their arrival timestamp.

Anti-caching is a memory-oriented DBMS design that allows the system to manage databases that are larger than the amount of memory available without incurring the performance penalty of a disk-oriented system [23]. When the amount of in-memory data exceeds a user-defined threshold, the DBMS moves data to disk to free up space for new data. To do this, the system dynamically constructs blocks of the coldest tuples and writes them asynchronously to the anti-cache on disk. The DBMS maintains in-memory "tombstones" for each evicted tuple. When a running transaction attempts to access an evicted tuple through its tombstone, the DBMS aborts that transaction and fetches the tuple from the anti-cache without blocking other transactions. Once the data that the transaction needs is in memory, the system restarts the transaction.

## 7.2 Benchmarks

The experiments in this section use H-Store's built-in benchmarking framework to explore three workloads:

**TPC-C:** The TPC-C benchmark is the current industry standard for evaluating the performance of OLTP systems [51]. Its five stored procedures simulate a warehouse-centric order processing application. Approximately 88% of the transactions executed in TPC-C modify the database. We configure the database to contain eight warehouses and 100,000 items.

**Voter:** This benchmark simulates a phone-based election application. It is designed to saturate the DBMS with many short-lived transactions that all update a small number of records. There are a fixed number of contestants in the database. The workload is mostly transactions that update the total number of votes for a particular contestant. The DBMS records the number of votes made by each user based on their phone number; each user is only allowed to vote a fixed number of times.

**Articles:** This workload models an on-line news website where users submit content, such as text posts or links, and then other users post comments to them. All transactions involve a small number of tuples that are retrieved using either primary key or secondary indexes. We design and scale the benchmark so that the transactions coincide roughly with a week of Reddit's [11] traffic.

| | B+tree | Hybrid | Hybrid-Compressed |
|---|---|---|---|
| 50%-tile | 10 ms | 10 ms | 11 ms |
| 99%-tile | 50 ms | 52 ms | 83 ms |
| MAX | 115 ms | 611 ms | 1981 ms |

**Table 3: Latency Measurements** – Transaction latencies of H-Store using the default B+tree, Hybrid B+tree, and Hybrid-Compressed B+tree as indexes for the TPC-C workload (same experiment as in Figure 8a).

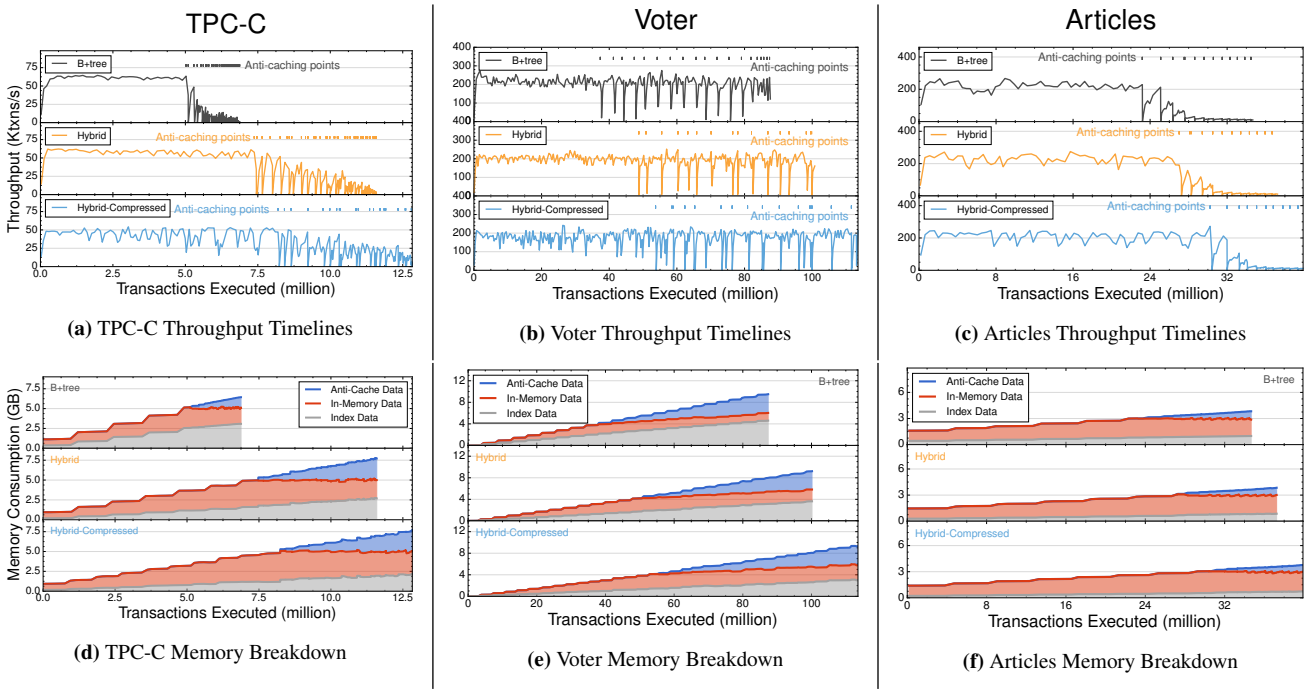
## 7.3 In-Memory Workloads

We first show that using hybrid indexes helps H-Store save a significant amount of memory. We ran the aforementioned three DBMS benchmarks on H-Store (anti-caching disabled) with three different index types: (1) B+tree, (2) Hybrid B+tree, and (3) Hybrid-Compressed B+tree. Each benchmark warms up for one minute after the initial load and then runs for five minutes on an 8-partition H-Store instance (one CPU core per partition). We deployed eight clients on the same machine using another eight cores on the other socket to exclude network factors. We compared throughput, index memory consumption, and total database memory consumption among the three index types. Figure 8 shows the results. The throughput results are the average throughputs during the execution time (warm-up period excluded); memory consumption is measured at the end of each benchmark. We repeated each benchmark three times and compute the average for the final results.

As shown in Figure 8, both Hybrid and Hybrid-Compressed B+tree have a smaller memory footprint than the original B+tree: 40–55% and 50–65%, respectively. The memory savings for the entire database depend on the relative size of indexes to database. Hybrid indexes favor workloads with small tuples, as in TPC-C and Voter, so the index memory savings translate into significant savings at the database level.

Hybrid B+tree incurs a 1–10% average throughput drop compared to the original, which is fairly small considering the memory savings. Hybrid-Compressed B+tree, however, sacrifices throughput more significantly to reap its additional memory savings. These two hybrid indexes offer a throughput-memory tradeoff that may depend on the application's requirements.

The results in Figure 8 are consistent with our findings in the microbenchmark evaluation (Section 6). The throughput drops associated with hybrid indexes are more noticeable in the TPC-C (10%) and Voter (8%) benchmarks because they are insert-intensive and contain a large fraction of primary indexes. Referring to the *insert-*




**(a)** TPC-C Throughput Timelines

**(b)** Voter Throughput Timelines

**(c)** Articles Throughput Timelines

**(d)** TPC-C Memory Breakdown

**(e)** Voter Memory Breakdown

**(f)** Articles Memory Breakdown

**Figure 9: Larger-than-Memory Workloads** – Throughput and memory measurements of the H-Store DBMS using B+tree, Hybrid, and Hybrid-Compressed B+tree as index structures when running workloads that are larger than the amount of memory available to the system. H-Store uses its anti-caching component to evict cold data from memory out to disk. The system runs 12 minutes in each benchmark trial.

*only* workloads in Figure 7, we see that hybrid indexes are slower when used as primary indexes because of the key-uniqueness check. The Articles benchmark, however, is more read-intensive. Since hybrid indexes provide comparable or better read throughput (see Figure 7), the throughput drop in Figure 8c is small (1%).

Table 3 lists the 50%-tile, 99%-tile, and MAX latency numbers for the TPC-C benchmark. Hybrid indexes have little effect on 50%-tile and 99%-tile latencies. For example, the difference in 99% latency between Hybrid B+tree and the original is almost negligible. The MAX latencies, however, increase when switching to hybrid indexes because our current merge algorithm is blocking. But the infrequency of merge means that the latency penalty only shows up when looking at MAX.

## 7.4 Larger-than-Memory Workloads

The previous section shows the savings from using hybrid indexes when the entire database fits in memory. Here, we show that hybrid indexes can further help H-Store with anti-caching enabled expand its capacity when the size of the database goes beyond physical memory. When both memory and disk are used, the memory saved by hybrid indexes allows the database to keep more hot tuples in memory. The database thus can sustain a higher throughput because fewer queries must retrieve tuples from disk.

We ran TPC-C, Voter, and Articles on H-Store with anti-caching enabled for all three index configurations: B+tree, Hybrid B+tree, and Hybrid-Compressed B+tree. Each benchmark executes for 12 minutes after the initial load. We used the same client-server configurations as in Section 7.3. We set the anti-caching eviction threshold to be 5 GB for TPC-C and Voter, 3 GB for Articles so that the DBMS starts anti-caching in the middle of the execution. The system's eviction manager periodically checks whether the total amount of memory used by the DBMS is above this threshold. If it is, H-Store selects the coldest data to evict to disk. Figure 9 shows the experiment results; note that we use the **total number of transactions executed** on the x-axis rather than time.

Using hybrid indexes, H-Store with anti-caching executes more transactions than the original B+tree index during the same 12-minute run. We note that the B+tree and Hybrid B+tree configurations cannot execute the Voter benchmark for the entire 12 minutes because the DBMS runs out of memory to hold the indexes as only the database tuples can be paged to disk.

Two features contribute to H-Store's improved capacity when using hybrid indexes. First, with the same anti-caching threshold, hybrid indexes consume less memory, allowing the database to run longer before the first anti-caching eviction occurs. Second, even during periods of anti-caching activity, H-Store with hybrid indexes sustains higher throughput because the saved index space allows more tuples to remain in memory.

H-Store's throughput when using anti-caching depends largely on whether the workload reads evicted tuples [23]. TPC-C is an insert-heavy workload that mostly reads new data. Thus, TPC-C's throughput decreases relatively slowly as the tuples are evicted to disk. Voter never reads evicted data, so the throughput remains constant. Articles, however, is relatively read-intensive and occasionally queries cold data. These reads impact throughput during anti-caching, especially at the end of the run when a significant number of tuples have been evicted. The throughput fluctuations for hybrid indexes (especially Hybrid-Compressed indexes) before anti-caching are due to index merging. After anti-caching starts, the large throughput fluctuations are because of the anti-caching evictions since the current version of anti-caching is a blocking process; all transactions are blocked until eviction completes.

# 8. RELATED WORK

Previous research explored unifying multiple underlying physical data structures, each with different optimization focuses, to construct a single logical entity. Some of the first examples include log-structured engines such as log-structured merge (LSM) trees [44], LevelDB [4] and LHAM [43]. Our hybrid index approach differs

from these techniques in several ways. First, log-structured engines are storage management systems that leverage the storage hierarchy while a hybrid index is an index data structure that resides only in memory. Such difference greatly influences a number of design decisions. For example, unlike LSM-trees, hybrid indexes avoid having too many stages/levels (unless the workload is extremely skewed) because the additional stages cause the worst case read latency to increase proportionally to the number of stages. In addition, log-structured engines focus on speeding up writes while hybrid indexes target at saving memory space.

SILT is a flash-based key-value store that achieves high performance with a small memory footprint by using a multi-level storage hierarchy with different data structures [39]. The first level is a log-structured store that supports fast writes. The second level is a transitional hash table to perform buffering. The final level is a compressed trie structure. Hybrid indexes borrow from this design, but unlike SILT, a hybrid index does not use a log-structured storage tier because maximizing the number of sequential writes is not a high priority for in-memory databases. Hybrid indexes also avoid SILT's heavyweight compression because of the large performance overhead. Similar systems include Anvil, a modular framework for database backends to allow flexible combinations of the underlying key-value stores to maximize their benefits [40].

The Dynamic-to-Static Rules are inspired by work from Bentley and Saxe [18]. In their paper, they propose general methods for converting static structures to dynamic structures; their goal is to provide a systematic method for designing new, performance-optimized dynamic data structures. In this paper, we use a different starting point, a dynamic data structure, and propose rules for creating a static version; furthermore, our focus is on creating space-optimized instead of performance-optimized variants.

Several DBMSs use compressed indexes to reduce the amount of data that is read from disk during query execution. There has been considerable work on space-efficient indexes for OLAP workloads to improve the performance of long running queries that access large segments of the database [20, 53]. SQL Server's columnar indexes use a combination of dictionary-based, value-based, and run-length encoding to compress the column store indexes [35]. MySQL's InnoDB storage engine has the ability to compress B-tree pages when they are written to disk [8]. To amortize compression and decompression overhead, InnoDB keeps a modification log within each B-tree page to buffer incoming changes to the page. This approach differs from hybrid indexes, which focus on structural reduction rather than data compression. Because hybrid indexes target in-memory databases and their concomitant performance objectives, data compression is prohibitive in most cases.

Other in-memory databases save space by focusing on the tuple stores rather than the index structures. One example is SAP's HANA hybrid DBMS [28, 47]. In HANA, all new data is first inserted into a row-major store engine that is optimized for OLTP workloads. Over time, the system migrates tuples to dictionary-compressed, in-memory columnar store that is optimized for OLAP queries. This approach is also used in HyPer [29]. Hybrid indexes take a similar approach to migrate cold data from the write-optimized index to the compact, read-only index. Both these techniques are orthogonal to hybrid indexes. A DBMS can use hybrid indexes while still moving data out to these compressed data stores.

Other work seeks to reduce the database's storage footprint by exploiting the access patterns of OLTP workloads to evict cold tuples from memory. These approaches differ in how they determine what to evict and the mechanism they use to move data. The anti-caching architecture in H-Store uses an LRU to track how often tuples are accessed and then migrates cold data to an auxiliary, on-disk data store [23]. Although the tuple data is removed from memory, the DBMS still has to keep all of the index keys in-memory. A similar approach was proposed for VoltDB (the commercial implementation of H-Store) where the database relies on the OS's virtual memory mechanism to move cold pages out to disk [48]. The Siberia Project for Microsoft's Hekaton categorizes hot/cold tuples based on sampling their access history [49] and can also migrate data out to an on-disk data store [27]. Hekaton still uses a disk-backed index, so cold pages are swapped out to disk as needed using SQL Server's buffer pool manager and the remaining in-memory index data is not compressed. Hybrid indexes do not rely on any tracking information to guide the merging process since it may not be available in every DBMS. It is future work to determine whether such access history may further improve hybrid indexes' performance.

Dynamic materialized views materialize only a selective subset of tuples in the view based on tuple access frequencies to save space and maintenance costs [55]. Similarly, database cracking constructs self-organizing, discriminative indexes according to the data access patterns [31]. Hybrid indexes leverage the same workload adaptivity by maintaining fast access paths for the newly inserted/updated entries to save memory and improve performance.

# 9. FUTURE WORK

Our work can be extended in several directions. First, developing space-efficient non-blocking merge algorithms for hybrid indexes can further satisfy the needs for tail-latency-sensitive applications. Second, with the non-blocking merge algorithms, it would be worthwhile attempting to make the entire hybrid index concurrent so that it can be applied to a concurrent main-memory DBMS. Third, the hybrid index architecture opens up the research opportunity to apply highly-compressed static data structures, including succinct data structures, to database indexes.

# 10. CONCLUSION

A hybrid index uses the dual-stage architecture to combine two data structures with different optimization emphasis to create a memory-efficient and high-performance order-preserving index structure. It uses a fast dynamic data structure as a write buffer in its dynamic stage to provide fast inserts and accesses to the recently added entries. For the second stage, it uses a compact, read-optimized data structure to serve reads for colder data. This design captures the characteristics of OLTP workloads, where newly inserted entries are likely to be accessed/modified frequently in the near future while older entries are mostly accessed through occasional look-ups. We provide a set of guidelines, called the Dual-Stage Transformation (DST), to help index designers convert any order-preserving index structure to a hybrid index. Experimental results show that hybrid indexes created by applying the DST guidelines provide throughput comparable to the original index structures while using significantly less memory. We hope that the various techniques discussed in the paper can form an important step towards a hybrid approach for index design in main-memory databases.

# ACKNOWLEDGMENTS

This work was supported by funding from U.S. National Science Foundation under awards IIS-1409802, CNS-1345305, and CCF-1535821, as well as Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

**For questions or comments about this paper, please call the CMU Database Hotline at** `+1-844-88-CMUDB`**.**

# 11. REFERENCES


[1] ALTIBASE Index Documentation. http://aid.altibase.com/display/migfromora/Index.

[2] Cache-Optimized Concurrent Skip list. http://sourceforge.net/projects/skiplist/files/Templatized%20C%2B%2B%20Version/.

[3] H-Store. http://hstore.cs.brown.edu.

[4] Leveldb. http://www.leveldb.org.

[5] LZ4. https://code.google.com/p/lz4/.

[6] MemSQL Documentation. http://docs.memsql.com/latest/concepts/indexes/.

[7] MySQL Memory Storage Engine. http://dev.mysql.com/doc/refman/5.7/en/memory-storage-engine.html.

[8] Mysql v5.5 – how compression works for innodb tables. http://dev.mysql.com/doc/refman/5.5/en/innodb-compression-internals.html.

[9] Overview of TimesTen Index Types. https://docs.oracle.com/cd/E21901_01/timesten.1122/e21633/comp.htm#TTOPR380.

[10] Performance Application Programming Interface. http://icl.cs.utk.edu/papi/index.html.

[11] Reddit. http://www.reddit.com.

[12] Redis Index. http://redis.io/topics/indexes.

[13] SAP Hana Indexes. http://saphanawiki.com/2015/09/2160391-faq-sap-hana-indexes/.

[14] Snappy. https://github.com/google/snappy.

[15] SQLite Documentation. https://www.sqlite.org/docs.html.

[16] VoltDB Blog. https://voltdb.com/blog/best-practices-index-optimization-voltdb.

[17] WiredTiger. http://wiredtiger.com.

[18] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[19] T. Bingmann. STX B+ tree C++ template classes. http://panthema.net/2007/stx-btree/.

[20] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. SIGMOD, pages 355–366, 1998.

[21] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[23] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *VLDB*, 6(14):1942–1953, Sept. 2013.

[24] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[25] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, pages 1–12, 2013.

[26] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, pages 195–198, 2011.

[27] A. Eldawy, J. J. Levandoski, and P. Larson. Trekking through siberia: Managing cold data in a memory-optimized database. *PVLDB*, 7(11):931–942, 2014.

[28] 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.

[29] F. Funke, A. Kemper, and T. Neumann. Compacting transactional data in hybrid oltp&olap databases. *VLDB*, 5(11):1424–1435, July 2012.

[30] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[31] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[32] G. J. Jacobson. Succinct static data structures. 1980.

[33] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. 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. *VLDB*, 1(2):1496–1499, 2008.

[34] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core cpus. *VLDB*, 5(1):61–72, Sept. 2011.

[35] P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. Sql server column store indexes. SIGMOD, pages 1177–1184, 2011.

[36] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *Computer*, 36(12).

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

[38] J. Levandoski, D. Lomet, S. Sengupta, A. Birka, and C. Diaconu. Indexing on modern hardware: Hekaton and beyond. SIGMOD, pages 717–720, 2014.

[39] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. SOSP '11, pages 1–13, 2011.

[40] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with anvil. SOSP, pages 147–160, 2009.

[41] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. EuroSys, pages 183–196, 2012.

[42] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. the third annual ACM-SIAM symposium on Discrete algorithms, 1992.

[43] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The lham log-structured history data access method. *The VLDB Journal*, 8(3-4):199–221, Feb. 2000.

[44] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[45] A. Prout. The story behind memsql's skiplist indexes. http://blog.memsql.com/the-story-behind-memsqls-skip list-indexes/.

[46] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[47] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: The end of a column store myth. SIGMOD, pages 731–742, 2012.

[48] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory oltp databases. DaMoN, 2013.

[49] R. Stoica, J. J. Levandoski, and P.-A. Larson. Identifying hot and cold data in main-memory databases. ICDE, pages 26–37, 2013.

[50] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[51] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, June 2007.

[52] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, 2013.

[53] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, Sept. 2000.

[54] A. C.-C. Yao. On random 2-3 trees. *Acta Informatica*, 9:159–170, 1978.

[55] J. Zhou, P.-A. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. ICDE, pages 526–535, 2007.

| | Year | Supported Index Types |
|---|---|---|
| ALTIBASE [1] | 1999 | **B-tree/B+tree**, R-tree |
| H-Store [3] | 2007 | **B+tree**, hash index |
| HyPer [29] | 2010 | **Adaptive Radix Tree**, hash index |
| MSFT Hekaton [38] | 2011 | **Bw-tree**, hash index |
| MySQL (MEMORY) [7] | 2005 | **B-tree**, hash index |
| MemSQL [6] | 2012 | **skip list**, hash index |
| Redis [12] | 2009 | linked list, hash, skip list |
| SAP HANA [13] | 2010 | **B+tree/CPB+tree** |
| Silo [52] | 2013 | **Masstree** |
| SQLite [15] | 2000 | **B-tree**, R*-tree |
| TimesTen [9] | 1995 | **B-tree**, T-tree, hash index, bitmap |
| VoltDB [16] | 2008 | **Red-Black Tree**, hash index |

**Table 4: Index Types** – The different type of index data structures supported by major commercial and academic in-memory OLTP DBMSs. The year corresponds to when the system was released or initially developed. The default index type for each DBMS is listed in **bold**.

# APPENDIX

## A. INDEX TYPES

In order to better understand how modern DBMSs use indexes, we examined several of the major commercial and academic in-memory OLTP DBMSs that were developed in the last two decades. Our listing in Table 4 shows the index types (i.e., data structures) supported by 12 DBMSs and the year that they were released. We also include which index type is used as the default when the user does not specify any hints to the DBMS; that is, the data structure that the DBMS uses when application invokes the `CREATE INDEX` command. Our survey shows that the order-preserving data structures are the most common, with the B+tree as the most popular default index type supported in these systems. This is notable since B+trees were originally designed for disk-oriented DBMSs in the 1970s [21]. We also note that In addition, most of the listed DBMSs support hash indexes, but none of them use this as their default option. Four of the systems do not even expose a hash index to the application developer, although they do use on internally for operator execution (e.g., joins). We attribute this to the fact that hash indexes cannot support range queries and thus may provide unexpected performance for users.

## B. DETAILED MERGE ALGORITHMS

This section extends Section 5.1 by providing more details about the merge algorithms for the different data structures.

As we mentioned in Section 5.1, the algorithm for merging B+tree to Compact B+tree is straightforward. The first step is to merge the new items from the dynamic stage to the leaf-node array of the Compact B+tree in the static stage, using the in-place merge sort algorithm. Then, based on the merged leaf-node array, the algorithm rebuilds the internal nodes level by level bottom up. Skip List merging uses a similar algorithm.

For Masstree and ART, merging uses recursive algorithms. Figure 10 shows the pseudo code for merging Masstree to Compact Masstree. The algorithm is a combination of merging sorted arrays and merging tries. We define three merge tasks that serve as building blocks for the merge process: merge two trie nodes, insert an item into a trie node, and create a trie node to hold two items. Note that the "==" sign between items in the pseudo code means that they have equivalent keyslices.

The initial task is to merge the root nodes of the two tries, as shown in the *merge_nodes(root_m, root_n)* function in Figure 10. Merging any two trie nodes, including the root nodes, involves merging the sorted arrays of keys within the nodes. Conceptually, the algorithm proceeds as in a typical merge sort, except that it recursively (depth-first) creates new tasks when the child nodes (or suffixes) require further merging. The merge process ends once the root node merge completes.

```
merge_nodes(node m, n, parent):
  //merge the sorted arrays together
  merge_arrays(m, n)
  link n to parent

merge_arrays(node m, n):
  //2 running cursors: x for m, y for n
  for item x in m and item y in n:
    if x == y:                    //equal keyslice
      recursively invoke:
      case 1: both x and y have child:
        merge_nodes(x.child, y.child, n)
      case 2: x has child, y has suffix:
        add_item(y.suffix, x.child, n)
      case 3: y has child, x has suffix:
        add_item(x.suffix, y.child, n)
      case 4: x.suffix != y.suffix:
        create_node(x.suffix, y.suffix, n)
    else
      move min(x, y) to new position in n

add_item(item x, node n, parent):
  //insert item x to the sorted arrays in n
  insert_one(x, n)
  link n to parent

insert_one(item x, node n):
  if x == (any item y in n):
    recursively invoke:
    case 1: y has child:
      add_item(x.suffix, y.child, n)
    case 2: y has suffix:
      create_node(x.suffix, y.suffix, n)
  else
    insert x to appropriate position in n

create_node(item x, y, node parent):
  //create a new node to hold x and y
  n = new_node(x, y)
  if x == y:
    create_node(x.suffix, y.suffix, n)
  link n to parent
```

**Figure 10: Algorithm of merging Masstree to Compact Masstree** – A recursive algorithm that combines trie traversal and merge sort.

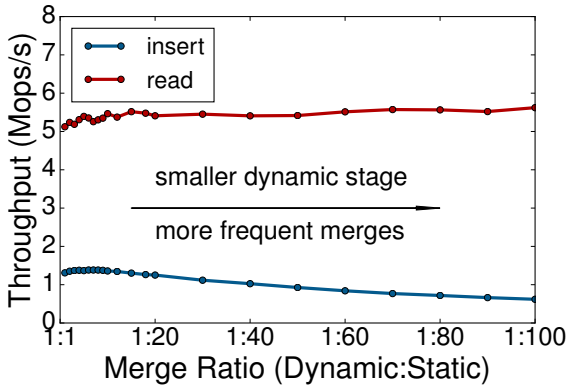
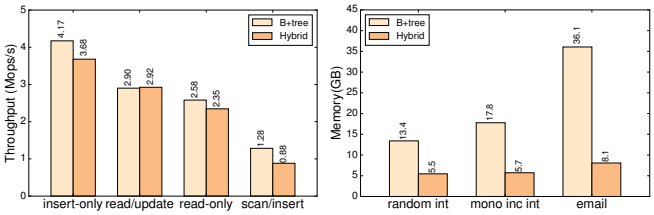



**Figure 11: Merge Ratio** – A sensitivity analysis of hybrid index's ratio-based merge strategy. The index used in this analysis is Hybrid B+tree.

Merging ART to Compact ART adopts a slightly more complicated recursive algorithm. Instead of checking the key suffixes directly within the node (as in Masstree), ART has to load the full keys from the records and extract the suffixes based on the current trie depth. The two optimizations (lazy expansion and pass compression) in ART [37] further complicates the algorithm because child nodes of the same parent can be at different levels.

## C. SENSITIVITY ANALYSIS OF RATIO-BASED MERGE STRATEGY

This section extends Section 6.3 by providing a sensitivity analysis of the ratio-based merge strategy. To determine a good default merge ratio that balances read and write throughput, we use the *insert-only* workload followed by the *read-only* workload with 64-bit integer keys to test ratios ranging from 1 to 100. For each ratio setting, we adjust the number of entries inserted so that the dynamic stage is about 50% "full" right before the read-only workload starts. We measure the average throughput of the hybrid index for the insert-only and read-only phases separately for each ratio. We only show the results for hybrid B+tree because they are sufficient




**(a)** Throughput **(b)** Memory

**Figure 13: Hybrid Index vs. Original (Secondary Indexes)** – Throughput and memory measurements for different YCSB workloads using 64-bit random integer keys when the data structures are used as secondary (i.e., non-unique) indexes. The data set contains 10 values for each unique key.

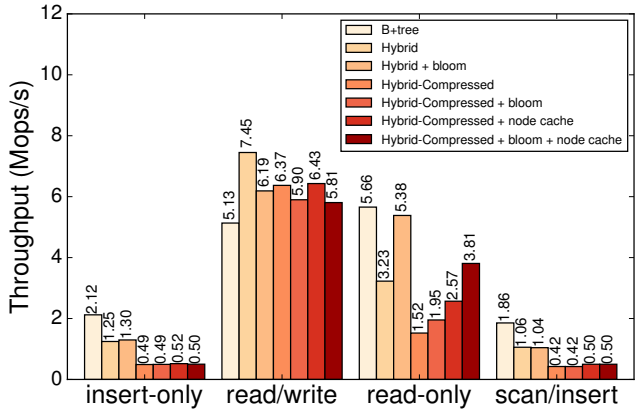



**Figure 12: Auxiliary Structures** – This figure is an extended version of the figure in the (B+tree Primary, 64-bit random int) cell of Figure 7 that shows the effects of the Bloom filter and the node cache in the hybrid index architecture.

to demonstrate the relationship between read/write throughput and merge ratio.

The results in Figure 11 show that a larger merge ratio leads to slightly higher read throughput and lower write throughput. A larger ratio keeps the dynamic stage smaller, thereby speeding up traversals in the dynamic stage. But it also triggers merges more frequently, which reduces the write throughput. As the merge ratio increases, the write throughput decreases more quickly than the read throughput increases. Since OLTP workloads are generally write-intensive, they benefit more from a relatively small ratio. Based on this finding, we use 10 as the merge ratio for all indexes in the subsequent experiments in this paper.

# D. AUXILIARY STRUCTURES

This section is an extension of Section 6.4, where we show the effects of two auxiliary structures presented in the hybrid index architecture: Bloom filter (see Figure 1) and node cache (see Figure 2). We extend the experiment shown in the (B+tree Primary, 64-bit random int) cell of Figure 7 by making the inclusion of Bloom filter or node cache controlled variables to show separately their effects on performance.

Figure 12 presents the results. We first focus on the read-only workload. For all variants of the hybrid index, the throughput improves significantly when adding the Bloom filter; similarly, adding a node cache also improves throughput over the same index variant without a node cache. In addition, Bloom filter and node cache improve read performance without noticeable overhead for other non-read-only workloads.

# E. SECONDARY INDEXES EVALUATION

This section is an extension of Section 6.4, where we provide the experiment results for hybrid indexes used as secondary indexes. The experiment setup is described in Section 6.4. We insert ten values (instead of one, as in primary indexes) for each unique key. Because we implement multi-value support for all indexes in the same way (described in Section 4.2), we only show the result for hybrid B+tree in the 64-bit random integer key case as a representative to demonstrate the differences between using hybrid indexes as primary and secondary indexes.

As shown in Figure 13, the secondary index results are consistent with the primary index findings with several exceptions. First, the insert throughput gap between the original and hybrid B+tree shrinks because secondary indexes do not require a key-uniqueness check for an insert, which is the main reason for the slowdown in the primary index case. Second, hybrid B+tree loses its large throughput advantage in the read/write (i.e., update-heavy) workload case because it handles these value updates in-place rather than inserting new entries into the dynamic stage (as for primary indexes). In-place updates prevent the same key from appearing in both stages with different sets of values, which would require a hybrid index to search both stages to construct a complete value list for a key. Third, the memory savings of hybrid B+tree are more significant in the secondary index case because the original B+tree stores duplicate keys while Compact B+tree does not.