Received September 23, 2021, accepted October 21, 2021, date of publication October 27, 2021, date of current version November 3, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3123379

A Performance Evaluation of DRAM Access for In-Memory Databases

ZHANG QIAN, (Student Member, IEEE), JIANHAO WEI, YIWEN XIANG, JR., AND CHUQIAO XIAO
Software Engineering Institute, East China Normal University, Shanghai 200241, China

Corresponding author: Zhang Qian (52184501012@stu.ecnu.edu.cn)

This work was supported in part by the National Key Research and Development Project of China under Grant 2019YFB2102600, and in part by the National Natural Science Foundation of China under Grant 61572194 and Grant 61672233.

ABSTRACT The latest CPUs(computer cpu processors) employ multiple cores, massively superscalar pipelines, out-of-order execution of tons of instructions, and advanced SIMD capabilities, which can hide the memory access latency. And most of recent memory-oriented data structures have already benefit from these features. However, due to the complexity of data organization, these CPUs do not always work well in memory resident database systems (MMDBs), particularly regarding storing data in dynamic random-access memory (DRAM). This article studies memory-efficient data structures by analyzing the run time, access latency, cache misses, instructions per cycle (IPC), and DRAM reads (bytes). Then, we design and implement two data organization schemas in the main memory database: dispersing data block organization and clustering data block organization. Using algorithmic engineering and careful attention to internal parallelism, cache alignment can hide the memory access latency. However, we find that these data structures work well in some cases, though they have been eclipsed in the face of complex access paths. To determine the reasons, we study the impact of database techniques on memory access latency, such as data partitioning, storage models, and by processing algorithms. With the specific main memory database system, we estimate the performance of each data organization schema based on DRAM DDR4 and the latest Intel Haswell microarchitecture. In conclusion, this work will make DRAM access applicable in real-world situations by implementing the schema to systems, such as in-memory databases.

INDEX TERMS DRAM access, in-memory databases, data structure, MMDBs, database techniques.

I. INTRODUCTION

Large -scale highly interactive online applications and batch processing offline applications require either low latency or a high throughput for processing huge transactional and analytical query workloads. With the development of memory hardware in recent years, to keep data access very high speed, systems designed for this type of big data application typically keep all of the data in the main memory. While emerging byte-addressable nonvolatile memories (NVMs) enable persistent data, dynamic random access memory (DRAM) resident data allow for faster access to hot data [1]. Many of today's applications have designed databases that reside in random access memory [2]–[9]. However, unlike traditional disk-based data management systems, disk I/O for data page accesses and buffer pools are omitted in main memory resident data applications. Consequently for many database operation performances, accessing DRAM main memory associated with modern processors is a challenge.

Memory hierarchy design can reduce the gap between processor memory requests and the latency of DRAM access. Data are always copied back and forth in the hierarchy (caches or in the main memory) in fixed-sized logical cache lines, such as 64 bytes. Optimized caching data is very beneficial to processors due to fewer cache misses and fewer stalls, which usually causes the average memory access latency. Moreover, the mainstream CPUs have adopted wider SIMD registers and more advanced instructions, such as Intel Haswell, kylake microarchitecture, support 256-bit SIMD instructions and separate 512-bit SIMD instructions, thus making more data parallelism performance possible. The performance of the code is limited by instruction execution

The associate editor coordinating the review of this manuscript and approving it for publication was Gianmaria Silvello.

and yet the data locality does play a substantial role in modern multicore architectures, which has been discussed in references [10]–[12]. If all values reside in one cache line, then one SIMD gather instruction is sufficient. However, each value is located in a different cache line, which will require time to gather. Hence, assuming all the required detector values are contiguous in the main memory and aligned in cache lines, data parallelism can be achieved by using SIMD instructions effectively.

Fortunately, many memory operations that leverage cache-intensive data structures (e.g., lookup, scan) have abundant memory access performance that can be exploited to optimize the data organization of the main memory systems. We gather diverse memory-efficient data structures, design the access patterns, and observe their memory access performance on target database system resident data in DRAM. We have been amazed that the advantages of the data structures have nearly disappeared as the complexity of the access paths increases. The memory access latency does not depend too much on the data block organization using memory-efficient data structures; however, it depends more so on the internal layout and access methods.

Although Garcia-Molina and Salem [13] say that sequential access is not notably faster than random access on memory resident data, the contiguousness and data locality of read data is a dominant factor that limits the overall execution performance of cache consciousness in main memory systems [14]. The data locality is a necessity in pure main memory persistent data management. If data is accessed in a sequence, it can be brought in multiple values at a time by using cache lines, and hardware or software prefetchers can predict the pattern to overlap memory accesses [15]. Hence, efficiently placing the data records inside the blocks is critical to system performance. We implement the decomposition storage model (DSM) [16], N-ary storage model (NSM) [17], and Partition Attributes Across (PAX) [18] in this evaluation, thus aiming to analyze modern OLTP-style and OLAP-style analytical workloads.

An access method enables the main memory database accesses the data records stored in the memory heap. In the vectorization-style processing algorithm, the data transfer between operators in a query pipeline uses batches of values having the same data type and typically, stored consecutively [19], [20]. The advantage of this approach while scanning is that it allows us to utilize SIMD instructions where it is most beneficial. In the materialization-style processing algorithm, the data tuples are often resident in CPU registers as long as possible to avoid copying the data. In the best case, a tuple is kept in registers while multiple operators are executed on it. This process, known as a pipeline, is cache friendly. Using the LLVM compiler framework to generate execution code can further improve performance, which has been implemented in various database systems [2], [3], [20]–[23]. However, it is difficult to comprehend and debug, since the generated code is quite low-level and the compile time also needs to be traded off. In addition to SQL statements, Linnea *et al.* [24] integrated lambda expressions into a relational memory database, which led to a higher data analytics performance for large datasets.

In main memory systems, using pointers can not only save considerable storage space, which is important for expensive DRAM, but can also simplify handling variable length fields [4], [13], [25], such as memory allocation and cache alignment. This fully benefits from the random access performance in the main memory. However, with the advent of big data workloads, this approach increasingly exhibits long-latency memory stalls within the complex relational data structure and data processing [15], [26]. Fortunately, recent proposals have examined software prefetching techniques for exploiting inter-lookup parallelism to hide the memory access latency [27]–[31]. In the program stream, software prefetching issues nonblocking loads for memory accesses, mainly by performing address calculations for future memory accesses by hand or automatically generating code leveraging novel compilers. Otherwise, the hardware prefetching mechanisms that do not require a core pipeline to execute additional instructions to compute and issue prefetches can avoid instruction overheads, inflexibility and limited latency tolerance of software prefetching, as described in References [32]–[34]. While these techniques work well for memory-latency bounds, they focus on long pointer chains and specific operations, such as hash table probs and tree traversals, which are only useful for a specific problem. Our evaluated memory access approaches differ in the faction of real-life workloads. In addition, we also observe that performance either improves or remains unchanged for the software prefetching compiler technique. Our TPC-C workload experiment offers the memory access latency gap between the physical pointers and logical pointers in a targeted application.

In summary, the goals of this work are as follows.

- Carefully choose five representative memory-efficient data structures, including Google BTree [35], Mass-Tree [36]–[38], Open BW-Tree [5], [39], [40], Hopscotch Hashing [41], [42], and Cuckoo Hashing [43], [44]. Then, we qualitatively and quantitatively compare these data structures using a unified framework PiBench.
- Design data organization for a specific in-memory database system using those memory-efficient data structures. This is because they are the latest main memory techniques, and trees and hash tables are most widely used in database systems for data organization. We correctly implement these techniques in a real-life memory resident application, PELOTON.
- Test a variety of workloads under the same hardware configurations to ensure the fairness of the experiments. Through the experimental results and by collecting metrics on DRAM, we present more detailed findings and insights for the different designs.

After introducing background on memory-efficient data structures in more detail in Section 2 and describing our design and implementation in Section 3, we present our evaluation methodology in Section 4. We then discuss the experimental configurations, as well as examine them in Section 5 using the YCSB workloads and the TPC-C workloads. In Section 6, We discuss our findings and insights obtained from this evaluation. We discuss related work in Section 7 and conclude this paper in Section 8.

II. BACKGROUND AND MOTIVATION

In this section, we survey representative memory-efficient data structures, including Google BTree [35], Mass-Tree [36]–[38], Open BW-Tree [39], [40], Hopscotch Hashing [41], [42], and Cuckoo Hashing [43], [44]. Then, we briefly discuss a performance comparison between these state-of-the-art in-memory data structures using the PiBench framework.

A. TREE STRUCTURES

B-Trees are widely known as data structures for secondary storage because they keep disk seeking to a minimum. For an in-memory data structure, the same property yields a performance boost by keeping cache-line misses to a minimum. Google BTree is an implementation of an ordered in-memory container based on a BTree data structure. It stores entry instances in an ordered structure, allowing easy insertion, removal, and iteration. Write operations are not safe for concurrent mutation by multiple go routines, but read operations are. Compared with standard C++ containers implemented using red-black trees, Google BTree, on average, makes use of fewer than one pointer per entry, leading to substantial memory savings.

As shown in Figure 1, the same node type is used for both internal and leaf nodes in the tree. The children array is only valid in internal nodes. Each entry of the array records the pointer of the corresponding child node holding the keys in a sorted order i.e., the n-th slot will point to the location of ValueType pairs with keys smaller than the n+1 slot. Therefore, if the key is an integral or floating-point type, using the linear search configuration might be the wise choice because array traversals (simple stride indirect) are friendly for software and hardware prefetching.

Mass-Tree is a fast key-value data structure designed for shared main memory multiprocessing (SMP) machines. Mass-Tree keeps all data in memory. Its main data structure is a trie-like concatenation of B+trees, each of which handles a fixed-length slice of a variable-length key. Query time is dominated by the total DRAM fetch time of successive nodes during tree descent. To reduce this cost, Mass-Tree uses a wide-fanout tree to reduce the tree depth, prefetches nodes from DRAM to overlap fetch latencies, and carefully lays out data in cache lines to reduce the amount of data needed per node. Mass-Tree achieves a high performance on multicore hardware using fine-grained locking and optimistic concurrency control. Mass-Tree readers and writers must cooperate to avoid confusion. The key communication channel between them is a per-node version counter that writers mark as "dirty" before creating intermediate states and then increment when done. Readers snapshot a node's version before accessing the node and then compare this snapshot to the version afterward. If the versions differ or are dirty, the reader may have observed an inconsistent intermediate state and must retry.

Since the tree is sorted by key, it will perform well on workloads with many keys that share long prefixes. The access time is dominated by the total DRAM fetch time of successive nodes during tree descent. As shown in Figure 1, Mass-Tree uses a wide-fanout tree to lower the tree depth, prefetches nodes from DRAM to overlap the access latencies, and carefully lays out data in cache lines to reduce the amount of data needed per node, which all effectively hide the memory latency. Suppose we want to scan a range key [k(i), k(i+j)). We first loop over the internal nodes and find the target nodes. Then, we traverse to the leftmost leaf node and start a left-to-right walk. Because of the versioned nodes, we have to thoroughly check the variables of each reach leaf.

BW-Tree is a lock-free index that provides a high throughput for transactional database workloads in an SQL server's Hekaton engine [5]. The high-level idea of the BW-Tree is that it avoids locks by using an indirection layer that maps logical identifiers to physical pointers for the tree's internal components. Threads then apply concurrent updates to a tree node by appending delta records to that node's modification log. This design provides two benefits. First, it avoids the coherent traffic of locks by decomposing every global state change into atomic steps; second, it incurs fewer cache invalidations on a multicore CPU because threads append delta records to make changes to the index instead of overwriting existing nodes. Wang *et al.* [40] designed and implemented an in-memory BW-Tree called the Open BW-Tree.

BW-Tree avoids directly editing tree nodes because it causes cache line invalidation. However, the mapping table serves as an indirection layer that maps logical node IDs to physical pointers. As shown in Figure 1, suppose a lookup key (k). We first loop search over the inner nodes to find the corresponding Node ID and then find the physical location by mapping the hash table for the ValueType. Otherwise, we must traverse the long delta chains for the valid version. The complicated delta chain traversal routines and mapping table all contribute to the Open BW-Tree higher instruction count and cache misses per operation.

B. HASHING STRUCTURES

Hopscotch hashing is an open addressing algorithm originally proposed by Herlihy, Shavit, and Tzafrir, which is known for fast performance and excellent cache locality. As shown in Figure 2, the main idea behind hopscotch hashing is that each bucket has a neighborhood of size H. The neighborhood of a bucket B is defined as the bucket itself and the (H-1) buckets following B contiguously in memory (H buckets total). This also means that at any bucket, multiple neighborhoods overlap (H to be exact). Hopscotch hashing guarantees that an entry will always be found in the neighborhood of

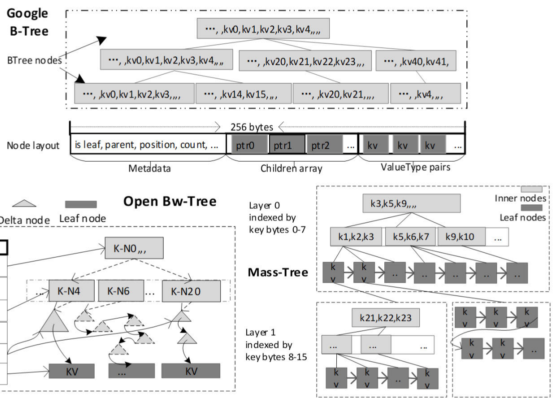


FIGURE 1. Architecture overview of Google BTree, Mass-Tree, and open BW-Tree.

its initial bucket. As a result, most H consecutive look-ups will be required in contiguous memory locations, which is extremely cache friendly. Moreover, the authors build upon this idea of the fixed size neighborhood using relative offsets to indicate where the next relevant entry is stored. Each bucket contains two integers: the first is the offset where the probe chain starts, and the second is the next entry in the probe chain. Lock-free hopscotch hashing [42] improves on both speed and cache locality. The progress guarantees of the original include a chimera of two concurrent hash tables. They employ relocation counters at each bucket to indicate when that bucket's neighborhood has experienced a bucket relocation. All operations read the relocation counter before and after to ensure that there is no concurrent operation consistency.

Cuckoo hashing is an open-addressed hash table design. Basic cuckoo hashing stores all key and value entries using a large array with no pointers or linked lists. If collisions occur, entries can be stored in one of two buckets in the array. However, they can be moved to the other location if the first is full. Second, each bucket has N "slots" for entries, i.e., N = 4 is a common configuration in practice. To insert a new key into the table, there may be at most hundreds of bucket reads and writes, which is a sequence during the whole insert process. While two prior approaches [45], [46] have implemented concurrent access, recent work [47] presents the design and implementation of the more cache friendly and memory efficiency cuckoo hash table. As shown in Figure 2, a cuckoo hash table can be viewed as an undirected graph called a cuckoo graph, which has a vertex for each bucket and an edge for each key in the table, connecting the two alternative buckets of the key. In the cuckoo graph, each alternative bucket of the keys in the current bucket are considered neighbors of that bucket. BFS (breadth-first search) scans all neighbors of a bucket to extend the cuckoo path. Before scanning one neighbor, the processor can load the next neighborhood in the cache, which will be accessed soon if no empty slot is found in the current neighbor.

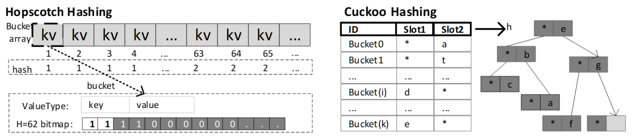


FIGURE 2. Architecture overview of hopscotch hashing and cuckoo hashing.

C. PERFORMANCE COMPARISON ON PiBench

PiBench [48] is a benchmark framework for an in-memory data structure, which can be used to benchmark volatile DRAM and gathers additional metrics specific to Intel Optane DC Persistent Memory. PiBench comprises a shared library and supports common operations by defining a set of common interfaces (e.g., insert, lookup, scan). Any data structures such as hash tables and trees can be implemented through a wrapper (libwrapper.so) and translate requests from PiBench's API.

PiBench allows us to fairly compare multiple in-memory data structures and rule out the impact of different benchmark implementations. We run experiments on a Linux (5.4), Ubuntu 18.04 server equipped with an Intel(R) Xeon(R) CPU E5–2620 v3, 64 GB of DRAM4 (4*16 GB DIMMs). The CPU has 6 cores (12 hyperthreads), 15 MB of L3 cache, and is clocked at 2.40 GHz. We implement[1] five in-memory data structures and another STL map using red–black trees. We run each data structure case with loading the 10 million records with 8-byte keys and 8-byte values. We then measure and report the performance during the run phase, in which 10 million operations are executed by a single thread. We set every time window to 100 ms of a list of operations and calculate the average throughput and latency time.

We configure the default node size for tree structures and bucket count for hash tables. For Hopscotchmap, each bucket has 64 neighbors. We set the Cuckoomap initialization bucket count to 4, the open BW-Tree leaf node size to 128, Mass-Tree node size to 15, and the Google BTree node size to 256 bytes, the same as the original paper's setup. As shown in Figure 3 and 4 (L3 cache misses: 10^6. L3 cache hit ratio: number of cache hits/num of lookups. Megabytes: number of bytes read from DRAM memory controllers, 10^6. Throughput: average operations of one second. IPC: instructions per cycle. Cycle: number of clock cycles, 10^9. 50% latency: 50% of sorted global latencies of sample operations, nanoseconds.), we will not present the scan operation performance of hash tables since they have not targeted it.

For the insert operation, all hash tables perform better than trees because of the low cache misses, instructions and cycles. Hopscotchmap has a higher throughput and lower latency than the other hash tables for both insert and lookup operations. This mainly benefits from low cache misses and IPCs benefit from the array. For the lookup operation, STLmap has the worst performance due to the high cache misses, cycle and

[1]https://github.com/gitzhqian/pibench-trees-hashing

lower IPC. This might be affected by the very deep search of the red–black tree. Open BW-Tree performs relatively poorly compared to other trees because it has a higher cycle count and last level cache misses per operation. The higher cache misses are generally caused by the mapping table. Mass-Tree has a higher insert operation throughput and lower access latency. This is because Mass-Tree creates a new space for the overflow instead of the copy-on-write. However, for the scan operation, Mass-Tree has high access latency and a low throughput. Although it has quite low cache misses and a high cache hit ratio, a large number of cycles and IPC degrade its performance. Mass-Tree with no prefetching has a lower throughput and higher latency compared with Mass-Tree by $\sim$20%. Google BTree outperforms Open BW-Tree and Mass-Tree in scan operations since it is a single version and requires fewer instructions.

In total, all cache misses pay the higher latency price of DRAM, which results in a lower throughput. Because of the complex access pattern resulting in a high instruction count and cycle count, multiple versions of insert optimization are less effective in the lookup or scan operation. In general, hardware and software prefetching can effectively hide the memory access latencies.

III. DESIGN AND IMPLEMENTATION

In this section, we discuss our design and implementation of data organization in the PELOTON main memory database system.

A. PARTITIONING AND DATA BLOCK

Partitioning is a ubiquitous operation for modern hardware query execution as a way to split large inputs into cache-conscious nonoverlapping subproblems. For example, in the main memory database, datasets can often be split into multiple small data blocks that are distributed among threads and now fit in the cache. Horizontal partitioning combined vertical partitioning approaches are widely used in modern database systems.

Dataset D consists of K attributes and contains C records. We design a data block layout using the decomposition storage model(DSM),D, N-ary storage model(NSM),N, and partition attributes across(PAX),X. If M$\leq$C, all possible layouts of a partition p are as follows:

$N = \{\{a_{11}, a_{12}, a_{13}, \ldots, a_{1k}\}, \{a_{21}, a_{22}, a_{23}, \ldots, a_{2k}k\}, \ldots, \{a_{m1}, a_{m2}, a_{m3}, \ldots, a_{mk}\}\}$.

$D = \{\{a_{1i}\}, \{a_{2i}\}, \{a_{3i}\}, \ldots, \{a_{mi}\}\}$.

$X = \{\{a_{11}, a_{12}, a_{13}, \ldots, a_{1k}\}, \{a_{21}, a_{22}, a_{23}, \ldots, a_{2k}\}, \ldots, \{a_{mj}, a_{mj}, a_{mj}, \ldots, a_{mj}\}\}$.

Since the size of each attribute can be a fixed-length or a variable-length, there could be different size data blocks of allocated memory. A fixed-size block is not a typical scenario, but it can be convenient. As there is no need to maintain non-contiguous storage space, fixed-size blocks are easier to manage. For example, a fixed-size block makes it possible to preallocate consecutive memory addresses, which avoids the scalability issue as the number of cores increases by frequently allocating small objects. In addition, fewer cache misses are incurred during memory accesses due to the spatial locality. The sequential block scan performance could fully benefit from these features. Variable-size blocks are more real-life application scenarios, and there are techniques to trade off the spatial locality and storage efficiency. Common implementations, such as fixed length placeholders, reference relocated attributes. Storing fixed-length (8-bytes) pointers in each block slot complicates the memory access because of the long pointer chain. Otherwise, there is a need to maintain a heap and carefully handle pointer-intensive operations with irregular access paths.

We use the naïve algorithm [49] to calculate the optimal attribute partitions based on the cost of each possible partition. The optimal partitions are the candidates that result in the fewest number of overall cache misses for the query workload. The cost of executing a query on a relation that uses a particular partition p is the sum of the cost to access all of the partitions of the derived attributes, as shown in Equation (1).

$$Cost(q, p) = \sum_{i=1}^{|p|} model\left(q, p_n\right) \tag{1}$$

In Equation (1), $|p|$ represents the number of all partitions of the derived attributes, p_i represents subpartition i in partition p, and $model(q,p_i)$ is the number of cache misses incurred for query q to access the derived attributes. The naïve algorithm calculates the cost of each possible partition and selects the partitions with the lowest overall cost for the set of queries Q. The method for calculating the lowest cost is shown in Equation (2).

$$Cost_{min} = min_{i=1\ldots|P|}(\sum_{j=1}^{|Q|} cost\left(Q_j, P_i\right)) \tag{2}$$

In Equation (2), $|P|$ is the number of possible partitions of the set of derived attributes,P_i is partition $i \in P$, $|Q|$ is the number of queries in the workload, and Q_j is query $j \in Q$.

B. DATA BLOCK ORGANIZATION

In this section, we will describe in detail how to organize the data blocks in the main memory database system. As in modern main-memory database systems, out-of-place schema is the popular approach used to handle insert/update operations. To update the existing tuple, the system creates a new empty slot instead of locking the older location, which is often favored in high contention environments because threads can make global progress. This out-of-place schema is not friendly for lookup operations and scan operations because there is a need for traversal data blocks to search for valid records. Otherwise, executions that access a record may be unsuccessful as a result of a higher overhead because the system follows the record's pointer chain to find the target chain [50].

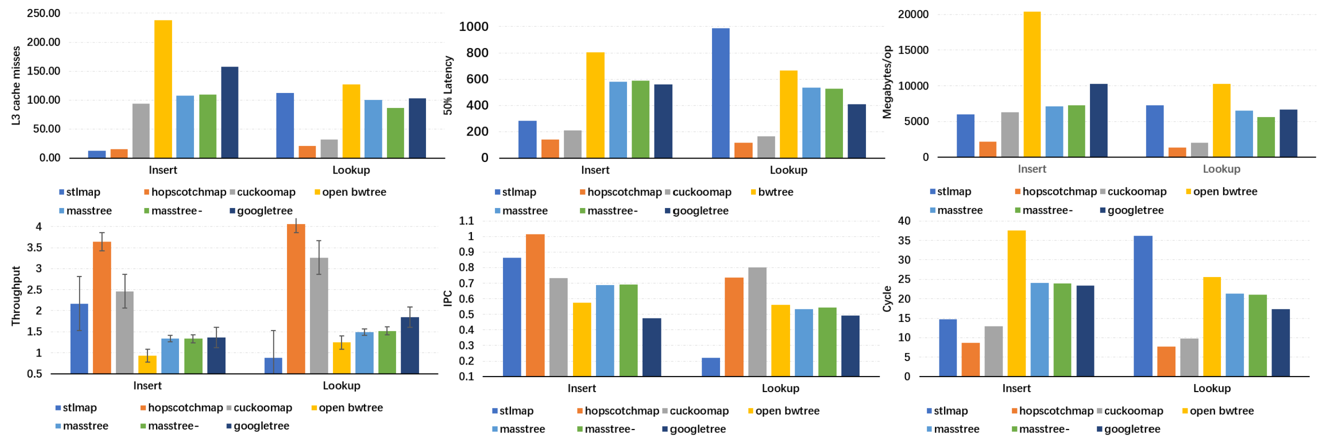


FIGURE 3. Under uniform distribution, a single thread executes insert-only operation or lookup-only operation.

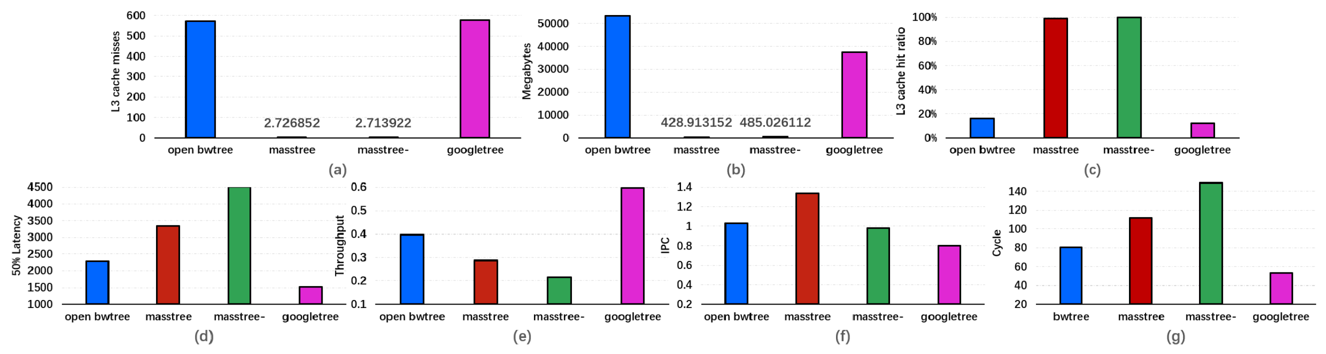


FIGURE 4. Under uniform distribution, a single thread executes scan operation.

1) DISPERSING DATA BLOCKS

As depicted in Figure 5, all blocks of the dataset are dispersed in a memory heap in this design. Both performance and available space considerations are important in allocating memory addresses. When insert and update operations make a request, the system tries to obtain an empty slot from the current active block. If the slot offset reaches the end of the block, a new fixed-size physical memory will be preallocated by requesting the OS. The new block should be mapped to the block organization. As an out-of-place update schema, data parallelism is allowed to leverage the multiple core processor by setting the number of active blocks. This optimization, however, is less effective in high contention workloads where the threads' throughput and latency performance are limited by getNextEmptySlot and updateBlockHeader of the block.

The main idea of using an indirect pointer is that the database system uses a fixed identifier that leverages expensive DRAM memory and does not change for each block location in its block organization. The random lookup operation and scan operation would not succeed because of long pointer chains. The pseudocode for the scan operation is shown in Algorithm 1. For each scan, the database system loops the block array held by the relational dataset. After fetching the block ID, hashing the block ID is needed to obtain the block physical location. If the attribute is a variable length, there is a visit to the referenced pointer. There are more than 4 pointings for each lookup search. In our experiments, tree iteration and hash prefetching can reduce the logical pointer penalty, which benefits less. This is mainly due to the outer loop and the inner loop, which make the access path longer. Accordingly, this organization schema is ideal for insert and update operations but may be less effective for lookup and scan workloads.

2) CLUSTERING DATA BLOCKS

In clustering data block organization, data blocks are sorted in the block memory heap on their key values. These lock-free data structures (Google B-Tree, Mass-Tree, CuckooMap, HopcotchMap) have achieved a high performance for write-heavy workloads. Figure 6 describes the organization schema of the relation datasets by leveraging memory-efficient data structures. For each attribute of the relation, there is a default area of fixed size consecutive memory addresses. When the sequential insertions are executed, values of the records are separately stored in the partitioned blocks. If the block has no other slot, a new fixed-size block will be created and put to the block-BTree or the block-HashMap.

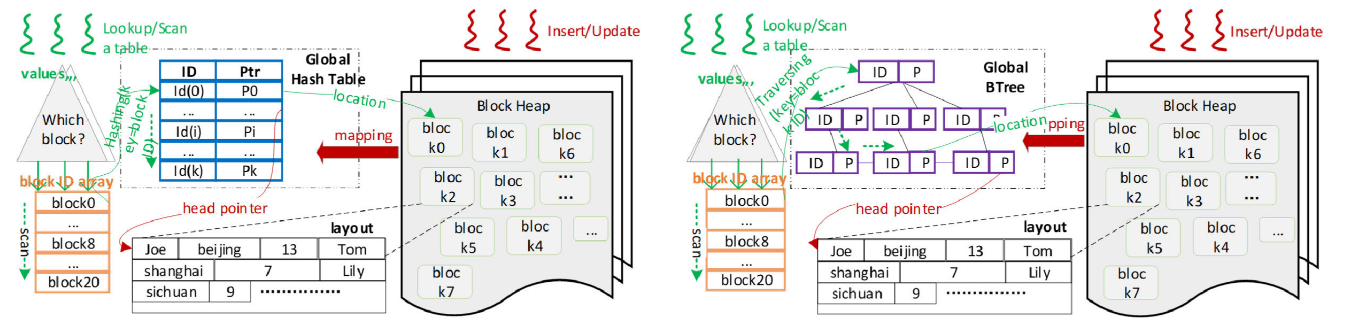


FIGURE 5. Dispersing data blocks architecture overview. Blocks are utilized in NSM fashion, i.e., the attributes of a given record are contiguous. The variable-length attributes are stored in another memory heap, referenced by the indirect pointers in the block. Horizontal partition is used to split a relational dataset.

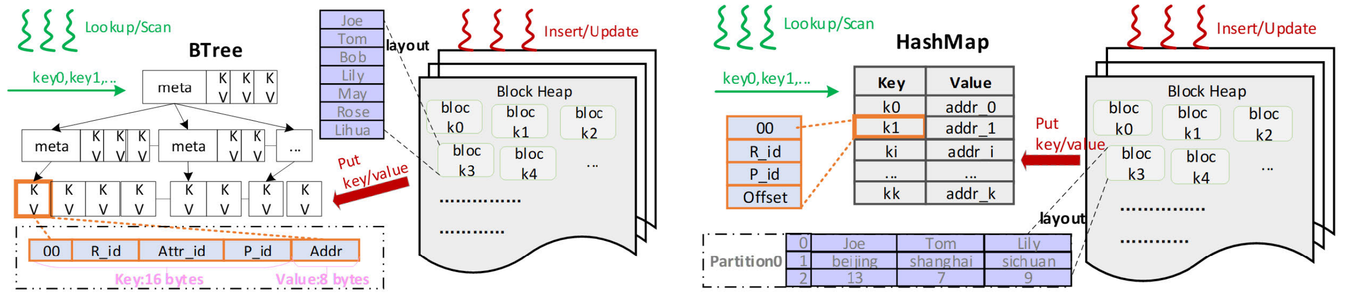


FIGURE 6. Clustering data blocks architecture overview. On the left, data blocks are organized in the BTree, such as Google B-Tree and Mass-Tree, utilizing the DSM fashion, i.e., each attribute of a given relation is contiguous. On the right, data blocks are organized in the Hash Map, such as CuckooMap and HopscotchMap, utilizing the PAX fashion, and each attribute of a partition is contiguous. The variable-length attributes are stored in another memory heap, referenced by the indirect pointers in the block. Horizontal and vertical partitions are used to split a relational dataset.

Algorithm 1 Dispersing Data Blocks Organization Scan Algorithm

```
Require: blockARRAY
Ensure: buffer < value >
1:  /*scan a table by looping though block array */
2:  while i ← 0 to sizeOf(blockARRAY) − 1 step 1 do
3:      /*fetch the block id*/
4:      blockID ← blockARRAY[i]
5:      /*find the location by searching a mapping table*/
6:      blockPTR ← SearchBlockOrganization(blockID)
7:      block ← Load(blockPTR)
8:      /*construct valid data set by scanning each block*/
9:      while j ← 0 to sizeOf(block) step 1 do
10:         /*fetch each value of the block*/
11:         while k ← 0 to sizeOf(attributes) step 1 do
12:             /*check the validity of the target*/
13:             attrOFFSET ← CalculateOffset(attr[k])
14:             attrPTR ← Load(attrOFFSET)
15:             attrVALUE ← Load(attrPTR)
16:             check ← CheckValidity(attrVALUE)
17:             /*put the valid values to the output*/
18:             buffer < value > ← Buffer(attrVALUE)
19:         end while
20:     end while
21: end while
```

The data blocks are aligned to the 64-byte cache block boundary with the aligned attribute. Using this particular partitioning, the number of cache misses per block shrinks to block size/cache line size when executing a full scan. In addition, this benefits the load (value) and condition check in the loop and improves performance for two reasons. As shown in Algorithm 2, first, it allows GetBlock and fetches each value as array iteration. Second, because instruction prefetching and pipelining can be adversely affected due to branch prediction misses, we can amortize the condition checking cost by performing it only once for the whole block. Usually, due to the complexity of the workloads, condition checks will occur frequently, and we thus improve the execution efficiency of the loop. If the execution plan is simple, then the throughput and access latency are limited by the data block loop. When the execution plan has more operators and condition checks, the lookup and scan performance drops as a consequence of high cycles and IPC. Even though this schema has a relatively poor performance, the experimental evaluation shows that it still outperforms the dispersing data block organization for most workloads. We expect that the performance of a perfect data organization will improve the throughput and memory access latency but with an additional computational cost, e.g., estimate the amount of address space to preallocate large virtual addresses for data blocks.

C. ACCESS METHOD

We now briefly discuss the effect of the memory access hindrance on the design of algorithms for common query processing operators. The access method describes how the database accesses the data stored in the memory heap.

Algorithm 2 Clustering Data Blocks Organization Scan Algorithm

```
Require: $tableID$, $columnID$, $constantPartitionSize$
Ensure: $buffer < value >$
 1: /*scan a table by traversing the ordered data structure*/
 2: while true do
 3:     /*find the location and fetch the target values*/
 4:     $blockKEY \leftarrow CaculateKey(PARAMETER)$
 5:     $blockPTR \leftarrow SearchBlockOrganization(blockKEY)$
 6:     $valueARRAY \leftarrow getBlockValue(blockPTR)$
 7:     /*check the validity of each value*/
 8:     while $j \leftarrow 0$ to sizeOf(valueARRAY) step 1 do
 9:         $validity \leftarrow CheckValidity(valueARRAY[i])$
10:         if $validity$ is true then
11:             /*put the valid values to the output buffer*/
12:             $buffer < value > \leftarrow Buffer(value_a rray[i])$
13:         end if
14:     end while
15: end while
```

In general, there are two popular processing models in modern main memory database systems. Our experiments evaluate the vectorization and materialization model using a sequential scan access method.

Considering an SQL query: Select count (R.b) from R, H where R.d = H.f and R.a > ? and H.y < ? and H.z > ?, which accesses the relation Table R(a,b,c,d) and H(x,y,z,e,f). It selects some tuples from R and some tuples from H, and then joins the resulting tuples. Finally, it computes the count of the resulting tuples. The corresponding access patterns and the pseudocode for parts of our realistic implementations are shown in Figure 7. In vectorization-style processing, each operation emits a block of data instead of a single record. A next function is essentially for in-flight loops iterating over the output of their child operator. This allows pipelining where the executor can process a batch of records through as many operators as possible before having to retrieve the next batch. The vectorization-style approach is ideal for read-heavy workloads in which there are a large number of records that need to be scanned, but fewer operator invocations are required. The code idea of materialization-style processing is to breakdown the memory accesses with N dependent operation stages, where each stage consumes the data from the previous stage and prefetches the data for the next stage, inspired by [27]. To hide the memory access latency by doing useful work, all output results of each stage of the in-flight request are kept in an output buffer. Once a lookup has been initiated, its stage state is saved in a temp buffer structure, e.g., valid position, valid count, hash result, status. This stage state is necessary to continue or terminate the lookup. Using the field status, the exact operator stage of an in-flight lookup can be captured easily. For example, stage 0 (sequential scan) loads the required values and performs condition checking, buffering the logical position of the visibility records. Stage 1 (hash table build) requests the hash attributes and consumes the last buffer, finally buffering the hash table. If the valid *count* $<= 0$ produced by stage 0, then the following stages will do nothing.

While accurate, the statements we show in the code are simplified. For instance, three optimizations, not seen in the code, are the following. (1) To eliminate branches to avoid misprediction penalties, we eagerly evaluate the selective predicates, transaction visibility check and reduce the code branches, which have been discussed in many studies [11], [19], [21], [31], [51]. We also try to make the data locality for each hash subprocessing. (2) The global buffer can be invoked between arbitrary operator stages during the in-flight lookup. (3) The logical key ID is used for each attribute to build projection using lazy materialization, which reduces the intermediate results compared to early materialization.

TABLE 1. Experimental platforms.

Processor	Intel Xeon E5-2620 v3
Technology	22nm @ 2.40GHz
ISA	X86
CMP Cores/Threads	6 / 12
Core Architecture	Haswell
SIMD width	256-bit
Issue width	8-way
L1 I/D Cache (per core)	32KB
L2 Cache	256KB
Last Level Cache(L3)	15MB
TLB entries (L1/L2)	64 / 1024
Main Memory	DRAM DDR4 64GB
Max Memory Bandwidth	59 GB/s

IV. EVALUATION METHODLOGY

A. EXPERIMENT SETUP

1) HARDWARE

The server machines used in our experiments are listed in Table 1. The Intel Xeon E5-2620 server features a two-socket CPU with 6 cores per socket. The server runs Ubuntu 18.04 Linux (kernel version 5.4). On x86, we compile our code with g++ 5.5 using the −O3 flag. For prefetching data blocks, on x86, we use the PREFETCHNTA instruction via the built-in g++ function.

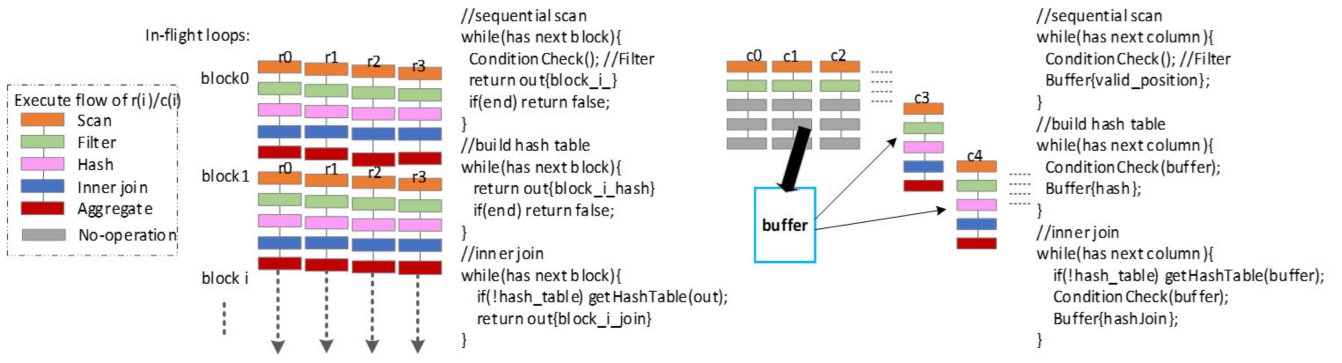
2) SOFTWARE

To reduce the impact of different implementations, we implement all data structures and design using the PELOTON in-memory database system. PELOTON is 100% open-source and completely provides the execution and storage engine. We use the OLTPBenchmark Framework [52] to produce a relational database of variable rates and variable mixture loads via JDBC. As it also provides data collection features, e.g., per-operation latency and throughput records, we can analyze details by fully utilizing these statistics.

B. WORKLOADS

1) YCSB

To analyze the memory access performance, we use a set of Yahoo! Cloud Serving Benchmark (YCSB) [53]



```
//sequential scan
while(has next block){
 Condition Check(); //Filter
 return out{block_i_}
 if(end) return false;
}
//build hash table
while(has next block){
  return out{block_i_hash}
  if(end) return false;
}
//inner join
while(has next block){
   if(!hash_table) getHashTable(out);
   return out{block_i_join}
}
```

```
//sequential scan
while(has next column){
 ConditionCheck(); //Filter
 Buffer{valid_position};
}
//build hash table
while(has next column){
 ConditionCheck(buffer);
 Buffer{hash};
}
//inner join
while(has next column){
  if(!hash_table) getHashTable(buffer);
  Condition Check(buffer);
  Buffer{hashJoin};
}
```

FIGURE 7. Execution patterns of vectorization-style processing, materialization-style processing and pseudocode of the critical operations. block(i) is the i block of the given relation. r(i) is the i record of the in-flight loop block. c(i) is the i attribute of the given relation.

microbenchmarks to emulate lookup and scan operations under relatively simple application scenarios. We configure workloads Read-only and Scan-only with Zipfian distributions, which have skewed access patterns common to random lookup and scan workloads. We use the default Zipfian constant of 0.99 to ensure that the read and scan items will be more uniform. For each workload, we test key and field types: 32-bit random integers. We test two field length: 10 and 100. Then, there are approximately 44 bytes or 404 bytes for each record in the usertable YCSB database. The scale factor varies from 1 thousand to 10 million records. Because our experimental logical processor is 12, the terminal is set to 10. In addition, we expect to evaluate how the dataset, data scale, field size, projectivity, and partition settings affect the DRAM memory access performance under different data block organizations.

2) TPC-C

This benchmark is the current standard for measuring the performance of OLTP systems [54]. It models a warehouse-centric order processing application with nine tables and five transaction types. We test only one transaction StockLevel that scans the order line table and stock table after looking up the district table. We set the number of warehouses to 20 and scale up the number of threads to measure the overall throughput and memory access latency. We also change the number of warehouses to compare the access methods, complexity of datasets, the physical pointer and logical pointer, thread contention, and data prefetching.

V. EXPREMENT ANALYSIS

A. DATASET

In this experimental evaluation, we analyze the performance impact of the dataset using the YCSB workload and the TPC-C workload. For the YCSB workload, there is only 1 relation dataset, and all the field value types are integers. For the TPC-C workload, there are 9 relation datasets, and the field value types vary, such as int, double, timestamp, data, varchar, and char. We limit the 10 threads of concurrent accessing for each workload. We load 10 million records for YCSB and 20 warehouses for TPC-C, which need 500 800 MB DRAM memory.

Figures 8 and 9 show the results for these two workloads. The clustered data block organization executes the scan-only workload up to 1200× faster than dispersing the data block organization. However, the clustering data block organization executes the stock-level workload to fall below 2× faster than dispersing the data block organization. This is because the complex value types of the data block need to store large indirect memory pointers that reference the start of allocated storage. Then, generally, regular and irregular traversal paths will require more complex access mechanisms. In addition, consecutive memory addresses of integer value allow us to implement the GetBlock, not the GetValue, which will incur the overhead of pointing.

To verify whether GetValue results in a greater number of cache misses, we measure the performance counters for stock-level workload using *Perf*. As Figure 10 shows, during the query execution, 80%, cache misses are caused by GetValue. Varying the data block organization is possible at a performance improvement, which is so small if the dataset is unregular and pointer intensive.

B. DATA SCALE

The results for the throughput and average latency experiments of the dispersing data block organization are shown in Figure 11 and Figure 12, respectively. We observe that the throughput drops as we increase the number of data scales from 1 thousand to 10 million records. After the data scale exceeds the L3 cache, the throughput drops seriously, and the average latency increases by 5×. If we implement the dense index for the dispersing CuckooMap and the sparse index for the dispersing Google BTree, we see that the dense index outperforms the sparse index, and the performance gap increases to 5× after the L3 cache. Scanning only the workload performance of the sparse index grows unsteadily as the data scale increases, but the dense index performs smoothly. These results are partly because the generated scan items follow a random distribution. As the data scale increases, the sparse index performance impacts the distribution of scan data.

Performance Indicator	dispersing cuckoomap	dispersing google btree	dispersing cuckoomap + dense index	dispersing googletree + sparse index	clustering google btree	clustering masstree	clustering cuckoomap	clustering hopscotchmap
Throughput (request/second)	1.3333	1.3114	2332.7104	923.2262	1969.6312	1915.1227	1891.4901	1861.0383
Average latency(millisecond)	6410.8440	6429.0994	4.2819	10.8254	5.0724	5.2171	5.2823	5.3686

FIGURE 8. YCSB workload, scan only, throughput and average latency with the different implementations.

Performance Indicator	dispersing cuckoomap	dispersing cuckoomap + dense index	clustering google btree	clustering mass tree	clustering cuckoomap	clustering hopscotchmap
Throughput (request/second)	0.8196	26.1148	1.5573	1.5409	1.6721	1.6885
Average latency(millisecond)	9349.4161	380.2715	5596.8148	5641.2158	4500.2121	5192.0101

FIGURE 9. TPC-C workload, stock level, throughput and average latency with the different implementations.

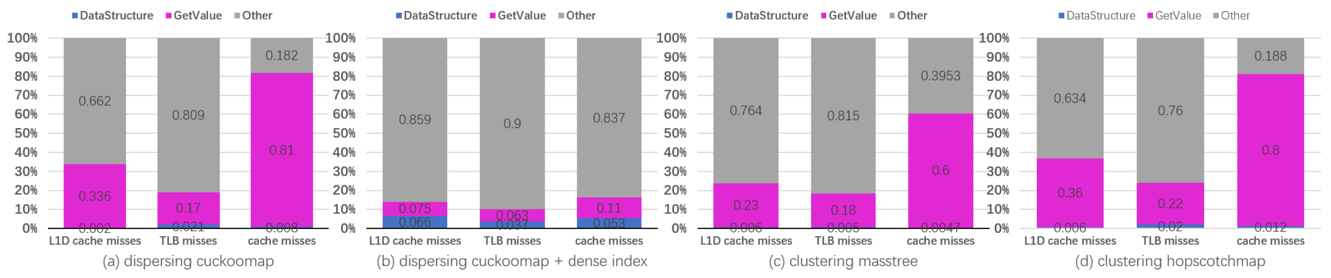


FIGURE 10. TPC-C workload, stock level, L1D cache misses, TLB misses, and cache misses for dispersing and clustering data block organization.

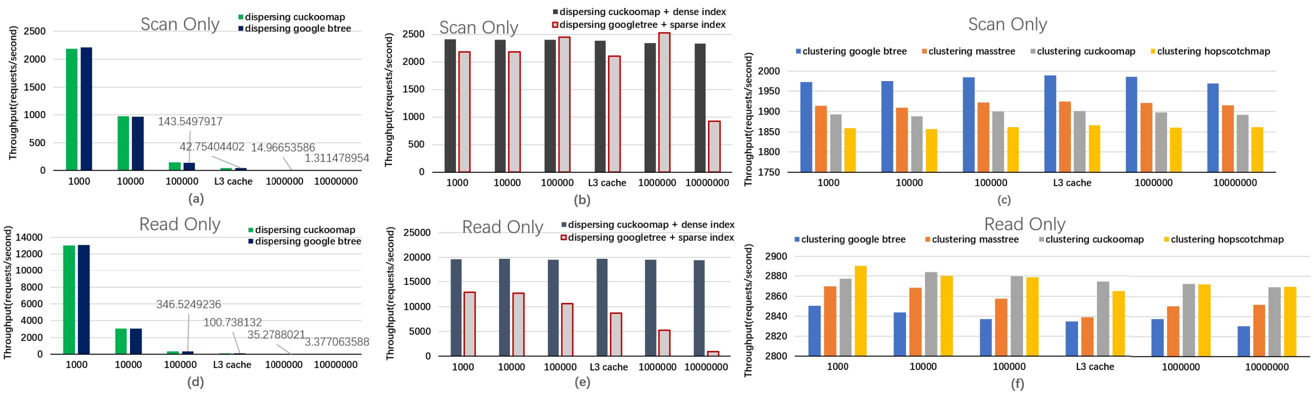


FIGURE 11. YCSB, read only, scan only, throughput of different data block organizations with varying data scales.

From this experiment, we can also see that the clustering data block organization increases slightly for the scan only and read only workload as the data scale increases. If the data scale is full with the L3 cache, the performance improves. Where the data scale does not match the L3 cache, the performance will drop. However, the performance gap between the different clustering organizations is small. The reason why the clustering data blocks achieve high performance is because the storage format makes GetBlock and the materialization-style access methods possible. This schema can leverage the CPU registers during query execution. We repeated the same experiment using the TPC-C workload. We set the 10 thread contentions and varied the warehouse from 1 to 20. From Figure 13, we observe that the average latency of all the implementations grows near-linearly with the increase in the number of warehouses, while the dense index schema maintains a latency that is lower when the warehouse is 20. We attribute this to the advantage of the physical pointers and logical pointers. The results show that at the 20 warehouses, compared with using logical pointers, the system's memory access latency when using physical pointers is 53% lower.

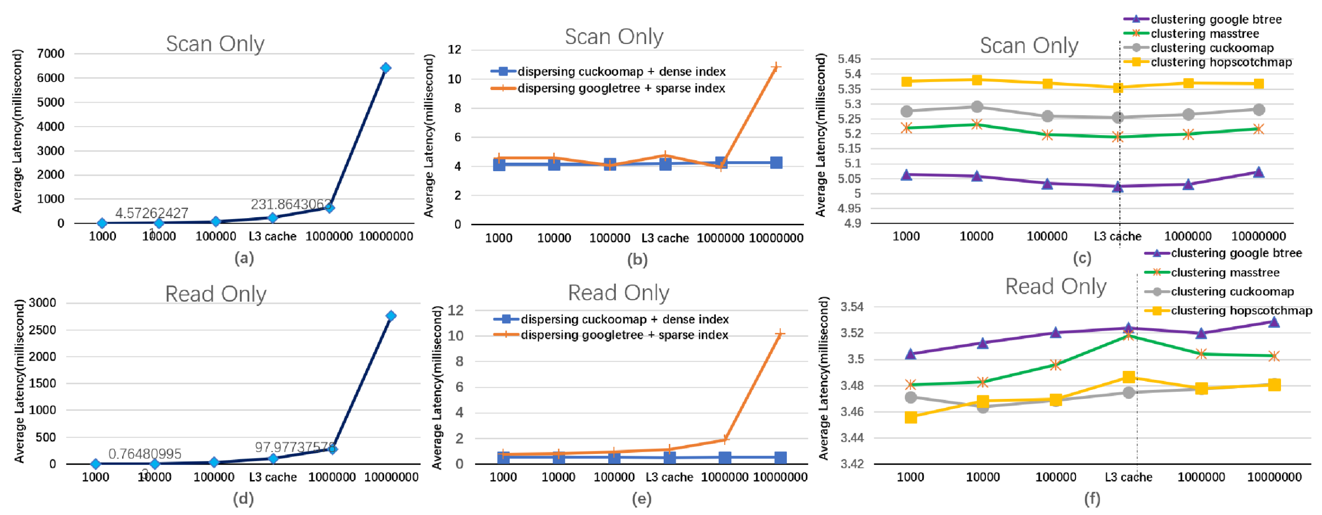


FIGURE 12. YCSB, read only, scan only, average latency of different data block organizations with varying data scales.

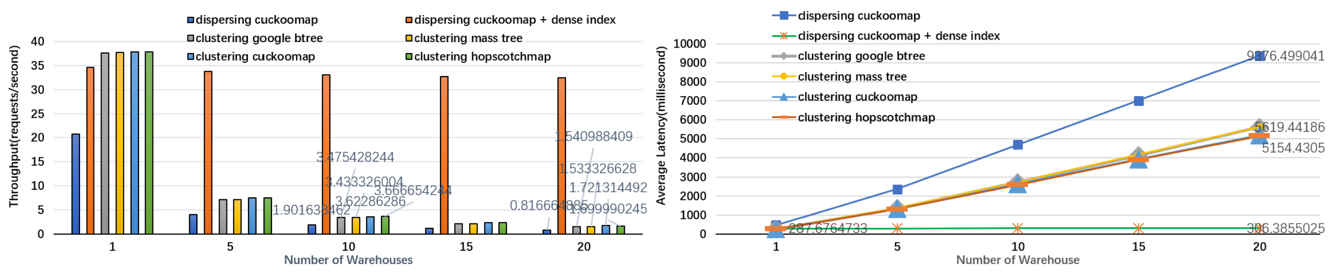


FIGURE 13. TPC-C, stock level, the throughput and average latency of different data block organizations with varying warehouses.

C. FIELD SIZE

In this experiment, we will examine the impact of field size on the throughput performance of the different data block organization schemas. The projectivity of the scan is set to 100% of all the attributes in the relation table, where one record is 44 bytes in size for 10 fields and 404 bytes in size for 100 fields. Figure 14 shows that for the throughput performance, when using the dispersing data block schema drops severely, but the clustering data block organization does not. We attribute this to their usage of memory bandwidth and multiple-cache by consecutively fetching the attributes required by the scan execution. In the dispersing data block organization schema, the system execution engine scans records only by using the GetValue interface and the references stored in the data blocks.

D. PARTITION

We next examine the impact of partitioning on the different implementations. In this experiment, we load 10 million records, and each record has 100 fields. We evaluate the alteration in the throughput and average latency due to the size of the data block. The results for the dispersing data block organization are shown in Figure 15. For the dispersing Cuckoomap and dispersing Google BTree organization, we see that the throughput and latency performance is stable regardless of the size of the data block that the system configurates. When the data block full the L3 cache, the average access latency is lower by 2%. In contrast, for the dispersing Google BTree + sparse schema, there is a high throughput and low latency when increasing the data block size. This is because the sparse index is partition intensive, which reduces the scan loop instructions. With the sparse index, the average latency of the scan operation downwards near-logarithmically with the increase partition.

As shown in Figure 16, we also have some interesting findings showing that the performance declines over partition size. This worst performance is a consequence of the materialization-style access method. As the partition size increases, the data blocks cannot fill the cache efficiency. Furthermore, GetBlock and 100% projectivity have an effect on access latency, which leads to large intermediate results.

E. PROJECTIVITY

In this experiment, we load the 10 million records and configure 100 fields for each record. We vary the projecting attributes from 1% to 100%. Figure 17 presents the

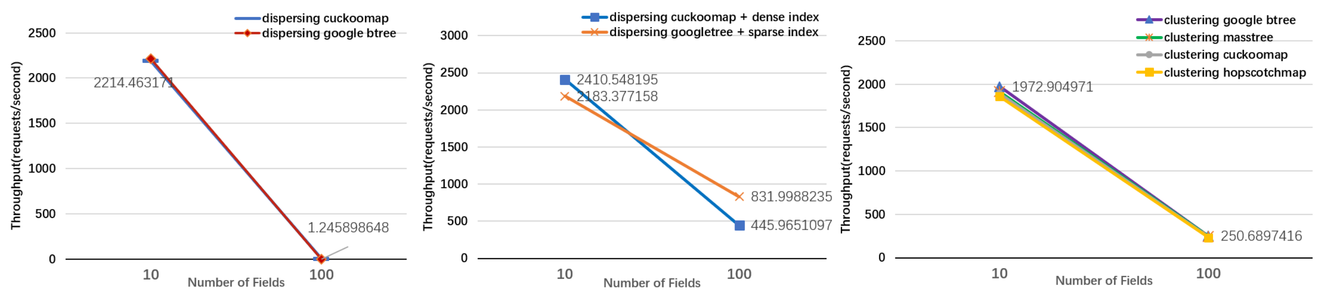


FIGURE 14. YCSB, scan only, throughput of different data block organizations with varying field sizes.

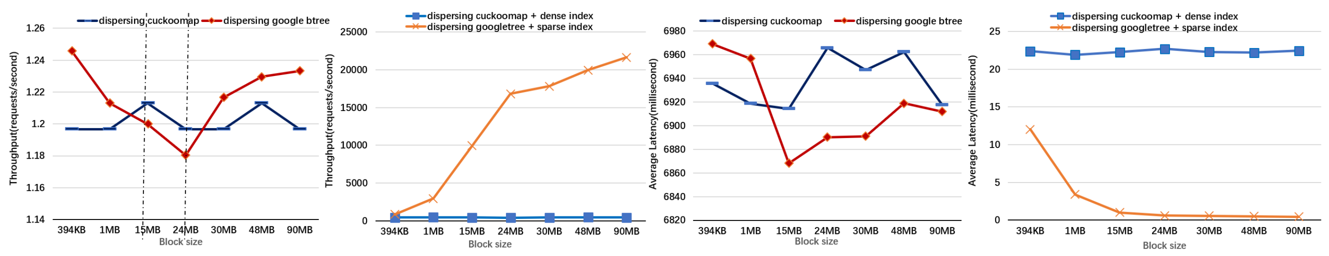


FIGURE 15. YCSB, scan only, throughput and average latency of dispersing data block organization. 394 KB (1000 records), 1 MB (4000 records), 15 MB (38000 records), 24 MB (64000 records), 30 MB (75000 records), 48 MB (128000 records), 90 MB (235000 records) are the different block size.

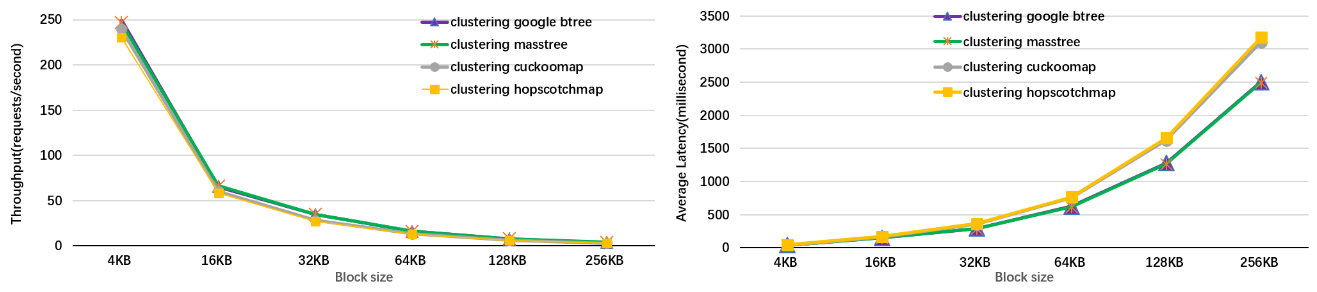


FIGURE 16. YCSB, scan only, throughput and average latency of clustering data block organization. 4 KB (1000 records), 16 KB (4000 records), 32 KB (8000 records), 64 KB (16000 records), 128 KB (32000 records), 256 KB (64000 records) are the different block size.

throughput and average latency performance when the system executes scans using the dispersing data block organization schema. Both the dispersing Cuckoomap and GoogleBTree perform up and down with varying projectivity. This is because of the lazy materialization for the project operation. In addition, we partly attribute this to hashing and tree data structures. In contrast, the sparse index schema's performance is stable regardless of the variety of projectivity.

As we expected, Figure 18 shows that the clustering data block organization schema will deteriorate from a low throughput and a high average latency as the projectivity ratio increases. This is because clustering data blocks the organization schema and only fetches the required attributes of records, which may require more projection overhead as the projectivity ratio increases.

F. THREAD CONTENTION

The salient aspect of these data structures' design is that they are all lock-free and ideal for modern multicore CPUs. Even though we only analyze the memory access latency in this evaluation, to better understand this issue, we create a high contention workload environment. We load 20 warehouses and vary the current threads from 1 to 20. The results shown in Figure 19 indicate that all six implementations degrade under high contention. Both dispersing and clustering schema performance drops as the threads increase. Overall, under high contention, dispersing the dense index has the best result, followed by clustering the hashing schema and then clustering the tree schema. The dispersing Cuckoomap suffers from logical pointers as threads contend for the data block slots. The average access latency performance gap between the clustering tree schema and hashing schema grows to 18% as the thread contention increases. We attribute this to the multithreads, in-memory data structures and the PAX storage model, which brings the data locality.

G. DATA PREFETCHING

These data structures in original papers claim that this design incurs fewer cache misses and leads to low memory access latency. We load the 20 warehouse and configured 10 threads.

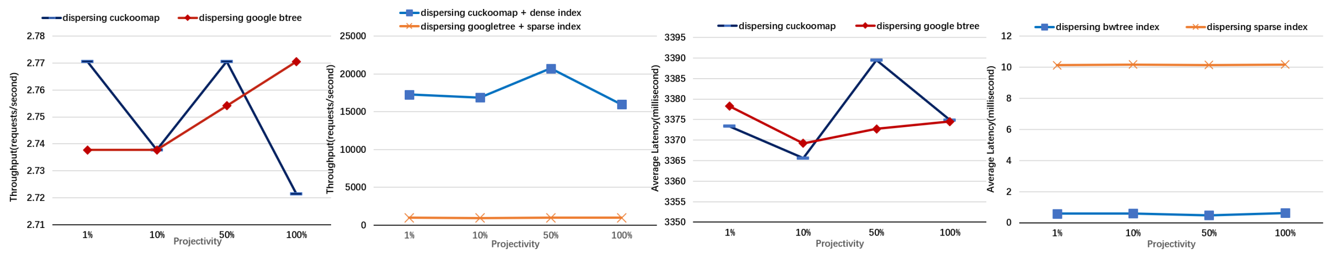


FIGURE 17. YCSB, scan only, throughput and average latency of dispersing data block organization with varying the projectivity.

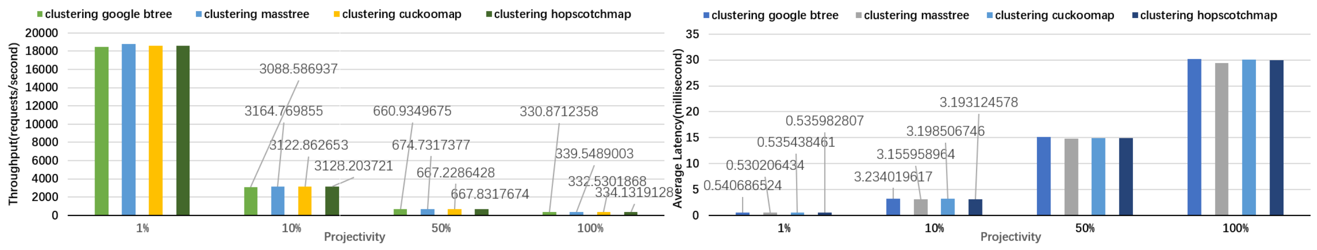


FIGURE 18. YCSB, scan only, throughput and average latency of clustering data block organization with varying the projectivity.

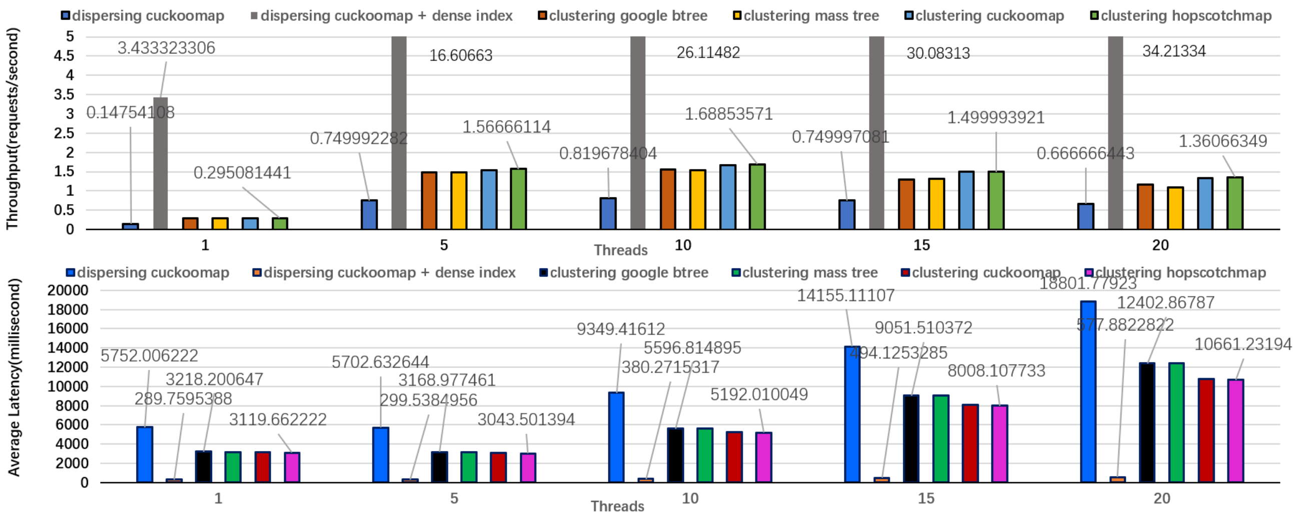


FIGURE 19. TPC-C, stock level, the throughput and average latency of different data block organizations with varying concurrent threads.

We use perf to measure running metrics, such as L1D cache misses, TLD misses and instructions. Figure 20 shows that the clustering Google BTree schema is unsuccessful because of the number of nodes, which requires fewer array loops as the partition size increases. Furthermore, in our experiments, the optimal node size threshold is 32 bytes when the partition size is set to 1000.

As we expect, the MassTree data block organization with prefetch outperforms the schema without prefetch. From Figure 21, we observe that the masstree with prefetch achieves a higher throughput and lower average latency due to fewer TLB cache misses. However, it also has the large instructions. Regardless, this software prefetching by performing address calculation for future memory accesses by hand in the program steam benefits the memory access latency reduction.

VI. SUMMARY OF EXPERIMENTS

We describe a large number of experiments in Section 5. In this section, we summarize our findings and connect them back to the impact discussed in Section 3. Our focus is to understand the relative performance of the two block organization schemas, which are alternatives to different implementations. Our high-level conclusion is that in the in-memory setting, for systems using block-based architecture, the performance has a gap between these two alternatives. We now analyze the impact of individual dimensions.

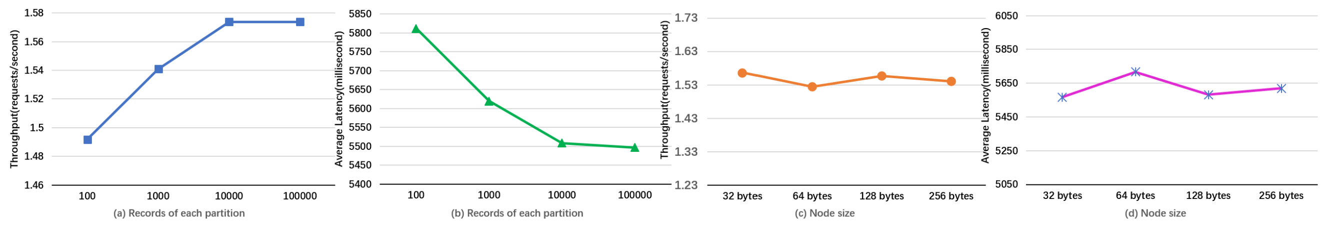


FIGURE 20. TPC-C, stock level, throughput and average latency of clustering GoogleBTree data block organizations with varying partitions (a), (b) and node sizes (c), (d), such as 32 bytes, 64 bytes, 128 bytes and 256 bytes.

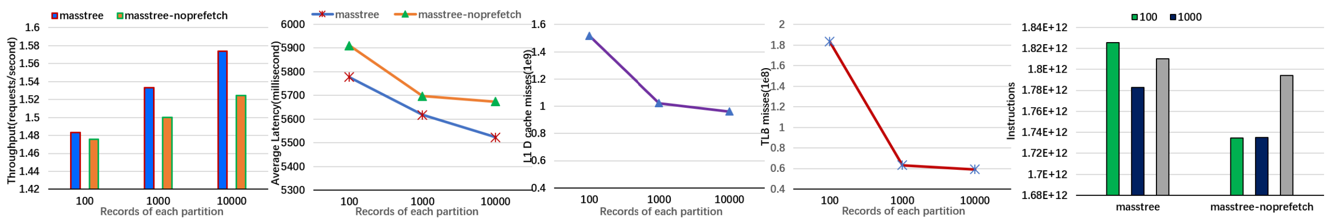


FIGURE 21. TPC-C, stock level, throughput, average latency, L1D cache misses, TLB misses and instructions of clustering masstree data block organization with varying concurrent threads.

A. WORKLOAD

We find that the database workload bridges the gap between the performance of the dispersing data block organization and clustering data block organization. Despite the different storage layouts, the two organization schemas have similar access latency performances resulting from complex data types that could produce long pointer access paths. Data scale variation largely impacts dispersing data block organization schemas. When the field size of the relation is wider, the clustering data block organization outperforms the dispersing schema. Under high thread contention, their performance degrades linearly.

B. BLOCK SIZE

Block size can largely affect the performance of clustering data block organization. As the materialization-style access algorithm we discussed in Section 3, the CPU register cannot fill the whole block.

C. PARALLELISM

The performance gap between the two alternatives is largely unaffected by data parallelism. We note that the dispersing data block organization works well in most workloads, although query processing is based on data block pipeline parallelism.

D. INDEX

The index is commonly used to speed up query processing when the data blocks are dispersed. In our experiments, the dense index and sparse index are implemented by using Open BW-Tree and Zone Map, respectively. The dense index always has high throughput and low access latency. However, the sparse index has unstable performance and performs poorly on a large data scale.

E. SOFTWARE PREFETCHING

Software prefetching improves the scan performance when using the Mass-Tree clustering data block organizations in a representative query. Prefetching did not perform well, as we expected, as Mass-Tree has a relatively poor performance for the YCSB scan workload.

VII. RELATED WORK

There have been many experimental studies on memory access performance for in-memory databases. For instance, Anastassia Ailamaki [18], [55] examines the data organization model PAX (Partition Attributes Across), which achieves high cache utilization and performance for modern processors. In main-memory experiment environments, PAX can reduce 50%∼75% compared to NSM (N-ary Storage Model) [17]-oriented disk pages. When compared to the DSM (decomposition storage model) [16] researched by George P, PAX also performs faster, and memory access remains stable as required attributes increase. In [49], Richard A. Hankins et al. presented a flexible data storage technique called data morphing that significantly improves performance by calculating the layout and trading optimality for faster time complexity. To optimize accesses to all levels of the memory hierarchy and for all the different workloads, [56] designed a buffer pool to narrow the access latency gap between volatile and nonvolatile storage. MonetDB/X100 ColumnBM [57] also stores all delta columns using PAX in the memory buffer pool to reduce the update operation cost.

Optimized cache efficiency and multicore parallelism data structures have emerged in recent years. These include adaptive radius trees [58], [59], open Bw-trees [39], [40], Bz-trees [26], Mass trees [37], [38], FPTrees [60], [61], master trees [4], [37], B+trees [62], cuckoo hashing [43],

[44], [63], and hopscotch hashing [41], [42]. Indices allow the system to look up data quickly, without scanning all of the records. Traditional concurrent data structures use locks to provide fine-grained concurrent access, which scales poorly on modern multicore CPUs. Optimistic locking not only improves the scalability of data structures on modern processors, but also allows the processing of more complex workloads when using multiversion concurrency control [50]. Otherwise, for these design decisions, such as index, version storage, and data representation, most HTAP transactional analytical implementations store records in global data structures in the main databases [3], [5], [7], [8], [64]–[66].

New hardware technologies create new opportunities and challenges for memory access latency in in-memory databases. Reference [11]researched mainstream CPUs with wider SIMD registers and presented the impact of efficient vectorization on the algorithmic designs and implementations of in-memory database operators. Pipelining query execution is critical in an in-memory database, but for system designers, pipelines or not pipelines, [20] examined the narrow gap between them. Reference [19] analyzed vectorization or data-centric code generation, which are query engines adopted by most modern in-memory databases. Both are efficient, vectorization is better at hiding cache miss latency, whereas data-centric compilation requires fewer CPU instructions, which benefits cache-resident workloads. Furthermore, in-memory database systems commonly use pointers for direct access to records in memory. This is because pointers can save space for large variable length values, just stored once, referenced by memory pointers everywhere. However, pointer chaining is truly limited for memory access latency. Reference [27], [32], [67] researched approaches using hardware-based memory traces and software-based prefetching, which all improved the memory access performance. In addition, hardware transactional memory (HTM) based on cache coherence has been studied in many works [68]–[71].

VIII. DISCUSSION AND CONCLUSION

In this work, we present generic DRAM access patterns for in-memory databases. We design data block organization schemas using recent memory-efficient data structures. We implement query processing algorithms in the in-memory database PELOTON.[2] Our evaluations show that data block organization schemas benefit memory access performance improvement. However, in some cases, their performance gap is narrow. We discuss the many dimensions that impact DRAM access performance for an in-memory database. Though the summary of experiments, we gain four insights. **I1**: The performance of memory access for dispersing organization rapidly degrades as read workload becomes heavy. This mainly due to the L3 cache misses. **I2**: The effectiveness of two data organizations to lower the access latency depends on the access methods. The access pointer chain can be pruned eagerly by using optimized processing algorithms. **I3**: Results produced from dispersing and clustering implementations may be inconsistent with the theory. **I4**: Even hardware and software prefetching techniques cannot improve the performance of memory access if there is more transaction validation mixed into the query workload.

In future work, we plan to support TPC-H, and to include index data structure modules based on our preliminary studies. We expect that our work will help to advance DRAM access for in-memory data management system research.

[2]https://github.com/gitzhqian/peloton-trees-hashing

REFERENCES

[1] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato, "Managing non-volatile memory in database systems," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 1541–1555.

[2] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 195–206.

[3] *Peloton—The Self Driving Database Management System*. Accessed: May 7, 2018. [Online]. Available: htts://pelotondb.io

[4] H. Kimura, "FOEDUS: OLTP engine for a thousand cores and NVRAM," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 691–706.

[5] 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 *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1243–1254.

[6] M. Stonebraker and A. Weisberg, "The VoltDB main memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, Jun. 2013.

[7] G. Martin, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "Hyrise: A main memory hybrid storage engine," *Proc. VLDB Endowment*, vol. 4, no. 2, pp. 105–116, 2010.

[8] J. Arulraj, A. Pavlo, and P. Menon, "Bridging the Archipelago between row-stores and column-stores for hybrid workloads," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 583–598.

[9] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews, "The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database," *Proc. VLDB Endowment*, vol. 9, no. 13, pp. 1401–1412, Sep. 2016.

[10] J. Hofmann, J. Treibig, G. Hager, and G. Wellein, "Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi-and manycore chips," in *Proc. Workshop Program. Models SIMD/Vector Process.*, 201, pp. 57–64.

[11] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD vectorization for in-memory databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 1493–1508.

[12] U. Sirin, P. Tözün, D. Porobic, A. Yasin, and A. Ailamaki, "Micro-architectural analysis of in-memory OLTP: Revisited," *VLDB J.*, vol. 4, pp. 1–25, Mar. 2021.

[13] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. Knowl. Data Eng.*, vol. 4, no. 6, pp. 509–516, Dec. 1992.

[14] J. Krueger, M. Grund, M. Boissier, A. Zeier, and H. Plattner, "Data structures for mixed workloads in in-memory databases," in *Proc. 5th Int. Conf. Comput. Sci. Converg. Inf. Technol.*, Nov. 2010, pp. 394–399.

[15] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, Mar. 2020, pp. 513–526.

[16] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model," *ACM SIGMOD Rec.*, vol. 14, no. 4, pp. 268–279, May 1985.

[17] E. F. Codd, "A relational model of data for large shared data banks," in *Software Pioneers*. Berlin, Germany: Springer, 2002, pp. 263–294.

[18] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proc. VLDB*, vol. 1, 2001, pp. 169–180.

[19] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *Proc. VLDB Endowment*, vol. 11, no. 13, pp. 2209–2222, Sep. 2018.

[20] H. Deshmukh, B. Sundarmurthy, and J. M. Patel, "To pipeline or not to pipeline, that is the question," 2020, *arXiv:2002.00866*.

[21] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and A. C. Koch, ''How to architect a query compiler,'' in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1907–1922.

[22] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh, ''Quickstep: A data platform based on the scaling-up approach,'' *Proc. VLDB Endowment*, vol. 11, no. 6, pp. 663–676, Feb. 2018.

[23] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and A. T. Zhang, ''Self-driving database management systems,'' in *Proc. CIDR*, 2017, pp. 1–6.

[24] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann, ''SQL and operator-centric data analytics in relational main-memory databases,'' in *Proc. EDBT*, 2017, pp. 84–95.

[25] M. Haubenschild, C. Sauer, T. Neumann, and V. Leis, ''Rethinking logging, checkpoints, and recovery for high-performance storage engines,'' in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 877–892.

[26] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, ''Bztree: A high-performance latch-free range index for non-volatile memory,'' *Proc. VLDB Endowment*, vol. 11, no. 5, pp. 553–565, Jan. 2018.

[27] O. Kocberber, B. Falsafi, and B. Grot, ''Asynchronous memory access chaining,'' *Proc. VLDB Endowment*, vol. 9, no. 4, pp. 252–263, Dec. 2015.

[28] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, ''FAST: Fast architecture sensitive tree search on modern CPUs and GPUs,'' in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2010, pp. 339–350.

[29] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, ''Improving hash join performance through prefetching,'' *ACM Trans. Database Syst.*, vol. 32, no. 3, p. 17, Aug. 2007.

[30] S. Ainsworth and T. M. Jones, ''Software prefetching for indirect memory accesses,'' in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2017, pp. 305–317.

[31] G. Zhang and D. Sanchez, ''Leveraging caches to accelerate hash tables and memoization,'' in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 440–452.

[32] M. Cavus, R. Sendag, and J. J. Yi, ''Informed prefetching for indirect memory accesses,'' *ACM Trans. Archit. Code Optim.*, vol. 17, no. 1, pp. 1–29, Mar. 2020.

[33] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, ''IMP: Indirect memory prefetcher,'' in *Proc. 48th Int. Symp. Microarchitecture*, Dec. 2015, pp. 178–190.

[34] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, ''Guided region prefetching: A cooperative hardware/software approach,'' in *Proc. 30th Annu. Int. Symp. Comput. Archit.*, 2003, pp. 388–398.

[35] *Google BTree*. Accessed: Feb. 6, 2013. [Online]. Available: https://code.google.com/archive/p/cpp-btree/downloads

[36] *Mass Tree*. Accessed: Jan. 2, 2020. [Online]. Available: https://github.com/kohler/masstree-beta

[37] Y. He, J. Lu, and T. Wang, ''CoroBase: Coroutine-oriented main-memory database engine,'' 2020, *arXiv:2010.15981*.

[38] Y. Mao, E. Kohler, and R. T. Morris, ''Cache craftiness for fast multicore key-value storage,'' in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 183–196.

[39] J. J. Levandoski, D. B. Lomet, and S. Sengupta, ''The bw-tree: A B-tree for new hardware platforms,'' in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 302–313.

[40] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen, ''Building a bw-tree takes more than just buzz words,'' in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 473–488.

[41] M. Herlihy, N. Shavit, and M. Tzafrir, ''Hopscotch hashing,'' in *Proc. Int. Symp. Distrib. Comput.* Berlin, Germany: Springer, 2008, pp. 350–364.

[42] R. Kelly, B. A. Pearlmutter, and P. Maguire, ''Lock-free hopscotch hashing,'' in *Proc. Symp. Algorithmic Princ. Comput. Syst.*, 2020, pp. 45–59.

[43] N. Nguyen and P. Tsigas, ''Lock-free cuckoo hashing,'' in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, Jun. 2014, pp. 627–636.

[44] R. Pagh and F. F. Rodler, ''Cuckoo hashing,'' *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[45] B. Fan, D. G. Andersen, and M. Kaminsky, ''MemC3: Compact and concurrent memcache with dumber caching and smarter hashing,'' in *Proc. 10th Symp. Netw. Syst. Design Implement.*, 2013, pp. 371–384.

[46] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The Art Multiprocessor Programming*. London, U.K.: Newnes, 2020.

[47] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, ''Algorithmic improvements for fast concurrent cuckoo hashing,'' in *Proc. 9th Eur. Conf. Comput. Syst. - EuroSys*, 2014, pp. 1–14.

[48] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, ''Evaluating persistent memory range indexes,'' *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 574–587, Dec. 2019.

[49] R. A. Hankins and J. M. Patel, ''Data morphing: An adaptive, cache-conscious storage technique,'' in *Proc. VLDB Conf.*, 2003, pp. 417–428.

[50] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, ''An empirical evaluation of in-memory multi-version concurrency control,'' *Proc. VLDB Endowment*, vol. 10, no. 7, pp. 781–792, Mar. 2017.

[51] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu, ''Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware,'' in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 362–373.

[52] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, ''OLTP-bench: An extensible testbed for benchmarking relational databases,'' *Proc. VLDB Endowment*, vol. 7, no. 4, pp. 277–288, Dec. 2013.

[53] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, ''Benchmarking cloud serving systems with YCSB,'' in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[54] TPC-C Benchmark (Revision 5.9.0). (Jun. 2007). *TPP Council—Web Site*. [Online]. Available: http://www.tpc.org

[55] A. Ailamaki, D. J. DeWitt, and M. D. Hill, ''Data page layouts for relational databases on deep memory hierarchies,'' *VLDB J. Int. J. Very Large Data Bases*, vol. 11, no. 3, pp. 198–215, Nov. 2002.

[56] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and A. R. Ganger, ''Clotho: Decoupling memory page layout from storage organization,'' in *Proc. Int. Conf. Very Large Data Bases*, vol. 30, 2004, pp. 696–707.

[57] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman, ''MonetDB/X100-A DBMS in the CPU cache,'' *IEEE Data Eng. Bull.*, vol. 28, no. 2, pp. 17–22, 2005.

[58] V. Leis, A. Kemper, and T. Neumann, ''The adaptive radix tree: ARTful indexing for main-memory databases,'' in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 38–49.

[59] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, ''In-memory big data management and processing: A survey,'' *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, Jul. 2015.

[60] K. Kim, T. Wang, R. Johnson, and I. Pandis, ''ERMIA: Fast memory-optimized database system for heterogeneous workloads,'' in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 1675–1687.

[61] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, ''FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory,'' in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 371–386.

[62] W. Zhang, Z. Yan, Y. Lin, C. Zhao, and L. Peng, ''A high throughput B+tree for SIMD architectures,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 707–720, Mar. 2020.

[63] T. Maier, P. Sanders, and R. Dementiev, ''Concurrent hash tables: Fast and general?'' *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–2, Nov. 2016.

[64] J. Böttcher, V. Leis, T. Neumann, and A. Kemper, ''Scalable garbage collection for in-memory MVCC systems,'' *Proc. VLDB Endowment*, vol. 13, no. 2, pp. 128–141, Oct. 2019.

[65] T. Li, M. Butrovich, A. Ngom, W. Shen Lim, W. McKinney, and A. Pavlo, ''Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats,'' 2020, *arXiv:2004.14471*.

[66] Y. Huang, W. Qian, E. Kohler, B. Liskov, and L. Shrira, ''Opportunities for optimism in contended main-memory multicore transactions,'' *Proc. VLDB Endowment*, vol. 13, no. 5, pp. 629–642, Jan. 2020.

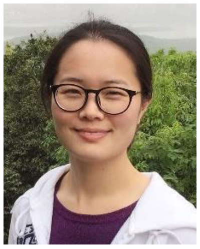
[67] S. Noll, J. Teubner, N. May, and A. Böhm, ''Analyzing memory accesses with modern processors,'' in *Proc. 16th Int. Workshop Data Manage. New Hardw.*, 2020, pp. 1–9.

[68] V. Leis, A. Kemper, and T. Neumann, ''Scaling HTM-supported database transactions to many cores,'' *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 2, pp. 297–310, Feb. 2016.

[69] D. Dice, M. Herlihy, and A. Kogan, ''Improving parallelism in hardware transactional memory,'' *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, pp. 1–24, Apr. 2018.

[70] M. Herlihy, J. Eliot, and B. Moss, ''Transactional memory: Architectural support for lock-free data structures,'' in *Proc. 20th Annu. Int. Symp. Comput. Archit.*, 1993, pp. 289–300.

[71] C. Jacobi, T. Slegel, and D. Greiner, ''Transactional memory architecture and implementation for IBM System z,'' in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2012, pp. 25–36.



ZHANG QIAN (Student Member, IEEE) received the B.S. and M.S. degrees in computer science and information engineering from Henan University, China, in 2012 and 2014, respectively. She is currently pursuing the Ph.D. degree with the School of Computer Science and Software Engineering, East China Normal University, China.

From 2014 to 2018, she was a Software Engineer with the Financial Technology Company of Hundsun, Hangzhou, China. Her current research interests include main memory database systems, distributed database, HTAP transactional analytical implementations, computer systems organization, data structure, memory access optimization techniques, and machine learning with database.

YIWEN XIANG, JR. was born in Shanghai, China, in 1999. She received the B.S. degree from the School of Computer Science and Software Engineering, East China Normal University, China, where she is currently pursuing the M.S. degree.

Her main research interests include concurrency control protocol, in-memory database systems, persistent hardware, and transaction management.

JIANHAO WEI was born in Nanchang, Jiangxi, China, in 1996. He received the M.S. degree from the School of Computer Science and Software Engineering, East China Normal University, China, in 2021, where he is currently pursuing the Ph.D. degree.

He has published one journal article in *Journal of Software*. His main research interests include multi-query sharing technology, in-memory database systems, persistent hardware, query optimization techniques, and machine learning with database.

CHUQIAO XIAO received the B.S. degree in computer science and technology from Tianjin Polytechnic University, in 2015, and the M.S. degree in computer technology from East China Normal University, in 2018, where she is currently pursuing the Ph.D. degree with the Software Engineering Institute.

Her research interests include distributed database, key-value storage systems, and erasure codes.