Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

A Novel Index-Organized Data Layout for Hybrid DRAM-PM Main Memory Database Systems

Qian Zhang
52184501012@stu.ecnu.edu.cn

East China Normal University

Xueqing Gong
East China Normal University

Jianhao Wei
East China Normal University

Yiyang Ren
East China Normal University

Research Article

Keywords: DRAM-PM, Main Memory Database System, Index-Organized, Hybrid Workload

Posted Date: October 16th, 2024

DOI: https://doi.org/10.21203/rs.3.rs-5249532/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

A Novel Index-Organized Data Layout for Hybrid DRAM-PM Main Memory Database Systems

First Qian Zhang, Second Prof. Xueqing Gong,
Third Jianhao Wei, Fourth Yiyang Ren

Software Engineering Institute, East China Normal University, Putuo North Zhongshan Road, Shanghai, 200062, China.

*Corresponding author(s). E-mail(s): 52184501012@stu.ecnu.edu.cn;
Contributing authors: xqgong@sei.ecnu.edu.cn;
52215902007@stu.ecnu.edu.cn; 51275902044@stu.ecnu.edu.cn;

Abstract

Large-scale data-intensive applications need massive real-time data processing. Recent hybrid DRAM-PM main memory database systems provide an effective approach by persisting data to persistent memory (PM) in an append-based manner for efficient storage while maintaining the primary database copy in DRAM for high throughput rates. However, they fail to achieve high performance under a hybrid workload because they are unaware of the impact of pointer chasing.
In this work, we investigate the impact of chasing pointers on modern main memory database systems to eliminate this bottleneck. We propose *Index-Organized* data layout that supports efficient reads and updates. We combine two techniques, *i.e.,* cacheline-aligned node layout and cache prefetching, to accelerate pointer chasing, reducing memory access latency. We present four optimizations, *i.e.,* pending versions, fine-grained memory management, Index-SSN, and cacheline-aligned writes, for supporting efficient transaction processing and fast logging. We implement our proposed data layout based on an open-sourced main memory database system. We conduct extensive evaluations on a 20-core machine equipped with Intel Optane DC Persistent Memory Modules. Experimental results demonstrate that *Index-Organized* obtains up to 3× speedup than the conventional data layouts, *i.e.,* row-store, column-store, and row+column.

Keywords: DRAM-PM, Main Memory Database System, Index-Organized, Hybrid Workload

1 Introduction

With the development of Non-Volatile Memory (NVM) technology, the practicality of byte-addressable persistent memory (PM) has become increasingly mature. Recent main memory database (MMDB) systems have designed in a hybrid DRAM-PM hierarchy [1–5]. These systems can benefit from the PM to achieve high performance, i.e., low latency reads and writes comparable to DRAM, but persistent writes and large storage capacity like an SSD.

However, modern MMDB systems cannot achieve high performance under hybrid workloads that include transactions that update the database while also executing complex analytical queries on this dataset [6–13]. For the hybrid workloads, some studies combine the advantages of row-store data layout and column-store data layout to design the row+column data layout [14–18]. However, we observe that modern MMDB systems face a new bottleneck in data access. Those systems have applied the optimization techniques to achieve a higher level of parallelism, e.g., Write-optimized index [19, 20], Multi-version concurrency control (MVCC) [21–23], Lock-free[20, 24, 25], Partitioning[26–28]. Unfortunately, such new designs implemented on very large, memory-resident, pointer-based data structures, result in complex read paths, leading to more pointer chasing and increased memory access latency. The bottleneck of data access in modern MMDB systems shifts to chasing pointers after the optimization techniques are applied.

In main memory systems, data structures are often implemented as pointer-based, e.g., nested arrays, linked lists. The relationships between different data items are maintained by using pointers. Pointer chasing is the fundamental behavior to traverse those linked data structures. Unfortunately, complex data access patterns are very difficult (if not impossible) for hardware prefetchers to predict accurately, leading to very high likelihood of last-level cache misses[1] upon pointer dereference. As a result, data stalls often dominate total CPU execution time [29–33].

In this work, we study the performance impact of pointer chasing on data layouts. The data access of an MMDB system typically consists of four steps: (1) a root-to-leaf traversal is always needed to search for a key in the tree-like table index; (2) lookup the indirection array to find the logical address of the key's version chain; (3) lookup the mapping table to find the physical blocks storing the HEAD of the version chain; (4) perform a linear traversal over the version chain until finding the target version. As shown in Fig.3, it is the massively expensive pointer chasing that hides the performance gap between the row-store, column-store, and row+column data layouts, thus heavily impacting the system's overall performance.

To overcome this problem, we present a novel index-organized data layout, *Index-Organized*. In summary, the contributions of this paper can be outlined as follows:

1. We observe that chasing pointers is becoming the new bottleneck in data access in current hybrid DRAM-PM main memory database systems. To the best of our knowledge, this paper is the first work to comprehensively study the performance impact of chasing pointers on the data layouts.

[1] Unless otherwise specified, throughout the paper cache misses refers to last-level misses that mandate accessing memory.

2. We design and implement an index-organized data layout for modern MMDBs. We combine two techniques to accelerate pointer chasing, i.e., *cacheline-aligned node layout* and *cache prefetching*.
3. We present four optimizations for support efficient transaction processing and fast logging, i.e., *pending versions*, *fine-grain memory management*, *Index-SSN*, and *cacheline-aligned writes*.
4. We experimentally analyze the performance of *Index-Organized* using a comprehensive set of workloads and show that *Index-Organized* outperforms conventional data layouts.

2 Background and Motivation

2.1 Typical Data Layout

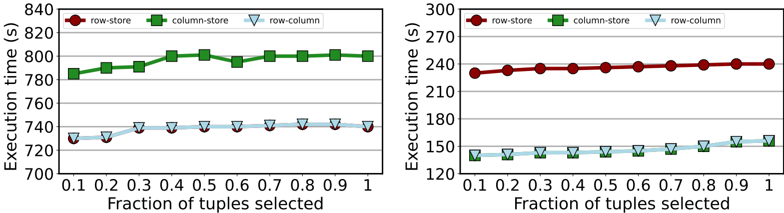

Fig. 1 Performance impact of the three typical data layouts(Projectivity=0.01) - Time taken by the execution engine to run workload 1(left) and workload 2(right).

For a relational table, there are mainly three data layouts: row-store [34, 35], column-store [36], and row+column [37]. In the row-store manner, all attributes of a record are stored consecutively on one page. It enables good performance for inserting large queries because records can be added to the table in the database by using a single write. In the column-store manner, each attribute of a record is stored on a different page. On one page, some type of attributes from different records are placed consecutively. For a query workload of accessing partial attributes, a column-store data layout is the best choice because it enables spatial locality. For a hybrid workload, some studies combine the advantages of row-store and column-store to design the row+column data layout.

In Fig. 1, we demonstrate previous research[37]. We construct a database with a single table $R(a_0,a_1,a_2,...,a_{500})$ that consists of 500 attributes. Each attribute a_k is a random integer value. We load 1 million records at initialization. We consider two queries: scan(*select* $a_0,a_1,a_2,...,a_k$ *from R where* $a_0 < \delta$) and insert(*insert into R values*($a_0,a_1,a_2,...,a_{500}$)). Additionally, we consider two workloads: (1) a heterogeneous workload of 1000 scan queries followed by 10 million insert queries, and (2) a read-only workload of 1000 scan queries. Note that the k and δ values affect queries' projectivity and selectivity, respectively.

As shown in Fig. 1(left), we observe that the row-store data layout outperforms the column-store data layout by up to 1.2×. As the number of insert queries increases, the

performance gap increases. This is because the column-store needs to split a record's attributes on every insert and store them in separate memory locations. For the scan queries, the results are shown in Fig. 1(right). We see that column-store executes the scan query up to 1.6× faster than row-store. This is because column-store improves the cache performance and bandwidth utilization by skipping unnecessary attributes.

2.2 Techniques to Absorb More Writes

As shown in Fig.2, in current MMDB systems, state-of-the-art techniques (*write-optimized index, lock-free structures, MVCC and data partitioning*) can significantly improve the write throughput but also complicate reads.

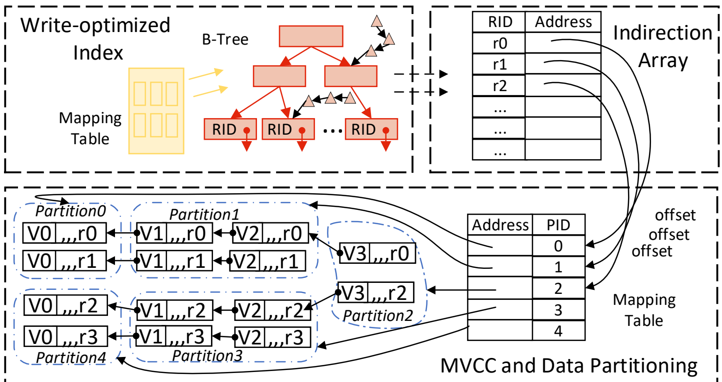

Fig. 2 In the modern MMDB systems, four state-of-the-art techniques to absorb more writes.

Write-optimized Index. To relieve the write amplification, researchers have proposed various modifications [10, 19, 20, 38–40] to the typical B-tree index. Among these, the Bw-tree's delta chain approach represents a notable advancement. A write to a leaf node or an internal node is appended to a delta chain, avoiding directly editing nodes. As the delta chains grow, they eventually merge with the tree nodes in a batched method, thereby reducing write amplification. However, long delta chains slow down data searches because more pointer chasing is needed to find the target key/value pair.

MVCC. Most main memory database systems have adopted *multi-version concurrency control* (MVCC) because updates never block reads[22]. The system maintains multiple physical versions of a record in the database to allow parallel operations on the same record. The versions of a record are managed using a linked list called *version chain*. During query processing, traversing a long version chain to find the required version is very slow because of pointer chasing, which will pollute CPU caches by reading unnecessary data[41]. Therefore, as the short-lived updates produce more and more versions, the length of the version chain increases, and the search structure and algorithms[42] could seriously limit the improvement of throughput performance.

Lock-free structures. Many main memory database systems [39, 43–45] implement data structures to avoid bottlenecks inherent in locking protocols. The widely used design involves an indirection layer that maps logical identifiers to physical pointers for the database's objects, e.g., indirection mapping table[20], indirection array[13, 46]. While this indirection allows the system to update physical pointers using compare-and-swap(CaS) atomically, it leads to degraded performance because it increases expensive pointer chasing, thereby resulting in a slower read path[39].

Data partitioning. Data partitioning can substantially improve the system performance[26, 47, 48]. It brings benefits of vectorized processing[49], which passes a block of records to each operator, thus achieving higher performance than the canonical tuple-at-a-time iterator model[50]. However, it adds additional cost to maintain volumes of data fragments(horizontal partitions), thereby increasing the database's data structure complexity[27], degrading the spatial locality of the data searches.

2.3 The Impacts of Pointer Chasing

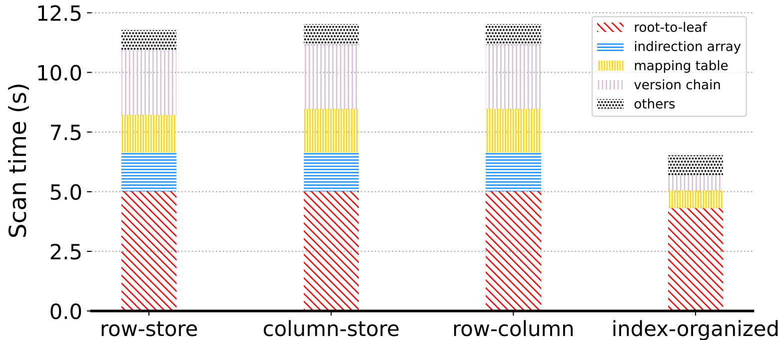

Fig. 3 Profiling results of pointer chasing codes vs. the rest of system codes. Time breakdown of various components of the MMDB system when running a hybrid workload.

Fig. 3 demonstrates the practical impact on performances of different data layouts on a real main memory database system[46]. For this experiment, we use the YCSB benchmarks[51] to build an HTAP workload. The hybrid workload consists of a write-only (100%update) thread and a read-only (100%scan) thread. We used a Zipfian distribution with a skew factor of 0.9 (80% of requests access 33% of records); We bulkload the database with 10K records (1K per tuple).

We observe that the column-store does not outperform the row-store across most scenarios. One reason for this gap was that the row-store reduced the record reconstruction cost. This eliminated unnecessary memory references and less pointer chasing, thus achieving a higher throughput rate. Second, the column-store benefits less from the inter-record spatial locality. Because complicated data access patterns result in massively expensive pointer chasing, which limits the performance of hardware prefetching, the cache-efficient column store could not have been better.

As shown in Fig.3, we also notice that the performance gap between the row-store and column-store was lower than shown in previous work. We observe that the performance gap was smaller under heterogeneous workloads, i.e., less than 1.2×. This is because, in this experiment, the complicated pointer-based data structures involved the long pointer-chasing sequences, which is a major practical problem that impacts throughput performance. Both row-store and column-store data layouts can not achieve high throughput performance. Due to its index organization and pointer chasing acceleration, index-organized scan exhibits much higher and more robust performance with growing chain lengths.

3 Characteristics of Modern Hardware

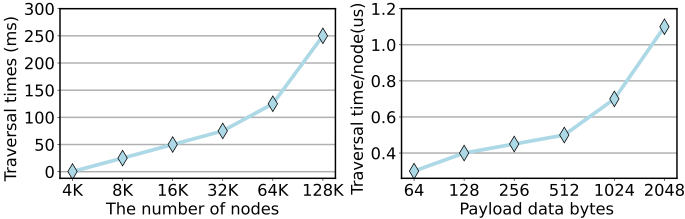

Fig. 4 Performance achieved traversing linked list containing pointers to the data payloads.

Table 1 Comparison of key characteristics of DRAM and PM(OPTANE DC PMMs).

		DRAM	**PM (OPTANE DC PMMs)**
Sequential	read latency(ns)	70	170
	read (6 channels)(GB/s)	120	91
	write (6 channels)(GB/s)	120	27
Random	read latency(ns)	110	320
	read (6 channels)(GB/s)	120	28
	write (6 channels)(GB/s)	120	6
Others	Addressability	Byte	Byte
	Access Granularity(bytes)	64	256
	Persistent	No	Yes
	Endurance (cycles)	10^{16}	10^{10}

In Fig. 4, we explore the pointer chasing of realistic systems on modern processors. In this experiment, we consider a simple reference behavior, namely a linked list traversal. We set several parameters for the reference behavior, including the number of nodes and the size of the data payload per node. Each node contains a "next" pointer and an indirect data payload that contains a pointer to the payload data. We separate the linked list nodes and data by 4 KB to defeat the cache block and prefetching. As shown in Fig. 4(left), we observe that the traversal time scales linearly with

the number of linked list nodes. We attribute this to spending more time retrieving each node payload from memory by pointer chasing. In Fig. 4(right), we notice a similar trend on the average traversal time per node with payload sizes varying from 64 Bytes to 2 KBytes. Over increasing payload sizes, the traversal time is dominated by the payload data fetch time through a reference pointer. This experiment highlights that the linked list traversal performance is very sensitive to pointer chasing due to its pointer-based characteristics and complex memory access patterns.

In Table 1, we summarize the characteristics of the memory storage used in the evaluation. Persistent memory (PM) is a new memory technology that bridges the gap between DRAM and flash-based storage by offering DRAM-scale latency as well as storage-like persistence [52]. Intel Optane DC Persistent Memory Module (Optane DCPMM) is the first commercially available PM product. We observe that Optane DCPMM has higher latency and lower bandwidth compared with DRAM. Both DRAM and Optane DCPMM show high random read latency. In the current MMDB system, data access patterns introduce large amounts of random memory accesses, notably impacting the efficiency of pointer-chasing operations. Therefore, to achieve high throughput, designers should optimize the data layout to minimize the number of random memory accesses.

4 Index-Organized

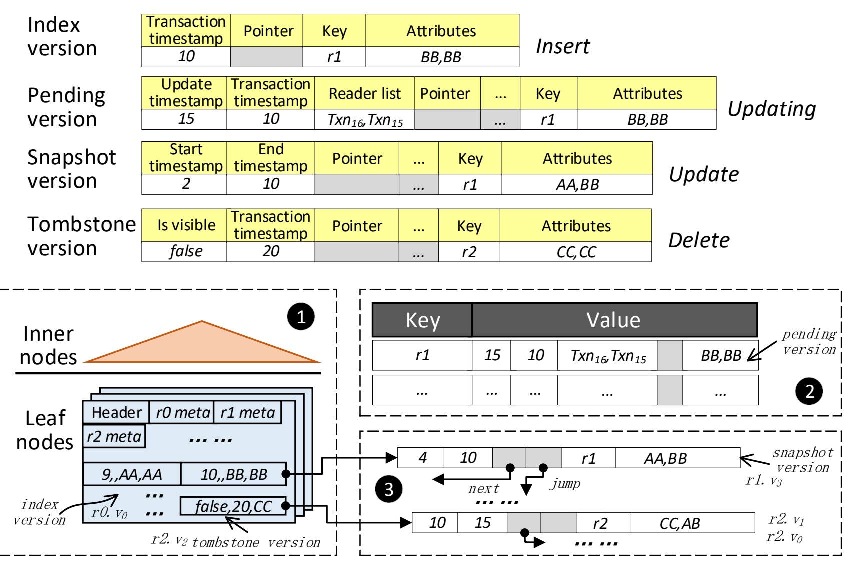

Fig. 5 Architecture of Index-Organized data layout and the version types of a logical record.

Under MVCC, a logical record corresponds to one or more *physically materialized* versions. These versions are organized as a singly linked list, which represents a version

chain. With the *New-to-Old ordering*, the chain's HEAD is the latest version, which refers to its predecessor. *Index-Organized* maintains the latest version of each record in the tree store and the older versions of the same record in the version store. Fig.5 shows the high-level architecture of *Index-Organized*. It consists of three parts: (1) the tree index for index versions and tombstone versions; (2) the mapping table for pending versions; and (3) the version store for snapshot versions.

Index-Organized can leverage tree indexes and cache-friendly layout to support efficient read and update operations. While this design can minimize the number of random memory accesses and improve the performance of pointer chasing, it comes with new concurrency, memory management, and resource utilization challenges. The rest of this section discusses the design of each component in detail, specifically, how we store versions in cache-friendly layouts and how we handle the concurrent operations. Finally, we present how we leverage cache prefetching to eliminate the increases in version search latency caused by pointer chasing operations.

4.1 Version Types

Index Versions are created upon the insertion of new records. Each index version consists of key and non-key attributes, which are stored together with the key attribute in a B-tree. The *pointer* (pointing to the predecessor) of the new version is included, as well as the *transaction timestamp* of the inserting transaction. The latter is essential for index-only visibility checks. For example, a transaction with *transaction timestamp 9* (Fig.5) inserts a new record (*r0*) in its initial version ($r0.v_0$), causing the creation of a *index version* in the leaf node of the tree. Generally, all of the table data can be held in its primary key index.

Pending Versions result from the updates on existing index versions. Such an update creates a new version that becomes the new HEAD of the chain, which needs to be reflected in the tree index. When an update transaction attempts to modify an existing record, it acquires a slot in the memory pool and copies the index version to this location. It then inserts the new version in the *mapping table* to logically replace the index version with the key and the version information (*update timestamp, create timestamp*). Before the update transaction commits, the concurrent transactions reading the record will be tracked in the *reader list*. Once the update transaction is committed successfully, the attributes of the index version will be changed, and the pending version will be evicted from the mapping table. For example, a transaction with *transaction timestamp 15* (Fig.5) updates the attributes of the record (*r1*), producing a pending version. Although the attributes remain unchanged, the version information of *r1* has to be updated, causing the creation of a *pending version* in the mapping table.

Snapshot Versions are no longer referenced by any other write operations. The snapshot versions are appended in the version store and are placed according to the new-to-old ordering. A snapshot version contains both the start and end timestamps that represent the lifetime of the version, together with its search key and the non-key attributes (Fig.5). When a transaction invokes a read operation on an existing record, the system searches for a visible version where the transaction's timestamp is in between the range of the *start timestamp* and *end timestamp* fields. For example,

a transaction with *transaction timestamp 10* (Fig.5) updates record (*r1*), creating a snapshot version ($r1.v_3$), modifying the attributes from 'AA, BA' to 'AA, BB'.

Tombstone Versions indicate the deletion of a record. If a record is logically deleted, it is not erased immediately in the tree index because it could be visible in a concurrent transaction. Rather, a tombstone version is inserted in the version chain, which needs to be reflected in the tree index. Tombstone versions mark the extinction of the whole version chain. A tombstone version contains the *is visible* flag and the transaction timestamp of the deleting transaction. For example, a transaction with *transaction timestamp 20* (Fig.5) deletes record (*r2*), creating a tombstone version ($r2.v_2$), reflecting deletion of the whole version chain $r2.v_2 -> r2.v_1 -> r2.v_0$.

4.2 Basic Operations

Insert. An insertion yields the creation of an *Index Version* in the tree index with the key attribute and non-key attributes of the newly created version and the timestamp of the creating transaction. The insertion first traverses the tree to locate the target leaf node, and checks the status word and the frozen bit of the leaf node. Then, it reads the occupied space and node size fields and computes the current spaces occupied. If there is enough free space available, it initializes a record meta and reserves storage space for the new version. Next, it copies the new version to the reserved space and updates the fields in the corresponding record meta, setting the visible flag to invisible and the state flag to in-inserting. When another insert is accessed, it may encounter this insertion. Then, the position will be rechecked until the in-process insert is finished. Finally, once the insert succeeds, the record-meta's visible flag field is set to visible, and the offset field is set to actual storage space offset.

Read. The read operation starts with searching the mapping table that may contain the target key and the record. If a pending version is found in the mapping table and is visible to the current transaction, we will return the version and terminate the read operation early. If the record is not found, we traverse the tree index to the leaf node. Afterward, the matching index version is requested and checked for visibility. Typically, a *read-for-update* operation searches for the *index version* in leaf nodes of the tree index, avoiding the complex retrieval of versions in the version store. For *read-only* operations, an index-only scan is sufficient. Once a leaf node is reached, both the key and the non-key attributes can be retrieved. This minimizes the number of random memory accesses needed to access a record. A long-running read transaction potentially needs to traverse a long version chain to find a visible version. *Cache prefetching* techniques are needed for looking ahead to the further version during the traversal of the version chain. Cache prefetching helps to speed up pointer-chasing operations and achieve good cache performance, improving the searches and scans.

Update. If a transaction attempts to update an existing record, it performs a *read-for-update* operation on the tree index. If finding the leaf node that contains the request key, it checks the record meta and ensures the record is not deleted. Then, within the leaf node, it replaces parts of the attributes that have changed using the updates. Before the replacement, it should do the following: (1) acquire a slot and copy the index version to the location, and then insert the *pending version* into the mapping table; (2) update the corresponding record meta, where the state flag field

will be set to in-processing to block other concurrent updates. Future readers can directly read pending versions from the mapping table. Once the update transaction is committed, the index version becomes visible, and the pending version is removed from the mapping table.

Delete. Like Update, deleting a record is essentially inserting a tombstone version into the tree index. When a read operation reads the tombstone in the leaf node, it returns a not-found result. When the tree index performs split/merge, the tombstone version is removed from the leaf node, if it exists.

4.3 Cacheline-aligned Node Layout

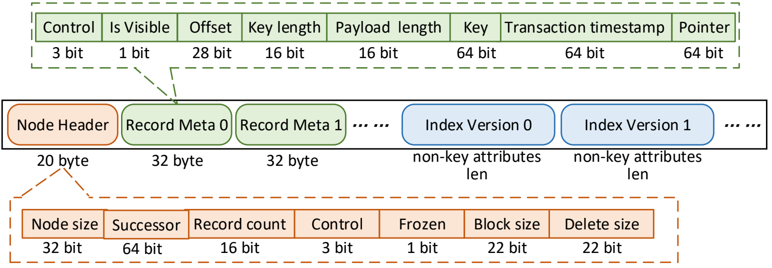

Fig. 6 Leaf node layout (64 bytes aligned).

Inner nodes and Leaf nodes share the same layout, storing *key-value* pairs in sorted/nearly-sorted order and allowing efficient lookups. The *key* is the record's indexed attribute. In the inner node, the *value* is a pointer to the children, while in the leaf node, the *value* is the *Index Version*, including the non-key attributes. Note that inner nodes are immutable once created and store *key-childpointer* pairs sorted in order by key. By default, the leaf nodes are fixed size (16 KB), storing the index versions in the nearly-sorted order.

As shown in Fig.6, the node layout starts with a 20 byte *Node Header*, which encodes the node size, successor pointer, record count (the number of child pointers/index versions on the node), control, frozen, block size and delete size. Control and frozen fields are used to flag in-process writes on the node. The block size field is used to compute the offset of the new index version in the free space. The deleted size field is useful for performing node merge.

The *Node Header* is followed by an array of *Record Meta*, which stores the keys and the metadata of the *key-value* pair. The *Record Meta* and the actual data are stored separately to support efficient searches. By storing the keys in a sorted array, the binary search can benefit from the prefetching to obtain good cache performance. Generally, a 16KB leaf node can hold more than 16 index versions; the *Record Meta* array are all 8×64 bytes. Before starting the binary search, we can leverage the great parallelism of the modern memory subsystem to prefetch the entire *Record Meta* array.

Furthermore, allowing the index versions to be nearly sorted shows efficient insertion, as we only need to sort out-of-order keys when splitting or merging the leaf nodes.

Each *Record Meta* in leaf nodes is 32 bytes, which stores the control and visibility of the key and index version and the offset of the index version in the node. It also stores the length of the key and index version, the key attribute, the transaction timestamp that creates the index version, and the pointer to the predecessors. Different from the leaf node, each *Record Meta* in the inner node is 16 bytes, not including the transaction timestamp and pointer fields.

4.4 Pending Versions and Mapping Table

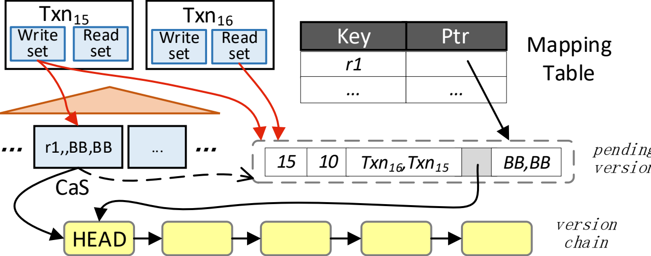

Fig. 7 Pending versions and mapping table.

Index-Organized uses *pending versions* to reduce index management operations and avoid contention while reducing the indirection cost of multi-version execution. *Index-Organized* maintains a mapping table that maps the search key to the physical location of the pending version.

When a transaction attempts to update an existing record, it first searches the index version of the record in the leaf node of the tree index, and then locks it by setting the control bit of the *Record Meta* to in-processing. Besides, it creates a pending version by acquiring a slot and copying the latest version to the location. If it successfully commits, it uses an atomic compare-and-swap (CAS) operation on the index version to change it to point to the new HEAD. Otherwise, if it is aborted, the pending version will be deallocated by the later reclamation. Before the update transaction commits, any concurrent transaction can read the record. The mapping table is first looked up to find the pending version of the record.

As shown in Fig. 7, the transaction Txn_{15} updates the record (*r1*) and creates a *pending version*. The pending version allows the *Index-Organized* to scale to high concurrency, as the concurrent transactions can access the visible version without blocking, such as transaction Txn_{16}. To accelerate the pending version searches, we use a mapping table to map the record key to the physical location of the pending version. We implement the mapping table using a hash table for its simplicity and

performance. To support transaction serializability, we leverage the mapping table to track the dependent reader transactions. The update transaction cannot be committed until all the transactions on which it depends have been committed.

4.5 Contention Management

In *Index-Organized*, a split is triggered once a node size passes a configurable maximal threshold, e.g., by default 16 *key-value* pairs. Likewise, a node merge is triggered once a node's size falls below a configurable minimal size. The problem is that split/merge operations and update transactions may contend with the latch of a leaf node, which will cause performance degradation.

Index-Organized implements optimistic latch-coupling[53] on tree nodes to reduce their contention based on the observation that although highly contended, nodes are rarely modified, i.e., split/merge. Specifically, we use an 8-byte status word (*record count, control, frozen, block size, delete size*) in the *Node Header* to check the state of the node. A split operation (write operation) will acquire an exclusive lock (*frozen*) on the node and bump the write lock after modification. If the split operation encounters an update transaction (read operation) by checking the *control* of the *Record Meta*, it will add the new node to the *successor* of the old node. The update transaction can read the *successor* to find the actual location of the accessed index version. If an update transaction encounters a node that needs splitting, it temporarily suspends its operation so that the split operation can be performed before continuing

4.6 Jump Pointers and Cache Prefetching

In current main memory database systems, version chains are generally implemented as linked lists, causing random access on traversals. A transaction, upon reading from the HEAD of the version chain, incurs random memory access until it finds the appropriate snapshot version in time.

Fig. 8 shows a version chain for a record that was updated multiple times. The version chain is organized in the new-to-old ordering. Since the transaction *Txn Y* starts with timestamp 2, it has to traverse the chain using the *next* pointer to retrieve the visible version v_1 with *start* and *end* timestamps (0,4). This is slow because of pointer chasing, and reading each version in a sequential access pattern introduces many cache misses. Furthermore, with many current updates, the number of active versions grows quickly, causing a long-tail version chain. The system may suffer from increased memory access latency caused by traversing a long version chain, leading to a significant drop in overall throughput performance.

Index-Organized solves this pointer-chasing problem by using the cache prefetching technique [29–31]. Specifically, we store a jump pointer field for each snapshot version, i.e., *jump* pointer, which points from a version to the one it should prefetch. For example, Fig. 8 shows how v_{i+2} could directly prefetch v_i using a two-ahead jump pointer. The goal is to ensure that by the time the traversal is ready to visit a version, that version is already prefetched into the cache, thereby hiding much miss latency of v_i. In this design, *Index-Organized* can support searching for the visible version in a parallel access pattern.

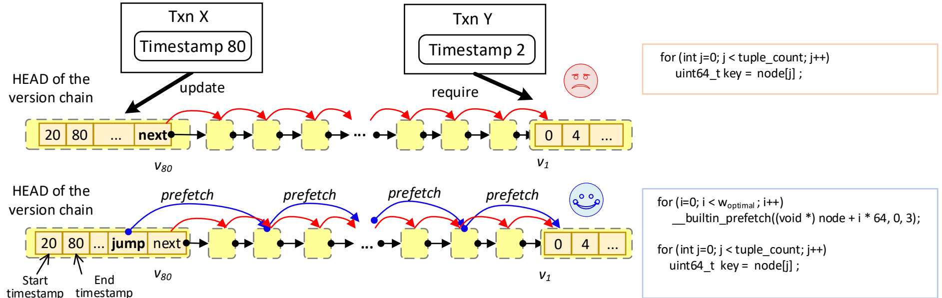

Fig. 8 Jump pointers and cache prefetching.

5 Optimizations

5.1 Fine-grained Memory Management

Valenblock. In *Index-Organized*, inner nodes are immutable once created and store *key-childpointer* pairs sorted in order by key. Since the inner nodes are read mostly and change less frequently, they are replaced wholesale by the newly created inner nodes when node splitting. While this benefits searches on the node, it needs variable-length memory space allocation. To overcome this problem, we design and implement a *valenblock-allocator* performing inner node storage space allocation. It includes two layers. In the first layer, we use a *chunk list* to maintain a list of fixed-size memory chunks. The *valenblock-allocator* requests a new chunk from the linked list, initializing its reference counter to 0, and adds it to the active chunk array. In the second layer, when an inner node is created, a thread acquires a small space from a random active chunk and increments its referenced count by one. When the inner node is released, the active chunk' referenced count decreases by one. A background thread periodically checks and recycles the active chunks once any inner node no longer references them.

Free-list. In the version store, snapshot versions are stored in fixed-size memory chunks aligned to cache-line size. This design significantly benefits table scan operations. The version store also maintains multiple *free lists*, each with a different size class, to track the recently de-allocated memory space of the snapshot version. A snapshot version de-allocation happens when it has no longer been referenced by any active transactions. Generally, a background thread periodically retrieve the global minimum from the global transaction list, gather the invalid snapshot versions at an ephemeral queue, and simultaneously unlink these versions from the associated transaction contexts. Finally, these de-allocated memory spaces are added to the *free lists* and reused for future version allocation.

5.2 Transaction Serializable

To ensure robust and scalable performance on heterogeneous workloads, *Index-Organized* adopts snapshot isolation concurrency control and provides serializability.

Accesses by transaction T generate serial dependencies that constrain T's place in the global partial order of transactions. There are two forms of serial dependencies: (1) T reads or overwrites a version that Ti created, with Ti dependency on T, T$i \leftarrow$ T; (2) T reads a version that Tj overwrites, with T an-dependency on Tj, T $\leftarrow$ Tj. Assuming a relation $\prec$ for a direct graph G, the serialization can also be defined as T$i \prec T \prec$ Tj, which the vertices are committed transactions and the edges indicate the required serialization ordering relationships. Then, for the case of T$i \xleftarrow{w:r/r:w}$ T or T$i \xleftarrow{w:w}$ T, Ti is a forward edge of T, and T$\xleftarrow{r:w}$ Tj, Tj is a backward edge of T. Considering a cycle in G T$i \prec$ T$j \prec$ Ti, that indicates a serialization failure. Therefore, to detect potential a cycle in the dependency graph G, two timestamps are associated with T: high watermarks, t_h_w, and low watermarks, t_l_w. The high watermarks can be acquired via all forward edges. The low watermarks can be acquired via all backward edges. When executing exclusion window checks for transaction T, if t_l_w≤t_h_w occurs, T must be abort.

This design, which we called Index-SSN, is inspired by SSN's design [54], where a similar goal is achieved. Algorithm 1 (Appendix A) shows the detailed implementation. Our key insight is to reduce the overhead for tracking transaction dependencies and cycle detection. We forbid the write/write dependency: T overwrites (T$i \xleftarrow{w:w}$ T) a version that Ti created. We design a state (1 bit) in the record meta to control updating over the tree index. We maintain the full version timestamps for the *pending version*, while the *index version* or *snapshot version* stores the created timestamp. We use the mapping table to track all overwriters and detect anti-dependency.

In the *READ* phase, the transaction t records the accessed version's create timestamp (cstamp) as the t_h_w the largest forward edge. It then searches the *mapping table* to check whether the concurrent transaction overwrites the version. If a concurrent transaction has overwritten the version, it records the version's sstamp as the t_l_w the smallest backward edge. Then, it verifies the exclusion window and aborts t if a violation is detected. If the version has not been overwritten, it will be added to the read set and checked for late-arriving overwrites during commit processing.

In the *COMMIT* phase, the transaction first produces a commit timestamp using a global counter incremented by the atomic fetch-and-add instruction. It first traverses the write set to calculate the high watermarks. Due to tracking all reads on the version's *reader list*, it just only needs to acquire the latest reader. If the reader r is found, it examines and waits r to acquire its commit timestamp. If the reader has committed, it updates the t_h_w using *r's* cstamp. It then traverses the read set to calculate the low watermarks. If a concurrent transaction has overwritten the version, it will detect the transaction in the tree index or the mapping table. It uses the transaction's timestamp to compute the t_l_w. Lastly, as t_h_w and t_l_w are both available, a simple check for **t_l_w** $\leq$ **t_h_w** identifies exclusion window violations. If yes, the transaction can be committed successfully. Otherwise, the transaction must be aborted.

5.3 Logging and Cacheline-aligned Writes

Persistent Memory is an emerging computer storage device with non-volatile and byte-addressable that can provide data persistence while achieving I/O throughput comparable to DRAM. However, as shown in Table 1, compared with DRAM, the

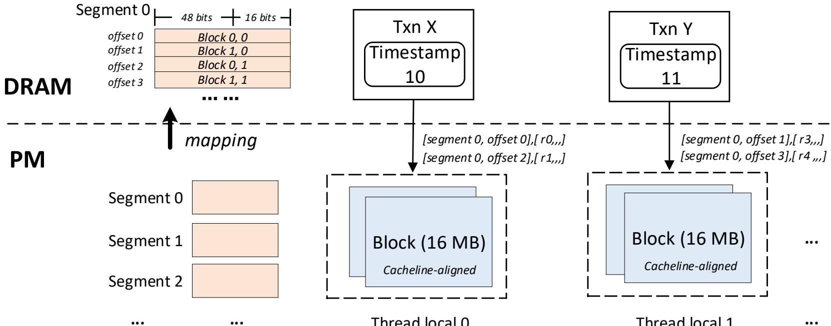

Fig. 9 LSN management and Log files

random read and write bandwidths of Optane DCPMM are much lower. To achieve high throughput, we perform sequential Optane DCPMM reads and writes.

Index-organized ensures recoverability by implementing the write-ahead logging [55]. A log record consists of the log sequence number(LSN), transaction context, log record type, and write contents. As shown in Fig. 9, we divide the PM log files into small-size (e.g., 16MB) pieces called blocks and sequentially write the log records to the thread local blocks. Each block includes many log records, which are cacheline-aligned, improving the write performance of PM by avoiding repeated flushing to the same cacheline.

In *Index-organized*, we leverage the physical segment file to generate the monotomic LSN. The LSN may not be contiguous but can be translated efficiently to physical locations on PM. Each LSN consists of two parts: the higher-order bits identify a physical segment file (e.g., *segment 0*), while the low-order bits identify an offset in the segment file (e.g., *offset 0*). Each segment file is 16KB by default, containing 2×1024 offsets (LSN) by default. We use dax-mmap to map a segment file residing on the PM device using the commands: *pmm_ptr = mmap((caddr_t) 0, len, prot_read | prot_write, map_shared, fd, 0))*. In this manner, we can use the resulting address to manage the LSN space directly.

A transaction can request LSN at any time, for example, during the execution with a valid log sequence number. The log manager provides the LSNs by incrementing a globally shared log offset by the size of the requested allocation. Then, after the transaction has finished writing the log records to the thread local block, the location of the log record stored in the block is inserted into the segment file offset as the value.

6 Evaluation

In this section, we perform experiments on *Index-Organized* with various workloads. To verify the efficiency of our solution, we compare it with the row-store, column-store, and row+column data layouts.

Implementation. We implement our *Index-Organized*[2] based on PelotonDB [14] with approximately 13,000 lines of C++ code. PelotonDB is an MMDB optimized for high-performance transaction processing [7], providing row-store, column-store, and row+column data layouts. Even though the Peloton project has died, many works researched based on PelotonDB continue [7, 23, 56–58].

In a row-store manner, we store all of the attributes for a single version contiguously. In a column-store manner, all attributes belonging to the first version are stored one after another, followed by all the attributes of the second version. In a row+column manner, we divide the columns into multiple column groups. We use the Index-SSN transaction concurrency control algorithm for all compared data layouts.

Evaluation Platform. We test our performance on a one-socket server with an Intel(R) Xeon(R) Gold 5218R processor (2.10 GHz, 40 logical cores, 20 physical cores, Cascade Lake) and OPTANE DC PMMs. It is equipped with 96 GB DRAM (6 channels*16 GB/DIMM) and 756 GB PM (6 channels*126 GB/PMM). We run Ubuntu with Linux kernel version 5.4 compiled from the source. All work threads are pinned to dedicated cores to minimize context switch penalties and inter-socket communication costs. For each run, we load data from the scratch and run the benchmark for 20 seconds. All runs are repeated for three times, we report the average numbers.

Workloads. We next describe the workloads that we use in our evaluation. These workloads differ with respect to skew and ratio of reads/writes.

(1)YCSB: This is a widely-used key-value store workload that is representative of the transactions handled by web-based companies [51]. It contains a single table composed of records with a primary key (8 bytes) and 10 columns of random string data, each 100 bytes in size. We mainly focus on lookup, update, and scan operations. We construct the following set of workloads mixture by varying the read ratios. **Scan-only:** 100% range scan; **Read-only:** 100% lookup; **Read-intensive:** 90% lookup, 10% update; **Write-intensive:** 10% lookup, 90% update; **Balanced:** 50% lookup, 50% update. The default access distribution is uniform distribution.

(2)TPC-C: This is the current standard benchmark for measuring the performance of transaction processing systems[59]. It is default write-intensive, with less than 20% read operations. It is also a dynamic adjustable workload. In this experiment, we use 100% of the New-Order transactions, with the number of warehouses set to 20.

(3)TPC-C-hybrid: To simulate heterogeneous workloads, we also use TPC-C to construct three transaction types: **TPC-C RA**, which is a 90% New-Order and 10% Stock-Level mix; **TPC-C RB**, which is a 50% New-Order and 50% Stock-Level mix; and **TPC-C RC**, which mixes New-Order with the Query2 in the TPC-CH benchmark[60]. CH-Q2 lists suppliers in a certain region and their items having the lowest stock level. Compared with Stock-Level, it reads more heavily. In this transaction, we set the ratio of CH-Q2 at 50% to measure the overall throughput and latency. In addition, because the majority of accesses of Stock-Level and CH-Q2 are in the *item* table and *stock* table, those transactions will frequently conflict with New-Order and each other.

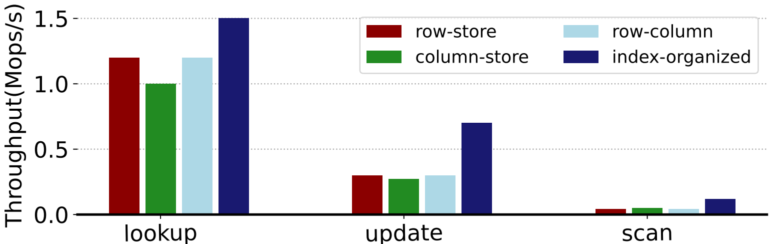

Fig. 10 Comparison of different data layouts with important workloads: lookup, update, and scan.

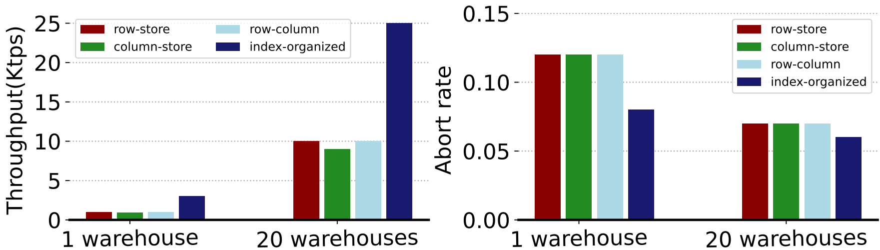

Fig. 11 Overall throughput and abort rate when running the TPC-C New Order transaction.

6.1 Overall Performance

In Fig. 10, this experiment examines how the four data layouts perform on the three most important workloads in practice: point lookup, update, and scan. We use the YCSB benchmark to set up these workloads and run each workload separately. Fig. 10 shows the throughput of the systems in these workloads with 20 threads. We see that *Index-Organized* outperforms the other three data layouts in all workloads. Row-layout, column-layout, and row+column all implement a heap-based organization that absorbs more writes, resulting in poor read performance. For range scan, *Index-Organized* achieves 3× higher throughput than row-store, column-store, and row+column layouts because it only needs to traverse the tree index.

In Fig. 11, we measure the overall throughput and abort rate in TPC-C with one warehouse and 20 warehouses, respectively, using 20 threads. Note that *Index-Organized* achieves up to 2× higher throughput than the other data layouts. New Order transactions are dominated by primary key accesses, i.e., lookup and update operations. *Index-Organized* clusters the index versions in the primary key order, thereby accelerating the lookup and update operations.

This section provides a high-level view of the system's performance on different workloads. It shows that *Index-Organized* can perform well in all representative workloads because it minimizes the number of random memory accesses.

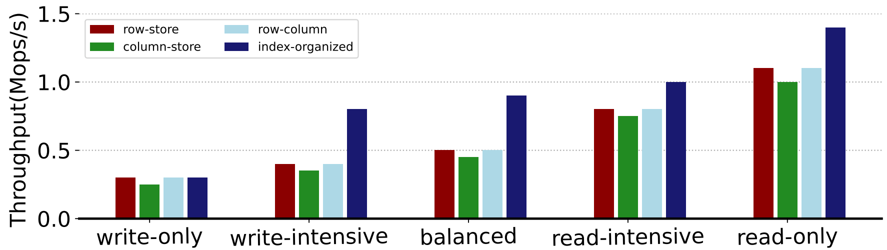

Fig. 12 Impact of read and write workloads. (YCSB)

6.2 Read Write Ratio

Fig. 12 shows the impact of read and write workloads on *Index-Organized* and other data layouts. The number of concurrent threads is 20. As expected, for all data layouts, the throughput is lower with more write operations. This is because we follow the "first-updater-wins" rule, so most write-write conflicts are detected if the head version is uncommitted. The doomed updater will abort immediately to avoid dirty writes, minimizing the amount of wasted work on aborts. Overall, *Index-Organized* performs significantly better than the conventional data layouts, i.e., row-store, column-store, and row+column. This is because *Index-Organized* tree index (including cache line-aligned node layout) can support index-only version searches and minimize the number of random memory accesses, leading to the best performance across all read and write ratios.

6.3 Scalability

This section examines the scalability of *Index-Organized* and conventional data layouts (i.e., row-store, column-store, and row+column) with YCSB and TPC-C-hybrid workloads.

Fig.13 uses the YCSB workloads and varies the number of work threads. We consider both uniform and Zipfian distribution and configure the skewness parameter of Zipfian distribution 0.9. All data layouts scale well before reaching the maximum physical cores. Among them, *Index-Organized* performs the best, with the lowest read starvation and the most efficient caching. Its efficient pending version handling also allows it to scale well without being bottlenecked by the concurrency control overhead. Under write-intensive workloads, conventional data layouts suffer from central memory heap management, achieving 1.5× lower throughput than *Index-Organized*.

Fig.14 shows the overall throughput and abort rate of TPC-C-hybrid workloads when varying the number of work threads. At low thread count, *Index-Organized* performs slightly better than conventional data layouts, as it avoids traversing the version chain overhead, specifically, for each primary key access, *Index-Organized* may have more than one less cache miss than conventional data layouts. At high thread count, *Index-Organized* achieves up to 1.6× higher throughput than conventional data

[2] https://github.com/gitzhqian/Stage-IndexOrganized.git

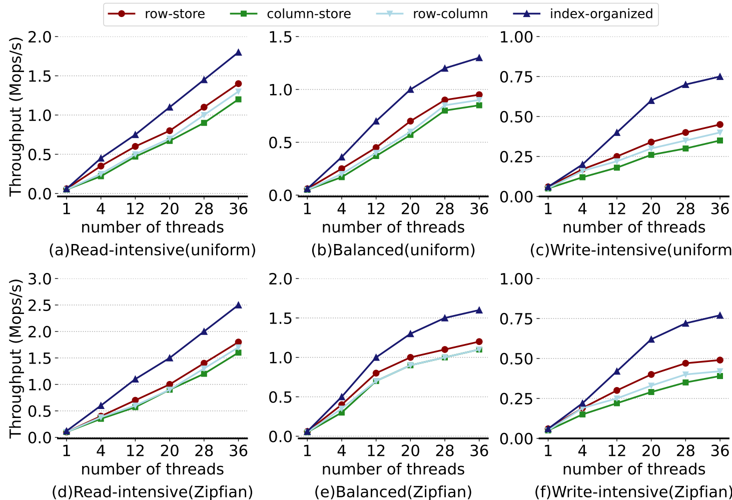

Fig. 13 The scalability of *Index-Organized* vs. conventional data layouts. (YCSB workloads, Uniform and Zipfian access distributions)

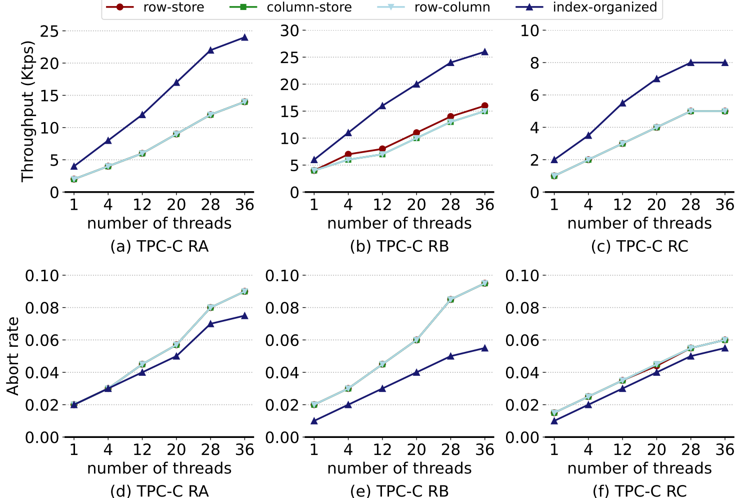

Fig. 14 The scalability of *Index-Organized* vs. conventional data layouts. (TPC-C-hybrid workloads, throughput and abort rate)

layouts, and has a 40% lower abort rate, indicating that *Index-Organized* is more efficient at handling contention.

Index-Organized systems achieve higher performance because their separate memory management structures effectively protect read-intensive transactions from updates. Furthermore, better performance gains also come from the index version and cache prefetching policy. This helps reduce memory latency and benefits transaction verification and commit processing. However, conventional data layouts employing

a heap-based storage scheme increase contention and make the version chains grow quickly. In contrast, they increase the memory access latency, which deteriorates the overall throughput performance.

6.4 Latency Analysis

In this section, we perform latency analysis. Fig. 15 shows the latency of TPC-CH-Q2 over a varying number of warehouses with 20 work threads. We calculated the average transaction latency within a run and presented the median value from three runs in the figure. At the low warehouse count, *Index-Organized* has a similar latency performance to the conventional data layouts. This is because when the database is small, the hot versions are more likely to be cached in CPU cache. The gap between *Index-Organized* and conventional data layouts is wider when the number of warehouses increases. Thanks to its index organization, *Index-Organized* have the lowest latency at 36 warehouses, almost 3× lower than conventional data layouts. Clearly, as the number of active versions increases, index-only scans gain importance because they can reduce unnecessary main memory accesses. However, for conventional data layouts, each version access may incur more than four random main memory accesses (tree searches, indirection searches, mapping table searches, and version chain searches).

Fig. 16 shows the flamegraph [61] of the *Index-Organized* for the TPC-C-RC workload. Tree index searches take roughly 70% of the time, involving multiple inner node searches and one leaf node search. Searching the leaf node first loads the record meta for keys, then binary searches the keys to find the request key, and finally searches the index version. Because the index versions are nearly sorted, each index version access first locates the offset and then uses the physical address to load the index version.

The mapping table searches spend 18.5% of the time. The read transaction always needs to access the mapping table to find the pending version, requiring only one memory access. The version chain searches use only 10.5% of the time. This is because an update transaction (New-Order) does not need to access version chains, while a long-running transaction (CH-Q2) can benefit from cache prefetching, spending less time accessing main memory.

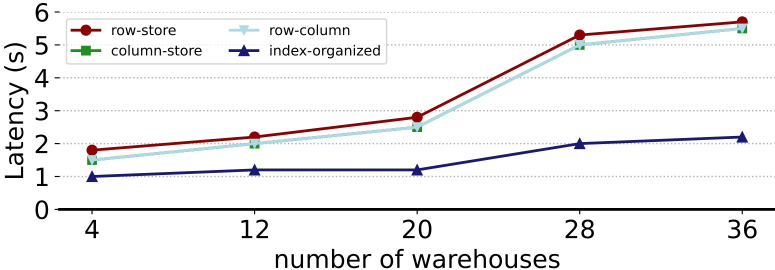

Fig. 15 Latency of CH-Q2 with the TPC-C-hybrid workload (TPC-C RC), varying the number of warehouses.

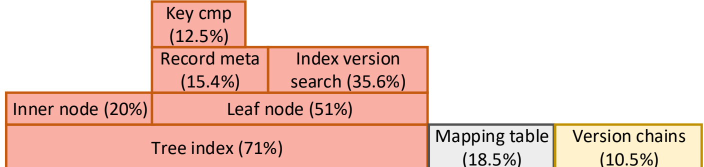

Fig. 16 Time spent on different components of *Index-Organized*-Recreated from the flamegraph for better clarity.

6.5 Version Chain Length

In this experiment, we examine the impact of version chain length. We run a long-running query (CH-Q2) and execute a varied number of short update transactions (New-Order). As the number of New-Order threads increases, the *stock* table will be frequently updated, quickly producing more and more versions. As seen in 4.6, the versions accumulate quickly over time, slowing down the readers. Fig.17 shows that the scan time of the CH-Q2 query thread is highly affected by concurrent New-Order transaction threads. As the version chain length increases, version scans slow down conventional data layouts by an order of magnitude. *Index-Organized*'s efficient caching mechanism, i.e., cacheline-aligned node layout and cache prefetching, reduces the pointer chasing overhead, hiding the latency. This makes its read performance superior to that of the other data layouts, which struggle because their version search pattern is largely ineffective for pointer chasing.

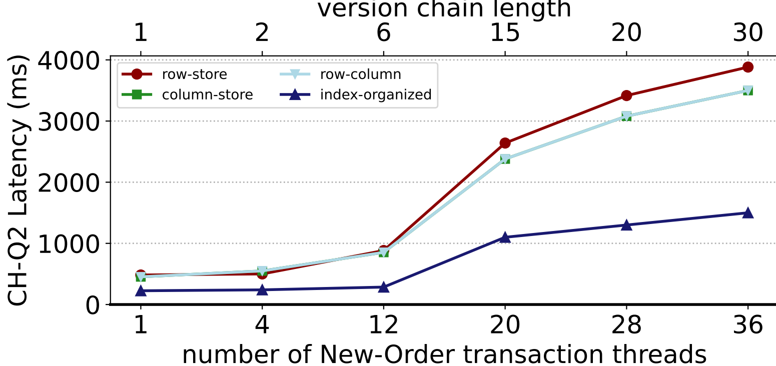

Fig. 17 Impact of version chain length - Increasing the number of New-Order transaction threads using one CH-Q2 query thread.

6.6 Impact of PM

To evaluate our design and implementations on the PM (Intel Optane DCP-MMs), we test microbenchmarks with YCSB insert-only benchmark using 20 work threads. We configure three logging modes: no logging (transactions do not write log records), logging-DRAM (transactions write log records to DRAM), and logging-PM (transactions write log records to PM), respectively.

As shown in Fig. 18, the overall throughput peaks when the duration is 10 s. This is because the server's memory utilization reaches 50% due to new versions being inserted continually. The memory usage contention becomes the limitation of the system's overall performance. In logging-PM mode, the *Index-Organized*'s throughput keeps growing slowly until the duration is in the 20s. We attribute this to the low write latency and PM's large capacity. In addition, during execution, we also use the **Intel Processor Counter Monitor (PCM)** library to collect hardware counter metrics[62]. The memory access plots in Table 2 confirm this behavior by showing DRAM writes and PM writes.

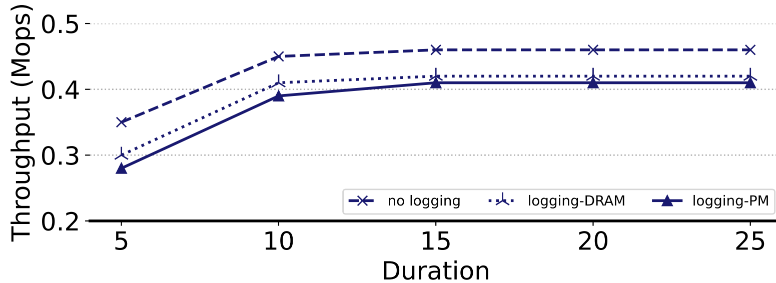

Fig. 18 YCSB insert-only performance on Intel Optane DCPMMs.

Table 2 Microbenchmarks For Insert-only with 20 work threads

	Metrics			
Duration Time(s)	***L2 hit ratio***	***L3 hit ratio***	***DRAM writes (bytes)***	***PM writes (bytes)***
5	0.74	0.90	27975517312	109719168
10	0.76	0.89	65096697088	498359552
15	0.76	0.90	95974431744	1901509120
20	0.75	0.90	127873320192	3732100224
25	0.75	0.89	160037408896	5578359936

7 Conclusion

As far as we know, this paper is the first to comprehensively discuss the impact of pointer chasing on data layouts for modern MMDBs. We observe that once state-of-the-art techniques have been applied, chasing pointers will become the new bottleneck

of data access in modern MMDB systems. In this paper, we propose *Index-Organized*, a novel storage model that supports index-only reads and efficient updates. We combine the cacheline-aligned node layout and cache prefetching to minimize the number of random memory accesses. To improve the update performance, we design and implement fine-grained memory management, pending versions, Index-SSN algorithm, and cacheline-aligned writes. Our evaluations show that *Index-Organized* outperforms conventional data layouts, achieving a 3× speedup for reads and a 2× speedup for transaction processing. In our future work, we aim to leverage machine learning algorithms to identify hot and cold data to improve logging and recovery further.

Acknowledgements. This work was supported in part by the National Natural Science Foundation of China under Grant 61572194 and Grant 61672233.

Author contributions. The authors confirm their contribution to the paper as follows: study conception and design: Qian Zhang and Xueqing Gong; software engineering, code development, and interpretation of results: Qian Zhang, Jianhao Wei, and Yiyang Ren; draft manuscript preparation: Qian Zhang and Xueqing Gong; all authors reviewed the results and approved the final version of the manuscript.

Declarations

Conflict of interest The authors declare no conflict of interest.

Appendix A Index-SSN Algorithm

References

[1] Zhou, X., Arulraj, J., Pavlo, A., Cohen, D.: Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In: Proceedings of the 2021 International Conference on Management of Data, pp. 2195–2207 (2021)

[2] Ziegler, T., Binnig, C., Leis, V.: Scalestore: A fast and cost-efficient storage engine using dram, nvme, and rdma. In: Proceedings of the 2022 International Conference on Management of Data, pp. 685–699 (2022)

[3] Ruan, C., Zhang, Y., Bi, C., Ma, X., Chen, H., Li, F., Yang, X., Li, C., Aboulnaga, A., Xu, Y.: Persistent memory disaggregation for cloud-native relational databases. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, pp. 498–512 (2023)

[4] Hao, X., Zhou, X., Yu, X., Stonebraker, M.: Towards buffer management with tiered main memory. Proceedings of the ACM on Management of Data **2**(1), 1–26 (2024)

[5] Liu, G., Chen, L., Chen, S.: Zen+: a robust numa-aware oltp engine optimized for non-volatile main memory. The VLDB Journal **32**(1), 123–148 (2023)

[6] Kissinger, T., Schlegel, B., Habich, D., Lehner, W.: Kiss-tree: smart latch-free in-memory indexing on modern architectures. In: Proceedings of the Eighth International Workshop on Data Management on New Hardware, pp. 16–23 (2012)

[7] Magalhaes, A., Monteiro, J.M., Brayner, A.: Main memory database recovery: A survey. ACM Computing Surveys (CSUR) **54**(2), 1–36 (2021)

[8] Raza, A., Chrysogelos, P., Anadiotis, A.C., Ailamaki, A.: One-shot garbage collection for in-memory oltp through temporality-aware version storage. Proceedings of the ACM on Management of Data **1**(1), 1–25 (2023)

[9] Yu, G.X., Markakis, M., Kipf, A., Larson, P.-Å., Minhas, U.F., Kraska, T.: Treeline: an update-in-place key-value store for modern storage. Proceedings of the VLDB Endowment **16**(1) (2022)

[10] Hao, X., Chandramouli, B.: Bf-tree: A modern read-write-optimized concurrent larger-than-memory range index. Proceedings of the VLDB Endowment **17**(11), 3442–3455 (2024)

[11] Xu, H., Li, A., Wheatman, B., Marneni, M., Pandey, P.: Bp-tree: Overcoming the point-range operation tradeoff for in-memory b-trees. In: Proceedings of the 2024 ACM Workshop on Highlights of Parallel Computing, pp. 29–30 (2024)

[12] Li, Z., Chen, J.: Eukv: Enabling efficient updates for hybrid pm-dram key-value store. IEEE Access **11**, 30459–30472 (2023)

[13] Kim, K., Wang, T., Johnson, R., Pandis, I.: Ermia: Fast memory-optimized database system for heterogeneous workloads. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1675–1687 (2016)

[14] Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T.C., Perron, M., Quah, I., *et al.*: Self-driving database management systems. In: CIDR, vol. 4, p. 1 (2017)

[15] Kemper, A., Neumann, T.: Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 195–206 (2011). IEEE

[16] Makreshanski, D., Giceva, J., Barthels, C., Alonso, G.: Batchdb: Efficient isolated execution of hybrid oltp+ olap workloads for interactive applications. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 37–50 (2017)

[17] NuoDB. 2020. https://nuodb.com/

[18] Alagiannis, I., Idreos, S., Ailamaki, A.: H2o: a hands-free adaptive store. In:

Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 1103–1114 (2014)

[19] Levandoski, J.J., Lomet, D.B., Sengupta, S.: The bw-tree: A b-tree for new hardware platforms. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 302–313 (2013). IEEE

[20] Wang, Z., Pavlo, A., Lim, H., Leis, V., Zhang, H., Kaminsky, M., Andersen, D.G.: Building a bw-tree takes more than just buzz words. In: Proceedings of the 2018 International Conference on Management of Data, pp. 473–488 (2018)

[21] Neumann, T., Mühlbauer, T., Kemper, A.: Fast serializable multi-version concurrency control for main-memory database systems. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 677–689 (2015)

[22] Wu, Y., Arulraj, J., Lin, J., Xian, R., Pavlo, A.: An empirical evaluation of in-memory multi-version concurrency control. Proceedings of the VLDB Endowment **10**(7), 781–792 (2017)

[23] Böttcher, J., Leis, V., Neumann, T., Kemper, A.: Scalable garbage collection for in-memory mvcc systems. Proceedings of the VLDB Endowment **13**(2), 128–141 (2019)

[24] Herlihy, M., Shavit, N., Luchangco, V., Spear, M.: The Art of Multiprocessor Programming. Newnes, ??? (2020)

[25] Kim, K., Wang, T., Johnson, R., Pandis, I.: Ermia: Fast memory-optimized database system for heterogeneous workloads. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1675–1687 (2016)

[26] Chevan, A., Sutherland, M.: Hierarchical partitioning. The American Statistician **45**(2), 90–96 (1991)

[27] Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 359–370 (2004)

[28] Bian, H., Yan, Y., Tao, W., Chen, L.J., Chen, Y., Du, X., Moscibroda, T.: Wide table layout optimization based on column ordering and duplication. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 299–314 (2017)

[29] Ebrahimi, E., Mutlu, O., Patt, Y.N.: Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In: 2009 IEEE 15th International Symposium on High Performance Computer Architecture, pp. 7–17

(2009). IEEE

[30] Hsieh, K., Khan, S., Vijaykumar, N., Chang, K.K., Boroumand, A., Ghose, S., Mutlu, O.: Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In: 2016 IEEE 34th International Conference on Computer Design (ICCD), pp. 25–32 (2016). IEEE

[31] Weisz, G., Melber, J., Wang, Y., Fleming, K., Nurvitadhi, E., Hoe, J.C.: A study of pointer-chasing performance on shared-memory processor-fpga systems. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 264–273 (2016)

[32] Ainsworth, S.: Prefetching for complex memory access patterns. Technical report, University of Cambridge, Computer Laboratory (2018)

[33] Huang, K., Wang, T., Zhou, Q., Meng, Q.: The art of latency hiding in modern database engines. Proceedings of the VLDB Endowment **17**(3), 577–590 (2023)

[34] Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill, Inc., ??? (2002)

[35] Codd, E.F.: A relational model of data for large shared data banks. Communications of the ACM **13**(6), 377–387 (1970)

[36] Copeland, G.P., Khoshafian, S.N.: A decomposition storage model. ACM sigmod record **14**(4), 268–279 (1985)

[37] Arulraj, J., Pavlo, A., Menon, P.: Bridging the archipelago between row-stores and column-stores for hybrid workloads. In: Proceedings of the 2016 International Conference on Management of Data, pp. 583–598 (2016)

[38] Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proceedings of the 7th ACM European Conference on Computer Systems, pp. 183–196 (2012)

[39] Arulraj, J., Levandoski, J., Minhas, U.F., Larson, P.-A.: Bztree: A high-performance latch-free range index for non-volatile memory. Proceedings of the VLDB Endowment **11**(5), 553–565 (2018)

[40] Kester, M.S., Athanassoulis, M., Idreos, S.: Access path selection in main-memory optimized data systems: Should i scan or should i probe? In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 715–730 (2017)

[41] Boroumand, A., Ghose, S., Oliveira, G.F., Mutlu, O.: Polynesia: Enabling effective hybrid transactional/analytical databases with specialized hardware/software co-design. arXiv preprint arXiv:2103.00798 (2021)

[42] Kim, J., Kim, K., Cho, H., Yu, J., Kang, S., Jung, H.: Rethink the scan in mvcc

databases. In: Proceedings of the 2021 International Conference on Management of Data, pp. 938–950 (2021)

[43] Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pp. 214–222 (1995)

[44] Wang, T., Levandoski, J., Larson, P.-A.: Easy lock-free indexing in non-volatile memory. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 461–472 (2018). IEEE

[45] Kelly, R., Pearlmutter, B.A., Maguire, P.: Lock-free hopscotch hashing. In: Symposium on Algorithmic Principles of Computer Systems, pp. 45–59 (2020). SIAM

[46] Peloton Database Management System. 2019. http://pelotondb.org

[47] Boissier, M., Daniel, K.: Workload-driven horizontal partitioning and pruning for large htap systems. In: 2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW), pp. 116–121 (2018). IEEE

[48] Al-Kateb, M., Sinclair, P., Au, G., Ballinger, C.: Hybrid row-column partitioning in teradata®. Proceedings of the VLDB Endowment **9**(13), 1353–1364 (2016)

[49] Polychroniou, O., Ross, K.A.: Towards practical vectorized analytical query engines. In: Proceedings of the 15th International Workshop on Data Management on New Hardware, pp. 1–7 (2019)

[50] Graefe, G.: Volcano/spl minus/an extensible and parallel query evaluation system. IEEE Transactions on Knowledge and Data Engineering **6**(1), 120–135 (1994)

[51] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154 (2010)

[52] Benson, L., Makait, H., Rabl, T.: Viper: An efficient hybrid pmem-dram key-value store (2021)

[53] Leis, V., Scheibner, F., Kemper, A., Neumann, T.: The art of practical synchronization. In: Proceedings of the 12th International Workshop on Data Management on New Hardware, pp. 1–8 (2016)

[54] Wang, T., Johnson, R., Fekete, A., Pandis, I.: Efficiently making (almost) any concurrency control mechanism serializable. The VLDB Journal **26**, 537–562 (2017)

[55] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks

using write-ahead logging. ACM Transactions on Database Systems (TODS) **17**(1), 94–162 (1992)

[56] Wu, Y., Guo, W., Chan, C.-Y., Tan, K.-L.: Fast failure recovery for main-memory dbmss on multicores. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 267–281 (2017)

[57] Lee, L., Xie, S., Ma, Y., Chen, S.: Index checkpoints for instant recovery in in-memory database systems. Proceedings of the VLDB Endowment **15**(8), 1671–1683 (2022)

[58] Wang, H., Wei, Y., Yan, H.: Automatic single table storage structure selection for hybrid workload. Knowledge and Information Systems **65**(11), 4713–4739 (2023)

[59] Tpc. Tpc Benchmark C (oltp) Standard Specification, Revision 5.11,2010. Available At. [Online]. http://www.tpc.org/tpcc

[60] Funke, F., Kemper, A., Neumann, T.: Benchmarking hybrid oltp&olap database systems (2011)

[61] Gregg, B.: The flame graph. Communications of the ACM **59**(6), 48–57 (2016)

[62] Intel Corporation et Al. Processor Counter Monitor. 2019. https://github.com/opcm/pcm

Algorithm 1 Index-SSN Algorithm (*Read* and *Commit*)

```
Require: t_h_w =∞, t_l_w = 0.
 1: function serializable_read(t, v, cstamp)
 2:     //1.update T high watermarks, Ti ←w:r-T
 3:     t_h_w = max(t_h_w, cstamp)
 4:     mapp_version = mapping_ table(v.rcd_meta.ptr)
 5:     if mapp_version != null then
 6:         if mapp_version.endstamp is not infinity then
 7:             //2.update T's low watermarks with T ←r:w-Tj
 8:             t_l_w = min(t_l_w, mapp_version.sstamp)
 9:         end if
10:     end if
11:     if t_l_w <= t_h_w then abort.
12:     end if
13:     Add v to t_readset return true.
14: end function
15: function serializable_commit(t)
16:     cmm_stamp = counter.fetch_add(1)
17:     //1.traverse t_wrset, update T's high watermarks
18:     //find the Ti ←r:w-T, forward edges
19:     for v in t_wrset do
20:         for reader r in mapp_version.read_list do
21:             if (r_txn = glb_act_t.find(r)) then
22:                 if r_txn.cmm_stamp<cmm_stamp then
23:                     while r_txn is finished do
24:                         t_h_w = max(t_h_w, r_txn_cmm_)
25:                     end while
26:                 end if
27:             end if
28:         end for
29:     end for
30:     //2. traverse t_rdset, update T's low watermarks
31:     for v in t_rdset do
32:         //find the T ←r:w-Tj, backward edges
33:         r = v
34:         if r.cs!=v.rcd_meta.cstamp then
35:             u = v.rcd_meta.cstamp
36:             u_txn = glb_act_txn.find(u.cstamp)
37:             t_l_w = min(t_l_w, u_txn.t_l_w)
38:         else
39:             mapp_version_loc = v.rcd_meta.ptr
40:             u = mapp_table(mapp_version_loc)
41:             if (u_txn = glb_act_txn.find(u.cstamp)) then
42:                 while u_txn is finished do
43:                     t_l_w = min(t_l_w, u_txn.t_l_w)
44:                 end while
45:             end if
46:         end if
47:     end for
48:     if t_l_w <= t_h_w then abort.
49:     end if
50: end function
```