Tabular: Efficiently Building Efficient Indexes

Ziyi Yan
Simon Fraser University
ziyi_yan@sfu.ca

Mohamed Farouk Drira
Simon Fraser University
mfd4@sfu.ca

Tianxun Hu
Simon Fraser University
tha110@sfu.ca

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

ABSTRACT

Concurrent indexes are hard to build by requiring complex, careful yet error-prone processes of design and implementation. As prior work has observed, modeling indexes as transactional tables can largely ease programming. The developer only needs to write single-threaded logic without worrying about concurrency or persistence, which are transparently supported by ACID table operations. However, this was deemed infeasible due to high overheads caused by the underlying OLTP engine.

In this paper, we argue that by adapting recent OLTP techniques which have been shown to deliver unprecedented performance, this idea is now feasible. We propose Tabular, a new library for building efficient indexes by modeling indexes as ACID tables which provide concurrency and persistence transparently. We elaborate the design of Tabular and its use cases. Our evaluation shows that compared to hand-crafted ones, indexes built using Tabular provide competitive performance with improved programming efficiency.

PVLDB Reference Format:
Ziyi Yan, Mohamed Farouk Drira, Tianxun Hu, and Tianzheng Wang. Tabular: Efficiently Building Efficient Indexes. PVLDB, 18(6): 1991 - 2004, 2025.
doi:10.14778/3725688.3725721

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sfu-dis/tabular.

1 INTRODUCTION

Database systems rely on concurrent indexes [3, 20, 47, 53, 61] for high performance. They need to be optimized for modern hardware as represented by multi-core processors, large DRAM and fast SSDs. To leverage the massive parallelism of modern CPUs, it is necessary to employ parallel programming techniques to facilitate multi-threaded accesses. The typical development cycle follows a *hand-crafted* approach that starts with a single-threaded proof-of-concept, which is then gradually added with more features and the necessary support for concurrency and persistence. After that, the data structure goes through multiple iterations of debugging and tuning to meet the quality and performance requirements.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725721

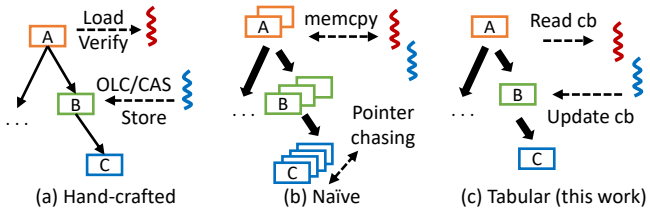


Figure 1: Hand-crafted indexes (a) are fast with in-place accesses but hard to build. (b) Modeling indexes as tables eases programming, but is prohibitively slow due to multi-versioning. (c) Tabular reduces overheads with single-version, callback (cb) based operations, while easing programming.

1.1 Index Programming Efficiency

Hand-crafted indexes can yield very high performance. As Figure 1(a) shows, however, they require DBMS developers use various parallel programming techniques, leading to notoriously low programming efficiency (i.e., they are hard to build) [14, 67],

First, the developer needs to make design decisions on synchronization. The classic solution was lock coupling [4] which locks both reads and writes pessimistically while traversing and updating an index (e.g., B+-tree). With more CPU cores, optimistic [52] and lock-free [57] approaches become more desirable since they allow reads to proceed without taking any locks. Yet, realizing this requires a mindset shift for developers to take account retry logic in various locations of the code. As we show later, this translates into various branching conditions that would not have existed in traditional pessimistic lock-based or single-threaded index implementations. Lock-free approaches [57] can get more complex; even replicating the effort itself is a significant research topic [91].

Second, the choice of optimistic and lock-free concurrency significantly complicates memory management. Unlike lock-based programming where the lifecycle of memory (e.g., B+-tree nodes) are aligned with critical section boundaries, memory blocks can continue to play an active role even after they have been retired from the data structure. For example, while one thread has finished splitting a B+-tree leaf node, another concurrent thread may still be using the old version as reads are not locked. The old node cannot be recycled until the oldest thread has finished using it. Handling such cases requires additional efforts on implementing approaches such as hazard pointers [68] and epoch-based reclamation [30], both involve delicate parallel programming techniques to get right. On top of these, developers also need to consider optimizations for storage (e.g., read/write asymmetry [56, 57]).

Finally, indexes have to evolve as hardware evolves, mandating index developers continuously customize their solutions and more tediously, repeat them for different indexes that already exist.

Overall, the need for fast indexes mandates DBMS developers to be also experts in neighboring areas, such as synchronization [52, 55, 79], memory management [74, 78], storage devices [15, 40] and networking [96]. Having to delve deep into other areas exacerbates the challenges faced by DBMS developers: the learning curve is steeper, the time-to-market is longer, code is more complex and harder to debug and maintain, and the "better" solution may turn out to be suboptimal [21, 91], causing confusion and wasted effort.

1.2 Tabular

This paper proposes Tabular, a new parallel programming library to ease index programming by revisiting the classic idea of objects-on-DB [41] with necessary optimizations to bring competitive performance. The gist is to model indexes as transactional tables that transparently provide ACID guarantees. For example, B+-tree nodes can be backed by a table whose schema defines the B+-tree node class members as table columns. Tree operations are programmed as Tabular transactions. This way, concurrency and persistence are handled transparently. The developer only needs to (1) model index components (e.g., B+-tree nodes) as table records, and (2) write sequential code that performs transactional record accesses.

Realizing this idea requires high-performance OLTP techniques. Early work [5, 41] has seen limited adoption and does not directly meet the need for today's high-performance indexes. As summarized in Figure 1(b), the first key culprit is excessive memory copying. Compared to Figure 1(a) where a hand-crafted B+-tree performs in-place accesses, a naïve object-on-DB design accesses records via the DBMS. Data has to be copied from tables to the transaction's local buffers; this is not needed by a hand-crafted index at all. To avoid excessive memory copying, Tabular presents a zero-copy, callback-based data access approach, inspired by prior work [35]. As Figure 1(c) shows, instead of performing traditional copy-based transactions, the index code provides a callback function that specifies the operation to be performed on the target data record. We also apply this technique to allow in-place updates on data records, benefiting a wide range of index operations.

While multi-versioning has been proven to benefit "real" database applications [92], it brings unnecessary overheads for indexes. For example, record versions are often chained in linked lists [16, 44, 59], so B+-tree nodes are transparently multi-versioned and accessing a node can incur pointer chasing operations, which are known to be a major source of memory stalls [39]. To solve this problem, we depart from the de facto standard multi-versioned approaches and advocate (1) single-version and (2) decentralized optimistic concurrency control (OCC) [45, 85]. This allows Tabular to eliminate unnecessary memory copying and pointer chasing overheads. We devise a set of techniques to work with callbacks while guaranteeing serializability. For example, classic decentralized OCC sorts the write-set upon precommit to avoid deadlocks, but this may lead to incorrect ordering for callback-based updates. We tweak the OCC protocol to solve this problem (details later).

Tabular uses parallel redo-only logging [85, 93] for durability. The developer only needs to indicate in a flag in Tabular whether persistence is needed, allowing (almost) effortless transformation between a volatile and persistent index using the same code.

Importantly, our goal is *not* to improve index performance beyond the state-of-the-art. Rather, we make it easier to build indexes with competitive performance. Our evaluation shows that on a dual-socket, 48-core server, B+-trees, tries and extendible hash tables built using Tabular can achieve up to ~95% of the performance of their hand-crafted counterparts. Using cognitive complexity,[1] we show that the code of Tabular-based indexes is similarly understandable to sequential implementations and is 41.8% less complex than hand-crafted concurrent indexes. We also integrated Tabular-based indexes into ERMIA [44], a representative memory-optimized OLTP engine. The resulted engine delivers comparable TPC-C performance to ERMIA, highlighting that Tabular-based indexes satisfy the performance need of modern OLTP engines.

This paper focuses on indexes as they directly impact DBMS performance and are complex enough to require non-trivial effort. But the applicability of Tabular goes beyond indexes. For example, it could be used to build queues and linked lists, which are building blocks for lock managers. Tabular's transaction and table abstractions are generic: they can be used to implement user-level transactions in an OLTP engine; we leave it as future work.

1.3 Contributions

This paper makes four contributions. ❶ We revisit the idea of mapping indexes to tables and make the case for its potential in improving programming efficiency while maintaining performance. ❷ We propose Tabular, a new parallel programming library to make the idea practical. Tabular is widely applicable and can be easily integrated into existing systems. ❸ We adapt popular indexes (B+-tree, extendible hashing and ART) to showcase the effect of Tabular. ❹ We provide a comprehensive evaluation of Tabular and other approaches under microbenchmarks and end-to-end benchmarks. Tabular is open-source at https://github.com/sfu-dis/tabular.

2 BACKGROUND

This section introduces the necessary background and motivates our work by identifying research challenges.

2.1 Concurrent Indexes and Programmability

Without losing generality, we use memory-optimized B+-trees as a running example to elaborate the programming efficiency issues in existing hand-crafted indexes.

Synchronization. A classic approach is lock[2] coupling [4]. Each B+-tree node is protected by a lock. To access a node, the thread explicitly acquires the lock in shared or exclusive mode. A node can be unlocked once no modification will be required. To split, we need to lock both the leaf and parent nodes in exclusive mode. However, the thread can only tell whether the leaf needs to be split after locking it, while the parent has been locked in shared mode. So the parent lock needs to be upgraded to exclusive mode, which is tricky to get right and can cause deadlocks when another thread holding a shared lock on the same parent is traversing down [76].

[1] A complexity metric that measures the difficulty level of understanding code [7].
[2] Also known as *latches* in database literature to differentiate from logical-level transaction locks; we use both terms interchangeably as our focus is physical-level indexes.

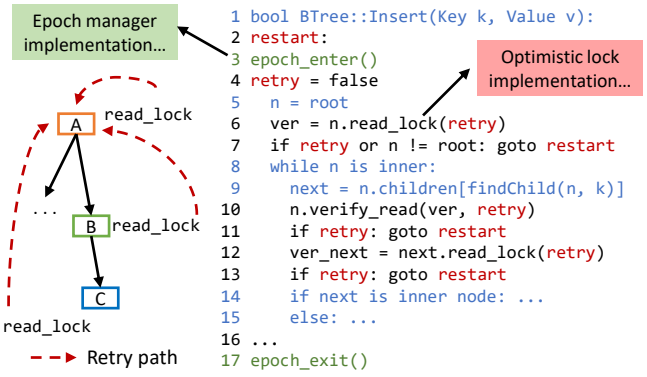
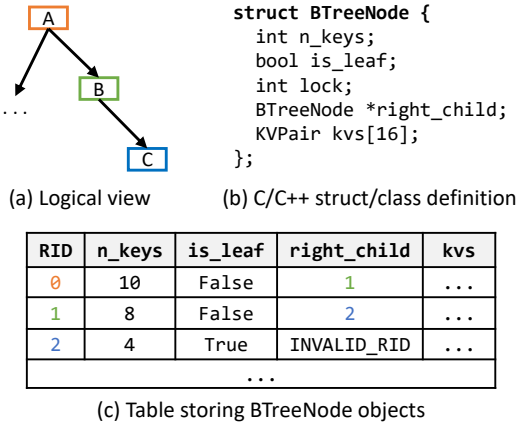


Figure 2: Compared to sequential logic (highlighted in blue), developers spend non-trivial extra effort on implementing optimistic locks, OLC protocols and memory reclamation.



RID	n_keys	is_leaf	right_child	kvs
0	10	False	1	...
1	8	False	2	...
2	4	True	INVALID_RID	...
...				

(c) Table storing BTreeNode objects

Figure 3: Modeling data structure objects on tables [5, 41]. Columns represent class fields and rows represent B+-tree nodes. C/C++ pointers are turned into record IDs (RIDs).

Such pessimistic approach also incurs shared memory writes even for read-only accesses for changing lock state.

More recent indexes use lock-free algorithms [57] or optimistic lock coupling (OLC) [55]. It is well-known that the former is challenging to design, debug and may not be optimal [21, 91]. An optimistic lock [11, 52, 79] carries a version number and lock state. Writers proceed as usual, except they also increment the version number after each round. Readers proceed without setting a shared lock state but must verify the version number did not change after the read. Compared to sequential code, optimistic locking is still much more complex. As Figure 2 shows, one has to tweak lock coupling with verification retry logic. Moreover, it is not uncommon for DBMS developers to propose new locks [79] for desirable features. Memory management also becomes non-trivial, resembling that in lock-free code since readers are not protected. Nodes unlinked from the tree must be kept available until all existing readers have finished. As a result, the index developer also has to implement memory reclamation algorithms [30, 68], or sometimes, propose new ones [58]. Each of these algorithms comes with its own trade-offs, requiring a DBMS developer to delve deep into another area, lowering programming efficiency and increasing code complexity.

Persistence. It can be desirable to persist large indexes to reduce service downtime. In storage-centric DBMSs, B+-tree nodes are database pages managed by the buffer pool. However, these pages are merely data containers: developers still need to consider concurrency issues. Recent memory-optimized DBMSs paid less attention to index persistence. Many assume indexes are volatile and must be rebuilt upon recovery; some use checkpointing and logging [48, 57]. More recent work such as Bf-Tree [28] allows more flexible caching granularity for on-disk pages. As a result, indexes are coded up with parallel programming techniques and considerations for managing I/O and memory. These quickly get complex, lowering programming efficiency and adding maintenance effort.

Given the additional complexity, it would be desirable to free developers from customizing features beyond the index logic itself. As we show later, Tabular allows developers to focus on sequential index logic, reducing the programming complexity to what is similar to programming sequential indexes.

2.2 Mapping Indexes to Tables

We follow the model as defined by prior work [5, 41] where a schema defines the data structure's class definitions.

High-Level Concepts. As Figures 3(a–b) show, a hand-crafted concurrent B+-tree would define a `BTreeNode` class, specifying members such as the number of keys. Under objects-on-DB, such class is defined by a schema, with the same fields in `BTreeNode`, except two notable changes. (1) `lock` is not needed because concurrent accesses are governed by transactions. (2) Pointers (e.g., `right_child` which points to the right-most child node if `is_leaf` is false) become record IDs (RIDs). To create a new node, the developer inserts a new record to the table in Figure 3(c), whereas a hand-crafted variant would use a memory allocator. To traverse the tree, we start with reading the root node record, and then follow the child RIDs to find other nodes.

Benefits. Since each B+-tree record is a database record, with proper ACID support, the B+-tree developer only needs to write sequential B+-tree logic. Persistence is also a given, without additional code. This drastically improves programming efficiency and code maintainability. The index is defined by a schema, so its internals can be easily queried by database operations, simplifying debugging. For example, to inspect the data stored in B+-tree nodes, the developer can directly issue database queries over the tables—exactly in the same way as querying other "normal" tables—to learn about the data structure. One can also easily duplicate, migrate or compact such data structures using DBMS table operations. We focus on leveraging the first benefit (transparent concurrency and persistence) and leave it as future work to explore other benefits.

Programming Interfaces. With objects-on-DB, instead of hand-crafting class definitions and index operation functions, DBMS developers (1) define data structure schemas and (2) devise transactions that perform index operations. Off-the-shelf DBMSs already provide such functionality, but they are usually declarative (e.g., in SQL) and do not fully satisfy the need of building indexes which are inherently imperative. Therefore, we envision a mostly imperative interface that combines schema definitions and table record

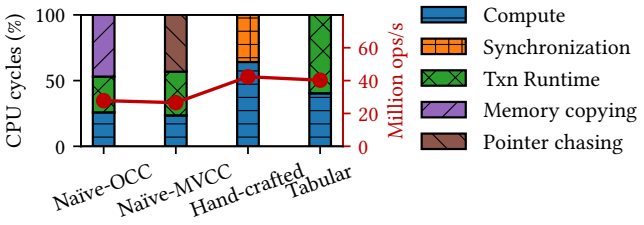


Figure 4: CPU cycles breakdown of hand-crafted vs. object-on-DB indexes. Naively mapping indexes on an existing OLTP engine spent almost half the CPU cycles on memory copying.

operations. An end-to-end objects-on-DB library can expose such an interface in common imperative programming languages (e.g., C++) or a domain-specific language for DBMS developers. Through an additional compiler pass (transpilation), we can generate code that includes (1) C/C++ definitions transformed from the schema, and (2) functions that directly use the interfaces provided by an OLTP engine. Records then become schema agnostic and in essence are raw bytes that can be cast to the defined classes. Certain aspects can be made declarative to further simplify programming. In the rest of this paper, we focus on the design of such an OLTP engine.

2.3 Modern OLTP and Motivation

Making the idea of index-on-DB practical requires the underlying OLTP engine to perform well. Although early work [5, 41] in the 1990s explored various optimizations, it was rarely adopted mainly due to the prohibitively high overhead of using a complex full-fledged DBMS. Fast forward to today, modern OLTP engines [16, 43–45, 71, 85] have optimized away many overheads in concurrency control [23, 72, 85, 89, 92, 95] and storage management [25, 38, 51, 78, 88]. It is common for them to deliver such high throughput (e.g., >10MTPS TPC-C [84]) that is even beyond practical needs [75]. This led us to ask "*Is it now feasible to build fast-enough data structures such as indexes with objects-on-DB on a modern OLTP engine?*"

To answer this question, we started with building a B+-tree on the ERMIA [44] memory-optimized OLTP engine. Like other engines, ERMIA's core is an embedded library that exposes C/C++ interfaces which we directly use to build indexes. This allows us to cut the unnecessary overhead of parsing and optimization, but the result still presents a significant performance gap between hand-crafted counterparts by being up to over ~80% slower (Section 6).[3] As Figure 4 shows (denoted as "Naive-MVCC"), close to 50% of the CPU cycles are spent on pointer chasing, required by the multi-versioning design commonly found in today's OLTP engines. We also tested the idea without using multi-versioning but uses single-versioned optimistic concurrency control (OCC) [85], as shown by the "Naive-OCC" bar in Figure 4, memory copying overhead takes over as the biggest bottleneck. Also, both approaches have transaction runtime overhead (e.g., for initializing transaction contexts and committing transactions). None of these exist in the hand-crafted counterparts. Overall, these issues still leave much room for optimizations to make the idea of objects-on-DB truly practical. In later sections, we first introduce the internal interfaces of Tabular, and then discuss the sources and mitigation of excessive memory copying for Tabular to reclaim most of the performance, matching hand-crafted counterparts.

[3]We only use ERMIA's table abstraction, without its hand-crafted indexes.

3 TABULAR OVERVIEW AND APIS

Tabular is a lightweight, transactional programming library that transparently provides concurrency control and persistence for data structures. As Section 2.2 discusses, we envision the DBMS developer to continue to write imperative C/C++ code (or using a domain-specific language), which then gets transpiled to use table-based interfaces in Tabular [41]. Creating such a transpiler is orthogonal to the design of Tabular; our current index implementations (discussed later in Section 5) directly use these internal Tabular interfaces, which are the focus of the rest of this paper.

Tabular APIs are similar to those exposed by an OLTP engine to other internal components of a DBMS (e.g., the query engine):

- **`Transaction *BeginTransaction()`**: Start a new transaction. Returns a reference/pointer to a `Transaction` object.
- **`bool Transaction::Commit()`**: Commit by performing a set of correctness checks (Section 4). Returns true if the transaction is committed; otherwise aborts the transaction and returns false.
- **`void Transaction::Rollback()`**: Abort the transaction.
- **`void Transaction::Read(Table *table, RID rid, void *out)`**: Read record identified by `rid` and store result into `out` which is a user-prepared buffer (e.g., a `BTreeNode` object).
- **`RID Transaction::Insert(Table *table, void *r, size_t len)`**: Allocate a new record of `len` bytes with content pointed to by `d`; Returns the RID of the new record.
- **`void Transaction::Update(Table *table, RID rid, void *new_data)`**: Update the record identified by `rid` with `new_data`.

Tabular offers new callback-based APIs that are the key for indexes to match the performance of hand-crafted counterparts:

- **`void Transaction::ReadCallback(Table *table, RID rid, Callback cb)`**: Same as `Read`, but performs operations encoded in a callback function `cb` over the record data, without copying the data to a user-specific memory location.
- **`void Transaction::InsertCallback(Table *table, size_t size, Callback cb)`**: Same as `Insert` but performs operations encoded in `cb` upon commit.
- **`void Transaction::UpdateCallback(Table *table, RID rid, Callback cb)`**: Same as `Update` but performs update operations encoded in `cb` over the record data upon commit.

With these APIs, all accesses are transactional with serializability and persistence guarantees (if desired).

4 TABULAR INTERNALS

We start with a baseline using an existing OLTP engine and describe the necessary optimizations that bring Tabular's performance to a competitive level while maintaining the easy-to-use APIs.

4.1 Single-Versioned, Direct Record Access

Many memory-optimized OLTP engines advocate multi-versioning because it can avoid readers and writers to block each other, thus providing superior performance than single-versioning. It may

therefore seem natural to adopt multi-versioning in Tabular because read performance can significantly affect a data structure's overall performance. Some existing indexes [57] are already multi-versioned where an update always appends a delta, rather than performing in-place updates.

Evaluation of Indexing on MVCC OLTP. We implemented a B+-tree in this way on top of ERMIA [44], a representative memory-optimized, multi-versioned OLTP engine.[4] ERMIA implements multi-versioning using indirection arrays and version chains following new-to-old order [92]. Each record is uniquely identified by a logical RID that indexes into the indirection array which represents a table. Each indirection array entry points to the latest version of the record and versions are allocated in the heap using a memory allocator such as `jemalloc` [19]. Figures 5(a–b) depicts the idea. A tree traversal starts by reading the record that stores the root node, which in turn is done by (1) using the root node's RID (`0`) as an index to the indirection array and (2) following the pointer stored in the indirection array entry (address `0x7bee0` in the figure). Depending on the freshness of the transaction issuing the read, we may need to traverse the version chain until a visible version is found (not shown in the figure for brevity). As Figure 5(c) shows, B+-tree nodes store child node RIDs instead of pointers. On a 48-core dual-socket server, compared to a hand-crafted OLC B+-tree, in Figure 6, such multi-versioned B+-tree (denoted as `Naïve-MVCC` which uses the copy-based `Read`/`Update` interfaces) is ~49–79% slower than the hand-crafted counterpart under a balanced read/update workload (details in Section 6) due to indirection and multi-versioning.

Single-Versioned Flat Table. Therefore, Tabular should not blindly follow current practices in OLTP engines which serve a different kind of workload than physical-level data structures. Rather, it is necessary to reduce indirection and versioning overheads. To this end, Tabular directly uses a flat memory space for allocating records per table, and each record is represented by a single version. RIDs are physical, much like in conventional storage-centric systems, indicating the offset into the flat memory space. As Figure 5(d) shows, records (B+-tree nodes) now are laid out one after another, without additional indirection. Each parent node still carries child RIDs, but accessing them now only requires adding the RID (offset) onto the base virtual address of the memory space backing the table. For example, node A (the root node) starts at offset 0 and node B is located at offset 256 (an RID stored in node A) from base address `0x7bee0`. In contrast, in Figures 5(a–b), accessing node B requires additionally traversing the indirection array and potentially the version chain, incurring at least two additional cache misses.

Tabular's flat memory space only needs to be logically contiguous and can be faulted in gradually [78]. Records are read and updated in-place. Upon deletion (e.g., during a B+-tree merge), our current implementation first marks the record as deleted. Compaction can then be performed later (e.g., during system maintenance time), similar to existing systems [16, 44, 56]. As Figure 6 demonstrates, avoiding indirection and versioning (`Naïve-OCC` which also uses the `Read`/`Update` interfaces) brings index performance to ~59–70% of the hand-crafted counterpart. We explore how Tabular further narrows the performance gap next.

[4]ERMIA is a full engine that only allows index-based table accesses. For fair comparison, we implemented core ERMIA functionality in Tabular codebase to enable direct table accesses without hand-crafted indexes and used the same benchmark driver.

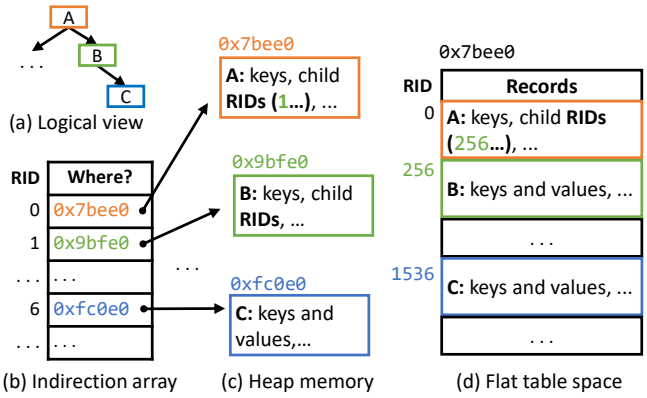


Figure 5: Building a B+-tree (a) on top of ERMIA (b–c) vs. using Tabular's flat table (d). Tabular eliminates additional indirection and versioning to improve performance.

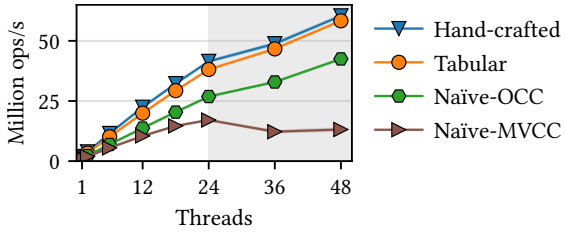


Figure 6: Effect of single-versioning (`Naïve-OCC` over `Naïve-MVCC`, Section 4.1) and zero-copy optimistic concurrency (`Tabular` over `Naïve-OCC`, Section 4.2) on a B+-tree balanced microbenchmark with 50% updates and 50% reads with 100 million 8-byte keys.

4.2 (Near) Zero-Copy Optimistic Concurrency

Optimistic, verification-based concurrency control has been the dominant approach in recent memory-optimized OLTP engines [45, 85, 89, 95]. Compared to traditional two-phase locking (2PL), the common advantage—which mimics optimistic locking in hand-crafted indexes—is readers can proceed without taking any locks and only need to verify at commit time that the data read is still valid. We start with the representative decentralized OCC [45, 85] and discuss how Tabular only needs simple tweaks to make it competitive for physical-level data structures.

Traditional Decentralized OCC. To set the stage, we briefly describe traditional decentralized OCC [45, 85]. Each record is stamped with a transaction ID (TID) that denotes its creator or latest updater. Part (e.g., 1 bit) of the TID (typically 64-bit) can be carved out as the record lock, acquired in exclusive mode when the record is being updated. During transaction execution, readers proceed without taking locks but remembering the TID value in unlocked state in a transaction-local read set. Writes are maintained locally in a write set. Upon commit, the write set is first sorted (to avoid deadlocks) and then locked. The protocol then performs read verification by checking weather the current TID matches the saved TID in the read set, and if not, the transaction will be aborted. After verification, the protocol generates a new commit TID for the

transaction. Finally, we iterate through the (now sorted) write set to apply writes in-place to the actual record locations and unlock each record. Note that at this point the write set is already in a different order and may not represent the order in which individual writes were performed forward processing. As we discuss later, this has implications for Tabular's callback-based interfaces.

Decentralized OCC is very lightweight and thus delivers high performance for database workloads, but is inadequate for data structures. The key culprit is a common design that induces excessive amounts of memory copying. To see why, consider a B+-tree built on top of decentralized OCC and models B+-tree nodes as table records. To read or update a node (table record), the B+-tree code performs a query using `Transaction::Read()` from Section 3. In addition to `table` and `rid`, the caller (e.g., the B+-tree probing function) also prepares a pre-allocated memory chunk for the engine to store the target record. We refer to this kind of read operations *materialized reads*. As Figure 6 has highlighted, `Naïve-OCC` with materialized reads can be ~30–41% slower than the hand-crafted upper bound, with the single-version flat table design in Section 4.1. This is also evidenced by the large amount of memory copying operations in Figure 4 when running a read-only B+-tree workload, leaving few cycles for useful work. Index updates and inserts are also affected because they rely on traversal to arrive at the leaf node; we evaluate these operations later in Section 6.

Zero-Copy Read with Callbacks. Tabular provides callback-based APIs to solve this problem. The application (e.g., B+-tree code) specifies the operations to be performed on top of a record in a callback function, instead of performing the operations after copying out the record. Similar to materialized `Read`, `ReadCallback` takes as input the table to be accessed, RID, and a callback function which will be invoked by Tabular internally. As shown in Algorithm 1, a read operation performs in the same way as that was done in traditional decentralized OCC, except that at line 6 we perform the callback on top of the record. In contrast, it would be a memory copying operation performed by traditional decentralized OCC.

For `ReadCallback` to work correctly, the callback should not modify the record. We believe this is a reasonable assumption as the users of Tabular are database index developers who know the sequential index logic well. The transpilation pass mentioned in Section 3 could enforce this requirement by transforming materialized reads into callback-based reads.

Zero-Copy Modification and Commit. Similar to read operations, updates and inserts can also be callback based. The main caveat is that unlike reads, the insert/update cannot be performed right away to retain the consistency the records. They can only be applied after the verification phase succeeds during commit time, which requires tweaking the commit protocol. Algorithm 2 shows the new commit algorithm where the new/modified steps on top of decentralized OCC are shaded. The commit protocol also must retain program order when performing callbacks. That is, the callbacks should be performed in the order of their addition to the write set during forward processing to preserve possible dependencies. For example, suppose an application first modifies record B with RID `100`, and then relies on B's value to update record A whose RID is `50`. This means the callback for modifying A would need to read B's new value. Both callbacks should only be applied at commit time, yet this can be at odd with the existing decentralized

Algorithm 1 Callback-based optimistic read in Tabular.

```
1  def Transaction::ReadCallback(table, rid, callback):
2    # Take a reference (pointer) to the record
3    Record &record = table.GetRecord(rid);
4  retry:
5    tid = record.get_consistent_tid()
6    callback(record) # Execute the callback in-place
7
8    # Verify the record did not change
9    tid_now = record.get_consistent_tid()
10   if (tid_now == tid):
11     read_set.Add(record, tid);
12   else:
13     goto retry
```

Algorithm 2 Tabular's commit protocol.

```
1  def Transaction::Commit(table, rid, callback):
2    # Sort out-of-place and lock write set
3    sorted_ws = sort(write_set)
4    foreach w in sorted_ws:
5      lock w
6
7    ... fences and get commit epoch ...
8
9    # Validate reads - exactly the same as OCC
10   foreach r in read_set:
11     if r.tid != r.record.get_tid() and !r.locked_by_me():
12       rollback and return
13
14   ... iterate read/write sets to get new commit_tid ...
15
16   # Apply write callbacks and updates
17   foreach w in write_set:
18     if w.is_callback:
19       w.callback() # invoke the callback
20     else: # "normal" materialized update
21       memcpy(w.record.data, w.data, w.size)
22     w.tid = commit_tid
23
24   ... other remaining operations ...
```

OCC protocol which sort all writes in a global consistent order (e.g., by RIDs) to avoid deadlocks. As a result, A's callback would be performed before B's, breaking program dependency requirements. To solve this problem, at lines 2–5 of Algorithm 2, we sort the write set out-of-place (e.g., by RID order) into an additional write set structure (`sorted_ws`), which is then iterated through to lock all the writes. After verifying all the read records and obtaining a new commit TID, we iterate over the original write set which follows program order (lines 17–20). This way, Tabular avoids deadlocks while guaranteeing program logic specified by the developer.

4.3 Correctness

On top of decentralized OCC, Tabular's commit CC protocol makes two changes. (1) Some reads are performed using callbacks. (2) Some updates and other logic associated with it are delayed until commit time. The correctness of decentralized OCC has been proven [85] by showing that it reduces to be S2PL-equivalent and is thus serializable. We argue for the correctness of Tabular's callback-based protocol by showing that these two changes do not change decentralized OCC's equivalence to S2PL. Like previous work [45, 85], for brevity we assume the write is a subset of the read set. We first consider callback-based reads in the read set. They are performed by reading a consistent version of the data during forward processing (Algorithm 1) and verified at commit time. This maintains the equivalence that S2PL would have been able to acquire all the read locks [85]. Next, we consider the callback-based write set entries. Algorithm 2 performs the callbacks after (1) locking all the writes and (2) verifying reads. The locking step is equivalent to upgrading the read locks to exclusive locks in S2PL, where the new updates are only visible to the transaction. Tabular retains the same property; the only difference is the actual writes are performed at commit time but still before the results become visible to other transactions.

Compared to hand-crafted optimistic locking, Tabular provides stronger and more conservative consistency guarantees by enclosing all record operations in one transaction. A hand-crafted protocol can be seen as multiple shorter transactions, each of which accesses a subset of records. For example, (optimistic) lock coupling unlocks the parent node once it is sure that the parent node will not be modified. This is equivalent to committing a transaction that reads the current node N and a parent node, but simultaneously starting another transaction that continues to drill down the tree based on the current version of N. This can admit more parallelism than Tabular, except when a split/merge is propagated to the root of the tree: both Tabular and lock coupling will lock the entire path, equivalent to enclosing all node/record operations in one transaction. Such larger transaction scope enlarges the conflict window and potentially lowers performance. However, as we have shown earlier and later in Section 6, its impact is very small since typical transactions used by indexes are not long in the first place.

4.4 Durability and Recovery

If durability and recovery support is needed, Tabular will generate and persist log records and provide transparent persistence support. For materialized updates and inserts the new values are collected during forward processing, while for callback-based operations, the log records are generated when the callbacks are performed at commit time. Similar to many memory-optimized OLTP engines, Tabular uses redo-only logging [69] for durability. Data generated by aborted transactions are discarded and will never make it to the persistent log. To avoid centralized logging bottleneck and I/O delays, we use a private log buffer per thread which can be flushed in the background, following classic pipelined commit protocols [38]. This allows Tabular to leverage the higher sequential write performance (than that of random accesses) of modern SSDs/disks, but a common problem of this approach is longer latency for each operation. For systems that need low latency, techniques such as placing log buffers in NVDIMMs [86–88] can be employed.

4.5 Discussions

We consciously made several tradeoffs for Tabular to deliver competitive performance. The first is that our table abstraction in Section 4.1 is by nature memory-centric by assuming a virtually-contiguous flat space for a table. This eliminates unnecessary indirection, but makes it more complex to support larger-than-memory data structures. A possible solution is to allow RIDs to refer to either an in-memory offset or a storage address, indicated by a dedicated a bit in the RID word (e.g., the most significant bit). The table space then becomes a buffer space that would need to be carefully crafted. One possible solution is to adopt recent lightweight out-of-memory solutions [50, 51, 71] that use techniques such as pointer swizzling [42] to reduce overheads, however, it is important to also maintain the same interfaces to avoid application-level changes.

Based on modern decentralized OCC [35, 45, 85], Tabular concurrency control protocol mimics the behavior of optimistic locking [11, 55] commonly used by hand-crafted data structures. While it avoids most unnecessary overheads, OCC is inherently vulnerable to high contention. Numerous efforts have attempted to address this problem [27, 35, 63, 79, 89, 95]. As promising future work, Tabular can also adopt such optimizations (e.g., by acquiring locks early) to deliver better performance under contention. Moreover, such optimizations can happen within Tabular library itself, without affect how index developers program, using the same transactional table abstractions. Existing/new hand-crafted indexes could be transformed/developed to use Tabular, which provide optimizations under the hood. This allows a potentially smaller team of engineers to focus on improving Tabular (e.g., to evolve it for new hardware or adding features), than having to retrofit each desirable feature into an existing hand-crafted index.

Data structures built by Tabular can be used to build a full DBMS. Combining Tabular and a Tabular-based B+-tree allows one to easily implement an OLTP engine, without having to (re-)implement the table abstraction and the associated ACID guarantees.

5 TABULAR USE CASES

In this section, we discuss how a DBMS developer may use Tabular APIs to build concurrent and persistent indexes relatively easily. After introducing the core ideas, we elaborate the process and design considerations using B+-tree and extendible hashing.

5.1 From Sequential Logic to Tabular

Tabular aims to free DBMS developers from complex parallel programming issues by using the transactional interfaces described in Section 3. This only requires the DBMS developer be familiar with (1) the target data structure's sequential (i.e., single-threaded) logic and (2) the concepts of relational tables and transactions. Both are commonly covered by introductory data structure and database courses in most undergraduate curricula, and neither requires advanced data structures or parallel programming backgrounds.

Modeling Index Structures as Tables. From DBMS developers' perspective, they would need to reason about data structure design using relations and transactions. We observe this can be done easily by realizing that most indexes consist of three building blocks. (1) Memory blocks (e.g., B+-tree nodes) connected through pointers. (2) Functions for accessing these memory blocks. (3) Concurrency

Algorithm 3 Sequential B+-tree update routine.

```
1 def BTree::Update(k, v):
2   node = self.root;
3
4   while node.is_leaf is false:
5     node = node.findChild(k)
6
7   if node.key_exists(k):
8     node.update(k, v);
```

control and persistence protocols. Based on this observation, transforming a single-threaded index to use Tabular is straightforward. Allocating a memory block is transformed into inserting a new record. Data manipulation are done using transactions on table records. Index code (transaction) accesses data records using RIDs, rather than pointers. Memory block sizes will determine database record sizes, which in turn can affect performance. For example, a B+-tree leaf node may inline data as values, requiring larger nodes to accommodate the same number of key-value pairs. Using the copy-based APIs would lead to suboptimal performance, making the callback-based ones preferable. A related question to answer when modeling a data structure class as a table (from class definitions to schema definitions) is "*what should be a record?*" In principle, a contiguous memory chunk in a C/C++ struct would translate into a record in Tabular. However, blindly following this principle may lead to suboptimal performance. We elaborate and provide solutions later in Section 5.3. Finally, memory reclamation, concurrency and persistence are handled transparently by Tabular transactions without developer intervention.

Measuring Code Complexity. It is highly subjective to measure code complexity. Each metric (e.g., cyclomatic complexity [66] and lines of code) has its own limitations. Recent work [70] has shown that cognitive complexity [7] is promising at correlating with the time spent on comprehension and subjective ratings of code understandability. We use it to measure programming efficiency and the mental burden of implementing indexes.

5.2 B+-Tree

To set the stage, Algorithm 3 shows a sequential B+-tree update function, which is very straightforward. The algorithm starts traversal from the root to the leaf level, and then searches for the target key in the leaf node. If the key is found, it performs the update.

Tabular Adaptations. Algorithm 4 presents Tabular-based B+-tree update using the copy-based interfaces. It (1) wraps node accesses in a transaction and (2) performs RID-based traversal. Line 2/16 starts/commits the transaction. Lines 4–6 corresponds to line 2 of Algorithm 3 for accessing the root node. Since nodes are represented as database records, they are addressable only by RIDs. So line 5 prepares a buffer in the transaction's local scratch area to store the node read at line 6, which uses the root node RID obtained at line 4. The buffer is an instance of `BTreeNode` as defined by Figure 3(b) except variables for concurrency control (`lock`) are omitted. Similarly, lines 8–10 of Algorithm 4 correspond to lines 4–5 of Algorithm 3 for traversing to the target leaf node. Lines 12–14 are in almost the same logic as lines 7–8 in Algorithm 3 to perform the update. The only difference is that update done in the local buffer should be reflected in the table (line 14).

We use B+-tree insert to demonstrate the callback interfaces in Algorithm 5. In the context of a transaction, the splitting thread

Algorithm 4 Tabular-based B+-tree update using copy-based interfaces. No concurrency control logic is needed.

```
1  def TabularBTree::Update(k, v):
2    t = BeginTransaction() # start a transaction
3
4    rid = self.root_rid
5    BTreeNode node # in-memory node representation
6    t.Read(self.table, rid, &node)
7
8    while node.type.is_leaf == false:
9      rid = node.findChild(k)
10     t.Read(self.table, rid, &node)
11
12   if node.key_exists(k):
13     node.update(k, v) # perform update in local buffer
14     t.Update(self.table, rid, &node)
15
16   if not t.Commit(): retry from line 2
```

Algorithm 5 Tabular-based B+-tree insert using callbacks.

```
1  def TabularBTree::Insert(k, v):
2    t = BeginTransaction()
3    Stack stashed_nodes
4
5    ...traverse to leaf; path recorded in stashed_nodes...
6
7    if not leaf.is_full(): # will not split
8      def leaf_insert_cb(leaf, k, v): leaf.insert(k, v)
9      t.UpdateCallback(nodes, leaf_id, leaf_insert_cb, k, v)
10   else:
11     # Define and register a callback to split a leaf
12     Key sep
13     def split_leaf_cb(&split_leaf, k, v, &sep):
14       sep = leaf.split_to(split_leaf, k, v)
15     new_rid = t.InsertCallback(nodes, split_leaf_cb,
16                                split_leaf, k, v, sep)
17     # Update inner nodes
18     while stashed_nodes.size() > 0:
19       [parent_id, parent] = stashed_nodes.pop()
20       if parent.is_full():
21         def split_inner_cb(&new_inner, &sep):
22           sep = parent.split_to(new_inner, sep, new_rid)
23         new_rid = t.InsertCallback(nodes, split_inner_cb)
24       else:
25         def update_inner_cb(): parent.insert(sep, new_rid)
26         t.UpdateCallback(nodes, parent_id, update_inner_cb)
27         if not t.Commit(): retry from line 2
28     root_rid = new_rid # Grow the tree if reached here
29   if not t.Commit(): retry from line 2
```

first probes down the tree to arrive at the leaf node (lines 2–6). Like in single-threaded logic, references to inner nodes along the path are collected in a stack in case later the split is propagated upwards. If no split is needed, we register a callback to perform the insert in the leaf node (lines 7–9). Otherwise, the algorithm splits the existing leaf node into a new node (at the same time, insert the new key into the proper target node) and registers a callback (line 15) to insert it at commit time. Lines 18–27 then iterate over the stack of inner nodes collected earlier to possibly propagate the split. For each level, a callback is registered to either insert a new node (lines 21–23) or update the existing node with the new separator key (lines 25–27). The latter case is followed by a transaction commit to conclude the operation. Otherwise, if all the parent levels are exhausted without triggering a commit at line 27, we have increased the height of the tree. Although more involved than the copy-based Algorithm 4, callback-based insert still follows single-threaded logic without involving concurrency or persistence handling, which is the complex effort Tabular aims to save.

Programming Efficiency. We calculate the cognitive complexity of major B+-tree functions using `clang-tidy` [13]. The sequential implementation whose update function is shown in Algorithm 3 has the lowest cognitive complexity score of 31 (lower is better). The hand-crafted variant has the highest score of 67, which is 1.71× over that of the Tabular-based B+-tree (39). As mentioned earlier, the adaptation process could be done by the developer or be automated by a transpiler to further increase programming efficiency.

5.3 Extendible Hashing

Extendible hashing [20] is a classic dynamic hashing design. In memory-optimized systems, the directory typically is implemented as an array, whose entries point to buckets allocated in the heap. Figures 7(b–c) define the C/C++ structures. Inserting into a full bucket will split the bucket and re-hash existing keys into both buckets. Consequently, a new pointer to the new bucket needs to be added to the directory. The directory always grows in the power of two, leading to 2^{global_depth} buckets. Each bucket carries a *local_depth*, which if is smaller than *global_depth* then the bucket is yet to be split and will be pointed to by ≥ 2 directory entries. Extendible hashing can also use (optimistic) locking: accesses to buckets only verify directory entry and bucket versions, while a split will need to lock both the bucket and directory.

Table Definitions. Buckets are fixed-sized, so it is easy to model and store them as table records, as Figure 7(e) shows. Similarly, the directory can be modeled as a (variable-length) record and expanding or shrinking the directory would be translated into updating the directory record. Alternatively, one could always insert a new directory record into the directory table as described in Figure 7(d). Neither table includes columns for concurrency control (i.e., `Bucket::lock`), which is handled transparently by Tabular.

Tabular Adaptations. Modeling the directory as a record is straightforward, but can be challenging when the directory grows: using the copy-based APIs we would need to copy the entire directory. As Section 6 shows, extendible hashing using these APIs perform poorly and consumes an excessive amount of memory. This necessitates the use of the callback-based APIs. Algorithm 6 shows the callback-based version which first uses `ReadCallback`

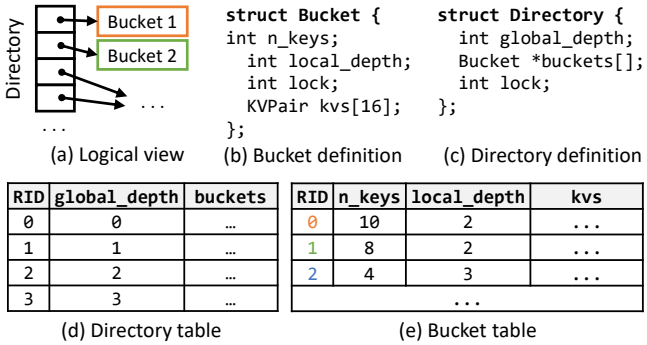


Figure 7: Modeling extendible hashing (a) in tables. The buckets (b) and directory (c) are stored in two separate tables (d–e).

Algorithm 6 Tabular-based hash table update with callbacks.

```
1  def TabularHashTable::Update(k, v):
2    t = BeginTransaction()
3
4    # Define a callback function to retrive entry
5    def get_cb(record, key, e):
6      Directory dir = (Directory)record
7      e = dir[hash(key) % size]
8
9    # Invoke callback-based read with get_cb
10   Entry e
11   t.ReadCallback(self.dir_table, self.dir_rid,
12                  get_cb, k, &e)
13
14   # Define an update function to update value in bucket
15   def update_cb(record, key, val):
16     Bucket bucket = (Bucket)record
17     if bucket.key_exists(key):
18       bucket.Update(key, val)
19
20   # Invoke callback-based update with update_cb
21   t.UpdateCallback(self.bucket_table, entry.bucket_rid,
22                    update_cb, k, v)
23   if not t.Commit(): retry from line 2
```

to obtain the directory entry pointing to the target bucket (lines 5–7). The callback is then passed to `ReadCallback` along with other parameters in line 11 of Algorithm 6. Lines 15–22 then searches and updates the bucket.

Programming Efficiency. Although more involved than the B+-tree case using copy-based APIs, the adaptation of extendible hashing using callbacks still avoids the handling concurrency and persistence issues, the largest sources of complexity.

6 PERFORMANCE EVALUATION

We have evaluated the effect of Tabular's individual techniques for B+-trees under certain workloads in Sections 2–4. In this section, we include more scenarios and show that:

- Tabular-based B+-trees, extendible hashing and adaptive radix tree (ART) can provide competitive performance close to the upper bound provided by state-of-the-art hand-crafted indexes.
- Tabular transparently enables persistence support for indexes and only incurs minor logging overheads.
- By replacing hand-crafted indexes with Tabular-based ones, an existing OLTP engine can improve programming efficiency while maintaining competitive overall performance.

6.1 Experimental Setup

We performed experiments on a dual-socket server with two 24-core Intel Xeon Gold 6252 CPUs and 384GB of DRAM. The CPU is clocked at 2.1GHz (3.7GHz with Turbo Boost) and has 35.75MB of caches. The server runs Arch Linux with kernel version 6.6.2. We use 1GB huge pages and `jemalloc` [19] to avoid memory allocation being a major bottleneck. We interleave huge page allocations when threads are spread across two sockets. Each worker thread is pinned to a dedicated CPU core without hyperthreading to avoid the impact of OS scheduler activities and ease the analysis of results.

Implementation. We implemented Tabular as a shared library. Index code includes a Tabular header file and links with the library. To focus on the performance of Tabular, we omit transpilation. All the code is implemented in C++20 and compiled with GCC 13.

Index Variants. We compare variants of B+-tree, extendible hashing, and ART implemented in six different ways:

- `Hand-crafted`: State-of-the-art hand-crafted in-memory variant with optimistic lock coupling.[5]
- `Naïve-MVCC`: Naïve baseline objects-on-DB variant with multi-versioning and materialized read/write in ERMIA [44].
- `Naïve-OCC`: Naïve baseline objects-on-DB variant with single-versioning, decentralize OCC [85] and materialized read/write.
- `Tabular`: Indexes implemented using Tabular with single-version flat tables and callback-based operations.
- `STD-LC`: Pessimistic lock coupling using `std::shared_mutex` in standard C++ (since C++17).
- `TBB-LC`: Pessimistic lock coupling using `tbb::spin_rw_mutex` in Intel oneAPI Threading Building Blocks (TBB) [36].

`Naïve-MVCC` uses snapshot isolation without serializability, adding which can add more overhead. We take recent B+-tree and ART implementations [79] as `Hand-crafted` variants. We implemented optimistic `Hand-crafted` extendible hashing [62]. All `Hand-crafted` variants are in-memory only as they were not design with persistence support to begin with, but present performance upper bound. `STD-LC` and `TBB-LC` represent two classic hand-crafted baselines built using popular synchronization primitives.

Benchmarks. We perform both index-only microbenchmarks and end-to-end TPC-C [84] benchmarks. For both types of benchmarks, each experiment runs for 10 seconds and is repeated for three times. We report the average throughput measured in million operations per second (million ops/s). Variances between runs are shown as shaded regions surrounding the lines in figures, although we observe that the variance is minuscule. We describe the details of each benchmark in their corresponding sections below.

[5]For range indexes, `Hand-crafted` refers to OLC B+-tree/ART, except in Section 6.4 which uses Masstree [44] as the hand-crafted variant.

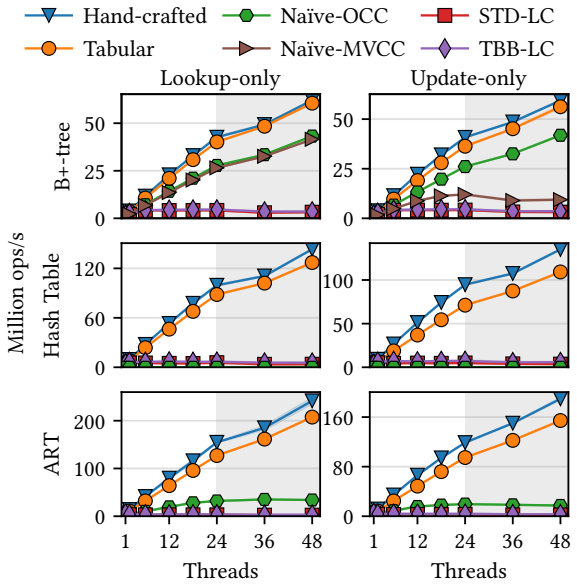


Figure 8: Throughput of B+-tree (top), extendible hashing (middle), and ART (bottom). `Tabular` maintains competitive performance (up 95% of `Hand-crafted`) by removing unnecessary memory copying.

6.2 Index Performance

Our first set of experiments stress test indexes. For each experiment, the index is loaded with 100 million 8-byte keys and 8-byte values; keys are drawn from a uniform random distribution. We focus on lookup-only and update-only workloads.[6]

B+-Trees. As shown in Figure 8(top), `Hand-crafted` performs the best as expected, providing performance upper bound. `Tabular` stays competitive in each workload by following closely the upper bound; we also performed pure insert tests, which showed similar trend as the update-only workload. `Naïve-OCC`'s throughput is ~70% of that of `Hand-crafted` across different numbers of threads in each workload, due to memory copying overheads. `Naïve-MVCC` exhibits ~70% of `Hand-crafted` performance under read-dominant workloads. However, its throughput further lowers when more updates are involved and starts to drop when memory accesses cross NUMA boundaries beyond 24 threads (gray areas in the figure) due to the increased latency caused by pointer-chasing in multi-versioning overheads. `STD-LC` and `TBB-LC` B+-tree are unable to scale due to their high pessimistic locking overhead. Thanks to the callback and single-version flat table designs, `Tabular` retains over ~90% of `Hand-crafted` across all core counts and workloads.

We performed detailed profiling (using perf [60]) to identify the root cause of the performance gap between hand-crafted and Tabular based indexes. We take B+-tree as an example, but profiling results of other indexes led to the same conclusion, so we omit them here. Figure 9 shows the CPU cycle breakdown of `Hand-crafted` and `Tabular` B+-trees under 24 threads (one socket to avoid NUMA effect and ease analysis). The main overheads of `Hand-crafted` are

[6]We also tested other mixes such as balanced 50–50% read–update workloads; their results fall between these two cases. So we omit them due to space limitation.

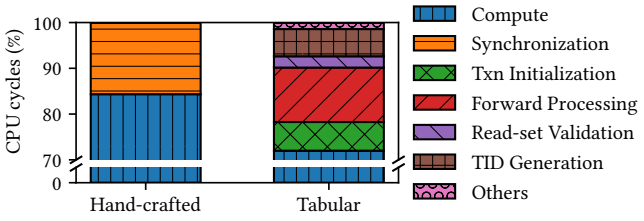


Figure 9: CPU cycle breakdown of `Hand-crafted` and `Tabular` B+-trees running the lookup-only workload (24 threads).

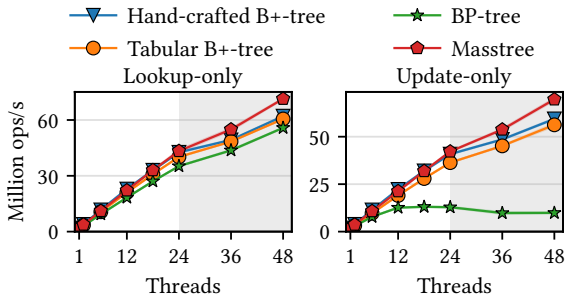


Figure 10: `Tabular` B+-tree vs. hand-crafted counterparts.

(1) verification and (2) write locking time which collectively take ~15% of the CPU cycles ("Synchronization" in the figure), leaving 85% for tree logic ("Compute"). Tabular uses 12% more cycles on synchronization which is represented by the remaining legends in the figure. In detail, Tabular takes ~8% of the cycles for initializing and cleaning up transaction contexts. The callback mechanisms (implemented using `std::function` which requires dynamic memory allocation) and read verification before commit (Algorithm 1) take another ~11% of CPU cycles. The commit protocol (including TID generation, read verification and other operations such as write-set sorting) takes ~9% of CPU cycles. These results corroborate with the performance gap shown in Figure 8.

Hash Tables. The relative trend between `Tabular` and other hash table variants is similar to that for B+-trees in Figure 8. `Tabular` retains over ~80% of `Hand-crafted`'s throughput at high core counts. For the same reason as we discussed previously, `STD-LC` and `TBB-LC` hash tables do not scale. Moreover, it is noticeable that `Naïve-OCC` and `Naïve-MVCC` perform extremely poorly due to excessive memory copying overheads. `Naïve-MVCC` hash table exhausted memory before completion due to excessive memory consumption of multi-versioning. Table 1 compares the memory consumed (in GB) by `Tabular` and baselines after loading 100M records without garbage collection. `Hand-crafted` uses 3.7GB without any additional overhead or metadata that is needed by the table structures in `Tabular`, `Naïve-OCC` and `Naïve-MVCC`. `Naïve-OCC` and `Tabular` both consume 4.72GB, whereas `Naïve-MVCC` consumes over 300GB and exhausted available DRAM in the server. The reason is that each directory update generates a new version by copying and creating a large amount of data. A proactive garbage collection policy could mitigate this issue for `Naïve-MVCC`, but it is still more likely to bloat memory usage if a large amount of versions cannot be reclaimed timely. This result highlights the need to move away from the conventional wisdom of building an OLTP engine for database workloads using multi-versioning, and the need to use single-versioned accesses in a library like Tabular.

ART. `Tabular` obtains up to ~85% of `Hand-crafted`'s throughput. `Naïve-OCC` suffers for the same reason as extendible hashing (large record size) and we found that ~ 50% of the CPU cycles are spent on memory copying. `STD-LC` and `TBB-LC` ART also do not scale for the same reason discussed earlier.

Tabular vs. State-of-the-Art Hand-Crafted B+-Trees. To put the performance numbers of Tabular-based indexes in perspective, we compare them with other hand-crafted ones. We focus on B+-tree variants and compare with Masstree [65] and BP-tree [94]. In Figure 10, `Tabular` B+-tree is comparable with Masstree. Masstree's optimizations such as prefetching mitigate NUMA issues, allowing it to scale (slightly) better than `Hand-crafted` B+-tree. BP-tree is generally slower than other variants, since it employs traditional lock coupling with reader-writer locks instead of OLC/OCC.

Table 1: Memory needed by hash table variants to store 100M records. `Naïve-MVCC`'s multi-versioning bloats memory usage.

`Hand-crafted`	`Tabular`	`Naïve-OCC`	`Naïve-MVCC`
3.5GB	4.72GB	4.72GB	> 300GB

Table 2: Throughput (million operations per second) of volatile and persistent `Tabular` B+-trees. The SSD (Intel P4800X) is saturated with 4 threads (2GB/s).

Threads	**1**	**2**	**4**	**8**
Volatile	1.54	3.09	6.19	12.39
Persistent	1.31	2.64	5.35	6.96

Summary. Tabular-based indexes match over ~80–95% of their `Hand-crafted` counterparts' throughput at high core counts, showing a tradeoff between performance and other desiderata.

6.3 Effect of Transparent Persistence

Tabular allows the developer to turn a volatile index into a persistent one by simply enabling the persistent option when creating tables. We use `Tabular` B+-trees and run the update-only workloads with a 375GB Intel P4800X SSD [37] which has a peak bandwidth of 2GB/s. The log buffer size is set to 32MB and is flushed with `O_DIRECT` to bypass the OS page cache. As Table 2 shows, adding persistence incurs ~14% of overhead on top of the volatile variant before the SSD is fully saturated under four threads. Beyond that, e.g., with eight threads, the system is bottlenecked by I/O and thus maintains the performance at four threads.

To explore in-memory logging overhead, we performed an experiment that skips actual I/O but keeps all other log-related operations. As Figure 11 presents, under the lookup-only workload volatile and persistent `Tabular` B+-tree perform exactly the same because no logging is involved. With logging, persistent `Tabular` B+-tree exhibits a slight drop of ~3% caused by log-record generation. This indicates ~2% of overhead is due to SSD I/O operations, compared to

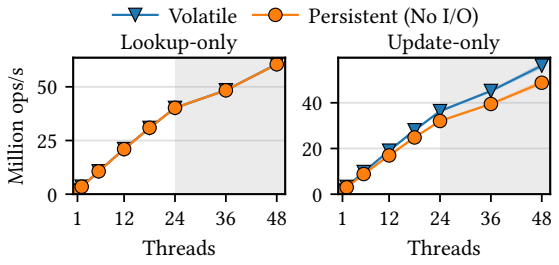


Figure 11: Throughput of volatile and persistent `Tabular` B+-tree under read-only (left) and update-only (right) workloads.

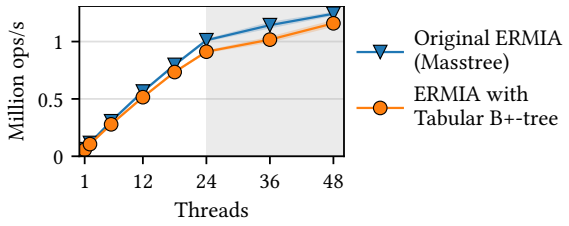


Figure 12: TPC-C throughput with Masstree vs. `Tabular`.

the previous SSD-based results. The same experiment for extendible hashing exhibited similar trends, so we do not repeat here.

6.4 End-to-End TPC-C Results

Our final experiment tests how Tabular-based indexes work in an OLTP engine by integrating `Tabular` B+-tree in ERMIA [44], replacing its hand-crafted Masstree [65]. We use the TPC-C benchmark [84] and assign each worker a home warehouse (about 10% of the payment transactions may access a remote warehouse). So the workload inherently does not present much contention, putting more pressure on indexes, accessing which takes ~50% of total CPU cycles in our profiling results. We set both ERMIA variants to use snapshot isolation.[7] Following prior work [16, 44, 59, 85], we use an equal number of warehouses and worker threads in TPC-C. As Figure 12 shows, with `Tabular` B+-tree ERMIA performs similarly to original ERMIA, matching over 95% of the latter's performance. This demonstrates that the performance of Tabular-based indexes is sufficient for representative end-to-end OLTP workloads.

7 RELATED WORK

Our work is closely related to prior efforts on modern OLTP, index synchronization, objects-on-DB and parallel programming.

Memory-Optimized OLTP. Many logical-level concurrency control (CC) algorithms have been proposed. A common theme is to use lightweight optimistic approaches [46]. Hekaton [16] uses verification for optimistic MVCC. Silo [85] removes the centralized TID allocation. A major drawback is they are not robust, i.e., performance can collapse under high contention and read-mostly workloads [44]. Much work has focused on mitigating this issue, e.g., by combining pessimistic and optimistic CC [16, 82, 89], leveraging MVCC [6, 16, 44] and novel ways of managing timestamps [95]. Virtual domains [2] separate data structure (e.g., indexes) logic from the actual execution strategies by using different configurations to improve robustness across different hardware platforms. Tabular can adopt these efforts to provide robust performance.

Index Synchronization. As we discussed earlier, memory-optimized indexes prefer optimistic locking which is lightweight but vulnerable to contention which can be improved by queue-based locking [79]. Some efforts have attempted to ease index implementation using multi-word CAS [26, 29, 90]. However, it still requires developers reason about concurrency logic, which can be complex [1, 83]. There is also no support for persistence on top of SSDs which is provided by Tabular transparently.

Objects-on-DB, OODBMS and ORM. Early work on objects-on-DB [41] targeted database applications with optimizations such as prefetching from a disk-based DBMS [5]. Cicada [59] also stores index nodes in tables, but lacks the optimizations in Tabular. Compared to objects-on-DB, object-oriented DBMSs design explicit object storage services with their own programming models [10]. Object-relational mapping (ORM) maps programming language structures to rows and columns. Unlike Tabular, accesses to data via ORM are transformed into SQL queries. DBOS [80] aims to build OS services on top of fast OLTP. Tabular can complement such approaches with better DBMS support.

Parallel Programming. Other research communities have proposed transactional memory (TM) [32] to ease implementation. Hardware TM has many limitations (e.g., spurious aborts) that hinder adoption [54, 64]. Software TM (STM) is more flexible but many sacrifice performance due to word-level tracking [9]. STO [34] tracks object-level accesses based on data types, whereas Tabular tracks access by database records. TDSL [81] adopts TM optimizations with data structure specific designs for better performance. Some STM solutions [17, 22, 33, 73, 77] use techniques similar to Tabular's. Hybrid solutions [8, 18] combine optimistic and pessimistic CC; Tabular could adopt them for robustness. Universal constructions [12, 14, 31] transform a sequential algorithm to a concurrent one, but often requires more copies and/or atomics, leading to low performance. Some work [24, 49] transforms volatile indexes into durable ones on persistent memory but has seen limited adoption given the only product has been canceled recently.

8 SUMMARY

We have presented Tabular, a new lightweight library to ease the programming of concurrent and persistent indexes. The core idea is to map indexes to relational tables with ACID guarantees. Tabular combines several important designs in modern OLTP engines to make this idea practical and deliver competitive performance while reducing programming complexity.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. We also thank Yue Deng who helped with benchmarking during early stages of this work. This work is partially supported by an NSERC Discovery Grant, Canada Foundation for Innovation John R. Evans Leaders Fund, B.C. Knowledge Development Fund and Mitacs Globalink Research Internship Awards.

[7] Not to be confused by the isolation level enforced by Tabular, which uses single-versioned serializable OCC to ensure correct index concurrency.

REFERENCES

[1] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: a high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.* 11, 5 (jan 2018), 553–565.

[2] Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. 2020. Robust Performance of Main Memory Data Structures by Configuration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 1651–1666.

[3] R. Bayer and E. McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET '70)*. 107–141.

[4] R. Bayer and M. Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Inf.* 9, 1 (mar 1977), 1–21.

[5] Philip A. Bernstein, Shankar Pal, and David Shutt. 1999. Context-Based Prefetch for Implementing Objects on Relations. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. 327–338.

[6] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (Dec. 2009), 42 pages.

[7] G Ann Campbell. 2018. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*. 57–58.

[8] Man Cao, Minjia Zhang, Aritra Sengupta, and Michael D. Bond. 2016. Drinking from both glasses: combining pessimistic and optimistic tracking of cross-thread dependences. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. Article 20, 13 pages.

[9] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? The promise of STM may likely be undermined by its overheads and workload applicabilities. *Queue* 6, 5 (sep 2008), 46–58.

[10] R. G. G. Cattell, Douglas K. Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade. 1997. *The object database standard: ODMG 2.0*.

[11] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. 181–190.

[12] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. 2010. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. 335–344.

[13] Clang. 2024. Clang-Tidy - Readability-Function-Cognitive-Complexity. https://clang.llvm.org/extra/clang-tidy/checks/readability/function-cognitive-complexity.html.

[14] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2020. A wait-free universal construction for large objects. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. 102–116.

[15] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 327–342.

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. 1243–1254.

[17] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06)*. 194–208.

[18] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching Transactional Memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. 155–165.

[19] Jason Evans. 2006. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of the BSDCan Conference*.

[20] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible hashing—a fast access method for dynamic files. *ACM Trans. Database Syst.* 4, 3 (sep 1979), 315–344.

[21] Jose M. Faleiro and Daniel J. Abadi. 2017. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. 9.

[22] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. 237–246.

[23] Michael Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-optimized multi-version concurrency control for disk-based database systems. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2797–2810.

[24] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: in NVRAM data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. 377–392.

[25] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. 2014. In-memory performance for big data. *Proc. VLDB Endow.* 8, 1 (sep 2014), 37–48.

[26] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. 2020. Efficient Multi-Word Compare and Swap. In *34th International Symposium on Distributed Computing (DISC 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 179. 4:1–4:19.

[27] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 658–670.

[28] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proc. VLDB Endow.* 17, 11 (July 2024), 3442–3455.

[29] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. 265–279.

[30] Thomas E. Hart, Paul E. McKenney, and Angela Demke Brown. 2006. Making lockless synchronization fast: performance implications of memory reclamation. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. 21.

[31] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (jan 1991), 124–149.

[32] Maurice Herlihy. 2005. The transactional manifesto: software engineering and non-blocking synchronization. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 280.

[33] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. 2003. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (PODC '03)*. 92–101.

[34] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Article 31, 16 pages.

[35] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. *Proc. VLDB Endow.* 13, 5 (jan 2020), 629–642.

[36] Intel. 2024. Intel® oneAPI Threading Building Blocks Documentation. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb-documentation.html

[37] Intel. 2024. Intel® Optane™ SSD DC P4800X Series. https://www.intel.com/content/www/us/en/products/sku/97161/intel-optane-ssd-dc-p4800x-series-375gb-2-5in-pcie-x4-3d-xpoint/specifications.html

[38] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *Proc. VLDB Endow.* 3, 1 (Sept. 2010), 681–692.

[39] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the killer nanoseconds. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[40] Minji Kang, Soyee Choi, Gihwan Oh, and Sang-Won Lee. 2020. 2R: efficiently isolating cold pages in flash storages. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2004–2017.

[41] Arthur M. Keller, Richard Jensen, and Shailesh Agarwal. 1993. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. *SIGMOD Rec.* 22, 2 (jun 1993), 523–528.

[42] Alfons Kemper and Donald Kossmann. 1993. Adaptable Pointer Swizzling Strategies in Object Bases. In *Proceedings of the Ninth International Conference on Data Engineering*. 155–162.

[43] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. 195–206.

[44] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687.

[45] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 691–706.

[46] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226.

[47] Per-Ake Larson. 1988. Dynamic Hash Tables. *Commun. ACM* 31, 4 (April 1988), 446–457.

[48] Leon Lee, Siphrey Xie, Yunus Ma, and Shimin Chen. 2022. Index checkpoints for instant recovery in in-memory database systems. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1671–1683.

[49] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 462–477.

[50] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1, Article 7 (may 2023), 25 pages.

[51] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 185–196.

[52] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42 (2019), 73–84.

[53] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE '13)*. 38–49.

[54] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2014. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering*. 580–591.

[55] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. Article 3, 8 pages.

[56] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proc. VLDB Endow.* 6, 10 (aug 2013), 877–888.

[57] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-Tree for New Hardware Platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE '13)*. 302–313.

[58] Tianyu Li, Badrish Chandramouli, and Samuel Madden. 2022. Performant Almost-Latch-Free Data Structures Using Epoch Protection. In *Proceedings of the 18th International Workshop on Data Management on New Hardware (DaMoN '22)*. Article 1, 10 pages.

[59] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 21–35.

[60] Linux perf wiki Contributors. 2025. perf: Linux profiling with performance counters. https://perfwiki.github.io

[61] Witold Litwin. 1980. Linear Hashing: A New Tool for File and Table Addressing. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6 (VLDB '80)*. 212–223.

[62] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161.

[63] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060.

[64] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. 2015. To lock, swap, or elide: on the interplay of hardware transactional memory and lock-free indexing. *Proc. VLDB Endow.* 8, 11 (jul 2015), 1298–1309.

[65] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 183–196.

[66] Thomas J. McCabe. 1976. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*. 407.

[67] Paul E McKenney. 2021. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* (2 ed.).

[68] M.M. Michael. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.

[69] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.* 17, 1 (mar 1992), 94–162.

[70] Marvin Muñoz Barón, Marvin Wyrich, and Stefan Wagner. 2020. An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)*. 1–12.

[71] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*.

[72] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 677–689.

[73] Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan. 2007. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. 365–375.

[74] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1166–1177.

[75] Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 3.

[76] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems* (3 ed.).

[77] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*. 187–197.

[78] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: rewired user-space memory access is possible! *Proc. VLDB Endow.* 9, 10 (jun 2016), 768–779.

[79] Ge Shi, Ziyi Yan, and Tianzheng Wang. 2023. OptiQL: Robust Optimistic Locking for Memory-Optimized Indexes. *Proc. ACM Manag. Data* 1, 3, Article 216 (nov 2023), 26 pages.

[80] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2021. DBOS: a DBMS-oriented operating system. *Proc. VLDB Endow.* 15, 1 (sep 2021), 21–30.

[81] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 682–696.

[82] Dixin Tang, Hao Jiang, and Aaron J Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All.. In *CIDR*, Vol. 2. 1.

[83] Xiangpeng Hao Tianzheng Wang. 2019. Experience on building a lock-free B+-tree in persistent memory. https://pirl.nvsl.io/2019/12/10/experience-on-building-a-lock-free-b-tree-in-persistent-memory/

[84] TPC. 2010. TPC Benchmark C (OLTP) Standard Specification, revision 5.11.

[85] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. 18–32.

[86] Unigen. 2024. Non-Volatile Memory. https://unigen.com/products/nvdimm/.

[87] Viking Technology. 2024. DDR4 NVDIMM. https://www.vikingtechnology.com/non-volatile-memory/ddr4-nvdimm/

[88] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876.

[89] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (Oct. 2016), 49–60.

[90] Tianzheng Wang, Justin Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 461–472.

[91] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 473–488.

[92] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (March 2017), 781–792.

[93] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: Lightweight Parallel Logging for in-Memory Database Management Systems. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 189–201.

[94] Helen Xu, Amanda Li, Brian Wheatman, Manoj Marneni, and Prashant Pandey. 2023. BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-Trees. *Proc. VLDB Endow.* 16, 11 (July 2023), 2976–2989.

[95] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 1629–1642.

[96] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. *Proc. ACM Manag. Data* 1, 2, Article 131 (jun 2023), 26 pages.