Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats

Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, Andrew Pavlo
Massachusetts Institute of Technology, Carnegie Mellon University, Ursa Labs
{litianyu,ngom}@mit.edu,{mbutrovi,wanshenl,pavlo}@cs.cmu.edu,wes@ursalabs.org

ABSTRACT

The proliferation of modern data processing tools has given rise to open-source columnar data formats. These formats help organizations avoid repeated conversion of data to a new format for each application. However, these formats are read-only, and organizations must use a heavy-weight transformation process to load data from on-line transactional processing (OLTP) systems. As a result, DBMSs often fail to take advantage of full network bandwidth when transferring data. We aim to reduce or even eliminate this overhead by developing a storage architecture for in-memory database management systems (DBMSs) that is aware of the eventual usage of its data and emits columnar storage blocks in a universal open-source format. We introduce relaxations to common analytical data formats to efficiently update records and rely on a lightweight transformation process to convert blocks to a read-optimized layout when they are cold. We also describe how to access data from third-party analytical tools with minimal serialization overhead. We implemented our storage engine based on the Apache Arrow format and integrated it into the NoisePage DBMS to evaluate our work. Our experiments show that our approach achieves comparable performance with dedicated OLTP DBMSs while enabling orders-of-magnitude faster data exports to external data science and machine learning tools than existing methods.

PVLDB Reference Format:
Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, Andrew Pavlo. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. PVLDB, 14(4): 534-546, 2021.
doi:10.14778/3436905.3436913

1 INTRODUCTION

Data analysis pipelines allow organizations to extract insights from data residing in their OLTP systems. The tools in these pipelines often use open-source binary formats, such as Parquet [10], ORC [9], and Arrow [4]. Such formats allow disparate systems to exchange data through a common interface without converting between proprietary formats. But these formats target read-only workloads and are not amenable to OLTP systems. Consequently, a data scientist must transform OLTP data with computationally expensive processes, which inhibits timely analysis.

Although recent advances in Hybrid Transactional Analytical (HTAP) DBMSs made it practical to run analytical queries (OLAP) together with OLTP workloads, modern data science pipelines involve specialized frameworks such as TensorFlow, PyTorch, and Pandas. Organizations are also heavily invested in the current data science eco-system of Python tools. The need for DBMSs to efficiently export large amounts of data to external tools will persist. To enable analysis of data upon arrival in a database, and to deliver performance gains across the entire data analysis pipeline, one must improve a DBMS's interoperability with external tools. Previous work explored DBMS data exporting methods and showed that they are encumbered by the inefficient transformation of data from native storage to wire formats [47]. If an OLTP DBMS stores data in a format used by downstream applications, the export cost is just the cost of network transmission. Achieving this is challenging in two ways. Foremost is that most open-source columnar formats are optimized for read/append operations, but an OLTP DBMS needs to perform well for in-place updates. Second, an OLTP DBMS's concurrency control protocol is often co-designed with the storage format to incorporate transactional metadata, such as versions and timestamps. DBMS developers must now implement transactional components that are storage-format-agnostic.

In this paper, we show that it is possible to overcome these challenges, build a performant OLTP system on an open-source columnar format, and support near-zero overhead data export to external tools. We leverage the natural cooling process of data, relaxing the columnar format for transactional throughput while the data is hot and transforming data back to the canonical format when write access becomes infrequent. We integrate this background transformation process with the concurrency control protocol to prevent writers from blocking for an extended period of time. We implemented our storage and concurrency control architecture in **NoisePage** [23] and evaluated its performance. We target Apache Arrow, although our approach is also applicable to other columnar formats. Our results show that we achieve good performance on OLTP workloads operating on the relaxed Arrow format. We also implemented an Arrow export layer for our system and show that it facilitates orders-of-magnitude faster exports to external tools.

To summarize, we make the following contributions:

1. We present the first (to our knowledge) transactional system that operates natively with a popular open-source data format, and discuss design and engineering considerations.
2. We present a novel background transformation algorithm to achieve this that is extensible to other tasks and formats.
3. We evaluate the Arrow-based storage engine of NoisePage and demonstrate its OLTP competitiveness and orders of magnitudes faster data export to downstream Arrow applications.

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436913

The remainder of this paper is organized as follows: we first discuss in Sec. 2 the motivation for this work. We then present our storage architecture and concurrency control in Sec. 3, followed by our transformation algorithm in Sec. 4. In Sec. 5, we discuss how to export data to external tools. We present our evaluation in Sec. 6 and discuss related work in Sec. 7.

2 BACKGROUND

We now discuss challenges in analyzing data stored in OLTP DBMSs with external tools. We begin with the data transformation and movement bottlenecks. We then present a popular open-source format (Apache Arrow) and discuss its strengths and weaknesses.

2.1 Data Movement and Transformation

A data processing pipeline typically consists of a front-end OLTP layer and multiple analytical layers. OLTP engines employ the *n*-ary storage model (i.e., row-store) to support efficient single-tuple operations, while the analytical layers use the decomposition storage model (i.e., column-store) to speed up large scans [25, 31, 40, 44]. Because of conflicting optimization strategies for these two use cases, organizations often combine specialized systems.

The most salient issue with this bifurcated approach is data transformation and movement between layers. This problem is made worse with the emergence of machine learning workloads that load the entire data set instead of a small query result set. For example, a data scientist will (1) execute SQL queries to export data from PostgreSQL, (2) load it into a Jupyter notebook on a local machine and prepare it with Pandas, and (3) train models on cleaned data with TensorFlow. Each step in such a pipeline transforms data into a format native to the target framework: a disk-optimized row-store for PostgreSQL, DataFrames for Pandas, and tensors for TensorFlow. The slowest transformation of all is from the DBMS to Pandas because it retrieves data over the DBMS's network protocol and then rewrites it into the desired columnar format. Many organizations employ costly extract-transform-load (ETL) pipelines that run nightly, introducing delays to analytics.

To better understand this issue, we measured the time it takes to extract data from a DBMS and load it into a Pandas program. We first create a 8 GB CSV file containing the TPC-H `LINEITEM` table (scalefactor 10, 60M tuples), and then load it into PostgreSQL (v10.6) and SAP HANA (v2.0). We then compare four approaches for loading the table into a Python program: (1) PostgreSQL SQL over a Python ODBC connection, (2) PostgreSQL's `COPY` command to export a CSV file to disk and then loading it into Pandas, (3) HANA's `EXPORT` command to write a binary file and then use SAP's Python libraries to load it into Pandas [3], and (4) loading data directly from a buffer already in the Python runtime's memory. The last method represents the theoretical best-case scenario to provide us with an upper bound for data export speed. We pre-load the entire table into PostgreSQL's buffer pool using the `pg_warm` extension. To simplify our setup, we run the Python program on the same machine as the DBMS. We use a machine with 128 GB of memory, of which we reserve 15 GB for shared buffers. We provide a full description of our environment for this experiment in Sec. 6.

The results in Fig. 1 show that ODBC and CSV are orders of magnitude slower than what is possible. This difference is because of

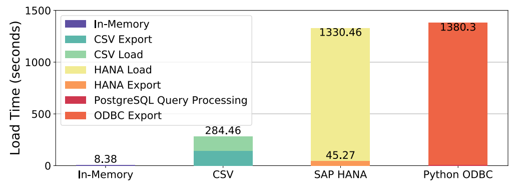
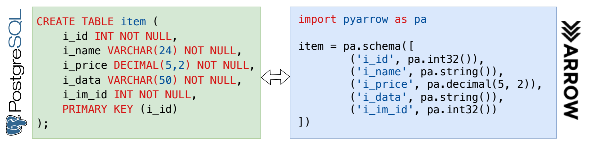


Figure 1: Data Transformation Costs – Time taken to load a TPC-H table into Pandas with different approaches.



```
CREATE TABLE item (
    i_id INT NOT NULL,
    i_name VARCHAR(24) NOT NULL,
    i_price DECIMAL(5,2) NOT NULL,
    i_data VARCHAR(50) NOT NULL,
    i_im_id INT NOT NULL,
    PRIMARY KEY (i_id)
);
```

```
import pyarrow as pa

item = pa.schema([
        ('i_id', pa.int32()),
        ('i_name', pa.string()),
        ('i_price', pa.decimal(5, 2)),
        ('i_data', pa.string()),
        ('i_im_id', pa.int32())
])
```

Figure 2: SQL Table to Arrow – An example of using Arrow's API to describe a SQL table's schema in Python.

the overhead of transforming into a different format and excessive serialization in PostgreSQL's wire protocol. Query processing itself takes 0.004% of the total export time. The rest of the time is spent in the serialization layer and in transforming the data. HANA's export performance is almost as slow as PostgreSQL, despite its HTAP design goal. To better understand the breakdown of HANA's performance, we also show the time it takes for HANA to export the target table to disk using its proprietary binary format. The results show that this only takes about 45 seconds and that most time is spent converting data into the Python format. Optimizing this export process will speed up analytics pipelines.

2.2 Column-Stores and Apache Arrow

The current inefficiency of data export requires us to rethink the data export process and avoid costly data transformations. Lack of interoperability between row-stores and columnar formats is a major source of the overhead. As discussed previously, OLTP DBMSs are row-stores because conventional wisdom says that column-stores are inferior for OLTP workloads. Recent work, however, has shown that column-stores can also support high-performance transactional processing [46, 50]. We propose implementing an OLTP DBMS directly on top of a data format used by analytics tools. To do so, we select a representative format (Apache Arrow) and analyze its strengths and weaknesses for OLTP workloads.

Apache Arrow is a cross-language development platform for in-memory data [4]. In 2015, developers from Apache Drill, Apache Impala, Apache Kudu, Pandas, and others joined together to develop a universal in-memory columnar data format based on their overlapping requirements. Arrow was introduced in 2016 and has since become the standard for columnar in-memory analytics and an interface between heterogeneous systems. There is a growing ecosystem of tools built for Arrow, including APIs for several programming languages and libraries. For example, TensorFlow now integrates with Arrow through a Python module [19].

At the core of Arrow is a columnar memory format for flat and hierarchical data. This format enables (1) fast analytical data processing and vectorized execution, and (2) zero-deserialization data interchange. To achieve the former, Arrow organizes data contiguously in 8-byte aligned buffers and uses separate bitmaps

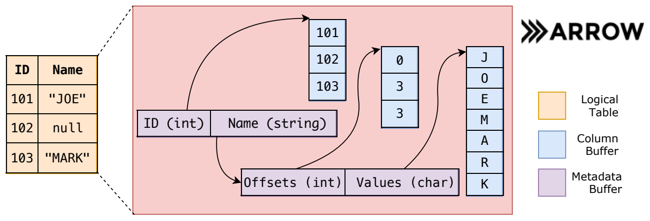


Figure 3: Variable Length Values in Arrow – Arrow represents variable length values as an offsets array into an array of bytes, which trades off efficient mutability for read performance.

for nulls. For the latter, Arrow specifies a standard in-memory representation and provides a C-like data definition language (DDL) for data schema. Arrow uses separate metadata data structures to impose a table-like structure on collections of buffers. An example of this for the TPC-C ITEM table is shown in Fig. 2.

Although Arrow's design targets read-only analytical workloads, its alignment requirement and null bitmaps also benefit write-heavy workloads on fixed-length values. Problems emerge in Arrow's support for variable-length values (e.g., VARCHARs). Arrow stores them as an array of offsets indexing into a contiguous byte buffer. As shown in Fig. 3, the length information is implicitly stored in the starting offset of the next value. This approach is not ideal for updates because of write amplification. Suppose a program updates the value "JOE" to "ANNA" in Fig. 3. It must copy the entire Values buffer to a larger one and update the Offsets array.

The core issue is that a single storage format cannot easily achieve simultaneously (1) data locality and value adjacency, (2) constant-time random access, and (3) mutability [29]. Some researchers have proposed hybrid storage schemes of row-store and column-store to get around this trade-off. Two notable examples are Peloton [28] and H2O [27]. Peloton uses an abstraction layer above the storage engine that transforms cold row-oriented data into a columnar format. In contrast, H2O uses an abstraction layer at the physical operator level and generates code for the optimal format on a per-query basis. Both solutions increase engineering complexity and offer limited speedup in the OLTP scenario (shown in Sec. 6.1). Therefore, we argue that while it makes sense to optimize the data layout differently based on access patterns, column-stores are good enough for both OLTP and OLAP use cases.

3 SYSTEM OVERVIEW

We now present NoisePage's architecture. We first discuss how its *transaction engine* is minimally intrusive to Arrow's layout. We then describe its table organization, along with its garbage collection and recovery components. For simplicity, we assume that data is fixed-length; we discuss variable-length data in the next section.

3.1 Concurrency Control Protocol

A requirement for our system is that transactional and versioning metadata are separate from the actual data; interleaving them complicates the mechanism for exposing Arrow data to external tools. NoisePage achieves this through multi-versioned delta storage [30]. We now describe the architecture illustrated in Fig. 4.

The DBMS stores tuple deltas in transaction-local buffers instead of Arrow storage. The system uses one extra column to store

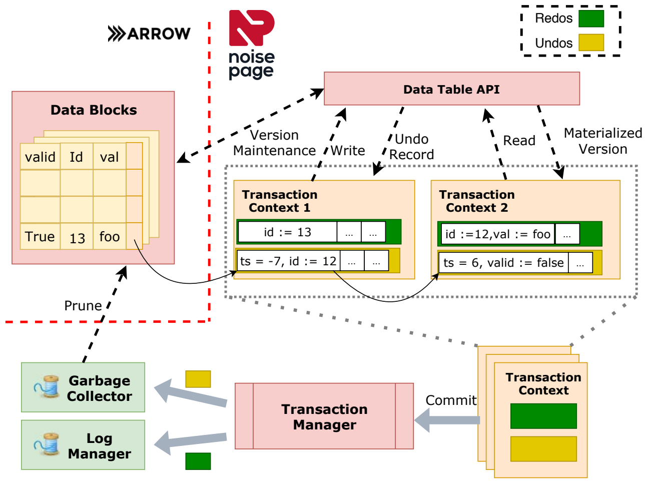


Figure 4: System Architecture – An example of how transactions modify the database and populate Arrow-compatible data blocks in NoisePage.

pointers to the head of the version chain (or null if no version), hidden from external readers. Transactions interact with Arrow exclusively through the Data Table API that abstracts away the underlying storage. For readers, the Data Table layer traverses the version chain to reconstruct the correct version of the tuple and materializes that version. For writers, the Data Table layer copies the before-image of modified tuple attributes into the transaction's local buffer and installs it onto the version chain before writing the changes to Arrow storage in-place. The DBMS handles deletes and inserts analogously through manipulating the tuple's validity bit.

Each transaction maintains two local buffers: (1) undos and (2) redos. They represent the before-image and after-image of modified tuple attributes, respectively. Clients write their intended changes to the redo-buffer for logging purposes. The undo buffer holds version deltas and serves to track the write set of a transaction. We give an example version chain in Fig. 4. Transaction 2 first inserts the tuple (id=12, val=''foo'') and populates its redo buffer. The version chain points to the entry on the undo buffer of Transaction 2, noting that the tuple did not exist before. When Transaction 1 modifies the tuple to (id=13), it first writes down the previous value of 12 in its undo buffer, adds the record to the version chain, and then writes 13 to the underlying data block. To accommodate arbitrarily large write sets, the DBMS must resize dynamically undo buffers while preserving the memory address of earlier entries, as they are pointed to on the version chains. NoisePage implements undo buffers as a linked list of fixed-sized segments (currently 4096 bytes) and incrementally adds new segments as needed. The system later passes the undo and redo buffers to its garbage collector and logging components, as we discuss in Secs. 3.3 and 3.4.

We now discuss the concurrency control mechanism of NoisePage on top of our storage architecture. We implement a variant of the Optimistic Concurrency Control protocol [55] with the centralized transaction engine component. The transaction engine assigns each transaction a timestamp pair (*start*, *commit*) that it generates from the same counter. When a transaction starts, *commit* is the same as *start* but with its sign bit flipped to denote that the transaction is uncommitted. Each update on the version chain stores the transaction's *commit* timestamp. Readers reconstruct their respective versions by copying the latest version and then traversing the version chain and applying before-images until it sees a timestamp less than its *start*. Because the system uses unsigned comparison for timestamps, uncommitted versions are never visible. The system disallows write-write conflicts to avoid cascading rollbacks. Using the example in Fig. 4, a reader with timestamp 8 would first read (id=13), before chasing the version pointer to find transaction 1's undo record with timestamp -7. The reader detects that the value is uncommitted, and applies the delta (id=12). Upon seeing the undo record of Transaction 2, it returns with the correct value for id as the timestamp of 6 is smaller than its timestamp.

When a transaction commits, the DBMS uses a small critical section to obtain a commit timestamp, update delta records' commit timestamps, and add them to the log manager's queue. During the critical section, new transactions must not start to avoid fractured reads, but existing transactions can continue to perform operations, commit, or abort concurrently. For aborts, the system uses the transaction's undo records to roll back the in-place updates. It cannot unlink records from the version chain, however, due to potential race conditions. If an active transaction copies a new version before the aborting transaction that modified it rolls back, then the reader traverses the version chain with the undo record already unlinked and to identify that the aborted version is visible.

A simple check that the version pointer does not change while the reader makes a copy is insufficient in this scenario as the DBMS can encounter the "A-B-A" problem. That is, an abort might occur between the checks and change the value of the tuple, but the reader cannot observe this through the version pointer. To avoid this issue, the DBMS instead restores the correct version before "committing" the undo record by flipping the sign bit on the version's timestamp. This record is redundant for any readers that obtained the correct copy and fixes the copy of readers with the aborted version. NoisePage achieves Snapshot Isolation with this implementation; one can replace the critical section with a validation phase to achieve full serializability [46].

With NoisePage's storage scheme, its transaction engine only reasons about tuple visibility using delta records and the version column. This abstraction comes at a cost for readers, as they are forced to materialize tuples early, which degrades scan performance. For many workloads, only a small fraction of the database is versioned at any point in time. As a result, the DBMS can ignore checking the version column for every tuple and scan large portions of the database in place. We discuss this further in Sec. 4.

3.2 Blocks and Physiological Identifiers

Separating tuples and transactional metadata introduces another challenge: the system requires globally unique tuple identifiers to associate the two pieces that are not co-located. Physical identifiers (e.g., pointers) are ideal for performance but work poorly with column-stores because a tuple does not physically exist at a single location. Logical identifiers, on the other hand, must be translated into a memory location through a lookup (e.g., hash table). This translation step is a severe bottleneck for OLTP workloads because it potentially doubles the number of memory accesses per tuple. To solve this, our DBMS organizes storage in 1 MB blocks and uses a physiological scheme to identify tuples. The DBMS arranges data in each block similar to PAX [26], where all attributes of a tuple are within the same block. Every block has a layout object that consists of (1) the number of slots within a block, (2) a list of attributes sizes, and (3) the location offset for each column from the head of the block. Each column and its bitmap are aligned at 8-byte boundaries. The system calculates layout once for a table when the application creates it and uses it to handle every block in the table.

Every tuple is identified by a `TupleSlot` that is a combination of (1) the physical memory address of the block with the tuple and (2) its logical offset in the block. Combining these with the pre-calculated block layout, the DBMS computes the physical pointer to each attribute in constant time. To pack both values into a single 64-bit value, we use the C++11 keyword `alignas` to force the system to store all blocks at 1 MB boundaries within its address space. A pointer to a block will always have its lower 20 bits be zero, which the DBMS uses to store the offset. There are enough bits because there can never be more tuples than there are bytes in a block.

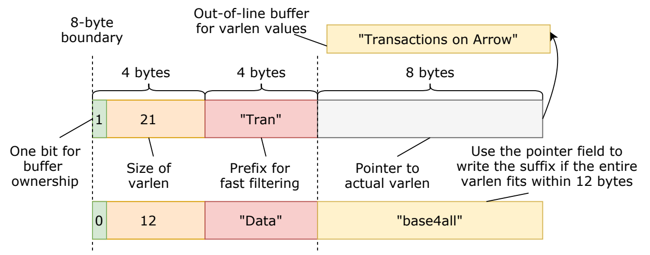
3.3 Garbage Collection

The garbage collector (GC) [42, 43, 53, 56] is responsible for pruning version chains and freeing the associated memory. The DBMS handles the recycling of deleted slots during the transformation to Arrow (Sec. 4.3). Because the DBMS stores versioning information in transactions' buffers, the GC only examines transaction objects.

At the start of each run, the GC first checks the transaction engine's transactions table for the oldest active transaction's *start* timestamp; changes from transactions committed before this timestamp are no longer visible and are safe for removal. The GC inspects all such transactions to compute the set of `TupleSlots` that have invisible records in their version chains and then truncates them exactly once. Deallocating objects is unsafe at this point, however, as concurrent transactions may be reading the unlinked records. To address this, GC obtains a timestamp from the transaction engine that represents the time of unlink. Any transaction starting after this time cannot possibly access the unlinked record; the records are safe for deallocation when the oldest running transaction in the system has a larger *start* timestamp than the unlink time. Our approach is similar to an epoch-protection mechanism [32] and is generalizable to ensure thread-safety for other aspects of the DBMS as well. We expose the functionality to register an action with the GC, such that the action is run only after all transactions concurrent to the registration of said action has ended.

3.4 Logging and Recovery

Our system achieves durability through write-ahead logging and checkpoints [34, 45]. Logging in our DBMS is analogous to the GC process described above. Each transaction maintains a redo buffer for physical after-images. Each transaction writes changes to its redo buffer in the order that they occur. At commit time, the transaction appends a commit record to its redo buffer and adds itself to the DBMS's flush queue. The log manager asynchronously serializes the changes from these buffers into an on-disk format before flushing to persistent storage. The system relies on an implicit ordering of the records according to their respective transaction's *commit* timestamp instead of log sequence numbers.



8-byte boundary
Out-of-line buffer for varlen values
"Transactions on Arrow"
4 bytes
4 bytes
8 bytes
1
21
"Tran"
One bit for buffer ownership
Size of varlen
Prefix for fast filtering
Pointer to actual varlen
Use the pointer field to write the suffix if the entire varlen fits within 12 bytes
0
12
"Data"
"base4all"

Figure 5: Variable-Length Value Storage – The system stores variable-length values as a 16-byte column in a block.

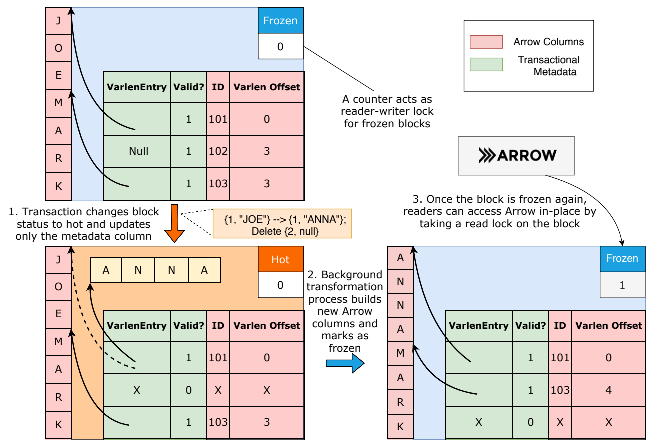
Similar to undo buffers, these redo buffers consist of buffer segments drawn from a global object pool. The system flushes out redo records incrementally before the transaction commits. In the case of an abort or crash, the transaction's commit record is not written, and the recovery process ignores it. In our implementation, we limit the redo buffer to a single buffer segment and observe moderate speedup due to better cache performance from more reuse.

The rest of the system considers a transaction as committed as soon as its commit record is added to the flush queue. All future operations on the transaction's write-set are speculative until its log records are on disk. The system assigns a callback to each committed transaction for the log manager to notify when the transaction is persistent. The DBMS refrains from sending a transaction's result to the client until the log manager invokes its callback. With this scheme, a transaction's modifications that speculatively accessed or updated the write-set of another transaction are not published until the log manager processes their commit record. We implement callbacks by embedding a function pointer in the commit record; when the log manager writes the commit record, it adds that pointer to a list of callbacks to invoke after the next `fsync`. The DBMS requires read-only transactions also to obtain a non-persisted commit record to guard against the anomaly shown above.

Log records identify tuples on disk using `TupleSlots`, even though pointers are invalid on reboot. The system maintains a mapping table between old tuple slots to their new physical locations in recovery mode. A checkpoint in the system is a consistent snapshot of all blocks that have changed since the last checkpoint; because of multi-versioning, it suffices to scan blocks transactionally to produce a consistent checkpoint. The system records the timestamp of the scanning transaction after the checkpoint is finished as a record in the write-ahead log. Upon recovery, the system is guaranteed to recover to a consistent snapshot, from which it can apply all changes where the commit timestamp is after the latest recorded checkpoint. It is also possible to permit semi-fuzzy checkpoints [48] where all checkpointed tuples are committed, but not necessarily from the same snapshot.

4 BLOCK TRANSFORMATION

As discussed in Sec. 2.2, the primary obstacle to running transactions on Arrow is write amplification. Our system uses a relaxed Arrow format to achieve good write performance and then uses a lightweight transformation step to put a block into the full Arrow format once it is cold. We now describe this modified format, present our algorithm for transforming them, and discuss the important assumptions and implementation details in NoisePage.



Frozen
A counter acts as reader-writer lock for frozen blocks
Arrow Columns
Transactional Metadata
ARROW
1. Transaction changes block status to hot and updates only the metadata column
{1, "JOE"} --> {1, "ANNA"}; Delete {2, null}
Hot
2. Background transformation process builds new Arrow columns and marks as frozen
3. Once the block is frozen again, readers can access Arrow in-place by taking a read lock on the block
VarlenEntry
Valid?
ID
Varlen Offset

Figure 6: Relaxed Columnar Format – The system briefly allows non-contiguous memory to support efficient updates.

4.1 Relaxed Columnar Format

Typical OLTP workloads modify only a small portion of a database at any given time, while the other parts of the database are mostly accessed by read-only queries [41]. Therefore, for the hot portion, we can trade off read speed for write performance at only a small impact on the overall read performance of the DBMS. To achieve this, we modify the Arrow format for update performance in the hot portion. We detail these changes in this subsection.

Arrow has two sources of write amplification: (1) it disallows gaps in a column, and (2) it stores variable-length values consecutively in a single buffer. Our relaxed format adds a validity bitmap in the block header and additional metadata for each variable-length value to overcome them. As shown in Fig. 5, within a `VarlenEntry` field, the system maintains 4 bytes for size and 8 bytes for a pointer to the underlying value. Each `VarlenEntry` is padded to 16 bytes for alignment reasons, and the additional 4 bytes stores a prefix of the value. If a value is shorter than 12 bytes, the system stores it entirely within the object, writing into the pointer. Transactions only access the `VarlenEntry` instead of Arrow storage directly. Relaxing adherence to Arrow's format allows the system to only write updates to `VarlenEntry`, turning a variable-length update into a constant-time fixed-length one, as shown in Fig. 6.

Any readers accessing Arrow storage will be oblivious to the update in `VarlenEntry`. The system adds a *status flag* and *counter* in block headers to coordinate access. A block in NoisePage can be in one of three states – hot, cooling, or frozen. Hot blocks are actively worked on by transactions, whereas frozen blocks are available for in-place scans in the Arrow format; cooling blocks are in the process of being transformed. The access counter on each block functions as a shared latch – each in-place reader adds one to the counter when starting a scan and subtract one when finished. When a transaction updates a frozen block, it first sets that block's status flag to **hot**, forcing any future readers to materialize instead of reading in-place. It then spins on the counter and waits for lingering readers to leave the block before proceeding with the update. Once the block is hot, transactional access elide latch protection and rely on the MVCC implementation for thread safety. Other than flipping the flag, there is no transformation process required for a transaction to modify a

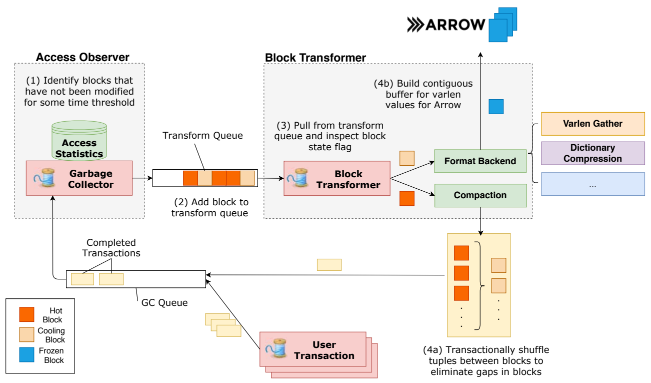

ARROW
Access Observer
(1) Identify blocks that have not been modified for some time threshold
Access Statistics
Garbage Collector
Transform Queue
(2) Add block to transform queue
Block Transformer
(3) Pull from transform queue and inspect block state flag
(4b) Build contiguous buffer for varlen values for Arrow
Block Transformer
Format Backend
Compaction
Varlen Gather
Dictionary Compression
...
Completed Transactions
GC Queue
Hot Block
Cooling Block
Frozen Block
User Transaction
(4a) Transactionally shuffle tuples between blocks to eliminate gaps in blocks

Figure 7: Transformation to Arrow – NoisePage implements a pipeline for lightweight in-memory transformation to Arrow.

frozen block because our relaxed format is a generalization of the original Arrow format. Once a block is hot, it remains so until a background process transforms it back to full Arrow compliance.

We now provide an overview of our transformation algorithm, also illustrated in Fig. 7. There are two components of our transformation pipeline, the *access observer* and *block transformer*, shown as boxes with dashed lines in Fig. 7. The access observer piggy-backs on the DBMS's normal garbage collection to inspect recent changes and identify any candidates for transformation to push onto a queue (Sec. 4.2). The block transformer polls from the queue. For correctness reasons, the transformer processes each block at least twice before emitting them as Arrow blocks. The first pass is transactional and rearranges tuples within blocks so that they are contiguous. This transaction passes through the access observer and prompts the access observer to enqueue the block again (Sec. 4.3).

4.2 Identifying Cold Blocks

The DBMS maintains statistics about each block to determine if it is cooling. Collecting them as transactions operate on the database adds overhead to the critical path [33, 36], which is unacceptable for OLTP workloads. Our system trades the quality of such statistics for better performance and then accounts for potential mistakes from this in our transformation algorithm.

A simple heuristic is to mark blocks that have not been modified for some threshold time as cold for each table. Instead of measuring this on the transaction's critical path, our system takes advantage of the GC's scan through undo records (Sec. 3.3). From each undo record, the system obtains the modification type (i.e., delete, insert, update) and the corresponding `TupleSlot`. Time measurement, however, is difficult because the system cannot measure how much time has elapsed between the modification and invocation of the GC. The DBMS instead approximates this by using a coarse-grained counter that periodically increments in GC (e.g. every 10 ms). If transactions have a lifetime shorter than the frequency of this "GC clock", the approximated time is never earlier than the actual modification and is late by at most one tick, which is good enough for short-lived OLTP transactions [51]. Once the system identifies a cold block, it adds the block to a queue for background processing.

Under this scheme, one thread may identify a block as cold by mistake when another thread is updating it due to delays in access observation. The DBMS reduces the impact of this by ensuring

```
groups := {};
while true do
    block := transformQueue.Dequeue();
    switch block.status do
        case hot do
            AssignGroup(groups, block);
        case cooling do
            if not CheckForLiveVersions(block) then
                break;
            if not block.status.CompareAndSwap(cooling, freezing) then
                break;
            Format (block);
            block.status = frozen;
    for group in groups ready for compaction do
        begin txn;
        if Compact (group, txn) then
            for block in group do
                block.status = cooling;
            commit txn;
        else
            abort txn;
```

Algorithm 1: Pseudo-code for Block Transformer

that the transformation algorithm is fast and lightweight. There are two failure cases: (1) a user transaction aborts due to conflicts with the transformation process or (2) the user transaction stalls. There is no way to safely eliminate both cases. Our solution is a two-phase algorithm. The first phase is transactional and operates on a microsecond scale, minimizing the possibility of aborts. The second phase eventually takes a block-level lock for a short critical section but yields to user transactions whenever possible.

4.3 Transformation Algorithm

Once the system identifies cooling blocks, it performs two transformation passes to prepare the block for Arrow readers. The DBMS first needs to compact each block to eliminate any gaps, and then copy variable-length values into a new contiguous buffer for the Arrow varlen representation. There are three approaches to ensure safety with concurrent transactions: (1) block copying, (2) transactional operations, or (3) block-level locks. None of these is ideal. The first is expensive, especially when most of the data is not changed. The second adds additional overhead and increases aborts. The third stalls user transactions and limits concurrency in the typical case even without transformation. As shown in Fig. 7, NoisePage uses a hybrid two-phase approach that combines transactional tuple movement and raw operations under exclusive access, which is orchestrated with a novel multi-stage locking scheme that cooperates with GC to guard against races. We extend the block status flag with two additional values: cooling and freezing. The former indicates that the transformation thread intends to lock, while the latter serves as an exclusive lock that blocks user transactions. Alg. 1 contains the pseudo-code for this operation.

The block transformer continually polls from the transform queue for new blocks to process. When a hot block arrives, the block transformer assigs it to a *compaction group*, a collection of blocks with the same layout. Within a group, the system uses tuples from less-than-full blocks to fill gaps in others and recycle blocks when they become empty. Larger compaction groups are more memory-efficient but also slower to compact. The DBMS uses one transaction per group in this phase to perform all operations;

```
cg := compaction_group.blocks;
txn := compaction_group.txn;
sort cg by #empty slots;
for (taker = 0, giver = cg.size - 1; taker ≤ giver and taker < cg.size; taker++) do
    for unfilled slots in cg[taker] do
        tuple := last filled tuple in cg[giver];
        txn.Delete(tuple);
        txn.Insert(tuple, slot);
        if txn has conflict then
            return false;
        if cg[giver] is empty then
            add to GC for later deallocation if compaction succeeds;
        if taker == giver and taker contiguous then
            return true;
return true;
```

Algorithm 2: Pseudo-code for Compact

moving is equivalent to deleting and then inserting. If the transaction executes without conflicts, it marks the block as cooling and commits the transaction. User transactions compare-and-swap the flag back to hot when modifying a cooling block.

The system now takes a cooling block and formats it into Arrow. The core challenge here is that user transactions can trigger a check-and-miss race — the flip to cooling can be interleaved between a user's check for the flag and subsequent modification. NoisePage guards against this with the GC, which does not prune any versions visible to running transactions. Because a user transaction must be concurrent with compaction to be susceptible to the race, the compaction transaction's versions must remain. Therefore, the block transformer safely flips the block status to freezing only if it sees no versioned entry at the end of the scan. Any modifications concurrent to the scan changes the status to hot, which will be detected by the final compare-and-swap to freezing.

After the transformation algorithm obtains exclusive access to the block, it scans each variable-length column to concatenate values into a contiguous buffer and update pointers without transactional protection. In the same pass, it also computes metadata information, such as null count into the Arrow block header. When the process is complete, the system marks the block as frozen and allow in-place readers. Although transactional writes are not allowed, reads still proceed as the formatting phase changes only the physical location of values and not the logical content of the table. Because a write to any aligned 8-byte address is atomic [2], reads are never unsafe as the DBMS aligns all attributes within a block.

Lastly, we describe how to shuffle tuples within a compaction group (`Compact` from Alg. 1); pseudo-code is given in Alg. 2. At the end of this routine, tuples in the group are logically contiguous; a group consisting of t tuples with b blocks each with s slots now have $\lfloor \frac{t}{s} \rfloor$ many blocks completely filled, one block filled from beginning to the $(t \mod s)$-th slot, and all others empty. The DBMS first sorts the blocks by the number of empty slots and then fills the empty slots from earlier blocks with tuples from later blocks one-by-one.

We measure the efficiency of our algorithm by the number of movements it performs; each movement can trigger index updates, which have performance implications. We show that the above algorithm is at most $(t \mod s)$ movements worse than optimal. It selects a block set F to be the $\lfloor \frac{t}{s} \rfloor$ blocks that are filled in the final state and a block p that is partially filled and hold $t \mod s$ tuples. The rest of the blocks, E, are left empty. It then fills all gaps within $F \cup \{p\}$ using tuples from $E \cup \{p\}$, and reorder tuples within p to make them contiguous. Let Gap_f be the set of unfilled slots in a block f, Gap'_f be the set of unfilled slots in the first $t \mod s$ slots in a block f, $Filled_f$ be the set of filled slots in f, and $Filled'_f$ be the set of filled slots not in the first $t \mod s$ slots in f. Then, for any valid selection of F, p, and E,

$$|Gap'_p| + \Sigma_{f \in F}|Gap_f| = |Filled'_p| + \Sigma_{e \in E}|Filled_e|$$

because there are only t tuples in total. Therefore, an optimal movement is any one-to-one movement between $Filled'_p \cup \bigcup_{e \in E} Filled_e$ and $Gap'_p \cup \bigcup_{f \in F} Gap_f$. The algorithm constructs F by picking the $\lfloor \frac{t}{s} \rfloor$ blocks with the most filled slots. Every gap in F needs to be filled with one movement, and our selection of F results in fewer movements than any other choice. In the worst case, our chosen p, which is the block with the next most filled slots, is empty in the first $(t \mod s)$ slots. The optimal one is filled, resulting in at most $(t \mod s)$ movements from the optimal for our algorithm. To achieve the optimal solution, the algorithm needs to scan through the blocks to find an optimal p by trying every possible candidate. From our experiments in Sec. 6, we observe only a marginal reduction in movements, which does not justify the overhead.

4.4 Additional Considerations

Now that we have presented our algorithm for transforming cold blocks into Arrow, we demonstrate its flexibility by discussing alternative formats for our transformation algorithm. We also give a more detailed description of the issues in memory management and scaling for larger workloads.

Alternative Formats: It is possible to change the formatting phase to emit a different format, although the algorithm performs best if the target format is close to our transactional representation. For example, the system can emit Parquet files by encoding a block and writing to disk at the end of the formatting phase, while retaining the memory content for efficient read access. To illustrate this capability, we implement an alternative columnar format with dictionary compression [38] similar to Parquet [10] and ORC [9]. The system instead creates a dictionary and an array of dictionary codes. The only difference is that within the critical section of the formatting phase, the algorithm now scans through the block twice – the first to build a dictionary corpus and replace pointers within `VarlenEntrys` to point to dictionary words and the second to sort the dictionary and build an array of dictionary codes.

Workload Assumptions: Our algorithm assumes that blocks are modified frequently for a short time before cooling down and becoming read-only. For write-mostly workloads, blocks are never considered cooling by the access observer, and therefore our scheme adds no benefit or overhead. It is conceivable, however, for some workloads to break this assumption by periodically switching between read-only and write-heavy workloads on a set of blocks. In this case, performance impact on transactions is mitigated as our transformation algorithm completes within milliseonds and allows writers to quickly update frozen blocks in place by resetting the flag; this is in contrast to earlier work [41] that are more heavy-weight and only allows for read-copy-update to frozen blocks.

Memory Management: Because the algorithm never blocks readers, the system cannot deallocate memory after the transformation process as its contents are visible to concurrent transactions. In the compaction phase, because writes are transactional, the GC can handle memory management. When moving tuples, the system makes a deep copy of the variable-length values to avoid reasoning about buffer ownership transfer. In the gathering phase, we extend our GC to accept arbitrary actions associated with a timestamp in the form of a callback, which it promises to invoke after the oldest alive transaction in the system is started after the given timestamp. As discussed in Sec. 3.3, this is similar to epoch protection [32]. The system registers an action that reclaims memory for this gathering phase with a timestamp that the compaction thread takes after it completes all of its in-place modifications. This delayed reclamation ensures no transaction reads freed memory.

Scaling Transformation and GC: A single GC or transformation thread cannot keep up with transaction throughput from many worker threads, and the DBMS can leverage partitioning opportunities for parallelization. For GC, the DBMS uses a transaction's identifier to assign it to a GC thread. Although the version chain pruning is thread-safe, multiple GC threads pruning the same chain can do so with an out-of-sync view of the current safe timestamp, and incorrectly deallocate different parts of the chain. Each GC thread starts an empty transaction in between refreshes of the global safe timestamp, which prevents other threads from deallocating parts of the chain it traverses. To parallelize transformation, the DBMS spawns multiple threads to poll from the same underlying queue. Because the transformation is independent across compaction groups, the threads never interfere with each other.

5 EXTERNAL ACCESS

Now that we have described how the DBMS converts data blocks into the Arrow format, we discuss how to expose access to external applications. We argue that native Arrow storage can benefit data pipeline builders, regardless of whether they take a "data ships to compute" approach or the opposite. We also present strategies for using native Arrow storage to integrate with downstream pipelines.

5.1 Data Export

The least intrusive method of integrating NoisePage with a data science eco-system is to maintain a data-ships-to-compute model, and speed up the data export process. We describe two ways to achieve this and empirically evaluate them in Sec. 6.

Improved Wire Protocol: There are still good reasons for applications to interact with the DBMS exclusively through a SQL interface (e.g., developer familiarity, existing ecosystems). As [47] pointed out, using column batches instead of rows in the wire format can increase performance substantially. Arrow data organized by block is naturally amenable to such wire protocols. However, replacing the wire protocol with Arrow does not achieve the full potential of the speed-up from our storage scheme. This is because the DBMS still serializes data into its wire format, and the client must parse the data. These two steps are not necessary if both the DBMS and client are natively Arrow. The DBMS should be able to send stored data directly onto the wire and land them in the client program's workspace, without writing to or reading from a wire format. For this purpose, Arrow provides a native RPC framework based on gRPC called Flight [5] that avoids serialization when transmitting data, with work in progress to define a standard protocol for sending SQL queries and results [22] . Flight enables our DBMS to send a large amount of cold data to the client in a zero-copy fashion. When most data is cold, Flight transmits data significantly faster than real-world DBMS protocols. To handle hot data, the system needs to start a transaction and materialize a snapshot of the block before invoking Flight. Even in this case, we observe that Flight still performs no worse than the state-of-the-art [47].

Shipping Data with RDMA: To achieve further speed-up, one can consider Remote Direct Memory Access (RDMA) technologies. RDMA bypasses the OS's network stack and permits high-throughput, low-latency transfer of data. Either the client or the DBMS can RDMA into the other's memory, and we sketch both.

The DBMS server can write data to the client's memory through RDMA (i.e., client-side RDMA). Under this scheme, the server retains control over access to its data, and no modification to the concurrency control scheme is required. Aside from increased data export speed, another benefit of using a client-side approach is that the client's CPU is idle during RDMA operations. Thus, the client starts working on partially available data, effectively pipelining data processing. To achieve this, the DBMS sends messages for partial availability of data periodically. This approach reduces the network traffic close to its theoretical lower-bound but still requires additional processing power on the server to service the request.

For workloads that require no server-side computation, allowing clients to read the DBMS's memory and bypass the DBMS CPU when satisfying bulk export requests. With this approach, the OLTP DBMS no longer needs to divide its CPU resources between serving transactional workloads and bulk-export jobs, but this can lead to other problems. Firstly, the DBMS loses control over access to its data as the client bypasses its CPU, which makes it difficult to lock the Arrow block to prevent updates. In addition to this extra complexity in concurrency control, this approach also requires that the client knows beforehand the blocks it needs to access, which requires a separate code path to convey this information.

5.2 Early ETL and Compute Shipping

To achieve further speed on top of fast data export, we look to reduce the amount of data that requires export. This possible if we pull computation later in the ETL pipeline (e.g., filtering) up into the DBMS layer. We now sketch the possibilities and discuss the benefits and challenges in doing so as future work.

Early ETL: ETL tasks periodically read OLTP data and reorganize it into the expected format of downstream analytics. We now give examples of how to modify NoisePage to support three basic ETL functionalities during the transformation process: (1) *filtering*, (2) *aggregation*, and (3) *deduplication*. There are two potential ways to support filtering: either compute a bit vector of predicate evaluation results against the tuples during the formatting pass or customize the compaction algorithm to group tuples according to their predicate evaluation results. The former adds minimum overhead, but rules out the use of RDMA for data export, as filtered data may not be contiguous within each block. The latter scheme adds more overhead to compaction, and cannot accommodate more than one predicate. To support aggregation, one may again take advantage of the formatting pass on each block to pre-aggregate data per block. For de-duplication, one can use the dictionary-encoding formatting pass without generating dictionary codes.

Shipping Computation to Data: Server- and client-side RDMA allow external tools to access data with extremely low data export overhead, but they require specialized hardware and are only viable when the application has such connection to the DBMS, which is unlikely for a data scientist working on a personal workstation. A deeper issue is that using RDMA requires the DBMS to "pull" data to the computational resources that execute the query. The limitations of this approach are widely known, especially in the case where server-side filtering is difficult to achieve.

If we adopt a "push" approach, then using native Arrow storage in the DBMS does not provide benefits to network speed. Instead, we leverage Arrow as an API between the DBMS and external tools to improve programmability. Because Arrow is a standardized memory representation of data, if external tools support Arrow as input, then it is possible to run the program on the DBMS by replacing Arrow references with mapped memory images from the DBMS process. This approach introduces a new set of problems involving security, resource allocation, and software engineering. By making an analytical job portable across machines, this also allows dynamic migration of a task to a different server. In combination with RDMA, this leads to true serverless HTAP processing where the client specifies a set of tasks, and the DBMS dynamically assembles a heterogeneous pipeline with low data movement cost.

6 EVALUATION

We next present an experimental analysis of our system. We implemented our storage engine in the NoisePage DBMS [23]. We performed our evaluation on a machine with a dual-socket 10-core Intel Xeon E5-2630v4 CPU, 128 GB of DRAM, and a 500 GB Samsung 970 EVO Plus SSD. For each experiment, we use `numactl` to interleave memory allocation on available NUMA regions. All transactions execute as stored procedures. We run each experiment ten times and report the average. We first evaluate our OLTP performance and quantify performance interference from the transformation process. We then provide a set of micro-benchmarks to study the transformation algorithm in detail. Finally, we compare data export performance in our system against current approaches.

6.1 OLTP Performance

We measure the DBMS's OLTP performance to demonstrate the viability of our storage architecture and that our transformation process is lightweight. We use TPC-C [52] in this experiment with one warehouse per client. NoisePage uses the OpenBw-Tree for indexes [54]. We report the DBMS's throughput and the state of blocks at the end of each run. We use `taskset` to limit the number of available CPU cores so that configurations with transformation turned on do not receive additional cores. The system has one logging thread, one transformation thread, and one GC thread for every 8 worker threads. We deploy the DBMS with three transformation configurations: (1) disabled, (2) variable-length gather, and (3) dictionary compression. For trials with NoisePage's block transformation enabled, we use an aggressive threshold time of 10 ms and

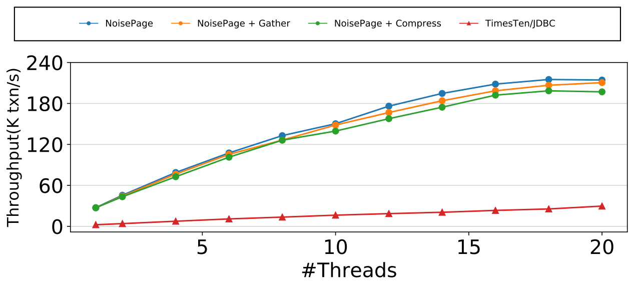


Figure 8: OLTP Performance – Runtime measurements for NoisePage and TimesTen for the TPC-C workload when varying the number of threads.

only target the tables that generate cold data: `ORDER`, `ORDER_LINE`, `HISTORY`, and `ITEM`. In each run, the compactor attempts to process all blocks from the same table in the same group.

The results in Fig. 8 show that the DBMS achieves good scalability and incurs little overhead from the transformation (at most 10%). The interference is more prominent as the number of workers increases due to more work for the transformation thread. At 20 worker threads, the DBMS's scaling degrades because our machine only has 20 physical CPU cores. Dictionary compression has a slightly larger impact because it is computationally more intensive. We also measured the abort rate of transactions and the number of transactions stalled due to the transformation process. We did not observe a statistically significant change in abort rates and a negligible number of stalled transactions (<0.01%). Almost all blocks that the DBMS could transform are in frozen at the end of the benchmark, with the exception of dictionary compression under high thread count. This is because the compression process is an order of magnitude slower than gathering, as we will show in Sec. 6.2. The transformation process yields resources to user transactions in this situation and does not result in a significant drop in transactional throughput. The DBMS can parallelize transformation by partitioning based on block address when the transformation thread is lagging behind. We ran the benchmark with additional transformation threads in this configuration to achieve full transformation and observe an additional ̃15% reduction in throughput.

As a baseline, we also deploy TimesTen Classic (v18.1) in-memory DBMS [24]. We measure TimesTen's TPC-C performance with its WAL disabled (`DurableCommits`=false) using OLTP-Bench [35]. Fig. 8 also shows that TimesTen performance is flat as the number of execution threads increases. We note here that because NoisePage runs TPC-C as stored procedures while TimesTen executes over JDBC, this speedup does not mean that it is superior to TimesTen. Instead, this shows that in a realistic OLTP deployment, transactional processing is not the only bottleneck on the storage performance. Overall, our storage architecture is competitive in performance, and our transformation technique adds a negligible overhead.

Row vs. Column: To investigate the impact of using a column-store for OLTP, we compare our storage architecture against a row-store, which we simulate by declaring a single, large column that stores all of a tuples' attributes contiguously. Each attribute is an 8-byte fixed-length integer. We fix the number of threads executing queries and scale up the number of attributes per tuple.

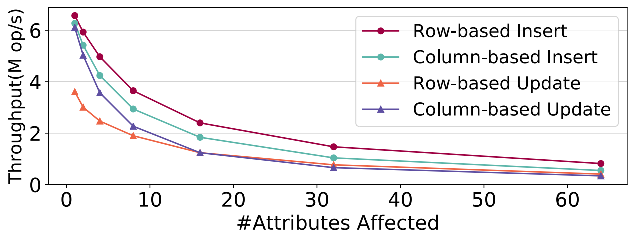


Figure 9: Row vs. Column – Measurements of raw storage speed of NoisePage, row vs. column, varying number of attributes modified. For inserts, the x-axis is the number of attributes of the inserted tuple; for updates, it is the number of attributes updated.

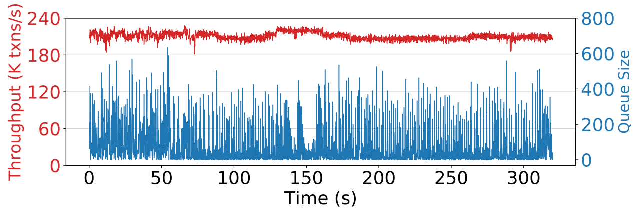


Figure 10: Compaction Queue Size Over Time – Measurements of compaction queue size and transactional throughput over time for NoisePage.

The workload comprises either (1) insert or (2) update queries (10 million each). We ignore index maintenance in our measurements as this overhead is the same for both storage models. The results in Fig. 9 show that the two approaches do not exhibit a large performance difference. For the insert workload, the gap never exceeds 40%. For the update workload, a column-store outperforms row stores when the number of attributes copied is small due to its smaller memory footprint. As the number of attributes grows, the row-store becomes slightly faster than the column-store. These results show that an optimized row-store is unlikely to provide a compelling performance improvement in an in-memory setting.

Throughput/Compaction Over Time: To showcase the behavior of our system over time, we plot the observed backlog of blocks of the transformation thread over time in Fig. 10. For this experiment, we use the varlen gather backend for our compaction thread, with 20 threads issuing TPC-C queries in the same manner as before. We allocate two compaction threads and three garbage collection threads for this experiment. We measure the total compaction queue size every 100 ms, along with the transactional throughput. We show the DBMS running for five minutes, with start-up and wind-down truncated for clarity. Fig. 10 shows that the compaction algorithm is able to keep up with the throughput. The fluctuation observed in transactional throughput is due to CPU scaling and does not appear correlated with compaction queue size. We observe a similar pattern for dictionary compression.

6.2 Transformation to Arrow

We next evaluate our transformation algorithm and analyze the effectiveness of each sub-component. We use micro-benchmarks to demonstrate the DBMS's performance when transforming blocks to Arrow. The database has a single table of ~16M tuples with two columns: (1) a 8-byte fixed-length column and (2) a variable-length column with values between 12–24 bytes. Under this layout, each block holds ~32K tuples. We also ran the same experiments on a table with more columns or larger varlens, but did not observe a difference in trends. An initial transaction populates the table and inserts empty tuples at random to simulate deletion.

Throughput: For this experiment, we assume there is no concurrent transactions and run the two phases consecutively without waiting. We benchmark both : (1) gathering variable-length values and copying them into a contiguous buffer (Hybrid-Gather) and (2) using dictionary compression on variable-length values (Hybrid-Compress). We also implemented two baseline approaches for comparison purposes: (1) read a snapshot of the block in a transaction and copy into an Arrow buffer using the Arrow API (Snapshot) and (2) perform the entire transformation in-place in a transaction (In-Place). We use each algorithm to process 500 blocks (1 MB each) and vary the percentage of empty slots in each run.

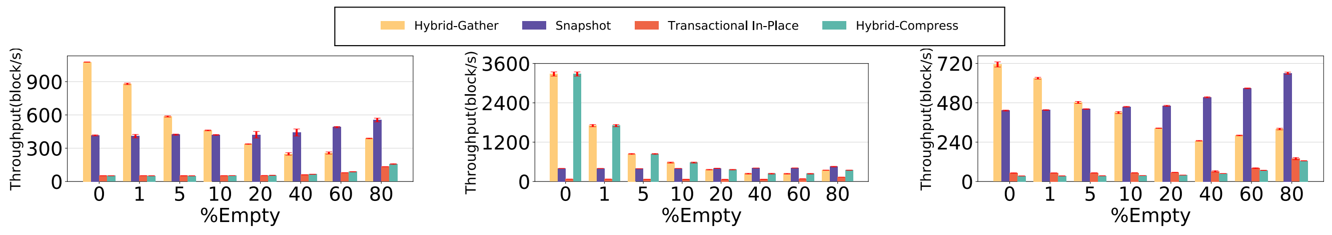
The results in Fig. 11a show that Hybrid-Gather outperforms the alternatives, achieving sub-millisecond performance when blocks are mostly full (empty $< 5\%$). Performance drops as *%empty* increases as the DBMS needs to move more tuples. Such movement is an order of magnitude more expensive than Snapshot due to the random memory access pattern. As the blocks become more than half empty, the number of tuples that the DBMS moves decreases. In-Place performs poorly because of the version maintenance overhead. Hybrid-Compress is also an order of magnitude slower than Hybrid-Gather and Snapshot because it is computationally expensive.

We provide a breakdown of each phase in Sec. 6.2. We present the graph in log-scale due to the large range of performance changes. When the number of empty slots in a block is low (i.e., <5%), the DBMS completes the compaction phase in microseconds because it is reduced to a bitmap scan. In this best-case scenario, the cost of variable-length gather dominates. The performance of the compaction phase drops as the number of empty slots increases and starts to dominate the cost of Hybrid-Gather at 5% empty. Dictionary compression is always the bottleneck in Hybrid-Compress.

We next measure how the impact of column types on transformation. We run the same micro-benchmark but make the database's columns either all fixed-length (Fig. 11b) or variable-length (Fig. 11c). These results show that the general performance trend does not change based on the data layouts. For the remaining experiments, we show only 50% variable-length columns results.

Write Amplification: The previous throughput results show that Snapshot outperforms our hybrid algorithm when blocks are ~20% empty. These measurements, however, fail to capture the overhead of updating the index entries for tuples that change their physical location in memory [55]. The effect of this write amplification depends on the indexes, but the cost for each tuple movement is constant. Therefore, it suffices to measure the total number of tuple movements that trigger index updates. The Snapshot algorithm always moves every tuple in the compacted blocks. We compare its performance against the compaction algorithms from Sec. 4.3.

As shown in Fig. 12, our algorithm is several orders of magnitudes more efficient than Snapshot in the best case, and twice as efficient when the blocks are half empty. The gap narrows as the number of empty slots per block increases. The approximate approach generates almost the same physical configuration for blocks as the optimal approach. Given this, and that the optimal algorithm



(a) Throughput (50% Variable-Length Columns) (b) Throughput (Fixed-Length Columns) (c) Throughput (Variable-Length Columns)

Figure 11: Transformation Throughput – Measurements of the DBMS's transformation algorithm throughput and movement cost when migrating blocks from the relaxed format to the canonical Arrow format.

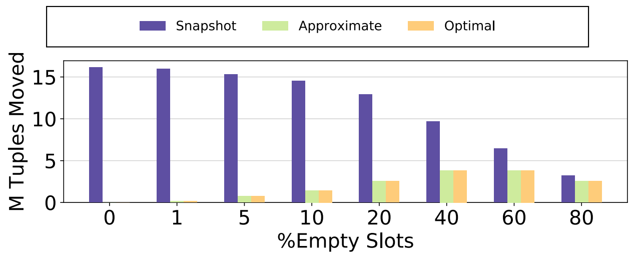


Figure 12: Write Amplification – Total write amplification is number of tuple movement times a constant for each table, determined by the layout and number of indexes on that table.

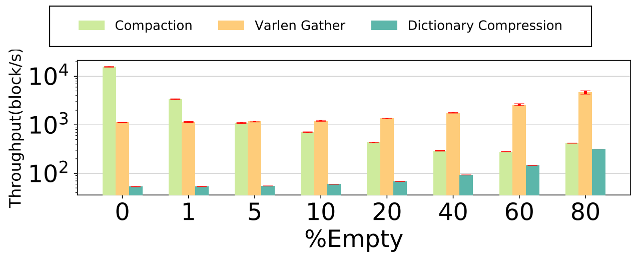
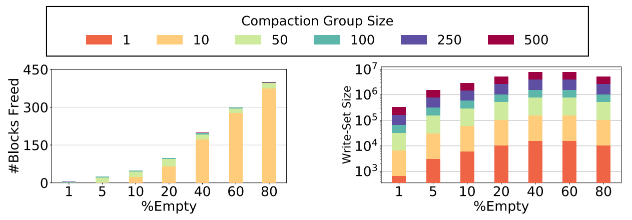


Figure 13: Performance Breakdown of Transformation – Measurement of throughput of different stages of the transformation algorithm, on 50% variable length columns workload.

requires one more scan across the blocks than the approximate one, we use the approximate algorithm for all other experiments.

Sensitivity on Compaction Group Size: We next evaluate the effect of the compaction group size on performance. The DBMS groups blocks together for compaction and then frees any empty blocks. This grouping enables the DBMS to reclaim memory from deleted slots. The size of each compaction group is a tunable parameter in the system. Larger group sizes result in the DBMS freeing more blocks but increase the size of the write-set for compacting transactions, which increases conflict and aborts. We use the same setup from the previous experiment, performing a single transformation pass through 500 blocks while varying group sizes.

Fig. 14a shows the number of freed blocks with different compaction group sizes. When blocks are only 1% empty, larger group sizes are required to release any memory. As the vacancy rate of blocks increases, smaller group sizes perform increasingly well, and larger values bring only marginal benefit. We show the cost of larger transactions as the size of their write-sets in Fig. 14b. These



(a) Number of Blocks Freed (b) Write-Set Size of Transactions

Figure 14: Sensitivity on Compaction Group Size – Transformation algorithm measurements when varying the number of blocks per compaction group while processing 500 blocks. (14a) shows the number of blocks freed during one round. (14b) shows the operations processed per second.

results indicate that larger group sizes increase transactions' write-set size, but yield a diminishing return on the number of blocks freed. The ideal fixed group size is between 10 and 50, which balances good memory reclamation and relatively small write-sets. To achieve the best possible performance, the DBMS should employ an intelligent policy to dynamically form groups based on its current requirements. We leave this problem for future work.

6.3 Data Export

We last evaluate the DBMS's ability to export data to an external tool. We compare four methods from Sec. 5 in NoisePage: (1) client-side RDMA, (2) Arrow Flight RPC, (3) vectorized wire protocol from [47], and (4) PostgreSQL wire protocol. We implement (3) and (4) in NoisePage according to their specifications. We run these experiments on two different servers with eight-core Intel Xeon D-1548 CPUs, 64 GB RAM, and a dual-port Mellanox ConnectX-3 10 GB NIC (PCIe v3.0, eight lanes).

We use the TPC-C ORDER_LINE table with 6000 blocks (~7 GB total size). On the client-side, we run a Python application and report the time taken between sending a request for data and the beginning of analysis execution. For each export method, we write a corresponding client-side protocol in C++, and use Arrow's cross-language API [21] to import it into the Python program. The client runs a TensorFlow program that passes data through a single linear unit, as the performance of this component is irrelevant to our system. We vary the percentage of blocks frozen in the DBMS to study the effect of concurrent transactions on export speed.

The results in Fig. 15 shows that NoisePage exports data orders-of-magnitude faster than the base-line implementations. When all blocks are frozen, RDMA saturates the available network bandwidth, and Arrow Flight can utilize up to 80% of the available network bandwidth. Putting this in the context of Fig. 1, NoisePage would

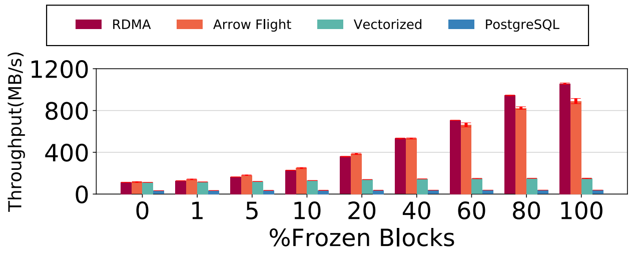


Figure 15: Data Export – Measurements of export speed with different export mechanisms in NoisePage, varying % of hot blocks.

complete the export task in about 15 seconds. When the system has to materialize every block, the performance of Arrow Flight drops to be equivalent to the vectorized wire protocol. RDMA performs slightly worse than Arrow Flight with a large number of hot blocks, because Flight has the materialized block in its CPU cache, whereas the NIC bypasses this cache when sending data. Both the PostgreSQL wire protocol and the vectorized protocol do not benefit from eliding transactions on cold, read-only data. This experiment indicates that the main bottleneck of the data export process in a DBMS is the serialization/deserialization step. Using Arrow as a drop-in replacement wire protocol in the current architecture does not achieve its full potential. Instead, storing data in a common format reduces this cost and boosts data export performance.

7 RELATED WORK

We now discuss three key facets of related work.

Universal Storage Formats: The idea of building systems on universal storage formats has been explored in other implementations. Systems such as Apache Hive [6], Apache Impala [7], Dremio [12], and OmniSci [17] support data ingestion from universal storage formats to lower the data transformation cost. Our DBMS, in contrast, natively generates data in the storage format as a data source for these systems. Of other artifacts, Apache ORC [9], a self-describing type-aware columnar file format designed for Hadoopis the most similar to our DBMS in its support for ACID transactions. Related to ORC is Databricks' Delta Lake engine [11] that acts as a ACID transactional engine on top of cloud storage. These solutions are intended for incremental maintenance of read-only data sets and not high-throughput OLTP, targeting infrequent, non-performance-critical transactions with large write-sets. Apache Kudu [8] is an analytical system that is similar in architecture to our system and integrates natively with the Hadoop ecosystem. However, transactional semantics in Kudu is restricted to single-table updates or multi-table scans [20].

OLTP on Column-Stores: The database community has implemented several OLTP-capable systems on column-stores. PAX [26] stores data in columnar format, but keeps all attributes of a single tuple within a disk page to reduce I/O cost. HYRISE [37] improved upon this scheme by vertically partitioning each table based on access patterns. SAP HANA [50] implemented migration from row-store to column-store in addition to partitioning. MemSQL's SingleStore [14] improved their transactional performance on columnar data by adding hash indexes, sub-segment access, and fine-grain locking. Peloton [28] introduced the logical tile abstraction to enable migration without a need for disparate execution engines. Our system is most similar to HyPer [36, 39, 41, 46] and L-Store [49]. HyPer runs exclusively on columnar format and guarantees ACID properties through a multi-versioned delta-based concurrency control mechanism similar to our system; it also compresses cold data chunks by instrumenting the OS for access observation. Our system is different from HyPer in that it is built around the open-source Arrow format and provides native access to it. HyPer's hot-cold transformation also assumes heavy-weight compression operations, whereas our transformation process is designed to be fast and computationally inexpensive, allowing more fluid changes in a block's state. L-Store also leverages the hot-cold separation of tuple access to allow updates to be written to *tail-pages* instead of more expensive cold storage. In contrast to our system, L-Store achieves this with append-only storage and data lineage tracing.

Optimized DBMS Networking: [47] demonstrated that transferring large amounts of data from the DBMS to a client is expensive over existing wire row-oriented protocols (e.g., JDBC/ODBC) and showed how to increase transmission performance by vectorizing the result set. A similar technique was proposed in the olap4j extension for JDBC in the early 2000s [1]. These works, however, optimize the DBMS's network layer, whereas this paper tackles the challenge more broadly through co-designing DBMS storage with export protocols. In addition, there has been considerable work on using RDMA to speed up DBMS workloads. IBM's DB2 pureScale [13] and Oracle Real Application Cluster (RAC) [18] use RDMA to exchange database pages and achieve shared-storage between nodes. Microsoft Analytics Platform Systems [15] and Microsoft SQL Server with SMB Direct [16] utilize RDMA to bring data from a separate storage layer to the execution layer. All of these work attempts to improve the performance of distributed DBMS, whereas our paper improves efficiency across the data processing pipeline through better interoperability with external tools.

8 CONCLUSION

We presented NoisePage's Arrow-native storage architecture for in-memory OLTP workloads. The system implements a multi-versioned, delta-store transactional engine capable of directly emitting Arrow data to external analytical tools. To ensure OLTP performance, the system allows transactions to work with a relaxed Arrow format and employs a lightweight in-memory transformation process to convert cold data into full Arrow in milliseconds. This allows the DBMS to support bulk data export to external analytical tools at zero serialization overhead. We evaluated our implementation, and show good OLTP performance while achieving orders-of-magnitude faster data export compared to current approaches.

ACKNOWLEDGMENTS

This work was supported (in part) by the National Science Foundation (IIS-1846158, IIS-1718582, SPX-1822933), Google Research Grants, the Alfred P. Sloan Research Fellowship program, and the Siebel Scholars program. We also thank David Andersen, Yu Xiang Zhu, and Tan Li of Carnegie Mellon University for their feedback. TKBM.

REFERENCES

[1] 2013. olap4j: Open Java API for OLAP. http://www.olap4j.org.

[2] 2016. Guaranteed Atomic Operations on Intel Processors. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.

[3] 2018. Diving into the HANA DataFrame: Python Integration. https://blogs.sap.com/2018/12/17/diving-into-the-hana-dataframe-python-integration-part-1/.

[4] 2019. Apache Arrow. https://arrow.apache.org/.

[5] 2019. Apache Arrow Source Code. https://github.com/apache/arrow.

[6] 2019. Apache Hive. https://hive.apache.org/.

[7] 2019. Apache Impala. https://hive.apache.org/.

[8] 2019. Apache Kudu. https://kudu.apache.org/overview.html.

[9] 2019. Apache ORC. https://orc.apache.org/.

[10] 2019. Apache Parquet. https://parquet.apache.org/.

[11] 2019. Databricks Delta Lake. https://databricks.com/blog/2019/04/24/open-sourcing-delta-lake.html.

[12] 2019. Dremio. https://docs.dremio.com/.

[13] 2019. IBM DB2 Pure Scale. https://www.ibm.com/support/knowledgecenter/en/SSEPGG$_1$0.5.0/com.ibm.db2.luw.licensing.doc/doc/c0057442.html.

[14] 2019. MemSQL SingleStore. https://www.memsql.com/blog/memsql-singlestore-then-there-was-one/.

[15] 2019. Microsoft Analytics Platform System. https://www.microsoft.com/en-us/sql-server/analytics-platform-system.

[16] 2019. Microsoft SQL Server with SMB Direct. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/jj134210(v=ws.11).

[17] 2019. OmniSci GPU-Accelerated Analytics. https://www.omnisci.com/.

[18] 2019. Oracle Real Application Cluster. https://www.oracle.com/technetwork/server-storage/networking/documentation/o12-020-1653901.pdf.

[19] 2019. TensorFlow I/O Apache Arrow Datasets. https://github.com/tensorflow/io/tree/master/tensorflow$_i$o/arrow.

[20] 2019. Transaction Semantics in Apache Kudu. https://kudu.apache.org/docs/transaction$_s$emantics.html.

[21] 2019. Using PyArrow from C++ and Cython Code. https://arrow.apache.org/docs/python/extending.html.

[22] 2020. Add a "Flight SQL" extension on top of FlightRPC. https://lists.apache.org/thread.html/rc4717b78f09bbf7a69347b6c126849e17323c491338fc73457cf7558%40%3Cdev.arrow.apache.org%3E.

[23] 2020. NoisePage. https://noise.page.

[24] 2020. Oracle TimesTen In-Memory Database. https://www.oracle.com/database/technologies/related/timesten.html.

[25] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. Row-stores: How Different Are They Really?. In *SIGMOD*. 967–980.

[26] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. 2002. Data Page Layouts for Relational Databases on Deep Memory Hierarchies. *The VLDB Journal* 11, 3 (Nov. 2002), 198–215.

[27] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: A Hands-free Adaptive Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. 1103–1114.

[28] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 583–598.

[29] Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2016. Designing Access Methods: The RUM Conjecture. In *EDBT*. 461–466.

[30] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control – Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983).

[31] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*.

[32] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. 275–290.

[33] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* 6 (September 2013), 1942–1953. Issue 14.

[34] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. 1–8.

[35] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288.

[36] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *Proc. VLDB Endow.* 5, 11 (July 2012), 1424–1435.

[37] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.* 4, 2 (Nov. 2010), 105–116.

[38] Allison L. Holloway and David J. DeWitt. 2008. Read-optimized Databases, in Depth. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 502–513.

[39] 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.

[40] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11 (September 2018), 2209–2222. Issue 13.

[41] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. 311–326.

[42] Per-Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance Concurrency Control Mechanisms for Main-memory Databases. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 298–309.

[43] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. 1307–1318.

[44] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.* 11, 1 (September 2017), 1–13.

[45] 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 (March 1992), 94–162.

[46] 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.

[47] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage: A Case for Client Protocol Redesign. *Proc. VLDB Endow.* 10, 10 (June 2017), 1022–1033.

[48] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. 2016. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 1539–1551. https://doi.org/10.1145/2882903.2915966

[49] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-time OLTP and OLAP System. In *Extending Database Technology*. 540–551.

[50] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD*. 731–742.

[51] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an Architectural Era: (It's Time for a Complete Rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. 1150–1160.

[52] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0). http://www.tpc.org/tpcc/spec/tpcc$_c$urrent.pdf.

[53] 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* (Farminton, Pennsylvania) *(SOSP '13)*. 18–32.

[54] 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 ACM International Conference on Management of Data (SIGMOD '18)*. 473–488.

[55] 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.

[56] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220.