Supporting Hybrid Workloads for In-Memory Database Management Systems via a Universal Columnar Storage Format

Tianyu Li

CMU-CS-19-112

May 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andrew Pavlo, Chair
David G. Andersen

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This work was supported (in part) by the National Science Foundation (IIS-1846158, SPX-1822933) and an Alfred P. Sloan Research Fellowship.

Keywords: Database Systems, Apache Arrow

Abstract

The proliferation of modern data processing ecosystems has given rise to open-source columnar data formats. The key advantage of these formats is that they allow organizations to load data from database management systems (DBMSs) once instead of having to convert it to a new format for each usage. These formats, however, are read-only. This means that organizations must still use a heavy-weight transformation process to load data from their original format into the desired columnar format. We aim to reduce or even eliminate this process by developing an in-memory storage management architecture for transactional DBMSs that is aware of the eventual usage of its data and operates directly on columnar storage blocks. We introduce relaxations to common analytical format requirements to efficiently update data, and rely on a lightweight in-memory transformation process to convert blocks back to analytical forms when they are cold. We also describe how to directly access data from third-party analytical tools with minimal serialization overhead. To evaluate our work, we implemented our storage engine based on the Apache Arrow format and integrated it into the CMDB DBMS. Our experiments show that our approach achieves comparable performance with dedicated OLTP DBMSs while also enabling orders of magnitude faster data exports to external data science and machine learning libraries than existing approaches.

Acknowledgments

I would like to thank my advisor, Professor Andy Pavlo for help and guidance in this work. He introduced me to research and taught me many lessons, both in databases and other things that do not belong in an official document. I would also like to thank other students and staff in the CMU Database Group that helped in this project: Matt Butrovich, Wan Shen Lim, Amadou Ngom and Pervaze Akhtar. Thanks to Yuxiang Zhu and Tan Li of CMU for their help in setting up experiments, and to Professor Dave Andersen for his guidance and feedback in the network portion of this work. Special thanks to Wes McKinney of Ursa Labs for his input in the formulation of this work. I would also like the thank other professors and staff of Carnegie Mellon's School of Computer Science for providing me with the awesome platform to learn and do research. Last but not least, thanks to Yifei Li for her support and understanding during this project.

Contents

1 Introduction **1**

1.1 Motivation for This Research 1

1.2 Overview of This Thesis 2

2 Background **5**

2.1 Data Movement and Transformation 5

2.2 Column-Stores and Apache Arrow 7

2.3 Hybrid Storage Considered Harmful 9

3 Related Work **11**

3.1 Universal Storage Formats 11

3.2 OLTP on Column-Stores 12

3.3 Optimized DBMS Networking 13

4 System Overview **15**

4.1 Transactions on a Column Store 15

4.2 Blocks and Physiological Identifiers 17

4.3 Garbage Collection 19

4.4 Logging and Recovery 21

5 Block Transformation **25**

5.1 Relaxed Columnar Format 26

5.2 Identifying a Cold Block . . . 27
5.3 Transformation Algorithm . . . 29
5.3.1 Phase #1: Compaction . . . 29
5.3.2 Phase #2: Gathering . . . 32
5.4 Additional Considerations . . . 34
5.4.1 Dictionary Encoding . . . 34
5.4.2 Memory Management . . . 35

6 Data Export **37**
6.1 Improved Wire Protocol for SQL . . . 37
6.2 Alternative Query Interface . . . 38
6.3 Client-Side RDMA . . . 39
6.4 Server-Side RDMA . . . 40
6.5 External Tool Execution on the DBMS . . . 40

7 Evaluation **43**
7.1 OLTP Performance . . . 43
7.2 Transformation to Arrow . . . 46
7.2.1 Throughput . . . 46
7.2.2 Write Amplification . . . 50
7.2.3 Sensitivity on Compaction Group Size . . . 50
7.3 Data Export . . . 51

8 Conclusion and Future Work **55**

Bibliography **57**

List of Figures

2.1 **Data Transformation Costs** – Time taken to load a TPC-H table into Pandas with different approaches. 7

2.2 **SQL Table to Arrow** – An example of using Arrow's API to describe a SQL table's schema in a high-level language like Python. 7

2.3 **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. 9

3.1 **Vectorized PostgreSQL Wire Protocol** – Instead of transmitting row-at-a-time, a vectorized protocol would transmit column batches. 14

4.1 **System Architecture** – CMDB's transactional engine is minimally intrusive to the underlying storage to maintain compatibility with the Arrow storage format. 18

4.2 **TupleSlot** – By aligning blocks to start at 1 MB boundaries, the DBMS packs the pointer to the block and the offset in a single 64-bit word. . . . 20

5.1 **Variable-Length Value Storage** – The system uses 16 bytes to track variable-length values as a fixed-size column in a block. 25

5.2 **Relaxed Columnar Format** – The system briefly allows non-contiguous memory to support efficient mutation of Arrow blocks. 27

5.3 **Transformation to Arrow** – CMDB implements a pipeline for lightweight in-memory transformation of cold data to Arrow. 31

5.4 **Check-and-Miss on Block Status** – A naïve implementation results in a race within the critical section of the gathering phase. 34

6.1 **Client-Side RDMA** – An illustration of the message flow between DBMS and client if the DBMS implements client-side RDMA 39

6.2 **Server-Side RDMA** – An illustration of the message flow between DBMS and client if the DBMS implements server-side RDMA. As shown, the message flow involved is much more complex than client-side RDMA. . 41

7.1 **OLTP Performance: Throughput** – Throughput measurements of the DBMS for the TPC-C workload when varying the number of worker threads. 44

7.2 **OLTP Performance: Block State Coverage** – Block state coverage of the DBMS at the end of a TPC-C run when varying the number of worker threads. 45

7.3 **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. 47

7.4 **Transformation Throughput on Alternative Layouts** – Measurements of the DBMS's transformation algorithm throughput and when varying the layout of the blocks being transformed. 49

7.5 **Write Amplification** – Total write amplification is number of tuple movement times a constant for each table, decided by the layout and number of indexes on that table. 51

7.6 **Sensitivity on Compaction Group Size** – Efficacy measurements of the transformation algorithm when varying the number of blocks per compaction group while processing 500 blocks. The percentage of empty slots is what portion of each block is empty (i.e., does not contain a tuple). We measure the number of blocks freed during one round. 52

7.7 **Sensitivity on Compaction Group Size** – Efficacy measurements of the transformation algorithm when varying the number of blocks per compaction group while processing 500 blocks. The percentage of empty slots is what portion of each block is empty (i.e., does not contain a tuple). We measure the average write-set size of transactions in the transformation algorithm. 53

7.8 **Data Export** – Measurements of data export speed of CMDB using different export mechanisms, with varying percentage of blocks treated as hot. 54

List of Tables

Chapter 1

Introduction

1.1 Motivation for This Research

Data analysis pipelines allow organizations to extrapolate new insights from data residing in their on-line transactional processing (OLTP) systems. Tools that make up these pipelines often use standard binary formats that grew out of the open-source community, such as Apache Parquet [par], Apache ORC [apa [c]] and Apache Arrow [apa [a]]. These formats are popular because they allow disparate systems to exchange data through a common interface without repeated conversion between proprietary formats. They are designed to be read-only, however, which means that a data scientists needs to use a heavy-weight process to export data from the OLTP DBMS into the desired format. This process wastes computing power and limits both the immediacy and frequency of analytics.

Enabling analysis of data as soon as it arrives in a database is an important goal in DBMS implementation. Over the past decade, several companies and research groups have developed hybrid transactional analytical processing (HTAP) DBMSs in attempts to address this issue [Pezzini et al. [2014]]. These systems, however, are not one-size-fits-all solutions to the problem. Modern data science workloads often involve specialized frameworks, such as TensorFlow, PyTorch, and Pandas. Legacy DBMSs cannot hope to outperform these tools for workloads such as machine learning. Additionally, the data science community has

increasingly standardized around Python as a platform for its tools. As such, organizations are heavily invested in personnel, tooling, and infrastructure for the Python ecosystem. It is unlikely HTAP DBMSs will overcome these barriers and replace external tools. We contend that these tools will co-exist with relational databases for the foreseeable future. To deliver performance gains across the entire data analysis pipeline, our focus should be to improve a DBMS's interoperability with external tools.

To address this challenge, we present a multi-versioned DBMS architecture that supports OLTP workloads directly on an open-source columnar storage format used by external analytical frameworks. We target Apache Arrow, although our approach is applicable to any columnar format. In this paper, we describe how to build a transaction engine with in-memory MVCC delta version chains [Neumann et al. [2015], Wu et al. [2017]] on Arrow without intrusive changes to the format. We relax Arrow specification constraints to achieve good OLTP performance, and propose a lightweight transformation process to convert cold data back into the canonical Arrow format. We show that our system facilitates fast exports to external tools by providing direct access to data through a bulk-export RPC layer with no serialization overhead.

1.2 Overview of This Thesis

We implemented our storage and concurrency control architecture in **CMDB** [cmu] and evaluated its OLTP performance. Our results show that we achieve good performance on OLTP workloads operating on the Arrow format. We also implemented new data export protocols assuming Arrow storage, and demonstrate that we are able to reduce data serialization overhead compared to existing DBMS wire protocols.

The remainder of this paper is organized as follows. We first present in 2 the motivation for this work and introduce the Arrow storage format. We then discuss the system's overall architecture in 4, followed by a detailed discussion of the transformation process and how we modify existing system components to detect cold blocks and perform the transformation

with little overhead in 5. The mechanism for data export to analytics is discussed in 6. We present our experimental evaluation in 7 and discuss related work in 3.

Chapter 2

Background

We now discuss challenges in using data stored in a transactional DBMS for external data analysis. We begin by describing how transformation and movement of data to analytics tools are bottlenecks in modern data processing. We then present a popular open-source format used by these tools, Apache Arrow, and show that the requirements of analytical data formats are not incompatible with good OLTP performance. Lastly, we argue that previous hybrid storage approaches are not optimal in this setting, and that storing data in a format close to its eventual use is more efficient.

2.1 Data Movement and Transformation

A data processing pipeline consists of (1) a front-end OLTP layer, and (2) multiple analytical layers. OLTP engines employ the n-ary storage model (i.e., row store) to support efficient operations on single tuples. In contrast, the analytical layers use the decomposition storage model (i.e., column store) to speed up large scans [Abadi et al. [2008], Boncz et al. [2005], Menon et al. [2017], Kersten et al. [2018]]. Because of disparate optimization strategies for these two use cases, organizations often implement the pipeline using a combination of specialized tools and systems.

The most salient issue with this bifurcated approach is data transformation between layers. The complexity of modern analytical pipelines have increased with the introduction of new machine learning (ML) workloads. ML tools are data-intensive and not interoperable with the OLTP system's format, making loading and preparation of the data from a DBMS expensive. For example, a data scientist will (1) execute SQL queries to bulk-export the 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, Pandas Dataframes for Pandas, and finally tensors for TensorFlow. The slowest by far is from the DBMS to Pandas. The data scientist first retrieves the data over the DBMS's network protocol, and then decomposes the data into the desired columnar format. This process is not optimal for high-bandwidth data movement [Raasveldt and Mühleisen [2017]].

To better understand this issue, we measured the time it takes to extract data from PostgreSQL (v10.6) and load it into an external Pandas program. We use the `LINEITEM` table from TPC-H with scale factor 10 (60M tuples, 8 GB as a CSV file, 11 GB as a PostgreSQL table). We compare three approaches for loading the table into the Python program: (1) SQL over a Python ODBC connection, (2) using PostgreSQL's `COPY` command to export a CSV file to disk and then loading it into Pandas, and (3) 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 132 GB of memory, of which 15 GB are reserved for PostgreSQL's shared buffers so that there is more than enough space to store a copy of the table both in the DBMS's buffer pool and in the Python script. We provide a full description of our operating environment for this experiment in 7.

The results in 2.1 show that Python ODBC and CSV are orders of magnitude slower than localized access. This is because of the overhead of transforming to a different format, as well as excessive serialization in the PostgreSQL wire protocol, where query processing

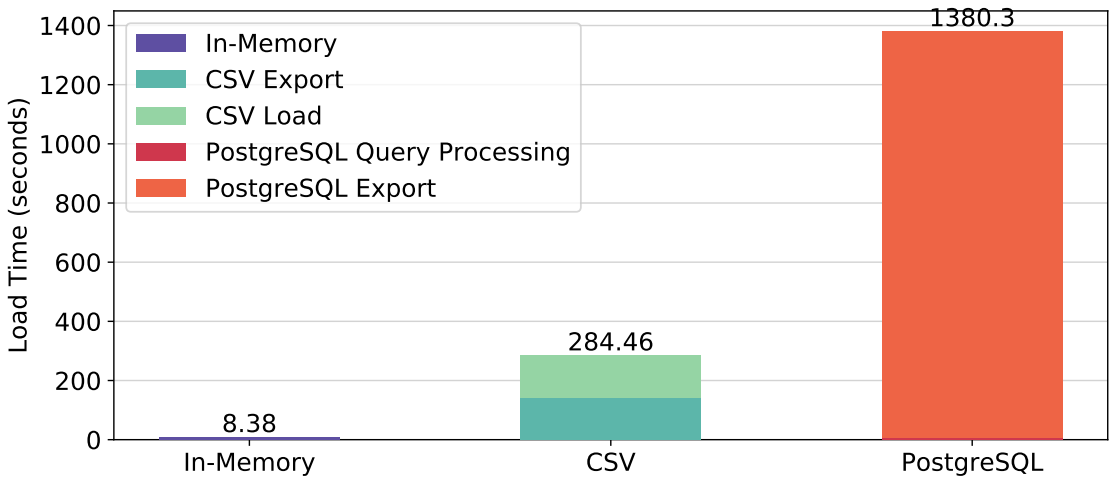


Figure 2.1: Data Transformation Costs – Time taken to load a TPC-H table into Pandas with different approaches.

```
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 2.2: SQL Table to Arrow – An example of using Arrow's API to describe a SQL table's schema in a high-level language like Python.

itself takes 0.004% of the total export time. The rest of time is spent in the serialization layer and in transforming the data. Optimizing this data export process can greatly speed up the entire analytics pipeline.

2.2 Column-Stores and Apache Arrow

The inefficiency of loading data through a SQL interface requires us to rethink the data export process. If the OLTP DBMS and these external tools operate on the same format,

the data export cost is reduced to just the cost of network transmission. But as discussed previously, OLTP DBMSs are row-stores because the conventional wisdom is that column-stores are inferior for OLTP workloads. Recent work [Neumann et al. [2015], Sikka et al. [2012]], however, has shown that column-stores can support high-performance transactional processing. We therefore propose to implement a high-performance OLTP DBMS directly on top of a format used by analytics tools. To do so, we identify a representative format, Apache Arrow, and analyze its strengths and weaknesses on OLTP workloads.

Apache Arrow is a cross-language development platform for in-memory data. It was conceived when groups of developers from Apache Drill, Apache Impala, Apache Kudu, Pandas, and others independently explored universal in-memory columnar data formats. These groups then joined together in 2015 to develop a shared format based on their overlapping requirements. Arrow was introduced in 2016 and has since become the standard for columnar in-memory analytics as a high-performance interface between heterogeneous systems. There is a growing ecosystem of tools built for Arrow, including bindings for several programming languages, computational libraries, and IPC frameworks.

At the core of Arrow is a columnar memory format for flat and hierarchical data. This format enables (1) fast analytical data processing by optimizing for data locality and vectorized execution and (2) zero-deserialization data interchange between disparate software systems. To achieve the former, Arrow organizes data contiguously in 8-byte aligned buffers and uses separate bitmaps to track nulls. For the latter, Arrow specifies a standard in-memory representation and provides a C-like data definition language (DDL) for communicating metadata about a data schema. Arrow uses separate metadata data structures to impose table-like structure on collections of buffers. The language is expressive enough to describe data layouts of existing systems. An example of an Arrow equivalent definition for the `ITEM` table in TPC-C is shown in 2.2.

Although Arrow is designed for read-only analytical applications, one can use it as an in-memory data structure for other workloads. Arrow's alignment requirement and use of null bitmaps benefit write-heavy workloads as well as data sets with fixed-length values. Problems emerge, however, in Arrow's support for variable-length values. Arrow stores variable-length values, such as `VARCHAR`s, as an array of offsets indexing into a contiguous

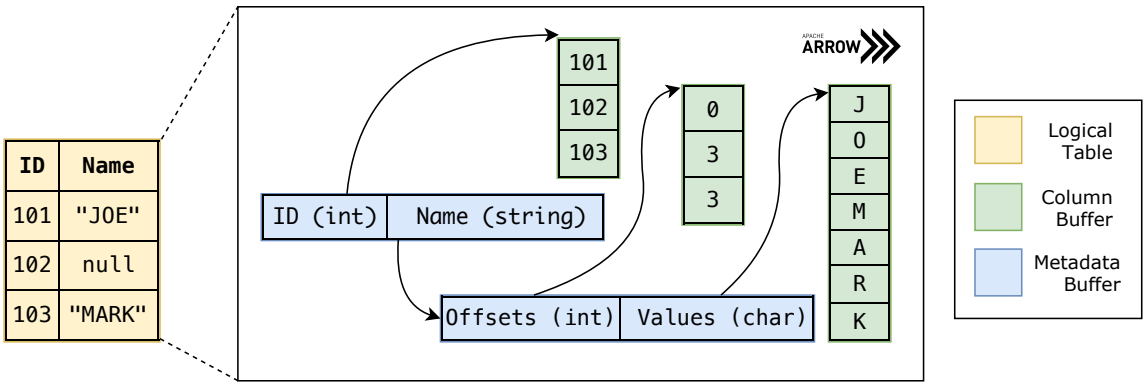


Figure 2.3: Variable Length Values in Arrow – Arrow represents variable length values as an offsets array into an array of bytes, which trades off efficient mutability for read performance.

byte buffer. As shown in 2.3, the length of the individual entry is computed as the difference between the starting offset of itself and the next entry, or the size of the character array if the entry is the last entry. This approach is not ideal for updates because of write amplifications. Suppose a program needs to update "JOE" from the example to "ANDY" in 2.3, it must copy the entire `values` buffer for the buffer to stay contiguous.

The reason behind this is that it is difficult in a single data format to simultaneously provide (1) data locality and value adjacency, (2) constant-time random access, and (3) mutability. Arrow trades off (3) in favor of the first two for read-only workloads. As we discuss in 5, however, it is possible to efficiently convert between designs that optimize for (1) and (3). Our system can leverage this to achieve (1) for read-heavy blocks and (3) for update-heavy blocks.

2.3 Hybrid Storage Considered Harmful

Researchers have proposed hybrid storage schemes to unify row-store and column-store within one system. Two notable examples are Peloton [Arulraj et al. [2016]] and H2O [Alagiannis et al. [2014]]. Peloton uses an abstraction layer above the storage engine that transforms cold row-oriented data to columnar format. In contrast, H2O proposes an abstraction layer at the physical operator level that materializes data into the optimal format

and generates code for it on a per-query basis. We contend here that neither approach is ideal and that the columnar format is good enough for most use cases.

There is a trade-off between flexibility and performance in such hybrid storage systems. Our previous system, Peloton, attempted to optimize for performance by installing the abstraction boundary at the storage level. This increases the complexity of the implementation. We had to develop mechanisms to explicitly deal with hybrid storage in the concurrency control and garbage collection (GC) components of the system, and think carefully about edge cases where a transaction spans both formats. The performance gain from a hybrid storage was lost in these other components. For example, the benefit of Peloton's row-store on updates is negated by its use of append-only multi-versioned storage to overcome the complexity of GC for its hybrid storage. H2O, on the other hand, maintains a higher-level operator abstraction and generates code dynamically for each operator to handle different layouts. This approach reduces complexity for query processing, but the overhead of dynamic layout adaptation and operator creation is large and must be amortized over large scans, making it unsuitable for OLTP workloads.

The benefit of using hybrid storage is also limited. Peloton organizes tuples in rows to speed up transactions on hot data. For an in-memory DBMS like Peloton, however, contention on tuples, and the serial write-ahead-log dominates transactional throughput compared to data copying costs. In contrast, H2O uses its hybrid storage to speed up analytical queries exclusively, as the authors observe that certain analytical queries perform better under row-store instead of column-store. Due to its lack of support for transactional workloads, however, H2O needs to load data from a separate OLTP system. As we have discussed, this movement is expensive and can take away the performance gain from using hybrid storage.

Overall, hybrid storage systems introduce more complexity to the DBMS and provide only minor performance improvements. They also do not speed up external analytics. As these external tools become more prominent in data processing pipelines, the benefits of a hybrid storage DBMS diminishes. Hence, we argue that the these gains do not justify the engineering cost. A well-written conventional system is good enough as the bottleneck shifts to data movement between layers of the data analytics pipeline.

Chapter 3

Related Work

We presented our system for high transaction throughput on a storage format optimized for analytics, and now discuss three key facets of related work. In particular, we provide an overview of other universal storage formats, additional systems that implemented OLTP workloads on column-stores, and efforts to accelerate data export by optimizing the network layer of DBMSs.

3.1 Universal Storage Formats

The idea of building a data processing system on top of universal storage formats has been explored in other implementations. Systems such as Apache Hive [hiv], Apache Impala [imp], Dremio [dre], and OmniSci [omn] support data ingestion from universal storage formats to lower the data transformation cost. These are analytical systems that ingest data already generated in the format from an OLTP system, whereas our DBMS natively generates data in the storage format as a data source for these systems.

Among the storage formats other than Arrow, Apache ORC [apa [c]] is the most similar to our DBMS in its support for ACID transactions. ORC is a self-describing type-aware columnar file format designed for Hadoop. It divides data into *stripes* that are similar to our concept of blocks. Related to ORC is Databricks' Delta Lake engine [del] that acts as a

ACID transactional engine on top of cloud storage. These solutions are different from our system because they are intended for incremental maintenance of read-only data sets and not high-throughput OLTP. Transactions in these systems are infrequent, not performance critical, and have large write-sets. Apache Kudu [kud [a]] is an analytical system that is similar in architecture to our system, and integrates natively with the Hadoop ecosystem. Transactional semantics in Kudu is restricted to single-table updates or multi-table scans and does not support general-purpose SQL transactions [kud [b]].

3.2 OLTP on Column-Stores

Since Ailamaki et al. first introduced the PAX model [Ailamaki et al. [2002]], the community has implemented several systems that supports transactional workloads on column stores. PAX stores data in columnar format, but keeps all attributes of a single tuple within a disk page to reduce I/O cost for single tuple accesses. HYRISE [Grund et al. [2010]] improved upon this scheme by vertically partitioning each table based on access patterns. SAP HANA [Sikka et al. [2012]] implemented migration from row-store to column-store in addition to partitioning. Peloton [Arulraj et al. [2016]] introduced the logical tile abstraction to be able to achieve this migration without a need for disparate execution engines. Our system is most similar to HyPer [Kemper and Neumann [2011], Funke et al. [2012], Neumann et al. [2015]] and L-Store [Sadoghi et al. [2018]]. 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 implements a compression for 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 through tracing data lineage and an append-only storage within the table itself.

3.3 Optimized DBMS Networking

There has been considerable work on using RDMA to speed up DBMS workloads. IBM's DB2 pureScale [pur] and Oracle Real Application Cluster (RAC) [rac] use RDMA to exchange database pages and achieve shared-storage between nodes. Microsoft Analytics Platform Systems [msa] and Microsoft SQL Server with SMB Direct [smb] utilize RDMA to bring data from a separate storage layer to the execution layer. Binnig et al. [Binnig et al. [2016]] and Dragojević et al. [Dragojević et al. [2015]] proposed using RDMA for distributed transaction processing. Li et al. [Li et al. [2016]] proposed a method for using RDMA to speed up analytics with remote memory. All of these work attempts to improve the performance of distributed DBMS through using RDMA within the cluster. This thesis looks to improve efficiency across the data processing pipeline through better interoperability with external tools.

Raasveldt and Mühleisen [Raasveldt and Mühleisen [2017]] 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). They then explored how to improve server-side result set serialization to increase transmission performance. Specifically, they proposed to use a vector-based protocol, transmitting column batches at a time, instead of only rows. An example of this technique applied to the PostgreSQL wire protocol is shown in 3.1. A similar technique was proposed in the olap4j extension for JDBC in the early 2000s [ola]. These works, however, optimize the DBMS's network layer, whereas this thesis tackles the challenge more broadly through changes in both the network layer and the underlying DBMS storage.

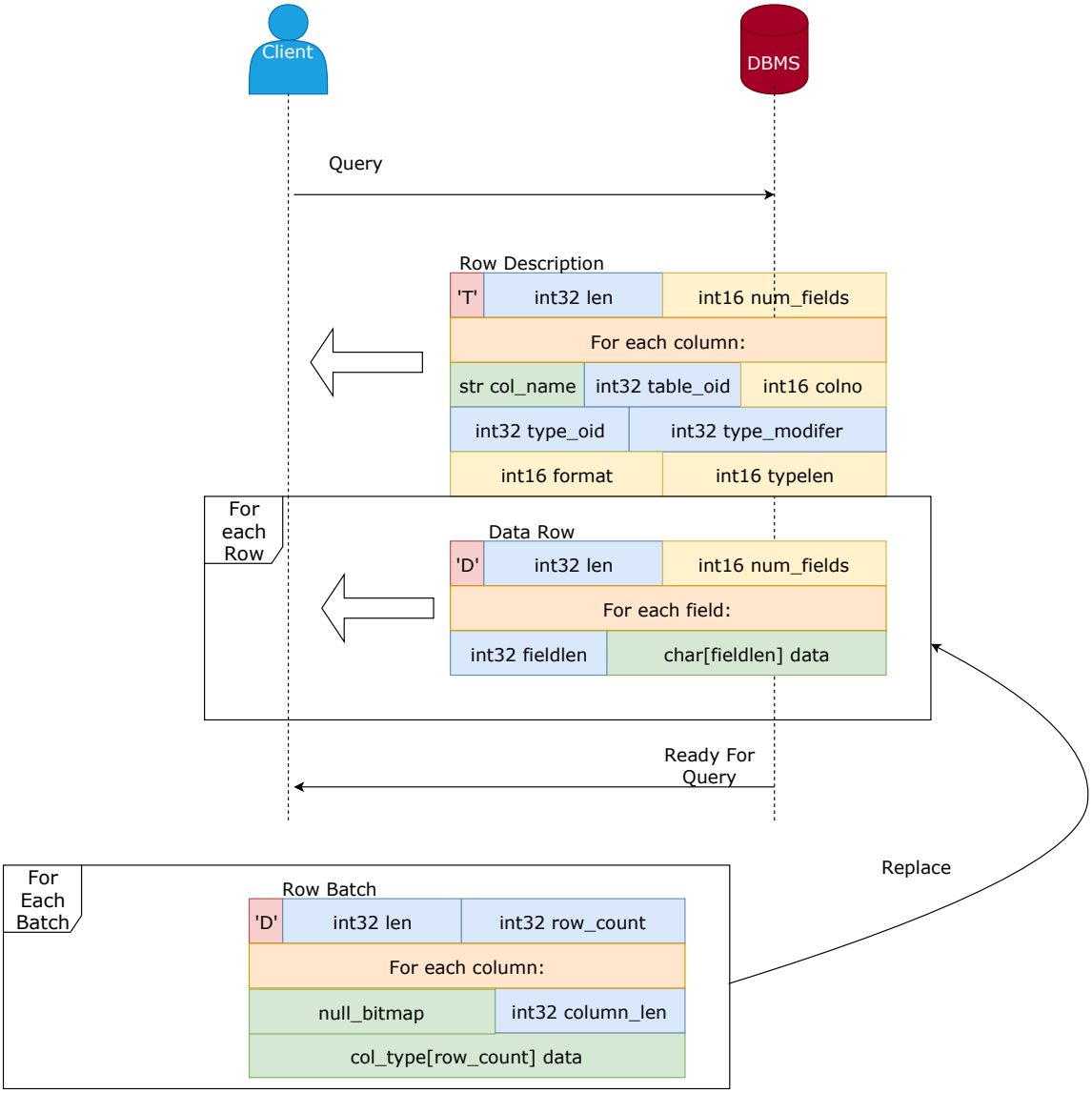


Figure 3.1: Vectorized PostgreSQL Wire Protocol – Instead of transmitting row-at-a-time, a vectorized protocol would transmit column batches.

Chapter 4

System Overview

We now present the system architecture for CMDB that runs natively on Arrow. We first discuss how we designed our *transaction engine* to be minimally intrusive to Arrow's byte-level layout. We then describe how the system organizes tables into blocks and uses an efficient scheme to uniquely identify tuples. Finally, we describe the system's garbage collection and recovery components, and provide insights that aid the transformation algorithm in 5. An overview of our system architecture is shown in 4.1. To simplify our discussion, we assume that all data is fixed length, and describe how to handle variable-length data in the next section.

4.1 Transactions on a Column Store

The key requirement for our system is that it stores transactional metadata and version information separately from the actual data; external tools are oblivious to transactions and should not see versioning information when scanning through Arrow blocks. To meet this requirement, the DBMS's transaction engine uses a multi-versioned [Bernstein and Goodman [1983]] delta-storage where the version chain is stored as an extra Arrow column invisible to external readers. The system stores physical pointers to the head of the version chain in the column, or null if there is no version. The version chain is newest-to-oldest

with delta records, which are physical before-images of the modified tuple attributes. Rather than storing these deltas in the Arrow blocks, the system assigns each transaction an undo buffer as an append-only row-store for deltas. To install an update, the transaction first reserves space for a delta record at the end of its buffer, copies the current image of the attributes to modify into the record, appends the record onto the version chain, and finally updates in-place. Deletes and inserts are handled analogously, but their undo records pertain to an allocation bitmap rather than the content of the tuple.

The undo buffer needs to grow in size dynamically as transactions can have arbitrarily large write sets. Because the version chain points directly into the undo buffer, however, the delta record cannot be moved during the lifetime of the transaction's version. This rules out the use of a naïve resizing algorithms that doubles the size of the buffer and copies the content. Instead, the system implements undo buffers as a linked list of fixed-sized buffer segments (4096 bytes in our system). When there is not enough space left in the buffer for a delta record, the system grows the undo buffer by one segment. A centralized object pool distributes buffer segments to transactions, and is responsible for preallocation and recycling.

The system assigns each transaction two timestamps, (*start*, *commit*), generated from the same counter. The *commit* timestamp is the *start* timestamp with its sign bit flipped if 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* timestamp. Because the system uses unsigned comparison for timestamps, uncommitted versions are never visible. The system disallows write-write conflicts to avoid cascading rollbacks.

On aborts, the system uses the transaction's undo records to roll back the in-place updates and unlinks the records from the version chain. This introduces a race where a concurrent reader copies the aborted version, the aborting transaction performs the rollback, and the reader traverses the version chain with the undo record already unlinked. To handle this, the reader checks that the version pointer does not change while it is copying the in-place version, and retries otherwise. 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. One can replace this critical section with an additional validation phase for full serializability [Neumann et al. [2015]]. Our system instead aims for snapshot isolation, as the isolation level of transactions has little impact on our mechanisms for interoperability with Arrow.

Through this design, the transaction engine reasons only about delta records and the version column, and not the underlying physical storage. Maintaining the Arrow abstraction comes at the cost of data locality and forces readers to materialize early, which degrades range-scan performance. Fortunately, due to the effectiveness of our garbage collector (4.3), 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. Blocks are natural units for keeping track of this information, and we use block-level locks to coordinate access to blocks that are not versioned and not frequently updated (i.e., cold). This is discussed in detail in 5.

We do not keep versioned indexes but instead model udpates as inserts and deletes into the index, which stores tuple slots as values. We clean up any stale entries in the index at the end of each transaction. Updates that change an indexed attribute on a tuple are treated as a delete and an insert, similar to Neumann et al. [2015].

4.2 Blocks and Physiological Identifiers

Storing tuple data and transaction information separately introduces another challenge: as the two are no longer co-located, the system requires global tuple identifiers to associate the two. Physical identifiers, such as 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). Such lookups are a serious bottleneck because every transactional operation in our system performs at least one lookup. To overcome this problem, the system organizes storage in blocks, and uses a physiological scheme for identifying tuples.

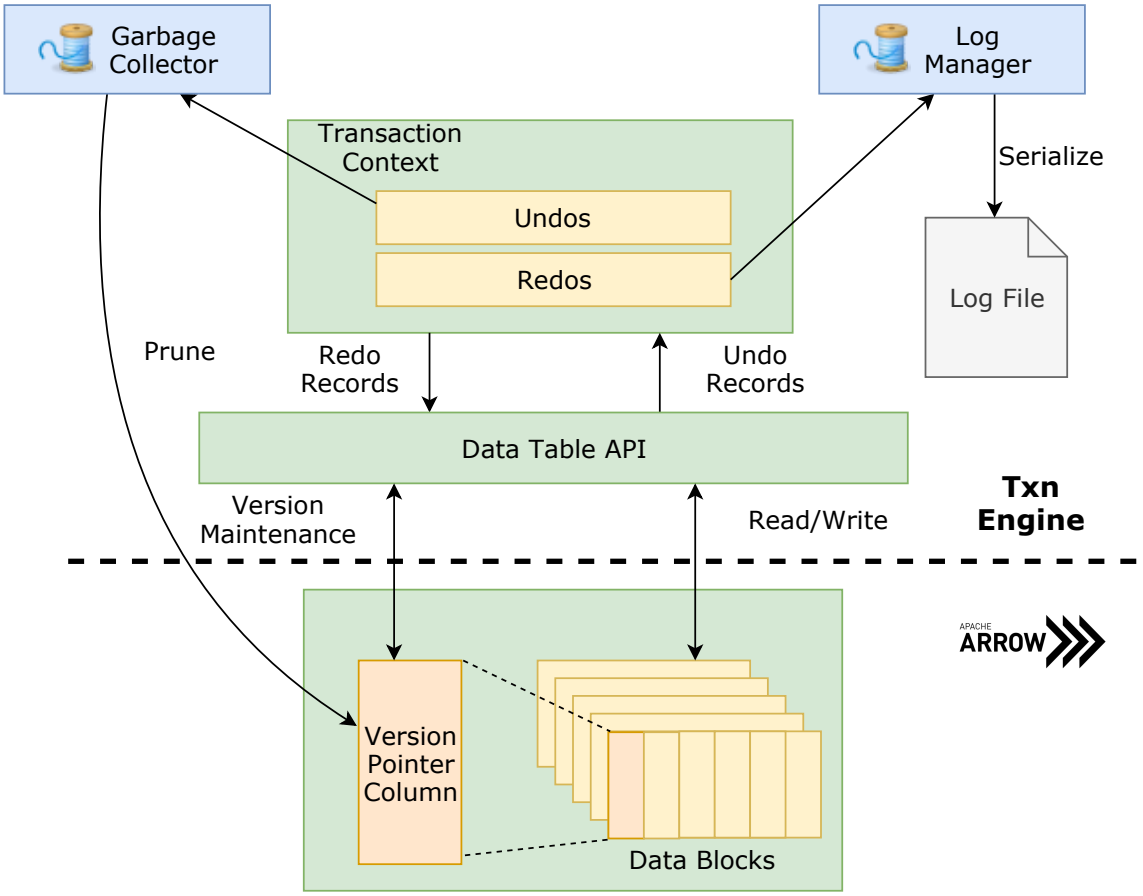


Figure 4.1: System Architecture – CMDB's transactional engine is minimally intrusive to the underlying storage to maintain compatibility with the Arrow storage format.

Each block in our system is 1 MB and stores attributes in a PAX-style layout [Ailamaki et al. [2002]]. This means data is organized in each block in columnar format, but all columns of a tuple are located within the same block. Every block has a layout structure associated with it that consists of (1) the number of slots within a block, (2) a list of the attributes and their sizes, and (3) the location offset for each column from the starting address of a block. Each column and its associated bitmap are aligned at 8-byte boundaries. The system calculates layout once for a table when the application creates it, and uses it to initialize and interpret every block in the table. In PAX, having an entire tuple located within a single block reduces the number of disk accesses on a write operations. Such

benefits are less significant in an in-memory system; instead our system uses blocks as a useful logical grouping for tuples that are inserted close to each other chronologically that the system can use as a unit for transformation. Having all attributes of a tuple located within a single block also ensures that tuple will never be partially available in Arrow format.

Every tuple in the system is identified by a `TupleSlot`, which is a combination of (1) physical memory address of the block the tuple resides in, and (2) a logical offset in the block. Combining these with the pre-calculated block layout, the system can compute the physical pointer to each attribute in constant time. To pack both values into a single 64-bit value, the system aligns all blocks to start at 1 MB boundaries within the address space of the process. A pointer to the start of a block will then always have its lower 20 bits set to zero, which the system uses to store the offset. There can never be a situation where the 20 bits is not enough because they are sufficient to address every single byte within the block.

We implement this using the C++11 keyword `alignas` as a hint to the DBMS's memory allocator when creating new blocks. It is possible to further optimize this process by using a custom allocator specialized for handing out 1 MB chunks. The system can also store extra information in the address in a similar fashion; because it reserve space within each block for headers and version information, and because x86 machines currently do not utilize the full 64 bits in a pointer, there are many bits to spare in a tuple slot. This allows the system to pack much information into the tuple identifier while keeping it in register for good performance when passing one around.

4.3 Garbage Collection

The garbage collector (GC) in our system is responsible for pruning the version chain and freeing any associated memory [Larson et al. [2011], Tu et al. [2013], Yu et al. [2014], Lee et al. [2016]]; recycling of deleted slots is handled in the transformation to Arrow described in 5.3. Because all versioning information is stored within a transaction's buffers, the GC only needs to process transaction objects to free used memory. The GC maintains a

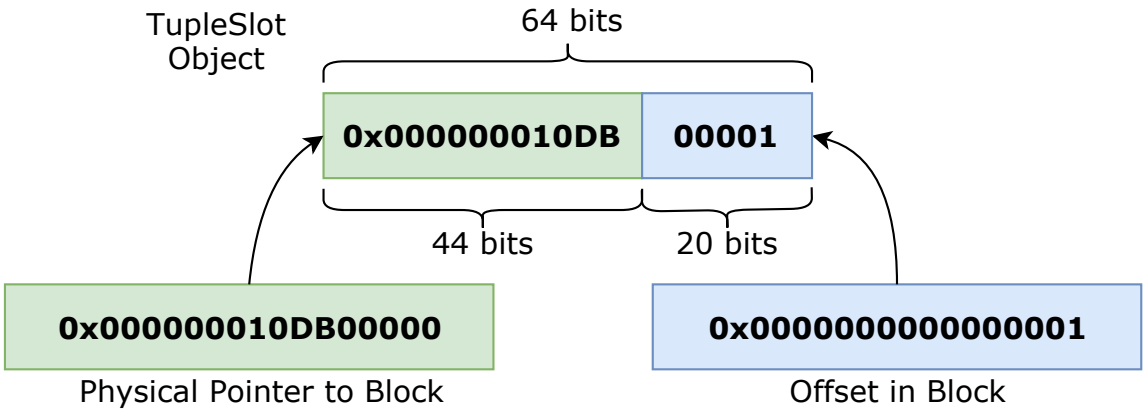


Figure 4.2: TupleSlot – By aligning blocks to start at 1 MB boundaries, the DBMS packs the pointer to the block and the offset in a single 64-bit word.

queue of completed transactions, and wakes up periodically to iterate through the queue. At the start of each run, it first checks the transaction engine's transactions table for the oldest active transaction's *start* timestamp; changes of any transaction committed before this timestamp are no longer visible and are safe for removal. For each such transaction, the GC first inspects its undo records for any variable length values to clean up, and computes a set of `TupleSlots` with invisible delta records on their version chains. The GC then truncates the version chain for each `TupleSlot` exactly once, so as to avoid the potentially quadratic operation of finding and unlinking each individual record. It is unsafe, however, to deallocate objects at this point. Transactions in our system stop traversing the version chain only after seeing a record that is invisible to it. Active transactions can therefore still be accessing the record GC truncated. 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. This is similar to an epoch-protection mechanism [Chandramouli et al. [2018]], and can be generalized to ensure thread-safety for other aspects of the DBMS as well.

The GC process traverses every single update installed in the system exactly once and requires no additional full table scans and easily keeps up with high throughput. The combination of these two facts gives us leeway to incorporate additional checks on each

garbage collector run without significant performance impact. We take advantage of this to compute statistics and determine if a block is becoming cold, as we will describe in detail in 5.2.

4.4 Logging and Recovery

Our system achieves durability through write-ahead logging and checkpoints [Mohan et al. [1992], DeWitt et al. [1984]]. The steps to logging a transaction in our system is analogous to GC. Each transaction has a redo buffer for physical after-images of writes. At commit time, the transaction appends a commit record to the redo buffer and adds itself to the log manager's flush queue. The log manager runs in the background and serializes the changes into an on-disk format before flushing to persistent storage. The system relies on an implicit ordering of the records instead of explicit sequence numbers to recover. Each transaction writes changes to its local buffer in the order that they occur. Among transactions, the system orders the records according to their globally unique *commit* timestamp, written in the commit record on disk.

Our system implements redo and undo buffers in the same way, using the same pool of buffer segments for both. As an optimization, instead of holding on to all records for a transaction's duration, the system flushes out redo records incrementally; when a transaction's redo buffer reaches a certain size, the system flushes partial changes to the log. This is safe because in the case of an abort or crash the transaction's commit record is not written, and thus the recovery process ignores these changes. In our implementation, we limit the size of a transaction's redo buffer to a single buffer segment.

One caveat worth pointing out is that the entries in the redo buffer are not marshalled bytes, but direct memory images of the updates, with paddings and swizzled pointers. The logging thread therefore has to serialize the records into their on-disk formats before flushing. This process happens off the critical path, however, and we have not found it to be a significant overhead in our evaluations.

The system performs group commit and only invokes `fsync` after some threshold in log size or in wait time is reached. A transaction is considered committed by the rest of the system as soon as its redo buffer is added to the flush queue. All future operations on the transaction's write-set are speculative until its log records are on disk. To address this, the system assigns a callback to each committed transaction for the log manager to notify when the transaction's records are persistent. The system refrains from sending a transaction's result to the client until the callback is invoked. In this scheme, changes of transactions that speculatively read or update the write-set of another transaction are not public until the log manager processes their commit record, which happens after the transaction they speculated on is persistent. We implement callbacks by embedding a function pointer in the commit record on the redo record; when the logging thread serializes the commit record, it adds that pointer to a list of callbacks to invoke after the next `fsync`. Because the log manager guards against the anomaly above, the DBMS requires read-only transactions to also obtain a commit record and wait on the log manager, but the log manager can skip writing this record to disk after processing the callback.

Log records identify tuples on disk using `TupleSlots`, even though the physical pointers are invalid on reboot. The system maintains a mapping table between old tuple slots to their new physical locations in recovery mode. This table adds no additional overhead. Even if the system uses logical identifiers on disk, it still needs to map logical identifiers to physical locations. It is possible to integrate some storage layout optimizations in this remapping phase, performing an implicit compaction run and possibly rearranging tuples based on access frequency.

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 in a transaction 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 of our database, 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 [Ren et al. [2016]] where all checkpointed tuples are committed, but not necessarily from the same snapshot. Because recovery in our

system has no undo phase, the system is still guaranteed to recover to a consistent snapshot under this scheme. As a further optimization, the checkpoint records the oldest committed version for each block, so some entries in the log can be safely skipped in the recovery pass.

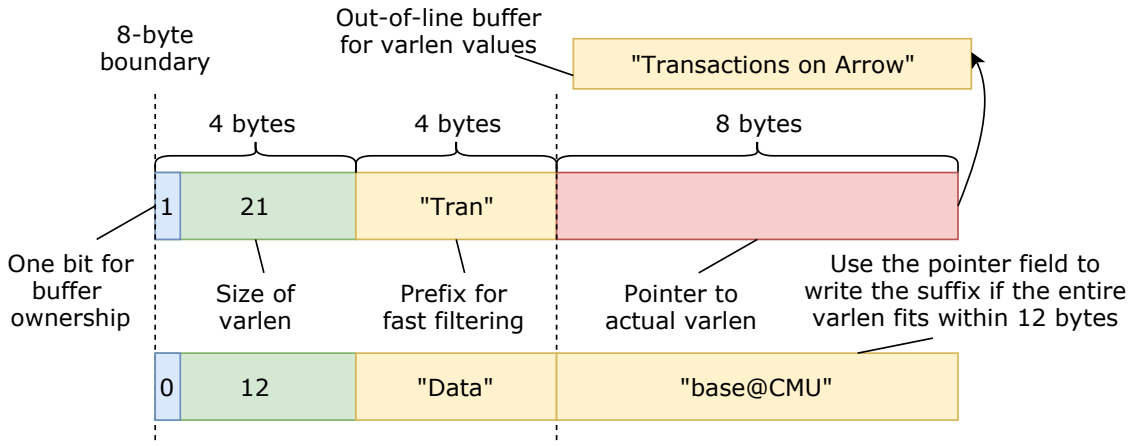


Figure 5.1: Variable-Length Value Storage – The system uses 16 bytes to track variable-length values as a fixed-size column in a block.

Chapter 5

Block Transformation

As discussed in 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. In this section, we describe this modified Arrow format, introduce a mechanism to detect cold blocks, and present our algorithm for transforming cold blocks to full Arrow.

5.1 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 cold and mostly accessed by read-only queries. Thus, to reduce write amplification, the DBMS delays writes until a block is cold. We therefore relax the Arrow format for the hot portion of the database to improve update performance. This forces all readers, both internal and external, to access data via the transactional slow path that materializes versions. We contend that the cost is acceptable as this materialization happens only for a small portion of the database.

There are two sources of write amplifications in Arrow: (1) it disallows gaps in a column and (2) it stores variable-length values consecutively in a single buffer. Our relaxed format overcomes these issues by adding a validity bitmap in the block header for deletions, and metadata for each variable-length value in the system. As shown in 5.1, within a `VarlenEntry` field, the system maintains a 4-byte integer size field and an 8-byte pointer that points to the underlying string. Each `VarlenEntry` is padded to 16 bytes for alignment, and the system uses the four additional bytes to store a prefix of the string to enable quick filtering in a query. If a string is shorter than 12 bytes, the system stores it entirely within the object using the pointer field. Transactions only access the `VarlenEntry` and do not reason about Arrow storage. This allows the system to write updates to `VarlenEntry` instead of the underlying Arrow storage, turning a variable-length update into fixed-length, which is constant-time. This process is demonstrated in 5.2.

Any readers accessing Arrow storage will be oblivious to the update in `VarlenEntry`. The system uses a status flag and a counter in the block header to coordinate access in this case. For a cold block, the status flag is set to frozen and readers add one to the counter when starting a scan and subtract one when they are done. When a transaction needs to update a cold block, it first sets the flag in the block header indicating that the block is hot, forcing any future readers to materialize instead of reading in-place. It then spins on the counter and wait for lingering readers to leave the block before proceeding with the update. There is no transformation process required for a transaction to modify a cold block because our relaxed format is a superset of the original Arrow format. This concurrency control

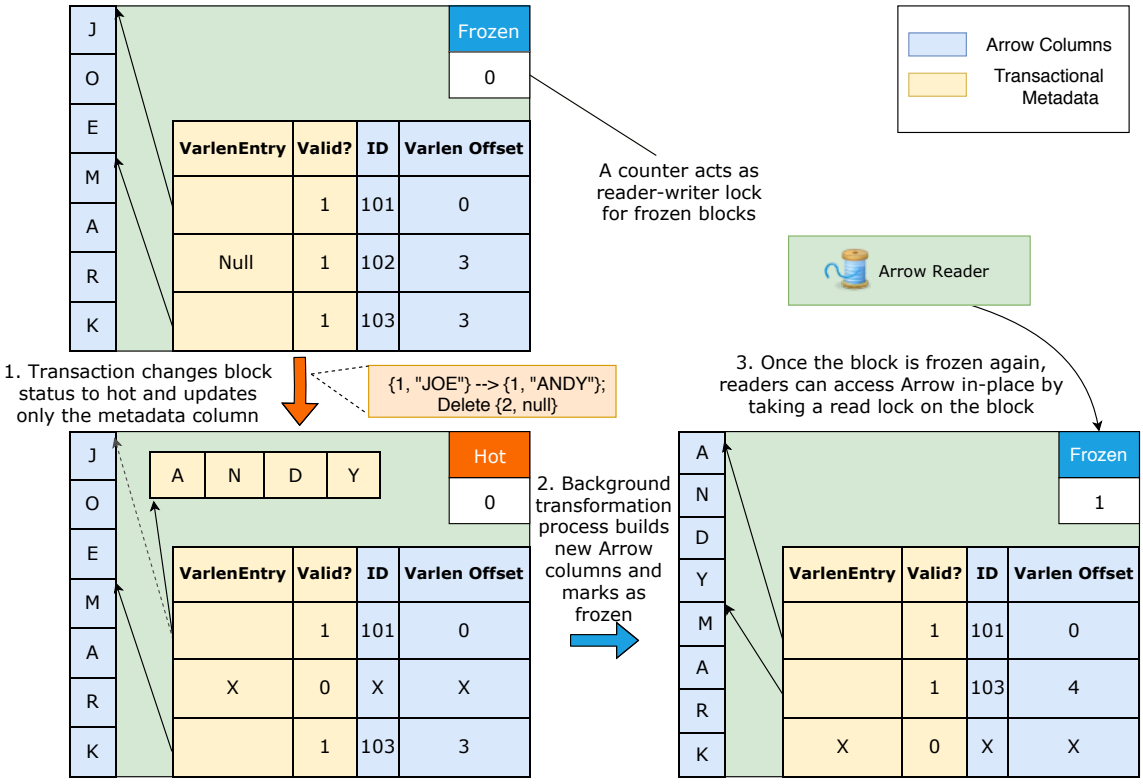


Figure 5.2: Relaxed Columnar Format – The system briefly allows non-contiguous memory to support efficient mutation of Arrow blocks.

method is similar to a reader-writer latch. The difference is that multiple writers are allowed to coexist, and that readers are not allowed to reacquire the latch; once a block is hot it stays hot until a background process transforms the block back to full Arrow compliance. We will discuss this process next.

5.2 Identifying a Cold Block

The system keeps access statistics about each block to determine if it is cooling down. Collecting them as transactions operate on the database adds overhead to the critical path [Funke et al. [2012]], which is unacceptable for OLTP workloads. Our system trades

off the quality of such statistics for better scalability and performance, and 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 GC's pass through all undo records (4.3). From each undo record, the system infers the modification type (i.e., delete, insert, update) and the `TupleSlot` it pertains to. Time measurement, however, is difficult because the system cannot measure how much time has elapsed between the modification and invocation of the GC. To address this, the system notes the time of each GC invocation and use that time as an approximation for all modifications processed in this GC run. If transactions have life time shorter than the frequency of GC (10 ms), this approximated time is never earlier than the actual modification and is late by at most one GC period. This "GC epoch" is a good enough substitute for real time for OLTP workloads, because most write transactions are short-lived [Stonebraker et al. [2007]]. Once the system identifies a cold block, it adds the block to a queue for a background transformation thread to process later.

The user can modify the threshold time value based on how aggressively they want the system to transform blocks. The optimal value of the threshold is workload-dependent. A value too low reduces transactional performance because of wasted resources from frequent transformations. A value too high results in reduced efficiency for Arrow readers. The policy used can be more sophisticated and even learned on-the-fly [Pavlo et al. [2017]], but this is not central to the arguments of this thesis and is thus left for future work.

This observation scheme misses changes from aborted transactions because GC does not unlink their records. Additionally, accesses are not immediately observed because of latency introduced by the GC queue. As a result, the observer may identify a block as cold by mistake when they are still being updated. The DBMS reduces this impact by ensuring that the transformation algorithm is fast and lightweight. There are two failure cases here: (1) a user transaction aborts due to conflicts with the transformation process or (2) the user transaction stalls. There is no way to prevent both cases, and eliminating one requires heavy-weight changes that make the other more severe. 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, but allows preemption from user transactions as much as possible, and has a short critical section.

5.3 Transformation Algorithm

Once the system identifies cooling blocks, it performs a transformation pass to prepare the block for readers expecting Arrow. As mentioned in 5.1, the DBMS needs to first compact each block to eliminate any gaps and then copy variable-length values into a new contiguous buffer. There are three approaches to ensure safety: (1) copying the block, (2) performing operations transactionally, or (3) taking a block-level lock to prevent races with potential user transactions. None of these is ideal. The first approach is expensive, especially when most of the block data is not changed. Making the transformation process transactional adds additional overhead and results in user transaction aborts. A block-level lock stalls user transactions and limits concurrency in the common case even without transformation. As shown in 5.3, our system uses a hybrid two-phase approach that combines transactions for tuple movement, but elides transactional protection and locking the block when constructing variable-length buffers for Arrow. We now discuss these phases in more detail.

5.3.1 Phase #1: Compaction

The access observer identifies a *compaction group* as a collection of blocks with the same layout to transform. Within a group, the system uses tuples from less-than-full blocks to fill gaps in others and recycle blocks when they become empty. The DBMS uses a transaction in this phase to perform all the reads and subsequent operations.

The DBMS starts by scanning the allocation bitmap of every block to identify empty slots that need to be filled. The goal of this phase is for all tuples in the compaction group to be "logically contiguous". Consider a compaction group consisting of t tuples and there are b many blocks with each block having s many slots, after compaction, there should

be $\lfloor \frac{t}{s} \rfloor$ many blocks completely filled, one block with slots in $[0, t \bmod s)$ filled, and the others empty.

The system performs shuffling of tuples between blocks as a delete followed by an insert using the transformation transaction. This operation is potentially expensive if the transaction needs to update indexes. The goal of our algorithm, therefore, is to minimize the number of such delete-insert pairs. We now present our algorithm. Observe first that the problem can be decomposed into two parts:

1. Select a block set F to be the $\lfloor \frac{t}{s} \rfloor$ blocks that are filled in the final state. Also select a block p to be partially filled and hold $t \bmod s$ tuples. The rest of the blocks, E, are left empty.
2. Fill all the gaps within $F \cup \{p\}$ using tuples from $E \cup \{p\}$, and reorder tuples within p to make them laid out contiguously.

Define Gap_f to be the set of unfilled slots in a block f, Gap'_f to be the set of unfilled slots in the first $t \bmod s$ slots in a block f, $Filled_f$ to be the set of filled slots in f, and $Filled'_f$ to be the set of filled slots not in the first $t \bmod s$ slots in f. Then, for any legal 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, given F, p, and E, 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 problem is now reduced to finding the optimal F, p and E.

1. Scan through each block's allocation bitmap for empty slots.
2. Sort the blocks by number of empty slots in ascending order.
3. Pick out the first $\lfloor \frac{t}{s} \rfloor$ blocks to be F.
4. Pick an arbitrary block as p and the rest as E.

This choice bounds our algorithm to within $t \bmod s$ of the optimal number of movements, and can be used as an approximate solution. Every gap in F needs to be filled with one movement, and our selection of F results in fewer movements than any other choice.

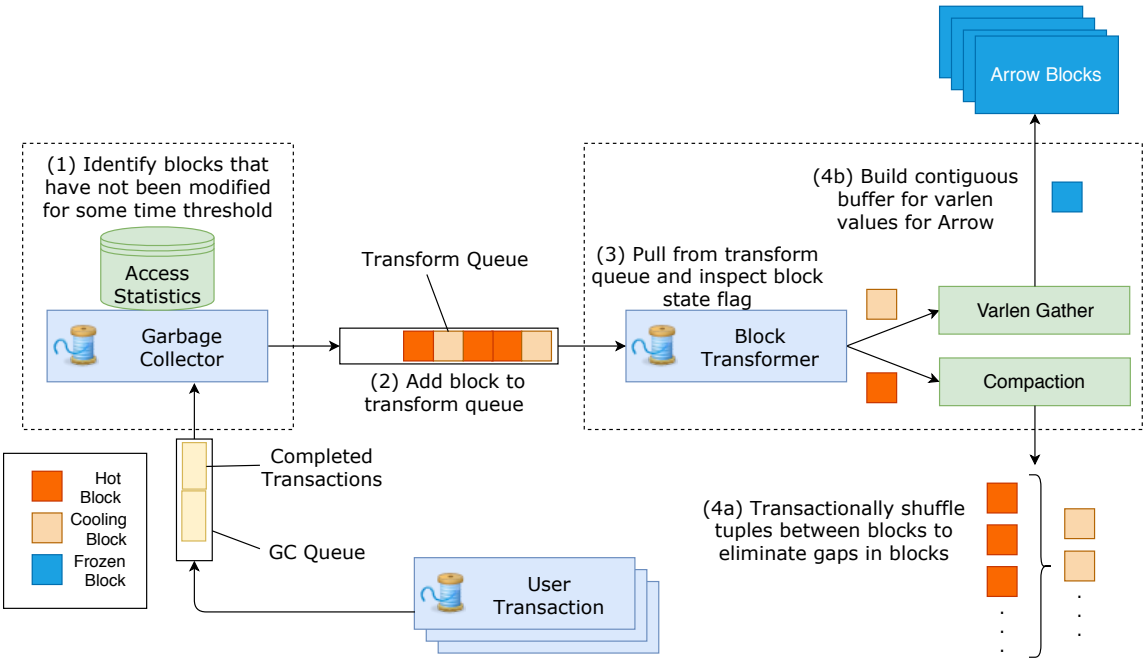


Figure 5.3: Transformation to Arrow – CMDB implements a pipeline for lightweight in-memory transformation of cold data to Arrow.

In the worst case, the chosen p is empty in the first $t \bmod s$ slots, and the optimal one is filled, resulting in at most $t \bmod s$ movements more than the optimal choice. In practice, the system chooses the block with fewest unfilled slots not in F to be p, and the worst case scenario is unlikely to happen. The system then sequentially fills all empty slots from $F \cup \{p\}$ using tuples from $E \cup \{p\}$.

For the optimal solution, the algorithm needs to additionally find the best value of p, which it achieves by trying every block. Given p, the algorithm picks the next best $|F|$ blocks for F, and calculates $|Gap'_p| + \Sigma_{f \in F}|Gap_f|$ as the total number of movements. An additional scan of the block header is required, however, to obtain the values of Gap'_p for each p. The marginal reduction in number of movements does not always justify this additional scan. We evaluate this algorithm experimentally in 7.

5.3.2 Phase #2: Gathering

The system now moves variable-length values with the transformed block into a contiguous buffer in compliance with Arrow. As we have discussed previously, neither transactional updates nor block locks are efficient enough by themselves. Therefore, we present a novel scheme of multi-stage locking that relies on the garbage collector to guard against races that otherwise cannot be prevented without explicit locking for every operation. In the naïve implementation, it suffices to continue updating the table in the same transaction used by the compaction phase. The system allocates a buffer, copies scattered variable length values into it, and updates the tuple's `VarlenEntry` columns to point to the new buffer. Eventually, either the transformation transaction is aborted due to conflicts, or it succeeds in updating every tuple in the blocks being transformed. Because our system is no-wait, any other transaction that attempt to update values in those blocks will be aborted, and the correctness of the operation is guaranteed.

Unfortunately, as we will demonstrate in 7, this naïve approach has suboptimal performance. This is because transactional updates do additional work to ensure isolation; transactional updates have irregular memory access patterns compared to sequential in-place updates, and uses the expensive compare-and-swap instruction for every tuple. Because we already assume the blocks being processed here are no longer transactionally updated, transactions incur much overhead to guard against contention that rarely happens. Therefore, to speed up the gathering phase further, it is necessary to elide transactional protection and use locking for the duration of the operation. That said, introducing a shared lock at the block level may slow down the common case scenario as well. To achieve the best of both worlds,

We extend the block status flag with two additional values: cooling and freezing. The former indicates intent of the transformation thread to lock, while the latter serves as an exclusive lock that blocks user transactions. User transactions are allowed to preempt the cooling status using a compare-and-swap and set the flag back to hot. When the transformation algorithm has finished compaction, it sets the flag to cooling and scans through the block to check for any version pointers indicating concurrent modification

of the block. If there are no versions, and the cooling status has not been preempted, the transformation algorithm can change the block's status to freezing and obtain the exclusive lock. The cooling flag acts as a sentinel value that detects any concurrent transactions that could have modified the block while the algorithm is scanning.

This scheme of access coordination introduces a race as shown in 5.4. A thread could have finished checking the status flag and was scheduled out. Meanwhile the transformation algorithm runs and sets the block to freezing. When the thread wakes up again, it proceeds to update, which is unsafe. Normally, there is no way to prevent this without locks, because the checking of block status and the update form a critical section but cannot otherwise be atomic.

To address this, the system relies on the visibility guarantees made by GC. Recall from 4.3 that GC does not prune any versions that is still visible to running transactions. If the algorithm sets the status flag to cooling after all the movement of tuples but before the compaction transaction commits, the only transactions that could incur the race in 5.4 must overlap with the compaction transaction. Therefore, as long as such transactions are alive, the garbage collector cannot unlink records of the compaction transaction. The algorithm can commit the compaction transaction and wait for the block to reappear in the processing queue for transformation. The status flag of cooling guards against any transactions modifying the block after the compaction transaction committed. If the transformation algorithm scans the version pointer column and find no active version, then any transaction that was active at the same time as the compaction transaction must have ended. It is therefore safe to change the flag of the transformed block into freezing.

After the transformation algorithm has obtained exclusive access to the block, it scans each variable-length column and performs the gathering process in-place. In the same pass, the algorithm also gathers metadata information such as null count for Arrow. When the process is complete, the system can safely mark the block as frozen and allow access from in-place readers. Throughout the process, transactional reads of the tuples within the block can still proceed regardless of the block status. The gathering phase changes only the physical location of values and not the logical content of the table. Because a write to any

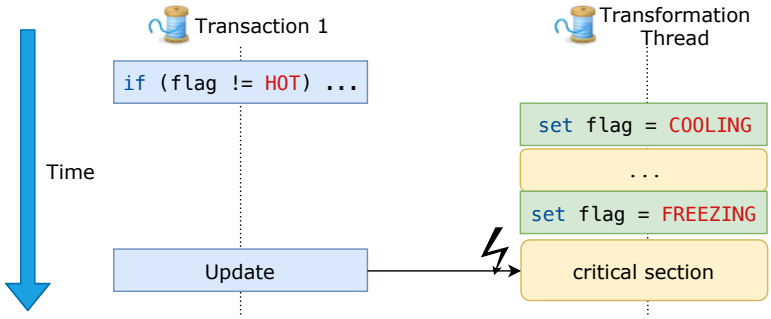


Figure 5.4: Check-and-Miss on Block Status – A naïve implementation results in a race within the critical section of the gathering phase.

aligned 8-byte address is atomic on a modern architecture [int], reads can never be unsafe as all attributes within a block are aligned.

5.4 Additional Considerations

We have presented our algorithm for transforming cold blocks into the Arrow format. We now demonstrate its flexibility by discussing an alternative backend of dictionary compression using the same algorithm. We also give a more detailed description of memory management within the algorithm for thread and memory safety.

5.4.1 Dictionary Encoding

Dictionary encoding is a common compression technique used in read-optimized databases, where each attribute value is replaced with an index into a value table [Holloway and DeWitt [2008]]. Arrow supports dictionary encoded columns natively in its DDL and RPC protocol. It is possible to implement an alternative gathering phase where instead of building a contiguous variable-length buffer, the system builds a dictionary and an array of dictionary codes. Much of the algorithm remains the same; the only difference is that within the critical section of the gathering phase, the algorithm now scans through the block twice. On the first scan, the algorithm builds a sorted set of values for use as a dictionary. On the second

scan, the algorithm replaces pointers within `VarlenEntry`s to point to the corresponding dictionary word and builds the array of dictionary codes. Although the algorithm remains the same, dictionary compression is an order of magnitude more expensive than a simple variable-length gather. This difference calls for a more conservative transformation strategy to minimize potential interference with running transactions.

5.4.2 Memory Management

Since the algorithm never blocks readers, the system cannot deallocate memory that can be visible to a transaction running concurrently with the transformation process. In the compaction phase, because the system performs writes using transactions, the GC can handle memory management. The only caveat here is that when moving tuples, the system needs to make a copy of any variable-length value being pointed to rather than simply copying the pointer. This is because the garbage collector does not reason about the transfer of ownership of variable-length values between two versions, and will deallocate them after seeing the deleted tuple. We do not observe this to be a performance bottleneck.

In the gathering phase, because the system elides transactional protection, memory management is more difficult. Fortunately, as discussed in 4.3, our system's GC implements timestamp-based protection that is similar to Microsoft's Faster epoch-based framework [Chandramouli et al. [2018]]. We extend our GC to accept arbitrary actions associated with a timestamp in the form of a callback, which it promises to invoke only after the oldest alive transaction in the system is younger than the given timestamp. The system utilizes this framework to register an action that reclaims memory for this gathering phase, which will be run by the GC only after no running transaction can possibly access it.

Chapter 6

Data Export

Now that we have described how the DBMS converts data blocks into the Arrow format, we discuss how to enable external applications to access these blocks. In this section, we present five approaches for enabling applications to access the DBMS's native Arrow storage to speed up analytical pipelines. We discuss these alternatives in the order of the engineering effort required to change an existing system (from easiest to hardest).

6.1 Improved Wire Protocol for SQL

There are still good reasons for applications to interact with the DBMS exclusively through a SQL interface (e.g., developer familiarity, existing ecosystems). If the DBMS provides other access methods as an alternative to SQL, then developers have to reason about their interactions with the transactional workload. In this case, Arrow improves the performance of network protocols used in SQL interfaces. As Raasveldt and Mühleisen [2017] pointed out, row-major wire protocols are slow, and network performance can be improved by using a vectorized protocol. The DBMS can adopt Arrow as the format in such a protocol and leave other components of the system unchanged, such as the query engine. This approach effectively reduces the overhead of the wire protocol in data movement. It does not, however, achieve the full potential speed-up from our storage scheme. This is because

the DBMS still copies data and assembles it into packets, and then the client must parse these packets. Although Arrow reduces the cost for both, it is possible to eliminate these two steps altogether.

6.2 Alternative Query Interface

As mentioned in 2.1, workloads that extract data in bulk for external processing do not require the full power of a transactional DBMS. Such workloads often execute read-only queries at a lower isolation level, and use little of the vast feature set of a DBMS. Much of the DBMS wire protocol, such as settings configurations and transactional status, is useless for such workloads. It is therefore possible to introduce a specialized interface that is simpler than SQL and optimize it for the purpose of data export. Arrow provides a native RPC framework based on gRPC called **Flight** [apa [b]] that avoids expensive serialization when transmitting data through low-level extensions to gRPC's internal memory management. This allows a DBMS using this framework to expose an RPC service to applications to retrieve entire blocks.

Although the RPC model can be used to support all of SQL, we observe most significant speed-up for queries that do not generate new attributes in its result set. If the query only filters data and does not compute or derive new attributes (e.g., aggregations, window functions), Arrow Flight can take full advantage of the existing Arrow blocks and achieve zero-copy communication. The DBMS can specialize query handling logic for these cases and compute only a selection vector over data blocks. The data is then directly streamed to the client using Arrow Flight. To handle hot blocks, the system starts a transaction to read the block and constructs the Arrow buffer on-the-fly. The cost of handling hot blocks is no worse than the current approach of materializing and serializing the result set. Compared to the previous approach, this approach eliminates the deserialization cost entirely for the client, but still requires the DBMS to assemble packets over TCP/IP.

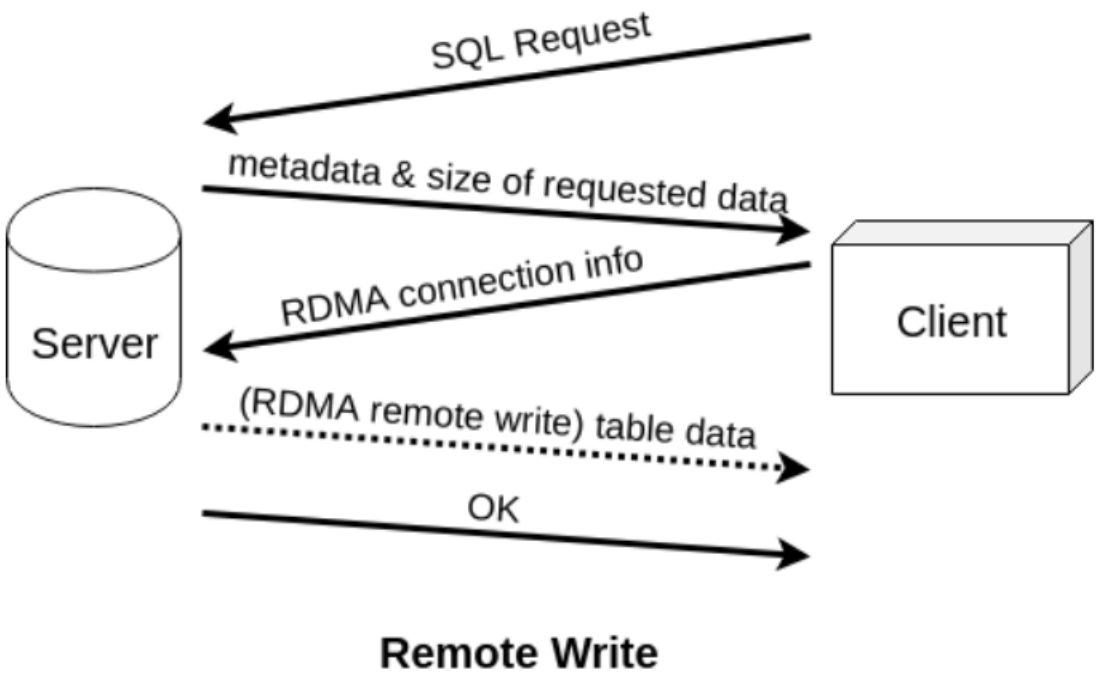


Figure 6.1: Client-Side RDMA – An illustration of the message flow between DBMS and client if the DBMS implements client-side RDMA

6.3 Client-Side RDMA

To eliminate the serialization step altogether, one can consider Remote Direct Memory Access (RDMA) technologies. RDMA bypasses the OS's network stack for both sides of the connection and permits high-throughput, low-latency networking. With this approach, the client sends a query to the DBMS through the RPC protocol described above. Instead of returning data blocks directly, the DBMS returns metadata about the result size and its schema. The client responds with instructions for the server about where to write the result in the client's memory over RDMA. The DBMS then notifies the client through a RPC response when the transfer is complete. Aside from increased data export throughput, another benefit of using a client-side approach is that the client's CPU is idle during RDMA operations. Thus, the client can start working on partially available data, effectively

pipelining the data processing. To achieve this, the DBMS can send messages for partial availability of data periodically to communicate if some given chunk of data has already been written. This approach reduces the network traffic close to its theoretical lower-bound, but still requires additional processing power on the server to handle and service the RPC request. An illustration of this is shown 6.1.

6.4 Server-Side RDMA

Allowing clients to read the DBMS's memory over RDMA bypasses the CPU when satisfying bulk export requests. This is beneficial to an OLTP DBMS because the system no longer needs to divide its CPU resources between serving transactional workloads and bulk-export jobs. Achieving server-side RDMA, however, requires major changes to the DBMS. Firstly, RDMA is a single-sided communication paradigm, and the server is unaware the completion of a client request. This makes it difficult to lock the Arrow block and guard against updates into them. If the system waits for a separate client completion message, the round-trip time introduces latency to any updating transactions. To avoid this, the DBMS has to implement some form of a lease system to invalidate readers for transactional workloads that have stricter latency requirements. In addition to introducing complexity in the concurrency control protocol of the DBMS, this approach also requires that the client knows beforehand the address of the blocks it needs to access, which must be conveyed through a separate RPC service or some external directory maintained by the DBMS. We envision these challenges to be significant in achieving server-side RDMA for data export. An illustration of this is shown in 6.2.

6.5 External Tool Execution on the DBMS

Server-side and client-side RDMA allows external tools to run with data export overhead that is close to its theoretical lower bound. The major issue, however, is that RDMA requires specialized hardware and is only viable when the application is co-located in the

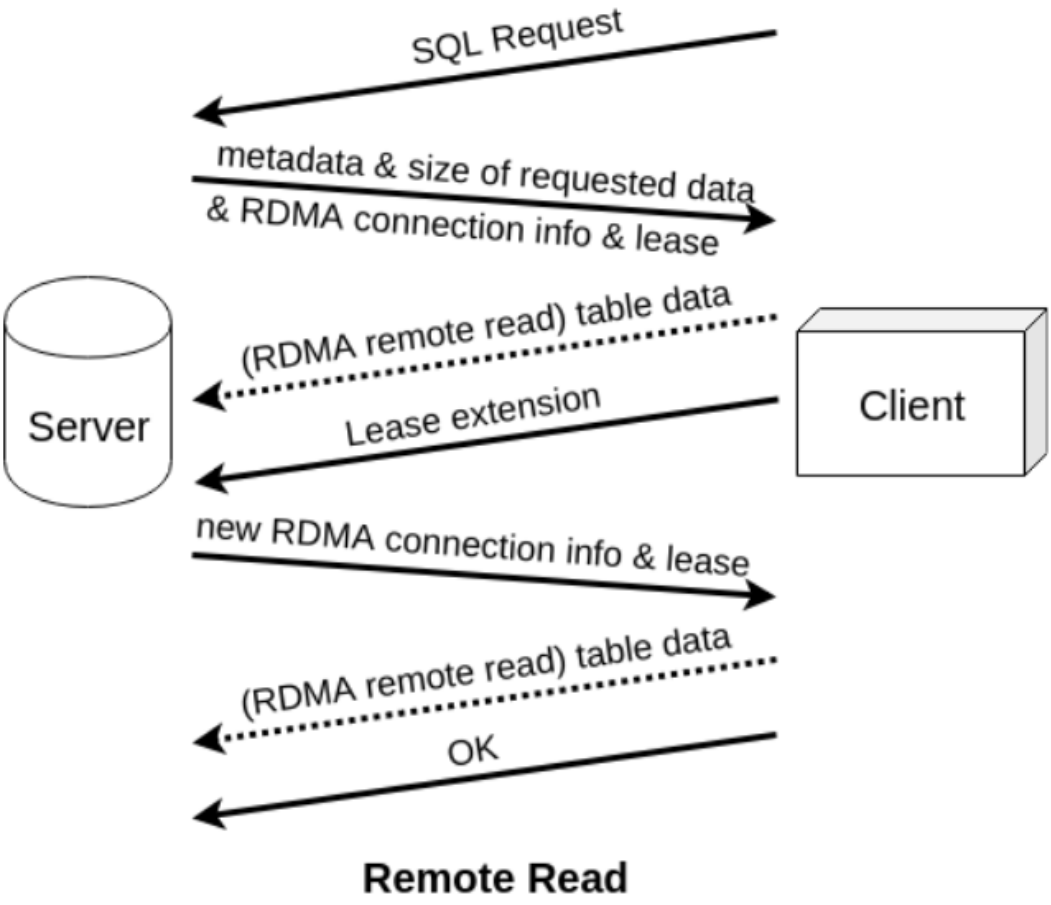


Figure 6.2: Server-Side RDMA – An illustration of the message flow between DBMS and client if the DBMS implements server-side RDMA. As shown, the message flow involved is much more complex than client-side RDMA.

same data center as the DBMS. This is unlikely for a data scientist working on a personal work station. To improve availability, we can leverage Arrow as an API between the DBMS and external tools. Because Arrow is a standardized memory representation of data, if external tools accept Arrow as input, it is possible to run the program on the DBMS by replacing its Arrow pointers with mapped memory images from the DBMS process. his brings a set of problems involving security, resource allocation, and software engineering. By making an analytical job portable across machines, it 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 to efficiently execute it with low data movement cost.

Chapter 7

Evaluation

We now present an analysis of our system architecture and design choices. For this evaluation, we implemented our storage engine in the CMDB DBMS [cmu]. 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 load the database into the same NUMA region to eliminate interference from cross-socket communication. All transactions execute as JIT-compiled stored procedures with logging enabled. We run each experiment ten times and report the average result over all runs.

We first evaluate our OLTP performance and interference from the transformation process. We then provide a set of micro-benchmarks to study the performance characteristics of the transformation algorithm. Next, we compare data export methods in our system against the common methods used in practice.

7.1 OLTP Performance

In this first experiment, we measure the DBMS's OLTP performance to demonstrate the viability of our storage architecture and that our proposed transformation process is lightweight. We use TPC-C [The Transaction Processing Council [2007]] with one

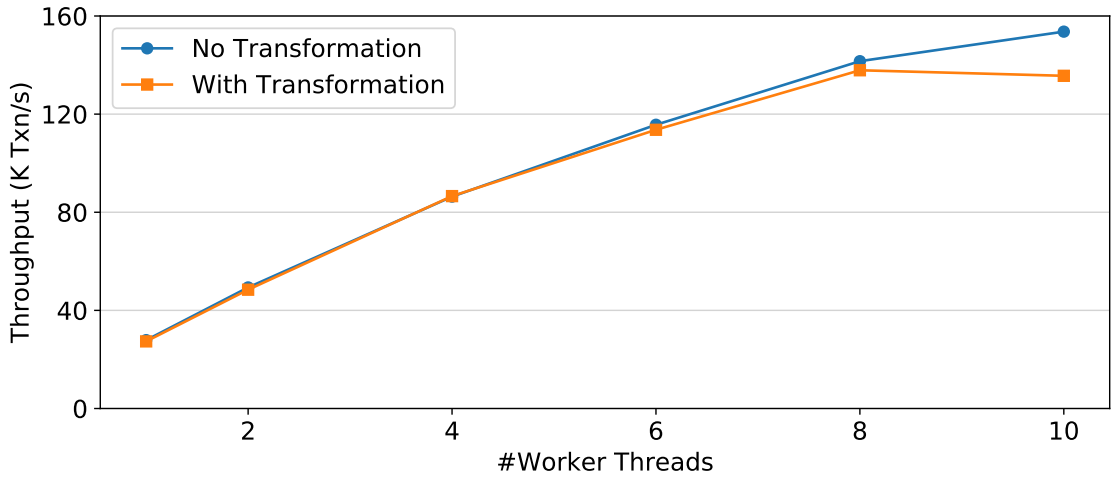


Figure 7.1: OLTP Performance: Throughput – Throughput measurements of the DBMS for the TPC-C workload when varying the number of worker threads.

warehouse per client to minimize contention. CMDB uses the OpenBw-Tree for primary and secondary indexes for each table [Wang et al. [2018]].

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 CPU cores available to be number of worker threads per trial plus logging and GC threads to account for the additional resource required by the transformation process. We deploy the DBMS with three transformation configurations: (1) disabled, (2) variable-length gather, and (3) dictionary compression. For trials with CMDB's block transformation enabled, we use an aggressive threshold time of 10 ms and 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 compaction group.

The results in 7.1 show that the DBMS achieves good scalability and incurs little overhead from the transformation process (1.8–15%). The interference becomes more prominent as the amount of work increases for the transformation thread due to more workers executing transactions. At 10 worker threads, there is reduced scaling or even a slight drop when transformation is enabled. This is because our machine has 10 CPU cores, and thus the GC and logging threads do not have dedicated cores. The problem of

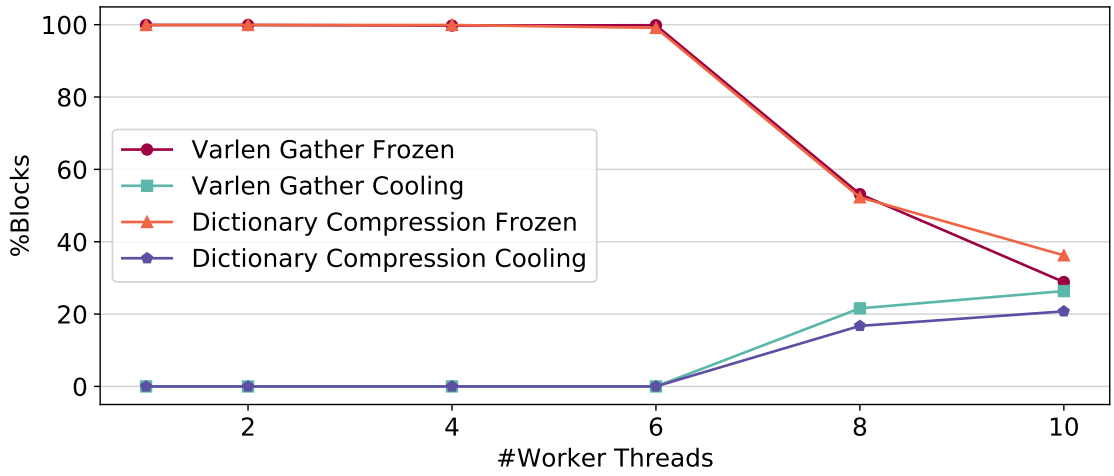


Figure 7.2: OLTP Performance: Block State Coverage – Block state coverage of the DBMS at the end of a TPC-C run when varying the number of worker threads.

threads swapping is made worse with the additional transformation thread. As expected, using dictionary compression as the gathering step increases the performance impact of our transformation process, as it is computationally more intensive.

To better understand this issue, we measured the abort rate of transactions and the number of transactions that was stalled due to the transformation process. We did not observe a statistically significant amount of change in abort rates and a negligible number of stalled transactions (<0.01%) for all trials.

In 7.2, we report the percentage of blocks in the cooling and frozen state at the end of each run. We omit results for the `ITEM` table because it is a read-only table and its blocks are always frozen. Because the DBMS does not transform any blocks that have empty slots and can be inserted into, these blocks remain in the hot state and thus we do not achieve 100% block coverage. These results show that the DBMS achieves nearly complete coverage up to 6 workers, but starts to lag behind for higher numbers of worker threads. This is because of increased work and more contention on limited CPU resources when approaching the number of physical cores available. In accordance with our design goal, the transformation process yields resources to user transactions in this situation, and does not result in a significant drop in transactional throughput.

7.2 Transformation to Arrow

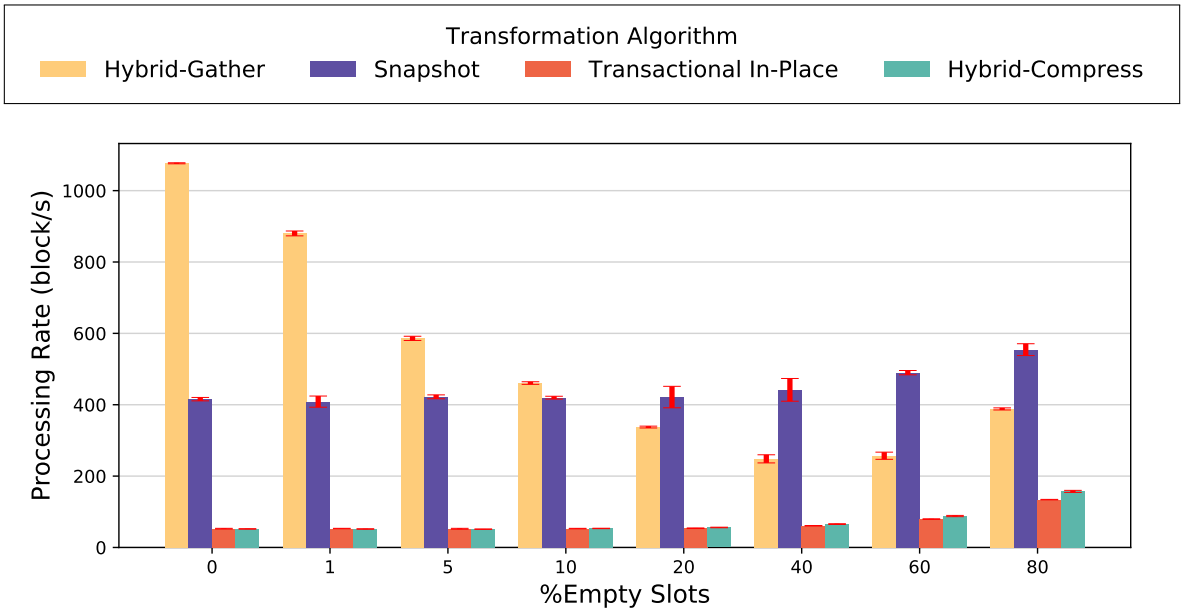
We next evaluate our transformation algorithm and analyze the effectiveness of each sub-component. We use micro-benchmarks to demonstrate the DBMS's performance when migrating blocks from the relaxed Arrow format to their canonical form. Each trial of this experiment simulates one transformation pass in a system to process data that has become cold since the last invocation.

The database for this experiment 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 these same experiments on a table with more columns but did not observe a major difference in trends. We use a synthetic workload to prevent interference from transactions. Each transaction selects a tuple slot at random (uniform distribution) and chooses whether to insert an empty tuple to simulate deletion or to populate it with random data.

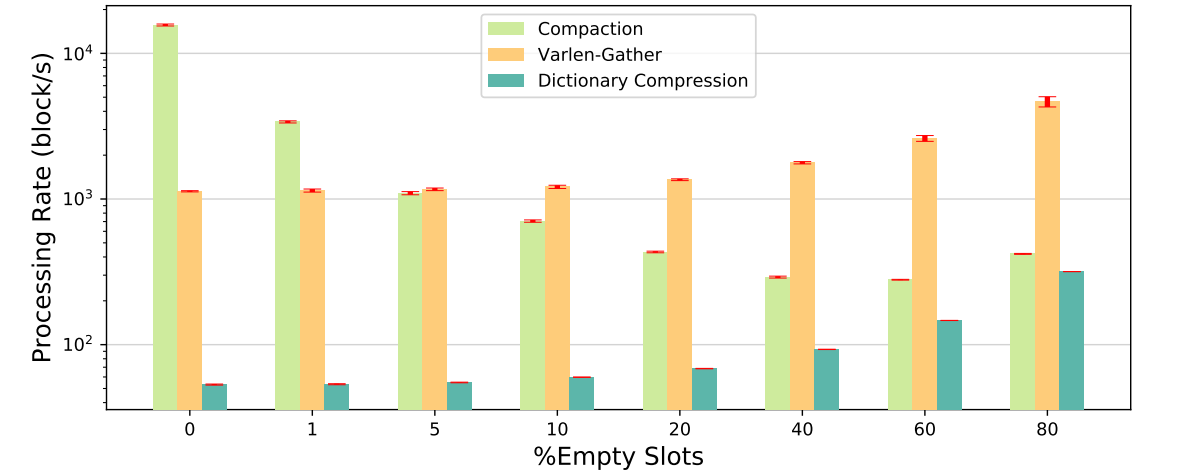
7.2.1 Throughput

Recall from 5.3 that our transformation algorithm is a hybrid two-phase implementation. For this experiment, we assume there is no thread contention and run the two phases consecutively without waiting. We benchmark both versions of our algorithm: (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 a new Arrow buffer using the Arrow API (Snapshot) and (2) perform the 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 7.3a show that our transformation algorithm with variable-length gather (Hybrid-Gather) outperforms the alternatives, achieving sub-millisecond performance when blocks are mostly full (i.e., the number of empty slots is less than 5%). The DBMS's



(a) Throughput (50% Variable-Length Columns)



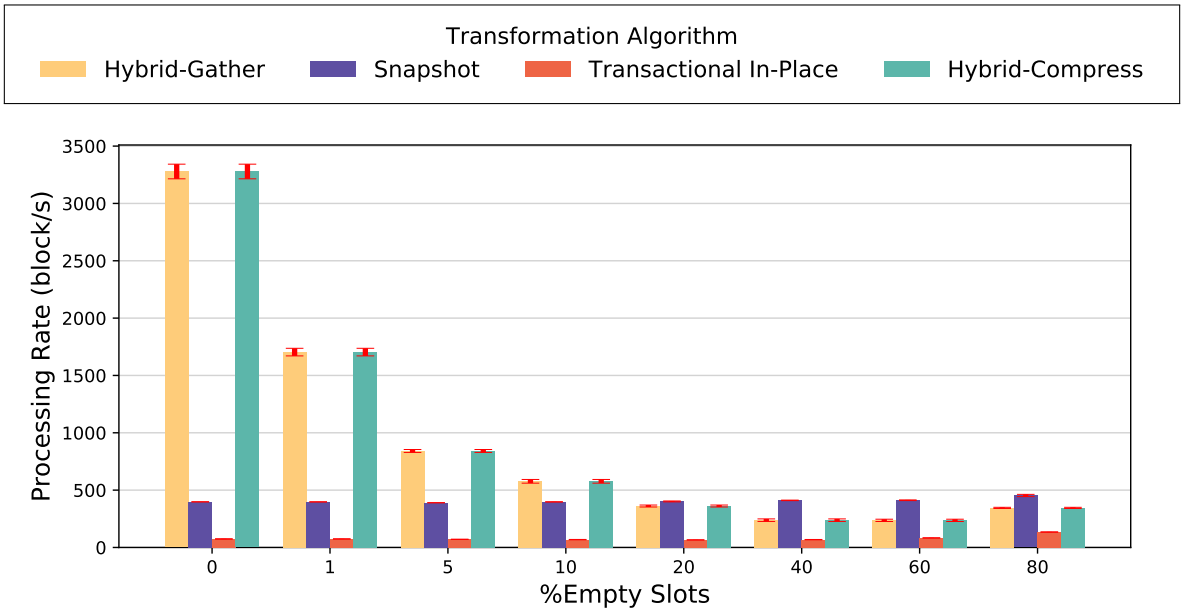
(b) Performance Breakdown (50% Variable-Length Columns)

Figure 7.3: 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.

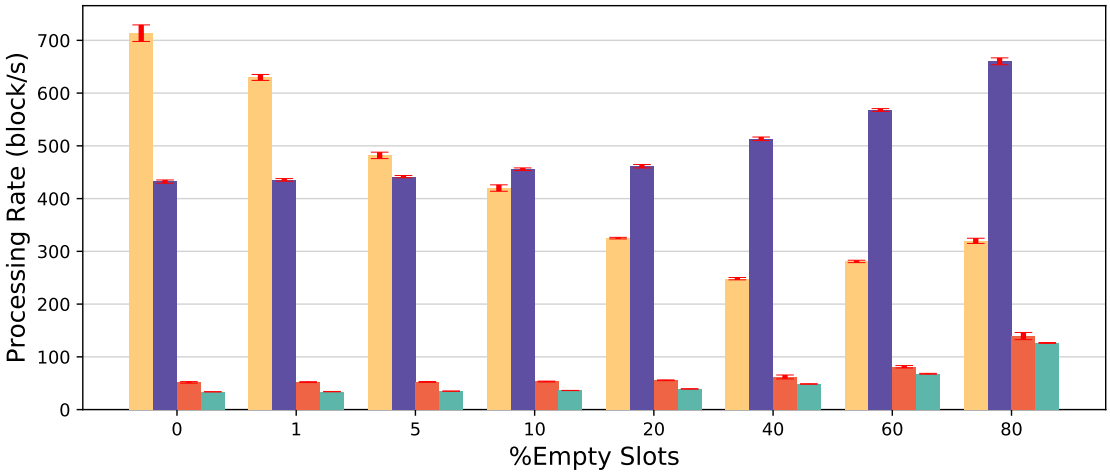
performance using approach drops, however, as this number increases. This is because more empty slots requires the DBMS to move more tuples. Such movement is random and is an order of magnitude more expensive than the sequential access pattern of the copying implementation (Snapshot). As the blocks become more than half empty in these experiments, the number of tuples that the DBMS moves decreases and thus the throughput increases. This trend is also evident for all other versions of the algorithm. The transactional approach (In-Place) performs poorly because the DBMS copies values for version maintenance and then updates each tuple's version chain one-by-one, which is an order of magnitude slower than a sequential memory copy. Hybrid-Compress is also an order of magnitude slower than Hybrid-Gather and Snapshot because building the dictionary is a computationally expensive procedure.

To understand the behavior of these algorithms better, we provide breakdown of the throughput for each phase in 7.3b. The graph is shown in log-scale due to the large range of change in performance. 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 simple bitmap scan. In this best case scenario, the cost of variable-length gather dominates. Because transactional modification is several orders of magnitude slower than raw memory access, the performance of the compaction phase drops as the number of empty slots increase, and starts to dominate the cost of Hybrid-Gather at 5% empty. Dictionary compression is always the bottleneck in the Hybrid-Compress approach.

We next measure how the column types affect the performance of the four transformation algorithms. We run the same micro-benchmark workload but make the database's columns either all fixed-length (7.4a) or variable-length (7.4b). These results show that our hybrid algorithm's performance does not change compared to 7.3a based on the data layouts. Hybrid-Compress is equivalent to Hybrid-Gather when all the columns are fixed-length because there are no variable-length data to compress. Snapshot performs better when there are more variable-length values in a block because it does not update nor copy the metadata associated with each value. Given this, we show only 50% variable-length columns in the table for other experiments.



(a) Throughput (Fixed-Length Columns)



(b) Throughput (Variable-Length Columns)

Figure 7.4: Transformation Throughput on Alternative Layouts – Measurements of the DBMS's transformation algorithm throughput and when varying the layout of the blocks being transformed.

7.2.2 Write Amplification

The throughput results in the previous experiment shows that the Snapshot approach outperforms our hybrid algorithm when blocks are around 20% empty. These measurements, however, fail to capture the overhead of the DBMS updating the index entries for any tuples that have their physical location in memory change [Wu et al. [2017]]. The effect of this write amplification depends on the type and number of indexes on the table, but the cost for each tuple movement is constant given a table. We can therefore approximate this overhead through 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 approximate and optimal algorithm presented in 5.3.

As shown in 7.5, 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. There is little difference in the result of the approximate algorithm versus the optimal algorithm. That is, the approximate approach generates almost the same physical configuration for blocks as the optimal approach. Given this, combined with the fact that the optimal algorithm requires one more scan across the blocks than the approximate one, we conclude that the marginal improvement does not justify the cost. Thus, we use the approximate algorithm for all other experiments in this thesis.

7.2.3 Sensitivity on Compaction Group Size

For the next experiment, we evaluate the effect of the compaction group size on performance. The DBMS groups blocks together when compacting and frees up any blocks that are empty after the transformation process. 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 results in the DBMS freeing more blocks, but increases the size of the write-set for compacting transactions, which increases the likelihood that they will abort due to a conflict. We use the same setup from the previous experiment, performing a single transformation pass through 500 blocks while varying group sizes to evaluate the trade-offs.

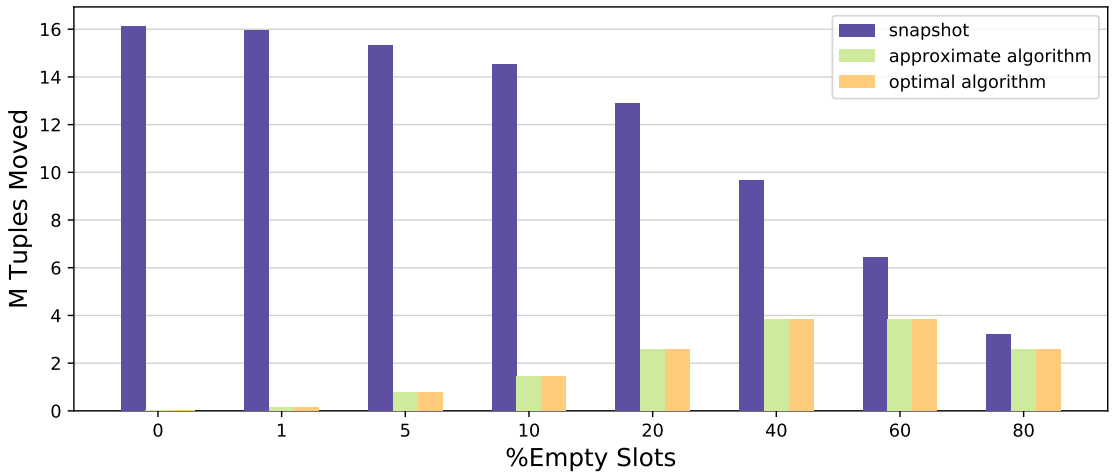
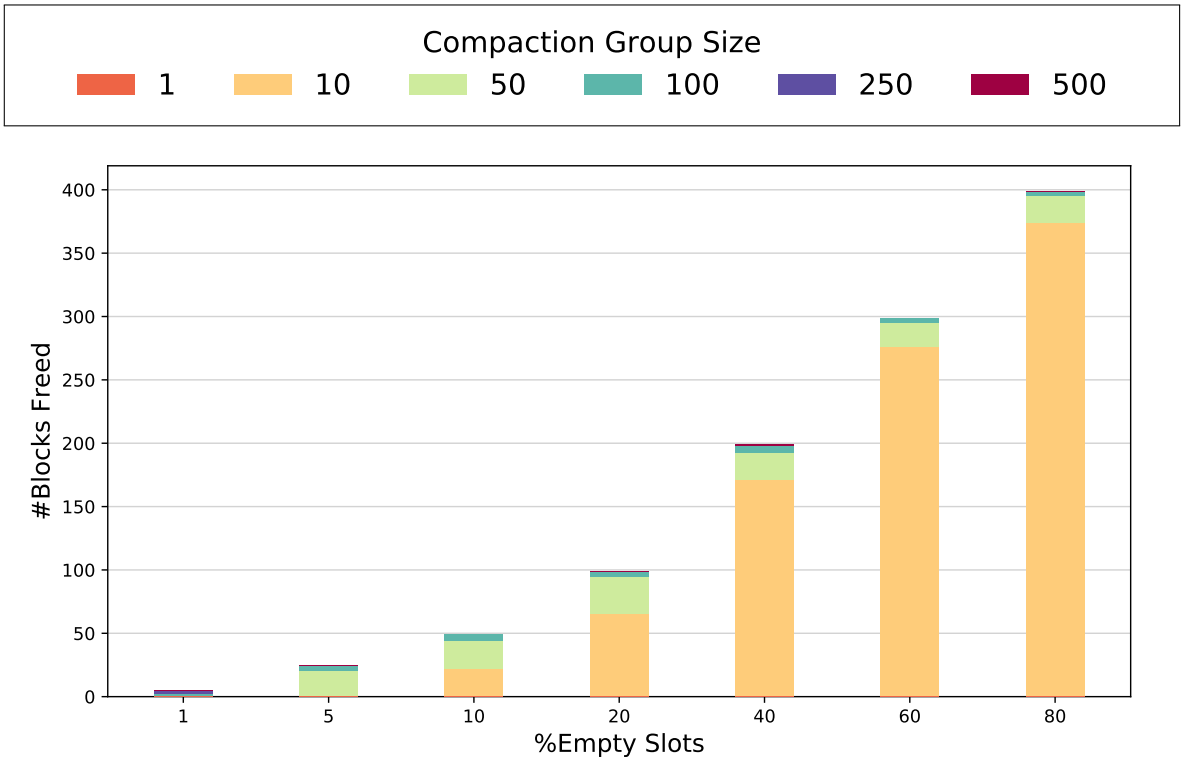


Figure 7.5: Write Amplification – Total write amplification is number of tuple movement times a constant for each table, decided by the layout and number of indexes on that table.

Numbers of blocks freed with different compaction group sizes is shown in 7.6. When blocks are only 1% empty, larger group sizes are required to free any memory. As blocks become more empty, smaller group sizes perform increasingly well, and larger values bring only marginal benefit. The cost of larger transactions is shown in 7.7 as the size of their write-sets. Size of the write-set increases almost linearly relative to compaction group size, which suggests that the total number of tuple movements decreases only slowly across possible values. For smaller compaction group sizes, the total number of movements is divided up into more transactions, leading to smaller write-sets. The best fixed group size is between 10 and 50, which balances good memory reclamation and relatively small write-sets. To achieve better performance, an intelligent policy needs to dynamically form groups of different sizes based on the blocks it is compacting.

7.3 Data Export

For this experiment, we evaluate the DBMS's ability to export data and how our Arrow-based storage approach affects this process. We compare four of the data export methods that we described in sec:export: (1) RDMA over Ethernet with Arrow storage, (2) the

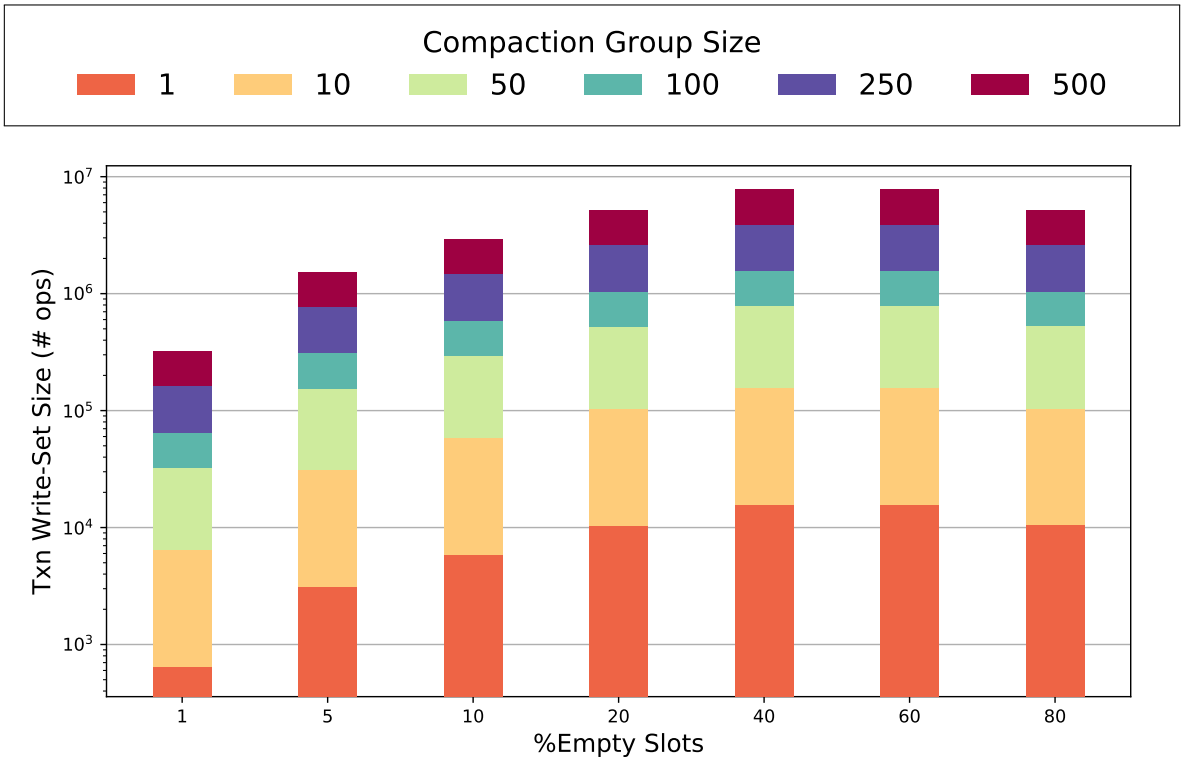


(a) Number of Blocks Freed

Figure 7.6: Sensitivity on Compaction Group Size – Efficacy measurements of the transformation algorithm when varying the number of blocks per compaction group while processing 500 blocks. The percentage of empty slots is what portion of each block is empty (i.e., does not contain a tuple). We measure the number of blocks freed during one round.

Arrow Flight RPC, (3) vectorized wire protocol from Raasveldt and Mühleisen [2017], and (4) row-based PostgreSQL wire protocol. We use two servers with eight-core Intel Xeon D-1548 CPUs, 64 GB of DRAM, and Dual-port Mellanox ConnectX-3 10 GB NIC (PCIe v3.0, 8 lanes).

We use the `ORDER_LINE` table from TPC-C as the table to export. We load the table with 6000 blocks (∼7 GB total size, including variable-length values) on the server, and use the different export methods to send the entire table over to the client side. For the DBMS wire protocols, the client then transforms the data into Arrow blocks. To simulate the interference of transactions updating the database during an export session, we force the



(a) Write-Set Size of Transactions

Figure 7.7: Sensitivity on Compaction Group Size – Efficacy measurements of the transformation algorithm when varying the number of blocks per compaction group while processing 500 blocks. The percentage of empty slots is what portion of each block is empty (i.e., does not contain a tuple). We measure the average write-set size of transactions in the transformation algorithm.

algorithm to treat some portion of the blocks in the table with the hot status. This causes the DBMS to transactionally read and materialize the blocks before transmitting them to the client.

The results in fig:export shows that our system export data orders-of-magnitude faster than the base-line implementations with Arrow Flight. When all blocks are frozen, RDMA is close to saturating the full bandwidth, and Arrow Flight saturates up to 80% of the available network bandwidth. The performance benefit of using Arrow Flight and RDMA diminishes as more blocks are hot and the DBMS has to materialize them. Interestingly, the performance of RDMA drops faster than Arrow Flight, most likely because the buffer

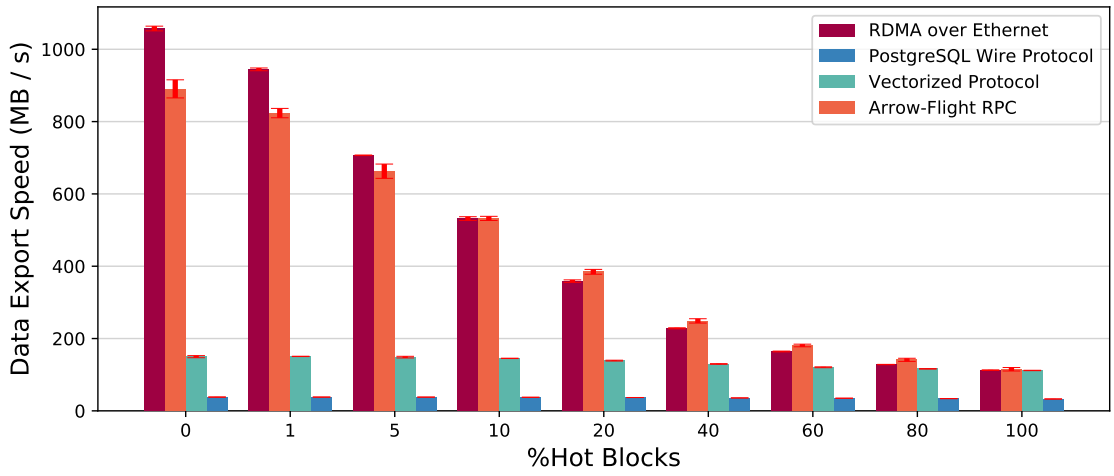


Figure 7.8: Data Export – Measurements of data export speed of CMDB using different export mechanisms, with varying percentage of blocks treated as hot.

used for materializing hot blocks is still in CPU cache at time of export, making it faster for the CPU (Arrow Flight) to access it compared to the NIC (RDMA). When the system has to materialize every block, the performance of these two methods drops to be equivalent to the vectorized wire protocol, as expected. Both the PostgreSQL wire protocol and the vectorized protocol benefit little from eliding transactions on cold, read-only data. This demonstrates that the primary bottleneck of the data export process in a DBMS is serialization and deserialization to and from a wire format, not network bandwidth. 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 serialization cost and improves data export performance. In a modern setting, the DBMS wire protocol should prioritize elimination of serialization/deserialization step over compression.

Chapter 8

Conclusion and Future Work

We presented CMDB's Arrow-native storage architecture for in-memory OLTP workloads. The system implements a multi-versioned, delta-store transactional engine directly on top of Arrow. 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-magnitudes faster data export compared to current approaches.

We implemented three out of the five proposed methods of utilizing native Arrow storage. The remaining two, server-side RDMA and external tool execution on DBMS requires major changes to the existing DBMS architecture. On the other hand, these two methods will blur the application/DBMS boundary and unlock new ways of assembling data processing pipelines in a cloud setting. We plan to extend our work to explore and implement these two methods in the future.

Bibliography

Apache Arrow. https://arrow.apache.org/, a. 1.1

Apache Arrow Source Code. https://github.com/apache/arrow, b. 6.2

Apache ORC. https://orc.apache.org/, c. 1.1, 3.1

Carnegie Mellon's New DBMS. https://github.com/cmu-db/terrier. 1.2, 7

Databricks delta lake. 3.1

Dremio. https://docs.dremio.com/. 3.1

Apache Hive. https://hive.apache.org/. 3.1

Apache Impala. https://hive.apache.org/. 3.1

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##page=258. 5.3.2

Apache Kudu. https://kudu.apache.org/overview.html, a. 3.1

Transaction semantics in apache kudu. https://kudu.apache.org/docs/transaction_semantics.html, b. 3.1

Microsoft analytics platform system. 3.3

olap4j: Open Java API for OLAP. http://www.olap4j.org. 3.3

Omnisci gpu-accelerated analytics. https://www.omnisci.com/. 3.1

Apache Parquet. https://parquet.apache.org/. 1.1

Ibm db2 pure scale. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_10.5.0/com.ibm.db2.luw.licensing.doc/doc/c0057442.html. 3.3

Oracle real application cluster. 3.3

Microsoft sql server with smb direct. 3.3

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980, 2008. 2.1

Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, November 2002. ISSN 1066-8888. 3.2, 4.2

Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2o: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1103–1114, 2014. ISBN 978-1-4503-2376-5. 2.3

Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 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, pages 583–598, 2016. 2.3, 3.2

Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983. ISSN 0362-5915. doi: 10.1145/319996.319998. URL http://doi.acm.org/10.1145/319996.319998. 4.1

Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, March 2016. ISSN 2150-8097. 3.3

Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005. 2.1

Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 275–290, 2018. ISBN 978-1-4503-4703-7. 4.3, 5.4.2

David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8. doi: 10.1145/602259.602261. URL http://doi.acm.org/10.1145/602259.602261. 4.4

Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, 2015. ISBN 978-1-4503-3834-9. 3.3

Florian Funke, Alfons Kemper, and Thomas Neumann. Compacting transactional data in hybrid oltp & olap databases. *Proc. VLDB Endow.*, 5(11):1424–1435, July 2012. ISSN 2150-8097. 3.2, 5.2

Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. Hyrise: A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4 (2):105–116, November 2010. ISSN 2150-8097. 3.2

Allison L. Holloway and David J. DeWitt. Read-optimized databases, in depth. *Proc. VLDB Endow.*, 1(1):502–513, August 2008. ISSN 2150-8097. 5.4.1

Alfons Kemper and Thomas Neumann. 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, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767867. URL http://dx.doi.org/10.1109/ICDE.2011.5767867. 3.2

Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11:2209–2222, September 2018. 2.1

Per-Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011. ISSN 2150-8097. doi: 10.14778/2095686.2095689. URL http://dx.doi.org/10.14778/2095686.2095689. 4.3

Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. Hybrid garbage collection for multi-version concurrency control in sap hana. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1307–1318, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2903734. URL http://doi.acm.org/10.1145/2882903.2903734. 4.3

Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 355–370, 2016. 3.3

Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, September 2017. 2.1

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

ISSN 0362-5915. doi: 10.1145/128765.128770. URL http://doi.acm.org/10.1145/128765.128770. 4.4

Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 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, pages 677–689, 2015. 1.1, 2.2, 3.2, 4.1, 4.1

Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017. URL https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf. 5.2

Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. https://www.gartner.com/doc/2657815/, 2014. 1.1

Mark Raasveldt and Hannes Mühleisen. Don't hold my data hostage: A case for client protocol redesign. *Proc. VLDB Endow.*, 10(10):1022–1033, June 2017. ISSN 2150-8097. 2.1, 3.3, 6.1, 7.3

Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1539–1551, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2915966. URL http://doi.acm.org/10.1145/2882903.2915966. 4.4

Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-store: A real-time OLTP and OLAP system. In *Extending Database Technology*, pages 540–551, 2018. 3.2

Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD*, pages 731–742, 2012. 2.2, 3.2

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 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*, pages 1150–1160, 2007. 5.2

The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, June 2007. 7.1

Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522713. URL `http://doi.acm.org/10.1145/2517349.2522713`. 4.3

Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 ACM International Conference on Management of Data*, SIGMOD '18, pages 473–488, 2018. 7.1

Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, March 2017. 1.1, 7.2.2

Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014. ISSN 2150-8097. doi: 10.14778/2735508.2735511. URL `http://dx.doi.org/10.14778/2735508.2735511`. 4.3