Freely Moving Between the OLTP and OLAP Worlds: Hermes – an High-Performance OLAP Accelerator for MySQL

Tim Gubner
Huawei Cloud
tim.gubner@huawei.com

Rune Humborstad
Huawei Cloud
rune.humborstad@huawei.com

Manyi Lu
Huawei Cloud
manyi.lu@huawei.com

ABSTRACT

Users often want to run analytics on their OLTP databases, to avoid costly and cumbersome Extract-Transform-Load (ETL) processes. Typically, analytical queries run rather slow on OLTP DBMS, making Hybrid Transaction/Analytic Processing (HTAP) solutions popular. One possible solution is to add an accelerator (for analytics) to the already existing OLTP DBMS.

Typically, analytical systems, especially for the cloud, focus on extremely large datasets ("exa-scale") and distributed query execution (across multiple machines). We argue that many customers do not have large enough datasets to justify expensive multi-node DBMSs. Compared to single-node systems, such multi-node systems typically come with a baseline drop in performance (but might scale), as they need to introduce data transfers across the network.

For this reason, we propose Hermes as cloud-native, but single-node, accelerator for MySQL. Hermes speeds up analytical queries by, often 2-3, orders of magnitude and outperforms competing systems by up to 5× (including multi-node systems). We achieve this by keeping Hermes relatively lean and focusing on the core features required. In the paper we describe Hermes' architecture, data storage and integration with MySQL as well as Hermes' query engine. Importantly, Hermes provides the highest degree of data freshness. If data is not replicated yet, Hermes waits. The waiting times, however, are practically negligible (single digit vs. three digit milliseconds).

We evaluate Hermes on TPC-H as well as micro-benchmarks. Besides the aforementioned improvements, our replication mechanisms achieved high and stable throughput rates of up to 60k changes per second, leading to low waiting times.

In summary, Hermes is a lean accelerator for MySQL. Its single-node design keeps costs for users low and performance high. Additionally, Hermes guaranteeing data freshness and compatibility with MySQL (both, Hermes and MySQL, return the same result).

PVLDB Reference Format:
Tim Gubner, Rune Humborstad, and Manyi Lu. Freely Moving Between the OLTP and OLAP Worlds:
Hermes – an High-Performance OLAP Accelerator for MySQL. PVLDB, 18(12): 5113 - 5125, 2025.
doi:10.14778/3750601.3750631

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750631

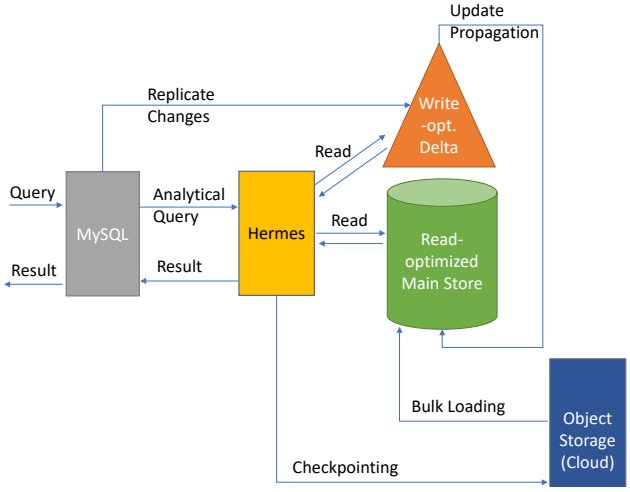


Figure 1: Hermes overview: Hermes acts as accelerator for analytical workloads. Changes from MySQL are continuously replicated to Hermes, which allows to answer queries on the most recent state of the database. Our data resides in main-memory, therefore we periodically checkpoint data to Object Storage. In case of node failure, we could replay changes from MySQL, but directly restoring the Main store from checkpoint is obviously significantly faster. (In this paper, we only refer to the main-memory-only setup. Hermes allows other setups as well. Most notably data can reside on Object Storage but cached in main-memory.)

1 INTRODUCTION

Customers often use OLTP database offerings in the cloud. However, frequently they also want to run analytics on the same dataset. Commonly, the big obstacle is that moving data from transactional to analytical (OLAP) systems, so called Extract-Transform-Load (ETL) is rather time-consuming as well as it is tricky to get 100% correct and performant.

Hybrid Transactional/Analytical Processing (HTAP). From the user perspective, it understandable that they often choose a system that allows both, fast transactions and fast analytical queries, a so called Hybrid Transactional/Analytical Processing (HTAP) system. There is a wide range of many different architectures for HTAP systems, from single-node to multi-node or systems that grew from different starting points (OLTP system extended to HTAP, HTAP from scratch, or OLAP with optimizations for OLTP).

The Case against Complex Multi-Node Distributed Systems. Recent decades have seen the creation of more and more complex DBMS sold as the "scalable" silver bullet for every use-case

from the tiny "mum and pop" shops to extremely large "exa-scale" workloads (e.g. Telecom data processing or the biggest web shops on this planet). These DBMS are often optimized for large multi-node clusters to deal with the enormous amount of data. A few of this optimizations include fancy partitioning and replication schemes, exploiting bleeding edge network technologies, as these systems often transfer lots of data across the network.

However, frequently customer workloads are, neither extremely large, nor do they require clusters with thousands of CPU cores and petabytes of memory. If we, for example, look at the PublicBI workload [5, 22], a realistic workload, many tables are not extremely large. In PublicBI most tables contain $\leq$ 1 M rows, only one tables few contain more than 100M rows [22]. However, queries are frequently rather complex and, often, process string data [22]. Similarly, in Snowset [21, 23], a realistic dataset released by Snowflake, most queries read less than 1 TB.

Distributed multi-node systems are typically *much* more complex than single-node DBMS, as they require complex transaction handling, smart data partitioning and failover[1] techniques etc. Moreover, multi-node systems usually come with a drop in performance for workloads that are not extremely-large, i.e. most customers.

Hermes – An Overview. In this paper, we present Hermes, a cloud-native accelerator for analytical workloads on MySQL. Hermes continuously replicates the changes coming for MySQL while allowing fast analytical queries. The resulting system, Hermes, improves runtimes on analytical workloads by, typically, 2-3 orders of magnitude on a *single node* and outperforms competing multi-node systems (optimized for analytical workloads) by up to 4.6×.

Contributions. Our contributions are the following:

- We present an architecture for accelerating OLAP queries for OLTP databases, MySQL in particular.
- We describe how our storage is optimized for both, fast data modifications (insert/update/delete) as well as analytical queries.
- We reveal our novel lightweight mechanism to efficiently merge Delta and Main store on-the-fly.
- We explain how to integrate an accelerator with MySQL. We focus on how data and transactions can be replicated, how snapshots consistent with MySQL can be accessed and MySQL's query semantics can be kept.

Structure. The remaining paper is structured as follows: The following section discusses necessary background as well as related work. Afterwards, we introduce Hermes from the "500ft view", followed by an explanation of how data is stored in Hermes, how data is migrated from one storage level to another, how the stored data is modified and how old data is cleaned up. Then, we reveal how we integrated Hermes with MySQL and solved some of the MySQL's semantic quirky-ness in Hermes, followed by a description of Hermes' query engine. Afterwards, we evaluate our Hermes implementation, followed by a brief summary of the "takeaways" and discussion of future work.

[1]Note that the more nodes a system has the more likely it is to fail, making failure handling and failover techniques more important for multi-node systems.

2 BACKGROUND & RELATED WORK

In this section, we describe necessary background as well as related worked. We start by giving a brief glimpse into different architectures for Cloud DBMS, followed by an introduction into different techniques for parallelizing queries on multiple nodes. Afterwards, we explain different systems and compare them to Hermes (this work), first with single-node OLAP systems followed by Hybrid Transactional/Analytical Processing (HTAP) systems.

2.1 Architectures for Cloud DBMS

Shared Nothing. Shared Nothing architectures partition data across all nodes. Each node is responsible for its partition. As long as each node stays within its own partition data access is cheap (local). However, network transfers occur when accesses span multiple partitions. In the cloud, Shared Nothing architectures are challenging, because Compute (nodes for transactions and queries) and Storage (nodes storing data) cannot be scaled independently and scaling will typically require costly re-partitioning of data.

Shared Storage. Shared Storage/Disk architectures rely one global access to storage. For cloud systems, this allows scaling storage without having the re-partition all data.

However, assuming the absence of a cache, each read/write would lead to network transfers. Writes can possibly lead to multiple transfers when replication is used (i.e. one per replica). For reads, the number of network transfers can be reduced by introducing a cache. But, a cache will lead to additional overhead for writes, i.e. cache invalidation and cache coherency.

Single Node. Whereas Shared Nothing and Shared Storage architectures, lead to costly network transfers, if data was stored on a single node, there will be no additional access cost. This works as long as all data fits into a single node. Importantly, with current hardware a significant amount of data – *terabytes* – can fit into a single node. We argue that many use cases can be handled by a single node (Hermes) and, thus, does not incur costly network transfers or re-partitioning costs.

2.2 Intra-Query Parallelism & Distributed Query Execution

Distributed query execution is typically built around Volcano model parallelism [8], i.e. they introduce special operators (*Exchange*) to parallelize pipelines. One major advantage is that operators can (mostly) stay unaware of parallelism. The parallelism model allows range and hash partitioning as well as broadcasting (to all partitions), initial partitioning as well as repartitioning (from e.g. N hash partitions into M range partitions).

In general, Volcano-model parallelism works reasonably well on a single machine [12]. On larger – many-core – systems, Volcano-model parallelism tends to underperform against Morsel-driven parallelism [16], which assumes a shared memory system and operators are aware of parallelism. Both use the same underlying building blocks which are hash and range partitioning to redistribute data across partitions [2].

[2]Morsel-driven parallelism also allows accessing shared data in e.g. in the probe phase of a hash join all threads access the same hash table. Volcano-model parallelism allows to broadcast data to all partitions.

Distributed Execution re-partitions frequently. For distributed systems, repartitioning is a costly operation as it requires data movement across the network. Worse, repartitioning has to occur rather frequently as well.

For example, if we join two tables (A, B) on the columns $A.c1 = B.c1$ and group by $c2$. This requires partitioning twice: Once to execute a distributed hash join (hash partition both sides on $c1$) and a second time to group (hash partition on grouping key $c2$) [3]. This essentially requires us to transfer the larger table across the network *twice*. Note that we can easily extend this example by adding more joins on other columns. Unfortunately, analytical queries often contain multiple with joins and group-bys and, therefore, costly repartitioning we can occur rather often.

Skew. In our example it can also happen that $c2$ ends up with only fewer distinct values than the number of partitions. Consequently, data will be distributed unevenly across partitions [4]. In the extreme case, $c2$ would only contain one value. On a cluster with 128 cores, we could end up using only 1 core. While on a single node, we can load-balance skew [9, 16], on a cluster load-balancing across nodes will require additional data movement.

Takeaway. To summarize, distributed query execution, especially on analytical queries, frequently re-partitions data. The consequence is:

Repeated re-partitioning precludes a cost and power efficient system.

2.3 Single Node Analytical DBMS

Hermes resembles a single node analytical system with optimizations for fast data modifications. Here, we compare to single node systems optimized for analytical workloads:

DuckDB [7] and Vector/Vectorwise [26] are two systems using Vectorized Execution [3] as well as support columnar data storage.

Similarly to both systems, Hermes also uses Vectorize Execution, but the major differences are in Hermes' data storage.

DuckDB adapted Hyper's MVCC mechanism, as described by Neumann et al. [18], to handle data modifications. Like Hermes, DuckDB stores data in RowGroups. However, in DuckDB data modifications also appear in the same RowGroups forcing them to always contain versioning information (for each row). Hermes' RowGroups only contain the data and a creation timestamp (of the RowGroup) and modifications end up in the write-optimized Delta.

Vector/Vectorwise uses a positional delta (positional delta tree, PDT) on top of its compressed columnar data store [11]. Changes from the PDT are merged periodically into Vectorwise's compressed "main" store. During scans, the PDT(s) are merged on-the-fly, by inserting/updating/deleting the decompressed vectors (cache-sized columnar chunks) using the delta. This merge-style operation is relatively cheap, because both stores are ordered by a row id. Hermes merges are even cheaper. We filtered updated rows from main store and just append the Delta. There is no need to expand the decompressed vectors (e.g. when a row is added in the middle). The obvious disadvantage of our approach is that the result is unordered. Also data modifications in Hermes, do not require traversing a tree (PDT). They either append at the end (`INSERT`) or only require finding the physical row (lookup in *RowIdMap*) and modifying that (`UPDATE`, `DELETE`).

[3] In practice, one would try to store A and B already partitioned on $c1$ to avoid one partitioning, but that is not always feasible (there could be similar joins on other columns as well)

[4] That can also happen after hashing due to a bad hash function or as a result of the birthday paradox that occurs during partitioning, or because processing one partition happens to run slower/faster than others.

2.4 Hybrid Transactional/Analytical Processing (HTAP) DBMS

The resulting combination of MySQL with Hermes as an accelerator for analytical workloads acts as a Hybrid Transactional/Analytical Processing (HTAP) system. Therefore, we focus on HTAP.

PolarDB-IMCI. PolarDB is a famous DBMS available in the Alibaba cloud and quite possibly Alibaba's top database product. With PolarDB-IMCI [24], PolarDB added an accelerator for analytical processing to their transactional database. To enable IMCI, one needs to mark the specific set of columns as `KEY_COLUMN_INDEX` [24]. Compared to IMCI, we can offload transparently, by setting Hermes as secondary engine, without requiring schema changes.

To replicate data PolarDB uses their REDO log [24]. We, instead, use the binary log (*binlog*). The data in PolarDB is organized in insertion ordered whereas Hermes organizes data either unordered (Delta) or ordered by row id (Main store). In addition PolarDB maps primary keys to row ids via an LSM tree [24] requiring $O(\log N)$ steps to find a row id. In Hermes, we use the row ids from MySQL. It can happen that in the Delta we need to lookup the physical location of a row, but this is a hash look and requires constant time ($O(1)$).

Similar to Hermes, PolarDB uses an engine with Vectorized Execution [3] and Morsel-driven parallelism [16].

On TPC-H, PolarDB underperforms Hermes. Note that PolarDB uses multiple machines, whereas Hermes uses a single machine.

ByteHTAP. A system that shares some similiarities with Hermes is ByteHTAP [4]. ByteHTAP combines an OLTP engine with an dedicated engine for OLAP. Similar to Hermes, ByteHTAP uses log-based replication to keep the OLAP data fresh. Also both, Hermes and ByteHTAP, split the data storage into Main store and Delta and allow merging both online (during the scan) as well as offline (asynchronously, *Update Propagation*).

However, there are notable differences: When merging Delta and Main during the scan (on-the-fly), ByteHTAP has two choices (a) it can for every row in the Main store check whether that row has been deleted (or updated), leading to one hash lookup for every row, or (b) try collect all changed rows from the Delta(s), compute a selection vector from rows from the Main surviving and only fetch these from the Main store. Tactic (a) is suited for small amounts of changes, whereas (b) handles lots of changes. Our merging strategy is similar to (a) but more lightweight: We only require a merge-syle algorithm based on row ids (integers), no hash lookup on the primary key (no hashing cost, no hash conflicts, no random memory lookups). Hermes does not support a similar tactic (b), because we assume that $< 10\%$ of the table has changes and, thus, rather eagerly flush data to the intermediate Deltas (which ByteHTAP does not have). Furthermore, it seems that ByteHTAP requires users to specify partitioning keys to enable intra-query parallelism across multiple machines and relies on Flink's parallelism model within a machine. Hermes features load-balanced Morsel-driven parallelism, but only utilizes a single machine.

Other Systems. Another well-known system is SAP HANA [20]: Like Hermes it contains multiple storage layers (both have 3), from write- to read-optimized. HANA, exploits dictionary-encoding for every value using an ordered global dictionary (per column). While this improves compression ratios, in some cases, this can lead to significant re-encoding effort, essentially forcing a complete rewrite of a compressed column. Our compression is limited to RowGroups, whenever adversarial values are inserted, they only affect the current RowGroup and not the complete Main store.

Oracle Database In-Memory [13] supports both row and column store. Users are required to specify whether to offload a specific table/schema/... to the column store (via keyword `INMEMORY`). We transparently offload data into Hermes. Like In-Memory we can populate Hermes with data without any downtime, furthermore does our column store involve compression. A major difference is that Hermes does not require analyzing a transaction journal for accessing data, instead we utilize our Delta store that allows us to relatively efficient read certain versions. Furthermore, In-Memory does not seem to be able to handle the corner case, where uniformly distributed changes would cause the "population" process (moving data from Delta to compressed column store) to essentially re-create all RowGroups. Specially this corner case, forced us to have three storage layers (row-based Delta, columnar Delta, Main store).

SQLServer Column Store Indexes [14, 15] stores data in columnar and compressed fashion (like Hermes). A compressed columnar index in SQLServer can also be updated via a Delta store and a bitmap for deleted rows [14]. Like Hermes, updated or inserted rows will enter the Delta store. A major difference is that the Delta store in SQLServer uses a different index structure (B-tree vs. Hash table). Additionally, deletes end up in a Delete bitmap in SQLServer's Main store. Like in Hermes, scans in SQLServer have to ignore changed data. However, in Hermes it is a cheap merge-based operation on a dense array of integers while in SQLServer it is a lookup into a sparse Delete bitmap. Updates in SQLServer become Deletes and Inserts, which is probablatic when some rows are updated very frequently. In SQLServer, this would require eleminating many inserts (that stem from updates) during scans. Hermes on the other side, will always have the newest version either in the Main store, or directly in the Delta (at head of the version chain).

3 HERMES

Often, regular customers, do not require solutions for fast analytics on petabytes of data. Since modern servers can easily stored terabytes of data (most of that often into main memory), we believe a single-node system is not only faster but also more cost-effective in most cases. We can skip architectural complications introduced by multi-node systems as well as avoid costly network transfers, which for distributed systems happen very frequently (see Section 2.2). Therefore, we chose to design Hermes as a single-node cloud-native OLAP system, which is used to enable fast analytics on MySQL.

In the following, we describe high-level architecture of Hermes shown in Figure 1 from multiple perspectives. Here, we focus on an overview, more detailed explanations of specific components can be found in dedicated sections. We first explain the path a query takes through the system, then briefly describe how changes from MySQL reach Hermes. Afterwards, we describe how checkpointing works in Hermes.

Query. The user issues a SQL query to MySQL. MySQL parses, analyzes and optimizes the query. If the cost of the query exceeds a certain threshold, we decide to execute the query on Hermes (analytical queries are typically more costly, analyze more data and have complex query plans). Hermes evaluates the query and fetches data from the respective data store. We try to fit most data into the Main store, which is optimized for large analytical reads. If the data contains recent changes, not yet integrated into the Main store, Hermes also needs to read the Delta store. Once Hermes completed the query it returns the result back to MySQL, which then returns the result to the user.

Replication. To allow operating on the most recent data, we continuously replicate all changes from MySQL to Hermes. Technically, we subscribe to the binary log stream. Whenever an event arrives, we analyze it and transform the event into an operation on the Delta store. We periodically clean up the Delta store by propagating table updates to the Main store (*Update Propagation*). Before applying the log stream, we first try to ingest the current (committed) state of the table. We fetch chunks of the table from MySQL and transform them and directly insert them into the Main store (instead of the Delta).

Checkpoint & Restore. While it is possible to fully replicate the state of MySQL by re-ingesting and re-applying the log stream, it is extremely costly. Therefore, we decided to periodically checkpoint the Main store and write the changes to Object Storage. Checkpointing the Main store has certain benefits: (1) The Main store is compressed. As a consequence, less data needs to be backed up. (2) The Main store is only updated, during *Update Propagation* or *Garbage Collection*. thus minimizing contention on latches (checkpoint will take read latches and do not conflict with data modifications or read transactions).

To checkpoint a table, we backup only its RowGroups, the horizontal partitions of the Main store of a table. For each RowGroup, we maintain a version timestamp. If the timestamp is newer (higher) than the last checkpoint, the RowGroup got modified in between and we need to back it up again. Note that this can only happen when Update Propagation applies changes from the Delta, which is unlikely to modify a lot of RowGroups.

Upon restore, we load checkpoint in bulk into Main store.

3.1 Node Failure

In case the node hosting Hermes fails or Hermes crashes, queries and transactions on MySQL can just continue without Hermes. When the node returns, Hermes will reconnect to MySQL and replicate the state from MySQL, either from scratch or from the last checkpoint.

Interestingly, one could connect multiple Hermes instances to MySQL to either allow a transparent failover (in case Hermes fails) or to parallelize the incoming analytical workload across multiple nodes (inter-query parallelism).

4 DATA STORAGE

One of the major challenges in the development of Hermes has been to combine fast insert/update/delete (modification) with fast

analytics. Typically both, fast modifications and fast analytics, lead to contradicting requirements.

For analytics, columnar data storage combined with lightweight compression seems paramount. Unfortunately, in compressed column stores it is very hard to modify rows. For example, updating a single row, supposed we already know its position, would require decompressing, updating the values (in a buffer) and recompressing, with the hope that the original compression scheme is not affected too much (data properties might drift away from chosen compression scheme).

On the other hand, data layouts optimized for fast modifications are sub-optimal for analytics. For example, a reading only one column essentially requires scanning the whole table.

We chose the, arguably, typical answer to this question: We use a columnar and compressed data store for analytics (*Main store*) and an uncompressed row-oriented delta (*Delta store*) for modifications (later in the paper, we introduce the Columnar Delta as an intermediate step row-based Delta and Main store).

First, we explain the Main store, where the lion's shared of data is stored in a compressed and read-optimized fashion. Afterwards, we describe the row-oriented Delta store. Then, we present how inserts, updates, deletes and scans work on the combined data store. Afterwards, we justify the need for multiple levels of deltas (for us 2, because uniform data modifications would lead to changes in many RowGroups), followed by a description of how we move data from the Delta store(s) to the Main store (Update Propagation). Last, but not least, we disclose how Hermes' garbage collection works.

4.1 Main Store

The Main store is optimized for analytical workloads. Data is partitioned horizontally into RowGroups. RowGroups are split, further, into PaxGroups, PaxGroups resemble PAX [1] data layout on a subset of columns. Within PaxGroups, we store data in a columnar fashion and compressed. This is illustrated in Figure 2.

Allocation Unit (AU). We allocate physical memory for each PaxGroup (Allocation Unit, AU). While a RowGroup *can* consist of multiple PaxGroups, we try to minimize the number PaxGroups and only use multiple PaxGroups when rows are extremely wide.

Physically, this limits the size of a PaxGroup to 1 GiB (self-imposed limit). Only if it is impossible to fit 16K rows into an Allocation Unit, we consider splitting into multiple PaxGroups.

Compressed Columnar Storage. Within a RowGroup, we store data in a columnar fashion. To reduce storage footprint, we choose to compress each column. Hermes supports lightweight compression schemes similar to Heman et al.[25]. For integers and decimals, we support: Frame-of-reference and Dictionary compression with exceptions. For strings, we only support Dictionary compression. The best compression scheme is determined automatically when a RowGroup is created.

To guarantee top-notch scan performance, we implemented the compression and decompression routines using SIMD (AVX-512).

Statistics. Within a RowGroup, we keep fine-grained statistics per column. It is typically more efficient to store statistics not physically within the RowGroup, but rather out-of-band because, then, accessing does not require reading the actual data. We chose to keep these statistics in our meta data. Most notably, we keep the minimum and maximum (aka Zonemaps), to prune not qualifying RowGroups early, but also keep statistics for query optimization.

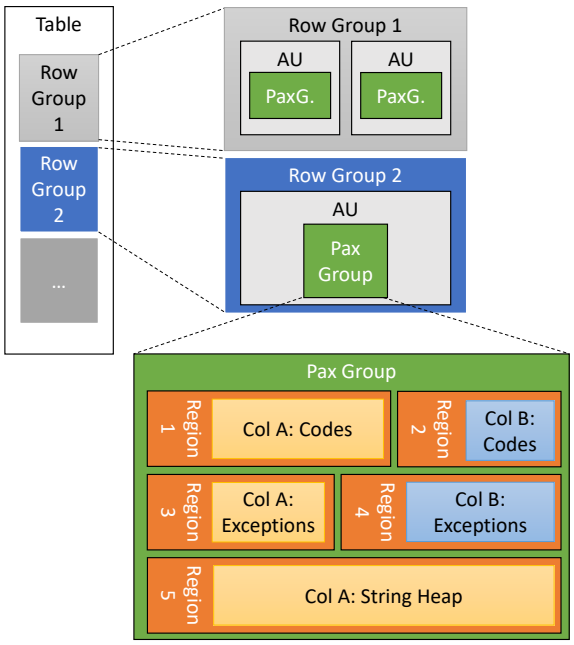


Figure 2: The Main store is optimized for analytical queries. In the Main store, data is partitioned into RowGroups. RowGroups consists of one or more PaxGroups. PaxGroups are allocated within physical storage (Allocation Unit, or AU). Within PaxGroups, data is stored in columnar format and compressed. The data within a PaxGroup consists of multiple regions. Each region can serve different purpose: e.g. store compressed codes, exceptions, dictionaries (not shown) or a heap for variable-length data.

Access. During scans, we only fetch the data regions we are interested in. This allows avoid fetching regions and columns that are not needed. In other words, we use RowGroups and PaxGroups only as logical units of storage.

4.2 Row-Based Delta Store

Our Delta store is optimized for fast modifications. Each table can have multiple deltas (originating from multiple Update Propagations). Each Delta is basically a row-wise table with payload columns (*Col A* and *Col B*) and additional columns to facilitate modifications and version reconstruction (*Delete* and *Version Vector*, which resembles the Version Vector from Hyper's MVCC implementation [18]).

As illustrated in Figure 3, the Delta is split into multiple regions: *RowIdMap*, *RowSpace*, *UndoSpace*, *DeleteLog* (not shown) and *String Heap* (not shown):

- *RowIdMap*: Unfortunately, Hermes does not control the row ids coming from MySQL[5]. Consequentially, we have to map row ids to physical locations (i.e. pointers). The *RowIdMap* is an append-only hash map from row id to offset into the *RowSpace*.

[5]Otherwise (if one could control the row ids), we could encode the physical location, thus, making each lookup more efficient.

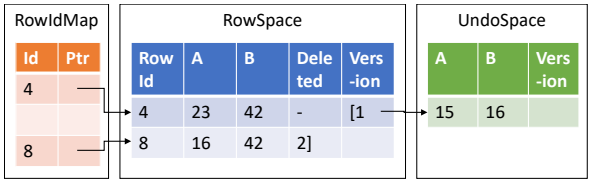


Figure 3: Each Delta consists of a *RowIdMap* that maps row ids to offsets into the *RowSpace*. The *RowSpace* stores newest version of the table plus Version Vector and row id. Sometimes rows in the *RowSpace* have older versions, in that case we have the newest version pointing into the *UndoSpace*.

- *RowSpace*: The *RowSpace* hosts the table. The pointer from the Version Vector points to the *UndoSpace*.
- *UndoSpace*: The *UndoSpace* is a heap of undo images that we use to reconstruct older versions.
- *DeleteLog*: It can happen, that we Update or Delete a row that is not in the Delta. For this case, we keep a list of row ids together with the respective timestamps of the change (current transaction id). During commit, we update the timestamp to the commit id.
- *String Heap*: Sometimes we have to store variable-length data. Such data is stored in the *String Heap*.

Example. In our example in Figure 3. The Delta contains 2 rows (row ids 4 and 8). Row 4 has been updated at timestamp 1. It changed from $A = 15$ and $B = 16$ to $A = 23$ and $B = 42$. Row 8 has been deleted at timestamp 2.

4.3 Operations on Delta & Main Store

We now describe how data modification and scans are performed on Main and Delta store.

Current Delta. In the Delta store, we can have multiple Deltas for the same table. They form a linked list with the most recent Delta at the top/head of the list. Within a transaction we always choose the most recent Delta. If Update Propagation would create another Delta more recent – the new head of the list – the transaction will still write to the older one (old head).

Data Modification. Data modification basically just modifies the most recent Delta. Before that operation, we lock a latch on the row (hash row ids and compute index in latch table and lock latch).

We now distinguish between insert, update and delete:

Inserts are just appends to the end of the *RowSpace*, potentially causing it to grow.

For *Updates*, we utilize the ideas from Hyper's MVCC implementation [18]. We maintain a *Version Vector* that allows us to re-construct older versions. When updating a row, we first create a copy in the *UndoSpace*, append it to the Version Vector (add link to chain and modify the timestamp to the transaction id) and, afterwards, update the row in-place. Concurrent transactions will not be able to see our update, but can reconstruct their version from the Version Vector. If the row does not exist in the Delta, we append the row id together with the current transaction id to the *DeleteLog*.

For *Deletes*, we maintain an extra column which marks the death time of the row as timestamp in the *RowSpace*. Consequentially, deleted rows become invisible to newer transactions. If such a row does not exist, we append the row id with the current transaction id to the *DeleteLog*. If the death timestamp has not been 0, there must have been a write-write conflict and we throw an error and abort the transaction.

Scans. Scanning multiple data stores, i.e. Main and Delta store, have one major challenge: We need to efficiently combine data from both. Unlike other approaches, we do not need to iterate through an ordered Delta structure (Positional Delta Tree [11], and merge depending on the row id) and, neither, do we require expensive key lookups (into the Delta, i.e. check whether current row from Main store is in Delta).

Instead, we chose to simplify the problem into Filtering and Appending. Both operations are rather lightweight (no random-access lookups) and typically touches only a subset of the rows from the Main store.

The basic idea is to first emit rows from the Main store, filter changed rows out, and, afterwards, emit the rows from the Delta. In detail, scans operate as follows:

(1) We read the all Deltas (D), buffer them and collect all row ids of rows that changed (depending on the visibility/timestamp of our transaction).
(2) Then, we read the Main store (M) and remove the rows that have row ids from step 1 (we emit $M - D$).
(3) Finally, emit the data from the Deltas (D).

This process takes one pass through all rows of the Delta and one through the Main store. To optimize the filtering (computing $M - D$), we build an ordered set of row ids (from the Deltas). Since now both, row ids from Delta and Main store, are ordered by row id. We can use a merge-style algorithm to filter the rows out. Together with our assumption that most data will be in the Main store (up to 10% data in Deltas), this algorithm is efficient and runs in amortized < 1 cycle per row, because it will mostly skip through.

4.4 Intermediate Columnar Delta

In Hermes the Delta store as explained above will be merged periodically into the Main store (*Update Propagation*). In brief, Update Propagation collects the changes from the Delta and creates new RowGroups in the Main store.

Worst-case For Update Propagation. For workloads with uniformly random updates or deletes the worst-case happens: We might have only a few updates/deletes triggering the creation of a new RowGroup. This is a rather costly process, which leads to extra memory footprint and often turns 1 GiB compressed (one Allocation Unit) into 2-3 GiB uncompressed data. But even more importantly leads to write amplification, as a few rows of changes will lead to writing 1 GiB of data.

Solution: 3 Storage Levels. To mitigate this problem, we add another storage level between the row-based Delta and the compressed columnar Main store: Columnar Delta.

Columnar Delta. The Columnar Delta contains the most-recent committed rows from the Delta, since the last Update Propagation, and a list of deleted rows. The Columnar Delta has two advantages: (1) We mitigate the worst-case by buffering the modifications for longer and (2) we can propagate modifications from the row-based Delta more frequently, which is required to cleanup Deltas.

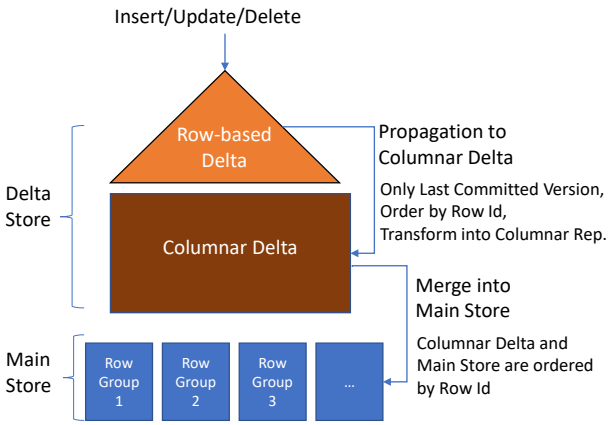


Figure 4: Storage Levels in Hermes: Insert/Deletes/Updates (modifications) first reach the Row-based Delta. Different processes move data from one level to another. We need an intermediate delta (Columnar Delta) to handle uniformly distributed modifications. Otherwise, uniformly distributed modifications would introduce into nearly every RowGroup forcing a complete rewrite of the Main store. Besides being incredible inefficient, this led also to significant write-amplification. The Columnar Delta mitigates this by buffering changes for longer.

We illustrated the flow of changes through the levels in Figure 4. In brief, incoming updates/deletes/inserts (from Change Propagation) will first go to the Delta. Over time they will be promoted to the Columnar Delta until they reach the compressed Main store.

4.5 Update Propagation: Moving Data from Delta to Main Store

Since reading from the Delta store is comparatively slow, we need to periodically move data from the write-optimized Delta to the read-optimized Main store. We call this process *Update Propagation*.

In general, Update Propagation moves data down the levels (as also illustrated in Figure 4). We differentiate between two different sub-processes: Propagation to the Columnar Delta and Merge into the Main Store, which we explain:

Propagation: From Row-based Delta to Columnar Delta. The first processes *Propagation to the Columnar Delta* moves data from the row-oriented Delta into a read-oriented Columnar Delta [6]. The Columnar Delta is column-oriented and only contains the most recent (last committed) subset of the data. To bring down the cost of the future Merge into Main store and to allow efficient scans, we also order the Columnar Delta by row id. The Update Propagation into the Columnar Delta, creates a new row-oriented Delta.

This process is triggered rather frequently. This happens when we need to clean up row-oriented Deltas, i.e. whenever we require more space for new data or need to consolidate data for faster read access. The Columnar Delta not only mitigates the issues mentioned in Section 4.4 but also allow preparing the data for the next step: Merging into the Main Store.

Merge: From Columnar Delta into Main Store. Once the size of the Columnar Delta exceeds a certain threshold or a certain time period is reached, we merge the Columnar Delta into the Main Store. On the high-level, the process is as follows: We first try to find matching RowGroups. Afterwards, we merge each RowGroup.

1. Finding Matching RowGroups. Since both Columnar Delta and Main Store are ordered by row id, we use a merge-style algorithm to find the matching RowGroups. The pseudocode is shown in Listing 1.

We iterate through all RowGroups of the Main store for table `T`. Note that the Columnar Delta `D` rarely contains changes for many RowGroups. Therefore, we can frequently skip through RowGroups (step 1). It can happen that the inserts/updates/deletes from Change Propagation would logically modify many RowGroups. Physically, these changes are buffered using the Columnar Delta (discussed in Section 4.4).

After skipping irrelevant RowGroups, we collect the changes relevant for each RowGroup (step 2). There are two special cases possible: (a) Row ids of changes can fall in between two RowGroups or (b) Changes need to be appended at the end of the Main store. For case (a): If the row ids of the changes are less than the first row id of the next RowGroup, we append the changes to the previous RowGroup. If there is no next RowGroup, case (b), we need to append at the end, thus, creating new RowGroups.

2. Merge each RowGroup. After collecting the changes for each RowGroup, we iteratively merge the changes into the RowGroups (in a background thread). Merging changes into a RowGroup `R` creates a new RowGroup `R'` and involves these steps:

(1) Decompress the RowGroup `R` into a buffer `B`. Note that this step can possibly introduce quite some memory footprint, as a compressed RowGroup of 1 GiB (one Allocation Unit) can often lead to roughly 2-4 GiB for uncompressed data (Section 7.5).
(2) Apply the changes to `B`. For all the changes, we just mark deleted rows as deleted, the following step will physically remove them. For updates, we modify the rows in the buffer.
(3) Create the new RowGroup `R'` by compressing the new buffer `B` and removing the deleted rows.

Each Merge creates a new version (V) of the Main store. RowGroups that were changed they are annotated with version V, otherwise they stay unmodified and keep their old version. Note that for unchanged RowGroups we merely copy a pointer to the RowGroup and, thus, do not re-create a new Main store from scratch but rather reuse RowGroups without changes. Besides transactions, Checkpointing uses this mechanism to determine what to back up (if V is newer than the version of previous checkpoint). Note that since, we use statistics per RowGroup, we do not need to update global statistics for query optimization as well as early pruning.

4.6 Garbage Collection

After Update Propagation, we clean up our data storage, a process that is also triggered periodically.

Row-Based Deltas. The propagation of updates from the row-based Delta to the Columnar Delta creates a new row-based Delta.

[6] We introduced the Columnar Delta as step in between because Merging into the Main Store can in some scenarios be rather costly and lead to significant write amplification. We described the rational in Section 4.4

Listing 1: Finding matching RowGroups for Table T

```
FindMatchingRowGroups(Table T, ColumnarDelta D) {
  curr_group := MainStore(T) // Current RowGroup
  delta_row := 0;            // Current index in D
  row_group_changes = {}     // Changes per RowGroup
  appends_at_end = {}        // Rows appended @ end

  while (curr_group) {
    // Step 1: Skip RowGroups without changes
    if (curr_group->first_id < D[delta_row].row_id) {
      curr_group := curr_group->next
      continue;
    }

    // Step 2: Collect changes for each RowGroup
    row_group = &row_group_changes[curr_group]
    while (curr_group->last_id >= D[delta_row].row_id
        && delta_row < D.size()) {
      // Append our change to RowGroup
      row_group->Append(D[delta_row])
      delta_row++
    }
    curr_group = curr_group->next

    // Step 3: Append remaining changes to previous RowGroup
    while (curr_group
        && curr_group->last_id >= D[delta_row].row_id
        && delta_row < D.size()) {
      row_group->Append(D[delta_row])
      delta_row++
    }

    // Remaining changes are appended at the end
    while (!curr_group && delta_row < D.size()) {
      appends_at_end.Append(D[delta_row])
      delta_row++
    }
  }
  return (row_group_changes, appends_at_end)
}
```

The old Delta(s) can be garbage collected, once no active transactions access them anymore. We track the active transactions via the GTID mapping that maps MySQL timestamps to Hermes timestamps (described in detail in Section 5.2).

Releasing the Delta in bulk solves two challenges: (a) It avoids fine-grained garbage collection to minimize the length of the version chains (due to costly row reconstruction). (b) Sub-structures can be append-only (e.g. `RowIdMap`, `RowSpace`, `UndoSpace`), making a lock-free implementation easier.

Columnar Delta. Garbage collecting the Columnar Delta is trivial i.e. after a Columnar Delta has been merged into the Main Store and it can be discarded.

Main Store. The Main store contains multiple versions. Essentially we only need to keep either the last version with active transactions or the most recent version. Similar to the row-based Delta, we use the GTID mapping to prune old RowGroups and Main store versions.

5 INTEGRATION WITH MYSQL

Hermes operates as an accelerator for MySQL. In this section, we describe how we integrated with MySQL. First, we explain how we replicate the state from MySQL to Hermes. Afterwards, we describe how we achieve consistent snapshots (between Hermes and MySQL), followed by how we keep functionally consistent to MySQL and work around some of MySQL's quirky-ness.

5.1 Replication

Our replication mechanism piggy backs on MySQL's replication mechanism. We replicate transactions from MySQL using two processes *Ingestion* and *Change Propagation*:

MySQL's Replication. One of MySQL's many features is: Replication. Replication allows many different solutions reaching from backups and failover to scale-out or accelerators (here) [17].

MySQL allows subscribing to the binary log (binlog) from which replicas can gather all committed transactions. Each transaction is identified through a GTID (Global Transaction Identifier).

Ingestion. *Ingestion* splits the table into chunks, and inserts each chunk in parallel. While it is possible to insert into the Delta, it is rather inefficient as it requires the changes to be propagated to the Main store via Update Propagation. Additionally, we know that we do not require fine-grained versioning as there is only one version. Therefore, we skip the Delta and directly append each chunk into the RowGroups, whenever the sorting order of the table in MySQL matches the row ids, which is almost always the case.

Change Propagation. *Change Propagation* replicates binlog events from MySQL to Hermes, essentially replicating the state of the database.

Naive Change Propagation. In our initial implementation, log events would go through 3 stages: (1) log parsing/interpretation would create the event, (2) we transform data (into the right shape from binlog format to format suited to the Delta) and, finally, (3) apply the changes via transactions directly on the Delta. Further we parallelized each stage, i.e. (1), (2) and (3) can run in parallel. Naively, one would apply events strictly in commit-order. This requires transactions with a GTID older than the current transaction's GTID is be committed, no matter of whether the second transaction actually depends on the first transaction. We noticed that this significantly limits the transactional throughput.

Out-of-Order Change Propagation. Relying on the strict commit-order introduces significant latency proportional to the amount of changes being replicated. To break the strict commit-order, we explicitly track the dependencies of transactions. When running a transaction against a specific version, we require all its dependencies to be committed, and its changes to be visible, before. Internally, requires Hermes to commit transactions out-of-order (even though T_1 has been committed before T_2 im MySQL, dependencies can force Hermes to first commit T_2). Technically, we keep a cyclic buffer of the last N transactions and track their state.

5.2 Consistent Snapshots via MVCC

When a transaction is committed in MySQL (with binary logging enabled), it receives a GTID (Global Transaction Identifier). When a MySQL transaction is replicated to Hermes, it is first assigned a Hermes Transaction Identifier (HTID) and a start timestamp (T_{start}) that defines the view (i.e. everything $< T_{start}$). To achieve view consistent with MySQl, we maintain a mapping from GTIDs to HTIDs. Whenever a Hermes transaction commits, it will receive a commit timestamp (T_{commit}, allocated from the same counter as T_{start}). After the transaction has successfully been committed T_{commit} is added to the mapping. Following transactions, will see that there is a mapping from the current GTID to a commit timestamp and will use that timestamp as T_{start}.

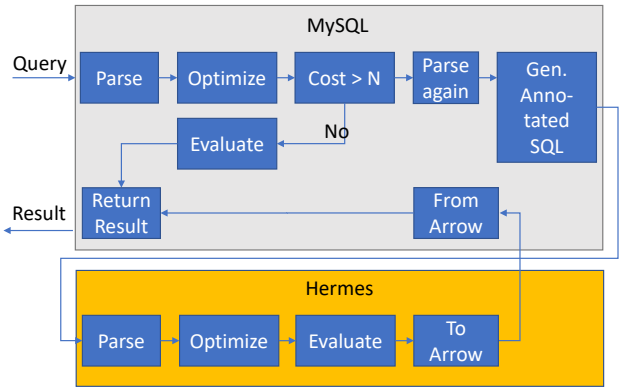


Figure 5: Flow of a SQL query through MySQL and Hermes. After MySQL optimized the query we check whether the cost is high enough and all features are supported. If offloading to Hermes is supported, we parse the query again, resolve types and, afterwards, generate an annotated SQL query that is sent to and evaluated by Hermes, which, then, returns the result serialized in the Apache Arrow format. This is, then "unwrapped" by MySQL and reported to the user.

It can happen that the changes for a transaction are not yet replicated. In that case, we will wait until the changes have been replicated (and to corresponding mapping from GTID to HTID exists). Practically, this waiting time is in the low milliseconds and negligible compared to the runtime of the query, which is often hundreds of milliseconds or more.

5.3 Functional Consistency

An often under-appreciate challenge is keeping the query results consistent with MySQL. Quite certainly countless hours have been spent on this possibly never-ending endeavour. For instance, MySQL is able to answer seemingly erroneous queries in unexpected ways:

- `select 3+'abc'` will return 3, as MySQL turns `'abc'` into 0.
- `select substr(123, 1, 1)` will return 1, because MySQL turns the integer 123 into the string `'123'`.

Approach. How a query flows through MySQL and Hermes is illustrated in Figure 5, a description follows: When we run a MySQL query, we first analyze it's query plan. We check whether Hermes supports the feature and whether the estimated cost metric is high enough to justify offloading to Hermes. Otherwise, MySQL will run the query anyways.

If Hermes supports the query, we let MySQL parse the query again. Then, MySQL resolves the types of each expression, and generates an annotated SQL query that resolves, above mentioned, quirky issues. In particular, we add optional type casts to solve above issues. Afterwards, Hermes uses the generated SQL to evaluate the query mostly oblivious to MySQL casting rules. Whenever, the optional casts is consistent with our own types, we skip the cast to save CPU cycles.

In summary, Hermes supports a wide range of MySQL's features and support is close to the top-notch cloud database products GaussDB for MySQL and Taurus [6].

6 QUERY EXECUTION

Hermes features a relatively complete query engine, including support for collations, Window functions and JSON.

Vectorized Execution with SIMD. For executing queries we use Vectorized Execution [3], which partitions the input into cache-sized data chunks (0.5-2k rows). Each chunk is then processed in a columnar fashion in a tight loop. Furthermore, we extended our engine with SIMD (AVX-512). Especially filters on base tables are optimized using AVX-512. AVX-512 shines are creating Selection Vectors via the `vpcompress` instruction [12] [7] Furthermore, we introduced micro-adaptive optimizations [19] for mostly full vectors and densely filled, we skip the evaluation via Selection Vector and directly access the underlying vector(s) like e.g. Excalibur does [10].

Intra-Query Parallelism. Each query exploits multiple cores using Morsel-driven parallelism [16]. Data is logically split into Morsels (data chunks, larger than vectors, typically 10-32k rows). Each thread dynamically consumes Morsels and processes the associated data until all Morsels are processed.

7 EXPERIMENTAL EVALUATION

We integrated Hermes with MySQL 8.0.22. We setup Hermes as a single node with storage located in main-memory with MySQL located on the same machine. For most benchmarks we used a two-socket with Intel Xeon Platinum 8378A with 512GB DRAM and 32 cores (64 logical cores) each. If we do not use that machine, we denote it in the experiment. First, we evaluate Hermes on a medium-scale analytical workload (TPC-H SF100), followed by a large scale workload (SF1000). Afterwards, we benchmark the throughput of Hermes' replication mechanism. Then, we investigate the impact of updates onto the scan performance as well as the compression rate our lightweight compression schemes achieve.

7.1 Medium-scale: TPC-H SF100

We believe most customers do not have extremely large datasets. Therefore, we focus on the medium-large dataset from TPC-H scale factor (SF) 100, which is roughly 100 GiB of uncompressed data.

Hermes vs. MySQL. First we compare our accelerator (Hermes) against MySQL. We ran the queries from the TPC-H benchmark on scale factor 100, the result can be seen in Table 1.

Compared to MySQL, we see speedups are 2-3 *orders of magnitude*, reaching up to 1,800×. The worst speedup seen is a "meager" 17×. Thus, Hermes can significantly accelerate analytical queries.

PolarDB. PolarDB-IMCI [24] is a close competitor to Hermes. Here, we compare against PolarDB-IMCI using their reported runtimes [2]. We used the best numbers shown for PolarDB which is in that case using four machines/nodes with 32/16 logical/real cores each. The results can also be seen in Table 1. Hermes outperforms PolarDB in 19 out of 22 queries and in 7 queries by more than 2×.

Hermes underperforms in 3 queries. Note that these queries are still executed reasonably fast (in under half a second).

PolarDB Single-Node. Additionally, we compare to single-node PolarDB-IMCI. Similarly, we used the numbers reported by the same website, but adjusted to runtimes to the number of cores, assuming linear scalability (shown as *IMCI 1 Node*; we assumed dividing the

[7] `vpcompress` builds copies values from the input into the resulting when a bit in the mask is set. The values in the result are stored sequentially without gaps.

Table 1: Runtimes on TPC-H SF100 (secs): On medium-large datasets, across the board, we significantly outperform MySQL (optimized for transactions) by orders of magnitude and PolarDB-IMCI (optimized for analytics) often by $2-4\times$.

Query	Hermes	MySQL		PolarDB-IMCI		IMCI 1 Node	
	Time	Time	$\frac{\text{MySQL}}{\text{Hermes}}$	Time	$\frac{\text{IMCI}}{\text{Hermes}}$	Time	$\frac{\text{IMCI}}{\text{Hermes}}$
Q1	0.81	964	1,190	1.50	1.9	2.33	2.9
Q2	0.21	13	61	0.45	2.1	0.36	1.7
Q3	0.56	394	703	0.81	1.4	1.22	2.2
Q4	0.51	88	173	0.37	0.7	0.73	1.4
Q5	0.52	214	411	0.98	1.9	1.08	2.1
Q6	0.30	129	430	0.18	0.6	0.27	0.9
Q7	0.55	831	1,511	0.88	1.6	1.21	2.2
Q8	0.54	531	983	1.01	1.9	1.05	1.9
Q9	2.49	301	121	4.74	1.9	7.15	2.9
Q10	0.89	151	170	2.42	2.7	2.23	2.5
Q11	0.18	333	1,847	0.74	4.1	0.53	2.9
Q12	0.46	172	375	0.53	1.2	0.85	1.8
Q13	1.14	1,143	1,003	3.56	3.1	2.92	2.6
Q14	0.45	156	347	0.38	0.8	0.55	1.2
Q15	0.56	300	536	0.82	1.5	1.23	2.2
Q16	0.33	25	76	1.19	3.6	1.13	3.4
Q17	0.53	46	86	2.51	4.7	1.46	2.8
Q18	1.46	1,196	819	2.46	1.7	6.36	4.4
Q19	0.66	11	17	0.68	1.0	1.06	1.6
Q20	0.45	38	84	0.80	1.8	0.61	1.4
Q21	1.69	1,392	824	1.82	1.1	3.46	2.0
Q22	0.28	112	401	0.98	3.5	0.70	2.5
Total	15.6	8539	336	29.81	1.9	38.51	2.5

runtimes by 2 would compensate for half the CPU cores). Except for the slightly slower Q6, we outperform single-node by up to 4.4×.

Reaching Single-Node Bandwidth. PolarDB, for example, outperforms in Q6. Q6 is a rather simple query (scanning, filtering and computing global aggregates) that is very sensitive to scan bandwidth (most rows are read, but thrown away). Since PolarDB uses 4 nodes, it can get a higher memory bandwidth (4× single-node bandwidth). While, Hermes can get a higher net bandwidth on scans due to lightweight compression [25], distinct nodes can still achieve a higher bandwidth (no NUMA traffic and more memory controllers with direct access to DRAM).

However, for more complex queries (with e.g. a group-by clause), multi-node PolarDB will have to re-partition data across the network, giving it a disadvantage.

Negligible Overhead of Additional Parsing and Optimization Steps. When a query travels from MySQL to Hermes, is goes through multiple steps. These include parsing and optimizing one query *twice* as well as generating an annotated SQL query from the plan (from MySQL). First we need MySQL to parse and optimize the query to decide whether to offload to Hermes (cost $> N$.), but we also require additional annotations (optional casts) to keep query semantics consistent with MySQL. Afterwards, we chose to generate SQL text from the annotated plan. Afterwards, the annotated SQL query is parsed and optimized again by Hermes. We did not notice significant overheads arising from these additional steps on TPC-H scale factor 1 and larger.

Table 2: Runtimes on TPC-H SF1000 (secs): Hermes is able to larger workloads as well. On the $10\times$ larger data set, we outperform, the multi-node, PolarDB-IMCI by roughly $2\times$.

Query	Hermes	PolarDB-IMCI		IMCI 1 Node	
	Time	Time	$\frac{\text{IMCI}}{\text{Hermes}}$	Time	$\frac{\text{IMCI}}{\text{Hermes}}$
Q1	16.4	14.6	0.9	54.1	3.3
Q2	3.6	5.4	1.5	13.0	3.6
Q3	10.3	9.2	0.9	37.0	3.6
Q4	8.7	4.6	0.5	28.4	3.3
Q5	7.9	10.1	1.3	38.2	4.8
Q6	2.6	1.7	0.7	20.5	7.9
Q7	6.2	9.6	1.5	43.6	7.0
Q8	7.7	7.5	1.0	42.2	5.5
Q9	16.9	58.7	3.5	164.4	9.7
Q10	9.4	35.9	3.8	58.6	6.2
Q11	2.1	6.4	3.0	5.4	2.6
Q12	5.8	5.8	1.0	40.8	7.0
Q13	17.2	63.8	3.7	46.9	2.7
Q14	6.0	3.8	0.6	26.5	4.4
Q15	5.2	8.5	1.6	50.2	9.7
Q16	2.7	10.0	3.7	11.5	4.3
Q17	7.1	24.4	3.4	48.9	6.9
Q18	24.1	63.6	2.6	239.4	9.9
Q19	10.2	21.6	2.1	48.6	4.8
Q20	7.4	7.3	1.0	29.4	4.0
Q21	26.6	22.5	0.8	94.3	3.5
Q22	4.4	10.2	2.3	8.4	1.9
Total	208.4	405.0	1.9	1150.0	5.5

7.2 Large-Scale: TPC-H SF1000

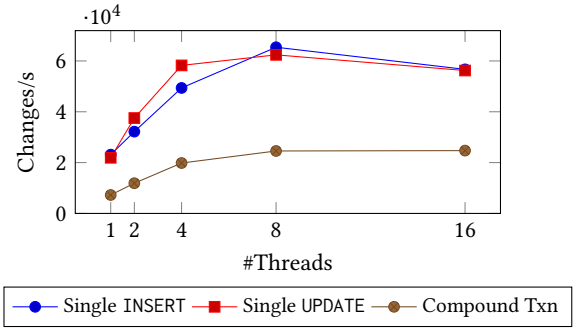
Besides, medium-large data sets Hermes is also able to run much larger workloads. Therefore, here, we show Hermes' performance on the 10× larger dataset of TPC-H SF1000. For PolarDB-IMCI, again, we used the numbers reported by them [2]. We used the best numbers shown for PolarDB which is in that case using four machines/nodes with 32/16 logical/real cores each (*PolarDB-IMCI*). Additionally, we compare to single-node PolarDB-IMCI (*IMCI 1 Node*). Similarly, to Section 7.1, we used the numbers reported, but adjusted to runtimes to the number of cores.

Table 2 shows our results: Hermes is able to run all 22 queries. Note that the compressed data size of SF1000 should be around 350 GiB, which leaves us with enough main memory to run queries.

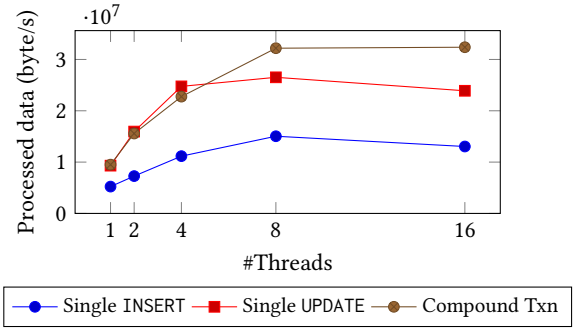
Again, we outperform our close competitor PolarDB-IMCI by a factor of 1.9× (total runtimes). Like in SF100, Hermes is limited in its scan bandwidth, compared to a multi-node system. However, in more complex queries with larger joins Hermes outperforms (e.g. Q2, Q9, Q10, Q22). Compared to single-node PolarDB, we outpeform across the board and by up to 9.9×.

7.3 Replication Throughput

Hermes guarantees access to "fresh data". If the wished version has not been replicated yet, Hermes will have to wait. In practice, we have seen wait times in the low milliseconds, thus negligible to the query runtime. Each change has to be committed on MySQL, first. Afterwards it is processed by Change Propagation (interpreting binary log, creating operations on Delta store and eventually



(a) Throughput in operations per second: Performance peaks at around 60k changes/s at 8 with cores. More complex transactions require more time per change and consequently have a lower throughput in terms of changes/s (this plot). However, in terms of bytes processes (Figure 6b) throughput is higher.



(b) Throughput in binary log bytes processed per second: #bytes of log entries is proportional to the data size of the changes. We can see that more complex transactions (*Compound Txn*) lead to better performance. However, too large transactions can lead to higher waiting time as they can have more dependencies that need to be resolved beforehand.

Figure 6: Change Propagation Throughput. *Compound Txn* is a transaction consisting of one insert, one delete and two updates.

committing the transaction.). In this experiment, we measure the throughput at which Hermes can receive changes (whole pipeline, from MySQL until commit on Hermes).

Hardware. We measure the throughput on a different machine, Intel Core i9-13900K with 8 performance and 16 efficient cores (i.e. Intel's version of ARM's big.LITTLE) and 128 GiB DRAM.

Workload. We created a table with two integer columns and two string column and 50k rows. Based on that table with run the three following experiments, each runs within transactions: In run (a) 10M single inserts, (b) 10M single updates and (c) 3M compound transactions consisting of one insert, one delete and two updates.

Changes per Second. Figure 6a shows the number of changes processed per second. All three experiments, show similar scaling behavior, i.e. they scale until 8 cores are reached and flatten out afterwards. This is likely a consequence of the hardware we chose, which only provides 8 high-performance cores. Peak throughput is at 60k changes per second (for single inserts and updates).

Interestingly, the throughput for the larger compound transaction (*Compound Txn*) is roughly 3× lower. The reason is that larger transactions involve more operations and, therefore, take longer to process. Note that *Compound Txn* contains 4 operations, if we normalize per operation *Compound Txn* could have a rate of roughly 80k changes per second.

Processed Bytes per Second. Additionally we also measure the amount of binary data our Change Propagation processes. This is a fairer comparison with respect to transaction size. We visualize the measurements in Figure 6b.

We see similar scaling behavior, for the same reason (8 high-performance cores). Furthermore, we see a higher throughput for larger transactions (*Compound Txn*), roughly 2× compared to *Single* `INSERT`. This indicates that there is some overhead per transaction.

Similarly, if we compare *Single* `INSERT` and *Single* `UPDATE`, we see a significant difference (although less than 2×), indicating that currently inserts (i.e. essentially appends) in Hermes are slower. This is likely caused by latch contention: We need a delta-wide (per table) latch to allocate a new row in the Delta, an update just needs a latch (determined by its row id). We are planning to optimize this case, replacing latches with atomic operations, in the near future.

Summary. Overall, Hermes can sustain up to 60k single transactions inserts/updates per second on a (small) single-node, and potentially higher rates on a larger machine. Nevertheless, we are actively working to increase replication throughput further.

7.4 Scan Performance with Updates

The efficiency of Hermes' scans depends on the amount of changes present in the delta. In this experiment, we investigate the scan performance with varying amounts of `UPDATE`s.

We created a table with $N = 2^{26}$ ($\approx$ 6.7M) rows and 11 64-bit columns: 1 primary key column, with values from 0 to N, and 10 payload columns that are uniformly random. The primary key column is only used for updates, which are also uniformly randomly distributed. After issuing the updates, we scan all 10 payload columns single-threaded. Figure 7 shows the timings of the scan.

Scan Performance. Without updates (i.e. without a Delta), the scan runs in 11.5 cycles/row, i.e. 1.2 cycles/attribute. In the plot, we show the performance degradation compared to the baseline (no updates).. In general, with an increasing number of updates, we see an increasing degradation of scan performance. Until 0.2% of the table updated, we can see almost no degration($\lesssim$ 20%). At around 0.5%, we see a 50% degradation. At 1%, we see a 100% degradation (i.e. scans roughly twice as expensive). Until, at around 3%, the scan is roughly 4× slower.

The reason for the performance degradation is the increasing cost of scanning and pre-processing the Delta as well as eliminating the changed rows during the scan.

Filtering Main Store with Delta. Below 0.01% updates, filtering the data from the Main store with the Delta costs $\leq$ 1 cycle per row. At around 1%, filtering costs only 2.5 cycles per row. However, compared to the other costs, filtering is rather insignificant.

Other Costs (*Rem. Scan*). During the scan we have other costs as well: We need to (1) scan the Delta and (2) collect and order the row ids. Both steps are included into *Rem. Scan* and are normalized per attribute (i.e. per row it would be 10× that).

From $\approx$ 1% of the table updated, we see scanning & pre-processing the rows, from the Delta, quickly becoming costly.

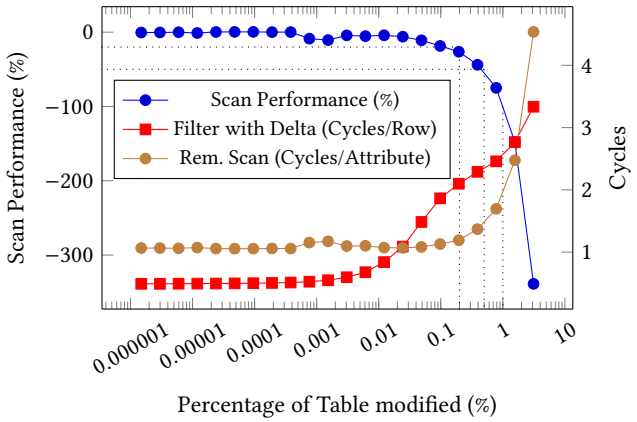


Figure 7: Scan performance on a (static) table with updates: Scan performance decreases with increasing number of updates. Below 0.2% of the table updated, scan performance degraded by ≤ 50%, whereas at 1% scan performance halves. In practice, we try stay below 1% by eagerly merging changes.

Table 3: Hermes lightweight compression compresses datasets by roughly 3×. Since we are using lightweight compression schemes, decompression is practically free. Furthermore, we compress into small cache-sized arrays, which very likely stay in cache. Compression ratio on SF100.

Dataset	Uncompressed (GiB)	Compressed (GiB)	Ratio
TPC-H	115.5	35.5	3.2
TPC-DS	79.1	31.3	2.5

One reason for this behaviour is that we did not optimize this code path (yet), because this only happens for OLAP queries whereas DML directly goes into the Delta..

Summary. Scans get slower the more rows are updated: Until 1% of the table the overhead is "negligible". Until 0.2% it is ≤ 20%. Until 1% scans become roughly twice as slow, but that would be barely noticeable in complex OLAP queries which are less likely limited by scan performance alone.

7.5 Compression Ratio

Hermes uses lightweight compression to (a) store more data in a single-node and (b) increase the net bandwidth when reading data (decompress into CPU cache). In this experiment we show how well our compression schemes compress. Table 3 shows the compression rate (uncompressed vs compressed) on TPC-H and TPC-DS. Both datasets include integers, `DECIMAL`s and strings (variable length data).

We noticed significant compression ratios of roughly $2 - 3\times$. This allows Hermes to fit very large datasets still into memory and avoid accessing memory from other nodes, which would require a multi-node systems involving costly data transfers across the network. While we achieve significant compression ratio on uniformly random data (TPH-H and TPC-DS), we did not notice significant time spent on decompression. This strengthens the point that lightweight compression does not introduce any significant overhead and thus bring data compression "for free".

8 CONCLUSION

In this work, we presented Hermes – a cloud-native accelerator for analytical workloads. Unlike many other cloud solutions, we chose to design Hermes for single-node. We decided for a single-node system, because customer workloads are often now extremely large (regarding data size) to justify a multi-node distributed system that frequently ends up mostly transferring data across the network, instead of performing useful work.

We present Hermes' architecture and highlight important details as data storage optimized for fast insert/delete/update as well as analytics. Another very important feature of Hermes is its compatibility and integration with MySQL. We transparently replicate MySQL's data (via its binary log) and allows reading specific versions of the data. Before we can run a query on a specific version we must ensure that version has been replicated. If that version has not yet replicated, we wait. However, in practice the waiting times are negligible compared to the runtime of analytical queries (1-5 ms waiting vs. 500 ms query). Especially, we highlight how we achieved an implementation that functionally emulates MySQL with most, if not all, its quirky-ness.

We evaluated Hermes and achieved speedups of 2-3 orders of magnitude compared to MySQL for analytical queries (TPC-H SF100). Compared PolarDB-IMCI [24], a cloud-native distributed system optimized for analytics, we achieved an overall speedup of roughly 2×. In many TPC-H queries, we significantly outperform PolarDB, in certain queries, by up to 4.7×. But Hermes is also able to operate on much larger workloads and even on TPC-H SF1000 outperforms PolarDB. Since high replication throughput is essential to Hermes, we evaluate replicate performance in an experiment. Hermes was able to handle to up 60k changes per second, on a workstation (server hardware is certainly more powerful).

Future Work. We are currently still actively working on improving both analytical and replication performance further. Replication throughput can improved further by replacing latching in the Delta store with lock-free operations. Our replication framework is quite flexible and can target other sources as well, if MySQL's binary log turns out to be too slow. Besides, we are exploring possibilities for multi-node setups (for storage and query execution) that do not suffer from excessive network traffic and complexity. In addition, we are also exploring alternative hardware architectures and optimizations via hardware-acceleration.

Closing Words. While in this paper, we only presented Hermes as an in-memory accelerator for MySQL. Hermes allows many different designs as well. Hermes can dynamically offload parts of the Main store to Object Storage and cache the hot data set in memory, essentially allow running workloads much larger than main memory, i.e. much larger than terabytes of data.

ACKNOWLEDGMENTS

Since this has been a team effort, we would like to thank the remaining team members: Albert Lesniewski, Daniel Seglsten, Diego Tomé, Eivin Hatvik, Erik Frøseth, Guilhem Bichot, Jiang Wei, Jon Olav Hauglid, Lars Eidnes, Matheus Nerone, Ole John Aske, Ole Hjalmar Kristensen, Pedro Figueiredo, Rahul Malik, Samuel Krempasky, Stig Bakken, Tiago Kepe, Valentyn Doroshchuk, Vitor Oliviera, Zhou Jiayu.

REFERENCES

[1] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*.

[2] Alibaba Cloud. 2025. IMCI performance. https://www.alibabacloud.com/help/en/polardb/polardb-for-mysql/imci-performance. [Online; accessed 10-March-2025].

[3] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[4] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: ByteDance's HTAP System with High Data Freshness and Strong Data Consistency. *VLDB* (2022).

[5] CWI. 2025. Public BI benchmark. https://github.com/cwida/public_bi_benchmark. [Online; accessed 10-March-2025].

[6] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, et al. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *SIGMOD*.

[7] DuckDB. 2025. DuckDB. https://duckdb.org/. [Online; accessed 10-March-2025].

[8] Goetz Graefe. 1990. Encapsulation of parallelism in the volcano query processing system. *SIGMOD Record* (1990).

[9] Tim Gubner. 2014. Achieving many-core scalability in Vectorwise. *Master's thesis, TU Ilmenau* (2014).

[10] Tim Gubner and Peter Boncz. 2022. Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA. *VLDB* (2022).

[11] Sándor Héman, Marcin Zukowski, Niels J Nes, Lefteris Sidirourgos, and Peter Boncz. 2010. Positional update handling in column stores. In *SIGMOD*.

[12] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *VLDB* (2018).

[13] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle Database In-Memory: A Dual Format In-Memory Database. In *ICDE*.

[14] Per-Ake Larson, Cipri Clinciu, Campbell Fraser, Eric N Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L Price, Srikumar Rangarajan, Remus Rusanu, et al. 2013. Enhancements to SQL Server Column Stores. In *SIGMOD*.

[15] Per-Åke Larson, Cipri Clinciu, Eric N Hanson, Artem Oks, Susan L Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL Server Column Store Indexes. In *SIGMOD*.

[16] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*.

[17] MySQL. 2025. Chapter 19 Replication. https://dev.mysql.com/doc/refman/8.4/en/replication.html. [Online; accessed 10-March-2025].

[18] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*.

[19] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro Adaptivity in Vectorwise. In *SIGMOD*.

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

[21] Alexander Van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *VLDB* (2023).

[22] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTEST*.

[23] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *USENIX*.

[24] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *SIGMOD* (2023).

[25] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *ICDE*.

[26] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. 2012. Vectorwise: a Vectorized Analytical DBMS. In *ICDE*.