Hermes at Scale: Powering Distributed Queries with a Unified Memory Fabric

Tim Gubner
Huawei Europe
tim.gubner@huawei.com

Matheus Nerone
Huawei Europe
matheus.agio.nerone1@huawei.com

Diego Tomé
Huawei Europe
diego.gomes.tome@huawei.com

Vitor Oliveira
Huawei Europe
vitor.oliveira@huawei.com

Rune Humborstad
Huawei Europe
rune.humborstad@huawei.com

Manyi Lu
Huawei Europe
manyi.lu@huawei.com

ABSTRACT

MySQL is a widely used database management system. However, it has clear deficits for analytical workloads. To improve analytical performance, we introduced Hermes as an accelerator for MySQL. Since many workloads easily fit a single node/machine, we originally developed Hermes for this case. However, Hermes on a single node has a major limitation: Storage space is limited to a single-node, even worse, to main memory of a single node. In this paper, we address this problem by making Hermes distributed. New nodes can be added that bring additional resources, i.e. memory and compute, to the cluster. Hermes' storage can automatically expand to the new nodes. Our design operates as logically "Shared Memory" via an abstraction layer (Memcom). We argue that modern interconnects make shared memory feasible without extreme performance penalties. To further minimize sending data over the network, we annotate larger data blocks with localities, which allows reader nodes to locate and read local data.

In addition, we extended Hermes' query execution to multiple nodes as well. This is implemented via a new Planner that splits the query plan into tasks and defines how query execution is parallelized (MPP intra-query parallelism). Each task is separated via data sharing operators (Portals) that write intermediates to a shared memory pool (Memcom). We extended scans to pick up row groups (our granularity of data storage) based on their locality. Therefore, distributed scans will preferably read local data (thanks to our predefined localities) and only when there is no more local data to be read consider remote data.

We experimentally show that our approach is not only competitive in terms of performance but outperforms well-known open-source systems by more than 11×. Compared to industrial-grade cloud systems, Hermes also outperforms by 1.5×.

PVLDB Reference Format:
Tim Gubner, Matheus Nerone, Diego Tomé, Vitor Oliveira, Rune Humborstad, and Manyi Lu. Hermes at Scale: Powering Distributed Queries with a Unified Memory Fabric. PVLDB, 19(12): 3969 - 3981, 2026.
doi:10.14778/3827998.3828009

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828009

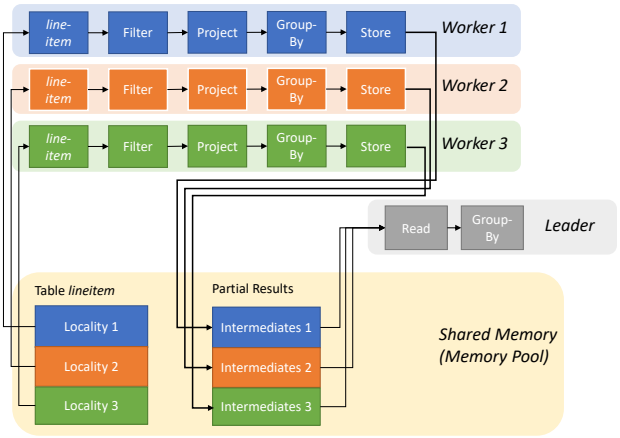


Figure 1: Hermes' Distributed Execution in a Nutshell: Scans *preferably* scan their respective local data. Once out of local data, Scans consider reading remote data. Worker nodes process part of the query and write the partial results back to Shared Memory. After all workers finished, the leader picks the partial results up and computes the final result. Accesses to Shared Memory go through our abstraction layer *Memcom*.

1 INTRODUCTION

MySQL has a wide user base, but since it is a database management system optimized for transactions (OLTP), it has very low performance for analytical workloads (OLAP). In the past, we addressed this issue by proposing Hermes [17], our single-node Accelerator for OLAP. Hermes supports fast transactions, replicated through MySQL binary log and applied to our DML-optimized Delta store. In order to support fast analytics, the system periodically moves rows from the Delta store to the Main store – a lightweight compressed columnar store, optimized for analytics. Beyond Hermes's performance, it is also semantically compatible to MySQL, i.e. expressions/functions behave the same and the data visible to transactions is of the same version.

In our previous work on Hermes [17], we argued that for most use-cases a distributed query execution is "overkill", because most datasets are reasonably small and can easily be processed on a single machine [28, 29]. Thus, distributed query execution is often rather a hurdle than an improvement due to (a) excessive data transfers over the network (b) partitioning can cause skew across nodes with unbalanced work and (c) additional intrinsic complexity.

Nonetheless, customers (or rather their "decision makers") often prioritize solutions with the ability to scale to multiple machines, not only in storage but also in execution. Cheaper memories have enabled the emergence of in-memory database management systems (DBMSs). While this method typically increases performance, it comes with the requirement that data must be memory resident, which can limit the amount of data that can be processed. When the data does not fit main memory and goes out of core, performance significantly degrades. To address this challenge, scale-out or distributed in-memory DBMSs have been developed. These systems distribute data across multiple servers and memory spaces, allowing for the efficient processing of large datasets while maintaining the performance benefits of in-memory DBMSs. Single-node Hermes [17] requires all data to be located in main memory and is, therefore, not scalable with the amount of data. In this paper, we lift this memory limitation and go beyond that allowing also multiple nodes to participate in query execution. The idea is that when a customer runs out of space, another machine can be added, extending the total main memory capacity shared across the cluster. Additionally, new machines also bring extra CPU power, which allows extending query execution to the new nodes. The positive effect is that, with more nodes, query plan fragments can be evaluated on local data, but do not necessarily have to. As a result, we minimize data transfers by first trying to evaluate locally and only if necessary, we consider reading (and processing) remote data.

All these requirements led us into extending Hermes with a distributed engine that processes data like in Figure 1. We have multiple Worker nodes, where each will pick up a task created by the Leader node (here there is just one task) and work on it until it is finished. Shown here is a simple pipeline reminisced of TPC-H Q1, which consists of a scan, filter, projection and a group-by. Inside the scan operator, each worker will read data from shared memory. Importantly, we annotate data with localities, such that a scan will read local data first and only consider remote data, once out of local data, and therefore minimizing network transfers. The following operators do further processing until the result of the pipeline is stored in shared memory. Once, all tasks (the pipeline shown here) have completed, the Leader will pick up the intermediates, compute the final group-by and deliver the result of the query.

Contributions. In this work, we describe our distributed storage and engine. We also describe a major component named *Memcom*[1], our memory abstraction library used to store and access data. Further, we experimentally verify that our approach shows, at least, competitive performance compared to the state-of-the-art. In particular, on the widely-used TPC-H benchmark, we outperform our closest (state-of-the-art) competitor by 1.5×.

Structure. In the next section we compare to related work, followed by a brief reintroduction of (single-node) Hermes. Afterwards, we explain how data is stored in distributed Hermes. Then, we describe the query execution in the distributed engine, followed by an experimental evaluation. Last but not least, we summarize our contributions and indicate future work.

[1]Was used in the first publication on Hermes [17] but not described due to not fitting the narrative.

2 RELATED WORK

In this section, we compare Hermes to the state-of-art in basic architectures for distributed systems as well as competing systems.

2.1 Architectures

Shared Nothing. While shared nothing architectures are known for performance, in the cloud, truly shared nothing architectures have shown major flaws [13]: (a) Although the hardware is homogeneous, the workload typically is not. A system configuration ideal for bulk loading (high I/O bandwidth, light computing) is poorly suited for complex queries. Additionally, (b) if the set of nodes changes due to node failures or because the user chooses to resize the system, large amounts of data need to be reshuffled.

Disaggregated Storage. Disaggregated storage introduces massive data transfers over the network. For instance, whenever a query reads a table and filters out most rows, the full table needs to be transferred over the network. Even, if afterwards, most rows are discarded. To mitigate this issue, systems with disaggregated storage, commonly, try to push down as many operations as they can (usually filters, Bloom filters etc.). While this works fine for simple queries (scan-filter), there are clear limits to push-down. Most notably, there is no guarantee that the query is selective (e.g. it might compute a join between tables that is summarized with a group-by). Then not only all rows are sent to each compute node, but also adds an (small) additional cost to the storage nodes due to unnecessary extra filters.

Shared Memory. Shared memory systems allow each node to access each other nodes' data. Technically, this comes with additional communication overhead when accessing remote data (messages and actually sending data). Remote writes, typically, involve cache coherency traffic as potentially cached copies need to be invalidated. But also local data accesses can introduce additional overhead, as we need to protect data against concurrent remote writes. Shared memory allows for easy development ("feels like one single machine"). However, historically performance penalties have precluded a wide adaption. Nowadays, modern interconnects have low latencies, high bandwidths and are co-developed with software. Such interconnects could lower the performance penalties and bring shared memory (memory pools) "back from the dead". Notable recent examples include CXL, UnifiedBus [22] or HCCS (Huawei Cache-Coherent System).

Hermes. All three, shared nothing, disaggregated storage and shared memory, have flaws that lead to detrimental performance. We have designed Hermes to use shared memory, e.g. for base tables and intermediates. But, try to steer clear of the flaws: With Hermes we optimized for modern super-fast interconnects and generally try to minimize the data transferred over the network by exploiting the location of the data and try to read local data whenever possible.

2.2 Systems

PolarDB-IMCI [30]. PolarDB is a cloud native database system developed at Alibaba Cloud, and adopts a shared storage architecture. It includes one primary read-write node and multiple read-only replicas in the compute node layer. Each Read-Write or Read-Only

node contains a SQL processor, a transaction engine, and a buffer pool to serve queries and transactions. In PolarDB's disaggregated architecture [21] CPU, memory and storage resources are no longer tightly coupled as in a monolithic machine. Data pages in the remote memory pool can be shared among multiple database processes, analogous to the storage pool being shared in shared storage architecture. As a result, the process of adding a read replica does not increase the cost of memory resources because it will access the shared storage already in place. Hermes stores data local to nodes, allowing execution within the node to be as fast as a single node with the benefit of processing more data by allowing multiple nodes to participate. PolarDB, on the other hand, needs to rely on index-aware prefetching to improve query performance when processing data, since CPU, memory and storage are not coupled as in a monolithic machine.

SparkSQL [5]. Using Resilient Distributed Datasets (RDDs) [32]. Spark typically loads data from Object Storage lazily into RDDs and computes the results per partition. Computations are done local to a machine (if possible, i.e. partitioned), and the assignment is dynamic. Until the next operation requires data in a different partitioning. Then, data has to be shuffled across machines. In case of Spark, this means writing the intermediates to local disk and, then, moved across the network. Afterwards, a new stage/pipeline can start. Similarly, Hermes first materializes all the intermediates before starting the next stage. Fortunately, Hermes does not write to disk, but to shared memory (via our abstraction layer Memcom) from which the other nodes can pick up blocks of interest. Furthermore, Hermes does not need to load the table from Object storage, but instead can read it from nodes within the cluster, but will preferably read local data. Besides, Hermes execution engine seems more efficient than Spark's, it is more comparable to Photon [7] – Databricks' new engine for certain workloads in Spark – written in C++ using Vectorized Execution [9].

Redshift [6, 10]. A Redshift cluster comprises a single coordinator (leader) node and multiple workers (compute nodes). Data is stored on Redshift Managed Storage, backed by Amazon S3, and cached in compute nodes on locally-attached SSDs in a compressed column-oriented format. The leader node receives the query and parses, rewrites, and optimizes. Redshift's cost-based optimizer includes the cluster's topology and the cost of data movement between compute nodes in its cost model to select an optimal plan. Redshift generates highly optimized C++ code that interleaves multiple query operators in a pipeline using one or more (nested) loops, compiles it, and ships the binary to compute nodes. In Hermes, the leader Node receives the query and decides how the query plan will be fragmented, each query plan fragment becomes a distributed task and is added to a task queue. Following that, each Worker Node picks a task (logical plan) from the queue and generates the physical plan to be executed. The leader does only the necessary coordination to organize and prepare tasks as quick as possible, once done, the leader joins the workers on processing distributed tasks. Once all the workers finish all the tasks for a query, the leader finalizes the query and delivers the result.

TiDB. TiDB [18] is a HTAP (Hybrid Transactional/Analytical Processing) and is formed of three main components: *Distributed Storage Layer* contains the row store (TiKV) and column store (TiFlash). *Placement Driver* manages physical placement of Regions (range partitioned data) to balance workloads, while also providing unique timestamps used by the system. *Computation Engine* is a stateless and scalable SQL engine, with a distributed query executor and a cost based optimizer. Regions are replicated to provide high availability, and kept consistent using the Raft consensus protocol [24]. TiFlash nodes do not participate in the consensus protocol, instead they receive the logs from TiKV and transform from row into column format. TiFlash contains a stable and a delta storage, similar to Hermes, where updates are appended to the delta in order of arrival and are later merged into stable storage. Hermes does not contain a Placement Driver component, the placement of data is done, so far, based on which node contains the least amount of data, via our abstraction for shared memory (Memcom).

Presto/Velox. Presto [27] is a massively distributed OLAP engine, developed at Meta. Initially developed in Java, but has evolved into a more sophisticated C++ engine with Velox [25]. Presto supports caching at multiple levels in the system to avoid data movement through the network as much as possible; adaptive filtering techniques like: filter reordering and dynamic join filtering; materialized views to avoid recalculating frequent plans; multiple query coordinators; and recoverable execution. On the other hand, in Hermes, so far there was no need for multiple query coordinators, although it is in our roadmap. Small tables are cached in all worker nodes, and intermediate results are kept local, which enables local scans on a best-effort basis.

VectorH. VectorH [12] extends Vector/Vectorwise [8] to multiple nodes. VectorH uses disaggregated storage (on HDFS) but tries to co-locate scans with data. Query execution in VectorH communicates via MPI (Message Passing Interface) which has one major flaw: One failing node, will take the whole cluster down. It can be restarted, but after some period. Hermes also co-locates scans with data. In Hermes' case Memcom handles that need to be shared between nodes as well as which nodes are part of the cluster. Furthermore, VectorH uses static partitioning and assignment, whereas Hermes uses dynamic "morsel-driven" assignment [19]. Unlike Hermes, VectorH pipelines execution across Exchange [16] operators, i.e. that producers and consumers, overlap. In practice, this leads to less memory footprint for intermediates (and lower allocation costs). So far, we have not yet noticed memory usage for intermediates becoming a problem, but are planning to implement pipelining in the future anyway.

3 HERMES: THE ACCELERATOR FOR MYSQL

We designed Hermes [17] as an accelerator for analytical workloads (for MySQL). On the high-level, it consists of a Delta store, optimized for OLTP, and a compressed columnar Main store – optimized for analytics. Furthermore, queries in Hermes will see a consistent view over MySQL data (and functions will behave the same).

Data Stores. Both, Delta and Main, data stores are depicted in Figure 2. Internally, the Delta store consists of several spaces that (1) map row ids to pointers (`RowIdMap`), (2) the "table" (`RowSpace`), (3) a table of undo images that allow replaying older versions (`UndoSpace`) as well as a heap for variable sized data (not depicted).

In the main store, a table is basically a table of meta data (`MetaTable`) where each record refers to a row group. For each row group, the meta data includes statistics and versioning information as well as a pointer to the actual row group data. We only use the row group as a logical construct to partition the table into chunks that we can efficiently compress and update in bulk. Internally, we can store columnar subsets of the row group (PAX [1] groups). Normally, when creating row groups, we try to keep all columns in the same PAX group. However, very wide rows can cause us to create multiple PAX groups per row group. Furthermore, we also allow custom settings that make Hermes a full column store (where each PAX group only contains one column). Throughout the paper, we assume that a PAX group will cover the whole row group and just refer to them as "row group".

In Hermes, we optimized the Merge processes that migrates data from Delta to Main store into a multi-level approach. Updates first go into the row-oriented Delta. Afterwards, the Merge propagates these changes to the columnar Delta – the second level. The columnar Delta is a structure that is read-optimized but still relatively easy to update. Later, we integrate changes from the columnar Delta into Main store. The two-level Delta has a major advantage in certain workloads (mostly uniform updates) as it allows buffering changes without requiring a full rewrite of the Main store.

Integration with MySQL. To achieve a consistent view across MySQL data, tables are initially ingested into Hermes (i.e. bulk loading). After ingestion, we replay changes from the binary log (binlog). To force transactions to read the – at least logically – same data, we map MySQL timestamps (versions) to Hermes timestamps. If Hermes receives a query where the MySQL version has not been replicated yet, it waits until the replication is done (usually that is in the low milliseconds). It is widely known that MySQL allows sometimes interesting SQL statements (that other database systems would typically not allow, e.g. `select 3+'abc'` returns 3). Hermes imitates that behavior by augmenting the SQL query it receives from MySQL with special casts. Furthermore, we also extended certain expressions to behave similarly on MySQL and Hermes.

Single-Node Improvements. After the previous paper [17], we significantly improved the single-node performance. As Table 1 shows Hermes improved by $1.5 - 2.2\times$ for analytic workloads, $> 10\times$ for ingesting large tables and $> 2\times$ for replicating changes from MySQL. This was achieved by addressing our performance bottlenecks:

- Minimize latching (to increase scalability)
- Improve Update Propagation (faster data movement from Delta to Main store)
- Improve Garbage Collection (to free memory, after Update Propagation)
- Optimize the change propagation pipeline
- Ingestion is heavily reliant on Update Propagation and Garbage Collection.
- Adaptive Bloom filters for joins (led to larger improvements on TPC-H).
- General performance tuning

Efficient DDL. Since schema modifications are important to customers, we added support for adding and removing columns from tables. While removing columns is mainly an operation on the meta data (unmapping a column id), adding columns might potentially involve data movement. To avoid rewriting all row groups and existing deltas, we track the columns and their respective default values inside the meta data (with versions). When a new columns is added, scans will then not read that column from existing row groups and delta, but, instead, generate the default value on-the-fly. To handle updates/DML on the new table (with the fresh new column), adding a column forces creating a new Delta, on top of the existing Deltas, that reflects the current structure of the table (i.e. with the new column added).

Table 1: Since the original Hermes publication [17], performance improved significantly. Change propagation as of March 2026 uses 6 threads, instead of 4 threads.

Name	Mar '25	Mar '26	Speedup
TPC-H SF100 (sec)			
Total time	44.2	20.3	2.2×
Total time (Kunpeng 920 [31])	9.0	5.9	1.5×
Ingestion time	6437.0	505.0	12.7×
TPC-DS SF100 (sec)			
Total time	146	97.3	1.5×
Change Propagation (changes/sec)			
`oltp_write_only`	6320	14240	2.3×
(directly from binlog)	6896	30514	4.4×
`oltp_insert`	12484	36951	3.0×
(directly from binlog)	13912	61585	4.4×

Storage on a single-node is not enough. Unfortunately, for extremely large datasets a single-node with all data stored in-memory is often not enough. Throughout the remainder of the section we explore our journey of how we adapted Hermes to scale to larger datasets. At the center sits our abstraction for shared memory (Memcom) which allows transparently accessing remote memory, if needed (more details in Section 4).

Offloading Main Store to Object Storage (OBS) & Cache Locally. One way to escape the limitations imposed by a single machine, is to offload the Main store to – slow, but cheap – Object Storage (OBS). However, that comes at a price (a) for smaller data sets this requires caching, to avoid costly round-trips to OBS and (b) efficient prefetching to minimize latencies within a query (and across queries). Furthermore, it turned out that row groups (stored as wide PAX groups [1]) are not very well suited to OBS, because accessing just one column forces us to read the whole row group from OBS. Arguably, this is a consequence of our method of caching and data loading. We store a row group as a Memcom variable ("allocation unit") which is also our granularity of data loading and caching. Therefore, for OBS, we modified our Main store to function fully columnar. We already use PAX groups [1], a subset of columns of a row group. Currently, we try store all columns in a PAX group and only split into multiple PAX groups if the rows are too wide. For OBS, we only modify the logic to only store one column per PAX group. The long latencies to OBS require prefetching, we implemented a variant of Cooperative Scans [33]. The basic idea is: With many queries in the system, we can exploit overlapping row groups. Therefore, at every fetch interval, we compute which row

groups are still required by queries, rank them (by utility), prefetch and serve these row groups to the queries whenever they have been fetched. This minimize overall data transfers (from OBS), but causes data (row groups) to arrive out of order.

This works well for medium-large data sets, where most data fits into the local cache. However, for very large datasets, queries will become substantially slower as we (a) reach the bandwidth of OBS and (b) still lack computational power.

Experiment. We ran TPC-H with SF1000 and managed to run SF4000 on our relatively large Kunpeng machine (2 Kunpeng 920B – a variant of the 920 [31] – in total 160 cores with 280 MiB L3 cache and 942 GiB main memory). For SF1000, all queries ran in 93.3 sec in-memory compared to 229.1 sec with data in OBS and a 150 GiB cache. For SF3000 and a 400 GiB cache, all queries ran in 898.11 sec. Compared to PolarDB [3] with four machines (2779.3 sec) Hermes is 3× faster and shows performance comparable to PolarDB with between 8 (1213.6 sec) and 12 machines (837.6 sec). Hermes can also run the larger SF4000, the 22 queries now run in 2469.8 sec. While Hermes is able to run on large scale factors, (a) we consider 40 minutes rather slow and (b) performances scaling is sub-optimal (to say the least), as with 4× more data, we got 10× slower queries.

In summary, the performance on large data sets is rather underwhelming, which has motivated us to allow Hermes to utilize multiple machines for storage as well as query processing.

4 SHARED MEMORY VIA *MEMCOM*

Hermes operates on a shared memory computing model, each node can access the same data as the others. But instead of relying only on hardware solutions, like CXL with cache coherence, we built a lightweight interface to distributed memory: Memcom.

In essence, Memcom provides (a) data access methods, (b) data consistency, (c) data allocation, (d) data reclamation and (e) data locality, over various storage and interconnect types.

Advantages. Having Memcom as a storage layer, provides two major advantages to us: flexibility and quick prototyping on novel internal prototypes.

Regarding flexibility, notable Hermes setups examples include:

- Single-node and in-memory ("Hermes – the single-node Accelerator" [17]),
- Single-node with the Main store on object storage but cached in-memory or
- Distributed with Main store in-memory with nodes connected to each other and partially cached remote data (this work).

But it is easy to see, many other configurations are possible as well.

Besides offering enormous flexibility, Memcom brings another major advantage: We can easily port Hermes to novel internal storage technologies, requiring only minor changes within Memcom. For instance, Hermes has been used to test internal solutions/interface, for instance pmAddr [15][2] or, as used in this publication, HCCS (Huawei Cache-Coherent System), Huawei's equivalent to CXL with cache coherence.

How we use Memcom. We store the Main store (columnar and compressed, optimized for fast analytics), its metadata (table of row groups, with extra metadata, like statistics) and the Delta stores (optimized for transactional updates) in Memcom. In addition to the actual data, we also store the Catalog (being built upon the Delta store) and metadata in Memcom. Furthermore, distributed Hermes uses Memcom to store intermediates and internal states that need to be accessible by all nodes.

4.1 Memcom

Memcom provides a distributed shared memory abstraction that isolates the Hermes query engine from the underlying distributed infrastructure. It builds on the idea that a group of application processes collaborate to manage a globally addressable pool of resources that can be accessed transparently and efficiently from any node in the system. Resources are identified by a global id, or optionally by global name, scoped to a parent resource, which are specific to each group in the system, ensuring resources from different tenants remain invisible to each other.

Data is stored in user-defined variables allocated from typed segments (e.g. in-memory, object storage or SHM, which is locally shared memory) available anywhere in the system, which can be read/written from any process belonging to the same group. In addition, resources such as queues and barriers allow coordination of distributed activities and optimized data transfer.

The programming model is based on an acquire/release interface enforcing release consistency [11], offering fine-grained control over coherence. This is supported by a hybrid communication model that uses shared memory for data transfer where physically possible and message passing for distributed coordination, a design that avoids the contention bottlenecks of purely distributed shared-memory systems as scale increases [26]. This approach reduces the need for explicit data movement logic within Hermes itself, while still exploiting data locality, as operations on remote data can be delegated to the nodes where it resides.

In the current implementation, the group metadata is managed by an external etcd service to which all nodes in the group connect. That is used to store group metadata, consisting of the id maps (which map pages of ids to specific nodes), the named resource hierarchy, the network configuration, etc.

The etcd service is also used for fault detection, as nodes are warned whenever a node is unable to renew its lease in the system and must be considered failed. Then recovery can be started by the leader of the group, taking advantage of the ordered message delivery from Raft [24]. Currently, on node failures the Hermes system is restarted, reloading the data from the last checkpoint available. Memcom can support a more fine-grained recovery, including the ability to protect the segment metadata with a write-ahead log.

Memcom has support for pluggable drivers for segment and transport for new memory technologies or interconnects to device-specific implementations, with no changes required to Hermes itself. These drivers present a uniform API that abstracts the complexities of consistent memory access and communication across different infrastructures, from shared memory to message-based protocols.

One of Memcom API's goals was to allow the implementation to fully explore the underlying system with minimal overhead above the basic technologies. We carefully designed it in such a way that we can avoid data copying and dynamic allocations in many

[2]Not as part of the cited publication

scenarios. That is important on modern transports with heavily-optimized networking stacks, but it becomes crucial on systems that do not involve traditional communications (e.g. memory fabrics).

In total, Memcom provides a flexible development environment that maintains safety, performance, and portability while enabling Hermes to leverage emerging hardware with memory fabrics, low-latency networking, and near-memory processing.

Data Access. Data within variables is accessed using the methods `mc_acquire` and `mc_release` that allow various locking and access modes. In particular, we allow locking $\in$ {none, shared, exclusive} and access $\in$ {read-only, write-only, read-write}. Memcom transparently handles all the details (e.g. transfer, cache data, ensure data is written back) behind the scenes.

`mc_acquire` requires that the global resource id is located in the local resource cache, which contains read only metadata on each resource that is touched locally. If the resource is not cached, Memcom finds the resource segment, which can be local or remote. From the resource segment, we obtain the required metadata, particularly the location of the resource metadata and the location of the resource data, and cache it. If the resource location itself is unknown, that information needs to be obtained from etcd.

After the metadata is available, `mc_acquire` locks the resource via a multiple-reader/single-writer latch (part of the resource metadata). In case it is a local resource that corresponds to a local atomic operations. Then, the pointer to the data location is returned and can be used like a normal local memory pointer. The same pattern is maintained if the data is remote, but in this case a temporary location is allocated and the data is copied to it. Generally, we try to avoid latches, either by having a single common latch for many resources, or by using read-only variables. After the work is done, `mc_release` is called, which will update the remote memory and release the local copy if it was created before, and it will release the latch if that was set.

Further, Memcom also allows atomic operations such as 64-bit load, store, compare-and-swap, fetch-and-add, but also 128-bit double-word-compare-and-swap.

Example. Memcom acts as a (mostly) standalone library and can be reused across projects. From higher layers, it can be used as in Listing 1, which creates and modifies a simple variable.

First a process joins a group (line 2). Afterwards, we can create segments that holds data/variables. In line 4, we create an in-memory segment located on the current process with a unique name. If names overlap with another segment, we gain shared access to it. Then, we create a anonymous variable in the segment (lines 6-7). This will force Memcom to allocate the space requested. Note that variables can also have names. Overlapping names allow shared access. However, often this is not advisable because it will lead to significant overhead, as we use `etcd` to store the names and lookups are very slow (double digit milliseconds).

Now, we access part of the variable. We first define the range we want to access (line 12). Then, we "acquire" access to `var` (lines 13-14). `mc_acquire` takes care of locking, if the variable is located somewhere else, fetching the variable (parts thereof, to be precise). Afterwards, we get the pointer associated to the fetched range (line 18) and try to modify the stored data. If the value is small enough, we modify (line 22) and afterwards release the variable, and its

Listing 1: Memcom example: Create a variable and update part of it.

```
1  // Process joins a group/cluster
2  mc_id process_id = mc_join("vldb-hermes");
3  // Create in memory segment
4  mc_id_t seg_id = mc_segment(process_id, "vldb" /* unique name */,
5          SEG_MEM /* type */, 0 /* or remote process_id */);
6  // Create variable
7  mc_id_t var_id = mc_variable(0 /* parent */, nullptr /* no name */,
8          64 /* #bytes */, seg_id /* home segment */);
9
10 // Access the [0, 8) bytes of 'var',/
11 // acquire takes care of locking, data movement ...
12 mc_range range {0 /* offset */, 8 /* length */}
13 auto local = mc_acquire(var_id, &range, 1 /* #ranges */,
14         AM_READ_WRITE /* exclusive lock with rw access */);
15
16 // Get pointer to buffer with index 0
17 // (here we only acquire 1 range)
18 void* buffer = mc_local_ptr(local, 0 /* range index */);
19 uint64_t *p = (uint64_t*)(buffer);
20 bool change = *p < 1000;
21 if (change) {
22         *p+=2; // Modify buffer
23         mc_release(local); // Release access
24 } else {
25         mc_discard(local); // No changes, no write back
26 }
27
28 mc_remove(var_id); // Delete variable
29 mc_remove(seg_id); // Delete segment
```

acquired ranges. Memcom will then release the lock and write the data back to its origin, if needed. If we did not modify the variable, we tell Memcom that there were no changes (line 25), which will release the lock but skip the write-back.

At the end, we remove, both, variable and segment (lines 28-29). Note that will not immediately trigger Memcom to free the associated space, but rather marks the objects as delete such that they can be garbage collected later.

4.2 Base Tables

Base tables in Hermes consist of an update-optimized Delta table and a read-optimized and Main table. We ensure that the Delta always contains $\leq$ 10% of the total number of rows of the table. Consequently, most rows will always reside in the Main table/store. This allows focussing on optimizing access to the Main stored to improve analytic performance.

Main store. As can be seen in Figure 2b, a table in the Main store consists of multiple row groups. Each row group is assumed to be one Memcom variable but is allowed to span multiple Memcom variables if required (e.g. if a single row is too large). The row groups are created through a Merge process that propagates data from the Delta (via a columnar Delta) to the compressed Main store.

When, we create a new row group we decide where it is going to be located, store that information in our meta data and allocate a Memcom variable on the target. We define locality as a pair of NUMA node/zone and process id. To allocate new row groups, we have to decide segment to allocate the Memcom variable in (as before). However, we now have multiple lists of segments, one per locality. We choose the locality with most free space, to allocate a new row group. Besides being a rather simple strategy, it provides

balance across all localities[3]. In case a new process joins, it will lack most of the Main store data. With ongoing updates from MySQL, we will continue to propagate data from Delta to Main, which will new row groups on the "newcomer". Obviously, this will lead to delays and a pro-active re-balancing strategy is on our roadmap.

Partitioning. For the Main store we allow hash partitioning by one or more attributes (partition keys). Different partitions are assigned to their own specific locality and are stable across tables. That means, if in table `T1` a row with partition key *i* is assigned to node *j*, then in table `T2` a row with partition key *i* will be assigned to the same node *j*. This guarantees that partitions are colocated.

Immutable Main store. We further optimized access to the Main store in Memcom. Usually, one would use a shared latch with read-only access. However, the Main store rarely changes. The changes can only come from the Merge process. Therefore, after new row groups have been create, we tag them as *immutable* Memcom variables[4]. This brings several major advantages:

- Read accesses do not require latching anymore. In a distributed setup, this minimizes the round trips required to fetch data. Furthermore, this minimizes cache coherency traffic on a single-node (caused by atomic updates to shared memory), especially when access is contented.
- More efficient space reclamation (Memcom compaction). Since immutable variables can just be copied without requiring latching. Once a immutable variable is moved to a new spot, we can just update the meta data to reflect its new position and eventually garbage collect the old spot.
- Caching becomes easy, since we know they will not require full cache coherency.
- Garbage collecting old row groups becomes trivial. Whenever, the Merge process creates a new row group that replaces an older row group. The older row group can just remain in cache and does not forcefully need to be evicted. Since the old row group will eventually not be accessed anymore, it will naturally be replaced over time.

In a distributed system, these advantages are critical. Most notably, immutable variables simplify already complex components, caching and garbage collection, immensely.

Delta store. Besides the Main store, also the Delta store is stored in Memcom. As depicted in Figure 2a, we store each space as a separate Memcom variable, what we mainly do to minimize overhead due to resizing (when the Delta grows). Unlike the Main store, the Delta is not optimized for large reads, but only for updates/DML. Therefore, we ignore locality (when accessing the Delta), we assume that change replication from MySQL (the writer) will always have the Delta local (thanks to Memcom caching it). Since the delta is considerably smaller than the compressed Main store ($\leq$ 10% rows modified, practically closer to 1%), the benefit of locality for reads is negligible.

[3]We assume all localities (NUMA nodes and processes), are equal.

[4]In fact, we initially avoided latching to access to Main store row groups. However, with the introduction of Memcom compaction, this became problematic. Now, we finally returned to latch-free access.

4.3 Intermediates

Since we not only want to expand our data storage but also exploit the additional compute power, we allow parallelizing a query across multiple nodes (intra-query parallelism, see Section 5). Often, this requires data sharing between the participating nodes inside the query which requires us to write these intermediate rows to Memcom and read them afterwards.

Intermediates have three levels: On the top, there is a high-level list of intermediates structures, followed by a thread-local list of blocks, followed by rows stored columnar within blocks.

When writing intermediates, each thread will first determine its locality (NUMA node and process id) and create an intermediate structure (thread-local list of blocks) associated with its locality and register it in the top-level list. Afterwards, each thread serializes its intermediate data into blocks and pushes them into the thread-local list. The layered structures minimizes locking (across threads, nodes ...) as only the top-layer requires global synchronization.

When reading intermediates, each thread will pick up one of the previously thread-local lists, read the blocks and deserialize them back into the data format used within our engine. Afterwards, it will continue until all lists of been consumed.

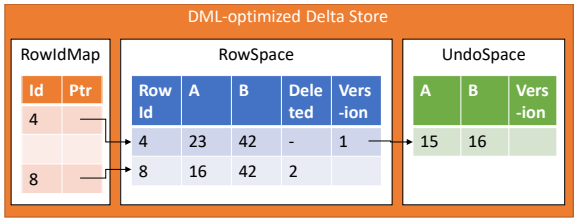
Block sizes. We opt for mostly fixed-sized blocks because they are easily allocated and reused. Usually a fixed sizes in not optimal because it will either be too large or too small. Too large blocks will not only waste space but also network bandwidth (because we need to transfer the full block). Blocks that are too small will lead to sub-optimal read and write performance. To work around this issue, we used two block sizes: We use first a small block size of 32 kiB. After we filled that first block, we assume more data is following and we opt for the large blocks size (8 MiB).

Serialization. We serialize fixed-sized values by copying them into the block. Hermes uses a vectorized engine with selection vectors and serialization also removes the selection vector from the intermediate data by compacting and, therefore, removing all temporarily eliminated/invisible rows. Let us look into serializing variable length values:

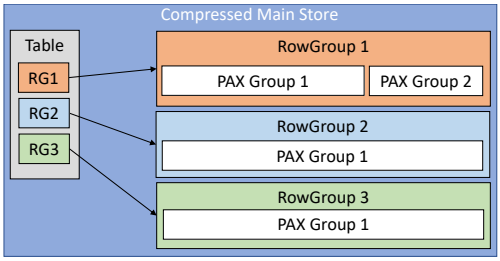
Hermes uses strings as in Umbra [23], which are 16-byte values. Each value contains a 4-byte length followed by either an inlined string ($\leq$ 12 bytes) or a string pointer with an inlined prefix (4 bytes). To serialize strings, we offer three options: (a) if all values have the same size, we serialize the size first, followed by just the values, (b) if all values have a length $\leq$ 12, we serialized all the sizes first, followed by all the values, or (c) we prefix each value with its size. Generally, we opted for lower memory footprint as opposed to serialization/deserialization performance.

5 DISTRIBUTED QUERY EXECUTION

This section describes how query execution works in Hermes' distributed engine. We first describe how query plans in Hermes a different from traditional trees and how we share data (within and across nodes). This is followed by a quick look into how we plan queries for distributed parallel execution and, afterwards, explain the the query is executed in a distributed fashion. During this section, we use Figure 4 as a running example.



(a) Each space of the Delta is stored as a separated Memcom variable. This has the advantage that these variables can grow independently minimizing data movement. Note that we do not specify a locality for the Delta since it is mostly irrelevant for analytic queries (insignificant number of rows stored there).



(b) The Main store is split into row groups. Each row group can contains PAX groups, a collection of columns. Each PAX group is its own Memcom variable and contains one or more compressed columns. The colors represent different localities.

Figure 2: Delta and Main store are stored in (logically) shared memory via our abstraction Memcom. White boxes represent Memcom variables.

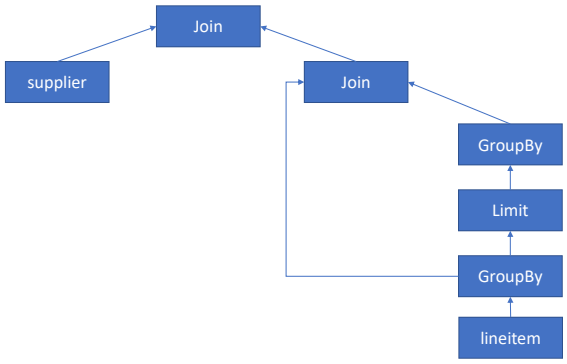


Figure 3: Result sharing inside the query plan: For joins, the right-hand side is probed through the hash table. Extract from TPC-H Q15.

5.1 Query Plans in Hermes

In Hermes, query plans are DAGs (Directed Acyclic Graphs) that allow data sharing between nodes. Figure 3 shows an example where the computed grouping of `lineitem` table is reused on both sides of the join, the result is afterwards joined with `supplier`. This is avoids recomputation, but leads to more complex plans – a DAG as opposed to a tree.

Many systems [4, 12] use *Exchange* operators [16] to parallelize execution on multiple nodes (and sometimes threads). However such *Exchange* operators are only suitable to tree-shaped query plan and is thus not directly applicable to DAGs. The underlying issue is that reuse can potentially happen in arbitrary spots (in the plan), which further complicates the process since all reuses need to be parallelize in the same manner (same partitioning ...), but also might allow reuse across machines. Instead, we introduce a *Portal*-based data sharing and shuffling mechanism and break the query plan into tasks around Portals. Portals are very similar to Exchange operators, but take data reuse into account.

Portal Operators. Portal operators allow data sharing between different parts of the plan. We introduce two operators for storing and retrieving intermediate data:

`Write` stores intermediates in Memcom. In case of partitioned group-by or partitioned join, `Write` partitions the data beforehand. In detail, each thread (on each node) creates a thread-local intermediate structure, to which a pointer is globally shared. Afterwards, threads just directly serialize intermediate data to their thread-local structure. This minimizes interactions with other threads, i.e. across cores or, even worse, across nodes.

`Read` retrieves intermediates from Memcom. It allows two different semantics: read-once and read-many/Broadcast. During a `Read` each thread picks a thread-local intermediate written by a *Write* thread and directly deserializes the data from the original buffer into our data format used within the "single-node"/node-local pipeline (i.e. zero-copy).

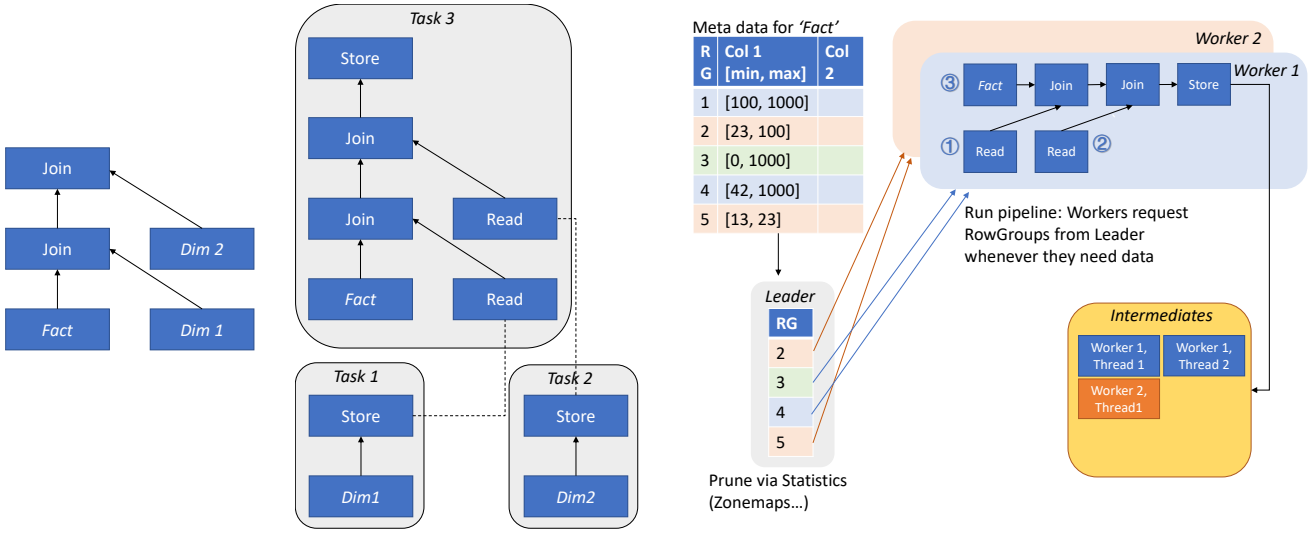
Distributed Scans. Many systems, such as VectorH [12], statically partition data (node 1 scans [0, 10), node 2 scans [10, 20)). However, in practice this tends to be a sub-optimal choice as for various reasons there can be skew between the processing pipelines (e.g. due to interference with other processes or queries). Instead, we chose a dynamic-approach:

When starting a scan. The leader creates a list of candidate row groups, after filtering based on our statistics (most importantly Zonemaps). Based on this list of rowgroups, each thread is technically free to pick any row groups (i.e. basically a Morsel-driven scan [19]). Similar to the original Morsel-driven scan [19], each thread will preferably scan row groups that it considers local, i.e. scans are *local-balanced* and *locality-aware*. In a multi nodes setting with NUMA within nodes, threads will first read NUMA-local data within its node, followed by NUMA-remote data within its node, until it eventually considers helping remote nodes. Consequently, (a) threads will read mostly local data and (b) thread will achieve some balance (faster threads will consume more data).

Since within a node, we use pipelined execution, the scan will effectively load-balance the whole pipeline, until some blocking portal (aka `Write`-`Read` pair).

5.2 Planner

We plan queries in a two-stage process: First, we plan and optimize for the single-node. Afterwards, we rewrite the optimized query into a distributed query using these basic patterns:



(a) Input Query Plan: Fact table joined with two small dimension tables.

(b) Distributed Query Plan: Both joins became Broadcast Joins. Task 1 and 2 can be executed independently of each other, whereas Task 3 has to be executed after 1 and 2 (dotted lines represent dependencies).

(c) Physical Execution of Task 3: As preparation, the Leader produces a list of candidate row groups. (1) We read intermediates from Task 1 and build the hash table of join 1, (2) the same for Task 2 and join 2, (3) followed by pipelined execution of the probe pipeline, that dynamically picks up row groups from the Leader. Note that (1) and (2) are done in parallel. Furthermore, all tasks are executed using multiple threads on the worker node.

Figure 4: Left-deep join tree: Probing the large *Fact* table is completely pipelined, hash tables are built for tables *Dim 1* and *2*.

- *Broadcast Join*: The inner/build relation is materialized in Memcom and shared with all nodes. This is useful when the inner relation is small (e.g. when a large fact table is joined with a small dimension table). Furthermore the Broadcast Join allows pipelined execution without materializing intermediates in Memcom. Consequently, when there is a few Broadcast Joins and on base tables, our dynamic Morsel-driven [19] scans allow Hermes to load-balance automatically (faster scan-join pipelines will just consume more data).
- *Partitioned Join*: Both relations are partitioned by hash, each node finds a partition and executes the join on that specific partition. The Partitioned Join is only useful when we do not support Broadcast Joins, which is either for Reverse Joins[5], when the inner relation is large or when both build and probe are pre-partitioned and co-located in the same nodes.
- *Pre-Aggregation & Single-node Aggregation*: We introduce an additional (Pre-)Aggregation on each node. Afterwards we materialize the results, of each pre-aggregation, in Memcom. Then, a single node reads the intermediates from Memcom and computes the final Aggregation. Since, this technique is most effective for only a few groups, we also support partitioning:
- *Partitioned Aggregation*: Before the Aggregation, we partition by the grouping keys. Each node will receive a partition to aggregate locally. This technique is more general, but we only use if for large group-bys.
- *Distributed Scan → Regular Scan*: We replace Distributed Scans with regular scans, when the Distributed Scan would lead to Broadcast Join. The direct effect is that each node will scan the same row groups of the same table (i.e. broadcast). While wasting some resources on repeated preprocessing (e.g. filtering via Zonemaps), this requires less coordination (no need to compute the current Morsel, less scheduling) and less materialization (no `Write` and `Read`).

Applying these patterns creates a basic distributed plan which is, then, optimized by following optimization passes. Most importantly, Hermes has a pass to minimize data movement. Essentially, this pass removes all redundant "shuffles" (`Write`-`Read` pairs) by merging tasks. In particular we remove such shuffles when (a) they are trivial (`Write`-`Read` with one consumer introduced to stich parts together), (b) when data is co-located (tasks operate on the same partition, e.g. Partitioned Join on pre-partitioned table or series of Partitioned Joins on same key) or (c) when the plan is cheap enough (typically, just small base table scans), we duplicate the plan (child of `Write`) for each `Read`. any data movement in between.

Cost-Based Optimizer. The decision of which patterns to apply, and whether to parallelize a query across nodes, is made cost-based. Instead of opting for complex cost function (e.g. number of bytes of a relation), we chose the estimated cardinality.

Example. In our running example, the planner takes the input plan (Figure 4a) and transforms it into a parallel and distributed plan (depicted in Figure 4b). This leads to the plan being partitioned into fragments (tasks). In our example, both joins became broadcast joins as the build sides were small (small dimension tables joined

[5]Reverse Joins still join build and probe relations. However, they assume that the build relation is larger. As a consequence, we still build a hash table for build. But mark matching rows in the hash table. After the marking, we scan the hash table and emit the rows with matches.

with large fact table). This leads to three tasks with dependencies. Task 1 and Task 2 each scan a dimension table and materialize the result in Memcom (via `store`). Task 3 requires Task 1 and 2 being completed (hence the dependency edges between tasks). Once Task 1 and 2 are completed, we can schedule Task 3, which will then execute the two joins, after reading the broadcasted tables from Memcom, and store the intermediates in Memcom.

5.3 Execution & Coordination

After the distributed plan is created, the leader node schedules the tasks for execution: Each task contains the plan of the query fragment as well as identifiers for the stitching points (i.e. `ReadWrite`). The worker nodes only pickup tasks, process them and forward the status to the leader node (successfully completed or an error code). The leader, then, decides how to proceed. Assuming no error, whenever the dependencies of tasks are fulfilled, these tasks are being scheduled for execution.

In our running example, the Leader first schedules Task 1 and Task 2 (in Figure 4b) as both tasks have no dependencies. After both tasks have finished, the Leader removes the, now fulfilled, dependencies to Task 3. In turn, this makes Task 3 ready to be scheduled (as is does not have any dependencies anymore).

In Figure 4c, we take a deeper look into how Task 3 is executed. Since the Leader has scheduled Task 3, workers now try to pick up sub-tasks of that task. In our example, Worker 1 and 2, both execute Task 3 using as many threads as the regular Hermes engine wants to employ. Each worker first analyzes the plan for dependencies. Here, they can read both (broadcasted) dimension tables from shared storage, and build the hash tables for the joins. Afterwards, we reach the pipeline that scans the *Fact* table and joins it with the dimension tables. This initiates a distributed scan, each worker will try to pick up row groups from a global list created beforehand. This list is created by the Leader after planning the parallel query and incorporates early pruning techniques (most importantly filtering via Zonemaps). Then, the workers pick up a row group, scan the row group, compute the result for the joins and writes the result into shared memory. Afterwards, the next task can start.

5.4 Work Placement

Another important topic is work placement, because it avoids data movement by moving processes to the data. For Hermes, this is relevant for main store data and intermediates. For Main store data, we either assume the table is small, allowing it to be cached on all nodes, or that the table is large and is partitioned and colocated in each node. This can happen in two flavors: The table can be pre-partitioned (by defining partitioning keys) or we dynamically partition the data round robin (with locality, i.e. morsel-driven). For intermediates, data is always fully shuffled between nodes and we assume no work placement is possible.

6 DYNAMIC PUSH-DOWN

Often, joins can introduce new predicates to the outer relation: For instance, supposed we have a join between two large tables `orders` as the inner/build relation, and `lineitem`, as the outer/probe relation. Since this is a large join, broadcasting the inner relation is not an option and both relations are partitioned and sent over the network. The join can be selective, filter rows from the probe side. If we have filters on the build relation, we try to push them into the probe relation (Sideways information passing). More generally, in larger join trees we can push predicates to the scan of the probe relation. Particularly, in complex queries with data movement this is effective, because it allows pruning rows at the scan – before data is sent.

Table 2: Hermes significantly outperforms widely-used open-source systems. TPC-H scale factor 100.

Query	Hermes (sec)	SparkSQL (sec)	SparkSQL / Hermes	TiFlash (sec)	TiFlash / Hermes	DuckDB (sec)	DuckDB / Hermes
1	0.21	9.62	46	1.17	6	0.54	3
2	0.18	8.58	48	1.02	6	0.47	3
3	0.25	20.96	84	2.36	9	1.06	4
4	0.26	16.61	64	9.82	38	0.49	2
5	0.24	20.48	85	4.93	21	0.69	3
6	0.08	6.55	82	0.38	5	0.45	6
7	0.27	14.55	54	2.46	9	1.04	4
8	0.22	14.52	66	4.51	20	0.82	4
9	0.70	17.79	25	13.22	19	2.04	3
10	0.46	13.06	28	2.15	5	1.21	3
11	0.14	9.55	68	7.05	50	0.18	1
12	0.17	11.65	69	0.90	5	1.88	11
13	0.63	12.80	20	2.34	4	1.29	2
14	0.18	16.02	89	0.54	3	0.65	4
15	0.23	19.61	85	0.62	3	0.48	2
16	0.28	4.39	16	0.50	2	0.57	2
17	0.18	24.99	139	7.60	42	0.64	4
18	0.47	45.22	96	8.13	17	1.37	3
19	0.30	6.81	23	0.79	3	2.61	9
20	0.30	12.59	42	1.06	4	0.91	3
21	0.86	43.92	51	6.16	7	2.85	3
22	0.19	10.40	55	0.22	1	0.71	4
Total	6.80	360.67	53	77.93	12	22.93	3

7 EVALUATION

We first compare Hermes with popular open-source systems. Afterwards, we show how well Hermes performs on larger dataset (TPC-H SF1000). This is followed by an experiment that investigates scalability of certain queries. Then, we explore the effect of the interconnect (and software stack) on Hermes. Finally, we experimentally compare the performance between the old state of Hermes – as of time of the first publication [17] – to now.

7.1 Distributed TPC-H SF100

To get an idea about the performance of our distributed engine, we compare Hermes to two widely-used systems for OLAP and one for HTAP (Hybrid transactional/analytical processing). In particular, we compare to SparkSQL [5], an analytical system with extremely large footprint in the data science field, and DuckDB 1.5.2 [14], a popular single-machine analytical DBMS, for OLAP and TiDB [18], a close open-source competitor in the HTAP field. For TiDB, we used TiFlash, which is the backend optimized for analytics.

Distributed Setup. For the distributed benchmarks, we used 4 machines: Each machine has 2 Kunpeng 920B with a total of 160

cores and 512 GiB main memory split into 4 NUMA regions. Of the 512 GiB per node, 256 GiB is special memory sharable via HCCS (Huawei Cache-Coherent System, Huawei's equivalent to CXL with cache coherency). Each Hermes VM is assigned 64 GiB for the operating system and Hermes, and 200 GiB of sharable memory, as well as 40 cores. We run the Hermes VMs on the first NUMA node of each machine, and leave the rest of the machine idle. A notable exception is the machine which will also run `etcd` and MySQL. To minimize interference, `etcd` and MySQL run in other NUMA nodes. Furthermore, we partitioned `lineitem` and `orders` on `orderkey`. Spark and TiDB use the same setup (same machines, one NUMA node per machine). We run DuckDB on a single-machine, comparable to one node of our distributed setup.

Open-Source Systems. We installed all three systems in our distributed setup (described above) with 4 nodes. Then, we loaded TPC-H scale factor 100, then, we warmed up the caches by running the 22 queries. Afterwards, we ran the 22 queries and measured the latency (hot power run). Table 2 shows the results. Across the board, Hermes significantly outperforms all three systems.

Hermes significantly outperforms SparkSQL by roughly 53× (total time). In Q17, Hermes outperforms SparkSQL by 139×. SparkSQL features slow query execution for various reasons: (1) slow engine as it relies on Resilient Distributed Datasets (RDDs) [32] to provide basic operations on data. To get high performance for basic operations, one would need the JVM (Java Virtual Machine) to JIT the whole pipeline. However, this is often rather complex code, making this a hard job for the JVM. Note that this horrendous performance also led to the development of Photon [7] – a new engine for Spark. Furthermore (2) RDDs might lead to repeated materialization (and serialization) and data shuffling. Of course, the optimizer would try to minimize that. On the other side, we have Hermes, which is optimized for single-node performance (low overhead data processing via Vectorized Execution [9]) extended with distributed execution that tries to minimize data movement and optimize for locality.

Compared to TiFlash (TiDB optimized for analytics), Hermes, at worst, outperforms by 1.2× (in Q22). But overall outperforms by 9×. In certain queries (Q4, Q5, Q8, Q11, Q17), Hermes outperforms by ≥ 20×. It is not inconceivable that TiDB does inferior query plans in these queries. However, also on most other queries Hermes often outperforms by ≥ 2×.

Compared to DuckDB, Hermes outperforms by up to 11× and by 3.4× (rounded to 3×) overall. While we acknowledge that comparing a 4 node setup to a single-node setup is unfair, but it helps establishing an independent baseline. In other words, distributed Hermes outperforms a fast single-node system.

7.2 Distributed TPC-H SF1000

We now move to larger datasets and measure how well distributed Hermes performs on scale factor 1000 using 4 nodes with distributed setup described in Section 7.1. To compare to PolarDB IMCI [30], we used their published query latencies [2, 3]. While we used the "old" query latencies from the website [2] as internal references, PolarDB recently released an updated set of performance numbers [3]. We compare to both, "old" and "new" performance numbers. Table 3 shows the results.

Table 3: Hermes outperforms PolarDB IMCI on their updated numbers [3] and more so on the older performance numbers [2]. TPC-H scale factor 1000.

Query	Hermes	IMCI [3]		IMCI-old [2]	
	(sec)	(sec)	$\frac{\text{IMCI}}{\text{Hermes}}$	(sec)	$\frac{\text{IMCI}_{\text{old}}}{\text{Hermes}}$
1	1.97	6.06	3.1	14.64	7.4
2	0.66	0.42	0.6	5.35	8.1
3	2.52	2.65	1.1	9.17	3.6
4	2.38	0.74	0.3	4.64	1.9
5	1.73	1.69	1.0	10.09	5.8
6	0.74	0.59	0.8	1.72	2.3
7	2.92	1.60	0.5	9.59	3.3
8	1.60	1.14	0.7	7.49	4.7
9	8.99	18.66	2.1	58.67	6.5
10	5.05	4.73	0.9	35.91	7.1
11	0.51	0.63	1.2	6.37	12.5
12	1.52	1.83	1.2	5.82	3.8
13	8.03	6.64	0.8	63.84	8.0
14	2.03	1.03	0.5	3.82	1.9
15	1.72	1.85	1.1	8.47	4.9
16	2.30	1.77	0.8	9.95	4.3
17	1.39	10.13	7.3	24.41	17.6
18	5.67	18.37	3.2	63.64	11.2
19	3.01	8.75	2.9	21.56	7.2
20	1.27	2.43	1.9	7.25	5.7
21	8.88	3.30	0.4	22.46	2.5
22	1.78	2.64	1.5	10.16	5.7
Total	66.7	97.6	1.5	405.0	6.0

Setup. We use the same setup as in Section 7.1. Note that we compare similar setups, for Hermes, 40 cores on Kunpeng with 264 GiB (200 + 64 GiB), to PolarDB IMCI, with 32 x86 cores and 256 GiB main memory. Using more, but slightly slower, cores (ARM vs x86), gives similar overall compute power, but exaggerates scalability bottlenecks (not in our favour).

Compared to the old IMCI, Hermes outperforms by roughly 6× (total), and in some queries by more (e.g. Q1 7.4×, Q17 17.6×).

If we compare to the updated performance numbers, the picture is more mixed. However, Hermes outperforms in total by 1.5×. Interestingly, we outperform on Q1 which is a rather simple single-table query where most work can be performed locally (each node can pre-aggregate on local data). This suggests that PolarDB's approach might be suboptimal, whereas Hermes locally pre-aggregates and mostly reads local data. On the other hand, certain queries are significantly slower in Hermes (e.g. Q2, Q7 and Q14), which suggests the Hermes probably has a suboptimal query plan, because these queries are rather complex.

7.3 Scalability

We investigate the scalability, by comparing single-node against 4 nodes, on a subset of the TPC-H queries. We used the same setup as described in the previous section. Table 4 visualizes the results.

If we look at the single table queries, Q1 and Q6, we notice that Q1 scales almost linearly (3.5× vs. 4×). Scans should read mostly local data and the queries should pre-aggregate in each node (Q1 produces 4 groups, Q6 is a global aggregate). Q6, however, scales worse (2.7× vs. 4×). The noticeable differences between Q1

Table 4: Scaling – to just 4 nodes – requires partitioning (i.e. partitioned join and group-by) and partitioned tables.

Query	1 Node (sec)	4 Nodes (sec)	Speedup $\frac{1 \text{ node}}{4 \text{ nodes}}$
Single table			
1	6.87	1.97	3.5
6	1.98	0.74	2.7
Joins			
9	29.53	8.99	3.3
w/o part. tables		16.54	1.8
w/o part. join/group		16.73	1.8
18	19.35	5.67	3.4
w/o part. tables		8.60	2.3
w/o part. join/group		17.16	1.1
Total	157.6	66.7	2.4
w/o part. tables		79.5	2.0
w/o part. join/group		99.3	1.6

Table 5: Hermes on Ethernet with TCP: Adding more nodes, decreases performance. With a faster network stack, adding nodes improves performance. TPC-H total time on SF100.

#Nodes	Intel/TCP	Kunpeng/HCCS
	Total Time (sec)	
1	10.8	14.1
2	15.4	13.0
3	18.3	10.6
4	20.2	7.8

and Q6: Q1 has more computations (filter, arithmetic and group-by into 4 groups) whereas Q6 just has selective filters and one multiplication and global `SUM` aggregate. This indicates the that with less computation our engines scales worse, further indicating that our scans probably require some more tuning (e.g. larger morsel size) or improvements (possibly Memcom is the issue).

Moving to queries with joins, we see that, without partitioning queries scale less well (1.1× – 1.8×). With partitioned group-by and joins, we can improve scalability to 1.8× – 2.3×. To further improve, we can pre-partition tables on common keys. Here we partitioned `lineitem` and `orders` on `orderkey`, which causes some partitioned joins and group-bys to become co-located and lead to 3.3× – 3.4×.

7.4 Hermes on Ethernet with TCP

Very often systems are forced to use standard protocols and interconnects to communicate. However, they tend to lead to suboptimal performance. Commonly, the standard TCP stack (a) goes through the operating system kernel, leading to very costly system calls, (b) multiple layers to (c) eventually use a proven, but suboptimal, interconnect. Modern approaches such as HCCS try to minimize the overhead by providing interfaces that provide a short-cut to the optimized network hardware bypassing points (a), (b) and (c). HCCS features latencies as low as 101 nanoseconds (local access, local across NUMA: 122-129 ns, cross machine: 257-285 ns) compared to 40 us for TCP.

In this experiment, we test Hermes on a "standard setup" which is 4 Intel VMs connected via Ethernet. Each VM has 2 Intel Xeon Platinum 8378A with 32 cores each (64 SMT cores) and is configured with 504 GiB of main memory. The VMs are connected via 2 ethernet network adapters with 100 GBit each. Hermes/Memcom has been configured to use TCP instead. We compare to our distributed setup, described in Section 7.1, without partitioned `lineitem` and `orders`.

Table 5 shows the results. Using a slow network stack leads to negative scaling, where adding more resources leads to worse performance. If we use the same setup as Section 7.1 which has slower CPUs, but faster interconnect (and heavily-optimized network stack), adding nodes improves query performance.

7.5 Single-Node Hermes

To showcase the improvements on our single-node Hermes [17], we compare the branch after submission of the Hermes paper, with the current 'master' branch (as of March 2026). We tested how Hermes performs on a TPC-H and TPC-DS power run (i.e. all queries sequentially), how fast tables are ingested from MySQL (bulk loading from MySQL until the table appears in Hermes) as well as how fast changes (from MySQL) can be applied (Change Propagation).

To measure the Change Propagation, we used two tests of SysBench, a widely-known and -used benchmark: `oltp_write_only` and `oltp_insert`. `oltp_insert` just consists of an `INSERT` whereas `oltp_write_only` consists of a sequence of an insert, delete and 4 updates. Further we measured the performance in two scenarios: With and without MySQL being active. In the first case a transaction runs on MySQL and changes are replicated to Hermes, whereas in the second case we just replicate from the binlog.

We measured these numbers on a cloud VM with two Intel Xeon Gold 6266C and a reported total of 32 cores and 128 GiB of main memory. A notable exception are the numbers on Kunpeng which we measured on a dual socket Kunpeng 920B with 160 cores (in total) and a total of 512 GiB main memory. Table 1 shows the results, which we summarized in detail in Section 3. Overall, we achieved significant improvements by fixing bottleneck after bottleneck.

8 CONCLUSION

Since MySQL is very slow for analytical workloads, we initially proposed Hermes as a single-node accelerator for analytics. However, a single-node with storage in main memory quickly reaches its limits. We explored offloading the majority of data to Object Storage. While larger data sets worked (e.g. TPC-H SF4000), query performance was underwhelming. This required us to rethink our approach of only using a single-node.

In this work, we introduced our distributed data storage (virtually shared memory via an abstraction layer) as well as our distributed query execution framework. Our scans mostly read local data which avoids excessive data transfers across the network and further automatically load-balance (on row group granularity). Furthermore, our cost-based planner supports many patterns to parallelize queries across nodes (machines).

Experimentally, we show that our approach beats widely-used open-source systems (SparkSQL [5] and TiDB [18]) by a wide margin (≥ 11×). Compared to state-of-the-art PolarDB-IMCI [30], Hermes still significantly outperforms (by 1.5×).

Future Work. We plan to improve the scalability and improve robustness execution against skew and less reliant on, often highly inaccurate [20], cardinality estimates.

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] Alibaba Cloud. 2026. IMCI performance. https://www.alibabacloud.com/help/en/polardb/polardb-for-mysql/imci-performance. [Online; accessed 1-Feburary-2026].

[4] Alibaba Cloud. 2026. Use multi-node MPP to accelerate mass data analysis. https://www.alibabacloud.com/help/en/polardb/polardb-for-mysql/user-guide/use-multi-machine-mpp-to-speed-up-mass-data-analysis. [Online; accessed 4-February-2026].

[5] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*.

[6] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD*. 2205–2217.

[7] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD*. 2326–2339.

[8] PA Boncz and M Zukowski. 2012. Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27.

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

[10] Mengchu Cai, Martin Grund, Anurag Gupta, Fabian Nagel, Ippokratis Pandis, Yannis Papakonstantinou, and Michalis Petropoulos. 2018. Integrated Querying of SQL database data and S3 data in Amazon Redshift. *IEEE Data Eng. Bull.* 41 (2018), 82–90.

[11] John B Carter, John K Bennett, and Willy Zwaenepoel. 1991. Implementation and performance of Munin. *ACM SIGOPS Operating Systems Review* 25, 5 (1991), 152–164.

[12] Andrei Costea, Adrian Ionescu, Bogdan Răducanu, Michał Switakowski, Cristian Bârca, Juliusz Sompolski, Alicja Łuszczak, Michał Szafrański, Giel De Nijs, and Peter Boncz. 2016. VectorH: Taking SQL-on-Hadoop to the Next Level. In *SIGMOD*. 1105–1117.

[13] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, V. N. Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. *SIGMOD* (2016).

[14] DuckDB. 2026. DuckDB. https://duckdb.org/. [Online; accessed 06-May-2026].

[15] Amit Golander, Shai Taharlev, and Yigal Korman. 2022. pmAddr: a persistent memory centric computing architecture. In *SYSTOR*.

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

[17] Tim Gubner, Rune Humborstad, and Manyi Lu. 2025. Freely Moving Between the OLTP and OLAP Worlds: Hermes – an High-Performance OLAP Accelerator for MySQL. *PVLDB* (2025).

[18] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *PVLDB* (2020).

[19] 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*.

[20] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.

[21] Feifei Li. 2023. Modernization of Databases in the Cloud Era: Building Databases that Run Like Legos. *PVLDB* 16, 12 (2023), 4140–4151.

[22] Heng Liao. 2025. UB-mesh: An New Interconnection Technology for Large AI SuperNode. In *2025 IEEE Hot Chips 37 Symposium (HCS)*. IEEE Computer Society.

[23] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[24] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *USENIX ATC 14*. 305–319.

[25] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *PVLDB* 15, 12 (2022), 3372–3384.

[26] Michel Raynal and Jiannong Cao. 2019. One for all and all for one: Scalable consensus in a hybrid communication model. In *ICDCS'19*. 464–471.

[27] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, et al. 2023. Presto: A decade of SQL analytics at Meta. *SIGMOD* (2023).

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

[29] 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*.

[30] 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).

[31] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The First 7-nm Chiplet-Based 64-Core ARM SoC for Cloud Services. *IEEE Micro* 41, 5 (2021), 67–75.

[32] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*. 15–28.

[33] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *PVLDB*. 723–734.