Hermes: Off-the-Shelf Real-Time Transactional Analytics

Elena Milkai
University of Wisconsin-Madison
milkai@wisc.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

Jignesh M. Patel
Carnegie Mellon University
jignesh@cmu.edu

ABSTRACT

Many modern applications require real-time analytics, where analytical processing (AP) workloads needs access to the latest data updates from a transactional processing (TP) engine. However, managing separate TP and AP engines across teams complicates achieving real-time analytics without switching to specialized HTAP systems. To address this challenge, we introduce *off-the-shelf real-time analytics*, a system design that leverages the existing TP and AP engines to provide (1) the latest transactional updates for analytical queries and (2) support for efficient *transactional analytics*–transactions that combine transactional logic and analytical queries within a single ACID transaction–at various isolation levels. We demonstrate this concept with a new service called *Hermes*, which acts as a middleware that merges log records with analytical reads without altering existing engines. Our evaluation utilizes two AP engines, *FlexPushdownDB* and *DuckDB*, with *MySQL* as the TP engine. Using the *HATtrick* benchmark and a new workload called *Transactional Analytics Workload* (TAW), we compare Hermes with the leading HTAP solution, *TiDB*. Our results indicate that Hermes performs comparably to current HTAP solutions for real-time analytics and surpasses them by 3× in transactional analytics performance.

PVLDB Reference Format:
Elena Milkai, Xiangyao Yu, and Jignesh M. Patel. Hermes: Off-the-Shelf Real-Time Transactional Analytics. PVLDB, 18(8): 2334-2347, 2025.
doi:10.14778/3742728.3742731

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/elenamilkai/Hermes_layer.git.

1 INTRODUCTION

Real-time data analytics is increasingly critical in modern applications [36, 39, 64] such as fraud detection, dashboarding, healthcare, and cluster monitoring. These applications require analytical processing (AP) to read fresh updates from the transactional processing (TP) engines in real time. It is commonplace for organizations to maintain multiple independent and heterogeneous TP and AP engines [57, 66]. These engines, potentially managed by distinct teams, cater to specific use cases. Providing real-time analytics in such an architecture is challenging due to the need for synchronization.

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742731

The state-of-the-art solution for achieving real-time analytics over fresh data is through hybrid transactional/analytical processing (HTAP) systems [25, 29, 31, 33, 37, 45, 50, 51, 54, 55, 61, 63, 65]. However, existing HTAP systems face a fundamental limitation:

Compulsory migration. HTAP solutions tightly couple AP and TP for high performance and fresh queries. Consequently, organizations must transition from their current TP and AP engines and migrate all data functions into a new HTAP database. This incurs significant cost and management overhead [9, 71]. Moreover, this migration may not be feasible if the HTAP engine fails to support essential functions provided by the current TP or AP engines. Supporting all functions from diverse engines into a single HTAP system is a daunting challenge.

To address this limitation and build a system that achieves real-time analytics using the existing TP and AP engines already deployed in an organization, we must tackle two key challenges:

- **Challenge 1: Support for pluggable engines.** We need a solution that simplifies the integration of *most* TP and AP engines that meet certain specific requirements, acknowledging that it is challenging to generalize for all engines. Our goal is to minimize or, ideally, eliminate the need for modifications within these engines by creating a solution compatible with foundational principles across TP and AP engines, which we discuss in Section 3.2. The optimal approach would involve implementing real-time analytics functionality externally, reducing the need for extensive alterations to individual engines. This approach enhances adaptability and seeks a solution that can be generalized across a wide range of engines, while recognizing the limitations in achieving complete universality.
- **Challenge 2: Efficient True HTAP Transactions.** *True HTAP transactions* [57] are transactions in which transactional logic and analytical queries are processed within a single ACID transaction. This capability simplifies application development [30] and enables new application scenarios [41, 75], which we discuss in detail in Section 4.3. In this paper, we refer to this capability as *Transactional Analytics* and label such a transaction an *Analytical Transaction*. Achieving efficient execution of real-time transactional analytics within a decoupled architecture imposes significant challenges. While distinct specialized engines enable effective execution of each component of these transactions, they complicate the maintenance of data consistency and correctness according to the specified isolation level.

In this paper, we introduce *off-the-shelf real-time analytics*, a novel architecture designed to address these challenges as follows:
Solution to Challenge 1: An off-the-shelf real-time analytics system is constructed using *existing TP and AP engines* with no or minimal modifications to them. The key insight is to introduce a new system layer between the database engines and the storage, which merges the transactional logs with the analytical reads for

analytical queries. Unlike existing HTAP databases that conduct this merging functionality within the database engines, we demonstrate the feasibility of performing this outside the TP/AP engines in a non-intrusive manner. This approach avoids the need for compulsory migration, allowing organizations to continue using their existing TP/AP engines. It also achieves fresh queries and delivers performance that is competitive with current HTAP systems.

Solution to Challenge 2: An off-the-shelf real-time analytics system aims to enable efficient *Transactional Analytics*. For high performance, the analytical components run on the AP engine, while the transactional components run on the TP engine. In these systems, achieving transactional analytics at the requested isolation level involves minimal modifications to the internals of the TP/AP engines. The solution relies on coordination between the off-the-shelf system and TP/AP engines for achieving various isolation levels.

To validate our architecture, we built Hermes, a prototype real-time transactional analytics system for the cloud. Hermes acts as a middle layer between computation and storage, intercepting storage requests from TP (e.g., logging to AWS EBS) and AP engines (e.g., reading from AWS S3). It delivers fresh analytics by merging log data with analytical reads and coordinating with the TP engine to maintain the correct isolation level for the AP engine.

We evaluate the performance of Hermes using MySQL [14] as the TP engine and FlexPushdownDB [73] and DuckDB [60] as the AP engines. Our results show that Hermes integration adds minimal overhead to existing engines. We compare Hermes' performance with MySQL and TiDB [37] on standard HTAP workloads, demonstrating a competitive performance and cost trade-off. To evaluate Hermes on transactional analytics, we developed the *Transactional Analytics Workload (TAW)* by enhancing existing HTAP workloads. Our results show that Hermes outperforms existing solutions (e.g., MySQL and TiDB) by 3×, demonstrating the feasibility of off-the-shelf real-time and transactional analytics.

In summary, this paper makes the following key contributions:

- We introduce the concept of *off-the-shelf real-time analytics*, that allows fresh analytics over existing TP and AP engines.
- We introduce and implement *transactional analytics*, a key feature for current HTAP systems.
- We develop Hermes, a system that enables off-the-shelf real-time analytics and transactional analytics.
- Our evaluation indicates that Hermes exhibits comparable performance to MySQL and TiDB on HATtrick, while outperforming both by 4× on TAW.

The remainder of the paper is organized as follows. Section 2 presents our motivation. Section 3 provides an overview of Hermes. Section 4 delves into how Hermes supports *Transactional Analytics*. Section 6 evaluates the performance of Hermes. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2 DESIGN GOALS

This section introduces the two main goals an *off-the-shelf real-time analytics system* should achieve and discusses the approach taken by existing HTAP systems, thereby motivating our solution.

Goal 1: Support for pluggable engines. Most existing HTAP solutions [22, 25, 25, 29, 31–33, 33, 37, 40, 45–47, 51, 54, 55, 61, 61, 63, 65] require data migration due to tight integration between their computation and storage engines, making transitions to other compute engines or cloud storage challenging. Systems like F1-Lightning [72] and Hudi [4] offer some flexibility with pluggable TP and AP engines, but only support *near real-time* analytics, with updates delayed by ~10 minutes. This makes them unsuitable for applications that require high freshness, which is critical in HTAP.

Key idea 1: Real-time analytics with existing TP/AP engines. An off-the-shelf real-time analytics system achieves real-time analytics on the latest transactional data without requiring engine migration; instead it uses existing TP/AP engines and storage services. This enables users to select optimized engines and storage for TP and AP, avoiding migration efforts. The trade-off is the need for efficient synchronization to maintain high or perfect *freshness*. A system achieves perfect freshness when each analytical query can read changes of all transactions that have committed (i.e., linearizability). The challenge lies in achieving fresh analytics without impacting TP/AP performance or modifying engine internals. In Section 3, we present our architecture, and our evaluation (Sections 6.2, 6.3) shows that our design avoids performance overhead while matching state-of-the-art HTAP systems.

Goal 2: Efficient Transactional Analytics. An analytical transaction consists of both transactional logic and analytical queries that are executed under the same isolation level [57]. The analytical part contains queries of varying complexity, which are significantly accelerated when processed in specialized engines (e.g., columnar databases) ensuring time and cost efficiency. The results of the queries can then be used to perform more operations on the transactional data (e.g., update a table). The analytical query must see the correct data based on the enforced isolation level.

Application scenario: In fraud detection, real-time data on users' recent behavior is analyzed alongside historical data to proactively prevent or address fraud [20]. This analysis must occur at the same isolation level as other operations within the analytical transaction. Upon detecting fraud, the database should reliably revert to a known state by rolling back the transaction or continuing remaining operations, ensuring accurate fraud detection and robust management of potential fraudulent activities.

Efficiently executing transactional analytics is a major challenge for decoupled HTAP systems. Our evaluation (Section 6.4) shows that even leading HTAP systems struggle with performance under such workloads. Additionally, some systems only partially support this functionality, offloading concurrency control to clients—a complex, error-prone approach that adds engineering overhead [30].

Key idea 2: Efficient transactional analytics in off-the-shelf systems. An off-the-shelf real-time analytics system enables efficient transactional analytics by selecting the optimal engine for each workload component—transactional logic is executed in the TP engine and analytical in the AP engine. Its decoupled design, however, challenges consistency and correctness across isolation levels. To address this, analytical queries must operate on an accurate data snapshot. The system achieves this by retrieving log statements that match the correct snapshot and coordinating with the TP engine to enforce it, reducing TP-AP communication. In Section 4, we detail these challenges and our solution for supporting transactional analytics across isolation levels.

3 HERMES OVERVIEW

In this section, we introduce Hermes, an off-the-shelf real-time analytics prototype system, and discuss its architecture, integration with existing engines and design details.

3.1 System Architecture

Figure 1 illustrates the architecture of a hypothetical organization using Hermes. The TP engine handles transactional requests, while the AP engine serves analytical queries. Data storage is decoupled, with AP data stored in a cloud storage service. The transactional log from the TP engine provides fresh data for the AP engine and is also persisted in a cloud storage service. Hermes acts as the synchronization hub between the TP and AP engines.

Hermes includes two in-memory caches: the Log Cache and M-Delta Cache, as well as three services: *DeltaPump*, *Foreground Merge (FGM)*, and *Background Merge (BGM)*. The Log Cache holds the transactional log tail from the TP engine in memory. DeltaPump processes and forwards the log tail to the M-Delta Cache. FGM merges the latest updates from the M-Delta Cache with stable data during analytical reads, while BGM merges asynchronously data from the M-Delta Cache into storage to prevent cache overflow.

Workflow. The TP engine directs the logs to Hermes (i.e, Log Cache), which forwards them to cloud storage. When an analytical query arrives, the AP engine reroutes cloud storage requests to Hermes. Upon receiving the first request, Hermes triggers the DeltaPump service to transfer log tail data from the Log Cache to the M-Delta Cache. Simultaneously, Hermes retrieves stable data from cloud storage and uses the FGM service to merge it with the latest updates from the M-Delta Cache. The merged, up-to-date data is returned to the AP engine in the same stable format (e.g., Parquet [6]), enabling the rest of query execution to proceed. In the background, Hermes runs the BGM service asynchronously to prevent indefinite M-Delta Cache growth.

Since a transaction is considered committed once the commit record hits storage, the AP engine can, by definition, observe only committed transactions. Consequently, Hermes ensures freshness and snapshot consistency with the current OLTP copy for all queries.

3.2 Hermes Integration

In this sections we outline key requirements for the TP and AP engines to support off-the-shelf real-time analytics. In addition, we discuss the Hermes integration details with MySQL [14] as TP engine, and FPDB [73] and DuckDB [60] as AP engines.

3.2.1 ***TP Engine Interface.*** In Hermes, the TP engine handles transactional requests, with the transaction log serving as the main interface between Hermes and the TP engine to maintain data freshness in AP engine queries. Off-the-shelf real-time analytics requires the TP engine to provide a row-level log where the updates can be extracted and merged with analytical reads in real-time.

Log Granularity and Hermes Integration. Systems like MySQL [14], SQL Server [32] and IBM Db2 [38] are well suited for Hermes as they can generate row-level log. In contrast, systems like PostgreSQL [59] and Oracle [56] generate logs that capture changes at different levels of granularity such as data pages or

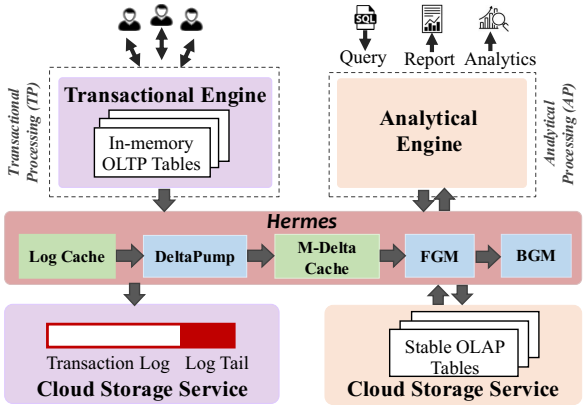


Figure 1: Hermes architecture and main components.

blocks. In this case, integration with Hermes requires post-processing of the log, which involves three key steps: (1) identifying the rows that were modified, inserted, or deleted based on the physical changes recorded at the page/block level, (2) decoding these changes and translating them into logical operations that accurately represent the database's behavior (e.g., update, insertion, deletion), and (3) converting the row-level changes into a standardized format (e.g., Avro) to ensure compatibility. Finally, systems that rely on command-level logging, such as VoltDB [52], are generally incompatible with Hermes; integration requires modifications to the TP engine internals, as post-processing of the log alone is insufficient.

MySQL Integration Details. We integrate Hermes with MySQL [14], an OLTP-optimized database management system (DBMS), by directing its row-based binary log to a network-accessible Log Cache on the Hermes server. For durability, Hermes forwards logs to AWS EBS. This integration requires only a log path update in MySQL's configuration, with no code modifications.

ACID Correctness in Hermes. By preserving TP engine's transactional integrity and merging only committed logs, Hermes ensures ACID compliance for transactions and analytical queries.

Preserving ACID in Transactions. Hermes acts as a log forwarding layer, leaving TP engine's transaction processing unchanged and preserving its ACID guarantees. When a transaction commits, the TP engine writes its log to Hermes' Log Cache, which forwards it to persistent storage. Once stored, the storage acknowledges the Log Cache, which then confirms to the TP engine. Since the TP engine commits only after this acknowledgment, its logging process remains unchanged. If log forwarding fails, the TP engine does not receive the acknowledgment, triggering its standard recovery.

Ensuring ACID in Queries. Hermes guarantees atomicity for queries by processing only fully committed transactions, preventing partial writes or exposure of incomplete data to the AP engine. If log forwarding fails, uncommitted log records are discarded, and only fully committed ones are replayed upon recovery.

To maintain consistency, Hermes relies on the TP engine's transaction log as the single source of truth, merging only committed records. This prevents the AP engine from observing inconsistent intermediate states. Furthermore, isolation is upheld as Hermes ensures that in-progress transactions remain hidden from the AP

engine. Only fully committed log records are made available, preventing anomalies from concurrent execution.

Finally, durability is preserved since transactions are considered durable only after being written to the TP engine's dedicated storage. Hermes adheres to this protocol, making log records accessible to the AP engine only after they are persistently stored.

3.2.2 ***AP Engine Interface.*** In Hermes, AP engines are critical in serving analytical requests. For real-time analytics, AP engines should direct their storage engine requests to Hermes. This setup renders the analytical storage transparent to the AP engine, shifting the responsibility to Hermes for providing the latest data. Consequently, Hermes necessitates that the AP engine consistently reads data directly from storage. As such, the AP engine cannot leverage its local data cache. To address this limitation, we propose offloading the AP cache to Hermes.

FPDB and DuckDB Integration Details. We integrated Hermes with two AP engines: FPDB [73], a cloud OLAP DBMS, and DuckDB [60], an embeddable OLAP DBMS. For both integrations, we followed a similar approach. Specifically, we redirected scan operators' data requests to Hermes, aligning with each AP engine's data access pattern (e.g., reading data from multiple partitions simultaneously). To direct requests to Hermes, we used an RPC protocol like Apache Thrift [8] for server-client communication. Each request specifies the data Hermes should provide, and Hermes returns the updated data to the AP engine for the rest of the query execution. Overall, we added fewer than 100 lines of code across both engines, primarily to implement Hermes and AP engines communication.

3.3 Hermes Design Details

This section outlines Hermes design, focusing on the Foreground and Background Merge algorithms, and its storage organization.

3.3.1 ***Data Organization.*** The data in Hermes is organized into three categories: stable data, transaction logs, and deltas.

Stable Data. Hermes manages multiple segments of data during the merging process, as shown in Figure 1. The AP data of Hermes are stored in a distributed cloud storage service; we call them *stable data*. Stable data is horizontally partitioned based on the primary key and sorted by the primary key within each partition. Each partition is saved as a separate file in the cloud storage and contains all columns corresponding to the rows within that partition. Currently, Hermes supports stable data in CSV and Parquet formats [6] and can handle other industry-standard formats (e.g., XML, JSON, Avro) [5] without modifying its internal functionality.

Transaction Log. The row-level transaction log records the history of changes to the TP data as data events. These events capture table row operations such as insertions (INS), updates (UPD), and deletions (DEL). In addition to storing the log in persistent storage, Hermes also maintains it in the Log Cache to enable faster access.

Deltas. Hermes uses DeltaPump to parse the transaction log and extract the most recent updates from its tail, referred to as *tail-deltas (t-deltas)*. Each t-delta contains the after-images of row changes along with the type of change (e.g., INSERT, UPDATE, DELETE). The t-deltas are sorted by the primary key, and DeltaPump assigns a timestamp to each t-delta, marking the time the log tail was parsed.

Upon retrieval, t-deltas are stored in the *M-Delta Cache* alongside previously fetched deltas; called *memory deltas* (*m-deltas*). All deltas share the same data format, but differing in their timestamps. The M-Delta Cache is designed to enhance the performance of foreground merges by keeping the most recent log data in memory. Older m-deltas are asynchronously written to cloud storage during background merges, referred to as *disk deltas* (*d-deltas*).

3.3.2 ***Foreground Merge (FGM).*** For a data partition, FGM processes a stable data file, several m-delta files, and a t-delta file, all sorted by the primary key. FGM merges these files to produce an output reflecting the latest TP updates. When a key appears in multiple files, FGM ensures the newest delta overwrites older versions, including only the latest key version in the output.

The version of a key that participates in the merging process depends on the analytical query's isolation level. Before merging, Hermes identifies the relevant entries from the log that must be incorporated into the current analytical read. This process ensures support for repeatable reads, enabling transactions that require older versions of keys to access the corresponding data. For this discussion, analytical queries executed outside a transaction follow the default isolation level, Read Committed.

FGM Algorithm. The FGM algorithm consists of two phases: the *bitMap generation* phase and the *filtering* phase. The bitMap generation phase uses the *k-way merge* algorithm to produce k bitMaps, one per input file, identifying rows to retain. The filtering phase applies these bitMaps to exclude unwanted entries from each file.

All input files are sorted by the primary key and are processed in order during the bitMap generation phase. For duplicate keys across files, bits in older files (by timestamp) are set to 0; otherwise, they are set to 1. The duration of this step depends on the log tail size, which remains constant across queries but is affected by the TP engine's T-Throughput. To optimize performance, Hermes caches bitMaps in memory, avoiding regeneration costs for subsequent FGM calls. Each new call updates bitMaps incrementally, flipping bits from 1 to 0 as needed. While bitMaps are small, they can be regenerated for large cold data if caching is infeasible.

After bitmap generation, the filtering phase follows excluding records marked as 0 in the bitmaps. To optimize filtering, Hermes uses the Gandiva [3] expression compiler, which generates LLVM-based native code and leverages the Arrow format and modern hardware (e.g., SIMD). After filtering, the remaining records from each input file are concatenated and forwarded to the AP engine. Hermes enables parallelism by concurrently scanning some data batches, processing others with FGM, and forwarding others to the AP engine, thereby reducing end-to-end latency.

Hermes avoids modifying the stable data or creating new versions through copy-on-write techniques. Instead, it leverages directly the data from the row-level transaction log that records both the updated and unchanged columns of each row, enabling the construction of up-to-date data snapshots for analytical queries. During analytical reads, Hermes incorporates updated rows into query results while suppressing outdated rows in stable data using bitmaps (filtering phase). This approach ensures analytics operate on the latest data without modifying stable data and resembles multi-versioning, with deltas accumulating in memory over time, each tied to a specific timestamp.

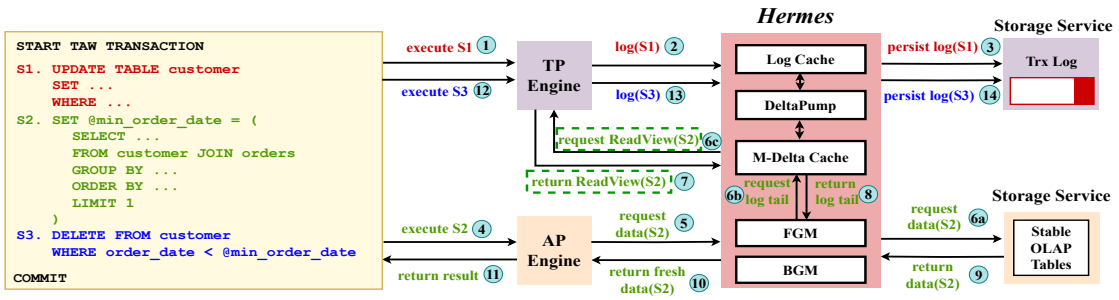


Figure 2: Execution flow of an analytical transaction in Hermes with Snapshot Isolation (SI). Note that the workflow remains unchanged whether Hermes is present or not. An additional coordination exists between the TP engine and Hermes, enabling Hermes to receive the list of visible transactions necessary for achieving SI. This coordination integrates seamlessly

.

3.3.3 ***Background Merge (BGM).*** As more queries are executed, m-deltas will accumulate, occupying more memory capacity. To mitigate this problem, we design a background process that periodically moves the m-deltas to the cloud storage service.

BGM Algorithm. The BGM process is triggered once the size of the m-delta cache exceeds the threshold T_{cache}. Then, each m-delta whose size exceeds a threshold T_{delta} is evicted to the storage engine. To avoid blocking the execution of upcoming queries, a dedicated thread is responsible for evicting the current m-deltas from cache and copy them to the cloud storage. This thread is different from the thread that handles the m-delta and bitMap cache. Depending on the size of the data that have to be uploaded in the cloud storage, the BGM process might take longer time to complete. Each m-delta is uploaded on the cloud storage as a new CSV or Parquet file with the timestamp of the delta attached in the name of the file. We call these files disk-deltas (d-deltas). The d-deltas will be included in the future FGM processes, and they will be treated similarly to an m-delta with an older timestamp. More specifically, a separate bitMap will be generated for the d-delta of a partition that will be cached and updated by the new t-delta entries.

4 TRANSACTIONAL ANALYTICS WITH HERMES

Hermes not only achieves off-the-shelf real-time analytics but also enables real-time transactional analytics.

4.1 Design Challenges

We identify the following two key challenges when integrating transactional analytics into off-the-shelf real-time analytics system.

#1: Efficient Engine Selection. To optimize the execution of transactional analytics within an off-the-shelf real-time analytics system, it is essential to process transactional statements in the TP engine and analytical statements in the AP engine. This approach ensures that each workload type is executed in its respective specialized engine, thus maximizing efficiency and performance.

#2: Isolation Level Consistency. Efficient engine selection mandates that analytical statements within an analytical transaction executes in the AP engine. Maintaining consistent isolation levels for both analytical and transactional statements across different engines is crucial [43, 76]. This synchronization responsibility falls to the TP engine, ensuring that the AP engine accesses accurate data based on the selected isolation level. Ideally, transactional analytics integration should align with TP/AP engines' inherent characteristics, avoiding internal logic modifications.

4.2 Hermes' Isolation Levels Solutions

This section introduces Hermes' generalized solutions for transactional analytics under Snapshot Isolation, Serializable, and Read Committed, as well as MySQL-specific implementations.

4.2.1 ***Snapshot Isolation.*** Snapshot Isolation (SI) [26] ensures that each transaction sees a consistent snapshot of the database as it existed at a specific point in time, typically at the start of the transaction. In database systems, SI is typically implemented using Multi-Version Concurrency Control (MVCC) [27]. In this design, a transaction accessing a table with SI must determine the visible snapshot and read only the data within that snapshot. Different systems represent the snapshot in different ways; some use a single timestamp [28, 32, 49] and others use a compact representation of a list of transaction IDs [14, 59]. We assume the list of transaction IDs in the following discussion but the solution apply to both scenarios.

The list of visible transaction IDs determines which versions are included in a transaction's snapshot. At the start of a transaction's execution, the TP engine gathers this list and shares it with Hermes. By the time the first analytical query, within the analytical transaction, is about to execute in the AP engine, Hermes will have the list and can use it to retrieve the correct log events.

This solution applies to any TP engine supporting SI with MVCC and a row-level transaction log, needing only minor code changes to transmit the transaction ID list to Hermes.

Workflow Example. Figure 2 illustrates the execution flow of an analytical transaction with Hermes under snapshot isolation, comprising three statements: (S1) an update, (S2) an analytical query, and (S3) a delete. The transaction begins with S1 executed in the TP engine (step 1), generating a log saved in Hermes's Log Cache (step 2) and persisted to the Storage Service (step 3). Next, S2 is sent to the AP engine (step 4), which requests data from Hermes (step 5). Hermes retrieves stable data from the Storage Service (step 6a), fetches the log tail (step 6b), and obtains the ReadView from

the TP engine for snapshot consistency (step 6c). After preparing the fresh data (steps 7-9), Hermes delivers it to the AP engine (step 10). The query result is returned to the client (step 11) and used in S3, executed in the TP engine (step 12) and persisted via Hermes (steps 13-14). With S3 complete, the transaction is ready to commit.

Algorithm 1: Hermes API for enabling Transactional Analytics with Snapshot Isolation in MySQL, with changes to the InnoDB storage engine highlighted in gray.

```
1  Function InnoDB::CreateSnapshot(request)
2      readView ← {active_trxs, l_bound, u_bound}
3      readViewMap ← {}
4      for each trxID in readView do
5          trxLogID ← MySQL::getTrxLogID(trxID)
6          readViewMap += {trxID, trxLogID}
7      return readViewMap
8  Function Hermes::ReadFromLog(readViewMap)
9      data ← {}
10     for each trxLogID in LogTail do
11         if trxLogID in readViewMap.active_trxs then
12             continue
13         else if trxLogID > readViewMap.u_bound then
14             continue
15         data += {after image of trxLogID entry}
16     return data
17 Function Hermes::EnableTAW(request)
18     readViewMap ← InnoDB::CreateSnapshot(request)
19     data ← Hermes::ReadFromLog(readViewMap)
20     return data
```

Implementation Details for MySQL. InnoDB, MySQL's default storage engine, uses Undo Logs as part of its MVCC implementation. Each transaction in InnoDB has a set of undo log records, enabling access to previous record versions. Every transaction is assigned a unique ID, and when a consistent read is needed, InnoDB creates a snapshot, or read view, that includes: (1) IDs of active transactions, (2) a lower bound of committed transaction IDs, and (3) an upper bound of future transaction IDs. InnoDB uses this snapshot to access appropriate data versions from the undo log, ignoring records with transaction IDs above the upper bound and those between the bounds if active. This process allows InnoDB to achieve a snapshot for reads, aligning more closely with SI than the Repeatable Read isolation level claimed by MySQL.

As shown in Algorithm 1, Hermes requires two pieces of information for transactional analytics with SI in MySQL: (1) the read view from the InnoDB engine (line 2) and (2) a mapping between InnoDB transaction IDs and those in the transaction log (lines 3-7). Note that MySQL's log transaction IDs differ from InnoDB's transaction IDs. This mapping, along with the ReadView, allows Hermes to retrieve the correct data snapshot for analytical queries (lines 8-16). To provide this data, we made minor modifications to MySQL (lines 3-7), extending the InnoDB's read view with MySQL log transaction IDs and adding code in InnoDB to send the updated read view to Hermes via Thrift clients. Overall, the modifications and additions made to MySQL are fewer than 100 lines of code.

4.2.2 ***Serializable.*** Serializable (SR) [26] isolation level is the strictest isolation level, ensuring transactions execute in a manner equivalent to a serialized order of execution. A conventional method to achieve SR isolation is through a variant of Two-Phase Locking (2PL) [28]. By using 2PL, the database system maintains read and write locks, guaranteeing conflicting transactions execute in a defined sequence, resulting in serializable execution schedules.

To achieve SR transactional analytics in Hermes, the TP engine should prevent concurrent modifications during the execution of the analytical queries within the analytical transaction. When a table needs to be read on the AP side, the entire table should be locked on the TP engine for the duration of the transaction.

This solution can be applied using any TP engine that offers SR using 2PL and supports granularity locking. Implementing it necessitates adjustments to the locking logic within the TP engine—hold locks even if the data is not accessed in the TP engine.

Workflow Example. In the SR isolation level, the workflow described in Figure 2 differs slightly in terms of communication between the TP engine and Hermes. Specifically, compared to steps 6c and 7 of the SI level, the SR level requires somewhat enhanced coordination between the two servers. However, the rest of the workflow remains unchanged.

Implementation Details for MySQL. MySQL's InnoDB storage engine ensures serializability through the implementation of 2PL and granular locking mechanisms. At the SR isolation level, when a read operation is performed within a transaction, InnoDB employs granular locks—such as row, range, or next-key locks—on the necessary data. These locks are acquired at the start of the transaction and held until its completion, effectively preventing other transactions from writing to the locked data.

In our system architecture, when Hermes receives a request to scan data for a specific query, it communicates with MySQL to acquire exclusive locks on the relevant tables, utilizing the InnoDB API to manage these locks at the table level. Once MySQL secures the exclusive locks, it notifies Hermes to proceed with the data scanning operation. This locking mechanism ensures that during Hermes's scanning process, the tables remain isolated from concurrent transactions, preventing any updates that could compromise data integrity. The communication between MySQL and Hermes is facilitated through RPC (e.g. Apache Thrift). Implementing these changes in MySQL required adding fewer than 150 lines of code.

4.2.3 ***Read Committed.*** In Read Committed (RC) [26] isolation, transactions view only committed data stored in the transaction log. Hermes achieves RC for transactional analytics by having analytical queries read all committed transactions of the relevant table from the log. This approach applies to any TP engine with row-level transaction logging.

Workflow Example. For RC isolation level, the workflow described in Figure 2 remains unchanged except that steps 6c and 7 are omitted. Hermes achieves RC by reading each committed transaction directly from the log, without TP engine coordination.

Implementation Details for MySQL. In Hermes, achieving RC isolation level with MySQL required no code changes or additions.

4.3 Transactional Analytics Workload (TAW)

This section explores the significance of *Transactional Analytics*, emphasizing on how existing benchmarks lack generalization compared to TAW and explains the TAW's design principles and details.

4.3.1 ***Motivation of Transactional Analytics.*** Transactional analytics provide real-time insights in HTAP systems by integrating transactional and analytical operations within a single workflow. Standard HTAP benchmarks typically assign separate clients for transactional and analytical workloads, addressing only strictly separated request types. However, prior research [41, 57, 75] shows that hybrid workloads often require mixed transactional and analytical operations within the same transaction. Full HTAP support enables analytical queries on fresh data both post-commit and within the same transaction, allowing subsequent actions based on query results in real-time. This integrated approach supports consistent isolation, avoiding partial updates or stale reads in workflows needing instant decisions on the latest data.

Transactional analytics are critical for applications like fraud detection, personalized healthcare, and supply chain optimization. HyBench [75] uses them for risk control, triggering actions like transaction rollbacks, while PocketData [41] focuses on data management, leveraging nested sub-queries for data deletion. Despite their benefits, existing benchmarks lack a generalized framework for diverse transactional analytics scenarios.

4.3.2 ***Generalized Transactional Analytics with TAW.*** TAW evaluates HTAP systems under generalized transactional analytics scenarios and models diverse transactional analytics patterns, rather than being restricted to a single, predefined workflow such as HyBench [75]. Specifically, the synthetic nature of TAW provides fine-grained control over access patterns—allowing updates, insertions, or deletions to occur before, after, or between analytical queries within the same transaction.

Our experiments show that varying the sequence of operations impacts query plans in HTAP systems (see Section 6.4.2), emphasizing TAW's ability to test broader scenarios. In general, TAW highlights limitations in current approaches and advocates for more adaptable benchmarks to address diverse TA patterns.

4.3.3 ***TAW Design Details.*** To create TAW, we build on HATtrick [53], an HTAP benchmark. HATtrick combines an adapted version of TPC-C [19] for transactional tasks and the Star-Schema Benchmark (SSB)[58] for analytical queries. Its transactional workload (issued by transactional clients T-clients) includes three types of transactions—*NewOrder*, *Payment*, and *CountOrders*—where *NewOrder* is an insertion transaction, *Payment* involves updates and insertions, and *CountOrders* is read-only. The analytical workload (issued by analytical clients A-clients) consists of 13 SSB[58] queries.

For TAW, we take the transactional component of HATtrick and, within each transaction (*NewOrder*, *Payment*, and *CountOrders*), append one of the 13 SSB [58] analytical queries either after or interleaved within the original workload. By adding SSB queries to each transaction, TAW integrates analytics within the same transactional request [57]. A random SSB query number is selected for each transaction to ensure equal probability across all queries. These adapted requests form one part of the TAW workload, issued by dedicated transactional analytics clients (TA-clients), while the original HATtrick transactional requests are issued by T-clients.

5 HERMES POTENTIAL EXTENSIONS

This section explores potential enhancements to the Hermes design, which are considered for future work.

5.1 Cache Offloading to Hermes

Integrating the AP engine with Hermes prevents it from using its local cache, necessitating cache offloading. However, this introduces challenges. First, remote cache access incurs network latency. Second, Hermes must align with the AP engine's caching mechanisms. Third, the optimizer may struggle to generate efficient plans without direct cache metadata. Finally, offloading may disrupt index-based query optimizations.

To mitigate latency, co-locating Hermes with the AP engine minimizes network overhead, enabling near-native caching. Hermes bridges remote caching with the AP engine's optimizer by sharing metadata—index structures, materialized views, and statistics—facilitating efficient execution plans. To maintain consistency, Hermes periodically synchronizes metadata with the AP engine, ensuring that query execution reflects the latest cache state. Moreover, Hermes supports indexing techniques such as min-max indexing and range partitioning, dynamically adjusting index boundaries to enhance accuracy and efficiency.

5.2 Hermes in a Distributed Setup

In distributed TP/AP environments, Hermes enhances scalability and resource efficiency, enabling real-time analytics without disrupting TP/AP engine functionality.

Hermes with Distributed TP. Distributed TP systems typically follow two architectures: (1) a single primary node with multiple read-only replicas [48] and (2) a partitioned shared-nothing model [67]. We describe how Hermes operates in each.

Primary and Replica. Here, a primary node manages writes while read-only replicas handle queries. The TP engine ensures consistency via replication consensus algorithms. With the Hermes integration the storage layer remains the single source of truth. Hermes guarantees fresh data delivery to the AP engine by verifying transaction log durability before merging logs with stable data.

Partitioned Shared-Nothing. In this architecture, data is partitioned across independent nodes, requiring a global transaction order to maintain consistency. Existing protocols, such as two-phase commit [28] and timestamp-based mechanisms in Multi-Version Concurrency Control, ensure a consistent transaction sequence. Hermes uses these mechanisms to merge log entries with the correct data partitions while preserving system integrity.

Hermes with Distributed AP. In both distributed and non-distributed AP engines, the interface between an AP node and the storage engine (e.g., S3) remains consistent and Hermes integrates seamlessly without architectural modifications. To meet distributed AP engines' I/O demands, Hermes scales by partitioning data across multiple servers, each responsible for specific table partitions. A consistent partitioning strategy ensures logs are routed correctly without altering Hermes' internal design.

5.3 Hermes Advancing Middle Layers

Hermes acts as an intermediary between database servers and storage services, enhancing cloud database performance. Traditional middle layers optimize transactions [23], accelerate filtering and aggregation [17, 74], and support caching for query optimization [35]. Hermes enhances these capabilities by supporting real-time and transactional analytics while preserving compatibility with existing architectures. It integrates seamlessly by first applying the latest TP engine updates, then performing pushdown computations to process relevant data before returning results to the AP engine.

6 EXPERIMENTAL EVALUATION

This section evaluates off-the-shelf real-time analytics against baseline systems, highlighting three key aspects of Hermes.

- Hermes can seamlessly integrate with existing TP/AP engines without introducing additional overhead (Section 6.2).
- Hermes' overall TP/AP performance is competitive to well-established solutions (Section 6.3).
- Hermes offers superior performance for transactional analytics compared to existing solutions (Section 6.4).

6.1 Experimental Setup

6.1.1 ***Cloud Server Configuration.*** The experiments are conducted on compute-optimized AWS EC2 instances in the US-West-2 region. We use three different instance types: (1) c5.4xlarge ($0.68 per hour) with 16 vCPU, 32 GB memory and 10-Gbps network bandwidth, (2) c5.9xlarge ($1.53 per hour) with 36 vCPU, 72 GB memory and 10Gbps network bandwidth, and (3) c5.12xlarge ($2.14 per hour) with 48 vCPU, 96 GB memory, and 12Gbps network bandwidth.

6.1.2 ***Systems Setup Configuration.*** In this section we present the different systems setups for Hermes, MySQL, and TiDB.

Hermes Setup. We use three instances, one for each component: (1) the Hermes server uses a c5.9xlarge, (2) the AP-engine (FPDB or DuckDB) uses a c5.4xlarge, and (3) MySQL uses a c5.4xlarge. DeltaPump uses the MySQL binary log connector [62] to parse the log. Both the TP and AP engines maintain a copy of the database with the same schema. The AP-engine's copy is stored in Amazon S3 in Parquet [6] format, while the TP-engine's copy is stored on disk; the TP-engine logs to AWS EBS, through Hermes.

Storage Cost. In our setup, MySQL stores one copy of the data on AWS EBS, while DuckDB stores another copy in AWS S3. The cost of a General Purpose SSD (gp3) is $0.08 per GB-month, and MySQL uses 55GB, resulting in a monthly cost of $4.40. The cost of S3 Standard storage is $0.023 per GB for the first 50 TB per month, and our data size in S3 is 10GB, leading to a monthly cost of $0.23. Therefore, the total storage cost for Hermes is $4.63 per month.

Note that, the AP data stored in the storage engine occupies 10GB in Parquet, whereas the same data requires 55GB in MySQL due to additional storage overhead. This 55GB consists of the base data, InnoDB metadata, and default TP indexes, which increase storage consumption. In contrast, Parquet's compressed columnar format optimizes storage for analytics, resulting in a smaller footprint.

Memory consumption. In our Hermes setup, the AP engine cache is not offloaded to Hermes. DuckDB, by default, avoids caching when reading from a storage engine, and FPDB's caching is disabled. Hermes allocates memory for three caching mechanisms: the Log Cache, M-Delta Cache, and bitmaps for FGM and BGM, as detailed in Sections 3.3.1 and 3.3.2, to accelerate merging processes.

MySQL Setup. We setup MySQL with InnoDB storage engine in a c5.12xlarge instance. Moreover, we created B+ tree indexes to optimize the analytical workload and fine-tuned several MySQL parameters to ensure optimal performance.

Storage Cost. MySQL baseline uses AWS EBS for database storage. Based on actual usage, MySQL stores 115GB of data, resulting in an estimated storage cost of $9.20 per month.

This storage consists of two main components: (1) TP data (~ 55GB), which includes InnoDB metadata and default TP indexes, and (2) AP indexes (~ 60GB), which improve query performance but significantly increase storage overhead.

TiDB Setup. We deploy twelve c5.4xlarge instances following TiDB's recommended configuration [18, 68]: two TiDB servers, six TiKV servers (three replicas per region), and four TiFlash servers. Utilizing TiFlash's disaggregated storage and compute architecture [18], we allocate two write nodes and two compute nodes for TiFlash. The write nodes handle logs from TiKV, convert them to columnar format, and periodically upload updated data to cloud storage. The compute nodes execute queries, accessing the latest data from the write nodes and remaining data from cloud storage.

Storage Cost. The TiDB setup utilizes six TiKV nodes with three replicas using a space of ~ 117GB in total. TiFlash nodes store data in AWS EBS (~ 7GB) and in AWS S3 (~ 8GB). This results in a total storage cost of $10.10 per month.

Baseline Selection. The selection of these systems is driven by their established strengths in their respective domains. MySQL, with its proven transactional processing capabilities and widespread use in cloud environments [12, 24, 70], serves as a key baseline for transactional workloads. DuckDB is a high-performance analytical engine, aligning with trends in data lakes and cloud-native analytics, essential to our architecture. Additionally, FPDB is included to evaluate Hermes' adaptability with less conventional AP engines. FPDB demonstrates Hermes' flexibility in integrating with a diverse range of engines providing valuable insights into the system's versatility. Finally, TiDB was selected as the state-of-the-art (SOTA) HTAP system due to its increasing adoption by major companies [69], which leverage TiDB's ability to manage large-scale transactional and analytical workloads concurrently in real-time. TiDB's robust support for hybrid workloads makes it an ideal baseline for evaluating Hermes' performance in HTAP.

6.1.3 ***Workloads.*** We use two workloads for evaluation: the HATtrick benchmark [53] and the *Transactional Analytics Workload (TAW)*, an adapted version of HATtrick. We discuss their characteristics in Section 4.3.3. To test Hermes under more demanding conditions, we modify HATtrick and TAW to simulate different update/insertion patterns, ensuring an update to every partition of the schema tables with a probability of one.

6.1.4 ***Metrics.*** HATtrick extracts the following metrics, a *throughput frontier graph* and a *freshness score* for every system under test.

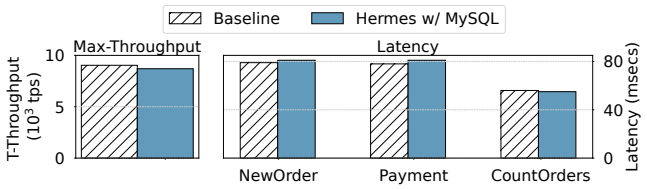


Figure 3: Transactional throughput (T-Throughput) results in tps (left) and transactions' latency results in msecs (right) when executing HATtrick in Hermes w/ MySQL vs. the standalone MySQL executing transactions.

The throughput frontier graph is a 2D plot with transactional throughput (T-Throughput) and analytical throughput (A-Throughput) on the x- and y-axis. It is generated by running various client mixes, showing the system's performance across the HTAP spectrum and its isolation capabilities. Ideally, the frontier aligns with the *bounding box*, defined by maximum T-Throughput and A-Throughput values, indicating perfect isolation. A frontier close to or below the *proportional line* suggests poor HTAP performance. The average freshness score is measured in seconds. A score of $f_{avg} = p$ seconds means that, on average, the data is p seconds out of date.

Similar to HATtrick, TAW generates a throughput frontier for each database, reflecting comparable insights. Note that, in TAW, the y-axis represents transactional analytics throughput (TA-Throughput), measured in analytical transactions per second (taps). In the experiments with HATtrick and TAW, we use scale factor 50 databases, resulting in data sizes of approximately 10GB in Parquet format.

The TAW clients operate similarly to HATtrick. However while HATtrick features an A-client, TAW features a TA-client that issues analytical transactions to extract the throughput frontier results.

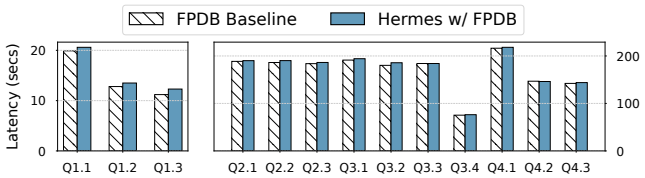
6.2 Hermes Evaluation

This section presents the end-to-end results of Hermes integration with MySQL [14], FPDB [73], and DuckDB [60], demonstrating that integration does not impact their original performance. It also provides results on Hermes resource utilization.

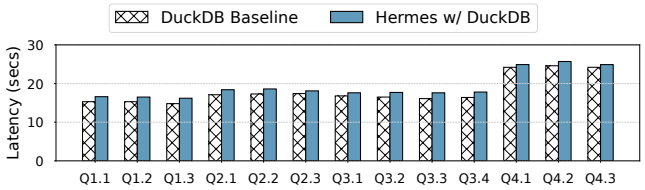
6.2.1 ***Hermes Integration with MySQL.*** Figure 3 (left) displays the maximum T-Throughput achieved in MySQL baseline and the Hermes with MySQL setup for the HATtrick benchmark. In MySQL baseline, only the transactional portion of HATtrick is executed, with analytical queries disabled. In contrast, for the Hermes setup, the results include the concurrent execution of analytical queries by the AP engine (FPDB or DuckDB). Both configurations use MySQL under Snapshot Isolation. Additionally, Figure 3 (right) shows the corresponding transaction latencies.

MySQL baseline achieves a maximum T-Throughput of 9,035 tps, while Hermes with MySQL reaches 8,700 tps. Latencies are similar in both setups, with the Hermes integration introducing up to 4% overhead—a minor trade-off for added functionality.

In the next sections, we discuss the latency results of Figure 4, focusing on Hermes' performance with FPDB and DuckDB. These measurements were taken with Hermes connected to MySQL operating at a fixed T-Throughput of 8,700 tps, as shown in Figure 3. For all queries, the updates merged with stable data correspond to this throughput, remaining consistent throughout the experiments.



(a) FPDB vs. Hermes w/ FPDB.



(b) DuckDB vs. Hermes w/ DuckDB.

Figure 4: Latency (secs) of the HATtrick analytical queries in the Hermes & FPDB setup vs. the case where FPDB executes the queries without using Hermes.

6.2.2 ***Hermes Integration with FPDB.*** Figure 4a illustrates the latency of the HATtrick queries when integrating Hermes with FPDB, compared to FPDB baseline. For each query the left bar corresponds to latency of the original query execution in FPDB baseline and the right bars correspond to the execution of the same query in Hermes with FPDB. In general, the closer the latency results of the Hermes with FPDB setup are to the original FPDB latency, the better for the overall performance of Hermes.

The results demonstrate an overhead of less than 4% in the latency across the 13 queries executed with Hermes, indicating that it imposes minimal performance impact on FPDB baseline.

6.2.3 ***Hermes Integration with DuckDB.*** Figure 4b shows the latency of HATtrick queries executed in Hermes with DuckDB, compared to the DuckDB baseline. For each query, the left bar represents the latency of the original DuckDB execution, while the right bar shows the latency in the Hermes with DuckDB setup. The results indicate that Hermes with DuckDB incurs only about a 2% latency increase compared to the baseline, demonstrating that even with a high-performance AP engine like DuckDB, Hermes can deliver real-time analytics with minimal impact on query latency.

Overall, the results in Sections 6.2.1, 6.2.2 and 6.2.3 demonstrate that the performance of MySQL, FPDB, and DuckDB remain stable after integration with Hermes. The next section shows the resource utilization of the Hermes with MySQL and DuckDB setup.

6.2.4 ***Resource Utilization.*** Figure 5 presents resource utilization for the experiment in Section 6.2.3, focusing on the integration of Hermes and DuckDB. It displays total CPU usage across all vCPUs (%), total network usage (GB) as the sum of received and sent data, and average memory usage (GB). For the Hermes setup, utilization is split into the Hermes and DuckDB nodes, with separate bars for FGM and BGM. FGM shows usage during Foreground Merges only, while BGM includes both Foreground and Background Merges. The DuckDB baseline is included for comparison.

CPU Usage. Figure 5 shows that, compared to the DuckDB baseline, the DuckDB node's CPU usage increases from 14% to 21% in

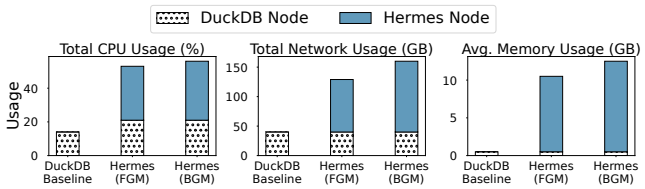


Figure 5: Resource utilization across three configurations: DuckDB baseline, Hermes with FGM, and Hermes with BGM. Figure shows Total CPU Usage across all vCPUs (%), Total Network Usage (GB) and Average Memory Usage (GB) for both the DuckDB and Hermes nodes.

the Hermes setup, consistently across both FGM and BGM configurations. This rise is attributed to the deserialization process on the DuckDB node, which incurs additional overhead as data serialized for network transmission is reconstructed upon receipt.

In the Hermes node, CPU usage increases from 32% to 35% when BGM is enabled alongside FGM. This is expected, as BGM requires the Hermes node to handle an additional background task.

Network Usage. Figure 5 indicates that the DuckDB node's total network usage remains consistent between the DuckDB baseline and the Hermes setup. This is expected, as Hermes integration does not alter the volume of data DuckDB receives from storage.

In the Hermes setup, the Hermes node's network usage exceeds the DuckDB node's due to receiving data from the storage engine and sending updates to DuckDB, effectively doubling usage. As expected, network usage rises further when BGM is active.

Memory Usage. Figure 5 shows that the DuckDB node's average memory usage remains consistent between the DuckDB baseline and the Hermes setup. This is expected, as integrating Hermes does not require additional data caching on the DuckDB node.

In the Hermes setup, the Hermes node utilizes memory mainly for components such as the Log Cache, M-Delta Cache, and bitmap caching, which are essential for accelerating FGM and BGM. The memory usage increases during BGM due to the additional process.

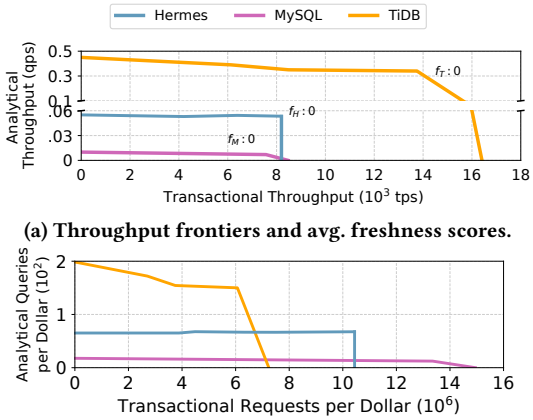
Integrating Hermes with DuckDB slightly increases DuckDB's CPU usage, mainly due to deserialization, while memory and network usage remain unchanged compared to the DuckDB baseline.

Estimated Cloud Cost. Hermes runs on one c5.9xlarge instance ($1.53/hour) and two c5.4xlarge instances ($0.68/hour each). Assuming continuous usage over 30 days, the total compute cost amounts to $2081 per month. For storage, Hermes uses a 64GB EBS volume for TP data in MySQL and a 10GB volume in S3 for AP data, resulting in a total storage cost of $5.53 per month. Thus, the combined compute and storage cost for Hermes is $2086.53 per month.

DuckDB, running on a single c5.4xlarge instance, incurs a compute cost of $490 per month under the same conditions. It stores 10GB of data in S3, contributing an additional $0.23 per month in storage costs. As a result, the total compute and storage cost for DuckDB is $490.23 per month.

6.3 HATtrick Evaluation Across Systems

This experiment compares Hermes, MySQL [14], and TiDB [37] using the HATtrick [53] benchmark, with MySQL and DuckDB as Hermes's TP- and AP-engines. The aim is to show that Hermes achieves performance comparable to established HTAP systems.



(a) Throughput frontiers and avg. freshness scores.

(b) Operations per dollar frontiers.

Figure 6: Throughput frontiers, freshness scores (f_{DB}) and operations per dollar frontiers for Hermes (with MySQL and DuckDB), MySQL, and TiDB when executing HATtrick.

6.3.1 ***Comparison Results.*** The discussion will focus on throughput frontier shapes, absolute throughput values, and cost frontiers.

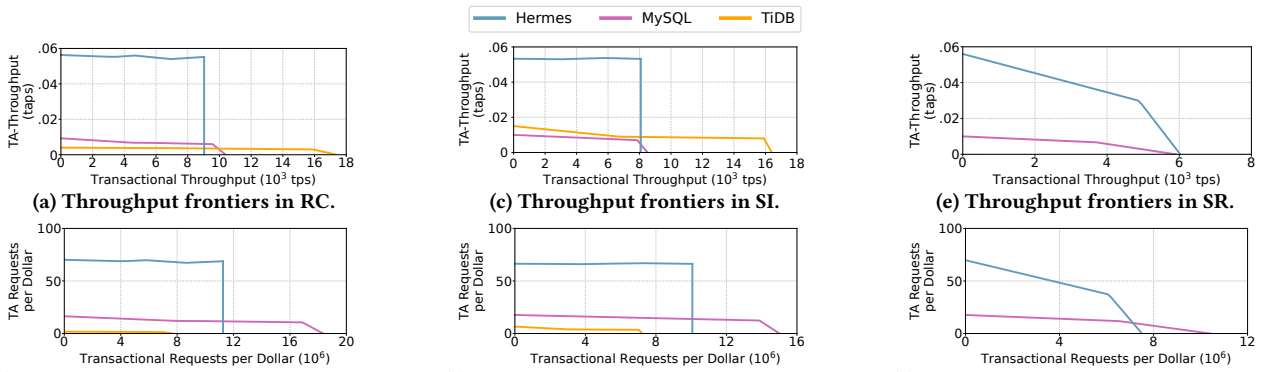
Throughput Frontier Shapes. Figure 6a shows HATtrick results for each system. The Hermes frontier (blue) aligns closely with its bounding box, demonstrating excellent *performance isolation* and minimal TP and AP workload interference. MySQL's frontier (purple) falls between its bounding box and proportional line, indicating resource contention. TiDB's frontier (yellow) initially follows its proportional line, with A-Throughput decreasing as T-clients grow, but later approaches its bounding box, mitigating this effect.

Absolute Throughput. TiDB hast the highest T- and A-Throughput values in Figure 6a. This is expected, particularly for T-Throughput, as TiDB distributes transactional requests across two TiKV servers. The consistently high A-Throughput is due to TiDB's ability to cache frequently accessed data on the local SSDs of TiFlash compute nodes [18]. However, as T-clients increase, A-Throughput declines since frequent updates make cached data outdated.

Freshness Values. We used HATtrick benchmark to measure the freshness of the analytical queries in Hermes, MySQL, and TiDB. Our results show that all the three databases achieve zero freshness, indicating that all queries are always executed on up-to-date data.

Cost Frontiers. Note that the three curves in Figure 6a are generated using different hardware settings. For a more fair comparison, we normalize the monetary cost and report the throughput per dollar in Figure 6b. The figure highlights that Hermes can execute more transactional requests per dollar than TiDB. Conversely, TiDB outperforms Hermes in analytical queries per dollar, but as the number of T-clients increases, this difference becomes smaller.

Our results in Sections 6.2 and 6.3 show that Hermes inherits the stability of its underlying TP/AP engines. Specifically: (1) Figures 3 and 4 show that Hermes maintains the original performance of each engine, and (2) Figure 6 confirms that this stability holds across varying client combinations. The shape of Hermes' throughput frontier highlights its ability to deliver stable HTAP performance, allowing concurrent transactions and analytics without mutual impact. Hermes achieves this stability while matching leading HTAP systems in performance and offering a cost-effective solution.



(a) Throughput frontiers in RC. (c) Throughput frontiers in SI. (e) Throughput frontiers in SR.

(b) Operations per dollar frontiers in RC. (d) Operations per dollar frontiers in SI. (f) Operations per dollar frontiers in SR.

Figure 7: Throughput frontiers and operations per dollar frontiers results for Hermes, MySQL, and TiDB when executing TAW at scale factor 50 under three different isolation levels Read-Committed (RC), Snapshot-Isolation (SI) and Serializable (SR).

6.4 TAW Evaluation Across Systems

In this section we assess the performance of Hermes, MySQL [14], and TiDB [37] when executing the TAW in three different isolation levels, Read Committed, Snapshot Isolation and Serializability.

6.4.1 ***Comparison Results.*** Figure 7 presents the results of executing the TAW in Hermes (blue), MySQL (purple), and TiDB (yellow) across three different isolation levels. Each isolation level includes a graph depicting the throughput frontiers and another graph showing the corresponding operations per dollar frontiers.

Read Committed (RC) Results. Figure 7a illustrates the throughput frontiers, while Figure 7b presents the corresponding operations-per-dollar frontiers in RC. Hermes (blue) demonstrates nearly perfect performance isolation, as its throughput frontier closely aligns with its bounding box, indicating minimal impact from the transactional workload. In contrast, MySQL (purple) and TiDB (yellow) show frontiers between their proportional lines and bounding boxes, revealing a significant decline in TA-throughput as the number of T-clients increases. This decline is evident in the sharp drop towards the end of their frontiers.

In terms of absolute performance, Hermes achieves the highest TA-throughput (0.056 taps), followed by MySQL (0.009 taps) and TiDB (0.004 taps). TiDB leads in T-throughput with 18,000 tps, compared to Hermes (9,000 tps) and MySQL (10,500 tps). Hermes outperforms both competitors in transactional analytics per dollar (Figure 7b) and ranks second in transactional requests per dollar.

Snapshot Isolation (SI) Results. Similar to RC, Figure 7c presents throughput frontiers, and Figure 7d illustrates operations-per-dollar frontiers for SI. Hermes (blue) maintains strong performance isolation, with its frontier near the bounding box. In contrast, MySQL (purple) and TiDB (yellow) exhibit frontiers between their proportional lines and bounding boxes, reflecting a decline in TA-Throughput under higher transactional workloads.

In terms of absolute performance, Hermes once again achieves the highest TA-Throughput (0.054 taps), followed by TiDB (0.015 taps) and MySQL (0.01 taps). TiDB leads in T-Throughput with 16,500 tps, followed by MySQL at 8,500 tps and Hermes at 8,000 tps. Hermes maintains its lead in transactional analytics requests per dollar, while ranking second in transactional requests per dollar.

Serializability (SR) Results. Figure 7e and Figure 7f depict the throughput and operations-per-dollar frontiers under SR, respectively. TiDB is omitted as it does not support SR. Hermes (blue) exhibits a distinct frontier with a unique shape compared to RC and SI cases, reflecting the dependent nature of transactional and analytical workloads under SR. In both Hermes and MySQL, traditional and analytical transactions compete for lock access, leading to a decline in TA-Throughput as T-clients increase. This explains why Hermes' frontier deviates from its bounding box, even though analytical queries are executed on the DuckDB side. MySQL (purple) follows a similar pattern but achieves lower TA-Throughput.

Hermes leads with the highest TA-Throughput (0.057 taps), followed by MySQL (0.01 taps). Both achieve a T-Throughput near 6,000 tps. Hermes excels in transactional analytics per dollar and ranks second in transactional requests per dollar.

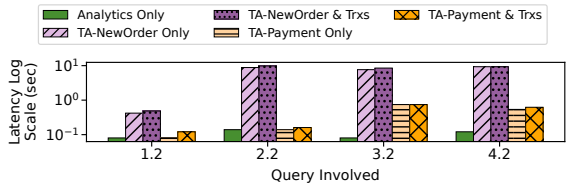
Overall, Hermes surpasses MySQL and TiDB in TAW across all isolation levels, in absolute performance and performance isolation.

6.4.2 ***TiDB Analysis.*** This section explores TiDB's results in detail, highlighting key findings and explaining why TiDB's performance in TAW falls significantly short of HATtrick.

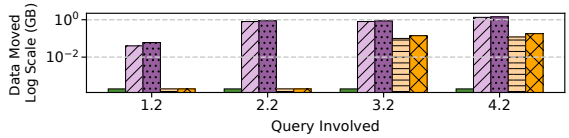
Workload Configurations. We use three workload configurations to analyze TiDB's performance differences between transactional analytics in TAW and traditional analytics in HATtrick. First, Analytics-Only runs only the analytical component of the HATtrick benchmark, measuring TiDB's performance on traditional analytics without transactional interference. Next, TA-X workloads (where X is NewOrder or Payment) execute TA-X analytical transactions alone, isolating the impact of transactional analytics. Finally, TA-X & Trxs includes both TA-X analytical transactions and regular transactions, revealing TiDB's limitations under mixed workloads.

Comparison Results. Figure 8a shows TiDB latency results for selected SSB queries across different workload configurations, omitting other queries with similar performance patterns (e.g., Queries 1.1 and 1.3 resemble 1.2). Figure 8b displays the total data transferred from TiFlash nodes to the TiDB server node during execution.

Figure 8a shows that the lowest latency occurs in the Analytics-Only workload, where queries run on TiFlash nodes optimized for analytics, aligning with HATtrick results (see Figure 6a) showing



(a) Latency measured in seconds (secs) for the involved queries.



(b) Total data moved measured in gigabytes (GB).

Figure 8: Figure 8a and 8b display data from experiments conducted on TiDB, featuring various workloads: Analytics Only (no Trxs), TAW with TA-NewOrder Only (no Trxs), TAW with TA-NewOrder and Trxs, TAW with TA-Payment Only, and TAW with TA-Payment and Trxs. Figure 8a show query latency and Figure 8b total data transferred during execution.

the highest A-Throughput for TiDB. The figure also reveals significantly higher latencies for TA-NewOrder (Only/& Trxs) and TA-Payment (Only/& Trxs) workloads compared to Analytics-Only, consistent with TiDB results under TAW (see Figures 7a, 7c, 7e).

Additionally, Figure 8a illustrates that the execution time for TA-NewOrder & Trxs and TA-Payment & Trxs consistently surpasses that of TA-NewOrder Only and TA-Payment Only, respectively. This emphasizes the influence of concurrent transaction execution on the latency of transactional analytics in TiDB.

Query Plan Analysis. In the Analytics-Only workload, queries run entirely on TiFlash compute nodes optimized for analytics. In contrast, TA-NewOrder and TA-Payment query processing extends beyond TiFlash. Data from updated tables—LINEORDER for TA-NewOrder and SUPPLIER and CUSTOMER for TA-Payment—are first retrieved from TiFlash nodes, then transferred to TiDB server nodes for processing by the *UnionScan* operator. This operator likely ensures isolation by merging recent data from TiFlash write nodes with S3-accessed data. Parts of the query then run across both TiFlash nodes and the TiDB server, causing data transfers (see Figure 8b). The extent of data movement depends on computation level in TiFlash, and larger tables like LINEORDER require more merging time, explaining the higher latency for TA-NewOrder (Only/& Trxs) compared to TA-Payment (Only/& Trxs).

TiDB's TAW performance is hindered by query plan changes introduced by transactional analytics. In contrast, Hermes maintains consistent query plans, making it well-suited for both workloads.

7 RELATED WORK

This section reviews current solutions for (near) real-time analytics.

HTAP Systems. HTAP systems unify TP and AP to enable real-time analytics. Single-system architectures often employ shared [21] or optimized [32, 33, 40, 45–47, 55, 63, 65] data structures for the two workloads, ensuring immediate availability of transactional data for analytical queries. This approach eliminates replication latency but may increase contention. Other HTAP systems separate TP and AP engines, either sharing the same storage layer [31, 34, 45, 54] for immediate data visibility or using decoupled storage [37, 72] to isolate resources and allow independent scaling. In decoupled setups, transactional changes are periodically propagated to the AP layer via Change Data Capture (CDC) or log-based replication, with minimal latency. Most HTAP systems tightly couple compute and storage components, though exceptions like F1 Lightning [72] theoretically support pluggable engines, albeit without verification.

Change Data Capture (CDC) Tools. CDC tools are designed to monitor and replicate changes—such as inserts, updates, and deletions—in source databases to maintain data consistency across systems. They typically analyze transaction logs (e.g., PostgreSQL's WAL or MySQL's binary logs) to detect modifications and then stream these changes in standardized formats (e.g., JSON, Avro) to target systems. While CDC tools are essential for data replication, migration, and synchronization between systems, they generally do not perform complex data processing. Their primary focus is to ensure that target systems accurately reflect the latest changes from source systems in real time. Notable CDC tools include Debezium [10], GoldenGate [15], pg_logical [16], and StreamSets [13].

Streaming Data Platforms (SDP). SDPs are designed for real-time ingestion, transportation, and processing of data streams from various sources. Unlike CDC tools, which primarily replicate database changes, SDPs offer advanced data processing capabilities such as windowing, aggregations, and joins, essential for real-time analytics and event-driven architectures. While SDPs can integrate CDC tools to capture and stream database changes in real-time, their primary function is to facilitate the flow of diverse data types—including logs, metrics, sensor data, and other event streams—across systems. They enable low-latency, high-throughput data movement and support robust integration options for real-time data pipelines across different systems. Examples of SDPs include Apache Kafka [1, 44], Apache Pulsar [7], Amazon Kinesis [2], and Google Pub/Sub [11].

Cloud-Based Storage Services. Storage services such as Delta Lake [22] and Hudi [4] are designed to add transactional capabilities over cloud-based object storage, enabling reliable data management for large-scale analytical and transactional processing. These services implement structured data formats (e.g., Parquet, ORC) and define access protocols, supporting transactions on data stored in distributed object storage. For example, Delta Lake utilizes versioned metadata and transaction logs to track changes, ensuring data consistency. However, Delta Lake typically requires modifications when integrated with various TP engines. In contrast, Hudi emphasizes flexibility, providing native support for multiple TP and AP engines. Hudi benefits from CDC tools for capturing data changes and SDPs for efficiently processing data streams, thereby enhancing its ability to manage evolving datasets in real-time. Both Hudi and Delta Lake follow principles of Lambda and Kappa architectures [42], with real-time and batch processing layers that support the continuous integration and historical accuracy of data.

8 CONCLUSION

In this paper, we introduce off-the-shelf real-time transactional analytics — a design that leverages existing TP and AP engines to deliver fresh analytics. We implement this as Hermes, an intermediate layer between computation and storage. Evaluation shows that Hermes outperforms current HTAP systems by up to 3× in transactional analytics.

REFERENCES

[1] 2024. Amazon Kafka. https://kafka.apache.org.

[2] 2024. Amazon Kinesis Streams Developer Guide. https://docs.aws.amazon.com/kinesis/.

[3] 2024. Apache Gandiva. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/.

[4] 2024. Apache Hudi. https://hudi.apache.org.

[5] 2024. Apache Iceberg: The open table format for analytic datasets. https://iceberg.apache.org.

[6] 2024. Apache Parquet. https://parquet.apache.org.

[7] 2024. Apache Pulsar: Cloud-Native, Distributed Messaging and Streaming. https://pulsar.apache.org.

[8] 2024. Apache Thrift. https://thrift.apache.org/about.

[9] 2024. Databricks Migration Strategy: Lessons Learned. https://www.databricks.com/blog/databricks-migration-strategy-lessons-learned.

[10] 2024. Debezium: Stream changes from your database. https://debezium.io.

[11] 2024. Google Cloud Pub/Sub: A Google-Scale Messaging Service. https://cloud.google.com/pubsub/docs/overview.

[12] 2024. Google Cloud SQL for MySQL. https://cloud.google.com/sql/docs/mysql.

[13] 2024. IBM StreamSets: Seamless hybrid and multicloud data integration. https://www.ibm.com/products/streamsets.

[14] 2024. MySQL. https://dev.mysql.com/doc/refman/8.0/en/.

[15] 2024. Oracle GoldenGate: Replicate and Transform Data. https://www.oracle.com/integration/goldengate/.

[16] 2024. pglogical: Logical replication for PostgreSQL. https://www.2ndquadrant.com/en/resources/pglogical/.

[17] 2024. S3 Select and Glacier Select. https://aws.amazon.com/blogs/aws/s3-glacier-select/.

[18] 2024. TiFlash Disaggregated Storage and Compute Architecture and S3 Support. https://docs.pingcap.com/tidb/stable/tiflash-disaggregated-and-s3.

[19] Revision 5.11. 2009. TPC BENCHMARK™ C.

[20] Aisha Abdallah, Mohd Aizaini Maarof, and Anazida Zainal. 2016. Fraud detection system: A survey. *Journal of Network and Computer Applications* 68 (2016), 90–113.

[21] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research*.

[22] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[23] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, Vol. 8. 28.

[24] Microsoft Azure. 2024. Azure Database for MySQL. https://learn.microsoft.com/en-us/azure/mysql.

[25] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, Rene Mueller, Vijayshankar Raman, Richard Sidle, Matt Spilchen, Adam J Storm, Yuanyuan Tian, Pinar Tözün, et al. 2017. Evolving Databases for New-Gen Big Data Applications.. In *CIDR*.

[26] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[27] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[28] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

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

[30] Chaoyi Cheng, Mingzhe Han, Nuo Xu, Spyros Blanas, Michael D Bond, and Yang Wang. 2023. Developer's Responsibility or Database's Responsibility? Rethinking Concurrency Control in Databases. In *13th Annual Conference on Innovative Data Systems Research (CIDR'23). January 8-11, 2023, Amsterdam, The Netherlands*.

[31] Google Cloud. 2024. AlloyDB: A fully managed PostgreSQL database service. https://cloud.google.com/products/alloydb?hl=en.

[32] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.

[33] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[34] Christian Garcia-Arellano, Hamdi Roumani, Richard Sidle, Josh Tiefenbach, Kostas Rakopoulos, Imran Sayyid, Adam Storm, Ronald Barber, Fatma Ozcan, Daniel Zilio, et al. 2020. Db2 event store: a purpose-built IoT database engine. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3299–3312.

[35] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[36] Riyaz Ahamed Ariyaluran Habeeb, Fariza Nasaruddin, Abdullah Gani, Ibrahim Abaker Targio Hashem, Ejaz Ahmed, and Muhammad Imran. 2019. Real-time big data processing for anomaly detection: A survey. *International Journal of Information Management* 45 (2019), 289–307.

[37] 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. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[38] IBM Corporation. 2024. *IBM DB2 Database*. IBM Corporation. https://www.ibm.com/products/db2-database Version 11.5.8.

[39] Arun Kejariwal, Sanjeev Kulkarni, and Karthik Ramasamy. 2017. Real time analytics: algorithms and systems. *arXiv preprint arXiv:1708.02621* (2017).

[40] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[41] Oliver Kennedy, Jerry Ajay, Geoffrey Challen, and Lukasz Ziarek. 2016. Pocket data: The need for TPC-MOBILE. In *Performance Evaluation and Benchmarking: Traditional to Big Data to Internet of Things: 7th TPC Technology Conference, TPCTC 2015, Kohala Coast, HI, USA, August 31–September 4, 2015. Revised Selected Papers 7*. Springer, 8–25.

[42] Martin Kleppmann. 2019. Designing data-intensive applications.

[43] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Matei Zaharia, and Xiangyao Yu. 2023. Epoxy: ACID Transactions across Diverse Data Stores. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2742–2754.

[44] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. Athens, Greece, 1–7.

[45] 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 *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

[46] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1740–1751.

[47] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. 2017. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1598–1609.

[48] Feifei Li. 2023. Modernization of databases in the cloud era: Building databases that run like Legos. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4140–4151.

[49] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 21–35.

[50] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.

[51] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 37–50.

[52] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 604–615.

[53] Elena Milkai, Yannis Chronis, Kevin P Gaffney, Zhihan Guo, Jignesh M Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *Proceedings of the 2022 International Conference on Management of Data*. 1810–1824.

[54] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. 2017. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics.. In *CIDR*.

[55] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[56] Oracle Corporation. 2024. *Oracle Database*. Oracle Corporation. https://docs.oracle.com/en/database/oracle/oracle-database/23/index.html Version 23c.

[57] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1771–1775.

[58] O'Neil Pat, O'Neil Betty, and Chen Xuedong. 2009. The Star Schema Benchmark.

[59] PostgreSQL Global Development Group. 2024. *PostgreSQL Database*. PostgreSQL Global Development Group. https://www.postgresql.org/ Version 16.

[60] Mark Raasveldt and Hannes Mühleisen. 2023. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[61] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[62] Stanley Shyiko. 2022. MySQL Binary Log connector. https://github.com/shyiko/mysql-binlog-connector-java.

[63] 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 *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 731–742.

[64] Eugene Siow, Thanassis Tiropanis, and Wendy Hall. 2018. Analytics for the internet of things: A survey. *ACM computing surveys (CSUR)* 51, 4 (2018), 1–36.

[65] Alex Skidanov, Anders J. Papito, and Adam Prout. 2016. A column store engine for real-time streaming analytics. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1287–1297. https://doi.org/10.1109/ICDE.2016.7498332

[66] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. *The VLDB Journal* (2024), 1–31.

[67] Michael Stonebraker. 1986. The case for shared nothing. *IEEE Database Eng. Bull.* 9, 1 (1986), 4–9.

[68] PingCAP TiDB. 2024. Deploy a TiDB Cluster Using TiUP. https://docs.pingcap.com/tidb/stable/production-deployment-using-tiup.

[69] PingCAP TiDB. 2024. TiDB Customers. https://www.pingcap.com/customers/.

[70] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[71] Reynold S Xin, William McLaren, Patrick Dantressangle, Steve Schormann, Sam Lightstone, and Maria Schwenger. 2010. MEET DB2: automated database migration evaluation. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1426–1434.

[72] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, et al. 2020. F1 Lightning: HTAP as a Service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3313–3325.

[73] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. Flexpushdowndb: Hybrid pushdown and caching in a cloud DBMS. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2101–2113.

[74] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1805.

[75] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. *Proceedings of the VLDB Endowment* 17, 5 (2024), 939–951.

[76] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2022. Skeena: Efficient and consistent cross-engine transactions. In *Proceedings of the 2022 International Conference on Management of Data*. 34–48.