HeraDB: Towards Real-time Analysis of Transaction-Centric HTAP with CPU-GPU Hybrid Query Execution

Zeshun Peng[†], Qincheng Cai[†], Siyuan Wei[†], Weixing Zhou[†], Yanfeng Zhang[†], Yansong Zhang[‡], Guoliang Li[§], Ge Yu[†]
[†]Northeastern University, China; [‡]Renmin University of China; [§]Tsinghua University

ABSTRACT

Modern OLTP applications, such as financial fraud detection and stock risk management, rely on analytical queries for real-time decision-making. In these scenarios, queries must access the freshest data without disrupting the performance of mission-critical transactions. We characterize them as *transaction-centric* HTAP workloads. Existing HTAP databases fail to reconcile these conflicting requirements. Co-located architectures, which deploy both the row store and the column store on the same machine, degrade isolation due to shared resource contention. Conversely, separated architectures with distributed deployments sacrifice data freshness due to log synchronization latency.

In this paper, we propose HeraDB, a GPU-accelerated HTAP database that achieves high-throughput transaction processing while maintaining strong performance isolation from analytical queries. To ensure query freshness, HeraDB introduces a hybrid CPU-GPU query execution strategy: the GPU path leverages massive parallelism to scan the stable but stale column-store snapshot, while the CPU path concurrently processes delta updates to retrieve complementary fresh results. To retrieve complementary results, HeraDB maintains timestamp-based filters on both paths. This design mitigates PCIe bottlenecks by synchronizing lightweight timestamp logs rather than massive delta updates. Evaluation on HATtrick shows that HeraDB outperforms Colibri in maximum individual TP/AP throughput, achieving up to 11.1× higher TP and 5.9× higher AP with high data freshness and strong performance isolation.

ACM Reference Format:
Zeshun Peng[†], Qincheng Cai[†], Siyuan Wei[†], Weixing Zhou[†], Yanfeng Zhang[†],, Yansong Zhang[‡], Guoliang Li[§], Ge Yu[†]. 2026. HeraDB: Towards Real-time Analysis of Transaction-Centric HTAP with CPU-GPU Hybrid Query Execution. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern online transaction processing (OLTP) applications must execute highly concurrent transactions, while simultaneously supporting analytics on the freshest data [13, 62, 73]. For instance, online payments, stock trading, and e-commerce order processing rely on real-time analytics for fraud detection, security monitoring, and risk management [51, 63, 64]. In these applications, analytical queries must read the most recent data versions without degrading transaction performance. These mission-critical scenarios constitute a *transaction-centric* hybrid transactional/analytical processing (HTAP) workload, where analytical processing (AP) requires high data freshness while maintaining strong performance isolation from high-throughput transactional processing (TP).

Because a pure column-store fails to support high-performance TP [4, 30], these systems typically adopt a dual-store design [24, 29, 31, 42]. Transactions operate on a row store optimized for random writes, while analytical queries perform range scans on a column store optimized for sequential reads. To ensure queries access the latest results, delta updates generated in the row store are periodically propagated to the column store.

These systems can be further classified into two types, which strike a trade-off between performance isolation and data freshness [74]. (i) **Co-located architectures** deploy both stores on the same machine [68, 75] (Figure 1a). Queries can access the latest updates residing in the delta table directly, enabling them to observe the latest committed updates. This ensures high data freshness but sacrifices performance isolation due to contention for shared CPU and memory resources. (ii) **Separated architectures** deploy the row store and column store on different machines [15, 45, 62, 76] (Figure 1b). It provides strong isolation through hardware-level decoupling. However, it fails to achieve high freshness because propagating batched delta updates to the remote column store incurs extra latency. Queries scan only the stale column snapshot, as fetching the latest delta table across machines is costly.

GPU-accelerated heterogeneous HTAP (H^2TAP) databases [4, 34, 55] are a promising solution for reconciling this trade-off. Compared with co-located architectures, they use GPUs to accelerate query execution, providing hardware-level isolation by reducing CPU and memory contention with TP transactions. Compared with separated architectures, they process TP and AP on the same machine, avoiding log replication delays and improving data freshness. However, existing H^2TAP databases adopt pure column-store designs, which are inefficient for high-throughput updates and therefore unsuitable for transaction-centric HTAP workloads.

In this paper, we propose HeraDB, a H^2TAP database designed for transaction-centric HTAP workloads. HeraDB adopts a dual-store architecture with a CPU row store and a GPU column store. While this design preserves high TP throughput while enabling GPU-accelerated AP, it introduces two key challenges:

Challenge 1: costly delta table access via PCIe. GPU query execution cannot directly access the delta table residing in CPU memory due to the physically separate memory spaces. To obtain

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

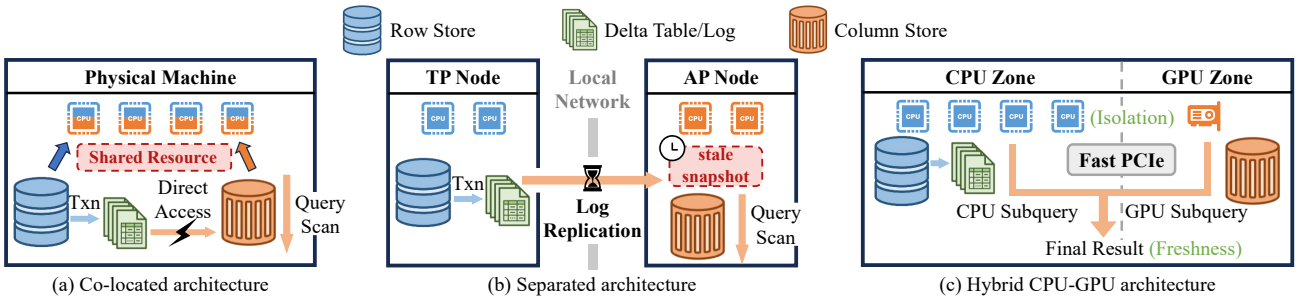


Figure 1: Comparison of HTAP architectures. (a) Co-located: Low isolation due to CPU and memory resource contention. (b) Separated: Low freshness due to log propagation delay. (c) HeraDB: Achieves high freshness and strong isolation by offloading heavy scans to GPU while CPU handles lightweight delta updates.

the latest updates during column scans, the GPU must retrieve the delta table via PCIe. Otherwise, queries can only operate on stale snapshots [55]. Caldera [4] uses CUDA Unified Memory [47] to share the delta table implicitly. However, it suffers from severe page thrashing [16], which incurs significant PCIe traffic. Since the delta table is typically small, another approach [34] is to transfer the delta updates to the GPU before each query. During query execution, the GPU scans both the column snapshot and the delta updates, then combines the results to obtain the freshest data. However, this approach also suffers from poor performance isolation (detailed discussion in Section 2.3).

Challenge 2: high overhead of applying delta updates. High-throughput TP rapidly generates row updates. Without timely merging these updates into the column store, the delta table may become excessively large. This increases the CPU subquery's scanning overhead, which in turn increases resource contention and degrades TP performance. Co-located architectures maintain the column store on the CPU, enabling delta merge to directly access the delta table and update the column store with low overhead. Separated architectures use logs (e.g., redo logs) [24] to efficiently propagate and apply updates without scanning the delta table. In contrast, propagating delta updates to a GPU-resident column store is more costly. Because the GPU employs a SIMT execution model distinct from the CPU, directly applying existing append-only logs to update the column store incurs massive synchronization overhead. Therefore, the CPU must scan the delta table to generate a GPU-friendly log before the merge. This delta log generation may contend for shared resources with transactions, degrading performance isolation.

To address these challenges, HeraDB proposes hybrid CPU-GPU query execution (Figure 1c). Inspired by distributed query processing [39], HeraDB partitions the table temporally. The GPU stores the stale but stable column-store snapshot, while the CPU stores recent delta updates. Each query is decomposed into two subqueries that produce complementary results. The GPU subquery executes on the snapshot in GPU memory. Meanwhile, the CPU subquery concurrently executes on the delta table in CPU memory to retrieve the latest updates. The GPU partial result is aggregated with the CPU partial result to produce the final result.

HeraDB maintains timestamp-based filters to ensure that the two subquery results are complementary. The column store maintains a Snapshot Column Vector (SCV) column for each table, and the delta table maintains a filter log consisting of version timestamps. During query execution, the GPU subquery uses the SCV to exclude stale snapshot tuples that have been updated since the snapshot, while the CPU subquery scans the updates with the filter log to retrieve the corresponding fresh versions. Before each query, HeraDB synchronizes only the incremental filter log entries to the GPU to update the SCV, which is lightweight because it transmits timestamps rather than actual tuple data (Section 4.3).

HeraDB periodically merges the delta updates into the column store to limit its size. The delta table is designed as a compact log format that stores only the latest version of each updated row. During a merge, HeraDB directly transmits the delta log to the GPU and applies it to the column store, avoiding the extra cost of generating a separate GPU-friendly log. By carefully handling read-write conflicts, this format also supports efficient CPU subquery scans. If the version stored in the delta table is invisible to the CPU subquery, it retrieves the correct historical version from the multi-version row store. In summary:

- We propose HeraDB, a GPU-accelerated database for transaction-centric HTAP workloads that reconciles the trade-off between data freshness and performance isolation.
- We propose a hybrid query execution strategy that minimizes PCIe bottlenecks by decomposing queries into complementary GPU and CPU subqueries via lightweight SCVs and filter logs.
- We propose a unified delta table format that enables both lock-free CPU scanning and efficient parallel GPU merging.
- We evaluate HeraDB on HATtrick [44] and SSB-TC (Section 7.1). HeraDB outperforms Colibri in max individual TP/AP throughput, achieving up to 11.1× higher TP and 5.9× higher AP, and preserves zero visibility latency under mixed TP/AP workloads.

2 BACKGROUND AND MOTIVATION

2.1 Design Goals

Data freshness. Following the definition of Perseus [62], we measure data freshness using visibility latency [24, 44]. Visibility latency captures how recent the query-visible snapshot is (i.e., the staleness of the snapshot observed when an AP query begins). Thus, its visibility latency is the time difference between the query start time (e.g., read timestamp) and the commit timestamp of the latest transaction included in the snapshot used by the query. A query has zero visibility latency when its query-visible snapshot includes all transactions committed before the query starts.

Co-located architectures typically have zero visibility latency. This is because both the row store and the column store reside

Table 1: Comparison of HTAP database architectures

Architecture	System Representative	Storage Layout	Perf. Isolation	Data Freshness	Trade Offs
Separated	TiDB [24], BatchDB [42] ByteHTAP [14, 15]	Full Row + Column	Physical Machine	Async. Log Propagation	Low Freshness (Log Replication Delay)
Co-located	SQL Server [31], Oracle IM [29] GaussDB-HTAP [68]	Full Row + Column	Shared CPU/RAM	Direct Memory Access	Low Isolation (Resource Contention)
	Colibri [57], SAP HANA [60] SingleStore [50, 61]	Column + Row Delta	Shared CPU/RAM	Direct Memory Access	Low Isolation (Resource Contention)
	OSIC [3]	MVCC Row Only	Shared CPU/RAM	MVCC Snapshot	Low Isolation
H^2TAP	Raza et al. [55]	Column Only	Hardware Isolation	Batched Delta Update	Low Freshness
	Caldera [4]	Column Only	Hardware Isolation	CUDA Unified Memory	PCIe Page Thrashing
	RateupDB [34]	Column Only	Hardware Isolation	Transmit Delta to GPU	Query-Specific Delta
	HeraDB **(Ours)**	Full Row + Column	Hardware Isolation	**Hybrid Query Execution**	**Minimal (Filter Log)**

in main memory within a unified memory space, enabling AP to directly access the latest committed updates in the delta table. In contrast, separated architectures physically isolate AP and TP resources, requiring log replication to synchronize updates. Updates from TP nodes are batched, transmitted, and replayed on AP nodes [24], introducing visibility latency of hundreds of milliseconds.

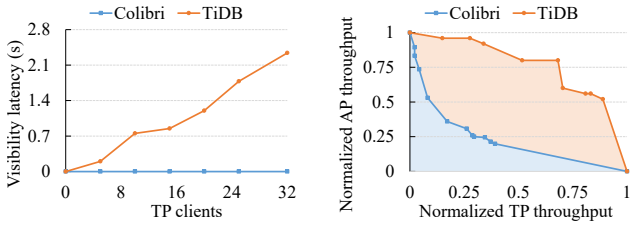
Performance isolation. AP must not degrade TP performance. Co-located architectures lack isolation due to shared resources. A heavy scan can saturate memory bandwidth and pollute CPU caches, increasing the tail latency of concurrent transactions. While software-level scheduling and memory management [9, 54] can increase isolation, they cannot eliminate contention from shared hardware. In contrast, separated architectures ensure strong isolation via physically decoupled resources.

2.2 Transaction-Centric HTAP Databases

Table 1 categorizes representative HTAP databases.

To maximize TP throughput, systems such as SQL Server [31], Oracle IM [29], and GaussDB-HTAP [68] maintain a full-row store and selectively replicate tables into the in-memory column store to accelerate query execution. When updating the row store, transactions also append these updates to a delta table (e.g., SMU in Oracle, tail index in SQL Server). These systems achieve high freshness by enabling query scans to access both the delta table and the column store directly. Specifically, the scan operator fetches the latest updates from the delta table, merges them with the column store snapshot on the fly, and forwards the merged results to downstream operators (e.g., joins, group-by) [29, 31, 34, 68]. Systems such as OSIC [3] maintain only a multi-version row store. Queries obtain a consistent snapshot by traversing version chains to retrieve proper tuple versions. Although this design provides fresh snapshots on CPUs, extending it to GPU is inefficient. First, keeping a GPU replica consistent with the CPU store requires fine-grained synchronization between CPU and GPU MVCC versions, causing high coordination overhead under write-heavy workloads. Second, GPU-side MVCC traversal introduces indirect memory accesses during per-tuple visibility checks, breaking the sequential scan pattern preferred by GPU execution.

Systems like Colibri [57], SAP HANA [60], and SingleStore [50, 61] adopt a column store as the primary storage and maintain a delta table in row format (e.g., L1 store in SAP HANA) to buffer recent



(a) Freshness (AP clients=3) (b) Normalized throughput frontiers

Figure 2: Performance metrics across HTAP architectures.

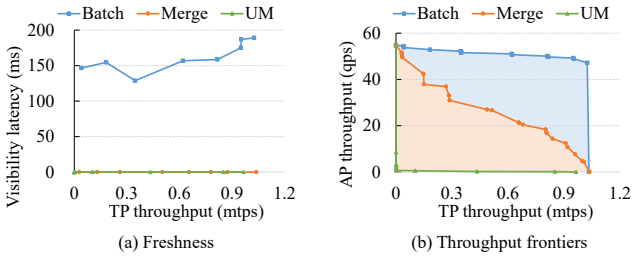
updates. This design achieves high AP throughput via efficient column scans, but with lower TP scalability [74] because the delta table stores only recently modified rows. However, these systems have low performance isolation due to resource contention.

Distributed databases such as TiDB [24], BatchDB [42], and ByteHTAP [14, 15] decouple row and column stores across physical machines, achieving strong isolation at the cost of low freshness. To access the latest updates, TiDB delays query execution until the relevant logs arrive and are fully applied. Similarly, BatchDB immediately triggers update propagation when queries request the latest snapshot version. However, these strategies sacrifice query latency (i.e., query response time) to ensure real-time analytics.

Experiments. To validate this trade-off, we compare Colibri and TiDB under SSB-TC (an update-heavy workload) with uniform distribution (detailed setup in Section 7.1). In Figure 2a, Colibri achieves high freshness (i.e., 0ms) since TP and AP operate on the same replica, enabling queries to access the delta table directly. In contrast, TiDB's visibility latency increases to 2.4s as TP clients scale from 1 to 32. This is because higher TP throughput generates more delta updates, which increases the latency of Raft log replication and log application in TiFlash, thereby degrading data freshness.

To demonstrate performance isolation, we measure throughput frontiers [44] [1] by varying the number of TP and AP clients across multiple configurations (i.e., various client mixes) and record the TP

[1] **How to read the frontier plot:** The throughput frontier connects the best observed TP-AP throughput points across tested workload configurations. Each point in the plot represents the TP and AP throughput achieved under one TP/AP client configuration. The x-axis and y-axis correspond to the standalone TP-only and AP-only throughput, respectively. Ideally, the frontier should coincide with the upper right corner of the bounding box defined by the maximum TP throughput and the maximum AP throughput, indicating perfect performance isolation between TP and AP.



(a) Freshness (b) Throughput frontiers

Figure 3: Results of CPU-GPU HTAP execution strategies.

and AP throughput. Figure 2b shows the normalized result. Colibri's frontier is closer to the axes than TiDB's, indicating lower isolation.

2.3 Heterogeneous HTAP Databases

H^{2}TAP systems (e.g., Caldera [4], RateupDB [34], and [55]) employ column-only storage to exploit the high GPU memory bandwidth for accelerated scanning. Caldera maintains identical data structures (e.g., DSM or PAX [2]) across the CPU and GPU, utilizing CUDA Unified Memory to ensure consistency between replicas. RateupDB transmits the entire delta table to the GPU before each query, and merges the updates with the GPU snapshot during table scans. [55] proposes a batch processing model that periodically updates the GPU column store. Existing H^{2}TAP databases execute queries entirely on the GPU, thus cannot simultaneously achieve strong performance isolation and high data freshness.

Experiments. We implement three internal variants on top of HeraDB's codebase, where `Batch`, `Merge`, and `UM` simulate the query execution strategies of [55], RateupDB, and Caldera, respectively. Section 7.1 summarizes their characteristics.

In Figure 3b, `UM` exhibits low performance due to page thrashing. When updating a row, the entire page must be migrated to the CPU. When an AP query subsequently scans it, the page is migrated back to the GPU. Since TP workloads feature high-throughput random writes, massive page migrations saturate the PCIe bandwidth. `Merge` exhibits poor isolation under high TP throughput (i.e., with a larger delta table). This is because generating query-specific delta updates requires scanning the row store, which is memory-intensive and contends for shared resources with concurrent transactions. `Batch` avoids CPU involvement in query execution, achieving the best isolation. However, as shown in Figure 3a, it introduces visibility latency proportional to the delta merge interval.

3 HERADB OVERVIEW

3.1 Architecture

HeraDB operates under snapshot isolation and employs timestamp-based multi-version concurrency control (MVCC). Transactions and queries execute based on a shared global clock.

As shown in Figure 4, HeraDB stores the row store and delta table in CPU and the column-store snapshot in GPU. For hybrid query execution, each AP query is decomposed into two subqueries. The GPU subquery scans the stable but stale column snapshot, while the CPU subquery scans the delta table to retrieve the latest updates. The two partial results are then merged on the CPU to produce the final result. By fetching recent updates on the CPU and using the GPU only for snapshot scans, HeraDB avoids the PCIe bottleneck

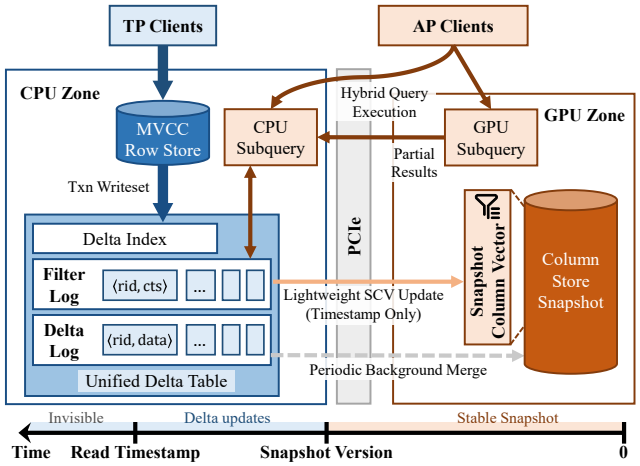


Figure 4: System architecture and workflow.

of transferring massive delta updates before each query, while still ensuring fresh AP results.

To ensure that the two subqueries produce complementary results, HeraDB uses timestamp-based filters on both paths. The GPU subquery must exclude stale rows from the column snapshot, because the latest versions of these rows are retrieved by the CPU subquery from the delta table. To support this filtering, each table in the GPU column store maintains a Snapshot Column Vector (SCV), which records the timestamp of the first update after the snapshot for each row. During GPU execution, rows whose SCV timestamps indicate stale snapshot versions are filtered out. Correspondingly, the delta table maintains a lightweight filter log that helps the CPU subquery identify these updated rows and retrieve their correct visible versions. Before each query, HeraDB transmits only incremental filter-log entries, which are lightweight than data tuples, to update the SCV on the GPU.

This hybrid strategy applies to most aggregatable queries. For queries containing non-aggregatable operators (e.g., outer joins and semi-joins), HeraDB falls back to GPU-only execution. The CPU path acts as a freshness-preserving scan operator and transfers visible delta tuples to the GPU, while the GPU executes the remaining query over the snapshot and delta tuples (Section 4.4).

3.2 Workflow and Key Features

Non-blocking TP execution. HeraDB first applies updates to the row store during transaction execution. After commit, it uses the delta index to install the transaction's write set to the delta table. Because this step happens after the result is returned to the user, updating the delta table does not affect end-to-end TP latency.

The delta table is stored in a compact form, where each modified row stores only its latest version. This design has two benefits. First, it reduces delta index maintenance overhead, as in-place updates only require index modifications when a row is first inserted into the delta log. Second, CPU subqueries can efficiently scan the delta log without handling duplicate rows. In contrast, append-only delta tables [31] may store multiple historical versions of the same row, requiring deduplication during scans. When the latest version in the delta log is invisible to a query, the CPU subquery retrieves the correct historical version from the row store (Section 4.2).

Algorithm 1 Read Timestamp Acquisition

```
1:  procedure TxCommitAndUpdateDelta(tid)
2:      slot ← slotArray[tid];                    ▷ Get worker thread slot
3:      slot.cts ← GetAndIncrementGlobalClock( ); ▷ Atomic op
4:      Commit(slot.cts);                         ▷ Commit and notify client
5:      UpdateDeltaTable(tx.writeSet);
6:      slot.cts ← null;                          ▷ Clear the timestamp
7:  procedure QueryAcquireReadTimestamp( )
8:      rts ← GetGlobalClock( );
9:      while ¬ IsTimestampSafe(rts) do
10:         continue;          ▷ Spin until timestamp becomes safe
11:     return rts;
12: procedure IsTimestampSafe(rts)
13:     for all slot ∈ slotArray do
14:         cts ← slot.cts;
15:         if cts is null then          ▷ No active commit in this worker
16:             continue;
17:         if rts ≥ cts then            ▷ AP would read incomplete delta
18:             return false;
19:     return true;
```

Real-time AP execution. When an AP query starts, HeraDB first acquires a read timestamp from the global clock. The CPU subquery scans the filter log to identify rows updated after the snapshot, and retrieves their visible versions from the delta log or the multi-version row store. In parallel, HeraDB transmits incremental filter-log entries to the GPU to update the SCV. The GPU subquery then scans the column store and uses the SCV to filter out stale snapshot rows. Finally, the two results are merged and returned to the client.

The CPU subquery could contend with concurrent transactions for shared resources. HeraDB achieves strong isolation through two mechanisms. First, the delta log is stored in a columnar format that supports lock-free scans. The CPU subquery falls back to accessing the multi-version row store only when the version in the delta log is invisible or concurrently updated by TP, thereby reducing random memory accesses. Second, the delta table size is small due to frequent delta merge (i.e., 400ms), minimizing scan overhead and memory bandwidth contention with TP.

Efficient delta merge. A background thread periodically merges the delta log into the column store. Since the delta table captures all updates since the last snapshot, the delta merge simply truncates the delta log and transmits it to the GPU for parallel application. The delta merge does not block concurrent transactions. New queries are briefly paused during the delta merge (4-13ms), without degrading query performance (detailed in Section 5.1).

4 HYBRID QUERY EXECUTION

Before execution, a query acquires a read timestamp to ensure it observes a complete snapshot (Section 4.1). The CPU path scans the delta table (Section 4.2), while the GPU path scans the column-store snapshot (Section 4.3). The partial results are merged to produce the final output (Section 4.4).

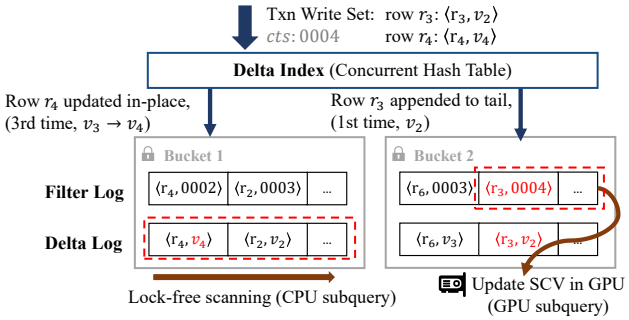


Figure 5: Detailed design of the delta table.

4.1 Read Timestamp Acquisition

When a query acquires a read timestamp rts, concurrent transactions with commit timestamps $cts \leq rts$ may still be in the process of committing. Specifically, they may have acquired commit timestamps but not yet finished writing their write sets to the delta table. If the query begins scanning the delta table immediately, it may miss these in-flight updates, violating snapshot consistency.

HeraDB maintains a fixed-size slot array to avoid scanning all active transactions during consistency checking. Each TP worker thread maintains a slot in the array (in Algorithm 1). Before committing, the thread acquires a commit timestamp from the global clock and stores it in its slot, which serves as a visibility marker for concurrent queries (line 3). After committing and updating the delta table, the thread clears its slot by setting it to *null* (lines 4-6).

When a query starts, it first acquires a read timestamp rts from the global clock (line 8). It then checks whether all transactions with $cts \leq rts$ have finished updating the delta table by iterating over all slots (lines 9-10):

- **Empty slot** (cts is *null*, lines 15-16): The TP thread is idle or executing a transaction. Since the transaction has not yet committed, it will obtain a commit timestamp cts later after rts (i.e., $cts > rts$). The query can proceed without waiting.
- **Non-empty slot with** $rts \geq cts$ (lines 17-18): The transaction has committed with $cts < rts$ but has not yet finished writing its updates to the delta table. The query must wait until this slot is cleared to ensure visibility of all committed updates.

Notably, this waiting process is negligible (≈0.01 ms in Table 2), in contrast to systems that incur millisecond-level delays [14, 24]

Lemma 4.1 (Snapshot Consistency). *After a query with a read timestamp rts begins execution, all transactions with commit timestamps $cts \leq rts$ have completed updating the delta table.*

4.2 Query Execution on the CPU Path

Delta table design. Figure 5 shows the delta table structure. With in-place updates, the delta log stores only the latest updated version of each modified row, while the filter log stores the timestamp of the first update after the GPU snapshot, which is used as a visibility filter. The delta and filter logs are organized in a columnar layout using fixed-size buffers.

The delta index is used to quickly install the write set of TP transactions into the delta table. It maps each row ID to its corresponding offset in the delta log and filter log. To reduce lock contention, the delta index is implemented as a concurrent hash

Algorithm 2 Lock-free Delta Table Scanning

```
1:  procedure QueryScanBatch(offset, rts, batchSize)
2:      rows ← ∅;
3:      for i from offset to offset + batchSize - 1 do ▷ Read phase
4:          if filterLog[i].ctsmin > rts then
5:              continue;                              ▷ No visible version
6:          r.data ← ExtractData(deltaLog[i]);
7:          r.offset ← i;
8:          rows.append(r);
9:      memory_fence;                                  ▷ Acquire fence
10:     for each r ∈ rows do                           ▷ Validation phase
11:         if deltaLog[r.offset].ctsmax ≤ rts then
12:             continue;                              ▷ Safe to read
13:         r.data ← ReadFromMVRowStore(r.data.rid);
14:     return rows;
15: procedure TxUpdateRecord(offset, cts, rowData)
16:     if filterLog[offset].ctsmin > cts then
17:         filterLog[offset].ctsmin ← cts;            ▷ Update filter
18:     if deltaLog[offset].ctsmax > cts then
19:         return;                                    ▷ Avoid overwrite update
20:     deltaLog[offset].ctsmax ← cts;
21:     memory_fence;                                  ▷ Release fence
22:     UpdateData(deltaLog[offset], rowData);
```

table. Each bucket in the hash table is protected by a bucket-level lock, enabling different buckets to be updated in parallel.

We track two timestamps for each row in the delta table to exclude stale or invisible versions during query execution:

- cts_{min}: the commit timestamp of the first update after the last GPU snapshot. It is mostly used to identify stale versions that must be filtered out by the GPU subquery.
- cts_{max}: the commit timestamp of the most recent update in the delta table. It is used by the CPU subquery to determine whether the current version in the delta log is visible to the current query.

Updating the delta table. After committing and updating the row store, a transaction installs its write set into the delta table. For each updated row, the transaction queries the delta index to locate the corresponding log entry. If the row is not yet in the delta table (e.g., row r_3 in Figure 5), the transaction appends a new log entry to its tail and updates the delta index with the new offset. If the row already exists in the delta table (e.g., row r_4), the transaction obtains the entry offset from the delta index and updates the entry.

This design ensures correctness when transactions install their write sets into the delta table out of commit order, as detailed in `TxUpdateRecord` of Algorithm 2. Consider two transactions T_1 and T_2 updating the same row r, where $T_1.cts < T_2.cts$. Due to concurrent execution, T_2 may install its version into the delta table before T_1 updating the delta table. To prevent T_1's older version from overwriting T_2's newer update, each transaction checks $r.cts_{\text{max}}$ before updating the delta log (lines 18-19). If T_2 has already installed its version, then $r.cts_{\text{max}} > T_1.cts$. T_1 skips updating the delta log. If T_1 installs first, then $r.cts_{\text{max}} < T_2.cts$. T_2 updates the delta log and sets $r.cts_{\text{max}}$ to $T_2.cts$. After a transaction updating the delta log, it

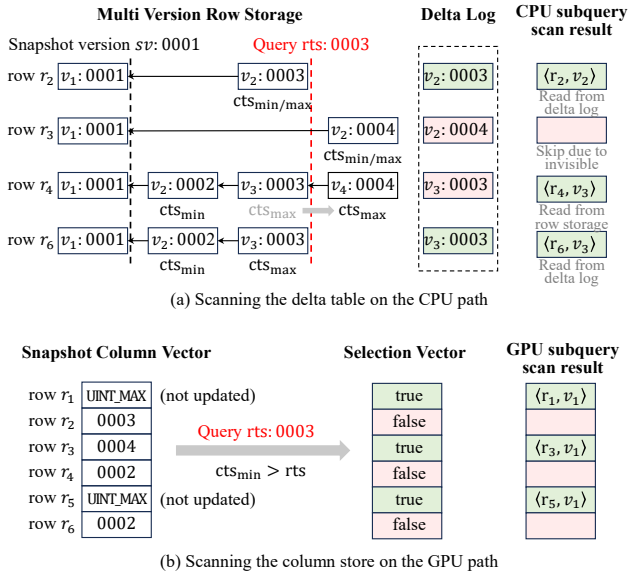


Figure 6: An example of hybrid query execution.

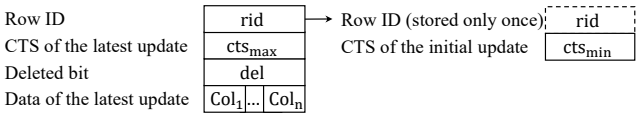


Figure 7: Structure of the log entry in the delta table.

uses a memory fence to ensure the update is visible to concurrent queries, then updates the actual data (lines 20-22).

Scanning the delta table. The structure of the delta and filter log entries is shown in Figure 7. A row has the same row ID when stored in both the row store and the column store. For each entry, *rid* denotes the corresponding row ID. The deleted bit *del* and column data $\langle col_1, \ldots, col_n \rangle$ store its most recently committed version.

The CPU subquery employs a vectorized execution model [8] to accelerate query execution. `HeraDB` first scans delta tuples from the row store (and delta logs) and transforms them into in-memory columnar batches. The CPU subquery then applies SIMD instructions (e.g., filtering) on these batches.

The CPU subquery scans the delta log to retrieve row updates since the last snapshot that are visible, as shown in `QueryScanBatch` of Algorithm 2. We employ an optimistic approach to scan the delta log. The query uses cts_{min} in the filter log to skip rows that were first modified after the query's read timestamp rts. In other words, for rows with $\text{cts}_{\text{min}} > rts$, their correct versions are in the column store and will be scanned by the GPU subquery (lines 4-5). Otherwise, the query reads the latest version in the delta log optimistically, without acquiring tuple locks (lines 6-8), since the version is likely visible to the current snapshot. To validate its visibility, the query checks cts_{max} for the read version. If $\text{cts}_{\text{max}} \leq rts$, the version was committed before the query started and is safe to read (line 11). Otherwise, the row was either concurrently updated or contains an invisible version. In this case, the query fetches the correct version from the multi-version row store (line 13).

To ensure correctness, we use a memory fence before the validation starts (line 10). This ensures that all cts_{max} updates by concurrent transactions are visible to the query before validation.

Lemma 4.2 (CPU Concurrency Guarantee). *A query can safely scan the delta table even when concurrent transactions update the filter log and delta log during query execution.*

Proof. First, the filtering results are not affected by concurrent transactions (lines 4-5). According to Lemma 4.1, if the cts_{min} of a row is updated when a query scans the delta table, the new value (which is the *cts* of a concurrent transaction) is still greater than the query's *rts* and will still be filtered out. Since timestamp updates are atomic (64-bit timestamp), queries will observe either the old value or the new value atomically.

Second, we use memory fences to ensure queries access the correct versions during concurrent updates. The update of cts_{max} occurs before updating the data (lines 20-22). Therefore, if a query successfully validates the version (i.e., $cts_{\text{max}} \leq rts$), its data read previously is guaranteed to be complete and visible. □

The lock-free scan effectively resolves read-write conflicts between queries and transactions. Multiple queries can execute concurrently since they only read from the delta table.

An example. In Figure 6a, a query starts with a read timestamp `rts:0003`. Concurrently, a transaction with commit timestamp `cts:0004` has updated cts_{max} for row r_4 but has not yet written to its delta log entry. The CPU subquery skips row r_3 because its cts_{min} (`0004`) exceeds *rts*, indicating the visible version is in the column store. Rows r_2 and r_6 are fetched from the delta log because their cts_{max} values (`0003` and `0002`) do not exceed *rts*, meaning their versions in the delta log are visible. Row r_4 is fetched from the row store because its cts_{max} (`0004`) exceeds *rts* (the delta log version may be invisible) while its cts_{min} (`0002`) does not (the column store version is stale).

4.3 Query Execution on the GPU Path

Concurrent queries execute on different snapshot versions, while the column store maintains only a single version that is periodically updated. To allow each GPU subquery to filter the column store based on its own snapshot, we maintain a Snapshot Column Vector (SCV) to enable timestamp-based filtering.

The following describes (i) how to efficiently apply the filter log to the SCV, and (ii) how to scan the column store with the SCV.

Snapshot column vector. The SCV is updated by the filter log before each query. The SCV stores the cts_{min} for each row, whose definition is identical to the cts_{min} in the filter log (Section 4.2). When generating a new snapshot, the SCV is initialized with special values: (i) For valid rows in the snapshot, cts_{min} is set to ∞ (i.e., `UINT_MAX`), ensuring they are visible to all queries. (ii) For rows deleted before the snapshot, cts_{min} is set to 0, ensuring they are filtered out without maintaining a separate deletion bitmap [31].

`HeraDB` caches the filter log in the GPU to reduce PCIe traffic. Before each query, only the incremental part of the filter log is transmitted to the GPU. Specifically, each bucket of the delta table maintains a watermark *wm* that records the offset of the last transmitted filter log entry. Only entries after *wm* are transmitted to the GPU. After transmission, the cts_{min} values from the filter log cache are applied to the SCV, ensuring it reflects the most recent update state (`ApplyFilterLog` in Algorithm 3).

Algorithm 3 Preparing the Selection Vector

```
1:  procedure ApplyFilterLog(filterLog, rts)        ▷ Cached in GPU
2:      for each entry e ∈ filterLog in parallel do
3:          if e.cts_min ≤ rts then
4:              SCV[e.rid] ← e.cts_min;
5:  procedure SVGeneration(rts)
6:      SV ← ∅;
7:      for i from 0 to len(SCV) - 1 in parallel do
8:          if SCV[i] > rts then
9:              SV[i] ← true;                       ▷ Valid version
10:         else
11:             SV[i] ← false;                      ▷ Stale version
12:     return SV;
```

Due to concurrent TP execution, transactions may update the delta table out of order. A transaction with a smaller commit timestamp may overwrite a larger cts_{min} already written to the filter log by a later-committing transaction. To ensure that all updates to the filter log are applied to the SCV, when a transaction updates an entry that has already been transmitted to GPU with a smaller cts_{min}, it resets the corresponding *wm* to the offset of this entry, forcing subsequent queries to retransmit the entries since this offset.

Scanning the column store. GPU subquery execution employs a tile-based execution model [58]. Each thread block (i.e., a group of GPU threads) processes a batch of rows at a time.

As shown in `SVGeneration` of Algorithm 3, before scanning the column store, a query first generates a selection vector (denoted as *SV*) based on its read timestamp *rts* and the cts_{min} of each row in the SCV. The selection vector identifies rows that have not been updated since the snapshot, which should be included in the scan (lines 8-9). Rows that have been modified after the snapshot (i.e., $SV[i]$=false) are filtered out, since their correct versions will be retrieved by the CPU path. Notably, multiple queries can concurrently generate their respective selection vectors and execute on different snapshots without interference.

Lemma 4.3 (GPU Concurrency Guarantee). *Queries on the GPU path can concurrently update the SCV and scan the column store under different snapshots.*

Proof. Consider a query with read timestamp *rts*. If the cts_{min} of a row in the SCV is greater than *rts* during execution, it will only be updated to a new value greater than *rts*. This is because all transactions with $cts \leq rts$ have already completed (Lemma 4.1). Therefore, although the SCV may be updated during scanning, these newly updated cts_{min} values must be greater than *rts* and will be filtered out, thus not affecting the final result. □

An example. In Figure 6b, rows r_1 and r_5 remain unmodified (i.e., $\text{rts:}0003 < cts_{\text{min}} \text{:UINT\_MAX}$) and are included in the columnar scan. Row r_3 was not updated before the query read timestamp (i.e., $\text{rts:}0003 < cts_{\text{min}} \text{:}0004$) and is also included in the scan.

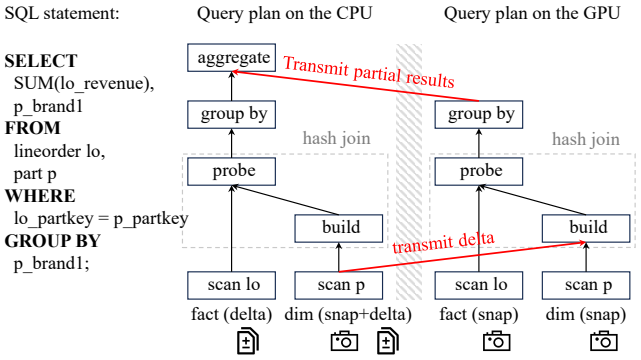


Figure 8: An example of query result aggregation.

Theorem 4.4 (Query Completeness). *The partial results from the GPU path and CPU path collectively constitute a consistent snapshot at the query read timestamp rts.*

Proof. By Lemma 4.3, the GPU subquery scans all tuples unmodified until the query read timestamp *rts*. By Lemma 4.2, the CPU subquery scans all tuples modified after the column snapshot version and visible at *rts*. Since these two sets are disjoint and their union covers all tuples committed before *rts*, the combined results constitute a complete and consistent snapshot at timestamp *rts*. □

4.4 Lightweight Result Aggregation

After executing the query plan on both paths, partial results from the GPU are transferred to the CPU and merged into the final result. This workflow is efficient for parallelizable operators such as projection, selection, and aggregation. However, operators such as joins may require multiple exchanges of intermediate results, incurring high PCIe traffic.

The hybrid query execution of HeraDB is similar to distributed query processing [39]. However, instead of partitioning tables horizontally by key ranges, HeraDB partitions tables temporally. The GPU stores the stable snapshot, while the CPU stores recent delta updates. Since the GPU lacks visibility for the latest updates, this temporal partitioning may prevent local join optimizations.

To address this challenge, HeraDB exploits the characteristics of HTAP workloads. HTAP workloads (e.g., HATtrick [44] based on SSB [48]) typically follow a star schema with one large fact table (e.g., LINEORDER) and several small dimension tables (e.g., SUPPLIER, CUSTOMER, PART). The fact table stores transactional records and is frequently updated by transactions. In contrast, dimension tables contain descriptive attributes and are relatively small with infrequent updates (e.g., at SF=100, dimension tables are 17-277MB, while the fact table is ≈60GB). Replicating these dimension tables incurs low overhead, which is widely adopted in existing systems [37]. Therefore, to minimize PCIe data transfer during multi-table operations (e.g., joins between the fact table and dimension tables), HeraDB maintains full replicas of dimension tables in both CPU and GPU memory.

An example. Figure 8 presents an example of a simplified SSB Q2.1 query, where a fact table LINEORDER joins with a dimension table PART. The CPU maintains the latest version of PART (delta + snapshot) and the delta updates of LINEORDER. The CPU subquery directly executes filter, join, and group-by operations to produce partial results. Since the GPU stores only table snapshots, the GPU subquery first employs a broadcast join by retrieving the delta updates of PART from the CPU. Subsequently, it performs scans based on both the delta and snapshot to obtain the latest version of PART, and executes filter, join, and group-by operations. Finally, the partial results are transmitted to the CPU for final aggregation.

Discussion. Our prototype implements SSB queries [48] with a single fact table. For queries involving joins across multiple fact tables or non-aggregatable operators (e.g., outer joins and semi-joins), HeraDB can fall back to a GPU-only query execution. In this mode, the CPU subquery acts only as a freshness-preserving delta scan operator. It scans the delta tables to retrieves the visible versions, and transfers these delta tuples to the GPU. The GPU then executes the remaining operators over its snapshot and delta tuples.

GPU-only execution is also preferred when the cost of transferring high-cardinality intermediate results exceeds the cost of transmitting delta tables directly to the GPU, such as in high-cardinality aggregations or group-by operations. The decision between hybrid and GPU-only execution can be guided by existing cost models [58, 69, 70] that estimate PCIe transfer overhead.

5 DIRECT COLUMN STORE UPDATE

The delta table grows rapidly over time as transactions continuously generate updates. Periodically merging these delta updates into the column store is essential to limit the delta table size. Section 5.1 discusses the design rationale of the delta merge strategy. Section 5.2 presents the detailed workflow.

5.1 Design Rationale

HeraDB blocks new queries from execution when updating the GPU column store due to the following reasons:

Memory efficiency. To avoid read-write conflicts, when merging the delta table, query execution must maintain multiple column-store versions on the GPU via copy-on-write or double-buffering. This would halve the available GPU memory, limiting the maximum table size. By blocking concurrent queries, HeraDB can perform in-place updates directly on the column store, thereby saving memory.

Query performance. Although blocking introduces a wait time (4-13 ms as shown in Table 3), it does not increase query latency. For the CPU path, the delta merge process clears the delta table, eliminating the overhead of scanning large delta logs on subsequent queries. For the GPU path, executing queries concurrently during merge would degrade performance due to GPU context switching and cache contention, slowing both merge and queries.

5.2 Merge Workflow

The merge process executes in the background periodically. It incurs minimal impact on TP performance because it requires only sequential, read-only access to CPU memory. Unlike systems that require the CPU to perform complex log-format conversion [42], HeraDB simply copies the delta logs to the GPU and applies them directly by GPU threads.

As shown in Figure 9, the merge process consists of four phases:

Phase 1 (Wait): Due to the single-version column store, queries may execute concurrently while the column store is being updated, causing read-write conflicts. To prevent this, the merge process

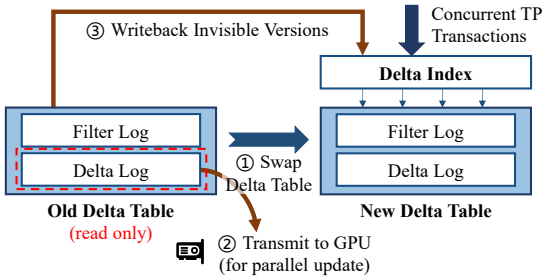


Figure 9: Workflow of the delta merge process.

Algorithm 4 Applying Delta Log to Column Store

```
1: procedure ApplyDeltaLog(deltaLog, sv)
2:     for each entry e ∈ deltaLog in parallel do
3:         if e.cts_max ≤ sv then
4:             if e.deleted then
5:                 SCV[e.rid] ← 0;                ▷ Mark as deleted
6:             else
7:                 SCV[e.rid] ← ∞;                ▷ Reset to infinity
8:                 UpdateData(e.rid, e.data);     ▷ Update row data
9:         else continue;          ▷ Skip tuples committed after sv
```

blocks new queries and waits for in-flight queries to complete. After obtaining exclusive access to the column store, it acquires a timestamp (via `QueryAcquireReadTimestamp` in Algorithm 1) that serves as the snapshot version sv of the updated column store. Upon receiving sv, all transactions with commit timestamps $cts \leq sv$ have finished writing the delta table (Lemma 4.1), ensuring the integrity of the generated snapshot.

Phase 2 (Swap): We maintain two delta tables during the merge process (i.e., using double buffering). The old delta table is read-only and can be accessed lock-free for DMA transfer to the GPU. Meanwhile, a new empty delta table is updated by in-flight transactions. To swap the delta table, the merge process notifies all TP threads to write to the new delta table. It then waits until all threads finish updating the old delta table and switch to the new one.

Phase 3 (Update): The delta log in the old delta table is transmitted to the GPU and applied in parallel. Transmitting and applying logs is pipelined to hide the transmit latency. Updating the column store is shown in Algorithm 4. For each tuple in the delta log, if its current version was committed before the snapshot (i.e., $e.cts_{\max} \leq sv$), its data is applied directly to the column store based on its row ID (lines 6-8), and its timestamp in the SCV is reset to ∞. If the tuple is deleted, the timestamp is set to 0, making it invisible to all subsequent queries (i.e., $SCV[rid] < rts$) and should be skipped during GPU scans (lines 4-5).

Notably, the old delta table may contain tuples committed after the snapshot version (i.e., $e.cts_{\max} > sv$). This is because acquiring sv and swapping the delta table are not atomic operations. In other words, some updates may be inserted into the old delta table after sv is acquired but before the swap completes (≈1034 rows with 32 TP clients). To generate a complete snapshot, a naive approach would fetch the correct versions of these tuples from the multi-version row store, incurring CPU-GPU coordination overhead.

Instead, we skip applying the data and filter logs of these tuples (line 9). This optimization is safe because: (i) their filter logs will be applied by subsequent queries, since their latest versions are stored in the new delta table (after write-back in Phase 4); (ii) their data in the column store will never be accessed by queries. This is because queries started after the merge will have read timestamps rts that are no less than the commit timestamps $cts_{\max}$ of all transactions committing during the merge (Lemma 4.1). This results in $SCV[e.rid] \geq rts$, causing these tuples to be skipped during scans on the GPU path (line 8 in Algorithm 3).

Phase 4 (Write-back): As discussed above, some tuples in the old delta table may have commit timestamps $cts_{\max}$ greater than the snapshot version sv of the column store. To ensure the CPU subquery observes these updates, these tuples must be written back to the new delta table.

Specifically, a group of CPU threads scans the old delta table to identify these tuples (i.e., those with $cts_{\max} > sv$). Since a tuple may be updated multiple times after sv (the delta log only stores a single version), the correct version inserted into the new delta table is obtained from the multi-version row store. Write-back these tuples follow the same logic as TP threads updating the delta table (i.e., `TxUpdateRecord` in Algorithm 2), ensuring concurrency correctness. New queries can execute immediately after the column-store update and the tuple write-back are complete (starting from read timestamp acquisition). The old delta table is reclaimed asynchronously in the background.

6 IMPLEMENTATION

6.1 Transaction Execution Engine

The row store is built on a Persistent Memory (PM) based OLTP engine to achieve high TP performance and fast recovery [6, 25, 26, 38]. `HeraDB` organizes the tuple heap at row level rather than page level, leveraging PM's byte-addressability. This avoids page-level read/write amplification in disk-based stores and improves transaction throughput. Since row-store tables are persisted in PM, `HeraDB` can restart without rebuilding them from checkpoints.

Notably, `HeraDB`'s architecture is not tightly coupled to PM and can be integrated with other timestamp-based MVCC OLTP engines (e.g., Postgres, Hekaton [20], HYRISE [21]). This is because `HeraDB` decouples delta table updates from transaction execution. Specifically, worker threads update the delta table *after* transactions are committed, which is loosely coupled to transaction execution.

Transaction processing. Transactions are executed under snapshot isolation. The TP engine follows a transaction-at-a-time execution model, where each worker thread executes one transaction sequentially. It employs in-place updates with undo logging. For tuple updates, the transaction first copies the current version from the tuple heap to the thread-local undo log, then writes the new version directly in place. The new version is also stored in the transaction's write set, which is used to update the delta table after commit. In addition to supporting transaction rollback, the undo log also serves as the historical version storage for MVCC, eliminating the overhead of maintaining separate version chains.

Index. We employ in-memory hash tables for performance. Indexes are stored separately from tuples, with the indexed field as the key and the row ID as the value. Since we adopt the in-place update approach, tuple updates do not require index modifications.

Garbage collection. We utilize a background thread to reclaim stale tuple versions in the historical version storage (i.e., the undo logs). Since a version may be accessed by both transactions and CPU subqueries, versions with commit timestamps smaller than the read timestamps of all in-flight queries and transactions can be safely reclaimed (by truncating the undo logs).

6.2 Query Execution Engine

The column store on the GPU is implemented using CUDA with the Crystal library [58]. Crystal adopts a tile-based execution model where each thread block collectively processes a tile of elements. By loading tiles into shared memory, Crystal enables multiple operations (e.g., filtering, projection, and aggregation) to execute in a single pass without materializing intermediate results, significantly reducing GPU global memory traffic.

To support hybrid query execution, the SCV is allocated as an additional column array in the GPU global memory. When executing a GPU subquery, after loading a tile of column data, we also load the corresponding SCV tile and generate the selection vector (Section 4.3) in the shared memory to filter out stale rows. Applying delta logs and filter logs is implemented as separate kernels.

6.3 Recovery

After a crash, the delta table and GPU column store are lost. During recovery, the row store scans the undo logs generated by TP worker threads to roll back uncommitted transactions and immediately accepts new transactions, while queries are blocked until the column store is fully rebuilt.

To minimize recovery time, delta updates are applied concurrently while the column store is rebuilt. Specifically, the transaction workflow remains unchanged. Updates are written to the delta table and periodically merged into the column store. Meanwhile, the uninitialized rows in the column store (rows that are not updated by transactions after restart) are filled from the row store. Multiple background threads scan the row store and transmit the row data to the GPU to update these uninitialized rows. This filling process is suspended when merging the delta table to prevent write-write conflicts to the same row. Additionally, it also skips any rows that have already been initialized to avoid overwriting more recent updates.

7 EVALUATION

7.1 Experimental Setup

Workloads. We utilize HATtrick [44] and SSB-TC for evaluation. HATtrick executes analytical queries using the Star Schema Benchmark (SSB) [48] and runs transactions based on a modified version of TPC-C [18]. The AP workload consists of all 13 SSB queries. The TP workload comprises three transaction types: NewOrder (48%, insert-only), Payment (48%, updates and inserts), and CountOrders (4%, read-only). In HATtrick, ≈71.4% of write operations are inserts.

However, in real-world transaction-centric workloads, a higher proportion of write operations are updates [73]. Therefore, we implement SSB-TC, an update-heavy variant of HATtrick that combines SSB with YCSB [17]. SSB-TC uses the same AP workload as HATtrick. The TP workload is similar to YCSB. Each transaction randomly selects multiple rows (10 by default) from the LINEORDER table and updates one non-key column per row. We use skew factors to control access patterns, with skew factors θ of 0 (Uniform, no hotspots) and 0.99 (Zipfian, with hotspots). The default scale factor (SF) for HATtrick and SSB-TC is 100.

External baselines. We compare `HeraDB` with external baselines to evaluate data freshness and performance isolation.

- **Colibri** [57]: a high-performance HTAP database that adopts the co-located architecture. It primarily stores data in columnar format, with hot data (i.e., delta) stored in an uncompressed row format. We use Colibri as the co-located HTAP baseline.
- **TiDB** [24]: an enterprise-grade HTAP database that adopts the separated architecture. It consists of a row store (TiKV) and a column store (TiFlash). Updates on TiKV are propagated to TiFlash using Raft [49] log replication. We use TiDB as the separated HTAP baseline.
- **DuckDB** [52]: an embedded OLAP database with high-performance AP. It employs a pure column format. We use DuckDB as the standalone AP baseline.
- **Postgres**: a row-store database optimized for TP. We use Postgres as the standalone TP baseline.

Internal variants. We also implement variants on top of `HeraDB` to evaluate the effectiveness of hybrid CPU-GPU query execution. For `HeraDB` and its variants, the delta table is periodically merged into the column store (i.e., merge interval) every 400 ms.

- **UM**: simulates Caldera's execution. It maintains identical column-store replicas on both CPU and GPU. After TP updates the row store, it also updates the CPU column store, with CUDA Unified Memory maintaining consistency across the CPU and GPU column stores. We include `UM` only in Section 2.3 due to its low performance.
- **Merge**: simulates RateupDB's execution. It requires each query to scan the row store and generate its own query-specific delta updates before execution, as different queries operate on distinct snapshots. The updates are then transmitted to the GPU and accessed during the column-store scan.
- **Batch**: simulates [55]'s execution. It updates the column store every 400ms. Queries execute only on the last GPU snapshot.

In the evaluation, Figure 10 and Table 2 compare `HeraDB` with external baselines and relevant internal variants. Figures 3, 13, 14, 17, and 18 use internal variants to isolate design choices, while Figures 11, 12, 15, and 16 focus on `HeraDB`'s performance behavior.

Physical environment. The experiments are conducted on a server equipped with four Intel Xeon Platinum 8360H CPUs (each having 48 vCPUs) and one NVIDIA RTX A6000 GPU (48 GB). The server is configured with 1.5 TB of DRAM and 3 TB of PM, running CentOS 7 with NVCC 12.4 and GCC 10.3.

All systems, except TiDB, are deployed on a single NUMA node. To simulate a distributed deployment, TiDB is deployed across three separate NUMA nodes: TiKV on node #1, TiFlash on node #2, and other components (e.g., PD) on node #3. By default, we use 32 TP clients (or threads) and 3 AP clients, and run experiments on SSB-TC under the uniform distribution. Each AP query executes with a degree of parallelism (DOP) of 4.

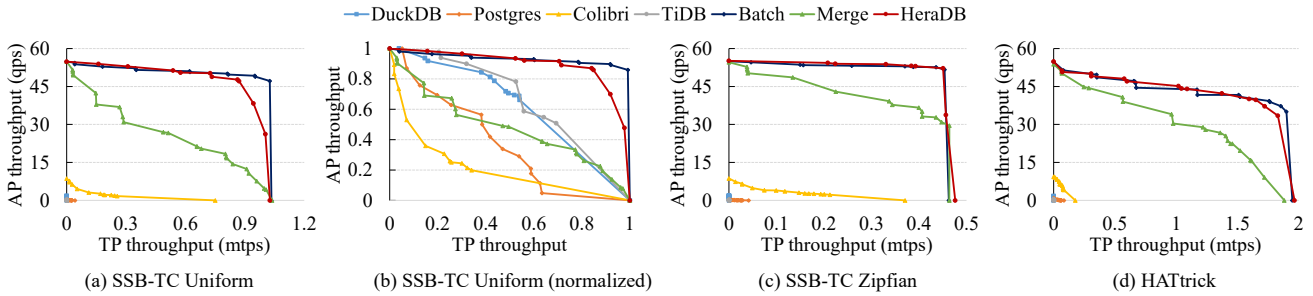


Figure 10: Comparison of throughput frontiers under SSB-TC and HATtrick (SF=100). Each frontier is constructed by connecting the best observed TP/AP throughput points from independent runs with different TP/AP client mixes.

7.2 Overall Performance

SSB-TC. Figure 10a shows the throughput frontiers under uniform distribution. HeraDB exhibits strong isolation approaching Batch, with only minor degradation due to resource contention from the CPU subquery during delta-table scanning. Merge requires each query to generate its own delta updates based on its snapshot version, leading to contention for CPU and memory bandwidth.

Figure 10b shows the normalized results to better illustrate performance isolation, since the large performance gap across systems makes the isolation behavior difficult to compare in the raw plot. Although TiDB has lower absolute throughput than Postgres in Figure 10a, its frontier is closer to the upper-right corner of the bounding box in Figure 10b, indicating better isolation. This is because its TP and AP resources are physically separated. DuckDB exhibits isolation comparable to TiDB. This is because its maximum TP throughput is much lower (47 vs. 2961 tps), resulting in fewer updates. Colibri stores hot tuples in row format, which benefits TP performance. However, its TP throughput drops sharply even with a small AP workload, indicating poor isolation.

Figure 10c shows the results under the Zipfian distribution. Since the delta table uses in-place updates, multiple updates on the same hot tuple result in a smaller delta size than under uniform distribution (0.34× the size with 32 TP and 3 AP clients). This reduces scanning overhead on the CPU path and moves the throughput frontier of HeraDB closer to Batch. The maximum TP throughput across all systems decreases due to high write-write conflicts on hot tuples, resulting in high abort rates (69% for HeraDB).

HATtrick. Figure 10d shows the throughput frontiers of HATtrick. Compared with the best external baseline, Colibri, HeraDB achieves 11.1× and 5.9× higher maximum TP and AP throughput, as shown by the points on the x- and y-axes, respectively. All systems exhibit lower throughput frontiers for two reasons. First, the high TP throughput of HATtrick results in a larger delta table (for HeraDB, the LINEORDER table is 1.38× larger than under SSB-TC Uniform), increasing merge overhead. Second, HATtrick also updates dimension tables replicated in both CPU and GPU memory. Updating dimension tables in the CPU becomes a bottleneck.

Latency. Figure 11 reports AP tail latency as the AP clients increases. The TP clients is fixed to 32. AP latency rises with AP concurrency because query execution is GPU-bound, causing concurrent queries compete for GPU execution resources. The delta merge introduces low overhead. Compared with the uniform workload, Zipfian AP latency is slightly lower because repeated updates

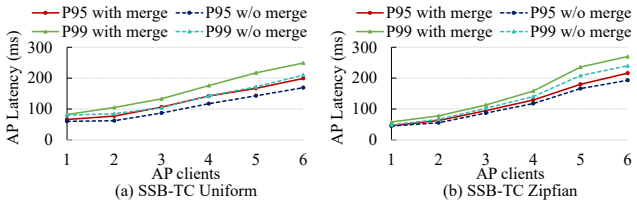


Figure 11: AP query tail latency with varying AP clients.

Table 2: Comparison of visibility latency with varying the number of TP clients (SSB-TC Uniform, 3 AP clients)

Metric	System	5	10	15	25	32
Freshness latency (ms)	TiDB	204	753	827	1785	2349
	Batch	162	164	164	165	167
	HeraDB	0	0	0	0	0
HeraDB read-ts wait (ms)		0.0020	0.0035	0.0048	0.0061	0.0101

on hot rows are combined by in-place updates, reducing the number of distinct delta tuples scanned by the CPU path and applied during merge. TP latency remains nearly unchanged as AP concurrency increases. Under SSB-TC Uniform, the average TP latency increases only from 33 us to 38 us, while TP P95 stays within 43–44 us across 1–6 AP clients, indicating high performance isolation.

Freshness. We measure data freshness using visibility latency [62], which is the time difference between the query start time and the commit time of the latest transaction included in the snapshot used by the query. Table 2 reports the results. For DuckDB, Postgres, and Colibri, TP and AP operate on the same replica, resulting in zero visibility latency. Merge achieves zero visibility latency because each query transmits the latest delta before execution. TiDB has high visibility latency due to Raft log replication. Batch incurs high visibility latency because it only periodically updates the column snapshot. HeraDB also provides zero visibility latency. The additional 2.0–10.1 us shown in Table 2 is the coordination time for acquiring a consistent read timestamp before query execution.

7.3 Performance Analysis

Aggregatable query execution. Table 3 shows the query latency breakdown. We execute each query individually with concurrent TP. New queries incur a wait time of 4-13 ms because they must wait for the ongoing merge to finish updating the column store. The

Table 3: Breakdown of AP execution time (SSB-TC Uniform)

	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3
Wait (ms)	5.84	3.83	3.66	8.21	7.89	7.30	10.67	5.82	5.28	4.96	13.16	11.50	6.38
CPU (ms)	11.82	12.10	11.77	32.72	33.07	32.03	40.93	37.69	38.30	36.26	46.45	45.51	42.95
GPU (ms)	44.91	33.63	32.69	60.77	59.28	58.32	74.23	47.88	41.10	42.58	82.94	79.78	51.11
Agg. (ms)	0.00	0.00	0.00	0.02	0.02	0.02	0.02	0.91	0.91	0.90	0.00	0.02	1.05
Total (ms)	51.38	38.16	37.08	69.93	68.16	66.76	86.11	59.02	54.79	54.11	97.30	92.63	64.00
QPS	58.25	78.39	80.66	42.83	43.94	44.85	34.79	50.36	54.16	54.87	30.78	32.34	46.80

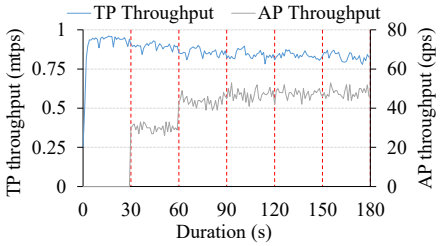


Figure 12: Throughput with varying AP clients over time (SSB-TC Uniform).

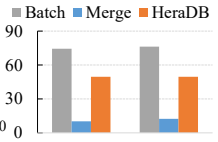
wait time varies with query complexity due to cascading waits: the merge process must wait for in-flight queries to complete before updating the column store, and new queries must wait for the merge to finish. Therefore, complex queries increase the merge wait time, which in turn increases the wait time for new queries. The CPU subqueries complete quickly due to a small delta table size, making GPU execution the bottleneck. Updating SCV on the GPU takes ≈1.97ms, which is relatively low. The aggregation cost of queries Q1.1-1.3 is negligible because their result sets are small.

```
SELECT
  c.custkey, c.name, lo.orderkey
FROM customer c
FULL JOIN lineorder lo
  ON c.custkey = lo.custkey
  AND c.custkey < 1000
  AND lo.orderkey < 100000
LIMIT 1000;
```

(a) Outer join schema

```
SELECT c.custkey, c.nation
FROM customer c
WHERE c.nation < 2
  AND c.custkey IN (
    SELECT lo.custkey
    FROM lineorder lo
    WHERE lo.orderkey < 100000
);
```

(b) Semi-join schema



(c) AP throughput

Figure 13: Results of non-aggregatable query execution.

Non-aggregatable query execution. Figure 13 evaluates the GPU-only fallback path for non-aggregatable queries (discussed in Section 4.3). HeraDB has lower AP throughput than Batch because it must transfer filtered delta tuple data at query time rather than timestamp filters, with tuple transfer accounting for about 30% of execution time. However, HeraDB preserves zero visibility latency, whereas Batch reads the last merged GPU snapshot. Compared with Merge, HeraDB avoids transmitting the entire delta table by applying predicate and column pushdown. For outer joins, pushdown LIMIT operator further reduces transmitted rows. Thus, HeraDB substantially outperforms Merge while preserving freshness.

Performance over time. To demonstrate performance isolation, we start with 32 TP clients and gradually add AP clients (one every 30 seconds) over 180 seconds. Figure 12 shows how TP and AP throughput change as the number of AP clients increases from 0 to 5. TP throughput remains stable with only a 12% decline (from 0.93 to 0.82 mtps), indicating strong isolation between TP and AP workloads. Query throughput increases with the number of AP clients. However, since queries are GPU-bound, AP throughput remains stable after the number of AP clients exceeds 3.

Transaction execution. Table 4 shows the breakdown of transaction execution time. The TP workflow includes (i) updating the row store during execution (i.e., Execution) and (ii) updating the delta table after commit (i.e., Update Delta). The total time of SSB-TC is longer than HATtrick due to its larger read/write sets. TP threads

Table 4: Breakdown of TP execution time (SSB-TC Uniform)

	SSB-TC Uniform	SSB-TC Zipfian	HATtrick
Execution (ms)	0.027 (77.1%)	0.029 (78.4%)	0.015 (83.3%)
Update Delta (ms)	0.008 (22.9%)	0.008 (21.6%)	0.003 (16.7%)
Total (ms)	0.035	0.037	0.018

spend 16.7%–22.9% of CPU time updating the delta table. This overhead does not affect end-to-end latency because delta table updates are performed after transaction commit. The cost is relatively low due to fast delta-index lookup. In contrast, our UM variant spends 50.9% of CPU time making updates visible to AP queries.

Varying scale factors. Figure 14 shows the throughput results as the scale factor increases from 20 to 100. All systems achieve similar TP throughput. This is because concurrent AP queries or delta merges do not block transaction execution.

At lower scale factors (e.g., SF=20), the AP throughput of HeraDB is lower than that of Batch. This is because the column store is relatively small. Scanning on the GPU path completes quickly, making scanning the delta table on the CPU path a bottleneck. When the scale factor is higher (i.e., SF≥60), the AP throughput is similar. Although the delta table size remains unchanged (since TP throughput is constant), the AP throughput of Merge is low because generating and transmitting query-specific delta updates becomes a bottleneck.

Scaling TP clients across NUMA nodes. We scale the number of TP clients from 32 to 176 across four NUMA nodes, while fixing the merge interval to 100 ms. Figure 16 shows the results. As TP clients increase, the proportion of rows fetched from the MV row store remains low, increasing from 0.02% at 32 clients to a peak of 0.15% at 112 clients. The time required to scan TP slots increases from 3.5 us to 10 us, which remains negligible. TP and AP throughput decrease beyond 48 clients is due to NUMA effects in the TP engine.

GPU memory consumption. At SSB-TC SF100, HeraDB uses 20.12GB (87.7%) for column tables, 2.23GB (9.7%) for SCV, and 0.58 GB (2.5%) for delta/filter-log buffers for merge. The selection flags and most of the intermediate results are stored within the GPU cache and do not consume GPU memory. The memory requirement is modeled as: $0.87\text{GB} + SF \times 0.22\text{GB}$. The log buffers and runtime structures consume 0.87 GB. The column store tables and their SCVs consume 0.22GB per SF.

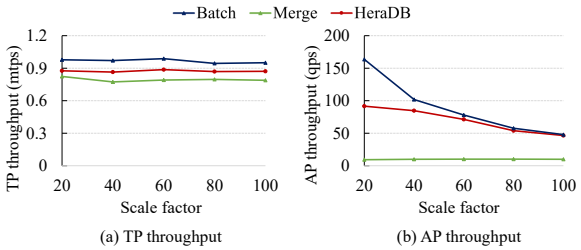


Figure 14: Throughput varying scale factors.

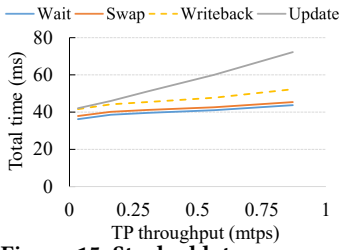


Figure 15: Stacked latency breakdown of delta merge.

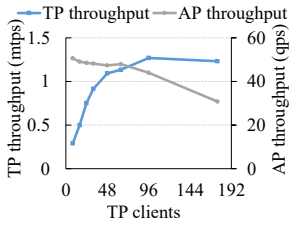


Figure 16: Throughput scaling TP clients across four NUMA nodes.

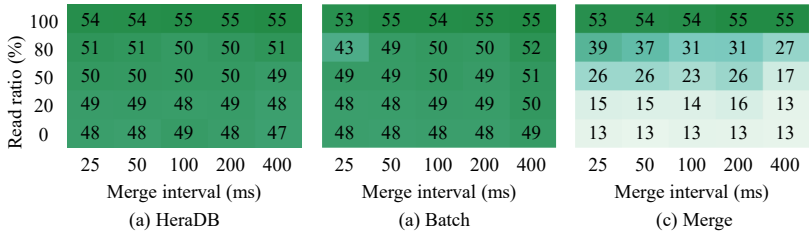


Figure 17: AP throughput (qps) with varying read ratios and merge intervals.

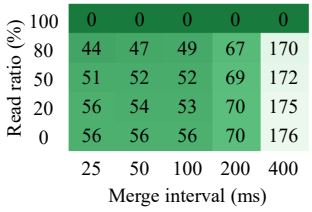


Figure 18: Visibility latency (ms) of Batch varying read ratios and merge intervals.

7.4 Merge Performance

Execution time breakdown. Figure 15 presents a stacked latency breakdown of delta merge with increasing TP throughput. As the number of TP clients increases from 1 to 32, TP throughput rises to 0.87 mtps, with 10.4 million tuple updates per second inserted into the delta table. As a result, the latency for transmitting and applying updates (i.e., `Update`) increases linearly.

Since the delta merge cannot execute concurrently with queries, waiting for in-flight queries to complete (i.e., `Wait`) is necessary. Because AP performance remains stable as TP throughput increases, the wait time is relatively constant. The overhead for tuple write-back (i.e., `Writeback`) and delta table swapping (i.e., `Swap`) remains stable, as they are independent of the delta table size (Section 5.2).

Varying merge interval. We vary the merge interval together with the read ratio under SSB-TC Uniform to identify the breaking points of `Batch` and `Merge`. Figure 17 and Figure 18 report the AP throughput and visibility latency. We omit the freshness results of `HeraDB` and `Merge` because they remain zero across all configurations. The TP results remains stable and is thus omitted, as the merge process does not block transaction execution.

As shown in Figure 18, `Batch` maintains AP throughput close to `HeraDB`, but this comes at the cost of stale snapshots. As the workload becomes more write-intensive, more updates accumulate between merges, increasing the time required to install a new GPU snapshot. Consequently, recently committed updates remain invisible for a longer time, leading to higher visibility latency. Even with short merge intervals, this freshness gap remains because each merge still requires blocking concurrent queries and applying delta logs before queries can observe the new snapshot.

As shown in Figure 17, `Merge` achieves freshness by sending query-specific delta logs to the GPU before each query, but this makes AP throughput scale inversely with the delta size. Lower read ratios generate more updates, and longer merge intervals cause updates to accumulate for longer before they are written to the GPU column store. Both factors increase the amount of delta data transferred per query and decrease `Merge`'s AP throughput.

8 RELATED WORK

GPU-accelerated query processing. Early GPU-accelerated OLAP databases [22, 66, 72] adopt an operator-at-a-time model, materializing intermediate results to execute queries. Crystal [58] proposes a tile-based execution model where a block of threads processes data cooperatively, thereby minimizing GPU memory traffic. Crystal is further optimized by Crystal-Opt [12] using lazy materialization and DMO-DB [36] via inter-pipeline data movement. Other optimizations include warp-level load balancing [23] and prefetching [19]. Systems such as Lancelot [69], [43], and Vortex [71] extend query processing to multiple GPUs. Although our prototype is built on Crystal, `HeraDB` can extend to systems with a tile-based execution model by adding an SCV column to each table and implementing kernels to apply delta and filter logs.

Heterogeneous OLAP databases, such as Mordred [70], GOLAP [7], and [35], primarily manage data in host memory, using the GPU as a coprocessor optimized via caching or filtering. In contrast, `HeraDB` targets transaction-centric HTAP workloads to ensure strong performance isolation for mission-critical TP.

High-freshness HTAP databases. The evolution of HTAP architectures is driven by performance and data freshness [10, 11, 27, 28, 32, 33, 40, 41, 46, 50, 53, 56, 59, 65, 67]. To ensure freshness, Perseus [62] and AETS [76] reduce the freshness gap by selective log propagation and prioritized log replay, respectively. However, they increase query latency because queries must wait for pending updates to be applied. ByteHTAP [14] enhances freshness via efficient log replication, fast log apply, and efficient delete handling. veDB-HTAP [15] mitigates the freshness gap with multi-level LSNs. Co-located systems such as Colibri [57], Data Blocks [30], Proteus [1], and FSM [5] inherently eliminate data freshness gaps by maintaining a single database replica. [54] dynamically shifts

CPU and memory resources between TP and AP engines to satisfy changing freshness requirements while balancing TP and AP performance. However, their techniques are non-trivial to extend to H^{2}TAP databases, because GPUs rely on a tile-based execution model that differs fundamentally from the CPU thread model.

9 CONCLUSION

This paper presents HeraDB, a GPU-accelerated HTAP database that enables fresh analytical queries with strong performance isolation. HeraDB employs a hybrid CPU-GPU query execution to efficiently access recent updates while minimizing PCIe traffic, and proposes a unified delta table format that enables lock-free CPU scanning and efficient GPU merging. Evaluation on HATtrick shows that HeraDB outperforms Colibri in maximum individual TP/AP throughput, achieving up to 11.1× higher TP and 5.9× higher AP with high data freshness and strong performance isolation.

REFERENCES

[1] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. Proteus: Autonomous Adaptive Storage for Mixed Workloads. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 700–714. https://doi.org/10.1145/3514221.3517834

[2] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.

[3] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1426–1438. https://doi.org/10.14778/3583140.3583157

[4] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The Case For Heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf

[5] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 583–598. https://doi.org/10.1145/2882903.2915231

[6] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 337–348. https://doi.org/10.14778/3025111.3025116

[7] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6, Article 237 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698812

[8] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[9] Amirali Boroumand, Saugata Ghose, Geraldo F. Oliveira, and Onur Mutlu. 2022. Polynesia: Enabling High-Performance and Energy-Efficient Hybrid Transactional/Analytical Databases with Hardware/Software Co-Design. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2997–3011. https://doi.org/10.1109/ICDE53745.2022.00270

[10] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 617–630. https://doi.org/10.1145/3514221.3517845

[11] Dennis Butterstein, Daniel Martin, Knut Stolze, Felix Beier, Jia Zhong, and Lingyun Wang. 2020. Replication at the speed of change: a fast, scalable replication solution for near real-time HTAP processing. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3245–3257. https://doi.org/10.14778/3415478.3415548

[12] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 441–454. https://doi.org/10.14778/3632093.3632107

[13] Shaosheng Cao, XinXing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. TitAnt: online real-time transaction fraud detection in Ant Financial. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2082–2093. https://doi.org/10.14778/3352063.3352126

[14] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3411–3424. https://doi.org/10.14778/3554821.3554832

[15] Jianjun Chen, Li Zhang, Yu Xie, Wei Ding, Lixun Cao, Ye Liu, Yonghua Ding, Fangshi Li, Ke Wu, Haibo Xiu, Kui Wei, Le Cai, Rui Chang, Yuxiang Chen, Yuanjin Lin, Shangyu Luo, Jianfeng Qian, Xu Wang, Zikang Wang, Jian Zhang, Mingyi Zhang, Shicai Zeng, Jason Sun, Lei Zhang, Rui Shi, and Pengwei Zhao. 2025. veDB-HTAP: A Highly Integrated, Efficient and Adaptive HTAP System. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 4896–4909. https://doi.org/10.14778/3750601.3750614

[16] Steven Chien, Ivy Peng, and Stefano Markidis. 2019. Performance Evaluation of Advanced Features in CUDA Unified Memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 50–57. https://doi.org/10.1109/MCHPC49590.2019.00014

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[18] Transaction Processing Council. 2026. TPC-C Benchmark. https://www.tpc.org/tpcc/ Accessed: 2026-01-10.

[19] Yangshen Deng, Shiwen Chen, Zhaoyang Hong, and Bo Tang. 2024. How Does Software Prefetching Work on GPU Query Processing?. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, AA, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 5, 9 pages. https://doi.org/10.1145/3662010.3663445

[20] 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* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[21] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: a main memory hybrid storage engine. *Proc. VLDB Endow.* 4, 2 (Nov. 2010), 105–116. https://doi.org/10.14778/1921071.1921077

[22] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.* 6, 9 (July 2013), 709–720. https://doi.org/10.14778/2536360.2536370

[23] Kijae Hong, Kyoungmin Kim, Young-Koo Lee, Yang-Sae Moon, Sourav S Bhowmick, and Wook-Shin Han. 2024. Themis: A GPU-Accelerated Relational Query Execution Engine. *Proc. VLDB Endow.* 18, 2 (Oct. 2024), 426–438. https://doi.org/10.14778/3705829.3705856

[24] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[25] Zhicheng Ji, Kang Chen, Leping Wang, Mingxing Zhang, and Yongwei Wu. 2023. Falcon: Fast OLTP Engine for Persistent Cache and Non-Volatile Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 531–544. https://doi.org/10.1145/3600006.3613141

[26] Yihe Jiang, Zeshun Peng, Yanfeng Zhang, and Ge Yu. 2025. Survey on Database Index Structures on Non-volatile Memory. *Journal of Chinese Computer Systems* 46, 9, Article 2291 (2025), 2291-2304 pages. http://xwxt.sict.ac.cn/CN/Y2025/V46/I9/2291

[27] 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*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[28] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 49–64. https://doi.org/10.1145/3514221.3526135

[29] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zait. 2015. Oracle Database In-Memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. 1253–1258. https://doi.org/10.1109/ICDE.2015.7113373

[30] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

[31] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1740–1751. https://doi.org/10.14778/2824032.2824071

[32] 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. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1598–1609. https://doi.org/10.14778/3137765.3137767

[33] Kitaek Lee, Insoon Jo, Jaechan Ahn, Hyuk Lee, Hwang Lee, Woong Sul, and Hyungsoo Jung. 2023. Deploying Computational Storage for HTAP DBMSs Takes More Than Just Computation Offloading. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1480–1493. https://doi.org/10.14778/3583140.3583161

[34] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product. *Proc. VLDB Endow.* 14, 12 (July 2021), 2999–3013. https://doi.org/10.14778/3476311.3476378

[35] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, Kaushik Rajan, Surajit Chaudhuri, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases Beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (July 2025), 4518–4531. https://doi.org/10.14778/3749646.3749710

[36] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Joonsung Kim, Jinho Lee, and Youngsok Kim. 2025. DMO-DB: Mitigating the Data Movement Bottlenecks of GPU-Accelerated Relational OLAP. In *2025 34th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 172–185. https://doi.org/10.1109/PACT65351.2025.00026

[37] Alexandre AB Lima, Camille Furtado, Patrick Valduriez, and Marta Mattoso. 2009. Parallel OLAP query processing in database clusters with data replication. *Distributed and Parallel Databases* 25, 1 (2009), 97–123.

[38] Gang Liu, Leying Chen, and Shimin Chen. 2021. Zen: a high-throughput log-free OLTP engine for non-volatile main memory. *Proc. VLDB Endow.* 14, 5 (Jan. 2021), 835–848. https://doi.org/10.14778/3446095.3446105

[39] Guy M Lohman. 1988. R* optimizer validation and performance evaluation for distributed queries. *Readings in Database Systems* (1988), 219.

[40] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi (Eds.). OpenProceedings.org, 1–12. https://doi.org/10.5441/002/EDBT.2019.02

[41] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2530–2542. https://doi.org/10.1145/3448016.3457562

[42] 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* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 37–50. https://doi.org/10.1145/3035918.3035959

[43] Tobias Maltenberger, Ilin Tolovski, and Tilmann Rabl. 2025. Efficiently Joining Large Relations on Multi-GPU Systems. *Proc. VLDB Endow.* 18, 11 (July 2025), 4653–4667. https://doi.org/10.14778/3749646.3749720

[44] 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* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1810–1824. https://doi.org/10.1145/3514221.3526148

[45] Elena Milkai, Xiangyao Yu, and Jignesh M. Patel. 2025. Hermes: Off-the-Shelf Real-Time Transactional Analytics. *Proc. VLDB Endow.* 18, 8 (April 2025), 2334–2347. https://doi.org/10.14778/3742728.3742731

[46] Niloy Mukherjee, Shasank Chavan, Maria Colgan, Dinesh Das, Mike Gleeson, Sanket Hase, Allison Holloway, Hui Jin, Jesse Kamp, Kartik Kulkarni, Tirthankar Lahiri, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Atrayee Mullick, Andy Witkowski, Jiaqi Yan, and Mohamed Zait. 2015. Distributed architecture of Oracle database in-memory. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1630–1641. https://doi.org/10.14778/2824032.2824061

[47] NVIDIA Corporation. 2026. *Unified Memory.* NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/unified-memory.html

[48] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–252.

[49] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[50] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2340–2352. https://doi.org/10.1145/3514221.3526055

[51] Jon T.S. Quah and M. Sriganesh. 2008. Real-time credit card fraud detection using computational intelligence. *Expert Systems with Applications* 35, 4 (2008), 1721–1732. https://doi.org/10.1016/j.eswa.2007.08.093

[52] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[53] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU acceleration: so much more than just a column store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[54] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through Elastic Resource Scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2043–2054. https://doi.org/10.1145/3318464.3389783

[55] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf

[56] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F. Ilyas. 2023. Real-Time LSM-Trees for HTAP Workloads. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 1208–1220. https://doi.org/10.1109/ICDE55515.2023.00097

[57] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. *Proc. VLDB Endow.* 17, 11 (July 2024), 3290–3303. https://doi.org/10.14778/3681954.3682001

[58] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[59] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 219–238. https://www.usenix.org/conference/osdi21/presentation/shen

[60] 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* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 731–742. https://doi.org/10.1145/2213836.2213946

[61] 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

[62] Haoze Song, Xusheng Chen, Ruijie Gong, Zekai Sun, Tianxiang Shen, Cheng Li, Hao Feng, Sen Wang, and Heming Cui. 2025. Perseus: Achieving Strong Consistency and High Data Freshness for Scalable Geo-distributed HTAP. *Proc. ACM Manag. Data* 3, 4, Article 260 (Sept. 2025), 29 pages. https://doi.org/10.1145/3749178

[63] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. 2015. Latency critical big data computing in finance. *The Journal of Finance and Data Science* 1, 1 (2015), 33–41. https://doi.org/10.1016/j.jfds.2015.07.002

[64] Alejandro Vera-Baquero, Ricardo Colomo-Palacios, and Owen Molloy. 2016. Real-time business activity monitoring and analysis of process performance on big-data domains. *Telematics and Informatics* 33, 3 (2016), 793–807. https://doi.org/10.1016/j.tele.2015.12.005

[65] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and

Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data* 1, 2, Article 199 (June 2023), 25 pages. https://doi.org/10.1145/3589785

[66] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2018. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 44–54. https://doi.org/10.1145/2544137.2544166

[67] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeff Naughton, and John Cieslewicz. 2020. F1 lightning: HTAP as a service. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3313–3325. https://doi.org/10.14778/3415478.3415553

[68] Jiani Yang, Sai Wu, Yong Wang, Dongxiang Zhang, Yifei Liu, Xiu Tang, and Gang Chen. 2025. Twisted Twin: A Collaborative and Competitive Memory Management Approach in HTAP Systems. *Proc. VLDB Endow.* 18, 10 (June 2025), 3312–3325. https://doi.org/10.14778/3748191.3748197

[69] Bobbi Yogatama, Weiwei Gong, and Xiangyao Yu. 2024. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. *Proc. VLDB Endow.* 17, 13 (Sept. 2024), 4709–4722. https://doi.org/10.14778/3704965.3704977

[70] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating data placement and query execution in heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (July 2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

[71] Yichao Yuan, Advait Iyer, Lin Ma, and Nishil Talati. 2024. Vortex: Overcoming Memory Capacity Limitations in GPU-Accelerated Large-Scale Data Analytics. *Proc. VLDB Endow.* 18, 4 (Dec. 2024), 1250–1263. https://doi.org/10.14778/3717755.3717780

[72] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 817–828. https://doi.org/10.14778/2536206.2536210

[73] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. *Proc. VLDB Endow.* 17, 5 (Jan. 2024), 939–951. https://doi.org/10.14778/3641204.3641206

[74] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. 2024. HTAP Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 11 (2024), 6410–6429. https://doi.org/10.1109/TKDE.2024.3389693

[75] Yilong Zhao, Mingyu Gao, Huanchen Zhang, Fangxin Liu, Gongye Chen, He Xian, Haibing Guan, and Li Jiang. 2025. PUSHtap: PIM-based In-Memory HTAP with Unified Data Storage Format. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 179–194. https://doi.org/10.1145/3676642.3736120

[76] Jun-Peng Zhu, Zhiwei Ye, Peng Cai, Donghui Wang, Fengyan Zhang, Dunbo Cai, and Ling Qian. 2024. Log Replaying for Real-Time HTAP: An Adaptive Epoch-Based Two-Stage Framework. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2096–2108. https://doi.org/10.1109/ICDE60146.2024.00167