HaSiS: A Hardware-assisted Single-index Store for Hybrid Transactional and Analytical Processing

Kecheng Huang, *The Chinese University of Hong Kong;* Zhaoyan Shen, *Shandong University;* Zili Shao, *The Chinese University of Hong Kong;* Feng Chen, *Indiana University Bloomington;* Tong Zhang, *Rensselaer Polytechnic Institute and ScaleFlux Inc.*

https://www.usenix.org/conference/fast25/presentation/huang

This paper is included in the Proceedings of the 23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

Open access to the Proceedings of the 23rd USENIX Conference on File and Storage Technologies is sponsored by

HaSiS: A Hardware-assisted Single-index Store for Hybrid Transactional and Analytical Processing

Kecheng Huang[†], Zhaoyan Shen[‡*], Zili Shao[†], Feng Chen[§] and Tong Zhang[♮♦]

[†]The Chinese University of Hong Kong, [‡]Shandong University, [§]Indiana University Bloomington, [♮]Rensselaer Polytechnic Institute and [♦]ScaleFlux Inc.

Abstract

Driven by the exploding demands for real-time data analytics, hybrid transactional and analytical processing (HTAP) has become a topic of great interest in academia and the database industry. To address the well-known conflict between optimal storage formats for online transactional processing (OLTP) and online analytical processing (OLAP), the conventional practice employs a mixture of at least two distinct index data structures (e.g., B$^+$-tree and column-store) and dynamically migrates data across different index domains. Unfortunately, such a *multi-index* design is notably subject to non-trivial trade-offs among OLTP performance, OLAP performance, and OLAP data freshness. In contrast to prior work that centered around exploring the multi-index design space, this work advocates a *single-index* design for a paradigm shift towards much more effectively serving HTAP workloads. This is made possible by computational storage drives (CSDs) with built-in transparent compression that are emerging on the commercial market. The key is to exploit the fact that compression-capable CSDs enable data management software to purposefully employ *sparsely filled* storage data blocks without sacrificing physical storage capacity. Leveraging this unique feature, we have developed an HTAP-oriented B$^+$-tree design that can effectively serve HTAP workloads and in the meantime can achieve almost instant OLAP data freshness. We have developed and open-sourced a fully functional prototype. Our results show that compared to the state-of-the-art solutions, such a CSD-assisted single-index design can ensure data freshness and deliver high performance for HTAP workloads.

1 Introduction

Hybrid transactional and analytical processing (HTAP) workloads present a critical challenge for database systems [6, 7, 31, 41, 52, 58, 71]. HTAP database management systems are expected to provide simultaneous services to on-line transactional processing (OLTP) workloads for latency-critical point queries in complex, multi-access environments [4, 9, 11, 44, 61] and on-line analytical processing (OLAP) workloads for throughput-driven decision support and data analysis [16, 19, 21]. Hence, HTAP systems must cohesively consider the OLTP performance, OLAP performance, and OLAP data freshness. Due to the increasing demands for real-time data analytics [8, 15, 17, 26, 29, 46, 49, 57, 67], OLAP data freshness has become a critical metric, especially for domains such as E-commerce and advertisement [3, 23, 27].

To support HTAP workloads, database management systems face a grand challenge stemming from the conflicting needs in storage formats for OLTP and OLAP workloads. OLTP workloads, characterized by a high volume of concurrent and small transactions, predominantly utilize row-based storage formats where each tuple (or row) is stored contiguously, with all tuples organized and managed through a tree index structure (e.g., B$^+$-tree [13, 20, 39]). Conversely, OLAP workloads, involving extensive column-specific queries across large datasets, favors columnar storage formats [6, 7] to enhance query efficiency and optimize data access throughput. To accommodate such row vs. column storage format conflict, a common practice in existing HTAP systems is to adopt a *multi-index* design, which employs distinct indexes (e.g., row-based B$^+$-tree index and column-based index) and dynamically migrate data across different index domains through processes, such as ETL (Extract, Transform, Load) [37] or log shipping [17, 26, 31, 41, 67, 71]. This conventional design approach faces the following two major issues:

• **Issue #1: Data freshness.** Data migration between distinct index domains can have a notable impact on the OLAP data freshness. To realize real-time analytics, OLAP clients must await the migration of newly ingested updates from the row-based index into the column-based index. This makes migration process a synchronous operation, often leading to significant degradation on the analytics *data freshness* that could range from tens of milliseconds to several minutes in existing systems [26, 31, 71]. Although one may improve the analytics data freshness by allowing OLAP clients to directly access the row-based index, conducting immediate, column-specific retrieval through the row-based index may result in

*Zhaoyan Shen is the Corresponding author

Table 1: HTAP Design Categories.

Design Approach	Implementation	Data Copies	Storage	Data Freshness
multi-index multi-store	ByteHTAP [17], PolarDB-IMCI [67], VEGITO [56], BatchDB [41], TiDB [31], F1 Lighting [71]	2	Disk-based	20ms to 8min
multi-index single-store	SAP HANA [57], Hyper [36], MemSQL [59]	1	Memory-based	Instant
single-index single-store	HaSiS (our approach)	1	Disk-based	Instant

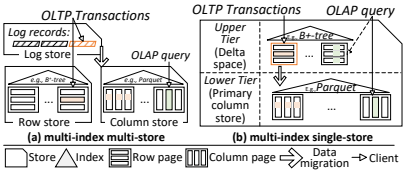


Figure 1: HTAP design alternatives.

substantial *read amplification* and hence prolonged latency. As a result, the benefits of directly accessing the row-based index may not necessarily outweigh the losses caused by large read amplification. Such a dilemma hinders existing HTAP designs from effectively performing truly real-time analytics.

• **Issue #2: Sub-optimal OLTP and OLAP performance.** Cross-index-domain data migration also causes notable performance degradation in HTAP systems. As an I/O-intensive operation, data migration extracts transactional updates from the row-based index and merges them into column-based data segments. It incurs significant data *write amplifications*, leading to severe I/O contention and interference with foreground OLTP/OLAP services. Additionally, migration load increases with the intensity of OLTP updates, causing an overall performance drop under high transactional workload pressure.

In contrast to conventional multi-index design practices, this work advocates for a paradigm shift towards *single-index* HTAP to fundamentally mitigate overheads caused by cross-index data migration. The **key** here is to leverage new storage hardware, other than relying on multiple distinct indexes, to reconcile the OLTP vs. OLAP storage format conflict. In particular, we utilize computational storage drives (CSDs) with built-in transparent compression, which have recently emerged on the commercial market [32, 53]. Such CSDs create an unprecedented opportunity for storage management software to purposefully employ *sparsely filled* storage data blocks without sacrificing physical storage capacity.

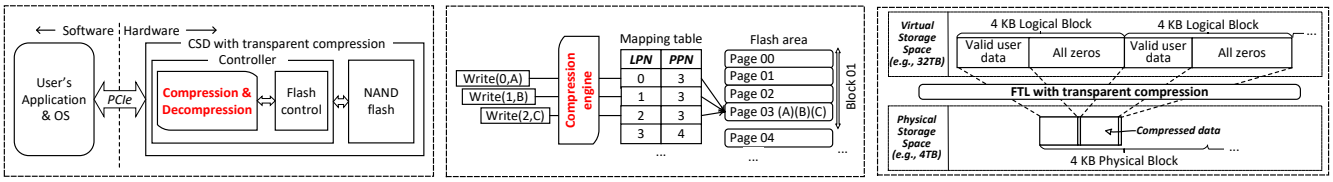
Based on this unique opportunity, we have developed a **H**ardware-**a**ssisted **S**ingle-**i**ndex **S**tore (**HaSiS**) design for HTAP with the following three features: (1) *Page size and write amplification decoupling*: Under conventional B$^+$-tree design practice, write amplification is directly proportional to page size [68, 69]. Hence most OLTP databases adopt small page size (e.g., 8KB in Oracle [45]/ PostgreSQL [61] and 16KB in MySQL [44]), which is fundamentally hostile to OLAP workloads. In contrast, our CSD-assisted B$^+$-tree design largely relaxes the dependency of write amplification on page size. This enables the use of very large page size (e.g., 128KB) without notably sacrificing OLTP performance, which creates a potential for B$^+$-tree to much more effectively serve OLAP workloads. (2) *Sparse intra-page column packing*: Inside each large-size B$^+$-tree page, following the PAX design principle [5], the tuples are stored in the column-based format. Again, leveraging the compression-capable CSDs, we *sparsely* pack all the columns to minimize the storage I/O read amplification under OLAP workloads, without sacrificing the physical storage capacity. This plays a key role in enabling our CSD-assisted B$^+$-tree to achieve high OLAP performance. (3) *Per-page clustered storage of multi-version records*: Once again, leveraging the unique property of compression-capable CSDs, HaSiS can conveniently store/index multi-version records on a per-page basis without using additional metadata or index. This makes it possible to support multi-version concurrency control without sacrificing the OLAP speed performance. All the above three features together ensure the use of a single B$^+$-tree to effectively serve HTAP workloads with instant OLAP data freshness.

We have implemented a fully functional and open-sourced prototype of HaSiS [30] and conducted experiments using a commercially available compression-capable CSD, namely ScaleFlux CSD-3310 [53, 54]. To demonstrate its effectiveness, we have compared its performance with (1) a state-of-the-art HTAP database, TiDB, (2) two baseline implementations of a row-based store and a column-based store, and (3) state-of-the-art row/column stores including MySQL, PostgreSQL, and Parquet. Our evaluation results demonstrate that HaSiS can achieve instant data freshness, comparable performance compared to TiDB, and preferable performance compared to baselines and other state-of-the-art solutions.

2 HTAP System Design: State of the Art

HTAP systems need to handle both OLTP and OLAP workloads. However, OLTP workloads typically favor row-based storage format, while OLAP workloads favor column-based storage format. Therefore, prior work primarily followed a multi-index design strategy (e.g., mixing row-based B$^+$-tree index and columnar index) and realized data synchronization/consistency via runtime cross-index-domain data migration. As shown in Table 1, existing multi-index HTAP design solutions largely fall into two categories: *multi-index, multi-store* and *multi-index, single-store*.

The multi-index, multi-store approach maintains row-based and column-based data engines separately: an OLTP engine with a row-based index for transactional updates and an OLAP engine with a column-based index for analytical queries. Due to the separate management of data, a transformation process is necessary to convert row-based data into a column-based format. In contrast, the multi-index, single-store approach consolidates row-based and column-based data within a single system. This approach typically lays a row-based delta tier on top of the main column-based store to enhance an OLAP system's support for OLTP. Here, row-based data serve as incremental updates (deltas), which are gradually merged into the main store indexed by a column-based format.



(a) Architecture overview of CSD with transparent compression (b) Internal operations that enable transparent compression (c) Large virtual storage space granted by transparent compression

Figure 2: Architecture overview and internal operations of CSD with transparent compression.

• **Existing multi-index, multi-store HTAP solutions.** As Cloud-native databases, ByteHTAP [17] and PolarDB-IMCI [67] realize data migration by shipping redo logs from OLTP engines to OLAP engines. As shown in Figure 1(a), upon receiving log records, OLAP engines merge them into their column-based data stores. VEGITO [56] and BatchDB [41] also rely on log shipping to realize data migration between their OLTP and OLAP engines. TiDB [31] and F1 Lighting [71] are distributed databases designed for HTAP workloads. They reuse the change log, originally intended for consensus protocols like Raft, to update column-based stores.

However, log-shipping incurs a *data freshness* problem —column store may not be updated in time as transactional updates are temporarily held in the log. As shown in Table 1, the reported delays range from tens of milliseconds to several minutes. Although some systems permit OLAP queries to retrieve data directly from the row store, this however forces analytical queries to navigate through multiple index structures and cause severe read amplification.

• **Existing multi-index, single-store HTAP solutions.** SAP HANA [58] is an in-memory database employing a layered design, which integrates tree-indexed deltas atop its primary column-based store. Row deltas are periodically merged into the columnar data in the background. Hyper [36] and MemSQL [59] maintain durable column chunks with in-memory tree-indexed row stores. Shown in Figure 1(b), these solutions can provide instant data freshness. However, since the dataset is distributed across column-based store and row-based delta store, analytical queries have to access data from both data sources, resulting in index and read amplification, affecting query efficiency. Although it can be alleviated by their in-memory nature, the performance can significantly degrade when they are forced to handle larger-than-memory datasets.

In this paper, our objective is to achieve *instant* data freshness through a *single* index, while minimizing read amplification and ensuring optimal performance for HTAP workloads.

3 Why CSD for Single-Index HTAP?

3.1 CSD with Transparent Compression

Any storage drives capable of executing computational tasks beyond storage functions can be categorized as CSDs [2, 32, 38, 42, 63, 64, 70, 72, 73]. This work focuses on a special type of *CSD that has a built-in transparent data compression functionality* [53, 54, 70, 72]. As shown in Figure 2(a), the controller chip inside CSD contains a dedicated hardware engine for (de)compressing individual 4KB LBA (logical block address) data blocks along the I/O path. Its Flash Translation Layer (FTL) organizes the storage of variable-length, post-compression data blocks on flash memory chips. In contrast to host-side compression [12, 25, 40], such CSDs relieve host CPUs from (de)compression and management of post-compression variable-length data blocks [70, 72]. In ordinary SSDs, every 4KB data block is mapped to a 4KB physical page, which is a one-to-one mapping between logical page numbers (LPNs) and physical page numbers (PPNs). In CSDs with transparent compression, one 4KB physical page may contain compressed data blocks of multiple logical pages, an N-to-one mapping, as shown in Figure 2(b).

Such CSDs have two unique features. First, it exposes a virtualized logical storage space that can largely surpass its internal physical storage capacity, similar to the concept of thin provisioning [14]. This enables users to allocate more logical storage capacity than what is physically available. Second, leveraging the high compressibility of low-entropy data patterns (e.g., all zeros), data management software could sparsely fill 4KB LBA blocks (e.g., 1KB real data and 3KB zeros) without sacrificing true physical storage space, as illustrated in Figure 2(c). These features facilitate the decoupling of user-perceived logical storage space usage from the physical storage space consumption, creating new opportunities.

3.2 Opportunities Brought by CSD

Aiming at single-index HTAP, this work focuses on making classical B$^+$-tree to much more effectively serve OLAP workloads without notably sacrificing its OLTP competence. How well a B$^+$-tree can serve OLAP depends on how much it can reduce its storage I/O read amplification under column-oriented data access. The first step is to store all the tuples in column-based format within each B$^+$-tree page, which is commonly referred to as PAX page format [5]. Let l_c denote the size of one intra-page column, and suppose we must fetch s consecutive 4KB LBA blocks from storage devices when reading this intra-page column. The corresponding storage I/O read amplification is $\gamma = \frac{s \cdot 4KB}{l_c}$. To reduce γ, we should (1) increase the value of l_c (i.e., the size of intra-page columns) by storing more tuples in each page, and/or (2) reduce the value of s by aligning the placement of intra-page columns with 4KB boundaries. Storing more tuples per page demands a larger page size. However, conventional wisdom suggests that a large B$^+$-tree page size leads to high write amplification and hence worse OLTP performance. Meanwhile, 4KB-aligned intra-page column placement induces page storage

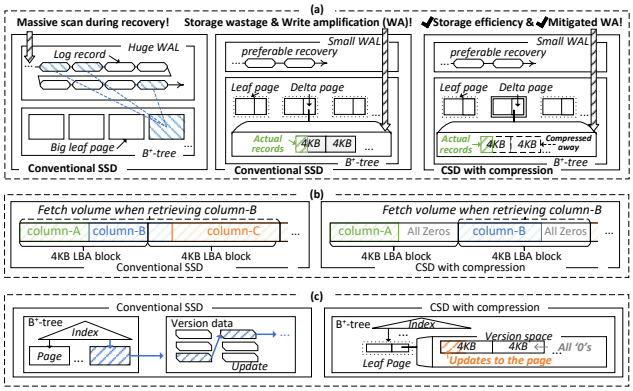


Figure 3: Illustration of the opportunities brought by CSDs.

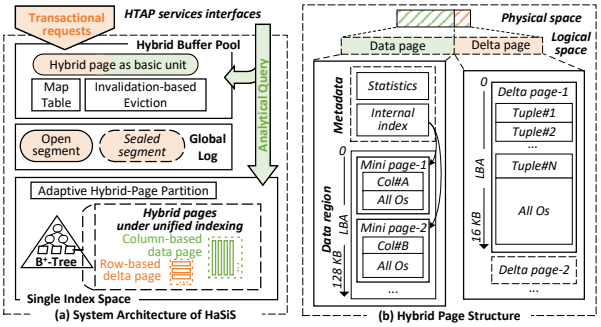


Figure 4: Architecture overview of HaSiS.

space waste. Moreover, given long-running OLAP queries, B$^+$-tree must be able to effectively store/index a large number of versioned tuples to maintain its efficacy for serving OLTP workloads. These issues prevent classical B$^+$-tree from effectively serving HTAP workloads. Fortunately, the emerging CSDs with built-in transparent compression bring three unique opportunities to address these issues.

Opportunity #1: Decoupling write amplification from page size. To increase B$^+$-tree page size without proportionally increasing write amplification, we could keep more log records in *write-ahead log* (WAL) before merging them into destination pages. In current practice, all B$^+$-tree pages share a global WAL. Since the global WAL must be scanned during crash recovery, a larger WAL directly causes a longer crash recovery time. Nevertheless, practical systems tend to have stringent requirements for crash recovery time. Intuitively, if each B$^+$-tree leaf page can have a dedicated storage space, called *delta page*, for accumulating its own associated log records, we could aggressively apply logging to reduce write amplification, without compromising crash recovery time.

However, since each delta page must occupy one or multiple 4KB storage blocks to hold log records for only one B$^+$-tree page, its content would be *sparse* most of the time, e.g., one 4KB delta page may contain 0.5KB log records and 3.5KB zeros. Hence, when B$^+$-tree operates on ordinary SSDs, sparse delta pages would result in significant physical storage space waste. As shown in Figure 3(a), replacing ordinary SSDs with compression-capable CSDs seamlessly solves the problem: With built-in compression capability, CSDs naturally compress away the all-zero segment in each sparsely filled delta page, avoiding wastage of the physical storage space. Therefore, compression-capable CSDs readily enable B$^+$-tree to employ very large pages without notably sacrificing its OLTP competence.

Opportunity #2: Reducing intra-page column read amplification. As discussed previously, the placement of each intra-page column should be aligned with 4KB LBA storage block boundaries to reduce the value of s and hence reduce the I/O read amplification. This can be further illustrated in Figure 3(b): If we compactly pack intra-page columns as in conventional practice, to read the column-B, we have to fetch two 4KB LBA blocks from the storage device even though the column-B itself is less than 4KB. In contrast, as shown in Figure 3(b), if the placement of columns is 4KB-aligned, column-B resides in only one 4KB LBA block, hence reading column-B only incurs the fetch of one 4KB LBA block from the storage device. Clearly, such 4KB-aligned column placement introduces *content sparsity* inside B$^+$-tree pages. Leveraging its built-in compression capability, CSD can natively accommodate such content sparsity without sacrificing the physical storage space.

Opportunity #3: Per-page clustered storage of multi-version records. Existing database management systems typically allocate global storage space to realize concurrent multi-version control [10, 18, 35, 50, 60, 62]. The co-existence of OLTP and OLAP workloads makes it a bigger challenge to efficiently manage the storage of multi-version records. In particular, long-running OLAP workloads could cause a long chain of versioned records spreading over the global storage space, leading to significant I/O read amplification and hence system speed performance degradation [28, 35, 50]. To address this, one option is to cluster all versioned records on a per-page basis, which can largely simplify the access/management of multi-versioned records. However, due to the runtime workload variation, the amount of per-page multi-version records can substantially vary from one page to the other and over time. This can notably complicate the storage management when using ordinary SSDs. Leveraging the native thin-provisioning support of compression-capable CSDs, we could conveniently realize per-page clustered storage of multi-version records. In Figure 3(c), for each B$^+$-tree page, we can pre-allocate an *over-provisioned virtual space* to store all multi-versioned records, adapting to the varying storage demand without incurring additional physical storage cost.

4 System Design

In this paper, we present a **H**ardware-**a**ssisted **S**ingle-**i**ndex **S**tore design solution, named **HaSiS**, by leveraging the above-said unique, CSD-enabled opportunities to serve HTAP workloads with a single index on a unified, single data store. Unlike existing multi-index solutions, HaSiS simultaneously achieves nearly identical performance as dedicated row- and column-based data stores for TP and AP workloads, while ensuring *instant* OLAP data freshness.

Figure 4 illustrates the architecture overview of HaSiS. Though based on the classical B$^+$-tree, HaSiS is OLAP-friendly while still retaining its high OLTP performance. This is achieved via several important measures in our designs: (1) A large page size (e.g., 128KB) is employed for handling OLAP workloads, and the induced negative impact on write amplification (hence OLTP performance) is mitigated by persistently buffering log records on the per-page basis in a dispersed manner; (2) A hybrid page structure, as illustrated in Figure 4(b), is designed to integrate the row-based data format for OLTP transactional updates and the column-based data format with 4KB-aligned placement in the B$^+$-tree pages to best serve OLAP workloads; (3) The buffer pool management is optimized to tackle the memory under-utilization and buffer pollution problems caused by the use of large page sizes for OLTP workloads; (4) Multi-version records are clustered on the per-page basis to more effectively accommodate long-running OLAP workloads. All the core techniques of HaSiS essentially trade *logical storage space* for *efficacy*. The unique abilities of the emerging CSD hardware enable us to purposefully make such tradeoffs without inducing wastage of physical storage space. In the rest of this section, we will describe each component in detail.

Transactional requests are processed by the hybrid buffer pool, with corresponding changes persisted in the global log. These requests are managed by retrieving or rewriting hybrid pages indexed by a B$^+$-tree. Analytical queries can either check the buffer pool or directly access hybrid pages to obtain the most recent data resulting from transactional updates.

4.1 Single-Index, Single-Store

Our main goal is to attain instant data freshness and serve both OLAP and OLTP workloads in a highly efficient manner. The key to achieving this goal is to create a single-index, single-store design to remove the need for maintaining multiple indexes and migrating data across separated index domains. However, the prior efforts we witnessed in past decades have proven that realizing such a design is difficult. This is mainly due to two critical challenges.

First, OLTP and OLAP workloads feature drastically distinct access patterns, demanding different data organizations on storage. Analytical queries generally involve retrieving a substantial volume of data, thus favoring column-based data organization. A common practice is to adopt a large granularity (128KB to MBs) as the basic data unit (B$^+$-tree's leaf node), which helps reduce the management overhead and facilitates batched retrieval. In contrast, transactional queries favor row-based data organization, typically using a small granularity (8KB or 16KB) to reduce write/read amplification caused by in-place updates or retrievals. A single-store design serving both workloads has to simultaneously accommodate the two radically different access patterns, which have almost opposite demands. Satisfying requirements for one often unfortunately means missing requirements for the other.

Second, maintaining a single index structure can lead to notable contention issues, particularly for OLAP queries. OLTP operations frequently involve structural modifications to the index, which are necessary to ensure data consistency and typically need to be performed exclusively. However, it presents a significant challenge to read-only OLAP queries, as they may be routinely blocked or delayed while waiting for the completion of OLTP's structural changes. This situation can significantly impede the efficiency of OLAP queries, as they are forced to wait, potentially affecting the overall performance of the system. Addressing this contention is crucial to the effectiveness of our indexing strategy.

To tackle these challenges, our key idea is to create a unified data structure, which organizes data in a carefully designed hybrid (column-based, OLAP-friendly and row-based, OLTP-friendly) page as a *single-index-addressed* unit. Leveraging the CSD hardware, which provides a virtualized storage space, we can purposefully create a *sparse* data structure on storage to remove undesirable read/write amplification and index contention. In the following, we will describe the related page structure, indexing, and buffer pool management designs.

Hybrid Page Structure. The core data structure in HaSiS is *hybrid page*, which is an index-addressable storage entity, functioning as a leaf node of a B$^+$-tree index. As illustrated in Figure 4(b), the data in a hybrid page is organized in two parts, a column-based *data page* and a row-based *delta page*. The data page is optimized for handling analytical queries; The delta page collects transactional data as a small, incremental log to the column-based data page.

• *Column-based Data Page* organizes large volumes of data by columns, which inherits the PAX-page layout [5–7] and is tailored for analytical queries. Similar to the PAX page, a column-based data page comprises a *data region* for column-based data storage and a *metadata region* for internal column data indexing, whereas the data region is divided into 4KB *mini-pages*, each of which sparsely manages values in columns. As shown in Figure 4(b), each mini-page only stores column values belonging to one specific column (e.g., mini-page-1 only holds column values belonging to column-A). The mini-pages inside the data region are sequentially allocated in the LBA order on storage, and the column values in a mini-page are stored consecutively one after another. If a column value cannot be entirely accommodated at the end of a mini-page, we move to the next mini-page and continue to store the value there; The unused mini-page space is filled up with all zeros, which would be transparently compressed away by the CSD. With hardware assistance, this sparse storage structure can effectively reduce read amplifications without wasting physical storage space.

In the metadata region, an internal index keeps the mapping between `<start key, column name>` of a mini-page to its `LBA address` in storage. This enables efficient point queries or small-range scans by directly accessing mini-pages rather than loading the entire data region. In addition, the meta-

data region also records essential information for B$^+$-tree's indexing (e.g., key range and capacity usage for page split or merge). To eliminate I/Os of metadata retrieval, we cache the small-sized metadata region of each hybrid page (around 2,670B), along with B$^+$-tree's index nodes in memory.

• *Row-based Delta Page* essentially mirrors the row-based page in relational database systems like MySQL [44] and PostgreSQL [61], specifically for efficient transactional data ingestion. As shown in Figure 4(b), the delta page manages data records (i.e., tuples) by rows, in which all tuples are stored one after another, sequentially based on the LBA order and acting as a compact, in-situ update log to patch the column-based data page. Similar to mini-pages, if the delta page is not fully occupied by tuples, the reserved space is filled by all zeros, which can be automatically compressed away by the CSD hardware. When the delta page is completely filled up, a *major compaction* operation is triggered to apply the changes to the column-based data page (see Section 4.3).

In our implementation, a hybrid page includes a 128KB column-based data page and a 16KB row-based delta page. As the delta page immediately follows the column data page on the logical storage space, it enables a hybrid page to be accessible with only one single I/O operation, streamlining the process and enhancing the overall system performance. The current 128:16 ratio works well in our experiments. In situations where all updates in the delta page have to be temporarily preserved for query, such as due to multi-version control as described in Section 4.4, the delta page space can be dynamically extended by pre-allocating or dynamically linking to an additional 128KB LBA space following the existing delta page. For each hybrid page, we can pre-allocate more than one delta page to ingest transactional updates. With transparent compression of the CSD hardware, the over-provisioned delta-pages would not occupy physical storage space.

Unified Indexing. The hybrid page design facilitates *unified indexing* of both transactional updates and column-based data. As illustrated in Figure 4(a), rather than separately indexing row-based updates and column-based data, a hybrid page unifies the data page and delta page into a single, index-addressable storage entity, functioning as the leaf nodes of a B$^+$-tree index. It enables analytical queries to instantly retrieve recently ingested transactional updates with the main column-based data through a single index search, thus fundamentally removing the need for migrating and transforming data across different index domains.

We further partition hybrid pages based on table relations to alleviate contention between transactional operations and analytical queries. Table relations have been widely used for indexing and organizing data for OLTP and OLAP workloads [9,31,44,61]. As illustrated in Figure 4(a), our approach involves partitioning hybrid pages by tables, with each table partition housing a dedicated B$^+$-tree index structure. Moreover, when a table grows beyond a certain size threshold, it can be progressively subdivided into smaller partitions. Given that the index volume scales proportionally to the number of indexed records, we set the partition size limit to 50,000 records, which works well in our experiments. This setting ensures that the depths of the B$^+$-tree index stay below three levels, effectively balancing and distributing operations across different partitions to minimize operational contention.

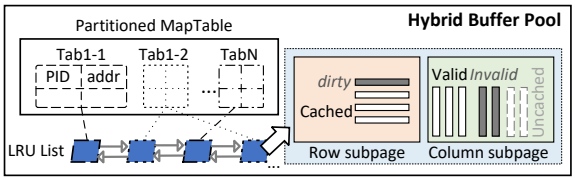


Figure 5: An overview of the hybrid buffer pool design.

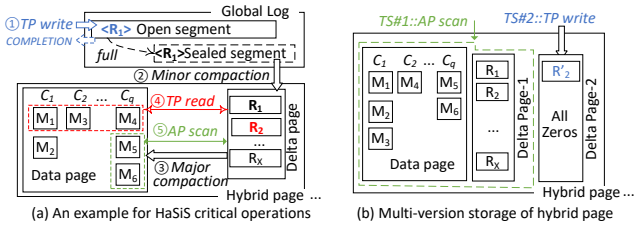
4.2 Hybrid Buffer Pool

The buffer pool is crucial in HTAP system performance by caching frequently accessed data in memory for fast access. As a hybrid page embeds two types of data format, in rows or columns, for serving hybrid workloads, introducing the hybrid page into HTAP systems brings two new issues.

First, simply fetching the entire hybrid page into the buffer pool would occupy an excessive amount of memory space without yielding corresponding benefits. This is particularly evident for analytical queries, which typically exhibit minimal locality. These queries often access large volumes of column data, which are unlikely to be reused. Holding such data in the buffer pool would lead to heavy "buffer pollution". Second, simply applying the existing flush-based eviction policies can cause significant write amplification. When a hybrid page is modified in the buffer pool and marked as "dirty", an LRU-based eviction policy would flush the entire hybrid page back to storage, even for minor updates. This is highly inefficient, especially considering the large size of a hybrid page.

To overcome these issues, we propose a *hybrid buffer pool* design, illustrated in Figure 5. This design incorporates *fine-grained, invalidation-based buffer pool management*, allowing for partial reservation of hybrid pages, to mitigate the memory space wastage and meanwhile address the write amplification issue induced by page flushes. This design integrates fundamental data structures commonly found in buffer pools, including (1) an LRU List that manages hybrid pages based on their access frequency, and (2) a mapping table that maintains the indirect association between a hybrid page's logical identifier (e.g., node ID allocated based on B$^+$-tree) and its block address. Specifically, it works as follows.

An *in-memory hybrid page* comprises two sub-pages, namely *column sub-page* and *row sub-page*, corresponding to the on-storage column-based page and row-based delta page, respectively. When fetching an on-storage hybrid page into memory, rather than loading the entire hybrid page, only the delta page along with involved 4KB mini-pages that intersect with the queried rows are retrieved into the hybrid buffer pool. Upon updating an in-memory hybrid page, we modify the row sub-page and invalidate relevant column values in the column



(a) An example for HaSiS critical operations (b) Multi-version storage of hybrid page

Figure 6: Examples of how HaSiS handles HTAP workloads.

sub-page in memory. Considering background compaction consistently applies updates to the on-storage data pages, when mini-pages, whether updated by major compaction or not, are fetched into the buffer pool, the invalidation approach guarantees that there is only one version for each mini-page in the hybrid buffer pool. This ensures consistency between the in-memory and on-storage versions of a hybrid page. Finally, when evicting an in-memory hybrid page, we simply discard it, as the most recent version is already maintained on storage by compaction operations. This eliminates the typical write amplification issues associated with flushing processes, making the buffer pool more efficient for HTAP workloads.

4.3 Log and Compaction

A critical challenge in HTAP systems is how to efficiently handle OLTP transactions, which involve massive small, latency-sensitive transactional updates and insertions. Such operations often incur high-cost "read-modify-write" operations, causing significant I/O amplifications and impairing system performance. HaSiS tackles this challenge by maintaining a small, indexed *global log* and a *two-phase compaction* process.

Global log. The hybrid page design in HaSiS essentially manages updates in a dispersed manner. For transactional updates, changes are made to the corresponding delta pages within hybrid pages. To further mitigate the write amplification of directly rewriting delta pages and decouple foreground operations from storage I/Os, as shown in Figure 6, a small, centralized *global log* temporarily accommodates data changes in an append-only manner. A small in-memory index keeps track of these temporary updates in the log. Once the changes are persisted in the global log, the system immediately returns `COMPLETION` to the client, while the changes are applied (compacted) to the corresponding hybrid pages asynchronously. The global log maintains an open *segment* to log the updates. When the segment size exceeds a predefined threshold (e.g., 100MB), it is sealed and a new open segment is allocated. Changes in the sealed segment are flushed (applied) to their corresponding hybrid pages via a two-phase compaction process as described below.

Two-phase compaction. When a sealed segment is generated, the *minor compaction* process is triggered to (1) retrieve all relevant row-based delta pages from storage, (2) append log records to update these pages in memory in a row-based manner, and (3) flush the updated delta pages back to storage. This batched process capitalizes on the parallel processing capabilities of the CSD with boosted efficiency: retrieving and flushing multiple delta pages can be conducted in a batched and parallelized way to fully utilize the storage bandwidth.

If a delta page exceeds its space limit, *major compaction* runs to merge the "delta" updates into their corresponding column-based data pages, reclaiming the delta page space for future updates. This process involves reading the corresponding mini-pages, applying changes in memory, and flushing the updated column data pages back to storage.

Different from conventional HTAP solutions that involve extensive data transfer between row and column storage, this compaction-based approach significantly mitigates migration costs with better data freshness and I/O efficiency: (1) Given the compaction is index-internal and operates in the background, AP queries can immediately access newly ingested TP data; (2) As hybrid pages are isolated data entities and independent from each other, the major compaction can merge multiple hybrid pages in a batched and parallelized manner and thus to guarantee I/O performance.

4.4 Operations

In this section, we summarize and illustrate the workflows of HaSiS for handling OLTP and OLAP workloads via an example. As shown in Figure 6, a hybrid page contains a delta page that stores X data records by rows (from R_1 to R_X), and a column-based data page that organizes data records in q columns with one or multiple mini-pages each.

• *OLTP writes* include transactional updates and insertions. HaSiS adopts a global log and a two-phase compaction mechanism to quickly log changes in the small global log, then asynchronously append them to the delta pages as a per-page log, and finally merge them into their final destination, the column-based data pages. As illustrated in Figure 6(a), an incoming TP write R_1 (Step ①) is recorded as a change log $<R_1>$ in an open segment of the global log in an append-only manner. After the change log is persisted and the log's in-memory index for it is updated, we can directly return success to users. As the segment size exceeds the predefined threshold, it is sealed and scheduled for flushing via minor compaction (Step ②). If the delta page is full, major compaction (Step ③) merges the logged updates, including R_1, from the delta page into the corresponding column page.

• *OLTP reads* include record lookup or small-range scans (e.g., selecting a few dozens of records). As shown in Figure 6(a), ongoing compaction processes periodically integrate transactional updates into the delta page and subsequently the column page. As a result, a record in a hybrid page could be in the delta page, the column data page, or both. To ensure correct OLTP reads, each read operation has to verify both pages. However, simply retrieving the entire hybrid page for OLTP reads can lead to notable read amplification. Our mini-page design can address this issue with fine-grained access. As shown in Figure 6(a), rather than retrieving an entire column page, a TP read (Step ④) only involves a small portion of mini-pages (e.g., M_1, M_3, M_4), i.e., to serve this request,

we only retrieve the delta page and the involved mini-pages, rather than retrieving the entire hybrid page. This approach significantly reduces read amplification.

• *OLAP scans.* For analytical queries that require scanning extensive volumes of hybrid pages, HaSiS is designed to execute hybrid page retrieval with the least read amplification. In Figure 6(a), an OLAP scan (Step ⑤) intends to retrieve a portion of column values from column-q (C_q). We only load the involved mini pages (M_5 and M_6) from the column data page, along with the delta page physically stored alongside it. As the mini-pages sparsely align attributes of the same column in 4KB logical blocks without cross-LBA placement, the read amplification for column-oriented data retrieval can be significantly reduced. Moreover, leveraging the hybrid page's metadata, the query process can be accelerated by conducting multiple 4KB mini-page reads in a batched-up way.

• *Multi-version support.* A benefit provided by the hybrid page design is that delta pages could be leveraged for simplified access/management of multi-versioned records. Figure 6(b) showcases the multi-version storage supported by the hybrid page. An AP scan operation starts at timestamp 1 (TS #1) and intends to retrieve data records from the hybrid page. After TS #1 and before the completion of the AP scan, another TP write at timestamp 2 (TS #2) intends to rewrite the record R_2 in the hybrid page. Naïvely, the record R_2 issued by the TP write (R'_2) should be persisted into the hybrid page, as it is the user-issued update that is supposed to be merged into the hybrid page. However, the new value, R'_2, should remain inaccessible to the AP scan, since it is ingested after the scan operation. Meanwhile, the old version of R_2 should not be overwritten by R'_2, because the old R_2 should remain accessible to the AP scan before its completion.

In this situation, we need to preserve both versions. Leveraging the pre-allocated LBA space, we can expand the hybrid page by allocating or linking a new delta page (Delta Page-2), such that the new value, R'_2, can be temporarily preserved to fulfill the TP write, and be merged into Delta Page-1 later after the completion of the AP scan. In real-world scenarios, AP scans seeking to gather insights from a large number of pages typically exhibit no locality. This means that it is rare in practice that updating a hybrid page is extensively blocked by AP scans. In Section 5.3, we will demonstrate in experiments that even in the worst-case scenario, the additional delta page allocation remains within a reasonable range.

4.5 Summary

Essentially, the core idea of HaSiS is to combine the B$^+$-tree index with hybrid delta and PAX pages. However, simply merging these existing techniques cannot fundamentally resolve the challenges presented by HTAP workloads. Compared to prior efforts, HaSiS achieves its design goal with several unique and important measures.

(1) The small global log and the two-phase compaction process decouple the processing of transactional and analytical queries. It allows a TP write to immediately signal success to users once the write is persisted into the open segment, which enables "blind updates" in transactional operations without requiring loading the original page into memory. Combined with the asynchronized two-phase compaction, the write path for transactional updates is significantly shortened, speeding up OLTP requests and reducing interference to OLAP queries.

(2) Emerging CSDs with transparent compression provide a virtualized storage space, allowing us to purposefully create a sparse data structure without wasting physical storage space. Aggressively over-provisioning storage space, such as aligning mini-pages in 4KB logical blocks and padding delta pages with all zeros, enables us to reduce read/write amplifications with a more efficient data layout on storage.

(3) The single-index, single-store design integrates both column- and row-based data layouts, which are originally tailored to two drastically different workloads, to maintain only one single copy and one single index to address the data. This fundamentally eliminates the need for a resource-demanding row-to-column transformation process, making all persisted data instantly available for analytical queries.

These design elements are the result of deliberate choices based on careful consideration of the properties of hardware and software, forming a comprehensive solution that effectively addresses the long-existing HTAP challenges.

5 Evaluation

• **Implementation.** We have implemented a fully functional prototype of HaSiS incorporating the single-store single-index design, the hybrid buffer pool, and the multi-version storage support. By default, the sizes of the column-based data page and delta page are set to 128KB and 16KB. For comparison, we have developed two baseline systems to represent the optimal performance of the pure column store (CS) and row store (RS): CS utilizes 128KB PAX pages indexed by a B$^+$-Tree without employing delta pages where incoming updates directly overwrite each 128KB PAX page; RS is implemented based on the pB$^+$-Tree architecture [33] leveraging per-page logging for blind updates where row-based data records are organized in 16KB data pages with 16KB per-page logs.

We also conduct comparisons with contemporary HTAP (TiDB), OLTP (MySQL and PostgreSQL), and OLAP (Parquet) designs to provide insights into the performance and effectiveness of HaSiS in the broader context of existing systems. All prototypes follow their default settings. We equally restrict the memory space for each prototype to 10% of the data set. Considering the weak locality of OLAP queries, we allow the OLAP queries in HaSiS to bypass the buffer pool.

• **Experimental Setup.** Our experiments are conducted on a server equipped with a 22-core 2.2GHz Intel Xeon E5-2696 v4 CPU, 64GB DDR4 DRAM, and a 7.68TB ScaleFlux CSD-3310 SSD. The server operates on Ubuntu 20.04 LTS with Linux Kernel 5.15 and Ext4 file system. The CSD-3310 SSD [53, 54] features a hardware-based zlib compres-

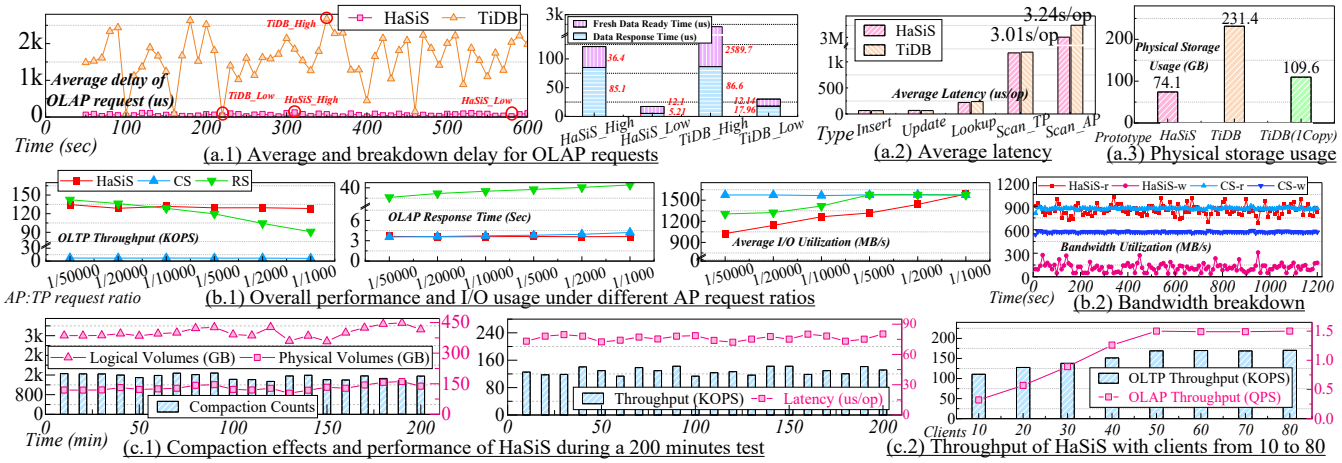


Figure 7: Overall comparison of HaSiS with TiDB and baseline stores.

sion engine that compresses each 4KB block along the internal I/O path, being seamlessly transparent to the host system. This engine operates with low latencies at around 5μs for (de)compression, which are significantly faster than typical read latencies (>50μs) and multiple orders of magnitude shorter than write latencies (>1ms) associated with TLC/QLC NAND flash memory. In terms of throughput, the drive, adhering to NVMe standards and supporting a PCIe Gen4 x4 interface, delivers high sequential read/write bandwidths (up to 7.2GB/s and 4.8GB/s), and random 4KB read/write IOPS (1,450K/380K) across a 100% LBA span. Thus, compression operations do not affect the system throughput even under high-intensity workloads. CSD-3310 is in volume production and widely deployed in data centers globally, demonstrating its reliability and robust performance in real-world settings. HaSiS does not rely on specific hardware implementations of ScaleFlux CSD but rather leverages the transparent compression commonly found in commercial CSDs [22, 34]. This compatibility makes HaSiS a flexible and scalable solution for modern storage environments.

• **Workload.** We have developed a benchmark tool to evaluate HaSiS in comparison to the baselines and SOTA solutions. This tool is capable of converting complex SQL queries, such as analytical queries, to simple and table-specific SQL requests (e.g., select column-A from table-B where column-A < X). It helps isolate the I/O execution from interference of other database-dependent components (e.g., planning and optimization) and provides detailed performance metrics, such as throughput and latency. This tool follows the table layout of CH-Benchmark [51] where "lineitem/part" table is aligned to TPC-C setting as "orderline/item" table, respectively.

We issue the TPC-C [65] and TPC-H [66] workloads to MySQL [44] and then collect the corresponding requests from InnoDB [43], MySQL's storage engine. The translated requests are essentially key-value operations, where the key is defined by the primary/secondary key following MySQL's standard. In this tool, OLTP requests include various transactional operations: (1) Insert for adding records to a table, (2) Update for rewriting records in a table, (3) Lookup for retrieving specific records, and (4) Scan (a.k.a., "Scan_TP"), for collecting column values from multiple records; OLAP requests (a.k.a., "Scan_AP") are groups of translated SQL operations extracted from TPC-H query plans. The workload distribution of translated requests strictly follows TPC-C/TPC-H workloads. Detailed implementation and benchmark explanation can be found in our open-source project.

• **Method.** Before the evaluation, we initiate a warm-up phase by sequentially populating each store with a TPC-C dataset containing 500 warehouses, during which the benchmark tool operates as OLTP clients issuing translated TPC-C requests. We then evaluate the performance of each prototype with the benchmark functioning as OLTP and OLAP clients that execute SQL requests derived from TPC-C and TPC-H requests. Throughout these experiments, we measure the throughput, latency across all request types, and the delay in serving OLAP requests for a comprehensive comparative analysis.

5.1 Overall Performance

• **Comparison with TiDB.** We first demonstrate the data freshness, overall performance, and storage efficiency of HaSiS and TiDB by issuing both OLTP (16 clients) and OLAP (8 clients) requests simultaneously (i.e., HTAP workload). We deploy TiDB in the local test cluster mode by Tiup [47] (one instance of TiDB, TiKV, PD, and TiFlash). We collect TiDB's metrics through [48]. For a fair comparison, we exclude the Raft log propagation for latency comparison.

As shown in Figure 7(a.1), we quantify the data freshness of HaSiS and TiDB by the *delay* of OLAP queries. Besides data retrieval, in HaSiS, the delay includes the time taken to register a record into global log's in-memory index; in TiDB, it includes its inherent row-to-column transformation [48]. The average delay is tracked per 10 seconds. Notably, HaSiS's delay stably spans from 17.31μs to 121.5μs, while that of TiDB spans from 30.1μs to 2,676.3μs with large fluctuation. A breakdown analysis for the lowest and highest delays of HaSiS and TiDB is shown to explain this fluctuation: due to

its row-to-column transformation, TiDB experiences 12.14μs and 2,589.7μs delay in waiting fresh data before they are ready for AP query while HaSiS with its single index design mitigates this wait to 12.1μs and 36.4μs. It highlights HaSiS's efficiency: by integrating row-based/column-based data into hybrid pages and indexing them through a single B$^+$-Tree, HaSiS can provide (almost) instant data freshness. This makes HaSiS particularly well-suited for applications in domains like e-commerce, finance, and real-time monitoring.

HaSiS exhibits comparable latency compared to TiDB in Figure 7(a.2). For the average latency of Insert/Update operation, benefiting from the blind update design, HaSiS only shows 3.77%/3.66% latency degradation compared to TiDB which adopts LSM-tree as the storage engine. For Lookup/Scan_TP, HaSiS exhibits 6.90%/1.21% lower latency compared to TiDB since HaSiS is a B$^+$-Tree design that only retrieves one page for each data record while the LSM-tree adopted by TiDB encounters read-amplification. For OLAP (Scan_AP) performance, although TiDB supports adaptive configuration (waiting until all data are converted as column-based ones or directly scanning both row and column data), TiDB shows a 6.64% longer average latency than HaSiS. HaSiS, with its single index and compaction design, can instantly retrieve all involved column data. The compaction occurs lazily, without impacting synchronous OLAP operations.

As a single-store design, HaSiS significantly reduces storage usage compared to the multi-store TiDB in Figure 7(a.3): HaSiS consumes 74.1GB of physical storage space in total, which is 67.98% less than TiDB. Compared with a hypothetical case where TiDB uses only a single copy of row data and the column store, HaSiS consumes 32.39% less storage space due to its more efficient data layout design.

• **Comparison with Baselines.** We then demonstrate the performance and I/O efficiency of HaSiS and baseline designs (CS and RS) in different ratios of OLTP and OLAP requests.

Performance isolation. Figure 7(b.1) evaluates the OLTP /OLAP performance isolation of HaSiS, RS and CS. By varying the AP:TP ratio (i.e., the ratio of the number of OLAP requests to OLTP requests), we evaluate the impact of increasing OLAP requests on the OLTP performance. As shown, HaSiS shows consistent stability in OLTP throughput and OLAP response time. In contrast, RS and CS exhibit notable performance fluctuations as the volume of AP requests rises. While initially RS slightly outperforms HaSiS in OLTP throughput at lower request ratios (below 1/10000), its performance declines with an increase in OLAP requests. This trend is also seen in OLAP performance. The observed performance interference in other systems stems from severe I/O contention and I/O amplification. HaSiS demonstrates an increasing trend in bandwidth utilization with the growing AP volumes, indicating its scalability potential. Conversely, when CS handles OLTP and RS handles OLAP, severe I/O amplification can saturate available bandwidth, consequently leading to I/O contentions limiting the scalability of both RS and CS. As shown in Figure 7(b.2), HaSiS efficiently manages OLTP writes (HaSiS-w) with lower bandwidth compared to CS which requires more bandwidth for writes (CS-w). With increasing AP requests, CS struggles to manage AP queries (CS-r) due to bandwidth constraints.

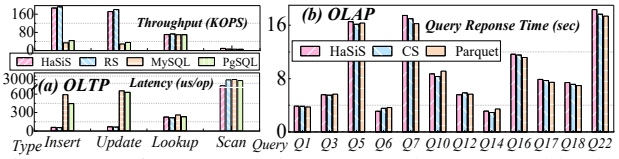


Figure 8: Performance under OLTP and OLAP workloads.

Performance of compaction. Figure 7(c.1) demonstrates the performance of HaSiS considering the impact of background compaction. We quantify the minor and major compaction occurrences and the associated logical and physical I/Os over a 200-minute test duration. Meanwhile, we measure the average throughput and latency of foreground operations collected at 10-minute intervals. As shown, background compactions exhibit a consistent triggering pattern and the performance stays rather stable throughout the long test. These findings aptly illustrate HaSiS' capability to effectively decouple performance from background I/O-intensive operations. Although logical I/Os induced by compactions are significant, the actual physical I/Os are only 23.1%-40.6% of logical ones.

Scalability. Figure 7(c.2) illustrates HaSiS's scalability. We measure OLTP/OLAP throughputs by increasing concurrent clients from 10 to 80. As shown, HaSiS's OLTP/OLAP throughputs linearly scale up from 10 clients (110.7KOPS/0.32QPS) to 50 clients (168.9KOPS/1.5QPS). The performance increment diminishes when exceeding 50 clients since the storage bandwidth and computation resources of the experiment platform become the system bottleneck.

5.2 Performance of OLTP and OLAP

• **Transactional Operations (OLTP).** We compare the OLTP performance of HaSiS with two representative OLTP systems, MySQL and PostgreSQL, in terms of average throughput and latency shown in Figure 8(a). We also involve RS, which adopts the per-page logging for blind updates, as the ideal performance baseline. The results show that HaSiS achieves competitive performance even compared to RS, and superior performance relative to MySQL and PostgreSQL.

HaSiS demonstrates slightly lower throughputs of Insert/Update/Lookup compared to RS that employs a 16KB row page with less write amplification during major compaction, as opposed to HaSiS which needs to rewrite a larger 128KB column page. However, leveraging modern NVMe SSDs capable of efficiently managing batched I/Os, the overhead can be minimized to within 5.11% (throughput) and 5.82% (latency). For Scan operations, HaSiS outperforms RS in both throughput (49.36%) and latency (26.92%). This advantage stems from HaSiS's capability of executing sub-page queries on large column pages, whereas RS requires multiple random retrievals of smaller (16KB) data pages.

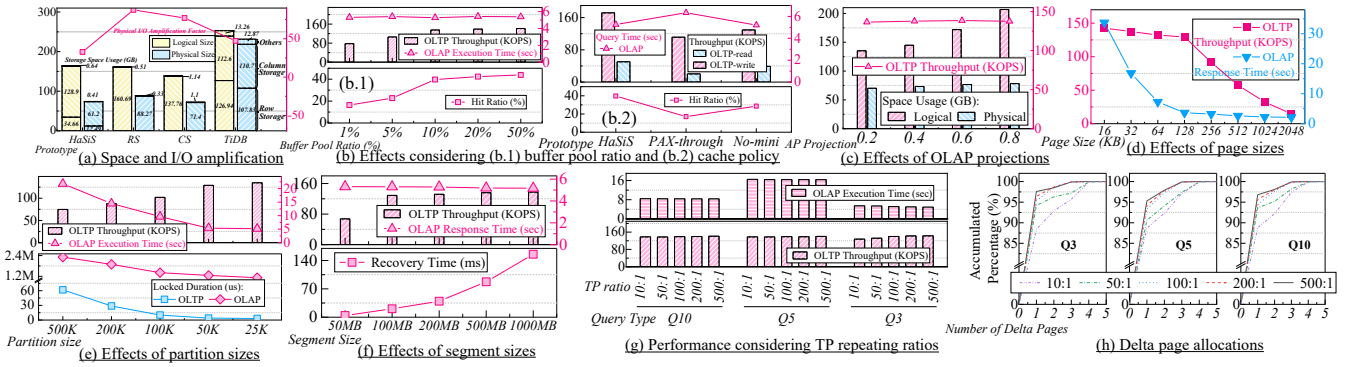


Figure 9: Effects of HaSiS optimizations.

• **Analytical Operations (OLAP).** Figure 8(b) illustrates the efficiency of HaSiS in handling analytical queries compared with CS and Parquet. Parquet is a widely used column format known for its performance in analytical queries on large-volume datasets. We choose 12 representative query plans from the TPC-H and translate them into SQL queries. These query plans do not include the ultra-long queries designed for full-table scans that simulate breakdown report tasks. HaSiS demonstrates marginal performance overhead compared to both CS (less than 7.71%) and Parquet (less than 7.12%) across all query types. The CS approach facilitates selective access to relevant column values within each 128KB page for less read amplification. Parquet employs a condensed data structure optimized for large datasets. In line with them, HaSiS adopts PAX-based column pages to store the majority of the data. When accessing a column page, only a sparse delta page needs to be additionally retrieved. These sparse delta pages, being highly compressible and contiguously stored, add minimal overhead to read performance.

5.3 Effects of Optimizations

• **Space usage and I/O amplification.** Figure 9(a) compares the logical/physical space usage and physical I/O amplification of HaSiS with RS, CS and TiDB. HaSiS requires 164.2GB/74.1GB of logical/physical storage space. The larger logical space (15.4% more than CS) is due to the allocation of sparse pages dedicated to processing transactional updates. However, using CSD with transparent compression can efficiently compress these sparse delta pages. As a result, the physical space requirement of HaSiS is only marginally higher (by 2.16%) than that of CS. For a detailed comparison, due to the sparse management of column data, HaSiS occupies more logical space (14.5%) than TiDB in column storage. After compression, the column storage space of HaSiS is 55.3% of that of TiDB. Moreover, HaSiS demonstrates the lowest level of physical I/O amplification, which stands at 37.4%, 42.5% and 69.5% of that observed in RS, CS and TiDB.

• **Effects of the hybrid buffer pool.** Figure 9(b.1) illustrates the linear increments of OLTP/OLAP performance and hit ratio of HaSiS across different buffer pool ratios (1% to 50% relative to the dataset size). This growth becomes negligible when the ratio exceeds 10%, given the insignificant locality of TPC-C. The OLAP performance remains consistent across all scenarios as we allow OLAP queries, which also show poor locality, to bypass the buffer pool, making them largely unaffected by buffer pool size changes.

Figure 9(b.2) shows the effectiveness of the hybrid buffer pool. For comparison, we disable the bypassing of OLAP queries ("PAX-through") and the mini-page-based hybrid page ("No-mini") that directly uses the entire hybrid page as the basic I/O unit. HaSiS delivers preferable OLTP/OLAP performance and hit ratios compared to PAX-through and No-mini. PAX-through shows the worst hit ratio and performance as it allows OLAP queries (rarely re-accessed) to go through the buffer pool, leading to severe pollution and I/O contention. Without mini-pages and the invalidation-based approach, hybrid pages have to be frequently evicted from the buffer pool, leading to sub-optimal performance.

• **Effects of multi-version storage.** Figure 9(c) shows the space usage and performance considering the multi-version storage by varying the OLAP projection. A higher projection indicates a broader range of column values (i.e., more hybrid pages) being retrieved. As depicted, the logical space usage increases with the increment of AP projection, due to the additional allocation of delta pages for reserving history records being accessed by OLAP queries. However, leveraging the in-storage compression of CSD which is capable of compressing these sparse over-provisioned delta pages, the physical space usage remains stable (70.1GB to 78.2GB) despite the increase in AP projection. Meanwhile, the OLTP throughput remains stable regardless of the increasing AP projection.

• **Effects of page size.** Figure 9(d) shows the OLTP/OLAP performance considering different column page sizes (from 16KB to 2048KB) of HaSiS. The delta page remains 16KB for all cases. As shown, increasing column page size contributes to reduced OLAP latency (larger batches during OLAP scan), but leads to worse OLTP throughput (higher write amplification during OLTP updates), and vice versa. Therefore, we choose 128KB, which achieves the balanced OLTP and OLAP performance, as the default column page size for HaSiS.

• **Effects of parallelism.** Figure 9(e) illustrates the effects of partition sizes, from 500K (e.g., if more than 500K records are indexed by one B$^+$-Tree, it is partitioned as two trees) to 25K. As shown, with the finer-grained partition, both OLTP and

Table 2: Performance and space usage under CSD and SSD.

Device	TP Throughput (KOPS)	AP Response Time (sec/op)	Space Usage (GB)
CSD-3310	139.6	3.12	71.2
SN570-SSD	139.4	3.11	168.7

OLAP performance improves due to better parallelism, which is achieved by finer-grained locking, as a smaller partition allows a fewer number of partitions to be accessed per operation: the measured lock duration time of OLTP and OLAP is reduced from 61.7μs/op (μs per operation) to 3.14μs/op and 2.31s/op (second per operation) to 1.12s/op, respectively.

• **Effects of segment.** Figure 9(f) shows the effects of the segment size (from 50MB to 1,000MB) of the global log, in terms of performance and recovery time. The observed increase in OLTP performance is significant (91.2%) when increasing the segment size to 100MB, and then it diminishes when it is larger than 100MB, as the benefits of accumulating temporary transactional updates diminish. The OLAP performance is stable (variance up to 2.64%). The recovery time of HaSiS climbs from 4.1ms to 155.5ms with increasing segment size. The larger the segment is, the more temporary logs need to be replayed during recovery, resulting in a longer recovery time.

• **Performance under heavy-hitter workloads.** Extremely overlapping OLTP and OLAP requests could cause a blocking effect between TP and AP operations. To exhibit the worst-case behaviors of HaSiS, we manually insert OLTP updates into TPC-C requests at different TP repeating ratios, from 10:1 to 500:1. Under the 10:1 case, for example, we insert one OLTP update identical to one of the previous ten OLTP requests. By doing so, we arbitrarily create "hot zones" undergoing intensive OLTP updates and are retrieved (blocked) by currently running OLAP queries. We choose Q10, Q5, and Q3 from the TPC-H workloads (from large to skewed scan range). In Figure 9(g), in Q3, OLTP/OLAP performance improves since fewer delta pages are required to temporarily maintain the blocked updates, as shown in Figure 9(h). Q5 and Q10 exhibit similar trends. Note that this case (rarely occurs in practice) only demonstrates the worst-case behaviors as analytical queries normally exhibit no localities.

• **Effects of CSD.** We demonstrate the effect of close integration of HaSiS and CSD by deploying HaSiS on a CSD and comparing with that on a 1TB Western Digital SN570 NVMe SSD. Shown in Table 2, by leveraging high bandwidths of modern SSDs, HaSiS, benefiting from its compaction-enabled batched I/Os, demonstrates superior OLTP/OLAP performance with both CSD and NVMe SSDs. However, without transparent compression, HaSiS on a normal SSD consumes over twice physical storage space due to the sparsely managed per-page logs, highlighting that CSD with transparent compression plays an essential role in the HaSiS design. In addition, HaSiS can scale effectively across multiple CSDs, leveraging their combined bandwidth and extended sparse data management capabilities. Thus, HaSiS can handle increasing workload demands and larger datasets while maintaining high performance and storage efficiency.

6 Other Related Work

Proteus [1] and FSM [8] dynamically adjust the row/column data format in adaptation to workload changes. Dziedzic *et al*. [24] proposes to adaptively allocate CPU resources between OLTP or OLAP tasks depending on workload demands. These solutions tend to have a prolonged period for data reorganization or resource reallocation, which impacts data freshness and system speed performance. P-Tree [60] is an in-memory column-based store that only merges index changes when transactional updates necessitate modifications to the global index layout. It can achieve instant data freshness and efficient index utilization, which however may compromise OLTP performance since each update requires time-consuming exclusive reorganization of the index. CedarDB [55], assuming row pages are hot and column pages are cold, may result in performance inefficiencies: OLTP updates to column pages and OLAP queries on row pages could cause significant write and read amplification. Tobias et al. [63] proposed an update-aware Near Data Processing (NDP) architecture that accumulates host-side update deltas and shares them with computational storage for in-situ OLAP execution. AIDE [38] proposed an intermediate-layer method for databases to offload vendor-specific OLAP executions to computational storage. Considering the ever-changing nature of data, these NDP solutions, which offload computation-heavy OLAP tasks to in-storage computing, still have to face the challenge of maintaining data freshness.

7 Conclusion

This paper presents HaSiS that effectively harnesses the capabilities of emerging Computational Storage Devices (CSDs) with built-in transparent compression. HaSiS addresses the longstanding trade-off between data freshness and performance in HTAP systems by leveraging CSD's ability to create sparse data structures on storage without impacting physical capacity. We propose a single-index design to achieve instant data freshness and a hybrid page approach to decouple OLTP performance from OLAP efficiency. The efficacy of HaSiS has been thoroughly evaluated and validated through comprehensive experiments on a commercially available CSD platform equipped with built-in transparent compression.

Acknowledgments

We thank our shepherd, Liuba Shrira, and the anonymous reviewers for their constructive feedback and insightful comments. The work described in this paper is supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14202123, GRF 14200224), US NSF grants (CCF-2210755, CCF-2312509, CCF-2210754, CCF-2312508), the National Key R&D Program of China (Grant 2023YFB2703600), and the National Natural Science Foundation of China (Grant 62272271).

References

[1] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. Proteus: Autonomous Adaptive Storage for Mixed Workloads. In *ACM International Conference on Management of Data (SIGMOD)*, pages 700–714, USA, 2022. ACM.

[2] Alberto Lerner and Philippe Bonnet. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2852–2858, China, 2021. ACM.

[3] Alejandro Vera Baquero and Ricardo Colomo Palacios and Owen Molloy. Real-time Business Activity Monitoring and Analysis of Process Performance on Big-data Domains. *Telematics Informatics*, 33(3):793–807, 2016.

[4] Amazon. Aurora. https://aws.amazon.com/rds/aurora/, 2025.

[5] Anastassia Ailamaki and David J. DeWitt and Mark D. Hill and Marios Skounakis. Weaving Relations for Cache Performance. In *International Conference on Very Large Data Bases (VLDB)*, pages 169–180, Italy, 2001. Morgan Kaufmann.

[6] Apache. Apache ORC. https://orc.apache.org/, 2025.

[7] Apache. Apache Parquet. https://parquet.apache.org/, 2025.

[8] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *ACM International Conference on Management of Data (SIGMOD)*, pages 583–598, USA, 2016. ACM.

[9] Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, Liran Mishali, Yeqin Mo, Nir Pachter, Dima Sivov, Vinoth Veeraraghavan, Vladi Vexler, Lei Wang, and Peng Wang. Industrial Strength OLTP Using Main Memory and Many Cores. *Proc. VLDB Endow.*, 13(12):3099–3111, 2020.

[10] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1–10, USA, 1995. ACM Press.

[11] Nils Boeschen and Carsten Binnig. GaccO - A GPU-accelerated OLTP DBMS. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1003–1016, USA, 2022. ACM.

[12] Peter A. Boncz, Thomas Neumann, and Viktor Leis. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.*, 13(11):2649–2661, 2020.

[13] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, USA, 2003.

[14] Burger, Charlie and Spagnolo, Paul. Accelerate with IBM Storage: DS8880/DS8880F Thin Provisioning. *IBM Washington Systems Center—Storage*, page 89, 2017.

[15] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *IEEE International Conference on Data Engineering(ICDE)*, pages 2859–2872, Malaysia, 2022. IEEE.

[16] Hongzhi Chen, Bowen Wu, Shiyuan Deng, Chenghuan Huang, Changji Li, Yichao Li, and James Cheng. High Performance Distributed OLAP on Property Graphs with Grasper. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2705–2708, USA, 2020. ACM.

[17] 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. ByteHTAP: ByteDance's HTAP System with High Data Freshness and Strong Data Consistency. *Proc. VLDB Endow.*, 15(12):3411–3424, 2022.

[18] Christian Riegger and Tobias Vinçon and Robert Gottstein and Ilia Petrov. MV-PBT: Multi-Version Indexing for Large Datasets and HTAP Workloads. In *International Conference on Extending Database Technology, (EDBT)*, pages 217–228, Denmark, 2020. OpenProceedings.org.

[19] Clickhouse. Query billions of rows in milliseconds. https://clickhouse.com/, 2025.

[20] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[21] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *ACM*

International Conference on Management of Data (SIGMOD), pages 339–351, CHINA, 2021. ACM.

[22] DapuStor. DapuStor R610X. https://www.dapustor.com/product/14.html, 2025.

[23] Dhruba Borthakur and Jonathan Gray and Joydeep Sen Sarma and Kannan Muthukkaruppan and Nicolas Spiegelberg and Hairong Kuang and Karthik Ranganathan and Dmytro Molkov and Aravind Menon and Samuel Rash and Rodrigo Schmidt and Amitanand S. Aiyer. Apache Hadoop Goes Realtime at Facebook. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1071–1080, Greece, 2011. ACM.

[24] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R. Narasayya, and Manoj Syamala. Columnstore and B+ tree - Are Hybrid Physical Designs Important? In *ACM International Conference on Management of Data (SIGMOD)*, pages 177–190, USA, 2018. ACM.

[25] Facebook. Zstandard Compression. https://facebook.github.io/zstd/, 2023.

[26] Nuno Faria, José Pereira, Ana Nunes Alonso, Ricardo Vilaça, Yunus Koning, and Niels Nes. TiQuE: Improving the Transactional Performance of Analytical Systems for True Hybrid Workloads. *Proc. VLDB Endow.*, 16(9):2274–2288, 2023.

[27] Gilad Mishne and Jeff Dalton and Zhenghua Li and Aneesh Sharma and Jimmy Lin. Fast Data in the Era of Big Data: Twitter's Real-time Related Query Suggestion Architecture. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1147–1158, USA, 2013. ACM.

[28] Gui Huang and Xuntao Cheng and Jianying Wang and Yujie Wang and Dengcheng He and Tieying Zhang and Feifei Li and Sheng Wang and Wei Cao and Qiang Li. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD*, pages 651–665, The Netherlands, 2019. ACM.

[29] Guoliang Li and Chao Zhang. HTAP Databases: What is New and What is Next. In *International Conference on Management of Data (SIGMOD)* , pages 2483–2488, USA, 2022. ACM.

[30] HaSiS. Implementation. https://github.com/ericaloha/Hasis, 2025.

[31] 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. TiDB: A Raft-based HTAP Database. *Proc. VLDB Endow.*, 13(12):3072–3084, 2020.

[32] Yunxin Huang, Aiguo Song, Chao Guo, and Yafei Yang. ASIC design of LZ77 compressor for computational storage drives. *Electronics Letters*, 59(22):e13000, 2023.

[33] Huang, Kecheng and Shen, Zhaoyan and Shao, Zili and Zhang, Tong and Chen, Feng. Breathing New Life into an Old Tree: Resolving Logging Dilemma of B+-tree on Modern Computational Storage Drives. *Proceedings of the VLDB Endowment*, 17(2):134–147, 2023.

[34] IBM. IBM FCM-V3. https://www.ibm.com/docs/en/flashsystem-7x00/8.5.0?topic=to-flashcore-modules, 2025.

[35] Jong-Bin Kim and Jaeseon Yu and Jaechan Ahn and Sooyong Kang and Hyungsoo Jung. Diva: Making MVCC Systems HTAP-Friendly. In *International Conference on Management of Data (SIGMOD)*, pages 49–64, USA, 2022. ACM.

[36] Kemper, Alfons and Neumann, Thomas. HyPer: A Hybrid OLTP&OLAP Main Memory Database System based on Virtual Memory Snapshots. In *IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, Germany, 2011. IEEE, IEEE.

[37] 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 Zaït. Oracle Database In-Memory: A Dual Format In-memory Database. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1253–1258, South Korea, 2015. IEEE Computer Society.

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

[39] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *IEEE International Conference on Data Engineering (ICDE*, Australia, 2013.

[40] Linux. Linux Gzip Compression. https://linuxize.com/post/gzip-command-in-linux/, 2022.

[41] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *ACM International Conference on Management of Data (SIGMOD)*, pages 37–50, USA, 2017. ACM.

[42] Mark Wilkening and Udit Gupta and Samuel Hsia and Caroline Trippel and Carole-Jean Wu and David Brooks and Gu-Yeon Wei. RecSSD: Near Data Processing for Solid State Drive based Recommendation Inference. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 717–729, USA, 2021. ACM.

[43] Oracle. InnoDB Storage Engine. https://dev.mysql.com/doc/refman/8.4/en/innodb-introduction, 2025.

[44] Oracle. MySQL. https://www.mysql.com/, 2025.

[45] Oracle. Oracle Database. https://www.oracle.com/database/, 2025.

[46] Pezzini, Massimo and Feinberg, Donald and Rayner, Nigel and Edjlali, Roxane. Hybrid Transaction/Analytical Processing will Foster Opportunities for Dramatic Business Innovation. *Gartner, Available at https://www.gartner.com/doc/2657815/hybrid-transactionanalyticalprocessing-foster-opportunities*, pages 4–20, 2014.

[47] PingCAP. TiDB: Deploy a local test cluster. https://docs.pingcap.com/tidb/stable/quick-start-with-tidb, 2025.

[48] PingCAP. TiDB: Key metrics description. https://docs.pingcap.com/tidb/stable/grafana-overview-dashboard, 2025.

[49] Aunn Raza, Periklis Chrysogelos, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. Adaptive HTAP through Elastic Resource Scheduling. In *ACM International Conference on Management of Data (SIGMOD)*, pages 2043–2054, USA, 2020. ACM.

[50] Rebecca Taft and Irfan Sharif and Andrei Matei and Nathan VanBenschoten and Jordan Lewis and Tobias Grieger and Kai Niemi and Andy Woods and Anne Birzin and Raphael Poss and Paul Bardea and Amruta Ranade and Ben Darnell and Bram Gruneir and Justin Jaffray and Lucy Zhang and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*, pages 1493–1509, USA, 2020. ACM.

[51] Richard L. Cole and Florian Funke and Leo Giakoumakis and Wey Guy and Alfons Kemper and Stefan Krompass and Harumi A. Kuno and Raghunath Othayoth Nambiar and Thomas Neumann and Meikel Poess and Kai-Uwe Sattler and Michael Seibold and Eric Simon and Florian Waas. The mixed workload ch-benchmark. In *International Workshop on Testing Database Systems (DBTest)*, page 8, Greece, 2011. ACM.

[52] SAP. SAP HANA. https://www.sap.cn/products/technology-platform/hana/what-is-sap-hana.html, 2025.

[53] Scaleflux. Computational storage drive with built-in transparent compression. https://scaleflux.com/, 2025.

[54] Scaleflux. ScaleFlux 3000-series SSDs. https://scaleflux.com/products/csd-3000/, 2025.

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

[56] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* , pages 219–238, USA, 2021. USENIX Association.

[57] Vishal Sikka, Franz Färber, Anil K. Goel, and Wolfgang Lehner. SAP HANA: The Evolution from a Modern Main-Memory Data Platform to an Enterprise Application Platform. *Proc. VLDB Endow.*, 6(11):1184–1185, 2013.

[58] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *ACM International Conference on Management of Data (SIGMOD)*, pages 731–742, USA, 2012. ACM.

[59] SingleStore, Inc. SingleStore (MemSQL). https://www.singlestore.com/, 2025.

[60] Yihan Sun, Guy E. Blelloch, Wan Shen Lim, and Andrew Pavlo. On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes. *Proc. VLDB Endow.*, 13(2):211–225, 2019.

[61] The PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/, 2025.

[62] Thomas Neumann and Tobias Mühlbauer and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 677–689, Australia, 2015. ACM.

[63] Tobias Vinçon and Christian Knödler and Leonardo Solis-Vasquez and Arthur Bernhardt and Sajjad Tamimi and Lukas Weber and Florian Stock and Andreas Koch and Ilia Petrov. Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads. *Proc. VLDB Endow.*, 15(10):1991–2004, 2022.

[64] Tobias Vinçon and Christian Knödler and Leonardo Solis-Vasquez and Arthur Bernhardt and Sajjad Tamimi and Lukas Weber and Florian Stock and Andreas Koch and Ilia Petrov. Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads. *Proc. VLDB Endow.*, 15(10):1991–2004, 2022.

[65] TPC. TPC-C benchmark. https://www.tpc.org/tpcc/, 2025.

[66] TPC. TPC-H benchmark. https://www.tpc.org/tpch/, 2025.

[67] 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. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data*, 1(2):199:1–199:25, 2023.

[68] Jing Wang, Hanzhang Yang, Chao Li, Yiming Zhuansun, Wang Yuan, Cheng Xu, Xiaofeng Hou, Minyi Guo, Yang Hu, and Yaqian Zhao. Boosting Data Center Performance via Intelligently Managed Multi-backend Disaggregated Memory. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, page 37, USA, 2024. IEEE.

[69] Wang, Jing and Li, Chao and Mei, Junyi and He, Hao and Wang, Taolei and Wang, Pengyu and Zhang, Lu and Guo, Minyi and Wu, Hanqing and Chen, Dongbai and Liu, Xiangwen. HyFarM: Task Orchestration on Hybrid Far Memory for High Performance Per Bit. In *IEEE International Conference on Computer Design (ICCD)*, pages 33–41, 2022.

[70] Xubin Chen and Ning Zheng and Shukun Xu and Yifan Qiao and Yang Liu and Jiangpeng Li and Tong Zhang. KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression. In *International Workshop on Data Management on New Hardware (DaMoN)*, pages 3:1–3:10, China, 2021. ACM.

[71] 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, Jeffrey F. Naughton, and John Cieslewicz. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow.*, 13(12):3313–3325, 2020.

[72] Yifan Qiao and Xubin Chen and Ning Zheng and Jiangpeng Li and Yang Liu and Tong Zhang. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 69–82, USA, 2022. USENIX Association.

[73] Zhe Yang and Youyou Lu and Xiaojian Liao and Youmin Chen and Junru Li and Siyu He and Jiwu Shu. λ-IO: A Unified IO Stack for Computational Storage. In *USENIX Conference on File and Storage Technologies(FAST)*, pages 347–362, USA, 2023. USENIX Association.