Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP

Tobias Schmidt
Technische Universität München
tobias.schmidt@in.tum.de

Dominik Durner
CedarDB
durner@cedardb.com

Viktor Leis
Technische Universität München
leis@in.tum.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Businesses are increasingly demanding real-time analytics on up-to-date data. However, current solutions fail to efficiently combine transactional and analytical processing in a single system. Instead, they rely on extract-transform-load pipelines to transfer transactional data to analytical systems, which introduces a significant delay in the time-to-insight. In this paper, we address this need by proposing a new storage engine design for the cloud, called *Colibri*, that enables hybrid transactional and analytical processing beyond main memory. Colibri features a hybrid column-row store optimized for both workloads, leveraging emerging hardware trends. It effectively separates hot and cold data to accommodate diverse access patterns and storage devices. Our extensive experiments showcase up to 10x performance improvements for processing hybrid workloads on solid-state drives and cloud object stores.

PVLDB Reference Format:
Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. PVLDB, 17(11): 3290 - 3303, 2024.
doi:10.14778/3681954.3682001

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/umbra-db/colibri-vldb2024.

1 INTRODUCTION

Data-intensive applications increasingly demand real-time analytics on up-to-date data. Current approaches require extract-transform-load (ETL) pipelines to transfer transactional business data into analytical systems, delaying the time to insight. To address this challenge, modern database systems should support both online transactional processing (OLTP) for small updates and online analytical processing (OLAP) for complex queries on large data sets within a single system [47]. Systems extend their support for hybrid transactional and analytical processing (HTAP) to meet this demand. Snowflake, for instance, recently added HTAP capabilities to its cloud data warehouse [63].

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682001

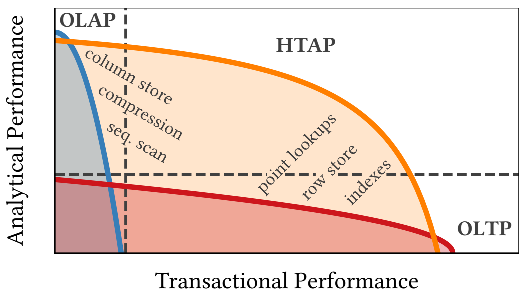


Figure 1: Performance characteristics and technologies of OLTP and OLAP systems.

Earlier research on in-memory database systems has shown the feasibility of hybrid transactional and analytical processing when the entire database fits into the server's main memory. By keeping all records in main memory, these systems can overcome the limitations of traditional storage engines. In-memory tables and index structures allow for efficient point lookups and sequential scans, and the absence of disk I/O reduces access times. However, the capacity and price of main memory have stagnated [29, 49], limiting scalability and making these systems uneconomical.

To alleviate the stagnation of main memory, current systems rely on persistent storage to handle larger-than-memory workloads. In such systems, the storage engine is one of the most important components for achieving high performance. Storage engines retrieve the data from persistent storage, buffer the data in memory, and synchronize access to the data. They are at the heart of any database management system, tightly coupled with the query engine, the transaction management, and the logging system. Implementation details of the storage engine, such as the physical data representation, the supported access paths, and the storage technology itself determine whether a database system is suitable for transactional or analytical workloads.

Current larger-than-memory systems struggle to combine transactional and analytical processing. Because these workloads have diverging requirements, existing storage engines optimize only for one of the workloads, as illustrated in Figure 1. For example, OLTP systems use a row-based storage format that is ideal for small updates and point lookups. However, row stores are inefficient for OLAP workloads due to high read amplification: analytical queries scan many tuples but typically access only a fraction of the columns. OLAP systems leverage compressed columnar formats for fast sequential scans. Because compression and columnar layout complicate updates to individual records, the system must rewrite large chunks of data to incorporate small changes. Additionally, some analytical systems lack secondary indexes, making point lookups prohibitively expensive for transactional workloads.

Besides the data layout, the storage technology is crucial for the overall performance. In the last decade, two trends have redefined the design of the storage engines: (1) the widespread adoption of Solid-State Drives (SSDs) and (2) the migration of data processing pipelines to the cloud. SSDs offer superior bandwidth and lower latency than traditional Hard Disk Drives (HDDs), making them a cost-efficient alternative to main memory for storage-intensive workloads. Cloud database-as-a-service (DBaaS) offerings provide unparalleled flexibility, elasticity, higher availability, and data durability. The separation of storage and compute, facilitated by inexpensive cloud storage, is driving the success of DBaaS solutions. They already account for 50 % of the database market [59].

The cloud offers multiple storage options, including instance-local SSDs, remote instances, and object stores. These technologies share similar characteristics: (a) high bandwidth, (b) moderate latencies, and (c) (almost) unlimited capacities. A single SSD achieves a read bandwidth of up to 7 GB/s (PCIe 4), and RAID configurations can retrieve more than 50 GB/s. Access latencies on modern SSDs are in the order of 100 µs [30]. Similarly, data centers nowadays provide network bandwidths of 100 Gbit/s or more, and cross-instance network latencies can be less than 80 µs, with further reduction opportunities using RDMA [75]. Cloud object stores, like Amazon S3, also achieve bandwidths of up to 200 Gbit/s. However, request latencies are significantly higher: 100 ms per request are to be expected [21]. Nevertheless, they offer almost unlimited capacities, high durability, and availability. While SSD RAIDs do not offer unlimited storage size, hundred TB configurations are practical.

In this paper, we propose the *Colibri* storage engine design. It optimizes for hybrid transactional and analytical processing beyond main memory, leveraging new hardware trends. The design (a) features a hybrid column-row store for HTAP workloads, (b) separates hot and cold data to optimize the different access patterns and storage devices, and (c) exploits the high bandwidth modern storage devices offer. Colibri retains the transactional performance of row stores and achieves the analytical performance of column stores. The proposed storage engine design is tailored to solid-state drives and cloud object stores and exploits their similarities. It efficiently processes hybrid workloads in on-premise and cloud deployments.

The paper is organized as follows: First, we analyze existing cloud-native OLTP, OLAP, and HTAP systems and highlight the design decisions that lead to the architectures in Section 2. Based on the insights, we propose the Colibri system architecture for cloud-native HTAP systems in Section 3. Section 4 describes the implementation of a storage engine optimized for the sketched architecture. We highlight several optimizations that improve transactional performance on a hybrid column-row store. In Section 5, we describe optimizations to fully utilize high-bandwidth storage devices and improve point lookups in column stores. We implement Colibri in the relational DBMS Umbra and evaluate it against other systems in Section 6.

2 CLOUD-NATIVE DATABASE DESIGNS

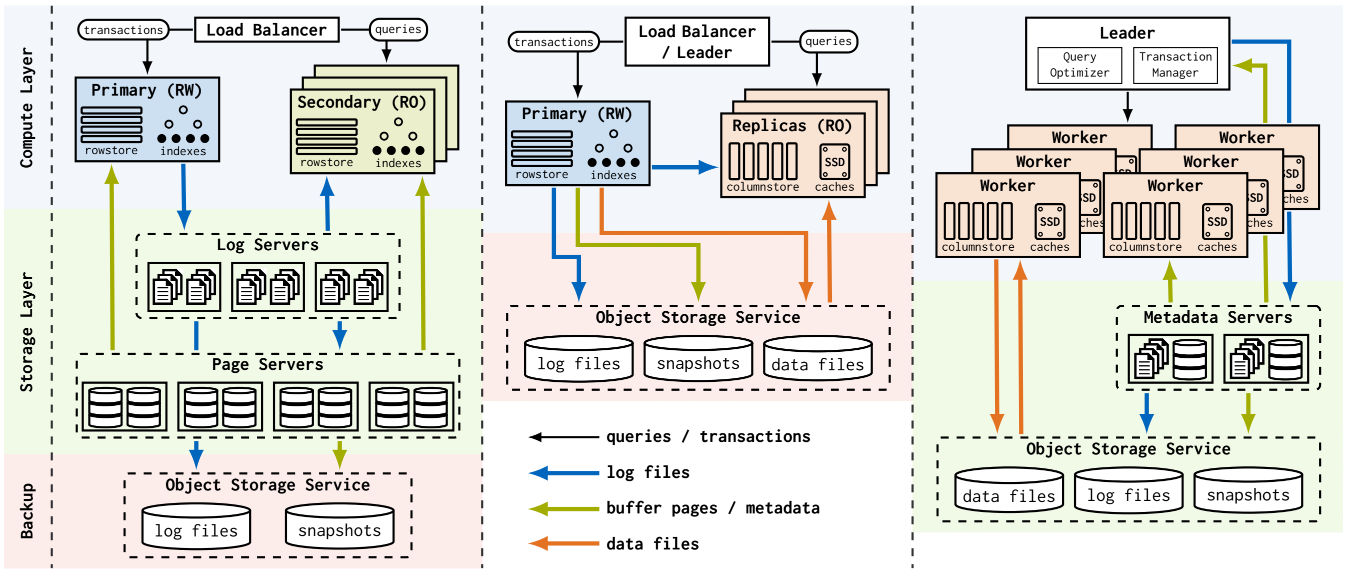
To design an efficient storage engine capable of processing analytical and transactional queries simultaneously, we first analyze state-of-the-art cloud-native database engine designs. Figure 2 depicts the architecture and components common to most systems developed by cloud service providers such as Amazon AWS, Microsoft Azure, or Alibaba Cloud. We distinguish three classes of designs based on the workloads they support: OLTP, OLAP, and HTAP.

OLTP. Early transactional designs have utilized the high availability disaster recovery (HADR) architecture, replicating the database to secondary nodes to improve availability and durability [26]. However, this architecture limits the database's size and fails to meet customer demands for elasticity and performance. Therefore, cloud-native designs separate the storage layer from the database system and provide a dedicated infrastructure, as shown in Figure 2a. The primary node ships the log entries to dedicated log servers. The log servers are responsible for persisting log entries and sending log records to secondary nodes and page servers. In contrast to the HADR architecture, page servers store the database state and serve buffer pages to the compute layer. The primary goal of the log servers is to ensure durability, while page servers provide availability. Microsoft's Socrates pioneered this architecture [8], and Huawei's Taurus database later adopted it [18].

Other systems, such as Amazon Aurora or PolarDB, follow this architecture but do not separate the log and page servers. Aurora combines the log and page servers into one storage node [66], and PolarDB writes logs and pages to a custom low-latency distributed filesystem [46, 69]. While the general architecture is based on a single writer, recent work has explored multiple write nodes through sharding [13] or multi-master designs [19].

Most cloud-native OLTP systems evolved from existing on-premise systems: Taurus, Aurora, and PolarDB are based on MySQL and Socrates builds upon SQL Server. Cloud vendors adapt the storage system and transaction management but reuse the other parts of the database. The cloud systems maintain the row-oriented storage format to support high transaction rates.
Insights — Cloud-native OLTP systems introduce an optimized storage layer to improve elasticity, availability, and durability. The new architecture allows to process workloads that exceed the capacity of a single machine. Separating the storage from the database is pivotal for scaling the two layers independently and sharing storage servers between multiple tenants. However, the row-oriented storage format limits the performance on analytical workloads.

OLAP. The architecture of cloud-native OLAP systems, depicted in Figure 2c, differs from OLTP designs. The leader, also known as the coordinator node, accepts queries, manages a cluster of worker nodes, and handles transactions. This node optimizes the incoming queries, generates execution plans, and assigns tasks to the worker nodes. The user data is typically stored in a column-oriented format on high-availability object stores like Amazon S3 or Azure Blob Storage and retrieved by the worker nodes. This storage-separated design is implemented by several cloud data warehouses, such as Redshift [11, 28], Snowflake [17], and AnalyticDB [72], enabling them to process petabytes of data.



(a) OLTP architecture **(b) HTAP architecture** **(c) OLAP architecture**

Figure 2: Cloud-native database designs. We sketch the components common to most systems and the data flow between them.

While object stores serve as the primary storage layer for user data, metadata like table schemas, transaction logs, and visibility information of data files are stored in a separate database. Snowflake, for instance, relies on FoundationDB [74], a distributed key-value store, to manage the metadata and guarantee transactional consistency (e.g., snapshot isolation) [65]. Open table formats like Delta Lake or Apache Iceberg avoid the metadata layer. Instead, they write a transaction log directly to the object store but sacrifice transactional performance for the simplicity of the architecture [2, 10].
Insights — Cloud data warehouses rely on column stores and scale out to process analytical workloads at petabyte-scale efficiently. The storage layer is decoupled from the database using object stores that offer high durability and availability. An external metadata service stores transactional data and ensures ACID properties.

HTAP. The customer demand for real-time analytics on transactional data has led to the development of cloud-based HTAP systems working on a single state. Cloud offerings like MySQL HeatWave [4] or PolarDB-IMCI [67] evolved from existing OLTP systems and copy the data to column stores to improve analytical performance. Similar to OLTP systems, a primary node processes write requests and ships the log entries to the replica nodes (cf. Figure 2b). The replica nodes replay the log and apply the changes to an in-memory column store. Analytical queries are answered on the columnar replica, while the primary node retains its transactional performance. F1 Lightning proposes a database-agnostic approach that allows seamless integration with existing OLTP systems [68]. ByteHTAP is a cloud-native OLTP system with page and log servers that integrates a distributed OLAP engine. The log servers send the log entries to the storage layer of the OLAP engine, which creates a column store for fast analytics [15].

SingleStore, a cloud-native HTAP system, follows a similar architecture but uses an in-memory row store for the most recent data. New records are periodically transformed into columnar format and written to an object store by the primary node. A cluster of worker nodes answers analytical queries on the database [57].
Insights — Most HTAP architectures are built upon OLTP systems and maintain a read-optimized replica of the data for fast analytics. Cloud-native systems like SingleStore avoid duplication and persist the data in a column store.

3 COLIBRI: A CLOUD-NATIVE HTAP DBMS ARCHITECTURE

Based on the analysis from Section 2, we propose Colibri, a cloud-native HTAP architecture illustrated in Figure 3. The design integrates the key features of OLTP and OLAP systems to support both workloads. Resembling a cloud-native OLAP system, the compute layer consists of a primary node (leader) and a cluster of worker nodes that execute queries. The leader node accepts incoming requests and executes transactions, similar to cloud-native OLTP systems. Analytical queries can be forwarded to worker nodes. A storage layer with dedicated log and page servers persists the data and ensures durability and availability. Furthermore, Colibri relies on a hybrid column-row storage format for transactional and analytical processing. In the following, we briefly explain our design considerations that led to this architecture:

(1) Separation of hot and cold data. Whereas OLTP transactions generally only access a small fraction of the data, OLAP workloads often scan large parts of the database. We, therefore, partition tables into hot and cold areas based on the access pattern and choose different data placement strategies and formats for each part. Hot data items are frequently queried by OLTP transactions through point accesses, requiring multiple columns to perform updates. We, therefore, store them in an uncompressed row-based format on page

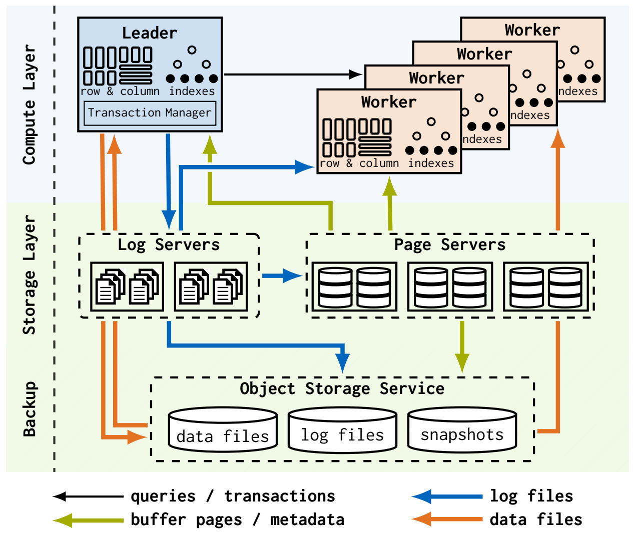


Figure 3: Colibri, a cloud-native HTAP architecture, combines the benefits of OLAP and OLTP systems.

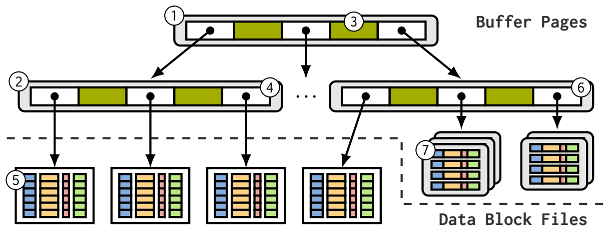


Figure 4: Hybrid column-row store. A B$^+$-tree indexes compressed and uncompressed blocks.

servers for fast access. Cold data elements, in contrast, are primarily used for analytics in sequential scans. Analytical queries typically access only some columns, making columnar storage formats with compression ideal for improving query runtime.

Colibri's hybrid column-row store is the key to achieving good HTAP performance. Several in-memory HTAP systems, such as SAP HANA or SQL Server, utilize a combination of row and column storage layouts to effectively handle both workloads [40, 61]. SingleStore also combines row and column stores but does not provide a storage layer with page servers. It relies on LSM-Trees instead of page-based data structures like B-trees [57].

(2) Exploiting high bandwidth storage devices. Cloud storage services like Amazon S3 offer up to 200 Gbit/s network bandwidth to a single compute node. Table scans on large datasets can exploit these high bandwidths and process data at the speed of local solid-state drives [21]. Therefore, all nodes in the cluster have direct access to the object storage service.

(3) Minimize the number of log entries. For durability and recoverability, most modern systems, including cloud-native databases, rely on ARIES-style logging [52]. Before a transaction can commit, the write-ahead log must be made persistent and sent to the log servers. Minimizing the number of log entries improves performance, especially in a cloud-native OLTP architecture: the primary node spends less time creating and shipping log entries, the log servers persist less data, and the page servers replay fewer changes.

(4) Bypassing page and log servers for bulk operations. Loading large amounts of data is a common pattern in analytical workloads. OLAP systems like Redshift or Snowflake directly write the data to object stores and only update the metadata referencing the new files. A single write to the metadata suffices to insert thousands of rows. Cloud-native HTAP systems must also adopt this approach to avoid excessive logging for bulk operations.

Compared to OLTP systems, Colibri stores most of the data in object stores, which offer high durability and availability. Only hot data items and indexes are kept on the page servers, to reduce access latencies. From the perspective of an analytical system, we add log and page servers to support transactional processing. The servers avoid high latencies caused by logging changes to the object store or an external database. In addition, the system uses a traditional buffer manager, including page-based index structures, which are essential for transactional workloads.

The sketched architecture relies on remote instances and object stores as storage layer. We can also deploy the system to a single instance with local storage: the system writes the buffer pages, write-ahead log, and data files to local drives instead of a remote storage layer. While this option reduces operation costs, it strongly impacts durability and availability. Since SSDs have similar characteristics, Colibri works for the cloud and on-premise.

This paper focuses on the efficient implementation of Colibri's architecture, in particular, the hybrid column-row storage format. We provide a blueprint for HTAP storage engine and describe various optimizations. Colibri facilitates high-performance query processing for transactional and analytical workloads. We describe the integration with object stores and adapt data access patterns to high-bandwidth storage devices. For the design of the page and log servers, we refer the reader to existing work [8, 18, 66].

4 HYBRID STORAGE ENGINE

Designing an HTAP storage engine is challenging due to the opposing characteristics of transactional and analytical processing. Our solution to this problem is the Colibri architecture from Section 3, which relies on a hybrid column-row store to separate hot and cold data. In this section, we describe the storage engine's design, the efficient implementation of transactional operations, such as insert, update, and delete, and the integration into the RDBMS Umbra [53].

4.1 Column-Row Store

To achieve fast performance for both transactional and analytical workloads, we use two different data formats within the same relation. Based on the access pattern, we decide which format to employ. Hot data items are stored in an uncompressed row-based format for fast OLTP, while cold data is compressed and stored in a columnar format. Our storage engine with a hybrid column-row store seamlessly queries and transitions between these two formats. Individual records are located through a sorted record tree.

We utilize a B$^+$-tree to organize the records based on their associated row ID. Every tuple is assigned a unique and monotonic increasing row ID upon creation, i.e., the records are implicitly sorted by the insertion order. The B$^+$-tree stores blocks of records, which are either compressed or uncompressed. A block is identified by the smallest row ID of its contained records. Our storage engine design is based on the observation that the newest tuples are typically hot tuples. Therefore, the most recent blocks in the B$^+$-tree will contain the uncompressed hot data items. In contrast, earlier created blocks will be compressed to minimize bandwidth constraints during cold data analytics. Note that once a cold record is updated, we decompress it and move it to the hot section.

Figure 4 shows the hybrid column-row store and the record tree. The B$^+$-tree consists of ① inner nodes and ② leaf nodes. We sort the tree using ③ row IDs as keys in both node types. Inner nodes reference child nodes, while the leaf nodes reference the *blocks*. A block consists of several thousand records in compressed columnar or uncompressed row-based formats. ④ Compressed blocks reference exactly one ⑤ data block file; ⑥ Uncompressed blocks reference several ⑦ data pages. Besides the file reference, the compressed blocks contain a header with statistics like min-max-bounds for pruning the search space during query processing. Data block files and data pages store the user data.

Uncompressed data pages use traditional buffer pages that store a few hundred records. We refer to these pages as row-store; however, the uncompressed pages materialize records in a PAX (partition attributes across) layout, i.e., every column is stored in a separate array on the page [6]. To ensure durability and recoverability, all modifications are written to the write-ahead log. Uncompressed data pages support inserting, deleting, and updating individual records. Colibri stores B$^+$-tree nodes on traditional buffer pages.

While uncompressed data pages are efficient for transactional workloads, they are suboptimal for analytical workloads since all columns are on the same buffer page. OLAP queries usually access only a few columns of a relation; thus, column stores are more efficient. Especially for datasets larger than main memory, reading unnecessary data wastes precious bandwidth. To overcome the bandwidth bottleneck, we compress the data to minimize the storage size using the read-optimized Data Blocks format [39]. Data Blocks offers fast decompression and predicate evaluation on encoded data and filters data based on small materialized aggregates [51].

A compressed data block file is immutable after creation, and we persist it before committing the transaction. Hence, logging the changes to the referencing leaf node in the B$^+$-tree is sufficient and no additional log entries for the data block files are required. The column-row store, therefore, consists of two parts: (1) the buffer pages stored in the buffer file or on page servers, where every change must be recorded in the write-ahead log, and (2) the data block files stored as regular files on disk or in an object store without any logging overhead.

This design is beneficial for large tables and saves a lot of space in the buffer file. For instance, the `lineitem` table of the TPC-H benchmark (SF1000) stores 354 GB in data block files, and only 41 MB of buffer pages are needed. Table 1 breaks down the space consumption of the different components of the column-row store. One data block file contains up to 262 144 records, and every leaf has a fan-out of 51 blocks, resulting in 13 million records per node.

Table 1: Structure of the column-store for the TPC-H table `lineitem` at SF1000. We report the size of the components and how often they are used. Note that the (un-) compressed blocks are part of leaf nodes and do not consume disk space.

	Size	Count	Comment
① inner node	64 KB	1	4093 children per inner node
② leaf node	64 KB	450	up to 51 blocks per leaf
④ compressed block	1272 B	22 909	1 data block file per block
⑤ data block file	~16 MB	22 909	262 144 records per file
⑥ uncompressed block	1272 B	3	up to 78 data pages per block
⑦ data page	64 KB	202	~380 records per page

A two-layer tree already stores more than 50 billion records, as inner nodes have up to 4 000 children. Only the last 73 000 tuples at the end of the table are uncompressed and use buffer pages.

4.2 Inserts

HTAP systems must optimize for two kinds of inserts: OLTP transactions that add only a few records to a table and analytical workloads that insert large batches. We implement two variants for inserting new tuples.

If the transaction inserts only a few tuples, we add the new records to an uncompressed data page. New tuples are always appended to the end of the table, i.e., the right-most leaf node in the B$^+$-tree. Each thread has an exclusive page to insert new rows into to minimize contention[1]. For batch inserts, we bypass the uncompressed data pages and immediately create a data block file. New records are first materialized in thread-local buffers large enough to fit an entire compressed block. Once the buffer is full, the collected rows are compressed, and a new block is added to the B$^+$-tree. Before the transaction commits, the database system flushes the data block files and syncs the data. The left-over tuples in the buffers are inserted into uncompressed data pages.

Compared to the first variant, the second implementation produces only one log entry per block instead of one per tuple. It also avoids torn writes, as the transaction only commits after persisting the data. Without this optimization, every new record requires three writes: (1) to the write-ahead log, (2) the dirty page is copied to a double-write buffer to avoid torn writes, and (3) it is written to the buffer file. Compressing the tuples upfront reduces the data size even further. Especially in the cloud, batching reduces cost and improves performance as we minimize the number of log entries and bypass the OLTP infrastructure.

4.3 Deletes

Deleting records from the hybrid column-row store requires two steps: (a) first, the database collects the row IDs of the tuples to remove, and (b) then removes the records. For an uncompressed block, we locate the data page containing the record and mark the tuple as deleted on the buffer page. This operation requires one log entry to update the page.

For compressed blocks, deletions are more involved as the file is immutable. Because the records are stored in a columnar format,

[1]A maintenance task later merges the thread-local pages to avoid underfull nodes.

deleting a single tuple and rewriting all columns is too expensive. Similar to SQL Server and SingleStore [40, 57], we maintain an additional buffer page with a bitset to track the deleted tuples based on their position in the block. The bitset page is kept on page servers for faster access and in-place updates. Deleting a compressed tuple sets the corresponding bit in the bit vector to indicate that the record is not visible anymore. This operation requires one log entry to update the bitset page and make the change durable.

Like inserts, deletions often occur in batches in OLAP workloads. We optimize for this pattern and delete multiple tuples in a single operation. Row IDs are sorted upfront, and all records from the same block are deleted together. We create a single log entry that contains the identifiers of all tuples to remove from the block.

After the delete operation is globally visible, i.e., no other transaction started before, we remove the deleted records and files. We schedule asynchronous maintenance tasks that rewrite a compressed data block if more than one-fourth is deleted. Empty blocks without visible records are removed from the B$^+$-tree, and the corresponding pages and files are freed.

4.4 Updates

In column stores, updates can be implemented as deletion of the old record, followed by inserting a new tuple with the modified columns. We also adopt this approach for the compressed data blocks: we mark the tuples as deleted, extract the original values, and then insert a new tuple using the regular insertion logic. The row-based layout makes in-place updates possible on uncompressed data pages: we modify the record on the page and log the old values in case of a rollback. We assume that modifications to hot tuples are more common, and most updates happen on uncompressed pages using the in-place logic. Especially for OLTP applications, in-place updates improve the throughput significantly.

Column stores can avoid out-of-place updates using differential update structures like the Positional Delta Tree [32]. However, this approach requires an additional tree structure to store the deltas and complicates data retrieval. We, therefore, prefer out-of-place updates and only use in-place updates for hot tuples. Additionally, reinserting the tuple at the end of the relation moves them from the cold section to the hot section, and subsequent updates are faster.

4.5 Maintenance

The hybrid column-row store requires several maintenance tasks to keep the B$^+$-tree balanced, eliminate dead records, and compact cooling parts of the table. We implement the following four tasks to avoid degradations:

(1) Merge under-full tree nodes and blocks.
(2) Move hot uncompressed tuples to cold compressed blocks.
(3) Remove deleted tuples from uncompressed data pages and compressed data blocks.
(4) Delete empty data files from the object store.

The cost of the maintenance tasks depends on the storage format. In hot areas, the data pages are small and contain only a few hundred tuples. Compressed data blocks, in contrast, are much larger and store more than 100 000 tuples. We, therefore, differentiate between short-running and long-running maintenance tasks.

Table 2: Query performance for varying hot-cold thresholds and buffer sizes (TPC-H SF100, queries per second).

		hot-cold threshold							
		0.1 s	0.3 s	1 s	3 s	10 s	30 s	100 s	300 s
buffer size	8 GB	0.45	0.45	0.46	0.42	0.43	0.31	0.29	0.23
	16 GB	0.60	0.59	0.60	0.57	0.54	0.44	0.35	0.30
	32 GB	1.10	1.09	1.08	0.98	0.94	0.89	0.79	0.61
	64 GB	2.04	2.08	1.96	1.85	1.82	1.74	1.69	1.60
	128 GB	2.77	2.77	2.71	2.62	2.57	2.57	2.54	2.46

Short maintenance tasks are executed on the fly while traversing the B$^+$-tree. They are fast and touch only a few buffer pages, e.g., merging under-full leaves and inner nodes in the B$^+$-tree or removing deleted tuples from uncompressed data pages that are not visible anymore. We also merge blocks in the leaves and half-full data pages from the same uncompressed block. Both tasks typically run in 50 µs and consume in total less than 0.5 % of the computational resources for TPC-C.

Long-running tasks, on the other hand, involve multiple pages and create a new compressed block. We execute these tasks in the background to avoid blocking queries. For tasks (1) and (3), we collect the IDs of compressed blocks that are less than half full and then schedule a background job to merge two data blocks. Similarly, for task (4), a background job deletes the data files from the object store and removes the block from the B$^+$-tree.

Compressing tuples is crucial for query performance. However, moving tuples to the cold section too early or too late can degrade transactional or analytical workloads. For example, if we compress tuples and a subsequent transaction tries to modify them, we cannot use the in-place update logic. Hence, every leaf in the B$^+$-tree tracks the transaction ID and timestamp of the last transaction that performed an insert, update, or delete in one of the blocks. If this transaction has become globally visible, i.e., no older transaction exists, and the last modification happened more than 10 seconds ago, the leaf and its blocks are cold, and we schedule a background job to compress multiple data pages into one data block.

The five-minute rule suggests a 40 s threshold for PCIe v3 SSDs [9, 27]; yet, our microbenchmarks show that a 10 s threshold works better for newer PCIe generations. Table 2 lists the query performance for different hot-cold thresholds on TPC-H. We use two clients: one to continuously insert new (hot) data and another to execute the 22 queries. The buffer size impacts the performance and the threshold to use. When the data fits in memory, scanning hot and cold data is equally fast. Compression enhances scan performance once the dataset exceeds the buffer pool. For small buffer sizes (8 to 32 GB), performance varies by 2x between a 0.1 s and a 300 s threshold. The 10 s are a tradeoff between performance and a reasonable threshold for transactional workloads.

4.6 Integration with Modern Buffer Managers

When loading the data block files into memory for query processing, we must store the files somewhere in memory. Umbra implements a state-of-the-art buffer manager optimized for transactional workloads [42]. We cache the retrieved data blocks in the buffer manager and store them beside traditional buffer pages. Note that we load the compressed column individually to avoid read amplification.

Table 3: Database recovery (TPC-H SF1000, `lineitem`).

	Load time	WAL size	#Log entries	Recovery
Umbra	3 099 s	1.28 TB	6 166 M	5 352 s
Colibri	453 s	1.02 GB	5.17 M	2.99 s

Storing all data in one buffer pool simplifies memory management and allows evicting data blocks in case of memory pressure. However, unlike regular pages, the size of the columns varies and depends on the compression ratio and the number of records in the block. We, therefore, use variable-sized pages to store the compressed columns [53]. Colibri loads the columns from the local file system or an external object store into the buffer manager. As the columns are immutable, the buffer manager does not need to write the pages back.

4.7 Multi-Version Concurrency Control

Database systems provide isolation and consistency guarantees through concurrency control. For transactions, isolation is crucial to ensure correctness but requires careful implementation to avoid performance degradation. Like many other in-memory and disk-based systems [20, 35, 40, 56, 61], Umbra implements a timestamp-based multi-version concurrency control protocol (MVCC). It maintains multiple versions of a tuple, and transactions read the version valid at their start. Updating an existing tuple adds a new version of this record to the database. MVCC allows efficient transaction isolation through snapshots and improves concurrency for long-running queries and transactional workloads.

Umbra adpots HyPer's version chain approach [54] to support in-place updates. Updates move the old version of a tuple into a version chain and link them to the modified page. We implement this protocol in the hybrid column-row store to ensure snapshot isolation. The in-memory version chains track the creation, deletion, or updates of blocks and tuples. This lightweight MVCC protocol provides the necessary isolation and consistency guarantees for transactional workloads and the required performance [24].

4.8 Recovery

Colibri exploits the immutable data block files to reduce the number of log entries. We batch inserts, compress tuples into data blocks, and bypass the log. In this section, we analyze the benefits and how it affects correctness in case of a crash.

Table 3 compares the recovery of Umbra and Colibri. We consider the case where the database crashes immediately after loading the `lineitem` table but before writing a checkpoint, forcing a full recovery from the WAL. Loading the 6 billion rows in Umbra takes 52 minutes, with a log size of 1.28 TB. Analyzing and replaying the changes takes more than one hour. Colibri, in contrast, only logs the changes to the B$^+$-tree and the uncompressed data pages. The vast majority of the data is compressed and written to a durable storage device before the transaction commits, reducing the WAL to 1 GB. Hence, Colibri recovers in less than 3 seconds.

Colibri considers data block files durable and, therefore, does not duplicate them in the WAL. We flush the data block files to a durable storage device before committing to ensure correctness and avoid data loss. The files can be viewed as part of the database's log; however, they do not need to be replayed and compress the data efficiently. In cloud environments, object stores like Amazon S3 or Azure Blob Storage offer sufficient bandwidths and are highly reliable and durable [21]. As a result, Colibri writes only 1 GB of log data and 354 GB of data block files, four times less than Umbra.

Table 4: Data Format Comparison (TPC-H SF100, `lineitem`).

	Full Table Scan [row/s]	Filtered Scan [row/s]	Point Lookups [row/s]	Size
PAX (uncompr.)	**68.4 M**	98.2 M	1.01 M	88.7 GB
Arrow	34.1 M	16.8 M	183	96.8 GB
Data Blocks	31.5 M	**99.4 M**	**1.18 M**	34.1 GB
Parquet	19.0 M	7.29 M	758	34.5 GB
Parquet (Snappy)	11.2 M	5.82 M	752	**23.9 GB**

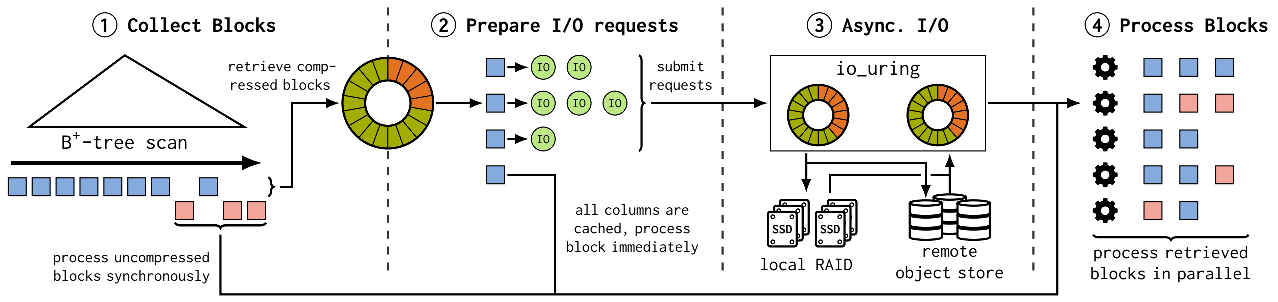
4.9 Columnar Data Formats

With the rise of data lakes and data warehouses, open storage formats such as Apache Parquet [3] and Apache ORC [1] have become popular. They offer interoperability between systems, high compression ratios, and optimize for analytical access patterns. Although some systems, like DeltaLake [10] or Iceberg [2], adopt these formats as primary storage formats, most databases use them only to import or export data. They lack efficient support for transactional workloads and cannot be fine-tuned for the system's query engine. However, off-the-shelf formats, such as CSV, JSON, or Apache Parquet, can leverage server-side filtering (e.g., S3 Select [7]) to reduce network requirements for an additional surcharge.

For the hybrid column-row store, two features are essential: (i) fast decompression of individual tuples for point access and (ii) evaluating filters on encoded data for query performance. The *Data Blocks* format supports both operations [39]. It employs only simple compression schemes like frame of reference, single value, or dictionary encoding and avoids heavyweight compression algorithms like LZ4 or Snappy [37]. Hence, the format is fast to decode, and conditions like equality or range filters can be evaluated on the compressed data [5, 77]. This is a good trade-off in cloud systems as the CPU is more expensive than storage space, and network bandwidths match decompression speeds nowadays.

Table 4 compares the Data Blocks format, Apache Parquet, Apache Arrow, and Umbra's PAX layout. We evaluate three common HTAP access patterns: full table scans, filtered scans[2], and point lookups. PAX and Arrow perform best for full table scans, storing data in memory without decompressing it. For filtered scans and point lookups, Data Blocks is superior due to vectorized filters and lightweight encodings. Parquet is slightly slower for sequential scans but still within the same order of magnitude. Point lookups are slow due to complex encodings, like run-length or delta encoding. Regarding the data size, Parquet and Data Blocks are almost identical: compressing the data using Snappy reduces the size further by 31 % but increases access times. Formats like ORC perform similarly to Parquet [48, 71].

[2]We disabled data skipping indexes like small materialized aggregates and zone maps for this workload.



① Collect Blocks
② Prepare I/O requests
③ Async. I/O
④ Process Blocks
B+-tree scan
retrieve comp-ressed blocks
process uncompressed blocks synchronously
IO
submit requests
all columns are cached, process block immediately
io_uring
SSD
local RAID
remote object store
process retrieved blocks in parallel

Figure 5: Bandwidth-optimized table scan. The scan is split into four tasks to asynchronously retrieve compressed blocks from local or remote storage devices. We use io_uring for asynchronous I/O as proposed by [21].

4.10 Retrieval Cost Optimization

In cloud object stores, customers pay for every request. Retrieving large objects is cheaper, and higher download bandwidths are possible. For Amazon S3, Durner et al. show that 16 MB requests are cost optimal and achieve the full per-request bandwidth [21]. The Data Blocks format typically requires one or two bytes per compressed value. With 262 144 records per block, a column is 0.25 MB or 0.5 MB large, far from the optimum.

We, therefore, combine multiple compressed data block files into a segment to increase the request size. A segment puts together the columns from multiple blocks, such that a single request retrieves one column from all merged blocks. We allocate one segment per leaf page in the B^+-tree and store the segment files in the object store. The segment files are created either on the fly during bulk inserts or through maintenance tasks. For the `lineitem` table, we merge 51 data blocks into one segment, and downloading one column for the entire table takes only 450 requests. This optimization reduces retrieval costs by up to 10x.

4.11 Secondary Indexes

Our relation does not sort the data but instead orders the tuples by time of insertion. This is beneficial for splitting the relation into hot and cold parts, as we assume that the most recent records are the most frequently modified ones. However, transactional workloads require efficient point lookups. Umbra, therefore, uses B^+-trees to access individual records efficiently. In Section 5.1, we explain how to efficiently extract individual tuples from compressed data blocks.

The secondary indexes are stored on standard buffer pages, and all modifications are logged. In a cloud setting, we can also use the secondary indexes efficiently as they reside on the page servers. Cloud-native OLAP systems fail to support traditional index structures, a critical component for transactional workloads. Instead, they rely on small-materialized aggregates (SMA) [51], zone maps [25], or caches [60, 62] to skip data. Colibri exploits both secondary indexes and SMAs to avoid full table scans.

5 QUERY PROCESSING

After describing the transactional operations, this section explores query processing on the hybrid column-row store. High-performance query engines provide outstanding analytical performance for in-memory data. However, if the data is not cached in memory, it is difficult to read sufficiently many records to exploit the query engine's full potential. Especially datasets larger than main memory pose a challenge to row-stores.

HTAP systems access relational data in two different ways: point lookups and (filtered) table scans. We first describe how we extract individual records efficiently. Then, we introduce a bandwidth-optimized table scan for reading data from SSDs and object stores.

5.1 Point Lookups

Point lookups are a common access pattern in OLTP workloads, but they also occur in OLAP queries with index nested-loop joins. Databases avoid expensive full table scans using indexes to search for individual tuples. A secondary index maps the searched key to a row ID and retrieves the corresponding tuple from the table. In the case of the column-row store, lookups (A) first traverse the B^+-tree, (B) find the corresponding block on the leaf page, and either (C.1) read an uncompressed data page or (C.2) extract the compressed tuple. For high scalability, it is essential to minimize contention on shared resources and efficiently decompress individual tuples [45].

We optimize these steps as follows: Colibri uses optimistic lock coupling to traverse the tree in steps (A) and (B) without acquiring a mutex [23, 44]. It combines lock coupling and optimistic latches to minimize contention [43]. On the leaf page, we perform a binary search to find the block that contains the row ID. In case of an uncompressed block, we search for the uncompressed data page, load it using lock coupling, and continue in step (C.1). For step (C.2), we use optimistic latches to read the columns of a compressed block without acquiring multiple shared latches for every column.

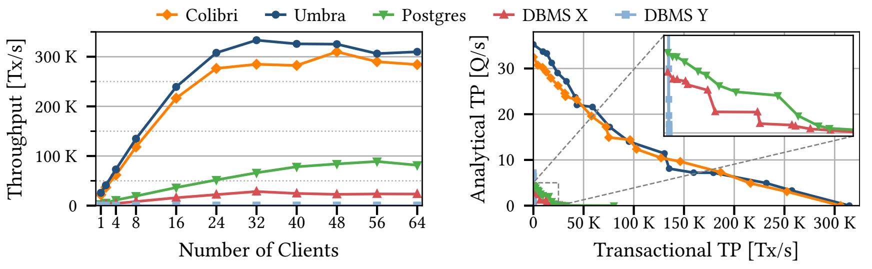
5.2 Bandwidth-Optimized Table Scans

Networks and SSDs easily retrieve multiple gigabytes per second nowadays. However, the database must adapt its storage engine to exploit these high bandwidths and hide I/O latencies [34]. Asynchronous I/O interfaces, like io_uring or Linux AIO, allow the system to schedule sufficient many requests to leverage the available bandwidth. We implement an asynchronous table scan that optimizes bandwidth and transparently handles network storage services and local solid-state drives. The scan is split into four tasks and builds on the cloud storage integration of Durner et al. [21].

Figure 5 illustrates the individual tasks and their interaction. Task ① is executed upfront and collects all blocks that qualify in a ring buffer. Task ② - ④ run in parallel and retrieve the missing files either from the SSD or an object store and process them. We

Table 5: Hardware Platforms

	epyc	i3en.metal
CPU	AMD EPYC 7713	Intel Xeon 8175
Threads	256 (2.0 GHz)	96 (3.1 GHz)
Memory	1024 GB	768 GB
SSDs	8x Samsung PM9A3	8x NVMe SSD
Storage	14 TB (56 GB/s)	60 TB (16 GB/s)
Network	-	100 Gbit/s



(a) TPC-C: transactional throughput **(b) CH-benCHmark: combined throughput**

Figure 6: Performance comparison on the TPC-C and Ch-benCHmark workload. We use up to 64 transactional and analytical clients.

adaptively schedule threads to each task depending on the current processing speed, retrieval bandwidth, and buffer space usage. Morsel-driven parallelism allows multiple threads to process the same block simultaneously in task (4).

We ensure that the speed at which new requests are created matches the retrieval bandwidth and that the retrieval speed does not overtake the processing speed. Furthermore, we limit the size of the outstanding requests to 10 % of the buffer pool; otherwise, performance degrades as other queries steal the buffer pages. We maintain global statistics to track resource usage and balance all tasks to fully utilize the available resources.

Efficient data filtering is crucial for analytical processing. Besides the small materialized aggregates, Umbra uses optimized vectorized scans to efficiently eliminate records before passing them on. Simple restrictions, like equality or comparisons with a constant, are evaluated on the compressed data block using modern SIMD instructions. Additionally, we use bloom filters as semi-join reducers and push them into table scans to apply joins early on. Cloud-native OLAP systems apply similar optimizations [11, 40]

6 EVALUATION

In the following section, we evaluate the performance of the Colibri design with the hybrid column-row store. All components are integrated into the relational database management system Umbra [53]. Our experiments show that the proposed approach processes analytical and transactional workloads efficiently. Colibri runs queries on data sizes that exceed the main memory and exploits the high bandwidth of modern storage devices. The proposed architecture is cost-effective and fits well into modern cloud environments.

6.1 Experimental Setup

We conduct experiments in two different settings on the machines from Table 5. Instance-local storage experiments use our local server **epyc** and the cloud storage experiments run on an AWS **i3en.metal** instance. Both systems have similar characteristics, and we maximize SSDs' bandwidth and capacity using a RAID 0 configuration.

For the instance-local experiments, we evaluate the performance of our storage engine against several other analytical and transactional database systems. We use the following systems: (a) three OLTP-optimized databases with row stores, the original version of Umbra, PostgreSQL, and a commercial system DBMS X, (b) two OLAP systems, the commercial DBMS Y and the open-source database DuckDB [58], (c) and the HTAP database SingleStore. DBMS Y and DuckDB use a column store, and SingleStore combines a column store for analytical queries with an in-memory row store for recent changes [57]. Note that Umbra uses PAX to organize the columns within a buffer page more efficiently [6]. In the following experiments, we refer to Umbra with the Colibri architecture as Colibri. All systems write their data and log files to the RAID array.

Our benchmarks assess the OLTP and OLAP performance in a combined setting using the throughput frontier metric [50]. Each system is evaluated with varying combinations of analytical and transactional clients, ranging from 1 to 64 clients each. The throughput frontier reports the skyline of the measured throughput a system achieves in each dimension. We prepare queries and transactions upfront and send multiple requests through message pipelining to minimize the communication overhead.

For the cloud experiments, we compare the performance of Colibri against Snowflake (large cluster) and Amazon Redshift (4-node ra3.16xlarge cluster) on AWS. For Umbra and Colibri, we use i3en.metal instances and S3 buckets in the eu-central-1 region.

6.2 Instance-local HTAP Performance

We first evaluate the column-row store on a single instance to highlight analytical and transactional performance differences. Our experiments demonstrate that the Colibri design outperforms the other systems on OLTP, OLAP, and HTAP workloads.

6.2.1 TPC-C. We begin our evaluation by comparing the transactional capabilities of the different systems. Figure 6a shows the transactional rates for the TPC-C benchmark with 100 warehouses. The workload consists of 92 % write and 8 % read transactions.

Umbra achieves the highest transaction throughput starting at 26 000 Tx/s, and scales to 325 000 Tx/s with 32 clients. The OLTP-optimized systems, PostgreSQL and DBMS X, peak at 90 000 Tx/s and 30 000 Tx/s. DBMS Y, which entirely relies on a column store, executes only 210 Tx/s, and the performance decreases with multiple clients. Colibri, with its row store for the hot part of data processes 300 000 Tx/s, only 8 % less than the row store.

Table 6 breaks down the performance improvements by combining column and row stores. Without any of the optimizations from an OLTP system, the column store processes only a few transactions per second. Adding secondary indexes improves the performance

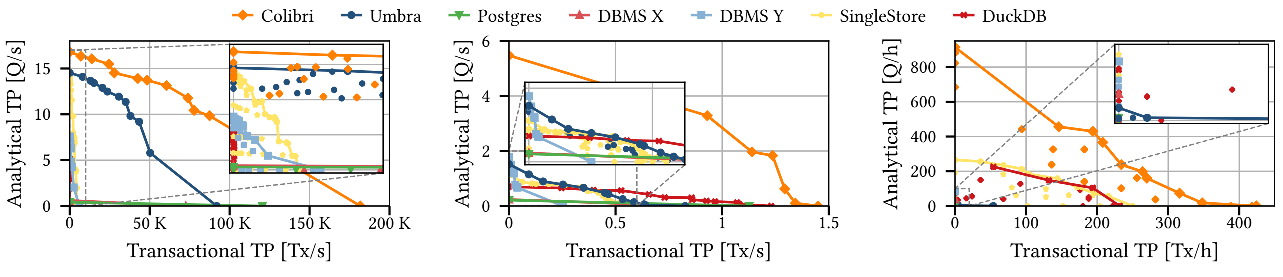


Figure 7: HTAP performance. We limit the buffer pool size to 128 GB (note that the buffer space for SingleStore is not restricted).

Table 6: Breakdown of the transactional performance. Colibri without secondary indexes, row store, and in-place updates is the baseline. We enable the optimizations step-by-step and report the transaction rates on TPC-C with a 100 warehouses.

Colibri versions	1 client	32 clients
column store	3 Tx/s	45 Tx/s
+ secondary indexes	465 Tx/s	2 685 Tx/s
+ row store	5 404 Tx/s	94 644 Tx/s
+ in-place updates	21 508 Tx/s	288 719 Tx/s

by two orders of magnitude. The row store further increases transaction rates by more than 10x. In-place updates, which are only supported in row stores, speed up the transactions by a factor of 4. Without the OLTP components, it is impossible to process transactional workloads efficiently.
Insights — Multi-versioning in combination with Umbra's in-memory version chains provides the highest transactional throughput. The Colibri design with a hybrid column-row store has only a small overhead and improves the transactional performance of analytical column stores by more than three orders of magnitude.

6.2.2 *CH-benCHmark.*

Next, we run the TPC-C benchmark with analytical queries to evaluate the systems' HTAP capabilities as suggested by Cole et al. [16]. The benchmark consists of 22 queries executed in parallel on the changing TPC-C tables; the queries are derived from the TPC-H benchmark. Figure 6b visualizes the transactional and analytical throughput: we execute the benchmark for different combinations of transactional and analytical clients to demonstrate how these workloads affect each other. The plotted line shows the skyline of the measured throughput for each system.

The OLTP-optimized systems (PostgreSQL and DBMS X) achieve the same transactional throughput as before but fail to provide a competitive analytical performance. Both systems execute no more than five queries per second, even without any update transactions. The OLAP system DBMS Y achieves a higher query throughput, but the transactional rates are low. We observe that neither of the other systems dominates in both dimensions. The OLAP and HTAP systems perform better in analytical workloads and OLTP systems in transactional workloads. Only Umbra maintains high transactional and analytical throughput at the same time. As the data set fits into the buffer pool, the system processes the analytical queries in memory, and the data format does not affect the performance.

Insights — Row stores provide competitive performance for in-memory analytics. However, systems must optimize both dimensions to achieve high HTAP throughput.

6.2.3 *HATtrick.*

Milkai et al. introduced the HATtrick benchmark as an HTAP workload derived from the star schema benchmark [50]. Besides the analytical queries, the benchmark includes three different transactions: two small update transactions and a read-only query that accesses roughly two hundred records.

This benchmark shows the benefits of column stores: Colibri, SingleStore, DuckDB, and DBMS Y run the analytical queries with high throughput. Umbra achieves similar performance, as the data fits into the buffer pool. Compression improves the performance even further: Colibri eliminates many blocks using small materialized aggregates and filters the data more efficiently. Surprisingly, Umbra's transactional performance is lower, and PostgreSQL and Colibri surpass it. Umbra performs a full table scan in a read-only transaction instead of using an index, as PostgreSQL does.
Insights — Read-heavy workloads benefit from columnar layouts and small materialized aggregates. However, column stores must optimize for small updates to compete with OLTP systems.

6.2.4 *HyBench.*

We further investigate the performance of the Colibri design with the HyBench benchmark [73]. The benchmark simulates a real-world finance application and consists of three stages that measure the OLAP (QPS), OLTP (TPS), and HTAP (XPS) performance. Figure 8 shows the performance of Colibri, Umbra, and Postgres using write-heavy (left) and read-heavy (right) workload patterns. Colibri performs better for the OLAP and HTAP stages, while Umbra executes more transactions in the OLTP stage. Postgres cannot match the performance of the other systems and executes up to 10 times fewer transactions and queries.

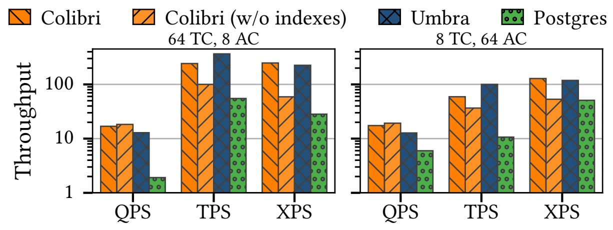
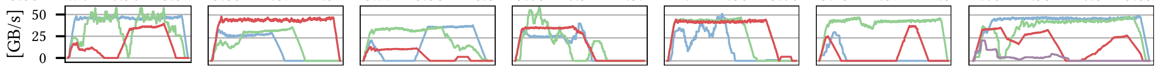


Figure 8: HyBench with different numbers of transaction (TC) and analytical (AC) clients (scale factor 100).

Table 7: Per query execution times for TPC-H SF1000 for on-premise and cloud. Colibri with the column-row store is used in *in-memory* and *on disk* / *remote*; the last two versions load all data from solid-state drives or from the object store. The bandwidth plots show the per-query data retrieval speed from disk of Colibri's asynchronous scan.

On-Premise	GM	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Row store [s]	15.6	19.1	0.30	27.4	24.1	24.4	23.2	26.5	25.0	41.4	27.5	0.56	25.8	10.1	24.6	24.9	3.20	46.6	60.1	23.4	23.7	73.1	3.37
In-Memory [s]	2.14	0.73	0.26	4.99	3.10	3.91	0.20	3.94	2.89	24.8	5.19	0.52	2.71	9.29	1.37	1.61	3.05	0.78	13.7	0.30	0.83	6.75	3.35
On disk [s]	3.88	1.90	0.70	5.62	3.65	4.89	1.31	5.99	5.36	26.8	7.08	0.74	4.39	11.1	2.86	3.08	3.15	2.64	14.8	2.05	2.77	10.5	3.67
Bandwidth																							

Cloud	GM	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
In-Memory [s]	1.92	1.74	0.40	3.09	2.59	3.03	0.28	2.48	2.38	10.3	3.84	0.52	1.59	7.19	0.82	0.92	2.40	1.79	11.6	0.85	0.83	6.38	1.58
On-disk [s]	4.73	4.14	1.41	5.20	3.69	5.91	3.02	6.52	7.49	11.8	7.20	0.91	4.67	10.1	4.25	4.24	2.49	5.34	13.6	4.68	5.28	12.7	1.94
Remote [s]	9.23	6.53	4.52	9.97	7.07	11.1	5.24	12.2	14.1	21.8	17.3	2.31	9.07	23.6	7.52	7.96	3.73	9.04	20.2	7.65	11.7	23.0	5.34
Redshift [s]	2.34	4.94	1.29	4.06	1.76	3.02	0.21	2.95	1.93	7.05	3.49	0.63	0.95	7.55	0.73	1.86	2.00	1.83	8.11	2.72	3.05	8.71	2.51
Snowflake [s]	9.15	7.30	4.10	11.1	7.02	12.2	2.79	10.5	11.4	29.7	21.1	2.41	9.16	18.6	6.63	9.25	6.60	7.19	27.6	8.13	8.04	21.4	5.19

The benchmark also shows the impact of indexes on the runtime. We turned off all index lookups and measured Colibri without indexes. The performance decreased in the OLTP and HTAP stage by more than 2x and 4x, while the analytical performance remains unaffected, as Colibri uses full table scans for the queries.

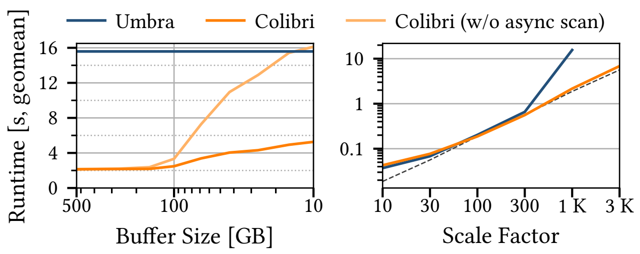
Insights — The HyBench benchmark confirms the performance of the Colibri design in a real-world scenario. Efficient index lookups are crucial for transactional and hybrid workloads.

6.2.5 TPC-H. Lastly, we run the TPC-H benchmark with scale factors 100 and 1000 to evaluate the analytical performance. We also include transactions that update large parts of the `orders` and `lineitem` tables to simulate an evolving data set. Figure 7b shows the same picture as before: the OLTP-optimized systems achieve high transaction rates while DBMS Y, DuckDB, and SingleStore perform well for analytics. DuckDB's vectorized execution engine is better suited for processing large transactions; hence, the transactional throughput increases compared to the HATtrick benchmark.

However, this time, Umbra is significantly slower than Colibri: both queries and updates are faster with a column store. The database size exceeds the 128 GB buffer space without compression, and the row store must access the disk. For scale factor 1000, this effect is even more visible (see Figure 7c). With compression, the database is already larger than the buffer pool, and every system must read from disk. Only the column-based systems provide competitive performance, as row stores suffer from high read amplification. Colibri executes 50 times more queries than Umbra using the bandwidth-optimized scan from Section 5.2.

Table 7 shows the importance of the bandwidth-optimized scan. The upper part of the table lists the execution times of the 22 TPC-H queries on Umbra (first row) and Colibri with cached data and without (second row and third row). Even as we load all the data from disk, the column-row store is 4x faster than the row store. When scanning the large `lineitem` or `orders` tables Colibri reads at least 25 GB/s for compute-heavy queries, and up to 56 GB/s.

Figure 9 investigates the impact of data set size and buffer pool size on the performance. The left plot gradually reduces the buffer size from 500 GB to 10 GB. A fraction of the buffer space is enough for Colibri to process the TPC-H queries efficiently. Without the asynchronous scan, the performance drops by more than 3x once records must be loaded from disk. Figure 9b shows that the column-row store scales almost linearly with the data set size. The row-based version provides competitive performance only if the data fits into the main memory.



(a) TPC-H SF1000 **(b) Buffer Size: 500 GB**

Figure 9: Impact of the buffer pool size and the data set size on the performance. Umbra always uses 500 GB buffer space.

Insights — Data sets that exceed the buffer pool are challenging. The column-row store, combined with the bandwidth-optimized scan, efficiently processes large data sets. Our architecture provides competitive performance for analytical and transactional workloads and exploits the high bandwidth that modern storage devices offer.

6.3 Cloud-Store Performance

We also evaluate the performance of Colibri with the hybrid row-column store in the cloud. Our experiments measure the analytical and transactional performance with different storage options. The proposed design achieves similar performance compared to commercial data warehouses.

6.3.1 OLAP. The analytical performance of Colibri on object stores or local SSDs is almost identical. Compared to the in-memory run, reading the data from disk is only 2.5x slower (cf. Table 7). Yet, the gap between the in-memory and the remote run, where Colibri retrieves all data from S3, is larger: The network is limited to 100 Gbit/s, and Colibri reads only 12 GB/s from S3, a quarter of the SSDs' bandwidth. Besides, network communication is more expensive than disk I/O, and Colibri schedules more threads as I/O workers to saturate the available bandwidth. As a result, queries answered from the object store are 4x slower than in-memory queries.

Compared to cloud-native OLAP systems, Colibri's query execution times are on par. Redshift uses four compute nodes with half the number of threads and memory each: roughly twice the compute power and memory Colibri uses. Nevertheless, Colibri achieves similar performance in a single instance. Snowflake does not disclose its hardware specifications, but a large cluster costs eight credits, equaling 16 $/h[3]. The Snowflake cluster costs one-fourth of a 4-node Redshift cluster, where each node costs 13 $/h. Hence, a four times performance slowdown is expected of Snowflake compared to Redshift. We use one i3en.metal instance that costs 10.85 $/h, without SSDs a similar instance costs 7.15 $/h.
Insights — The column-row store also performs well in the cloud. Fast data retrieval from the object store is crucial for the performance of analytical queries. The bandwidth-optimized scan exploits the available resources and provides competitive performance.

6.3.2 OLTP. Cloud-native transactional systems have multiple options where to store the database pages and log files. We evaluate the performance of Colibri on TPC-C with different storage configurations on the i3en.metal instance: (a) on the local SSDs, (b) using Amazon's elastic block storage (EBS)[4], and (c) shipping the data to another compute instance (EC2). The columnar files are stored on Amazon S3. Shipping the log to another instance corresponds to the proposed HTAP architecture in Section 3.

Surprisingly, on all three storage options, Colibri executes around 230 000 Tx/s for 32 clients. Umbra uses a low-overhead logging protocol with group commit that hides latencies [23, 31]. For all storage options, the available bandwidth is much higher than the amount of data written.
Insights — The cloud offers the necessary infrastructure to sustain high transactional throughput. Remote storage options like elastic block storage or log/page servers perform similarly to local storage.

7 RELATED WORK

This section reviews several on-premise and cloud-native database systems and their analytical and transactional capacity. In addition, we discuss how the different systems try to bridge the gap between OLAP and OLTP performance. Most systems are first implemented for one of the two workloads and use the respective storage format.
High-Performance OLTP Systems. Main-memory OLTP systems provide excellent transactional performance as the entire data set is already cached [20, 22, 35]. With the adaption of solid-state drives, modern systems try to extend this performance to larger data sets, exceeding the capacity of the main memory. Systems like LeanStore rely on asynchronous I/O to exploit the high throughput of SSDs [30, 43]. However, they struggle to provide competitive analytical performance once the data exceeds the capacity of the main memory. Some systems, like Oracle Dual-Format or HyPer, improve OLAP performance using in-memory column stores [38, 39].
Efficient Updates in Column Stores. Several on-premise systems base their storage on column stores to process analytical queries efficiently [41, 61, 64, 76]. The systems rely on delta stores [40, 61] or differential updates [32, 36] to avoid rewriting the column store for every update. However, this comes at a cost: the changes must be merged with the columnar data every time the table is scanned.

Other systems try to bridge the gap between row and column stores using a flexible storage format. Arulraj et al. propose a system that decides whether to represent an attribute in a row or columnar format based on the workload [12].
Transactions in Cloud-Native OLAP Systems. Analytical systems, like Snowflake [17], Amazon Redshift [11], or AnalyticDB [72], rely on a distributed query engine combined with a compressed column store on object storage services, like Amazon S3. Transactional data is stored in an external database, like FoundationDB [74], or on the object storage itself [10]. This design limits OLTP performance as every update modifies the column store. To improve transaction rates, Snowflake recently added support for tables in row format with secondary indexes [63].
Analytics in Cloud-Native OLTP Systems. Socrates efficiently exploits the cloud's infrastructure to provide a scalable and elastic OLTP system. Besides separating computation and storage, Socrates further splits the storage layer into log and page servers [8]. Other cloud-native OLTP systems, like Amazon Aurora [66], Taurus [18], or PolarDB [46], implement similar architectures.

MySQL Heatwave [4], PolarDB-IMCI [67], and F1 Lightning [68] evolve transactional databases into HTAP systems. They replicate the row store into a columnar format and use a distributed query engine to process analytical queries. The database system manages both representations and eliminates the need for extracting, transforming, and loading records into external analytical systems.
Cloud-native HTAP Systems. SingleStore relies on an in-memory row store on the primary node for transaction processing and a distributed column store for analytics. It converts the accumulated records from the row store into a column format and stores them in an LSM tree. SingleStore's design shows similarities with Colibri, e.g., it stores cold data on object stores. However, we rely on B^+-tree for fast point accesses and integrate with page-based systems [57].

TiDB and ByteHTAP implement a similar design: TiDB uses a distributed row store based on the Raft consensus protocol [55] and replicates the data into a column store to process analytical queries using Spark [33, 70]. ByteHTAP combines a cloud-native OLTP engine based on MySQL with an OLAP engine based on Flink [14] and creates a column store from the log message [15].

8 CONCLUSION

This paper introduces the Colibri storage engine design for hybrid transactional and analytical processing in the cloud and on-premise. Our design integrates the strengths of OLAP and OLTP architectures, enabling efficient analytics on larger-than-memory data sets while accommodating high transaction rates. We separate hot and cold data and leverage optimized storage formats for different access patterns and storage devices. Colibri's hybrid column-row store bridges the gap between these two worlds.

Our experiments demonstrate that Colibri's implementation in Umbra outperforms state-of-the-art OLAP and OLTP systems. On combined workloads, the proposed design achieves up to one order of magnitude better performance than specialized systems. We showcase that hybrid transactional and analytical processing can be achieved efficiently on solid-state drives and cloud object stores.

[3]assuming the cheapest option with 2 $ per credit
[4]We use the ebs-optimized r5b.metal instance instead of i3en.metal and combine multiple ebs devices to reach 7.5 GB/s upload and download bandwidth.

REFERENCES

[1] 2023. *Apache ORC*. Retrieved July 1, 2024 from https://orc.apache.org/

[2] 2024. *Apache Iceberg*. Retrieved July 1, 2024 from https://iceberg.apache.org/

[3] 2024. *Apache Parquet*. Retrieved July 1, 2024 from https://parquet.apache.org/

[4] 2024. *MySQL HeatWave*. Retrieved July 1, 2024 from https://www.oracle.com/mysql/heatwave/

[5] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*. ACM, 671–682.

[6] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.* 11, 3 (2002), 198–215.

[7] Amazon. 2024. *Filtering and retrieving data using Amazon S3 Select*. Retrieved July 1, 2024 from https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html

[8] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD Conference*. ACM, 1743–1756.

[9] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. 2017. The Five-minute Rule Thirty Years Later and its Impact on the Storage Hierarchy. In *ADMS@VLDB*. 1–8.

[10] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

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

[12] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD Conference*. ACM, 583–598.

[13] 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. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *ICDE*. IEEE, 2859–2872.

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[15] 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 (2022), 3411–3424.

[16] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *DBTest*. ACM, 8.

[17] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference*. ACM, 215–226.

[18] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *SIGMOD Conference*. ACM, 1463–1478.

[19] Alex Depoutovitch, Chong Chen, Per-Åke Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. 2023. Taurus MM: bringing multi-master to the cloud. *Proc. VLDB Endow.* 16, 12 (2023), 3488–3500.

[20] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference*. ACM, 1243–1254.

[21] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782.

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

[23] Michael Johannes Freitag. 2023. *Building an HTAP Database System for Modern Hardware*. Ph.D. Dissertation. Technische Universität München.

[24] Michael J. Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems. *Proc. VLDB Endow.* 15, 11 (2022), 2797–2810.

[25] Goetz Graefe. 2009. Fast Loads and Fast Queries. In *DaWaK (Lecture Notes in Computer Science)*, Vol. 5691. Springer, 111–124.

[26] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis E. Shasha. 1996. The Dangers of Replication and a Solution. In *SIGMOD Conference*. ACM Press, 173–182.

[27] Jim Gray and Gianfranco R. Putzolu. 1987. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *SIGMOD Conference*. ACM Press, 395–398.

[28] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD Conference*. ACM, 1917–1923.

[29] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*. www.cidrdb.org.

[30] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (2023), 2090–2102.

[31] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD Conference*. ACM, 877–892.

[32] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter A. Boncz. 2010. Positional update handling in column stores. In *SIGMOD Conference*. ACM, 543–554.

[33] 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 (2020), 3072–3084.

[34] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proc. VLDB Endow.* 17, 3 (2023), 577–590.

[35] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[36] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. 2011. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *Proc. VLDB Endow.* 5, 1 (2011), 61–72.

[37] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26.

[38] 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. 2015. Oracle Database In-Memory: A dual format in-memory database. In *ICDE*. IEEE Computer Society, 1253–1258.

[39] 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 *SIGMOD Conference*. ACM, 311–326.

[40] 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 (2015), 1740–1751.

[41] Per-Åke Larson, Eric N. Hanson, and Susan L. Price. 2012. Columnar Storage in SQL Server 2012. *IEEE Data Eng. Bull.* 35, 1 (2012), 15–20.

[42] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25.

[43] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. IEEE Computer Society, 185–196.

[44] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[45] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*. ACM, 3:1–3:8.

[46] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (2019), 2263–2272.

[47] Guoliang Li and Chao Zhang. 2022. HTAP Databases: What is New and What is Next. In *SIGMOD Conference*. ACM, 2483–2488.

[48] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proc. VLDB Endow.* 16, 11 (2023), 3044–3056.

[49] David B. Lomet. 2018. Cost/performance in modern data stores: how data caching systems succeed. In *DaMoN*. ACM, 9:1–9:10.

[50] Elena Milkai, Yannis Chronis, Kevin P. Gaffney, Zhihan Guo, Jignesh M. Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *SIGMOD Conference*. ACM, 1810–1824.

[51] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. Morgan Kaufmann, 476–487.

[52] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

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

[54] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD Conference*. ACM, 677–689.

[55] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*. USENIX Association, 305–319.

[56] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861.

[57] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric N. Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *SIGMOD Conference*. ACM, 2340–2352.

[58] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD Conference*. ACM, 1981–1984.

[59] Adam Ronthal, Rick Greenwald, Xingyu Gu, Ramke Ramakrishnan, Aaron Rosenbaum, and Henry Cook. 2023. Magic Quadrant for Cloud Database Management Systems.

[60] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2022. Predicate caching: Query-driven secondary indexing for cloud data warehouses. In *SIGMOD Conference*. ACM.

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

[62] Snowflake. 2023. Search Optimization Service. https://docs.snowflake.com/en/user-guide/search-optimization-service. accessed: 2024-05-29.

[63] Snowflake. 2024. *Snowflake Hybrid Tables*. Retrieved July 1, 2024 from https://docs.snowflake.com/en/user-guide/tables-hybrid

[64] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. ACM, 553–564.

[65] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425.

[66] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD Conference*. ACM, 1041–1052.

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

[68] 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. 2020. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow.* 13, 12 (2020), 3313–3325.

[69] Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Wenchao Zhou. 2023. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proc. VLDB Endow.* 16, 12 (2023), 3754–3767.

[70] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[71] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (2023), 148–161.

[72] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *Proc. VLDB Endow.* 12, 12 (2019), 2059–2070.

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

[74] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *SIGMOD Conference*. ACM, 2653–2666.

[75] Tobias Ziegler, Dwarakanandan Bindiganavile Mohan, Viktor Leis, and Carsten Binnig. 2022. EFA: A Viable Alternative to RDMA over InfiniBand for DBMSs?. In *DaMoN*. ACM, 10:1–10:5.

[76] Marcin Zukowski and Peter A. Boncz. 2012. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.* 35, 1 (2012), 21–27.

[77] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. IEEE Computer Society, 59.