ByteHTAP: ByteDance's HTAP System with High Data Freshness and Strong Data Consistency

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, Yuming Liang*
ByteDance US Infrastructure System Lab, *ByteDance, Inc

ABSTRACT

In recent years, at ByteDance, we see more and more business scenarios that require performing complex analysis over freshly imported data, together with transaction support and strong data consistency. In this paper, we describe our journey of building ByteHTAP, an HTAP system with high data freshness and strong data consistency. It adopts a separate-engine and shared-storage architecture. Its modular system design fully utilizes an existing ByteDance's OLTP system and an open source OLAP system. This choice saves us a lot of resources and development time and allows easy future extensions such as replacing the query processing engine with other alternatives.

ByteHTAP can provide high data freshness with less than one second delay, which enables many new business opportunities for our customers. Customers can also configure different data freshness thresholds based on their business needs. ByteHTAP also provides strong data consistency through global timestamps across its OLTP and OLAP system, which greatly relieves application developers from handling complex data consistency issues by themselves. In addition, we introduce some important performance optimizations to ByteHTAP, such as pushing computations to the storage layer and using delete bitmaps to efficiently handle deletes. Lastly, we will share our lessons and best practices in developing and running ByteHTAP in production.

PVLDB Reference Format:
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, Yuming Liang. ByteHTAP: ByteDance's HTAP System with High Data Freshness and Strong Data Consistency. PVLDB, 15(12): 3411-3424, 2022.
doi:10.14778/3554821.3554832

1 INTRODUCTION

In recent years, at ByteDance, we start seeing more and more business scenarios that require performing complex analysis over freshly imported data, together with transaction support and strong data consistency. For example, Bytedance's User Growth Department demands complex SQL queries over constantly changing data such as business attribute relationships and advertisement costs, where the latest data changes are expected to be visible in subsecond delay.

However, we can not find an off-the-shelf solution to satisfy our customer's needs. Traditional OLAP systems typically load large amounts of data in bulk periodically but suffer from the problem of stale data. In contrast, traditional OLTP systems support DMLs and can execute point look-up queries efficiently, but lack the massive parallel processing capability. Therefore, they cannot process complex queries over large amounts of data efficiently. To meet our business needs, a hybrid transaction/analytical processing (HTAP) system is the most appropriate choice.

Specifically, we built a large-scale real-time analytics HTAP system that supports both fresh data changes and strong data consistency with the following design goals:

- **Large scale**. Several ByteDance's popular apps, such as TikTok, Douyin and Toutiao, have hundreds of millions of daily active users. Hence, we want to build a distributed real-time analytic system that can scale up to petabytes of data.
- **Real time**. We want OLTP and OLAP queries in ByteHTAP to have comparable performance when running them in standalone OLTP/OLAP systems. This is important for migrating existing customers' workloads to ByteHTAP, where those workloads are currently run in standalone systems.
- **Highly fresh data changes**. To explore new business opportunities, some customers want the most recent data changes to be available for querying within a one-second delay. This imposes a strong requirement on our system design. Currently, our customers have a delay of minutes or even hours in terms of data freshness.
- **Strong data consistency**. Currently, many customers import data from their OLTP databases to their data warehouses to do data analysis. Therefore, it is hard for them to get a consistent global snapshot across OLTP and OLAP engines, and application developers have to spend extra effort handling data consistency issues. Hence, customers want ByteHTAP to provide native support for strong data consistency.

HTAP systems have been widely discussed in recent years in both academia and industry [37, 51, 62]. Several dedicated HTAP systems have been developed, such as SAP Hana [32] , TiDB [34] and MemSQL [13, 29]. In addition, many traditional OLTP and OLAP vendors also claim that their systems support HTAP [38, 39, 44, 52]. In general, HTAP systems can have quite different architectures. A fairly recent survey [50] classifies an HTAP system into the following categories based on its architectural choice:

- **Single engine**. HTAP systems in this category normally have unified HTAP engines, such as SAP Hana [32] and MemSQL [29]. These systems can further be divided into two categories based

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554832

Dr. Jianjun Chen is the corresponding author, jianjun.chen@bytedance.com.

on the data formats they support: *single data format* or *mixed data formats*.

- **Separate engine**. HTAP systems in this category use separate query engines to process OLTP and OLAP workloads, such as WildFire [23] and TiDB [34]. These systems can further be divided into two categories based on their storage structure: *separate storage* or *shared storage*. Even though the former is widely adopted in production, it has a shortcoming that data freshness is usually low for OLAP queries.

After a careful study of the different design choices listed in [50], we decide to go with the *separate engine* and *shared storage* design due to the following reasons:

- **Separate engine**. Developing a single query engine that can handle both OLTP and OLAP workloads is non-trivial. Few existing open source systems can handle such mixed workloads well. In contrast, there are multiple standalone OLTP and OLAP systems, either proprietary or open source, that are available for us to use. Hence, we choose the separate-engine route to allow each engine to do its best, while avoiding interference between OLTP and OLAP workloads. We build ByteHTAP using a proprietary OLTP system, *ByteNDB*, and an open source OLAP engine, *Apache Flink* [5].
- **Shared storage**. ByteDance's infrastructure systems usually adopt a cloud native architecture that separates compute from storage. ByteNDB has an architecture similar to Amazon Aurora [61]. We extend its replication storage to support columnar storage beyond row storage. In this way, data changes are propagated in the storage layer with minimal delay. Our columnar storage also contains an in-memory Delta Store to allow the most recent data changes to be queryable by the OLAP engine.

Note that the architectural choice is mostly transparent to users since we provide a unified SQL API for OLTP and OLAP queries. Queries are automatically routed to the corresponding engines by ByteHTAP's well-designed proxy. Besides the advantages mentioned above, such a modular design is also easy for future extensions. For example, we are currently building a new distributed MPP SQL engine that can easily replace Flink in ByteHTAP in the future. We only need to implement a new connector for the new engine to talk with our storage layer.

In this paper, we want to showcase our journey of building a high data freshness and strong data consistency HTAP system with a small engineering team. We started the design and development at the beginning of 2020 with less than 10 developers and released version 1.0 in the middle of 2021. By now, we already have multiple internal customers using ByteHTAP in production and the number of customers is expected to grow significantly in 2022. We have learned tremendously from this journey and hope that our story can be helpful to readers with similar needs.

In summary, our key contributions are as follows:

- We demonstrate how to build a competitive HTAP system with a *separate engine* and *shared storage* architecture. Our modular system design fully utilizes an existing ByteDance's OLTP system and an open source OLAP system. This design saves a lot of resources and development time, and allows easy future extensions such as replacing the query processing engine with other alternatives.
- ByteHTAP can provide high data freshness with less than one second delay, which brings many new business opportunities to our customers. Customers can also configure different data freshness thresholds based on their business needs.
- ByteHTAP provides strong data consistency through global timestamps across OLTP and OLAP systems, which relieves application developers of handling complex data consistency issues in their systems.
- ByteHTAP's replication storage layer utilizes a unified replication framework to seamlessly build both row and column stores. We also describe some important performance optimizations inside ByteHTAP, such as pushing computations to the storage layer and using delete bitmap to efficiently handle deletes.
- We talk about lessons we learned and our best practices in developing and running the ByteHTAP system in production.

The rest of the paper is organized as follows: Section 2 gives an overview of related work. Section 3 describes the overall architecture of ByteHTAP as well as the implementations of its key components. Section 4 focuses on how ByteHTAP achieves high data freshness. Section 5 describes the strong data consistency in ByteHTAP. Section 6 shows some performance optimizations we did inside ByteHTAP. In Section 7, we provide some empirical measurements of ByteHTAP. Section 8 lists some of the major lessons we learned in production. Finally, Section 9 concludes our work.

2 RELATED WORK

In the past decade, HTAP has been widely discussed in both academia and industry [28, 35, 37, 47, 48, 51, 53, 57]. Many database products on the market claim themselves either HTAP databases or supporting HTAP functionalities [6, 7, 9, 10, 20]. In this section, we firstly present several well known dedicated HTAP systems. Next, we give an overview of HTAP extensions in existing OLTP and OLAP database products. Finally, we discuss some recent work on HTAP.

SAP Hana [32, 41, 46, 60] is an in-memory multi-model database system that supports both OLTP and OLAP. It provides a unified interface for OLTP/OLAP components, including the language interface, the plan tree, the operator framework, and the table interface for storage. It contains a layered in-memory store that supports both row and column formats. One recent work [59] extends Hana's in-memory column store to support both in-memory and on-disk storage. Hana adopts a single-engine architecture that differs from ByteHTAP's separate-engine unified-storage architecture.

MemSQL [29] also adopts a single engine architecture. It is a shared-nothing and memory-optimized distributed HTAP system. It provides an in-memory row store with an on-disk column store. It uses LLVM [40] and lock-free in-memory data structures for fast query execution.

TiDB [34] is built on top of TiKV [34], a distributed row-based store for transactional queries. TiDB enhances TiKV with TiFlash [34], a column-based store for analytical queries. By deploying TiKV and TiFlash on different servers, TiDB shows that it can process transactional and analytical queries on isolated resources. TiDB asynchronously replicates logs from TiKV to TiFlash, transforms row-format data into column format, and provides data consistency among logs. TiDB shares some architectural similarity with ByteHTAP, but has some design differences. For example, TiDB 5.0's

OLAP query engine adopts a massively-parallel-processing (MPP) architecture where computing happens on TiFlash's storage nodes. In contrast, ByteHTAP separates computing from storage to allow great elasticity in both computing and storage layers.

WildFire [22, 23, 43] utilizes Spark [63] as the compute engine and uses a shared storage model. It extends OLTP support with simple DML statements for fast data ingestion. DML statements are committed when writing to sharded logs over SSD is finished. A background grooming process periodically merges logs with data in HDFS. The OLAP queries expressed in SparkSQL [19] can query data in both logs and HDFS. Later, Wiser [21] is developed to provide high availability for WildFire. While sharing some architectural similarities with WildFire, ByteHTAP can support general OLTP workloads with MySQL [8] compatibility.

Traditional relational databases usually store data in a row format and are usually more suitable for OLTP than OLAP. Therefore, many relational databases propose specific solutions (such as column-major data format) to accelerate OLAP workloads. Oracle introduces *Database In-memory Option* [38, 49] (DBIM) as a dual format architecture to support HTAP applications. In DBIM, the row-format data is persisted in permanent storage and the column-format data is purely stored in memory. Transaction consistency is ensured between those two formats. Microsoft SQL Server 2016 [39] enhances column store indices to reinforce the processing of HTAP workloads. IBM Db2 for Linux, UNIX, and Windows utilizes BLU Acceleration [52] as a column store to accelerate Business Intelligence queries. In contrast, our solution uses separate native OLTP and OLAP engines over a unified storage layer, so it has more isolation and flexibility for the OLTP and OLAP components.

Traditional data warehouse vendors [27, 44] are also enhancing their OLTP capabilities to provide better support for HTAP workloads. For example, Greenplum Database [44] uses a resource group model to separate OLAP and OLTP workloads, and processes them with different amounts of resources. The experiments show that it can boost OLTP's performance while keeping OLAP's performance.

NoSQL databases have also explored HTAP. For example, Couchbase Server [24] introduces Couchbase Analytics Service [35] to complement its Query Service to support complex analytical queries. Different from Couchbase Server that focuses on document data, ByteHTAP is developed for relational data.

Google proposes F1 Lightning [62] as an HTAP enhancement for its existing transactional database systems. F1 Lightning consists of three components: an OLTP database, the Lightning component, and a federated query engine (F1 Query [56]). The Lightning component reads data from the OLTP database, and transforms them from the row-major, write-optimized format to a column-major, read-optimized format. F1 Lightning adopts a Change Data Capture [62] mechanism to improve data freshness for the OLAP engine. Different from F1 Lightning, ByteHTAP uses unified storage for OLTP and OLAP engines. ByteHTAP's OLAP engine could read newly-committed data directly from the unified storage. Therefore, ByteHTAP provides strong data consistency and high data freshness for both OLTP and OLAP engines.

Recently, IBM also enhances HTAP capability for IBM Db2 for z/OS (Db2z), and proposes a new hybrid system named IBM Db2 Analytics Accelerator (IDAA) [26]. IDAA adds Db2 Warehouse as a column-store to Db2z and processes OLAP workloads there. Db2 Warehouse maintains a copy of table data of Db2z and synchronizes them with DB2z as per requested. Also, IDAA proposes a new data replication method called Integrated Synchronization [26] to support incremental updates of data. This method has improved data freshness for Db2 Warehouse.

HTAP systems are also widely discussed in academia. BatchDB [45] is a database system designed for HTAP workloads. It adopts primary-secondary replicas for OLTP and OLAP workloads, where the OLTP workloads operate on the primary replica, and the OLAP workloads are executed on the secondary one. The updates on the primary replica will be periodically propagated to the secondary replica to ensure that data is consistent between replicas. Compared to BatchDB, the newly-committed changes from ByteHTAP's OLTP engine are immediately available for its OLAP engine. Therefore, ByteHTAP has a better data freshness.

Raza et. al [54] proposes a system that can dynamically adjust its HTAP architecture to meet different requirements of data freshness and runtime performance. The system has three components: an OLTP engine, an OLAP engine, and a Resource and Data Exchange engine. By adjusting the resource distribution between the OLTP and OLAP engine, [54] can explore a spectrum of HTAP architecture design, ranging from fully co-located OLTP-OLAP engines to fully isolated OLTP-OLAP engines. Different from the single-server-based in-memory store used by [54], ByteHTAP has a scalable, distributed, and persistent storage. Also, ByteHTAP's OLTP and OLAP engines have good resource isolation as they operate on their own resources (such as CPU and memory) independently.

VEGITO [58] is a distributed in-memory HTAP system. It utilizes data backups to enhance data freshness and to improve its runtime performance. Specifically, VEGITO added three new techniques to its backups: a gossip-style log-apply scheme, a block-based multi-version columnar data layout, and a tree-based index. Different from VEGITO, ByteHTAP utilizes log sequence number (LSN) in the storage layer to provide strong data consistency. Also, ByteHTAP uses separate OLTP and OLAP engines to provide a good isolation for HTAP workloads.

3 SYSTEM ARCHITECTURE AND IMPORTANT COMPONENTS

ByteHTAP uses ByteNDB as its OLTP system. It also extends ByteNDB's Replication Framework to bridge its OLTP row store and OLAP columnar store. In this section, we firstly give a brief overview of ByteNDB and its Replication Framework. Then, we talk about ByteHTAP's architecture and some important components.

3.1 ByteNDB Overview

Figure 1 shows the overall architecture of ByteNDB. It supports the "log is database" principle as Amazon Aurora [61] does, and can have multiple read replicas besides a read/write master instance. ByteNDB adopts the buffer pool, the transaction and lock management components from MySQL [8], and makes some modifications to achieve the master-replica synchronization. The main components in the computing layer primarily consist of a proxy and a SQL engine. The proxy knows the system configuration and the routings of queries between the master and read-only replicas.

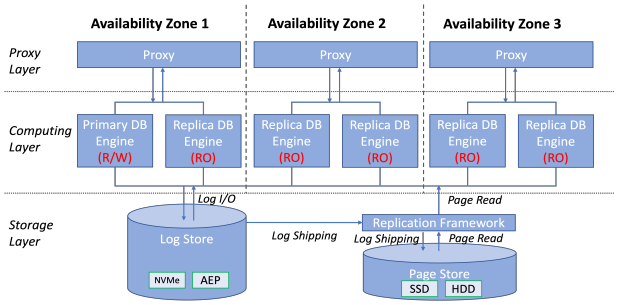

Figure 1: An illustration of ByteNDB architecture.

ByteNDB's replicated storage layer consists of a Log Store that persists redo logs and a Page Store that stores versions of data pages and continuously applies the redo logs to construct the latest version of data pages. Both stores are built on distributed storage to provide high availability and persistence. The Log Store utilizes append-only distributed BLOB storage to provide fast redo log persistence with large capacity. The Page Store provides the capability of *log apply* to construct data pages and supports random read at page granularity.

At the core of ByteNDB's replicated storage layer, a Replication Framework is adapted for the replication and distribution of the redo logs. Each redo log will be assigned a unique LSN based on their persistence order. Redo logs in the same transaction are replicated as a whole batch to ensure atomicity. Each log from Log Store is replicated to three storage servers in Page Store and a quorum protocol [33] is used to guarantee data consistency across these replicas. To further ensure data consistency, the redo logs received by the storage servers will be sorted by their LSNs and be persisted in sequence. Each log also contains a back-link with the LSN of the previous log, any potential holes (i.e. missing logs) in the log sequence can be detected immediately. A gossip protocol [31] is implemented between the storage servers, so that the missing logs on one server can be retrieved from fellow storage servers. As a result, logs are sorted as a sequence without any holes before they are ready to be applied/replayed in the storage servers.

To improve read efficiency, Replication Framework keeps track of the LSNs of each replica so that a query can be sent to a single replica whose LSN is larger than the query's LSN instead of reading from two replicas required by the quorum protocol.

3.2 System Overview

As shown in Figure 2, ByteHTAP adopts an architecture of *separate engines* over a *shared storage*. It supports one unified API and queries can be automatically directed to OLTP or OLAP engines by the proxy. ByteHTAP utilizes ByteNDB as the OLTP engine and Flink as the OLAP engine. ByteHTAP employs a smart proxy layer to automatically direct different queries to the OLTP and OLAP engines. In short, DMLs, DDLs, simple queries and queries suitable for OLTP (e.g. with predicates over indexes on OLTP tables) are sent to the OLTP engine, while complex queries, such as those with multiple joins and aggregations, are sent to the OLAP engine. This approach avoids interference between OLTP and OLAP workloads and allows queries to be processed by an appropriate engine.

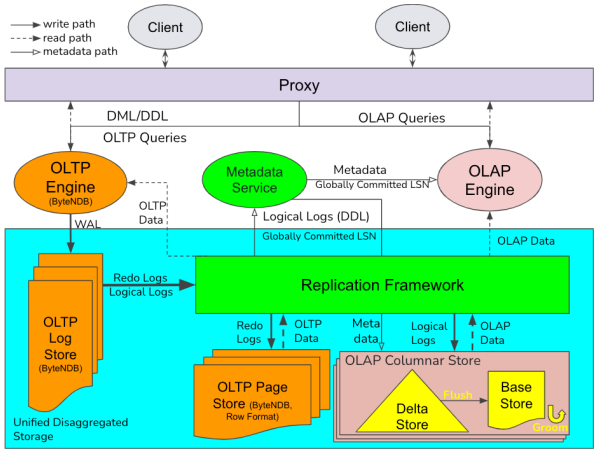

Figure 2: An illustration of ByteHTAP architecture.

ByteHTAP follows ANSI SQL [4] standard, familiar to most database users. One specific requirement is that each ByteHTAP table must contain a primary key, based on which column store data files are sorted to provide efficient data access. Users can issue DMLs to update primary key values in ByteHTAP with data constraints automatically enforced by the OLTP engine. In addition, users can also specify partition keys for the OLAP column store, which enables ByteHTAP's OLAP query engine to process a query in parallel. Currently, ByteHTAP's column store only supports hash partitioning and we will support more partitioning schemes in the future.

ByteHTAP extends ByteNDB's Replication Framework that provides a reliable log distribution to multiple storage nodes for each partition to build a columnar data store, which may reside on different storage nodes from its corresponding row store. Logical logs (i.e. MySQL Binary logs) for committed DML transactions are continuously dispatched to columnar storage nodes based on partitioning scheme defined in user tables. ByteHTAP's Columnar Store consists of an in-memory Delta Store and a durable Base Store. OLAP query scans both Base Store and Delta Store with a specified LSN as its snapshot version. The Metadata Service provides centralized metadata access for OLAP query optimizer and storage nodes. It loads metadata at starting up and caches them in memory.

ByteHTAP guarantees strong data consistency by providing consistent data snapshots for its queries. Each DML and DDL statement generates a unique LSN in the system. Statements in the same transactions are wrapped together and passed through the system atomically. Metadata service relays a globally committed LSN to the OLAP compute engine. Any LSN before this LSN is guaranteed to be received (and persisted) by the OLAP Columnar Store. Metadata server clients in OLAP query engine can periodically fetch the latest globally committed LSN from the storage layer and cache it. A query will be assigned with a read LSN based on the globally committed LSN. In most cases, ByteHTAP allows querying over data changes with less than one second delay.

Note that ByteHTAP currently does not allow transactions with mixed DMLs and OLAP read-only queries since it does not support distributed transactions across OLTP and OLAP engines. We may relax this restriction in the future if customers require this feature.

3.3 Metadata Service

To provide a unified service aligning the catalog information and partition scheme across both OLTP and OLAP engines and reduce the state information in other ByteHTAP modules, a centralized Metadata Service (MDS) is implemented, as shown in Figure 2. It also provides the globally committed LSN from the Replication Framework as the read LSN for OLAP queries.

In ByteHTAP, there are three different kinds of metadata that we need to consider: catalog information, partition information for OLTP and OLAP engines, and statistic information for OLTP and OLAP data. ByteHTAP's OLTP system manages its metadata in its own data store. MDS stores other metadata information, such as OLAP storage partitioning schema and statistics for the partitioned columnar store.

The MDS server is a dedicated process built on top of a Zookeeper [36] cluster for high availability. MDS clients are integrated into the OLAP compute engine and storage servers, in order to communicate with the MDS server for multi-version metadata information. MDS is integrated with a DDL parser that handles most MySQL DDLs. DDL logical logs, which carry metadata changes, are generated on the OLTP side, relayed by the Replication Framework, and parsed by MDS upon arrival. The resultant catalog/partition scheme changes are deserialized, persisted, and made available for service. More details about DDL handling are discussed in Section 5.3.

3.4 OLAP Engine

To support complex analytical queries, we leverage Apache Flink [5] as our OLAP compute engine. We evaluated several widely-used open source compute systems including Flink [5], Presto [11] and Apache Spark [63]. While we saw similar TPC-H [17] and TPC-DS [16] performance in these engines, we decide to use Flink since it is widely used within our company, and it can provide a good support for streaming quires in the future. By combining Flink with our own Columnar Store, we construct an OLAP system for ByteHTAP in a short development time, and our OLAP system can directly inherit the functionalities and benefits from Flink.

To enable Flink to read from our store efficiently, we built a custom connector that supports reading from the Columnar Store in parallel. In addition, ByteHTAP supports a wide range of computation pushdown including but not limited to selection predicates and aggregations. To reduce the read load to our store, we actively apply partition pruning logic during the query compilation phase. The query optimizer determines if there are any predicates on partition columns that match our partition scheme so that it can remove irrelevant partitions. We also made a few other important improvements to Flink's core engine to better serve our query patterns (details are discussed in Section 6).

One of our major contributions is extending ByteNDB's replicated storage to support both row and column format data with high data freshness and strong data consistency. In the next section, we will describe ByteHTAP's Columnar Store in detail.

4 SHARED STORAGE WITH HIGH DATA FRESHNESS

In this section, we describe the columnar store of ByteHTAP, which consists of *Delta Store* and *Base Store*. The Delta Store is a distributed in-memory and row-format store. It can apply logs in low latency and provides high fresh data to the OLAP engine for querying. The Base Store is a distributed and persistent columnar store. Delta Store and Base Store plus Log Store and Page Store on the OLTP side are governed under the unified Replication Framework. They form the shared storage layer of ByteHTAP. We adopt several techniques to ensure high data freshness in the storage.

4.1 Delta Store

In ByteHTAP, an OLAP table can be partitioned with each partition having three replicas to ensure high availability. Therefore, we maintain a Delta Store for each partition replica of a table. A Delta Store consists of two lists: *insertion list* and *deletion list*, which record insert and delete operations, respectively, in the order of the LSN of rows. In ByteHTAP, an update operation is transformed to a delete operation followed by an insert operation with the same LSN in logic logs.

In ByteHTAP, a scan over both Delta Store and Base Store needs to check if a scanning row has been deleted, i.e., if that row is in the deletion list of Delta Store. Therefore, we maintain an additional delete hash map to record all deletions in a Delta Store to accelerate the lookup of the deletion list.

There are four major operations in a Delta Store: *LogApply*, *Flush*, *Garbage Collection*, and *Scan*. Data structures and algorithms in Delta Store are carefully designed so that these operations can run in parallel with high data freshness and strong data consistency. We will discuss how strong data consistency is achieved in Section 5.

LogApply. As shown in Figure 2, Replication Framework distributes logical logs to Delta Store based on the partition key of a table. These logical logs are ordered by their LSNs. Then, the LogApply operation appends each insert log and delete log entry to the insertion list and deletion list, respectively. A delete log is also inserted to the deletion hash map, where the key of the map is the primary key stored in the log.

Flush. Flush is a background task in Delta Store that periodically transfers accumulated row-format data to column-format, and stores them in corresponding durable Base Store located on the same storage node. The process of a Flush can be described as follows. First, we select an LSN as the end point of this Flush based on a pre-defined threshold in terms of either data block size or number of rows. The end point of the previous Flush and this new end point form a flush range. Second, for all rows in the flush range of the insertion list, we sort them based on their primary keys, and convert them from row format to column format. Third, we write the column data as a data block to the Base Store. Note that we need to exclude the rows that are deleted in the flush range by checking the deletion hash map in the Delta Store in the above process. Lastly, we handle deletes in the deletion list by updating the delete bitmap of data blocks in the Base Store. Details of the delete bitmap will be described in Section 4.2.

Garbage Collection (GC). Garbage Collection is a background task that periodically checks Delta Stores in a storage node. If the flushed data in a Delta Store reaches a threshold and no active scans need them, the GC task can truncate these flushed data from the Delta Store to release memory.

Scan. An OLAP query will scan both the Delta Store and Base Store, and union the results together. In ByteHTAP, we provide snapshot read with a read LSN for each query to achieve strong data consistency.

As Delta Store currently does not support spilling data to disk, we need to ensure that it utilizes memory efficiently, and never hits an out-of-memory error in the case of a high rate of data changes. ByteHTAP has a *workload management* module to monitor the resource utilization, and manage the resource usage to prevent severe performance issues in the system. For example, when memory utilization is high, urgent Flushes are triggered to get the memory utilization back to a normal state. Due to space constraints, we skip the details of workload management in this paper.

4.2 Base Store

Base Store is a persistent column store that is created for each partition replica. It is co-located on the storage node with an associated in-memory Delta Store. We explicitly made the decision that does not store the LSN for each data record in the Base Store to reduce storage overhead and to improve update and scan efficiency. The disadvantage of this decision is that we cannot support reading snapshot versions older than the oldest version in the Delta Store, which is not a problem in our current use cases.

Base Store data is stored in a Partitioned Attributes Across (PAX)-like [18] format in the local file system of a storage node. PAX-like format is used in many open source OLAP systems such as Kudu [42]. Each Base Store contains many data blocks, and each block is a collection of rows with a default size of 32MB. Data within each block is ordered by the table's primary key. In each data block, we persist both block-level metadata and the encoded data for each column. A block's metadata includes the number of rows, key range, bloom filter for primary keys, and per-column statistics like min/max, and they will be used during a read operation to trim the data in advance. Currently, we only support value-based index on primary keys, but we will support secondary indices in the future.

Flush operation from Delta Store will generate new Base Store blocks. In order to support data changes, one important design decision is that we use a delete bitmap to track deletes over a data block. To efficiently apply Delta Store deletes on immutable data blocks, we leverage RocksDB [12] to store delete bitmaps for rows removed from the Base Store. Each data block's delete bitmap is stored inside RocksDB as a single key/value pair, with block id as key, and the bitmap in bytes as value, since RocksDB provides a reliable and fast lookup/update store in addition to the append-only Base Store.

Groom. Base Store implements a groom mechanism that includes *Compaction* and *Garbage Collection (GC)* to reduce disk usage and improve query performance. Since a delete operation of Base Store only marks bits corresponding to the deleted rows in the delete bitmap but does not remove them from its data block, the disk usage of Base Store will continue to grow without a grooming process. In addition, the data blocks of a partition generated by subsequent Flushes from its Delta Store may overlap in primary key range. When a query scans Base Store for a given key range, it may have to scan multiple data blocks with overlapping key ranges, which will greatly reduce query efficiency.

Compaction. When the ByteHTAP is running, a background thread periodically measures the percentage of deletes in data blocks and the extent of overlapping of primary keys among different data blocks to select data blocks for Compaction. Specifically, data blocks with a high deletion rate or a high overlapping rate will be prioritized for Compaction. The background thread aggregates the data rows of two or more selected data blocks and writes the aggregation result into new data blocks. The new data blocks do not contain deleted rows and minimize overlap in the key range. After new blocks are generated, the background thread atomically updates the metadata of new blocks accordingly. At the end of Compaction, the background thread also inserts the old data blocks into a GC list. Data blocks stored in the GC list will be recycled during the asynchronous GC.

Garbage collection. A background thread periodically checks the data blocks stored in the GC list and reclaims the storage space by permanently removing them if there are no active queries still accessing them. Details will be described in Section 5.

4.3 High Data Freshness

We define data freshness as how long the recently changed data (insert/update/delete) from the OLTP (TP) system can be visible to an OLAP (AP) query. It is critical to query recently changed data in an HTAP system, especially for real-time analytical workloads. In ByteHTAP, a committed log in the TP system can be read by an AP query in less than one second in most cases. We achieve this high data freshness by adopting the following innovations:

Efficient Log Replication. As Figure 2 shows, Replication Framework will replicate the transactional logs to TP and AP data stores in the unified storage layer. As described in Section 3, the Quorum-based voting protocol is adopted in log distribution to achieve good performance and high availability. During the log distribution, if there are holes due to the replica inconsistency, we apply the gossip protocol using pull-and-push synchronization mode to fill the holes. As such, the delay on the log replication is typically low.

Fast LogApply with Efficient Delete Handling. In ByteHTAP, an update operation is replaced by a delete and an insert operation. ByteHTAP uses a soft delete approach to achieve fast LogApply and Flush, which contributes to high data freshness, by pushing delete handling to the time of query scans. During the LogApply, delete logs are inserted into the delete list and delete hash map, both at a constant cost in the Delta Store. With the use of delete bitmap, Delta Store Flushes do not need to change existing data blocks. In addition, our TP system guarantees that the logical logs are valid, which saves the cost of validation on the insert/delete logs during the LogApply. For example, LogApply does not need to validate that an insert may contain duplicate primary keys with existing data in the Base Store since such a primary key violation would have been caught when the DML statement is executed in the OLTP system.

Efficient Memory Management Using a Vector of Arenas. An efficient memory management using a vector of arenas improves the performance of LogApply and Delta Store GC. There are lots of memory allocations during LogApply and memory releases during Delta Store GC. The cost of these operations is important to the

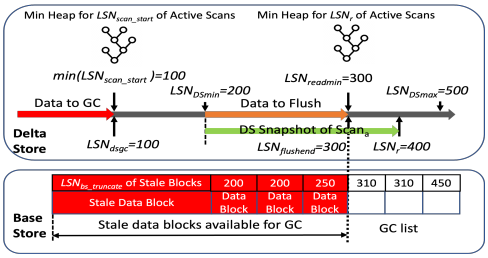

Figure 3: LSNs kept in Delta Store and Base Store for Data Consistency

high data freshness. To efficiently manage the in-memory Delta Store, we introduce the vector of arenas for the memory management of Delta Store. Each Delta Store initially has one arena with configurable starting size. A new arena will be allocated with doubled size (capped on 1MB) and added to the vector if the Delta Store uses up the memory in the current arena. To efficiently allocate and release arenas, we limit the arena size to one megabyte. Each arena is associated with an LSN that is the LSN of the last row saved in this arena. During the Delta Store GC, we release those arenas whose LSN is smaller than the GC LSN.

5 LSN-BASED STRONG DATA CONSISTENCY

As mentioned in Section 3, ByteHTAP guarantees strong data consistency with an LSN-based versioning mechanism. At a high level, our OLAP storage can provide consistent snapshot reads. Delta Store keeps a limited history of recent versions to support concurrent queries. On the other hand, Base Store only keeps a single version that is the maximum data version generated by Delta Store's most recently Flush. Each query carries a read LSN obtained from Metadata Service that it can use to get its read snapshot. Also, the data operations on OLAP side are carefully coordinated through various LSNs as there can be many concurrent operations in our shared storage. In addition, not keeping record-level versions in Base Store also brings additional challenges for supporting snapshots in ByteHTAP. In this section, we describe our algorithms for supporting consistent data operations in detail.

5.1 Important LSNs in ByteHTAP

As the essential building blocks of the versioning mechanism, several LSNs are maintained in a Delta Store as shown in Figure 3, including:

- LSN_{DSmin}: The lowest LSN for the active data entries in a Delta Store that have not been flushed to the Base Store.
- LSN_{DSmax}: The highest LSN of the active data entries in a Delta Store.
- LSN_r: The read LSN of a scan. It is also the upper bound of a scan snapshot.
- $LSN_{readmin}$: Smallest LSN_r of all active scans. We use a min heap to track the read LSNs of all active scans.
- LSN_{scan_start}: The lower bound of a scan's snapshot, which is the LSN_{DSmin} when a scan arrives at the Delta Store. We also maintain a min heap for LSN_{scan_start} for all active scans in the system.
- $LSN_{flushend}$: The largest LSN of a Delta Store's data entries in the next Flush. Once a Flush succeeds, logically speaking, the Base Store's version is upgraded to the value of that Flush's $LSN_{flushend}$.
- LSN_{dsgc}: Delta Store's data entries with LSNs less than this LSN will be truncated by the next Delta Store GC.
- $LSN_{bs\_truncate}$: An LSN-based version assigned to the stale data block at the end of Base Store Compaction, depicting the Delta Store's state at that time and is used for Base Store GC.

These LSNs, along with conventional concurrency control techniques, ensure snapshot read consistency for concurrent scans on a data partition, even when multiple foreground and background data operations are conducted concurrently within the same partition.

5.2 Query and DML Handling

In this section, we explain how snapshot read consistency is guaranteed by examining the interactions between a scan and other data operations on a data partition.

Scan. From the whole system's perspective, an OLAP scan on a table is distributed as one scan task per partition, using the same read LSN (i.e. LSN_r) across all tasks. As for a single partition, LSNs of all the unflushed data in the Delta Store of a partition form an LSN window $[LSN_{DSmin}, LSN_{DSmax}]$. There are three possible situations:

- $LSN_r > LSN_{DSmax}$: Not all visible logs of the scan have been applied in the Delta Store, and Replication Framework will put this scan on hold until the data is available.
- $LSN_r < LSN_{DSmin}$: The snapshot is no longer valid in this case and an error will be returned. The OLAP compute engine will fetch updated LSNs from Metadata Service and will retry the query with an up-to-date LSN_r.
- $LSN_{DSmin} <= LSN_r <= LSN_{DSmax}$: This is the most common case and the visibility of data includes all inserts and deletes in $[LSN_{DSmin}, LSN_r]$ in the Delta Store, plus all the data in the Base Store. Since Base Store does not maintain LSNs for data records, when ByteHTAP generates the execution plan, it takes a snapshot by fetching all the relevant persisted data blocks' names and making a copy of the current delete bitmaps over those blocks.

In addition, we need to make sure that other concurrent operations on a partition will not affect existing snapshots of active scans.

LogApply. LogApply is a foreground operation that appends a batch of inserts and deletes to the end of a Delta Store's lists. It will only increase LSN_{DSmax}, but will not affect the scan snapshots.

Flush. Flush is a background operation that persists Delta Store data in the window $[LSN_{DSmin}, LSN_{flushend}]$ to the Base Store in columnar format. Normally, we choose current $LSN_{readmin}$ as the $LSN_{flushend}$ at the beginning of a new Flush to prevent later arrived scans suffering from invalid snapshots (i.e. $LSN_r < LSN_{DSmin}$). At the end of the Flush, it will atomically add new data blocks into the Base Store, updates delete bitmaps of existing data blocks, persists $LSN_{flushend}$ and updates the new LSN_{DSmin} to the LSN of the oldest active data entry accordingly (setting to $LSN_{flushend} + 1$ if no data is left in Delta Store). The above procedures in Scan and Flush are synchronized by appropriate locking for concurrency control. ByteHTAP users can configure the Flush frequency to

control how much data to be preserved in the Delta Store. The more data kept in the Delta Store, the larger range of valid snapshot windows ByteHTAP can support, with the cost of more memory consumption. The persisted $LSN_{flushend}$ also marks the latest data that has been persisted, which is used as the starting point for the Replication Framework to synchronize logical logs in the case of failure recovery. One potential issue is that a long running query may block Delta Store Flush for a long period of time. Currently, a long running query will be automatically killed after it is over a pre-defined time threshold in ByteHTAP.

Delta Store GC. Delta Store GC is a background operation to remove already-flushed data (i.e. $LSN < LSN_{DSmin}$) on which there are no active scans. LSN_{dsgc} is set to $min(min(LSN_{scan\_start}), LSN_{DSmin})$ when the GC starts.

Base Store Compaction. A Base Store Compaction operation makes an atomic switch between metadata of new and stale data blocks upon its completion. During the atomic metadata switch, the largest read LSN of all the *currently* active scans in Delta Store will be recorded as the $LSN_{bs\_truncate}$ of the stale data blocks, in order to make sure that no active scans are accessing the blocks when they get GC-ed. The stale blocks will be appended to the Base Store GC list along with their $LSN_{bs\_truncate}$, as shown in the Base Store part of figure 3.

Base Store GC. To prevent deleting on-disk data which is being accessed by an active scan, the Base Store GC thread compares the current $LSN_{readmin}$ with the $LSN_{bs\_truncate}$ of each staled block. Only data blocks with $LSN_{bs\_truncate} < LSN_{readmin}$ are guaranteed not serving any active scan, thus can be safely GC-ed.

We can use an example to illustrate the usages of the LSNs discussed above. Figure 3 illustrates a specific time point in a partition. At the time point, the LSN_{DSmin} and LSN_{DSmax} in Delta Store are 200 and 500, respectively. Assume $Scan_a$ with LSN_r=400 just arrives, while a Flush, a Delta Store GC and a Base Store GC are scheduled to happen concurrently. Assume there is an ongoing scan that arrived earlier than $Scan_a$, i.e. a $Scan_b$ with a LSN_r=300, which takes its snapshot when LSN_{DSmin} is 100. Thus, the *current* $min(LSN_{scan\_start})$ is 100, and $LSN_{readmin}$ is 300.

$LSN_{flushend}$ of the concurrent Flush is set to $LSN_{readmin}$, thus the concurrent Flush will flush data entries in *[200, 300]* to Base Store. At the end of the Flush, the new LSN_{DSmin} will be set to the LSN of the next data entry, say, 301. Concurrently, $Scan_a$ will atomically take a snapshot of Delta Store and Base Store data blocks (not shown) with LSN_{scan_start} and LSN_r inserted into the two min heaps respectively. So if $Scan_a$ takes its snapshot before the end of the Flush, it will have a snapshot window of *[200, 400]* with the old metadata, Base Store data blocks and delete bitmap, otherwise it will have a snapshot window of *[301, 400]* with the newly Flush-modified metadata, data blocks and delete bitmap.

As for the Delta Store GC, LSN_{dsgc} is set to 100. An $LSN_{bs\_truncate}$ value of 250 means that when its corresponding data blocks are turned stale by a previous Compaction, the largest read LSN of active queries at that time in Delta Store is 250, thus the current oldest active scan $Scan_b$ must come later than that. Thus, this block and other older staled blocks can be safely purged, but not the data blocks with *LSN=310* or newer, as they may be accessed by $Scan_b$.

5.3 DDL Handling

Besides DML and query operations, ByteHTAP also needs to make sure its DDL operations can provide strong data consistency. To achieve this, ByteHTAP keeps multiple versions of database metadata in its Metadata Service (MDS), which is the source of truth for metadata in ByteHTAP.

When a DDL statement arrives in ByteHTAP, it will first be processed by the OLTP engine in a MySQL-compatible fashion. Next, the Replication Framework is responsible to send the DDL logical log generated for the DDL operation to MDS and the AP storage servers. MDS will parse the DDL logical log to generate a new version of metadata, and the LSN of the logical log is served as the metadata version number. MDS will persist and then make this new version of metadata available for AP queries, along with all historical versions of metadata.

AP storage servers are integrated with an MDS client, which is capable of periodically pulling the newly available metadata, and caching a copy of a complete history of metadata locally. When a DDL logical log reaches a Columnar Store partition, during the LogApply, the MDS client will fetch the corresponding metadata according to the received DDL LSN from MDS. Currently, we only support DDL changes that do not require data-reorganization, such as ADD, DROP and RENAME COLUMN. More advanced support is under development.

6 OLAP QUERY PERFORMANCE OPTIMIZATION

Besides the high data freshness and strong data consistency, performance is another important metric for ByteHTAP, especially when it needs to handle the real-time analytical workloads. In this section, we discuss our performance optimizations for the storage engine and OLAP query engine, including delete handling, computation pushdown, statistics collection, asynchronous read, and parallelism optimization. We currently do not have an OLAP plan cache so each query is compiled freshly with its own read LSN. We plan to add this feature in the future.

6.1 Delete Optimization for Scans

To support efficient deletes handling, we store delete bitmaps that contain information about rows that are removed from Base Store after the original Flush. We have described the bitmap format in Section 4.2.

During a Base Store scan operation, the scanner firstly takes a snapshot of the block ids to be scanned, then fetches the bitmap based on each block id, and finally, scans the base column data and applies the deletes. In addition to checking deletes from a bitmap during a Base Store scan, we also need to efficiently handle deletes that are still residing in Delta Store. We implemented two methods to achieve this:

- Lazy approach: Since the deletes in Delta Store are stored in a delete hash map within Delta Store, we scan the Base Store first, then for each row in the result, we do a hash lookup to see if it has already been deleted before we return the final result.
- Eager approach: During the Base Store scan initialization stage, deleted keys from Delta Store in the scan's snapshot are retrieved first. For each deleted key, we leverage our primary key index

in Base Store to do an index lookup and construct a selection vector before we scan the Base Store data.

Each approach has its advantages and disadvantages. The lazy approach is similar to a hash join where we probe the base data, and then for each row we do a hash lookup using the Delete Hashmap in Delta Store, and return the joined result. Note that this approach always requires primary key columns being selected even if it is not required by the query. The eager approach, on the other hand, is similar to an index-based nested loop join where we read the deletes from the Deletion List in Delta Store first, then for each delete, we do index lookup on the Base Store table, and return the joined result. The advantage of the eager approach is that it can be very efficient when there are not many deletes in Delta Store and a large amount of data resides in Base Store. On the other hand, the eager approach is less efficient when Delta Store contains a large number of deletes that affect many data blocks in Base Store. To produce the optimal execution plan, we have built a cost model to intelligently determine the best approach to use based on the actual statistics. The statistics are collected on the fly from Delta Store as well as Base Store during each Flush. The experimental results (given in Section 7.4) show that our cost model can select the proper approach and yield the best performance.

6.2 Computation Pushdown to Storage Engine

As Figure 2 shows, ByteHTAP decouples storage and compute. Therefore, we design and implement *predicate pushdown* and *aggregate pushdown* to reduce data transferring from the storage engine to the query engine.

Predicate Pushdown. Query planner decides if a predicate of a scan operator can be pushed down to the storage engine by checking whether the predicate can be evaluated by the storage engine and its corresponding cost savings. With predicate pushdown, scanned data will be filtered by the predicate at the storage layer and only results satisfying the predicate will be sent back to the query engine for further processing. Base Store maintains min/max column values for each column of a data block as metadata, which is computed during Delta Store Flush or Base Store Compaction. If the predicate evaluation based on the min/max values can filter out a data block, we do not need to load the data from disk for that data block at all. In addition, we also perform lazy materialization on the Base Store data by evaluating the columns with predicates first. In the case that the evaluation of predicates can filter out all rows in a data block, we can skip that data block to avoid reading unneeded columns.

Aggregate Pushdown. In a typical AP workload, aggregate operations are widely used over a large volume of data. Usually, an aggregate operation can be split into partial aggregates (local aggregates) and final aggregates. The AP query optimizer can consider pushing down qualified partial aggregates to the storage engine, e.g. when there is only a table scan under the aggregate and all filtering predicates (if exist) of the scan can also be pushed down to the storage engine. In each partition, storage engine will aggregate the scanned data after applying filtering predicates (if exist) and returns the partially aggregated results to the compute engine. Then, the compute engine will conduct a final aggregate over the partial results.

6.3 OLAP Query Engine Optimization

Since we leverage Flink to support complex analytical queries, we have made a few improvements to Flink's core engine to improve its OLAP capability. As our current customers' OLAP queries are time-sensitive and require a high queries-per-second (QPS), our optimizations focus on reducing latency in query optimization and query execution, and improving the overall system throughput. Below are some improvements we have done.

Statistics Collection. We use Flink to compute statistics of tables, and store the results in ByteHTAP's Metadata Service (MDS). During query compilation, our Flink connector will fetch those statistics, either from local cache or MDS, to help Flink's cost-based optimization. For simple queries, we also provide a fast path to bypass the full cost-based optimization. To provide queries with up-to-date statistics, we also collect the incremental statistics from Flush and Compaction operations, and store them in MDS. Those additional statistics may trigger Flink to re-compute full statistics when changes seem significant enough.

Asynchronous Read. By default, a Flink connector is a single thread repetitively reading and processing data segments until it finishes processing the entire data set. As a sequential process, it takes significant I/O wait time during the reading and processing of data, making the whole process less efficient. We split the connector into two separate threads, one for reading from columnar store and the other for processing the data and passing them to other operators. They communicate through an adjustable buffer. With this approach, we see significant improvement in I/O throughput and TPC-DS total running time reduces by 10%.

Parallelism Optimization. Flink uses pre-configured task parallelism that does not consider the size of data and the scale of the job. This design wastes task container resources for simple queries, and increases end-to-end delay for complex queries that need larger parallelism. We added optimizer rules to adjust source scan parallelism based on the data statistics and the number of partitions. In addition, operators above scan can adjust their parallelism according to the source parallelism. This work improves our cluster QPS by 20% on TPC-DS workloads.

7 PERFORMANCE STUDY

In this section, we describe our system's evaluation through a set of experiments. First, we present experiments conducted for measuring our system's HTAP capability. We measure the performance interference between OLTP and OLAP engines using the benchmark, CH-benCHmark [30], and measure data freshness using Sysbench [14]. Then, we evaluate our system's OLAP compatibility and performance using TPC-DS [16]. Lastly, we present the results of OLAP query performance optimization discussed in Section 6. All of the experiments are conducted on a cluster of seven machines. The machine and cluster's setup are listed in Table 1. Note that some ByteHTAP components are co-located on the same machine for saving resources.

7.1 Hybrid OLTP and OLAP Workload

An HTAP system's performance on mixed OLTP/OLAP workloads is critical. Therefore, we run CH-benCHmark [30] on ByteHTAP to test its performance on mixed workloads. CH-benCHmark is a

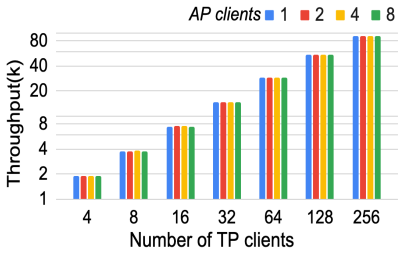

Figure 4: Throughput for OLTP

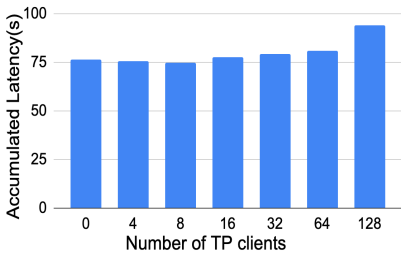

Figure 5: CH-benCHmark queries latency

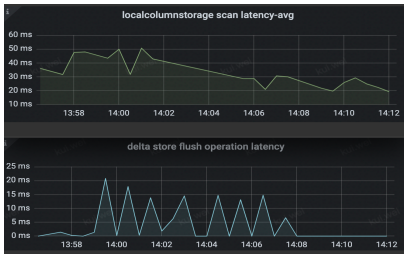

Figure 6: Operation latency over time in CH-benCHmark

Table 1: Machine and cluster setup.

CPU	Intel(R) Xeon(R) Gold 5218 (2 NUMA Nodes, 32 cores)
Memory	378GB
Cache	22MB shared L3 cache
OS	Debian 4.14.81
Network	25Gbps Ethernet
Page Store & Log Store	3 servers
Columnar Store & OLAP engine	3 servers
OLTP engine & Metadata Service	1 server

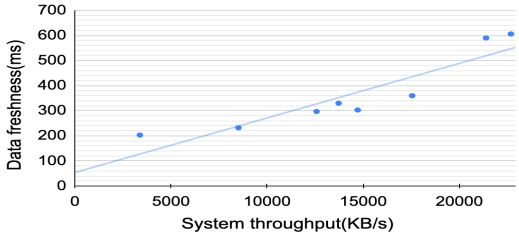

Figure 7: Data freshness with different throughputs

standard HTAP benchmark, which bridges the gap between the standard TPC-C benchmark for OLTP and TPC-H benchmark for OLAP. CH-benCHmark is built on the unmodified version of the TPC-C benchmark, and its OLAP part contains 22 analytical queries inherited from TPC-H. It enables running both OLTP and OLAP queries on a set of shared tables in one database. Our experiments are conducted based on CH-benCHmark with 100 warehouses data set [30]. The data is firstly loaded into ByteHTAP's OLTP data store and is then replicated to the OLAP columnar store.

Figure 4 shows the throughput of the TP engine with a different number of TP and AP clients. The throughput is measured as transactions-per-minute (tpmC [15]). As the figure shows, the throughput of TP engine increases almost linearly as the number of TP clients increases. For a fixed number of TP clients, the TP throughput is almost the same with different number of AP clients. The results show that ByteHTAP's OLTP performance is barely impacted by the OLAP workloads due to its separate-engine design.

Next, we look at the AP performance on CH-benCHmark, where the accumulated query latency of the 22 analytical queries is used as the performance metric. When we keep updating the data from the TP side through multiple TPC-C transactional queries, binary logs are replicated to the Delta Store on the AP side that triggers Flush operations. As Figure 5 shows, when increasing the TP clients from 0 to 64, there is only about 5% performance reduction. This is because that most of the flush latencies are less than 20ms and the scan latencies remain low during the flush operations, as shown in Figure 6.

However, when the number of TP clients is increased to 128 in Figure 5, the accumulated query latency shows an obvious increase. This is because in this case the log replication between TP and AP becomes a bottleneck, which could introduce 2s to 3s delay in log replication. When this happens, if an AP query wants to fetch the data with latest committed LSN, it needs to wait until the latest logs up to this LSN are replicated and become visible to AP. We could solve this issue through adding more machine resources to avoid overloading the Replication Framework. So far this rarely happens in our production scenarios.

In summary, ByteHTAP's AP performance is stable in most cases, even during the execution of Flush and Compaction operations.

7.2 High Data Freshness Experiment

Data freshness is an important metric to HTAP systems. However, we cannot find a common benchmark for measuring it. Some previous work adopts *freshness-rate metric* [25, 55]. However, we decide to use *freshness-time metric*, that is, how fast the data modification on the OLTP side becomes available to queries on the OLAP side. From our perspective, this metric is more appropriate for our users as it matches well with their requirements.

As described before, the logical logs of a transaction will be synced from TP store to the AP storage servers, and become visible to AP queries after being applied to Delta Store. Thus, the total time of this process can be used as our freshness-time metric T_f. According to our design goal, T_f should be less than one second for a typical workload.

We use the same experiment setup given in Table 1. We design our data freshness experiment based on the SysBench [14] benchmark and use a hash-partitioned table with 257 partitions. We choose to use 16 concurrent write threads, which represents one of the typical numbers of TP clients observed in the production environment. We vary the transaction sizes by adjusting the number of data operations within one transaction during the experiment. We start from 1 update, 1 delete, and 1 insert, and double

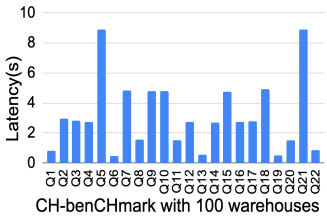

Figure 8: CH-benCHmark OLAP performance

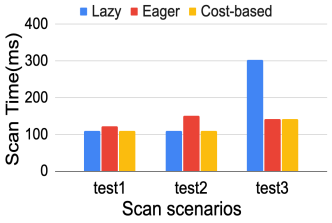

Figure 9: Cost-based mode optimization

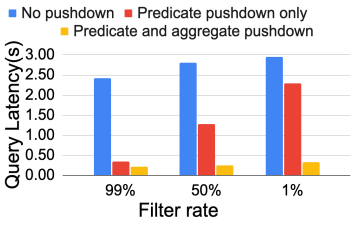

Figure 10: Computation pushdown improvement

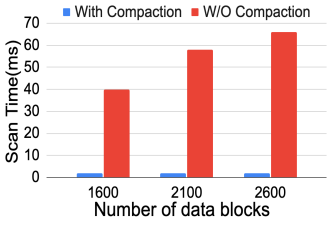

Figure 11: Compaction impact on range scan

each operation subsequently until 128. The throughput is calculated by multiplying the transaction size with transactions-per-second obtained from SysBench.

Figure 7 shows the average freshness-time metrics over 257 partitions with different transaction throughput. As expected, the freshness-time metric generally increases with the increase of total data throughput. Note that with 16 write threads and a relatively high data throughput of 22681 KB/s we still observe a low T_f of 606 ms. This satisfies our design goal. We also conduct experiments with 32 write threads, and observe similar patterns.

7.3 OLAP Compatibility and Performance

In this section, we discuss our experiments for evaluating ByteHTAP's OLAP performance. We firstly test ByteHTAP using TPC-DS [16] benchmark that is a standard benchmark for general-purpose decision support systems. The results show that our system can support all TPC-DS queries. The ability to support complex OLAP queries is very important for users to adopt ByteHTAP. We also evaluated the TPC-DS performance of a well-known open source HTAP system on the market, referred to as HTAP-T [1]. The results show that only 66 out of 103 TPC-DS queries are supported in HTAP-T. Due to space constraints, we omit the benchmark results.

We next discuss ByteHTAP's performance on CH-benCHmark. Currently, ByteHTAP supports all 22 analytical queries, but in HTAP-T, the query 16 has inconsistent results on its OLTP and OLAP engines. Figure 8 shows our system's performance on CH-benCHmark with 100 warehouses. We can see that the latency of most queries is less than 5 seconds, and the accumulated latency of 21 queries is comparable to that of HTAP-T. We also test ByteHTAP's performance on CH-benCHmark with 1000 warehouses with seven more OLAP nodes. All queries finish within 10 seconds. We omit the details due to space constraints.

7.4 Performance Optimization Evaluation

In this section, we conduct multiple experiments to evaluate the performance optimization presented in Section 6.

Scan Optimization. We conduct an experiment to test the three delete-handling modes described in Section 6.1: lazy-mode, eager-mode and cost-based-mode. In the experiment, we create a base case (test1) with 500, 000 rows in Base Store, and half of them are deleted in Delta Store. As shown in Figure 9, when we scan a small number of rows (i.e. 500) in test1, the lazy mode has a slightly better scan time, since the eager mode would pay a relatively high upfront cost to pre-process Delta Store deletes. In test2, we increase the number of deletes to 99% in Delta Store. The result shows that the upfront cost paid by eager mode increases accordingly, while the time in lazy mode does not have a big change. In test3, instead of changing the number of deletes, we increase the number of rows to scan to 50, 000. The scan cost in lazy mode grows more sharply than the eager mode, as it needs to pay a fixed hash-lookup cost for each scanned row.

Figure 9 also shows that our cost-based model can smartly choose between lazy and eager mode for a good scan performance based on collected statistics.

Computation pushdown. To evaluate the effectiveness of computation pushdown from the query engine to the storage engine, we conduct experiments using the TPC-DS [16] benchmark. We use the query with predicate and aggregate on the table, *store_sales*, in TPC-DS:

```
SELECT sum(ss_net_profit) AS profit
FROM store_sales
WHERE ss_ticket_number < C
GROUP BY ss_store_sk
ORDER BY profit DESC;
```

The constant C in the query is chosen with different filter rates: 99%, 50%, and 1%. For each filter rate, we have three modes: *no pushdown*, *predicate pushdown*, and *predicate and aggregate pushdown*.

Figure 10 shows that both predicate pushdown and aggregate pushdown can substantially improve ByteHTAP's performance.

Compaction. Figure 11 shows the scan performance with and without using compaction. In the experiment, we develop unit tests that can precisely generate data blocks overlapping in their primary key ranges. We run the tests on a server with the specification described in Table 1. Three tables are defined using the same schema and are populated with different amounts of data. The total number of data blocks in the three tables are 1500, 2000, and 2500, respectively. We make the data blocks in each table overlap at their primary key ranges. The rows in the key range between 0 and 100 are further deleted and re-inserted 100 times to add additional 100 overlapping data blocks. A range scan is executed on each table to select the data with a primary key range between 0 and 100.

Figure 11 shows that the scan time reduces dramatically when compaction is used, since without compaction all data blocks with overlapping primary keys have to be accessed. In contrast, compaction merges overlapping data blocks and removes data blocks that have been deleted. Therefore, fewer data blocks are scanned, and the scan efficiency is significantly improved.

[1]Due to the restrictive proprietary licensing agreement, we omit the vender's name.

Delta Store vs Base Store Scan Performance. In this experiment, we measure the scan performance on column-format data in Base Store *vs.* row-format data in Delta Store. The test query scans one of the 34 columns in the *web_sales* table of TPC-DS (100GB). Table 2 shows the scan performance with different flush rates: all data in Delta Store flushed to Base Store (100%), half amount of the data flushed (50%), and none of the data flushed (0%). Table 2 shows the scan performance in Base Store is better than Delta Store although data in Delta Store resides in memory, and *Scan Speedups* increases with *Flushed Data (%)* increasing. In the ByteHTAP system, most data is typically flushed to Base Store and stored in the column format, and only the most recent data temporarily resides in Delta Store and is stored in the row format.

Table 2: Scan performance with different flush rates.

Flushed Data (%)	Scan Speedups
100%	2.90
50%	1.78
0%	1.00

8 LESSONS LEARNED FROM PRODUCTION

Ever since ByteHTAP becomes generally available, it has attracted many ByteDance internal customers that have HTAP needs. Before ByteHTAP, many customers' pipelines consist of multiple connected systems. For example, in certain settings, their row data often comes from a large number of MySQL databases and are pre-aggregated by an Extract-Transform-Load (ETL) system. Then, the data gets loaded into a specialized OLAP system for analysis. The whole process imposes a delay of more than $1-2$ hours before analysts can get the reports, which becomes unsuitable when data freshness requirement is under a few minutes. Another drawback is that customers have to maintain multiple systems and ETL pipelines for above processing that incur significant operational overhead. After switching to ByteHTAP, they see a data freshness improvement from hours to less than 1 minute, as well as a great reduction in management overhead.

While ByteHTAP is still at an early stage of product adoption, we have learned some important lessons from our production experience. As more and more customers start using ByteHTAP, we expect to learn more in the future.

Allow ByteNDB customers to painlessly upgrade to ByteHTAP. Many ByteHTAP's potential customers come from existing customers of ByteNDB, which typically have multiple online ByteNDB clusters with several hundred GB of data generated daily. Thanks to the decoupled OLTP/OLAP design of ByteHTAP, migrating these ByteNDB clusters is fairly easy, as OLAP components can be deployed and enabled separately. No data migration or offline time is needed for the existing online OLTP clusters, and the whole process can be transparent to the customer. In general, we think in practice many HTAP use cases come from existing OLTP customers. Because of this observation, we decide to add a single-button upgrade option for ByteNDB customers to upgrade their OLTP databases to ByteHTAP.

Cross OLTP database query ability. Customers often have multiple ByteNDB databases for their OLTP workload, e.g. one for each department. However, when they perform OLAP analysis, they want to be able to query across these databases. After learning this requirement from some early customers, we made simple changes to allow ByteHTAP OLAP tables to sync from multiple databases inside ByteNDB, which makes the customer's life much easier. In general, providing some flexibilities between the mapping of OLTP and OLAP data seems a good choice.

Efficient data import. In the case of migrating a large database instance from traditional OLTP to HTAP systems, we need to create the initial AP store based on the TP database. One naive approach is to simply stream the historical logical log to OLAP columnar store, which is very slow and not practical in practice. We develop a tool to obtain a consistent snapshot from the Page Store, then parse the data pages in the snapshot, directly write them to the Base Store in batches to create the instance data, which significantly reduced the time to migrate large instances.

Flink enhancements. As our users migrate more and more workloads to our ByteHTAP system, we are seeing a high QPS growth and more diverse query types that expose performance bottlenecks of Flink's query engine. Accordingly, we need to modify Flink to address those issues. For example, we improved Flink's plan generation from a single-threaded method to a parallel approach. We also redesigned Flink's core scheduler to remove the heavy role of task manager, which greatly improved the throughput of task scheduling by up to 200%. With all these efforts, we are seeing up to 25% performance improvements on TPC-H queries compared to Flink's open source version. This work has been presented at Flink Forward Asia 2021 [1–3].

9 CONCLUSIONS

ByteHTAP is a large-scale real-time analytics system supporting both fresh data changes and strong data consistency, and is designed to meet ByteDance's growing business demands. In this paper, we demonstrate how a competitive HTAP system with a separate-engine and shared-storage architecture is built. Our modular system design fully utilizes an existing ByteDance's OLTP system and an open source OLAP system. ByteHTAP can provide high data freshness with less than one second delay, which enables many new business opportunities for our customers. Since its launch around the middle of 2021, we have seen more and more internal customers use it to replace their previous systems that are made up of a combination of OLTP databases, OLAP databases, and additional ETL pipelines. ByteHTAP gives our customers real-time insights with consistent data and less operational overhead.

ACKNOWLEDGMENTS

We would like to extend our thanks to the anonymous reviewers for their valuable comments. We heartily thank all people who made contributions to the design and development of the ByteHTAP system: Ron Hu, Jie Zhou, Yupeng Jin, Liwen Shao, Xikai Wang, Shicai Zeng, Xiahao Zhang, Fangwen Su, Xiangrui Meng, Yong Fang, Weihua Hu, Dizhou Cao, Runkang He, Guanghui Zhang. Finally, we thank Vipul Gupta for his careful proofreading of this manuscript.

REFERENCES

[1] 2021. *Flink Forward Asia 2021.* Retrieved February 23, 2022 from https://flink-forward.org.cn/

[2] 2021. *Improvements of Job Scheduler and Query Execution on Flink OLAP.* Retrieved February 23, 2022 from https://www.bilibili.com/video/BV1j34y1B72o?p=7

[3] 2021. *Powering HTAP at ByteDance with Apache Flink.* Retrieved February 23, 2022 from https://www.bilibili.com/video/BV1j34y1B72o?p=3

[4] 2022. *ANSI SQL Standard.* Retrieved February 23, 2022 from https://webstore.ansi.org/Standards/ISO/ISOIEC90752016

[5] 2022. *Apache Flink.* Retrieved February 7, 2022 from https://flink.apache.org

[6] 2022. *BaikalDB.* Retrieved February 7, 2022 from https://github.com/baidu/BaikalDB

[7] 2022. *Microsoft Azure Synapse Analytics.* Retrieved February 23, 2022 from https://azure.microsoft.com/en-us/services/synapse-analytics/

[8] 2022. *MySQL.* Retrieved February 23, 2022 from https://www.mysql.com/

[9] 2022. *OceanBase.* Retrieved February 7, 2022 from https://open.oceanbase.com

[10] 2022. *PolarDB-X.* Retrieved February 7, 2022 from https://www.alibabacloud.com/product/polardb-x

[11] 2022. *Presto.* Retrieved February 23, 2022 from https://prestodb.io

[12] 2022. *RocksDB.* Retrieved February 18, 2022 from http://rocksdb.org/

[13] 2022. *SingleStore.* Retrieved February 7, 2022 from https://www.singlestore.com

[14] 2022. *Sysbench.* Retrieved February 11, 2022 from https://github.com/akopytov/sysbench

[15] 2022. *TPC-C Specification.* Retrieved February 23, 2022 from http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[16] 2022. *TPC-DS.* Retrieved February 11, 2022 from http://www.tpc.org/tpcds/

[17] 2022. *TPC-H.* Retrieved February 11, 2022 from http://www.tpc.org/tpch/

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

[19] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[20] Hillel Avni, Alisher Aliev, Oren Amor, Aharon Avitzur, Ilan Bronshtein, Eli Ginot, Shay Goikhman, Eliezer Levy, Idan Levy, Fuyang Lu, et al. 2020. Industrial-strength OLTP using main memory and many cores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3099–3111.

[21] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, Guy Lohman, C Mohan, Rene Muller, Hamid Pirahesh, Vijayshankar Raman, Richard Sidle, Adam Storm, et al. 2019. Wiser: A highly available HTAP DBMS for iot applications. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 268–277.

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

[23] Ronald Barber, Matt Huras, Guy Lohman, C Mohan, Rene Mueller, Fatma Özcan, Hamid Pirahesh, Vijayshankar Raman, Richard Sidle, Oleg Sidorkin, et al. 2016. Wildfire: Concurrent blazing data ingest and analytics. In *Proceedings of the 2016 International Conference on Management of Data*. 2077–2080.

[24] Dipti Borkar, Ravi Mayuram, Gerald Sangudi, and Michael Carey. 2016. Have your data and query it too: From key-value caching to big data management. In *Proceedings of the 2016 International Conference on Management of Data*. 239–251.

[25] Mokrane Bouzeghoub. 2004. A framework for analysis of data freshness. *Proceedings of the 2004 international workshop on Information quality in information systems*, 59–67.

[26] Dennis Butterstein, Daniel Martin, Knut Stolze, Felix Beier, Jia Zhong, and Lingyun Wang. 2020. Replication at the speed of change: a fast, scalable replication solution for near real-time HTAP processing. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3245–3257.

[27] Le Cai, Jianjun Chen, Jun Chen, Yu Chen, Kuorong Chiang, Marko A. Dimitrijevic, Yonghua Ding, Yu Dong, Ahmad Ghazal, Jacques Hebert, Kamini Jagtiani, Suzhen Lin, Ye Liu, Demai Ni, Chunfeng Pei, Jason Sun, Li Zhang, Mingyi Zhang, and Cheng Zhu. 2018. FusionInsight LibrA: Huawei's Enterprise Cloud Data Analytics Platform. *Proc. VLDB Endow.* 11 (2018), 1822–1834.

[28] Jianjun Chen, Yu Chen, Zhibiao Chen, Ahmad Ghazal, Guoliang Li, Sihao Li, Weijie Ou, Yang Sun, Mingyi Zhang, and Minqi Zhou. 2019. Data management at huawei: Recent accomplishments and future challenges. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 13–24.

[29] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1401–1412.

[30] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. 1–6.

[31] Alexandros G. Dimakis, Soummya Kar, José MF Moura, Michael G. Rabbat, and Anna Scaglione. 2010. Gossip algorithms for distributed signal processing. *Proc. IEEE* 98, 11, 1847–1864.

[32] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *Proceedings of the VLDB Endowment* 40, 4 (2011), 45–51.

[33] David K Gifford. 1979. Weighted voting for replicated data. *Proceedings of the seventh ACM symposium on Operating systems principles*, 150–162.

[34] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[35] Murtadha AI Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Lychagin, Ian Maxon, and Till Westmann. 2019. Couchbase analytics: NoETL for scalable NoSQL data analysis. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2275–2286.

[36] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

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

[38] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

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

[40] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

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

[42] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. 2015. Kudu: Storage for fast analytics on fast data. *Cloudera, inc* 28 (2015).

[43] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *EDBT*. 1–12.

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

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

[46] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. SAP HANA–The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)* (2017).

[47] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, et al. 2015. S-Store: Streaming Meets Transaction Processing. *Proceedings of the VLDB Endowment* 8, 13 (2015).

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

[49] Niloy Mukherjee, Shasank Chavan, Maria Colgan, Dinesh Das, Mike Gleeson, Sanket Hase, Allison Holloway, Hui Jin, Jesse Kamp, Kartik Kulkarni, et al. 2015. Distributed architecture of oracle database in-memory. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1630–1641.

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

[51] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems. In *CIDR*, Vol. 4. 1.

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

[53] Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. 2016. Snappydata: A hybrid transactional analytical store built on spark. In *Proceedings of the 2016 International Conference on Management of Data*. 2153–2156.

[54] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through elastic resource scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2043–2054.

[55] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through elastic resource scheduling. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2043–2054.

[56] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, et al. 2018. F1 query: Declarative querying at scale. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1835–1848.

[57] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F Ilyas. 2021. Real-Time LSM-Trees for HTAP Workloads. *arXiv preprint arXiv:2101.06801* (2021).

[58] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 219–238.

[59] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, et al. 2019. Native store extension for SAP HANA. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2047–2058.

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

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

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

[63] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.