TiDB: A Raft-based HTAP Database

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, Xin Tang

PingCAP

{huang, liuqi, cuiqiu, fangzhuhe, maxiaoyu, xufei, shenli, tl, z, menglong, weiwan, liucong, zhangjian, jay, wuxuelian, songlingyu, sunruoxi, yusp, zhaolei, nick, liquanpei, tangxin}@pingcap.com

ABSTRACT

Hybrid Transactional and Analytical Processing (HTAP) databases require processing transactional and analytical queries in isolation to remove the interference between them. To achieve this, it is necessary to maintain different replicas of data specified for the two types of queries. However, it is challenging to provide a consistent view for distributed replicas within a storage system, where analytical requests can efficiently read consistent and fresh data from transactional workloads at scale and with high availability.

To meet this challenge, we propose extending replicated state machine-based consensus algorithms to provide consistent replicas for HTAP workloads. Based on this novel idea, we present a Raft-based HTAP database: TiDB. In the database, we design a multi-Raft storage system which consists of a row store and a column store. The row store is built based on the Raft algorithm. It is scalable to materialize updates from transactional requests with high availability. In particular, it asynchronously replicates Raft logs to learners which transform row format to column format for tuples, forming a real-time updatable column store. This column store allows analytical queries to efficiently read fresh and consistent data with strong isolation from transactions on the row store. Based on this storage system, we build an SQL engine to process large-scale distributed transactions and expensive analytical queries. The SQL engine optimally accesses row-format and column-format replicas of data. We also include a powerful analysis engine, TiSpark, to help TiDB connect to the Hadoop ecosystem. Comprehensive experiments show that TiDB achieves isolated high performance under CH-benCHmark, a benchmark focusing on HTAP workloads.

PVLDB Reference Format:
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, Xin Tang. TiDB: A Raft-based HTAP Database. *PVLDB*, 13(12): 3072-3084, 2020.
DOI: https://doi.org/10.14778/3415478.3415535

1. INTRODUCTION

Relational database management systems (RDBMS) are popular with their relational model, strong transactional guarantees, and SQL interface. They are widely adopted in traditional applications, like business systems. However, old RDBMSs do not provide scalability and high availability. Therefore, at the beginning of the 2000s [11], internet applications preferred NoSQL systems like Google Bigtable [12] and DynamoDB [36]. NoSQL systems loosen the consistency requirements and provide high scalability and alternative data models, like key-value pairs, graphs, and documents. However, many applications also need strong transactions, data consistency, and an SQL interface, so NewSQL systems appeared. NewSQL systems like CockroachDB [38] and Google Spanner [14] provide the high scalability of NoSQL for Online Transactional Processing (OLTP) read/write workloads and still maintain ACID guarantees for transactions [32]. In addition, SQL-based Online Analytical Processing (OLAP) systems are being developed quickly, like many SQL-on-Hadoop systems [16].

These systems follow the "one size does not fit all" paradigm [37], using different data models and technologies for the different purposes of OLAP and OLTP. However, multiple systems are very expensive to develop, deploy, and maintain. In addition, analyzing the latest version of data in real time is compelling. This has given rise to hybrid OLTP and OLAP (HTAP) systems in industry and academia [30]. HTAP systems should implement scalability, high availability, and transnational consistency like NewSQL systems. Besides, HTAP systems need to efficiently read the latest data to guarantee the throughput and latency for OLTP and OLAP requests under two additional requirements: *freshness* and *isolation*.

Freshness means how recent data is processed by the analytical queries [34]. Analyzing the latest data in real time has great business value. But it is not guaranteed in some HTAP solutions, such as those based on an Extraction-Transformation-Loading (ETL) processing. Through the ETL process, OLTP systems periodically refresh a batch of the latest data to OLAP systems. The ETL costs several hours or days, so it cannot offer real-time analysis. The ETL phase can be replaced by streaming the latest updates to OLAP systems to reduce synchronization time. However, because these two approaches lack a global data governance model, it is more complex to consider consistency semantics. Interfacing with multiple systems introduces additional overhead.

Isolation refers to guaranteeing isolated performance for seperate OLTP and OLAP queries. Some in-memory databases (such as HyPer [18]) enable analytical queries to read the latest version of data from transactional processing on the same server. Although this approach provides fresh data, it cannot achieve high performance for both OLTP and OLAP. This is due to data synchronization penalties and workload interference. This effect is studied in [34] by running CH-benCHmark [13], an HTAP benchmark on HyPer and SAP HANA. The study found that when a system co-runs analytical queries, its maximum attainable OLTP throughput is sig-

*Zhuhe Fang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415535

nificantly reduced. SAP HANA [22] throughput was reduced by at least three times, and HyPer by at least five times. Similar results are confirmed in MemSQL [24]. Furthermore, in-memory databases cannot provide high availability and scalability if they are only deployed on a single server.

To guarantee isolated performance, it is necessary to run OLTP and OLAP requests on different hardware resources. The essential difficulty is to maintain up-to-date replicas for OLAP requests from OLTP workloads within a single system. Besides, the system needs to maintain data consistency among more replicates. Note that maintaining consistent replicas is also required for availability [29]. High availability can be achieved using well-known consensus algorithms, such as Paxos [20] and Raft [29]. They are based on replicated state machines to synchronize replicas. It is possible to extend these consensus algorithms to provide consistent replicas for HTAP workloads. To the best of our knowledge, this idea has not been studied before.

Following this idea, we propose a Raft-based HTAP database: TiDB. It introduces dedicated nodes (called *learners*) to the Raft consensus algorithm. The learners asynchronously replicate transactional logs from leader nodes to construct new replicas for OLAP queries. In particular, the learners transform the row-format tuples in the logs into column format so that the replicas are better-suited to analytical queries. Such log replication incurs little overhead on transactional queries running on leader nodes. Moreover, the latency of such replication is so short that it can guarantee data freshness for OLAP. We use different data replicas to separately process OLAP and OLTP requests to avoid interference between them. We can also optimize HTAP requests based on both row-format and column-format data replicas. Based on the Raft protocol, TiDB provides high availability, scalability, and data consistency.

TiDB presents an innovative solution that helps consensus algorithms-based NewSQL systems evolve into HTAP systems. NewSQL systems ensure high availability, scalability, and data durability for OLTP requests by replicating their database like Google Spanner and CockroachDB. They synchronize data across data replicas via replication mechanisms typically from consensus algorithms. Based on the log replication, NewSQL systems can provide a columnar replica dedicated to OLAP requests so that they can support HTAP requests in isolation like TiDB.

We conclude our contributions as follows.

- We propose building an HTAP system based on consensus algorithms and have implemented a Raft-based HTAP database, TiDB. It is an open-source project [7] that provides high availability, consistency, scalability, data freshness, and isolation for HTAP workloads.
- We introduce the learner role to the Raft algorithm to generate a columnar store for real-time OLAP queries.
- We implement a multi-Raft storage system and optimize its reads and writes so that the system offers high performance when scaling to more nodes.
- We tailor an SQL engine for large-scale HTAP queries. The engine can optimally choose to use a row-based store and a columnar store.
- We conduct comprehensive experiments to evaluate TiDB's performance about OLTP, OLAP, and HTAP using CH-benCHmark, an HTAP benchmark.

The remainder of this paper is organized as follows. We describe the main idea, Raft-based HTAP, in Section 2, and illustrate the architecture of TiDB in Section 3. TiDB's multi-Raft storage and HTAP engines are elaborated upon in Sections 4 and 5. Experimental evaluation is presented in Section 6. We summarize related work in Section 7. Finally, we conclude our paper in Section 8.

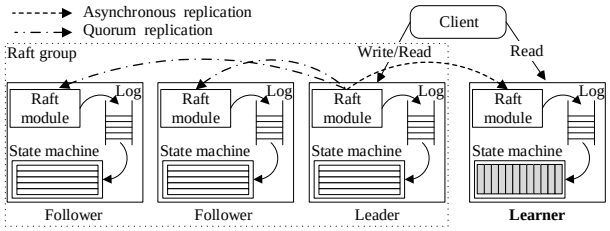


Figure 1: Adding columnar learners to a Raft group

2. RAFT-BASED HTAP

Consensus algorithms such as Raft and Paxos are the foundation of building consistent, scalable, and highly-available distributed systems. Their strength is that data is reliably replicated among servers in real time using the replicated state machine. We adapt this function to replicate data to different servers for different HTAP workloads. In this way, we guarantee that OLTP and OLAP workloads are isolated from each other, but also that OLAP requests have a fresh and consistent view of the data. To the best of our knowledge, there is no previous work to use these consensus algorithms to build an HTAP database.

Since the Raft algorithm is designed to be easy to understand and implement, we focus on our Raft extension on implementing a production-ready HTAP database. As illustrated in Figure 1, at a high level, our ideas are as follows: Data is stored in multiple Raft groups using row format to serve transactional queries. Each group is composed of a leader and followers. We add a learner role for each group to *asynchronously* replicate data from the leader. This approach is low-overhead and maintains data consistency. Data replicated to learners are transformed to column-based format. Query optimizer is extended to explore physical plans accessing both the row-based and column-based replicas.

In a standard Raft group, each follower can become the leader to serve read and write requests. Simply adding more followers, therefore, will not isolate resources. Moreover, adding more followers will impact the performance of the group because the leader must wait for responses from a larger quorum of nodes before responding to clients. Therefore, we introduced a learner role to the Raft consensus algorithm. A learner does not participate in leader elections, nor is it part of a quorum for log replication. Log replication from the leader to a learner is asynchronous; the leader does not need to wait for success before responding to the client. The strong consistency between the leader and the learner is enforced during the read time. By design, the log replication lag between the leader and learners is low, as demonstrated in the evaluation section.

Transactional queries require efficient data updates, while analytical queries such as join or aggregation require reading a subset of columns, but a large number of rows for those columns. Row-based format can leverage indexes to efficiently serve transactional queries. Column-based format can leverage data compression and vectorized processing efficiently. Therefore, when replicating to Raft learners, data is transformed from row-based format to column-based format. Moreover, learners can be deployed in separate physical resources. As a result, transaction queries and analytical queries are processed in isolated resources.

Our design also provides new optimization opportunities. Because data is kept consistent between both the row-based format and column-based format, our query optimizer can produce physical plans which access either or both stores.

We have presented our ideas of extending Raft to satisfy the freshness and isolation requirements of an HTAP database. To make an HTAP database production ready, we have overcome many engineering challenges, mainly including:

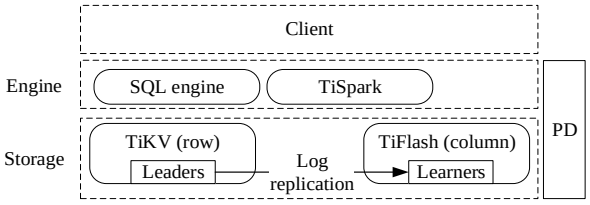


Figure 2: TiDB architecture

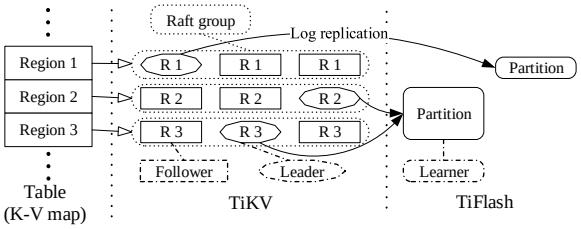


Figure 3: The architecture of multi-Raft storage

(1) How to build a scalable Raft storage system to support highly concurrent read/write? If the amount of data exceeds the available space on each node managed by the Raft algorithm, we need a partition strategy to distribute data on servers. Besides, in the basic Raft process, requests are processed sequentially, and any request must be approved by the quorum of Raft nodes before responding to clients. This process involves network and disk operations, and thus is time-consuming. This overhead makes the leader become a bottleneck to processing requests, especially on large datasets

(2) How to synchronize logs into learners with low latency to keep data fresh? Undergoing transactions can generate some very large logs. These logs need to be quickly replayed and materialized in learners so that the fresh data can be read. Transforming log data into column format may encounter errors due to mismatched schemas. This may delay log synchronization.

(3) How to efficiently process both transactional and analytical queries with guaranteed performance? Large transactional queries need to read and write huge amounts of data distributed in multiple servers. Analytical queries also consume intensive resources and should not impact online transactions. To reduce execution overhead, they also need to choose optimal plans on both a row-format store and a column-format store.

In the following sections, we will elaborate the design and implementation of TiDB to address these challenges.

3. ARCHITECTURE

In this section, we describe the high-level structure of TiDB, which is illustrated in Figure 2. TiDB supports the MySQL protocol and is accessible by MySQL-compatible clients. It has three core components: a distributed storage layer, a Placement Driver (PD), and a computation engine layer.

The distributed storage layer consists of a row store (TiKV) and a columnar store (TiFlash). Logically, the data stored in TiKV is an ordered key-value map. Each tuple is mapped into a key-value pair. The key is composed of its table ID and row ID, and the value is the actual row data, where the table ID and row ID are unique integers, and the row ID would be from a primary key column. For example, a tuple with four columns is encoded as:

Key:{*table*{*tableID*}_*record*{*rowID*}}
Value: {*col0, col1, col2, col3*}

To scale out, we take a range partition strategy to split the large key-value map into many contiguous ranges, each of which is called a *Region*. Each Region has multiple replicas for high availability. The Raft consensus algorithm is used to maintain consistency among replicas for each Region, forming a Raft group. The leaders of different Raft groups asynchronously replicate data from TiKV to TiFlash. TiKV and TiFlash can be deployed in separate physical resources and thus offer isolation when processing transactional and analytical queries.

Placement Driver (PD) is responsible for managing Regions, including supplying each key's Region and physical location, and automatically moving Regions to balance workloads. PD is also our timestamp oracle, providing strictly increasing and globally unique timestamps. These timestamps also serve as our transaction IDs. PD may contain multiple PD members for robustness and performance. PD has no persistent state, and on startup a PD member gathers all necessary data from other members and TiKV nodes.

The computation engine layer is stateless and is scalable. Our tailored SQL engine has a cost-based query optimizer and a distributed query executor. TiDB implements a two-phase commit (2PC) protocol based on Percolator [33] to support transactional processing. The query optimizer can optimally select to read from TiKV and TiFlash based on the query.

The architecture of TiDB meets the requirement of an HTAP database. Each component of TiDB is designed to have high availability and scalability. The storage layer uses the Raft algorithm to achieve consistency between data replicas. The low latency replication between the TiKV and TiFlash makes fresh data available to analytical queries. The query optimizer, together with the strongly-consistent data between TiKV and TiFlash, offers fast analytical query processing with little impact on transactional processing.

Besides the components mentioned above, TiDB also integrates with Spark, which is helpful to integrate data stored in TiDB and the Hadoop Distributed File System (HDFS). TiDB has a rich set of ecosystem tools to import data to and export data from TiDB and migrate data from other databases to TiDB.

In the following sections, we will do a deep dive on the distributed storage layer, the SQL engine, and TiSpark to demonstrate the capability of TiDB, a production-ready HTAP database.

4. MULTI-RAFT STORAGE

Figure 3 shows the architecture of the distributed storage layer in TiDB, where the objects with the same shape play the same role. The storage layer consists of a row-based store, *TiKV*, and a column-based store, *TiFlash*. The storage maps a large table into a big key-value map which is split into many Regions stored in TiKV. Each Region uses the Raft consensus algorithm to maintain the consistency among replicas to achieve high availability. Multiple Regions can be merged into one partition when data is replicated to TiFlash to facilitate table scan. The data between TiKV and TiFlash is kept consistent through asynchronous log replication. Since multiple Raft groups manage data in the distributed storage layer, we call it multi-Raft storage. In the following sections, we describe TiKV and TiFlash in detail, focusing on optimizations to make TiDB a production-ready HTAP database.

4.1 Row-based Storage (TiKV)

A TiKV deployment consists of many TiKV servers. Regions are replicated between TiKV servers using Raft. Each TiKV server can be either a Raft leader or follower for different Regions. On each TiKV server, data and metadata are persisted to RocksDB, an embeddable, persistent, key-value store [5]. Each Region has a configurable max size, which is 96 MB by default. The TiKV server for a Raft leader handles read/write requests for the corresponding Region.

When the Raft algorithm responds to read and write requests, the basic Raft process is executed between a leader and its followers:

(1) A Region leader receives a request from the SQL engine layer.
(2) The leader appends the request to its log.
(3) The leader sends the new log entries to its followers, which in turn append the entries to their logs.
(4) The leader waits for its followers to respond. If a quorum of nodes respond successfully, then the leader commits the request and applies it locally.
(5) The leader sends the result to the client and continues to process incoming requests.

This process ensures data consistency and high availability. However, it does not provide efficient performance because the steps happen sequentially, and may incur large I/O overheads (both disk and network). The following sections describe how we have optimized this process to achieve high read/write throughput, i.e., solving the first challenge described in Section 2.

4.1.1 Optimization between Leaders and Followers

In the process described above, the second and third steps can happen in parallel because there is no dependency between them. Therefore, the leader appends logs locally and sends logs to followers at the same time. If appending logs fails on the leader but a quorum of the followers successfully append the logs, the logs can still be committed. In the third step, when sending logs to followers, the leader buffers log entries and sends them to its followers in batches. After sending the logs, the leader does not have to wait for the followers to respond. Instead, it can assume success and send further logs with the predicted log index. If errors occur, the leader adjusts the log index and resends the replication requests. In the fourth step, the leader applying committed log entries can be handled asynchronously by a different thread because at this stage there is no risk to consistency. Based on the optimizations above, the Raft process is updated as follows:

(1) A leader receives requests from the SQL engine layer.
(2) The leader sends corresponding logs to followers and appends logs locally in parallel.
(3) The leader continues to receive requests from clients and repeats step (2).
(4) The leader commits the logs and sends them to another thread to be applied.
(5) After applying the logs, the leader returns the results to the client.

In this optimal process, any request from a client still runs all the Raft steps, but requests from multiple clients are run in parallel, so the overall throughput increases.

4.1.2 Accelerating Read Requests from Clients

Reading data from TiKV leaders is provided with linearizable semantics. This means when a value is read at time t from a Region leader, the leader must not return a previous version of the value for read requests after t. This can be achieved by using Raft as described above: issuing a log entry for every read request and waiting for that entry to be committed before returning. However, this process is expensive because the log must be replicated across the majority of nodes in a Raft group, incurring the overhead of network I/O. To improve performance, we can avoid the log synchronization phase.

Raft guarantees that once the leader successfully writes its data, the leader can respond to any read requests without synchronizing logs across servers. However, after a leader election, the leader role may be moved between servers in a Raft group. To achieve reading from leaders, TiKV implements the following read optimizations as described in [29].

The first approach is called *read index*. When a leader responds to a read request, it records the current commit index as a local read index, and then sends heartbeat messages to followers to confirm its leader role. If it is indeed the leader, it can return the value once its applied index is greater than or equal to the read index. This approach improves read performance, though it causes a little network overhead.

Another approach is *lease read*, which reduces the network overhead of heartbeats caused by the read index. The leader and followers agree on a lease period, during which followers do not issue election requests so that the leader is not changed. During the lease period, the leader can respond to any read request without connecting to its followers. This approach works well if the CPU clock on each node does not differ very much.

In addition to the leader, followers can also respond to read requests from clients, which is called *follower read*. After a follower receives a read request, it asks the leader for the newest read index. If the locally-applied index is equal to or greater than the read index, the follower can return the value to the client; otherwise, it has to wait for the log to be applied. Follower read can alleviate the pressure on the leader of a hot Region, thus improving read performance. Read performance can then be further improved by adding more followers.

4.1.3 Managing Massive Regions

Massive Regions are distributed on a cluster of servers. The servers and data size are dynamically changing, and Regions may cluster in some servers, especially leader replicas. This causes some servers' disks to become overused, while others are free. In addition, servers may be added to or moved from the cluster.

To balance Regions across servers, the Plancement Driver (PD) schedules Regions with constraints on the number and location of replicas. One critical constraint is to place at least three replicas of a Region on different TiKV instances to ensure high availability. PD is initialized by collecting specific information from servers through heartbeats. It also monitors the workloads of each server and migrates hot Regions to different servers without impacting applications.

On the other hand, maintaining massive Regions involves sending heartbeats and managing metadata, which can cause a lot of network and storage overhead. However, if a Raft group does not have any workloads, the heartbeat is unnecessary. Depending on how busy the Regions' the workloads are, we can adjust the frequency of sending heartbeats. This reduces the likelihood of running into issues like network latency or overloaded nodes.

4.1.4 Dynamic Region Split and Merge

A large Region may become too hot to be read or written in a reasonable time. Hot or large Regions should be split into smaller ones to better distribute workload. On the other hand, it is possible that many Regions are small and seldom accessed; however, the system still needs to maintain the heartbeat and metadata. In some cases, maintaining these small Regions incurs significant network and CPU overhead. Therefore, it is necessary to merge smaller Regions. Note that to maintain the order between Regions, we only merge adjacent Regions in the key space. Based on observed workloads, PD dynamically sends split and merge commands to TiKV.

A split operation divides a Region into several new, smaller Regions, each of which covers a continuous range of keys in the original Region. The Region that covers the rightmost range reuses the Raft group of the original Region. Other Regions use new Raft groups. The split process is similar to a normal update request in the Raft process:

Table 1: Log replaying and decoding

Raw logs	{1}{insert}{prewritten@1}{k1→ (a1, b1)} {2}{insert}{prewritten@2}{k2→ (a2, b2)} {3}{update}{prewritten@3}{k3→ (a3, b3)} {1}{insert}{rollbacked@1} {2}{insert}{committed#4} {3}{update}{committed#5} {4}{delete}{prewritten@6}{k4} {4}{delete}{committed#7}
Compacted logs	{2}{insert}{prewritten@2}{k2→ (a2, b2)} {3}{update}{prewritten@3}{k3→ (a3, b3)} {2}{insert}{committed#4} {3}{update}{committed#5} {4}{delete}{prewritten@6}{k4} {4}{delete}{committed#7}
Decoded tuples	{insert}{#4}{k2→ (a2, b2)} {update}{#5}{k3→ (a3, b3)} {delete}{#7}{k4}
Columnar data	{insert,update,delete,} {#4,#5,#7,} {k2,k3,k4,} {a2,a3,,} {b2,b3,,}

(1) PD issues a split command to the leader of a Region.
(2) After receiving the split command, the leader transforms the command into a log and replicates the log to all its follower nodes. The log only includes a split command, instead of modifying actual data.
(3) Once a quorum replicates the log, the leader commits the split command, and the command is applied to all the nodes in the Raft group. The apply process involves updating the original Region's range and epoch metadata, and creating new Regions to cover the remaining range. Note that the command is applied atomically and synced to disk.
(4) For each replica of a split Region, a Raft state machine is created and starts to work, forming a new Raft group. The leader of the original Region reports the split result to PD. The split process completes.

Note that the split process succeeds when a majority of nodes commit the split log. Similar to committing other Raft logs, rather than requiring all nodes to finish splitting the Region. After the split, if the network is partitioned, the group of nodes with the most recent epoch wins. The overhead of region split is low as only metadata change is needed. After a split command finishes, the newly split Regions may be moved across servers due to PD's regular load balancing.

Merging two adjacent Regions is the opposite of splitting one. PD moves replicas of the two Regions to colocate them on separate servers. Then, the colocated replicas of the two Regions are merged locally on each server through a two-phase operation; that is, stopping the service of one Region and merging it with another one. This approach is different from splitting a Region, because it cannot use the log replication process between two Raft groups to agree on merging them.

4.2 Column-based Storage (TiFlash)

Even though we optimize reading data from TiKV as described above, the row-format data in TiKV is not well-suited for fast analysis. Therefore, we incorporate a column store (TiFlash) into TiDB. TiFlash is composed of learner nodes, which just receive Raft logs from Raft groups and transform row-format tuples into columnar data. They do not participate in the Raft protocols to commit logs or elect leaders so they induce little overhead on TiKV.

A user can set up a column-format replica for a table using an SQL statement:

ALTER TABLE x SET TiFLASH REPLICA n;

where *x* is the name of the table and *n* is the number of replicas. The default value is *1*. Adding a column replica resembles adding an asynchronous columnar index to a table. Each table in TiFlash is divided into many partitions, each of which covers a contiguous range of tuples, in accordance with several continuous Regions from TiKV. The larger partition facilitates range scan.

When initializing a TiFlash instance, the Raft leaders of the relevant Regions begin to replicate their data to the new learners. If there is too much data for fast synchronization, the leader sends a snapshot of its data. Once initialization is complete, the TiFlash instance begins listening for updates from the Raft groups. After a learner node receives a package of logs, it applies the logs to the local state machine, including replaying the logs, transforming the data format, and updating the referred values in local storage.

In the following sections, we illustrate how TiFlash efficiently applies logs and maintains a consistent view with TiKV. This meets the second challenge we described in Section 2.

4.2.1 Log Replayer

In accordance with the Raft algorithm, the logs received by learner nodes are linearizable. To keep the linearizable semantics of committed data, they are replayed according to a first-in, first-out (FIFO) strategy. The log replay has three steps:

(1) Compacting logs: According to the transaction model described in later Section 5.1, the transactional logs are classified into three statuses: prewritten, committed, or rollbacked. The data in the rollbacked logs does not need to be written to disks, so a compact process deletes invalid prewritten logs according to rollbacked logs and puts valid logs into a buffer.
(2) Decoding tuples: The logs in the buffer are decoded into row-format tuples, removing redundant information about transactions. Then, the decoded tuples are put into a row buffer.
(3) Transforming data format: If the data size in the row buffer exceeds a size limit or its time duration exceeds a time interval limit, these row-format tuples are transformed to columnar data and are written to a local partition data pool. Transformation refers to local cached schemas, which are periodically synchronized with TiKV as described later.

To illustrate the details of the log replay process, consider the following example. We abstract each Raft log item as transaction ID-operation type[transaction status][@start_ts][#commit_ts]operation data. According to typical DMLs, the operation type includes inserting, updating, and deleting tuples. Transactional status may be prewritten, committed, or rollbacked. Operation data may be a specifically-inserted or updated tuple, or a deleted key.

In our example shown in Table 1, the raw logs contain eight items which attempt to insert two tuples, update one tuple, and delete one tuple. But inserting k1 is rolled back, so only six of the eight raw log items are preserved, from which three tuples are decoded. Finally, the three decoded tuples are transformed into five columns: operation types, commit timestamps, keys, and two columns of data. These columns are appended to the DeltaTree.

4.2.2 Schema Synchronization

To transform tuples into columnar format in real time, learner nodes have to be aware of the newest schema. Such a schemaful process is different from schemaless operations on TiKV which encode tuples as byte arrays. The newest schema information is stored in TiKV. To reduce the number of times TiFlash asks TiKV for the newest schema, each learner node maintains a schema cache.

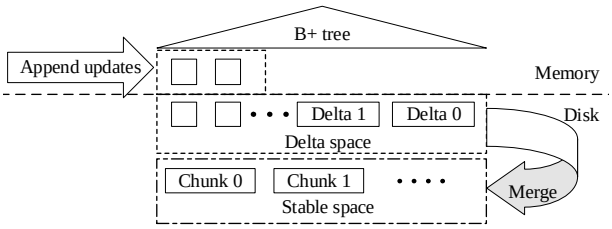


Figure 4: The columnar delta tree

The cache is synchronized with TiKV's schema through a schema syncer. If the cached schema is out of date, there is a mismatch between data being decoded and the local schema, and data must be retransformed. There is a trade-off between the frequency of schema synchronization and the number of schema mismatches. We take a two-stage strategy:

- Regular synchronization: the schema syncer fetches the newest schema from TiKV periodically and applies changes to its local cache. In most cases, this casual synchronization reduces the frequency of synchronizing schemas.
- Compulsive synchronization: if the schema syncer detects a mismatched schema, it proactively fetches the newest schema from TiKV. This can be triggered when the column number differs between tuples and schemas or a column value overflows.

4.2.3 *Columnar Delta Tree*

To efficiently write and read the columnar data with high throughput, we design a new columnar storage engine, DeltaTree, which appends delta updates immediately and later merges them with the previously stable version for each partition. The delta updates and the stable data are stored separately in the DeltaTree, as shown in Figure 4. In the stable space, partition data is stored as chunks, and each of them covers a smaller range of the partition's tuples. Moreover, these row-format tuples are stored column by column. In contrast, deltas are directly appended into the delta space in the order TiKV generates them. The store format of columnar data in TiFlash is similar to Parquet [4]. It also stores row groups into columnar chunks. Differently, TiFlash stores column data of a row group and its metadata to different files to concurrently update files, instead of only one file in Parquet. TiFlash just compresses data files using the common LZ4 [2] compression to save their disk size.

New incoming deltas are an atomic batch of inserted data or a deleted range. These deltas are cached into memory and materialized into disks. They are stored in order, so they achieve the function of a write-ahead log (WAL). These deltas are usually stored in many small files and therefore induce large IO overhead when read. To reduce cost, we periodically compact these small deltas into a larger one, then flush the larger deltas to disks and replace the previously materialized small deltas. The in-memory copy of incoming deltas facilitates reading the latest data, and if the old deltas reach a limited size, they are removed.

When reading the latest data of some specific tuples, it is necessary to merge all delta files with their stable tuples (i.e., read amplification), because where the related deltas distribute is not known in advance. Such a process is expensive due to reading a lot of files. In addition, many delta files may contain useless data (i.e., space amplification) that wastes storage space and slows down merging them with stable tuples. Therefore, we periodically merge the deltas into the stable space. Each delta file and its related chunks are read into memory and merged. Inserted tuples in deltas are added into the stable, modified tuples replace the original tuples, and deleted tuples are moved. The merged chunks atomically replace original chunks in disks.

Table 2: Read performance of DeltaTree and LSM tree

Tuple number	Storage engine	Transactions per second			
		150	1300	8000	14000
100 M	Delta tree	0.49	0.48	0.5	0.45
	LSM tree	1.01	0.97	0.94	0.95
200 M	Delta tree	0.71	0.74	0.72	0.75
	LSM tree	1.59	1.63	1.67	1.64

Merging deltas is expensive because the related keys are disordered in the delta space. Such disorder also slows down integrating deltas with stable chunks to return the latest data for read requests. Therefore, we build a B+ tree index on the top of the delta space. Each delta item of updates is inserted into the B+ tree ordered by its key and timestamp. This order priority helps to efficiently locate updates for a range of keys or to look up a single key in the delta space when responding to read requests. Also, the ordered data in the B+ tree is easy to merge with stable chunks.

We conduct a micro-experiment to compare the DeltaTree's performance to the log-structured-merge (LSM) tree [28] in TiFlash, where data is read as it is updated according to Raft logs. We set three TiKV nodes and one TiFlash node, and the hardware configurations are listed in the experimental section. We run the only write workload of Sysbench [6] on TiKV and run "*select count(id), count(k) from sbtest1*" on TiFlash. To avoid the large write amplification of data compaction, we implement the LSM storage engine using a universal compaction, rather than a level style compaction. This implementation is also adopted in ClickHouse [1], a column-oriented OLAP database.

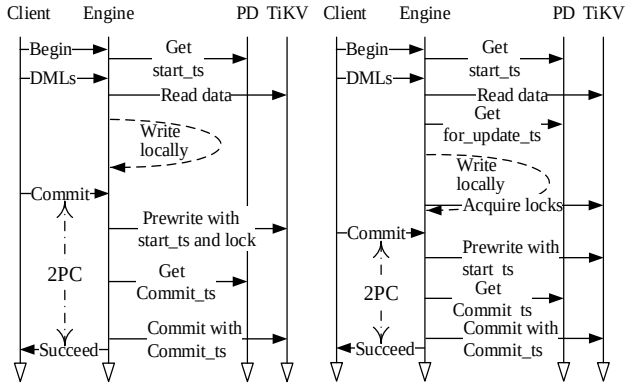
As shown in Table 2, reading from the delta tree is about two times faster than the LSM tree, regardless of whether there are 100 or 200 million tuples, as well as the transactional workloads. This is because in the delta tree each read accesses at most one level of delta files that are indexed in a B+ tree, while it accesses more overlapped files in the LSM tree. The performance remains almost stable under different write workloads because the ratio of delta files is nearly the same. Although the write amplification of DeltaTree (16.11) is greater than the LSM tree (4.74), it is also acceptable.

4.2.4 *Read Process*

Like follower read, learner nodes provide snapshot isolation so we can read data from TiFlash at a specific timestamp. After receiving a read request, the learner sends a read index request to its leaders to get the newest data that covers the requested timestamp. In response, the leaders send the referred logs to the learner, and the learner replays and stores the logs. Once the logs are written into DeltaTree, the specific data from the DeltaTree is read to respond to the read request.

5. HTAP ENGINES

To solve the third challenge mentioned in Section 2, i.e., processing large-scale transactions and analytical queries, we provide an SQL engine to evaluate transactional and analytical queries. The SQL engine adapts the Percolator model to implement optimistic and pessimistic locking in a distributed cluster. The SQL engine accelerates analytical queries by using a rule- and cost-based optimizer, indexes, and pushing down computation to the storage layer. We also implement TiSpark to connect with the Hadoop ecosystem and enhance OLAP capability. HTAP requests can be processed separately in isolated stores and engine servers. In particular, the SQL engine and TiSpark benefit from using both row and column stores at the same time to get optimal results.



(a) Optimistic transaction (b) Pessimistic transaction

Figure 5: The process of optimistic and pessimistic transaction

5.1 Transactional Processing

TiDB provides ACID transactions with snapshot-isolation (SI) or repeatable read (RR) semantics. SI lets each request within a transaction read a consistent version of the data. RR means that different statements in a transaction might read different values for the same key, but that repeating a read (i.e., two reads with the same timestamp) will always read the same value. Our implementation, based on multi-version concurrency control (MVCC), avoids read-write locking and protects against write-write conflicts.

In TiDB, transactions are collaborative between the SQL engine, TiKV, and PD. The responsibility of each component during transaction is as follows:

- SQL engine: coordinates transactions. It receives the write and read requests from clients, transforms data into a key-value format, and writes the transactions to TiKV using two-phase commit (2PC).
- PD: manages logical Regions and physical locations; provides global, strictly-increasing timestamps.
- TiKV: provides distributed transaction interfaces, implements MVCC, and persists data to disk.

TiDB implements both optimistic and pessimistic locking. They are adapted from the Percolator model, which selects one key as the primary key and uses it to stand for the status of a transaction, and base 2PC to conduct transactions. The process of an optimistic transaction is illustrated on the left of Figure 5. (For simplicity, the figure ignores exception handling.)

(1) After receiving a "begin" command from a client, the SQL engine asks PD for a timestamp to use as the start timestamp (*start_ts*) of the transaction.
(2) The SQL engine executes SQL DMLs by reading data from TiKV and writing to local memory. TiKV supplies data with the most recent commit timestamp (*commit_ts*) before the transaction's *start_ts*.
(3) When the SQL engine receives a commit command from the client, it starts the 2PC protocol. It randomly chooses a primary key, locks all keys in parallel, and sends prewrites to TiKV nodes.
(4) If all prewrites succeed, the SQL engine asks PD for a timestamp for the transaction's *commit_ts* and sends a commit command to TiKV. TiKV commits the primary key and sends a success response back to the SQL engine.
(5) The SQL engine returns success to the client.
(6) The SQL engine commits secondary keys and clears locks asynchronously and in parallel by sending further commit commands to TiKV.

The main difference between optimistic and pessimistic transactions is when locks are acquired. In optimistic transactions, locks are acquired incrementally in the prewrite phase (step 3 above). In pessimistic transactions, locks are acquired as DMLs are executed before prewrite (part of step 2). That means that once prewrite starts, the transaction will not fail due to conflict with another transaction. (It can still fail due to network partition, or other issues.)

When locking keys in a pessimistic transaction, the SQL engine acquires a new timestamp, called the *for_update_ts*. If the SQL engine cannot acquire a lock, it can retry the transaction starting at that lock, rather than rolling back and retrying the whole transaction. When reading data, TiKV uses *for_update_ts* rather than *start_ts* to decide which values of a key can be read. In this manner, pessimistic transactions maintain the RR isolation level, even with partial retries of transactions.

With pessimistic transactions, users can also opt to require only the read committed (RC) isolation level. This causes less conflict between transactions and thus better performance, at the expense of less isolated transactions. The difference in implementation is that for RR TiKV must report a conflict if a read tries to access a key locked by another transaction; for RC, locks can be ignored for reading.

TiDB implements distributed transactions without a centralized lock manager. Locks are stored in TiKV, giving high scalability and availability. Moreover, the SQL engine and PD servers are scalable to handle OLTP requests. Running many transactions simultaneously across servers achieves a high degree of parallelism.

Timestamps are requested from PD. Each timestamp includes the physical time and logical time. The physical time refers to the current time with millisecond accuracy, and the logical time takes 18 bits. Therefore, in theory, PD can allocate 2^{18} timestamps per millisecond. In practice, it can generate about 1 million timestamps per second as allocating timestamps only costs a few cycles. Clients ask for timestamps once a batch to amortize overhead, especially network latency. At present, getting timestamps is not a performance bottleneck in our experiments and in many productive environments.

5.2 Analytical Processing

In this section, we describe our optimizations targeted at OLAP queries, including an optimizer, indexes, and pushing down computation in our tailored SQL engine and TiSpark.

5.2.1 Query Optimization in SQL Engine

TiDB implements a query optimizer with two phases of query optimisation: rule-based optimization (RBO) of the query which produces a logical plan, then cost-based optimization (CBO) which transforms a logical plan to a physical plan. Our RBO has a rich set of transformation rules, including cropping unneeded columns, eliminating projection, pushing down predicates, deriving predicates, constant folding, eliminating "group by" or outer joins, and unnesting subqueries. Our CBO chooses the cheapest plan from candidate plans according to execution costs. Note that TiDB provides two data stores, TiKV and TiFlash, so scanning tables typically has three options: scanning row-format tables in TiKV, scanning tables with indexes in TiKV, and scanning columns in TiFlash.

Indexes are important to improve query performance in databases, which are usually used in point-get or range queries, providing more cheaper data scan paths for hash joins and merge joins. TiDB implements scalable indexes to work in a distributed environment. Because maintaining indexes consumes a significant amount of resources and may affect online transactions and analysis, we asynchronously build or drop indexes in the background. Indexes are

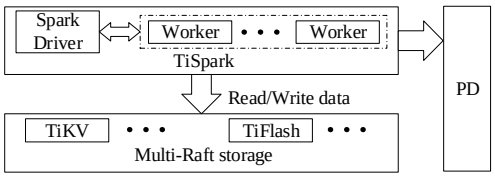


Figure 6: The interaction of TiSpark and TiDB

split by Regions in the same way as data and stored in TiKV as key-values. An index item on a unique key index is encoded as:
Key: {*table*{*tableID*}*_index*{*indexID*}*_indexedColValue*}
Value: {*rowID*}
An index item on a nonunique index is decoded as:
Key: {*table*{*tableID*}*_index*{*indexID*}*_indexedColValue_rowID*}
Value: {*null*}
Using an index requires a binary search to locate the Regions which contain relevant parts of the index. To increase the stability of index selection and reduce the overhead of physical optimization, we use a skyline pruning algorithm to eliminate useless candidate indexes. If there are multiple candidate indexes that match different query conditions, we merge partial results (i.e., a set of qualified row IDs) to get a precise results set.

Physical plans (the result of CBO) are executed by the SQL engine layer using the pulling iterator model [17]. Execution can be further optimized by pushing down some computation to the storage layer. In the storage layer, the component that performs the computation is called a *coprocessor*. The coprocessor executes subtrees of an execution plan on different servers in parallel. This reduces the number of tuples that must be sent from the storage layer to the engine layer. For example, by evaluating filters in the coprocessor, rejected tuples are filtered out in the storage layer, and only the accepted tuples need to be sent to the engine layer. The coprocessor can evaluate logical operations, arithmetic operations, and other common functions. In some cases, it can execute aggregations and TopN. The coprocessor can further improve performance by vectorizing operations: instead of iterating over whole rows, rows are batched and data is organized by column, resulting in much more efficient iteration.

5.2.2 *TiSpark*

To help TiDB connect to the Hadoop ecosystem, TiDB adds TiSpark on the multi-Raft storage. In addition to SQL, TiSpark supports powerful computation such as machine learning libraries and can process data from outside TiDB.

Figure 6 shows how TiSpark integrates with TiDB. In TiSpark, the Spark driver reads metadata from TiKV to construct a Spark catalog, including table schemas and index information. The Spark driver asks PD for timestamps to read MVCC data from TiKV to ensure it gets a consistent snapshot of the database. Like the SQL engine, Spark Driver can push down computation to the coprocessor on the storage layer and use available indexes. This is done by modifying the plans generated by the Spark optimizer. We also customize some read operations to read data from TiKV and TiFlash, and assemble them into rows for the Spark workers. For example, TiSpark can simultaneously read from multiple TiDB Regions, and it can get index data from the storage layer in parallel. To reduce dependency on specific versions of Spark, most of these functions are implemented in additional packages.

TiSpark differs from common connectors in two aspects. Not only can it simultaneously read multiple data Regions, it can also get index data from the storage layer in parallel. Reading indexes can facilitate the optimizer in Spark to choose optimal plans to reduce execution cost. On the other hand, TiSpark modifies plans generated from the raw optimizer in Spark to push down parts of execution to the coprocessor in the storage layer, which further reduces execution overhead. In addition to reading data from the storage layer, TiSpark also supports loading large data in the storage layer with transactions. To achieve this, TiSpark takes the two-phase commit and locks tables.

5.3 Isolation and Coordination

Resource isolation is an effective way to guarantee the performance of transactional queries. Analytical queries often consume high levels of resources such as CPU, memory, and I/O bandwidth. If these queries run together with transactional queries, the latter can be seriously delayed. This general principle has been verified in previous work [24, 34]. To avoid this problem in TiDB, we schedule analytical and transactional queries on different engine servers, and deploy TiKV and TiFlash on separate servers. Transactional queries mainly access TiKV, whereas analytical queries mainly access TiFlash. The overhead of maintaining data consistency between TiKV and TiFlash via Raft is low, so running analytical queries with TiFlash has little impact on the performance of transactional processing.

Data is consistent across TiKV and TiFlash, therefore queries can be served by either reading from TiKV or TiFlash. As a result, our query optimizer can choose from a larger physical plan space, and the optimal plan can potentially read from both TiKV and TiFlash. When TiKV accesses a table, it provides a row scan and an index scan, and TiFlash supports a column scan.

These three access paths differ from each other in their execution costs and data order properties. Row scan and column scan provide order by primary key; index scan offers several orderings from the encoding of keys. The costs of different paths depend on the average tuple/column/index size ($S_{tuple/col/index}$) and estimated number of tuples/Regions ($N_{tuple/reg}$). We express the I/O overhead of the data scan as f_{scan}, and the file seeking cost as f_{seek}. The query optimizer chooses an optimal access path according to Equation (1). As shown in Equation (2), the cost of the row scan comes from scanning contiguous row data and seeking Region files. The cost of the column scan (Equation (3)) is the sum of scanning m columns. If the indexed columns do not satisfy the needed column of the table scan, the index scan (Equation (4)) should consider the cost of scanning index files and the cost of scanning data files (i.e., double read). Note that double read usually scans tuples randomly, which involves seeking more files in Equation (5).

$$C_{opt\_scan} = \min(C_{col\_scan}, C_{row\_scan}, C_{index\_scan}) \quad (1)$$

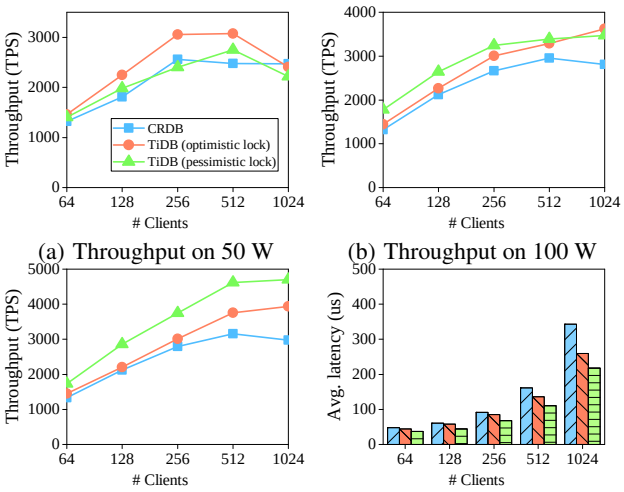
$$C_{row\_scan} = S_{tuple} \cdot N_{tuple} \cdot f_{scan} + N_{reg} \cdot f_{seek} \quad (2)$$

$$C_{col\_scan} = \sum_{j=1}^{m} (S_{col\_j} \cdot N_{tuple} \cdot f_{scan} + N_{reg\_j} \cdot f_{seek}) \quad (3)$$

$$C_{index\_scan} = S_{index} \cdot N_{tuple} \cdot f_{scan} + N_{reg} \cdot f_{seek} + C_{double\_read} \quad (4)$$

$$C_{double\_read} = \begin{cases} 0 & (\texttt{if without double read}) \\ S_{tuple} \cdot N_{tuple} \cdot f_{scan} + N_{tuple} \cdot f_{seek} \end{cases} \quad (5)$$

As an example of when the query optimizer will choose both row- and column-format stores to access different tables in the same query, consider *"select T.*, S.a from T join S on T.b=S.b where T.a between 1 and 100"*. This is a typical join query, where `T` and `S` have indexes on column `a` in the row store, as well as column replicas. It is optimal to use the index to access `T` from the row store, and access `S` from the column store. This is because the query needs a range of complete tuples from `T` and accessing data by tuple through the index is cheaper than the column store. On the other hand, fetching two complete columns of `S` is cheaper when using the column store.



(a) Throughput on 50 W (b) Throughput on 100 W

(c) Throughput on 200 W (d) Latency on 200 W

Figure 7: OLTP performance

The coordination of TiKV and TiFlash can still guarantee isolated performance. For analytical queries, only a small range scan or point-get scan may access TiKV through the follower read, which makes little impact on leaders. We also limit the default access table size on TiKV for analytical queries to at most 500 MB. Transactional queries may access columnar data from TiFlash to check some constraints, such as uniqueness. We set more than one columnar replica for specific tables, and one table replica is dedicated to transactional queries. Handling transactional queries on separate servers avoids affecting analytical queries.

6. EXPERIMENTS

In this section, we first separately evaluate TiDB's OLTP and OLAP ability. For OLAP, we investigate the SQL engine's ability to choose TiKV and TiFlash, and compare TiSpark to other OLAP systems. Then, we measure TiDB's HTAP performance, including the log replication delay between TiKV and TiFlash. Finally, we compare TiDB to MemSQL in terms of isolation.

6.1 Experimental Setup

Cluster. We perform comprehensive experiments on a cluster of six servers; each has 188 GB memory and two Intel® Xeon® CPU E5-2630 v4 processors, i.e., two NUMA nodes. Each processor has 10 physical cores (20 threads) and a 25 MB shared L3 cache. The servers run Centos version 7.6.1810 and are connected by a 10 Gbps Ethernet network.

Workload. Our experiments are conducted under a hybrid OLTP and OLAP workload using CH-benCHmark. Source code is published online [7]. The benchmark is composed of standard OLTP and OLAP benchmarks: TPC-C and TPC-H. It is built from the unmodified version of the TPC-C benchmark. The OLAP part contains 22 analytical queries inspired by TPC-H, whose schema is adapted from TPC-H to the CH-benCHmark schema, plus three missing TPC-H relations. At run time, the two workloads are issued simultaneously by multiple clients; the number of clients is varied in the experiments. Throughput is measured in queries per second (QPS) or transactions per second (TPS), respectively. The unit of data in CH-benCHmark is called a warehouse, the same with TPC-C. 100 warehouses take about 70 GB of memory.

6.2 OLTP Performance

We evaluate TiDB's standalone OLTP performance with optimistic locking or pessimistic locking under the OLTP part of CH-benCHmark; i.e., the TPC-C benchmark. We compare TiDB's performance to CockroachDB (CRDB), another distributed NewSQL database. CRDB is deployed on six homogeneous servers. For TiDB, the SQL engine and TiKV are deployed on six servers, and their instances are bound to the two NUMA nodes separately on each server. PD is deployed on three of the six servers. To balance requests, both TiDB and CRDB are accessed through an HAProxy load balancer. We measure the throughput and average latency on 50, 100, and 200 warehouses using various numbers of clients.

Table 3: Performance of timestamp retrieval

Servers	Per-server statistics			
	Count	Max (ms)	>= 1ms	>=2ms
6	602594	1	248	0
12	319585	2	254	100
24	135278	2	322	15

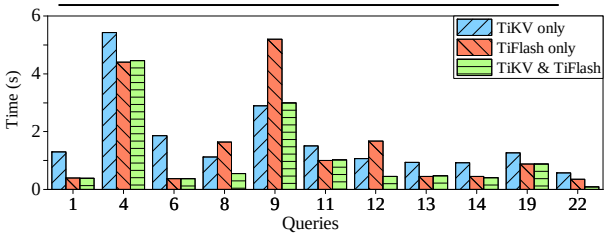


Figure 8: Choice of TiKV or TiFlash for analytical queries

The plots of the throughput in Figures 7(b) and 7(c) differ from Figure 7(a). In Figure 7(a), for less than 256 clients, the throughput of TiDB increases with the number of clients for both optimistic locking and pessimistic locking. For more than 256 clients, the throughput with optimistic locking remains stable and then starts to drop, whereas the throughput of pessimistic locking reaches its maximum with 512 clients and then drops. The throughput of TiDB in Figures 7(b) and 7(c) keeps increasing. This result is expected as the resource contention is heaviest with high concurrency and small data size.

In general, optimistic locking performs better than pessimistic locking except for smaller data sizes and with high concurrency (1024 clients on 50 or 100 warehouses) where resource contention is heavy and causes many optimistic transactions to be retried. Since resource contention is lighter with 200 warehouses, optimistic locking still produces better performance.

In most cases, the throughput of TiDB is higher than CRDB, especially when using optimistic locking on large warehouses. Even taking pessimistic locking for fair comparison (CRDB always uses pessimistic locking), TiDB's performance is still higher. We believe that TiBD's performance advantage is due to optimization of transaction processing and the Raft algorithm.

Figure 7(d) shows that more clients cause more latency, especially after reaching maximum throughput, because more requests have to wait for a longer time. This also accounts for the higher latency with fewer warehouses. For certain clients, higher throughput leads to less latency for TiDB and CRDB. Similar results exist with 50 and 100 warehouses.

We evaluate the performance of requesting timestamps from PD since this might be a potential bottleneck. We use 1200 clients to continuously request timestamps. The clients are located on varying servers in a cluster. Emulating TiDB, each client sends timestamp requests to PD in batches. As Table 3 shows, each of the six servers can receive 602594 timestamps per second, which is more than 100 times the required rate when running the TPC-C benchmark. When running TPC-C, TiDB requests at most 6000 timestamps per second per server. When increasing the number of servers, the number of received timestamps decreases on each server, but the total number of timestamps is almost the same. This

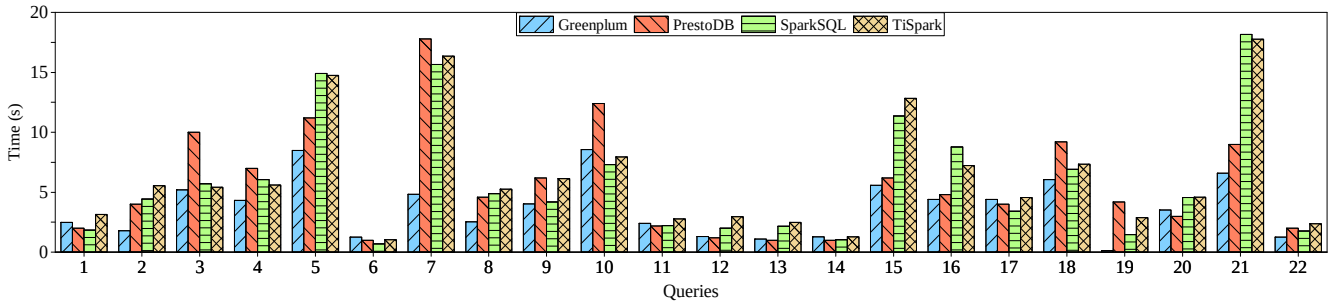


Figure 9: Performance comparison of CH-benCHmark analytical queries

rate greatly exceeds any real-life demand. Regarding latency, only a small proportion of requests cost 1 ms or 2 ms. We conclude that getting timestamps from PD is not a performance bottleneck in TiDB at present.

6.3 OLAP Performance

We evaluate the OLAP performance of TiDB from two perspectives. First, we evaluate the capability of the SQL engine to optimally choose either a row store or a column store under the OLAP part of CH-benCHmark with 100 warehouses. We set up three types of storage: TiKV only, TiFlash only, and both TiKV and TiFlash. We run each query five times and compute the average execution time. As shown in Figure 8, fetching data only from a single kind of store, neither store is superior. Requesting data from both TiKV and TiFlash always performs better.

Q8, Q12, and Q22 generate interesting results. The TiKV-only case costs less time than the TiFlash-only case in Q8 and Q12, but it takes more time in Q22. The TiKV and TiFlash case performs better than the TiKV-only and TiFlash-only cases.

Q12 mainly contains a two-table join, but it takes different physical implementations in each storage type. In the TiKV-only case, it uses an index join, which scans several qualified tuples from table `ORDER_LINE` and looks up table `OORDER` using indexes. The index reader costs so much less that it is superior to taking a hash join in the TiFlash-only case, which scans needed columns from the two tables. The cost is further reduced when using both TiKV and TiFlash because it uses the cheaper index join that scans `ORDER_LINE` from TiFlash, and looks up `OORDER` using indexes in TiKV. In the TiKV and TiFlash case, reading the column store cuts the execution time of the TiKV-only case by half.

In Q22, its *exists()* subquery is converted to an anti-semi join. It uses an index join in the TiKV-only case and a hash join in the TiFlash-only case. But unlike the execution in Q12, using the index join is more expensive than the hash join. The cost of the index join is reduced when fetching the inner table from TiFlash and looking up the outer table using indexes from TiKV. Therefore, the TiKV and TiFlash case again takes the least time.

Q8 is more complex. It contains a join with nine tables. In the TiKV-only case, it takes two index merge joins and six hash joins, and looks up two tables (`CUSTOMER` and `OORDER`) using indexes. This plan takes 1.13 seconds and is superior to taking eight hash joins in the TiFlash-only case, which costs 1.64 seconds. Its overhead is further reduced in the TiKV and TiFlash case, where the physical plan is almost unchanged except for scanning data from TiFlash in six hash joins. This improvement reduces the execution time to 0.55 seconds. In these three queries, using only TiKV or TiFlash gets varying performance and combining them achieves the best results.

For Q1, Q4, Q6, Q11, Q13, Q14, and Q19, the TiFlash-only case performs better than the TiKV-only case, and the TiKV and TiFlash case gets the same performance as the TiFlash-only case. The reasons differ for these seven queries. Q1 and Q6 are mainly composed of aggregations on a single table, so running on the column store in TiFlash costs less time, and is an optimal choice. These results highlight the advantages of the columnar store described in previous work. Q4 and Q11 are separately executed using an identical physical plan in each case. However, scanning data from TiFlash is cheaper than TiKV, so the execution time in the TiFlash-only case is less, and it is also an optimal choice. Q13, Q14, and Q19 each contains a two-table join, which is implemented as a hash join. Although the TiKV-only case adopts the index reader when probing the hash table, it is also more expensive than scanning data from TiFlash.

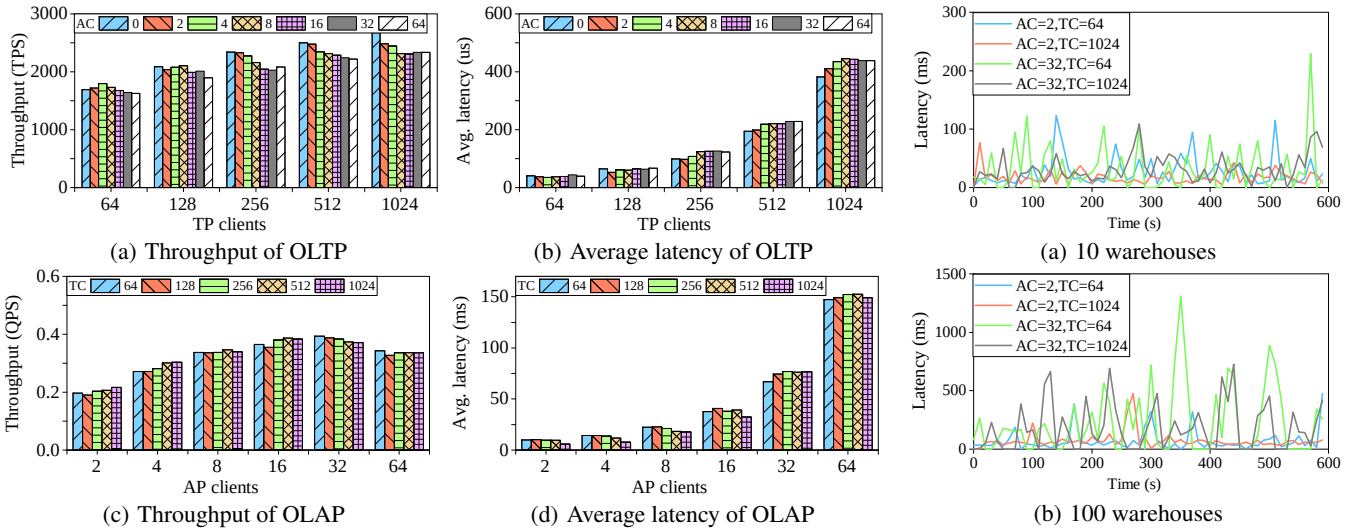
Q9 is a multi-join query. In the TiKV-only case, it takes index merge joins on some tables using indexes. It is cheaper than doing hash joins on TiFlash, so it becomes the optimal choice. Q7, Q20, and Q21 produce similar results, however they are elided due to limited space. The remaining eight of 22 TPC-H queries have comparable performance in the three storage settings.

In addition, we compare TiSpark to SparkSQL, PrestoDB, and Greenplum using the 22 analytical queries of CH-benCHmark with 500 warehouses. Each database is installed on six servers. For SparkSQL and PrestoDB, data is stored as columnar parquet files in Hive. Figure 9 compares the performance of these systems. TiSpark's performance is comparable to SparkSQL because they use the same engine. The performance gap is rather small and mainly comes from accessing different storage systems: scanning compressed parquet files is cheaper, so SparkSQL usually outperforms TiSpark. However, in some cases that advantage is offset where TiSpark can push more computation to the storage layer. Comparing TiSpark to PrestoDB and Greenplum is a comparison of SparkSQL (the underlying engine of TiSpark) to the other two engines. This is out of the scope of this paper, however, and we will not discuss it in detail.

6.4 HTAP Performance

As well as investigating transactional processing (TP) and analytical processing (AP) performance, we evaluate TiDB with a hybrid workload based on the entire CH-benCHmark, with separate transaction clients (TC) and analysis clients (AC). These experiments were conducted on 100 warehouses. The data is loaded into TiKV and simultaneously replicated into TiFlash. TiKV is deployed on three servers and is accessed by a TiDB SQL engine instance. TiFlash is deployed on three other servers and collocates with a TiSpark instance. This configuration services analytical and transactional queries separately. Each run is 10 minutes with a 3-minute warmup period. We measured the throughput and average latency of TP and AP workloads.

Figures 10(a) and 10(b) show the throughput and average latency (respectively) of transactions with various numbers of TP clients and AP clients. The throughput increases with more TP clients but reaches the maximum value at slightly less than 512 clients. With



(a) Throughput of OLTP (b) Average latency of OLTP (a) 10 warehouses

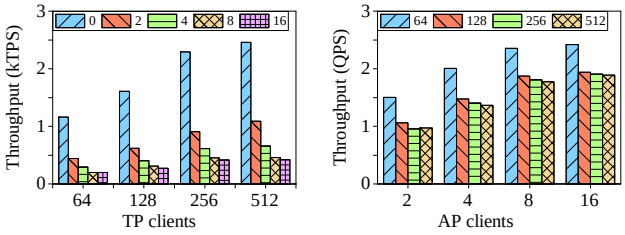
(c) Throughput of OLAP (d) Average latency of OLAP (b) 100 warehouses

Figure 10: HTAP performance of TiDB

Figure 11: Visibility delay of log replication

Table 4: The count distribution of visibility delay

W	Time (ms)	AC=2 TC=64	AC=2 TC=1024	AC=32 TC=64	AC=32 TC=1024
10	< 100	95.98%	96.22%	84.13%	90.96%
	< 500	99.43%	99.36%	98.98%	98.45%
	< 1000	99.62%	99.90%	99.74%	99.49%
100	< 100	82.98%	86.31%	48.51%	48.77%
	< 500	95.50%	97.69%	71.24%	72.06%
	< 1000	98.64%	99.58%	84.82%	87.17%



(a) The throughput of OLTP (b) The throughput of OLAP

Figure 12: HTAP performance of MemSQL

the same number of TP clients, more analytical processing clients degrade the TP throughput at most 10% compared to no AP clients. This confirms that the log replication between TiKV and TiFlash achieves high isolation, especially in contrast to the performance of MemSQL in Section 6.6. This result is similar to that in [24].

The average latency of transactions increases without an upper bound. This is because even though more clients issue more requests, they cannot be completed immediately and have to wait. The wait time accounts for the increasing latency.

Similar throughput and latency results shown in Figures 10(c) and 10(d) demonstrate the impact of TP on AP requests. The AP throughput soon reaches the maximum under 16 AP clients, because AP queries are expensive and compete for resources. Such contention decreases the throughput with more AP clients. With the same number of AP clients, the throughput remains almost the same, with at most only a 5% drop. This indicates that TP does not significantly affect AP execution. The increasing average latency of analytical queries results from more waiting time with more clients.

6.5 Log Replication Delay

To achieve real-time analytical processing, transactional updates should be immediately visible to TiFlash. Such data freshness is determined by the log replication delay between TiKV and TiFlash. We measure the log replication time while running CH-benCHmark with different numbers of transaction clients and analysis clients. We record that delay for every replication during 10 minutes of running CH-benCHmark, and compute the average delay every 10 seconds. We also compute the distribution of the log replication delay during the 10 minutes, shown in Table 4.

As shown in Figure 11(a), the log replication delay is always less than 300 ms on 10 warehouses, and most delays are less than 100 ms. Figure 11(b) shows that the delay increases with 100 warehouses; most are less than 1000 ms. Table 4 gives more precise details. With 10 warehouses, almost 99% of queries cost less than 500 ms, regardless of the client settings. With 100 warehouses, about 99% and 85% queries took less than 1000 ms with 2 and 32 analysis clients, respectively. These metrics highlight that TiDB can guarantee data freshness of about one second on HTAP workloads.

When comparing Figure 11(a) and Figure 11(b), we observe that the delay time is related to data size. The more warehouses, the larger the latency, because more data introduces more logs to be synchronized. In addition, the delay also depends on the number of analytical requests, but suffers less due to the number of transactional clients. This can be clearly seen in Figure 11(b). Thirty-two ACs cause more latency than two ACs. But with the same number of analytical clients, the latency does not differ a lot. We show more precise results in Table 4. With 100 warehouses and two ACs, more than 80% of queries take less than 100 ms, but with 32 ACs less than 50% take less than 100 ms. This is because more analytical queries induce log replication with a higher frequency.

6.6 Comparison to MemSQL

We compare TiDB with MemSQL 7.0 using CH-benC -Hmark. This experiment aims to highlight the isolation problem of state-of-the-art HTAP systems, rather than OLTP and OLAP performance. MemSQL is a distributed, relational database that handles both transactions and real-time analytics at scale. MemSQL is deployed on six servers: one master, one aggregator, and four leaves. We loaded 100 warehouses into MemSQL and ran the benchmark with various numbers of AP and TP clients. The benchmark ran for 10 minutes with a five-minute warm up period.

In contrast to Figure 10, Figure 12 illustrates that workload interference has a significant effect on the performance of MemSQL. In particular, as the number of AP clients increase, the transaction

throughput significantly slows, dropping by more than five times. The AP throughput also decreases with more TP clients, but such an effect is not as marked, because transaction queries do not require the massive resources of analytical queries.

7. RELATED WORK

Common approaches for building HTAP systems are: evolving from an existing database, extending an open source analytical system, or building from scratch. TiDB is built from scratch and differs from other systems in architecture, data origination, computation engines, and consistency guarantees.

Evolving from an existing database. Mature databases can provide HTAP solutions based on existing products, and they especially focus on accelerating analytical queries. They take custom approaches to separately achieve data consistency and high availability. In contrast, TiDB naturally benefits from the log replication in the Raft to achieve data consistency and high availability.

Oracle [19] introduced the Database In-Memory option in 2014 as the industry's first dual-format, in-memory RDBMS. This option aims to break performance barriers in analytic query workloads without compromising (or even improving) performance of regular transactional workloads. The columnar storage is a read-only snapshot, consistent at a point in time, and it is updated using a fully-online repopulation mechanism. Oracle's later work [27] presents the high availability aspects of its distributed architecture and provides fault-tolerant analytical query execution.

SQL Server [21] integrates two specialized storage engines into its core: the Apollo column storage engine for analytical workloads and the Hekaton in-memory engine for transactional workloads. Data migration tasks periodically copy data from the tail of Hekaton tables into the compressed column store. SQL Server uses column store indexes and batch processing to efficiently process analytical queries, utilizing SIMD [15] for data scans.

SAP HANA supports efficiently evaluating separate OLAP and OLTP queries, and uses different data organizations for each. To scale OLAP performance, it asynchronously copies row-store data to a columnar store distributed on a cluster of servers [22]. This approach provides MVCC data with sub-second visibility. However, it requires a lot of effort to handle errors and keep data consistent. Importantly, the transactional engine lacks high availability because it is only deployed on a single node.

Transforming an open-source system. Apache Spark is an open-source framework for data analysis. It needs a transactional module to achieve HTAP. Many systems listed below follow this idea. TiDB does not deeply depend on Spark, as TiSpark is an extension. TiDB is an independent HTAP database without TiSpark.

Wildfire [10, 9] builds an HTAP engine based on Spark. It processes both analytical and transactional requests on the same columnar data organization, i.e., Parquet. It adopts last-write-wins semantics for concurrent updates and snapshot isolation for reads. For high availability, shard logs are replicated to multiple nodes without help from consensus algorithms. Analytical queries and transactional queries can be processed on separate nodes; however, there is a noticeable delay in processing the newest updates. Wildfire uses a unified multi-version and multi-zone indexing method for large-scale HTAP workloads [23].

SnappyData [25] presents a unified platform for OLTP, OLAP, and stream analytics. It integrates a computational engine for high throughput analytics (Spark) with a scale-out, in-memory transactional store (GemFire). Recent updates are stored in row format, and then age into a columnar format for analytical queries. Transactions follow a 2PC protocol using GemFire's Paxos implementation to ensure consensus and a consistent view across the cluster.

Building from scratch. Many new HTAP systems have investigated different aspects of hybrid workloads, which include utilizing in-memory computing to improve performance, optimal data storage, and availability. Unlike TiDB, they cannot provide high availability, data consistency, scalability, data freshness, and isolation at the same time.

MemSQL [3] has an engine for both scalable in-memory OLTP and fast analytical queries. MemSQL can store database tables either in row or column format. It can keep some portion of data in row format and convert it to columnar format for fast analysis when writing data to disks. It compiles repeat queries into low-level machine code to accelerate analytic queries, and it uses many lock-free structures to aid transactional processing. However, it cannot provide isolated performance for OLAP and OLTP when running HTAP workloads.

HyPer [18] used the operating system's fork system call to provide snapshot isolation for analytical workloads. Its newer versions adopt an MVCC implementation to offer serializability, fast transaction processing, and fast scans. ScyPer [26] extends HyPer to evaluate analytical queries at scale on remote replicas by propagating updates either using a logical or physical redo log.

BatchDB [24] is an in-memory database engine designed for HTAP workloads. It relies on primary-secondary replication with dedicated replicas, each optimized for a particular workload type (i.e., OLTP or OLAP). It minimizes load interaction between the transactional and analytical engines, thus enabling real-time analysis over fresh data under tight SLAs for HTAP workloads. Note that it executes analytical queries on row-format replicas and does not promise high availability.

Lineage-based data store (L-Store) [35] combines real-time analytical and transactional query processing within a single unified engine by introducing an update-friendly, lineage-based storage architecture. The storage enables a contention-free update mechanism over a native, multi-version columnar storage model in order to lazily and independently stage stable data from a write-optimized columnar format into a read-optimized columnar layout.

Peloton [31] is a self-driving SQL database management system. It attempts to adapt data origination [8] for HTAP workloads at run time. It uses lock-free, multi-version concurrency control to support real-time analytics. However, it is a single-node, in-memory database by design.

Cockroach DB [38] is a distributed SQL database which offers high availability, data consistency, scalability, and isolation. Like TiDB it is built on top of the Raft algorithm and supports distributed transactions. It offers a stronger isolation property: serializability, rather than snapshot isolation. However, it does not support dedicated OLAP or HTAP functionality.

8. CONCLUSION

We have presented a production-ready, HTAP database: TiDB. TiDB is built on top of TiKV, a distributed, row-based store, which uses the Raft algorithm. We introduce columnar learners for real-time analysis, which asynchronously replicate logs from TiKV, and transform row-format data into column format. Such log replication between TiKV and TiFlash provides real-time data consistency with little overhead. TiKV and TiFlash can be deployed on separate physical resources to efficiently process both transactional and analytical queries. They can be optimally chosen by TiDB to be accessed when scanning tables for both transactional and analytical queries. Experimental results show TiDB performs well under an HTAP benchmark, CH-benCHmark. TiDB provides a generic solution to evolve NewSQL systems into HTAP systems.

9. REFERENCES

[1] Clickhouse. https://clickhouse.tech.

[2] LZ4. https://github.com/lz4/lz4.

[3] MemSQL. https://www.memsql.com.

[4] Parquet. https://parquet.apache.org.

[5] RocksDB. https://rocksdb.org.

[6] Sysbench. https://github.com/akopytov/sysbench.

[7] TiDB. https://github.com/pingcap/tidb.

[8] J. Arulraj, A. Pavlo, and P. Menon. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*, pages 583–598. ACM, 2016.

[9] R. Barber, C. Garcia-Arellano, R. Grosman, R. Müller, et al. Evolving Databases for New-Gen Big Data Applications. In *CIDR*. www.cidrdb.org, 2017.

[10] R. Barber, M. Huras, G. M. Lohman, C. Mohan, et al. Wildfire: Concurrent Blazing Data Ingest and Analytics. In *SIGMOD*, pages 2077–2080. ACM, 2016.

[11] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, 2010.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, pages 205–218. USENIX Association, 2006.

[13] R. L. Cole, F. Funke, L. Giakoumakis, W. Guy, et al. The mixed workload CH-benCHmark. In *DBTest 2011*, page 8. ACM, 2011.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, et al. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.

[15] Z. Fang, B. Zheng, and C. Weng. Interleaved Multi-Vectorizing. *PVLDB*, 13(3):226–238, 2019.

[16] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full Circle Back to Shared-Nothing Database Architectures. *PVLDB*, 7(12):1295–1306, 2014.

[17] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

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

[19] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, et al. Oracle Database In-Memory: A dual format in-memory database. In *ICDE*, pages 1253–1258. IEEE Computer Society, 2015.

[20] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[21] P. Larson, A. Birka, E. N. Hanson, W. Huang, M. Nowakiewicz, and V. Papadimos. Real-Time Analytical Processing with SQL Server. *PVLDB*, 8(12):1740–1751, 2015.

[22] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, W. Han, C. G. Park, H. J. Na, and J. Lee. Parallel Replication across Formats in SAP HANA for Scaling Out Mixed OLTP/OLAP Workloads. *PVLDB*, 10(12):1598–1609, 2017.

[23] C. Luo, P. Tözün, Y. Tian, R. Barber, et al. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *EDBT*, pages 1–12. OpenProceedings.org, 2019.

[24] D. Makreshanski, J. Giceva, C. Barthels, and G. Alonso. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD*, pages 37–50. ACM, 2017.

[25] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. SnappyData: A Unified Cluster for Streaming, Transactions and Interactive Analytics. In *CIDR*. www.cidrdb.org, 2017.

[26] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: A Hybrid OLTP&OLAP Distributed Main Memory Database System for Scalable Real-Time Analytics. In *DBIS*, volume P-214 of *LNI*, pages 499–502. GI, 2013.

[27] N. Mukherjee, S. Chavan, M. Colgan, M. Gleeson, X. He, et al. Fault-tolerant real-time analytics with distributed Oracle Database In-memory. In *ICDE*, pages 1298–1309. IEEE Computer Society, 2016.

[28] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, 1996.

[29] D. Ongaro and J. K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*, pages 305–319. USENIX Association, 2014.

[30] F. Özcan, Y. Tian, and P. Tözün. Hybrid Transactional/Analytical Processing: A Survey. In *SIGMOD*, pages 1771–1775. ACM, 2017.

[31] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, et al. Self-Driving Database Management Systems. In *CIDR*. www.cidrdb.org, 2017.

[32] A. Pavlo and M. Aslett. What's Really New with NewSQL? *SIGMOD*, 45(2):45–55, 2016.

[33] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, pages 251–264. USENIX Association, 2010.

[34] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, and K. Sattler. Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling. In *TPCTC*, volume 8904, pages 97–112. Springer, 2014.

[35] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim. L-Store: A Real-time OLTP and OLAP System. In *EDBT*, pages 540–551. OpenProceedings.org, 2018.

[36] S. Sivasubramanian. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *SIGMOD*, pages 729–730. ACM, 2012.

[37] M. Stonebraker and U. Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract). In *ICDE*, pages 2–11. IEEE Computer Society, 2005.

[38] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, et al. CockroachDB: The Resilient Geo-Distributed SQL Database. In *SIGMOD*, pages 1493–1509. ACM, 2020.