Cloud-Native Transactions and Analytics in SingleStore

Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, Nikita Shamgunov
SingleStore
{adam,szupo,joseph,zhou,eli,jack,evan,hanson,rob,rodrigo,nikita}@singlestore.com

ABSTRACT

The last decade has seen a remarkable rise in specialized database systems. Systems for transaction processing, data warehousing, time series analysis, full-text search, data lakes, in-memory caching, document storage, queuing, graph processing, and geo-replicated operational workloads are now available to developers. A belief has taken hold that a single general-purpose database is not capable of running varied workloads at a reasonable cost with strong performance, at the level of scale and concurrency people demand today. There is value in specialization, but the complexity and cost of using multiple specialized systems in a single application environment is becoming apparent. This realization is driving developers and IT decision makers to seek databases capable of powering a broader set of use cases when looking to adopt a new database. Hybrid transaction and analytical (HTAP) databases have been developed to try to tame some of this chaos.

In this paper we introduce SinglestoreDB (S2DB), formerly called MemSQL, a distributed general-purpose SQL database designed to have the versatility to run both operational and analytical workloads with good performance. It was one of the earliest distributed HTAP databases on the market. It can scale out to efficiently utilize 100s of hosts, 1000s of cores and 10s of TBs of RAM while still providing a user experience similar to a single-host SQL database such as Oracle or SQL Server. S2DB's unified table storage runs both transactional and analytical workloads efficiently with operations like fast scans, seeks, filters, aggregations, and updates. This is accomplished through a combination of rowstore, columnstore and vectorization techniques, ability to seek efficiently into a columnstore using secondary indexes, and using in-memory rowstore buffers for recently modified data. It avoids design simplifications (i.e., only supporting batch loading, or limiting the query surface area to particular patterns of queries) that sacrifice the ability to run a broad set of workloads.

Today, after 10 years of development, S2DB runs demanding production workloads for some of the world's largest financial, telecom, high-tech, and energy companies. These customers drove the product towards a database capable of running a breadth of workloads across their organizations, often replacing two or three different databases with S2DB. The design of S2DB's storage, transaction processing, and query processing were developed to maintain this versatility.

SIDMOD'22, June 12-17, 2022, Philadelphia, PA USA

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3526055

CCS CONCEPTS

Information systems~Data management systems~Database management system engines~DBMS engine architectures

KEYWORDS

Databases, Distributed Systems, Separation of storage and Compute, Transactions and Analytics

ACM Reference format:
Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, & Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore In *Proceedings of the 2022 Int'l Conference on Management of Data (SIGMOD'22), June 12-17, 2022. Philadelphia, PA, USA.* ACM, NY, NY, USA. 13 pages. https://doi.org/10.1145/3514221.3526055

1 Introduction

The market is saturated with specialized database engines. As of January 2022, DB-Engines [18] ranks over 350 different databases. Amazon Web Services alone supports 15+ different database products[1]. There is value in special-case systems [2], but when applications end up built as a complex web of different databases a lot of that value is eroded. Developers are manually rebuilding the general-purpose databases of old via ETL and data flows between specialized databases.

We believe two industry trends have driven this proliferation of new databases. The first trend is the shift to cloud-native architectures designed to take advantage of elastic cloud infrastructure. Cloud blob stores (S3 [3]) and block storage (EBS [44]) allow databases to tap into almost limitless, highly-available and durable data storage. Elastic compute instances (EC2 [4]) allow databases to bring more compute to bear at a moment's notice to deal with a complex query or a spike in throughput. The second trend is the demand from developers to store more data and access it with lower latency and with higher throughput. Modern applications generate a lot of data. This performance and data capacity requirement is often combined with a desire for flexible data access. These access patterns are application-specific but can range from low-latency, high-throughput writes (including updates) for real-time data loading and deduplication, to efficient batch loading and complex

analytical queries over the same data. Application developers have never been more demanding of databases.

A common approach to tackle these requirements is to use a domain-specific database for different components of an application. In contrast, we believe it is possible to design a database that can take advantage of elastic cloud infrastructure while satisfying a breadth of requirements for transactional and analytical workloads. There are many benefits for users in having a single integrated, scalable database that can handle many application types. These include: reduced training requirements for developers, reduced need to move and transform data, reduction in the number of copies of data that must be stored and resulting reduction in storage costs, reduced software license costs, and reduced hardware costs. Furthermore, S2DB enables modern workloads to provide interactive real-time insights and decision-making, enabling both high-throughput low-latency writes and complex analytical queries over ever-changing data, with end-to-end latency of seconds to sub-seconds from new data arriving to analytical results. This outcome is difficult to achieve with multiple domain specific databases.

Moreover, adding incrementally more functionality to cover different use cases with a single distributed DBMS leverages existing fundamental qualities that any distributed data management system needs to provide. This yields more functionality per unit of engineering effort on the part of the vendor, contributing to lower net costs for the customer. For example, specialized scale-out systems for full-text search may need cluster management, transaction management, high availability, and disaster recovery, just like a scale-out relational system requires. Some specialized systems may forgo some of these capabilities for expediency, compromising reliability.

This paper introduces the architecture of the SingleStore database engine, a cloud-native database that excels at running complex interactive queries over large datasets (100s of terabytes) as well as running high-throughput, low-latency read and write queries with predictable response times (millions of rows written or updated per second). The same SingleStore database engine is used both in SingleStore Managed Service, a cloud database service, and in the SingleStoreDB (S2DB) database product, which can be installed wherever desired. In the rest of the paper, we'll simply refer to the SingleStore database engine as S2DB.

S2DB can support a breadth of workloads over disaggregated storage by pushing only cold data to blob storage and making intelligent use of local state when running queries to minimize network use. The rest of this paper expands on other important design decisions made while building S2DB. We believe our design represents a good trade-off between efficiency and flexibility and can help simplify application development by avoiding complex data pipelines.

This paper presents two key components of S2DB that are important for cloud-native transactional and analytical workloads [42].

Separation of storage and compute

S2DB is able to make efficient use of the cloud storage hierarchy (local memory, local disks, and blob storage) based on data hotness. This is an obvious design, yet most cloud data warehouses that support using blob storage as a shared remote disk don't do it for newly written data. They force new data for a write transaction to be written out to blob storage before that transaction can be considered committed or durable [26, 27, 30]. This in effect forces hot data to be written to the blobstore harming write latency. S2DB can commit on local disk and push data asynchronously to blob storage. This gives S2DB all the advantages of separation of storage and compute without the write latency penalty of a cloud data warehouse. For example, S2DB:

- Can store more data than fits on local disks by keeping cold data in blob storage and only the working set of recently queried data on local disks.
- Stores history in blob storage (deleted data can be retained). This enables point-in-time restores to points in the past without needing to take explicit backups or copy any data on a restore.
- Can provision multiple read-only replicas of a database from blob storage without any impact to the read-write master copy of the database. The read-only replicas are created on their own hosts (called a workspace in S2DB) and can be attached and detached to the workspace on demand. This allows S2DB to support OLTP and OLAP workloads over the same data but using isolated compute for each.

Unified table storage

S2DB tables support transactions that need both the scan performance of a columnstore (scanning 100s of millions to trillions of rows in a second[49]) and the seek performance of rowstore indexes to speed up point reads and writes. In S2DB, both OLAP and OLTP workloads use a single unified table storage design. Data doesn't need to be copied or replicated into different data layouts as other HTAP systems often do [23].

S2DB's unified table storage internally makes use of both rowstore and columnstore formats, but end users need not be aware of this. At a high level, the design is that of a columnstore with modifications to better support selective reads and writes in a manner that has very little impact on the columnstore's compression and table scan performance. The columnstore data is organized as a log-structured merge tree (LSM) [8], with secondary hash indexes supported to speed up OLTP workloads. Unified tables support sort keys, secondary keys, shard keys, unique keys and row-level locking, which is an extensive and unique set of features for table storage in columnstore format. Unified table storage is also sometimes referred to as universal storage [39] for its ability to handle a universal set of workloads.

The rest of the paper is structured as follows. Section 2 gives a brief overview of how a S2DB cluster functions: how it distributes data, maintains high availability and runs queries. Section 3 describes how S2DB separates storage and compute without sacrificing support for low-latency writes. Section 4 details the design of our unified table storage, which we believe is ideal for HTAP. Section 5 describes how query execution

adapts to tradeoffs between different data access methods on the unified table storage. Section 6 shows some experimental results using industry-standard benchmarks to demonstrate that S2DB is competitive with both operational and analytical databases on benchmarks specific to each workload.

2 Background on SingleStoreDB

SingleStoreDB is a horizontally-partitioned, shared-nothing DBMS [35] which is optionally able to use shared storage such as a blob store for cold data. An S2DB cluster is made up of aggregator nodes, which coordinate queries, and leaf nodes, which hold copies of partitions of data and are responsible for the bulk of compute for queries. Each leaf holds several partitions of data. Each partition is either a master which can serve both reads and writes, or a replica which can only serve reads.

Tables are distributed across partitions by hash-partitioning of a user-configurable set of columns called the shard key. This enables fast query execution for point reads and query shapes that do not require moving data between leaves. When join conditions or group-by columns match their referenced tables' shard keys, S2DB pushes down execution to individual partitions avoiding any data movement. Otherwise, SingleStore redistributes data during query execution, performed as a broadcast or reshuffle operation, as described in [46]. S2DB's query processor is able to run complex analytical queries such as those in the TPC-H and TPD-DS benchmarks competitively with cloud data warehouses [19, 46].

SingleStore also supports full-query code generation targeting LLVM [40] through intermediate bytecode. LLVM compilation happens asynchronously while the query begins running via a bytecode interpreter. The compiled LLVM code is hotswapped in during query execution when compilation completes. Using native code-generation to execute queries reduces the instructions needed to run a query compared to the more typical hand-built interpreters in other SQL databases [41]. The details of S2DB's query compilation pipeline are omitted from this paper.

S2DB maintains high availability (HA) by storing multiple replicas of each partition on different nodes in the cluster. By default, data is replicated synchronously to the replicas as transactions commit on the master partitions. Read queries never run on HA replicas, they exist only for durability and availability. Queries only run on master partitions or specifically created read replicas. Since HA replicas exist on the same set of hosts that store masters for other partitions (see Figure 2), using HA replicas to run queries wouldn't help spread the load across the cluster (the same hosts are already busy running queries). S2DB does support the creation of read replicas for scaling out queries on other hosts without any master partitions, the details of which are described in section 3.3. HA replicas are hot copies of the data on the master partition such that a replica can pick up the query workload immediately after a failover without needing any warm up. Failovers and auto-healing are coordinated by a special aggregator node called the master aggregator. If a node stops responding to heartbeats for long enough then replica partitions for any of its master partitions will be promoted to master and take over running queries.

S2DB also supports the creation of asynchronously replicated replicas in different regions or data centers for disaster recovery. These cross-region replicas can act as another layer of HA in the event of a full region outage. They are queryable by read queries by default so can also be used to scale out reads. Failovers across regions are not automated in S2DB today, they must be triggered by a DBA.

2.1 Table storage formats

S2DB uses two storage types internally: an in-memory rowstore backed by a lockfree skiplist [6], and a disk-based columnstore [5]. In early versions of S2DB, users had to choose either storage type on a per-table basis according to the workload characteristics. Unified table storage (described in section 4) combines both formats internally to support OLAP and OLTP workloads using a single storage design.

2.1.1 Rowstore storage

Each index in an S2DB in-memory rowstore table uses a lockfree skiplist to index the rows. A node in the skiplist corresponds to a row, and each node stores a linked list of versions of the row to implement multiversion concurrency control so that readers don't need to wait on writers. Writes use pessimistic concurrency control, implemented using row locks stored on each skiplist node to handle concurrent writes to the same row. Each version of the row is stored as a fixed-sized struct (variable-length fields are stored as pointers) according to the table schema, along with bookkeeping information such as the timestamp and the commit status of the version.

In addition to writing to the in-memory skiplists, write operations also write the affected rows to a log before committing. A log is created for each database partition, and it's persisted to disk and replicated to guarantee the durability of writes. On node restarts, the state of each database partition is recovered by replaying the writes in the persisted log. A background process periodically creates a snapshot file containing the serialized state of the in-memory rowstore tables at a particular log position. This allows the recovery process to start replay from the latest snapshot's log position to limit recovery time.

2.1.2 Columnstore storage

The data in a columnstore table is organized into segments, where each segment stores a disjoint subset of rows as a set of data files on disk. Within a segment, each column is stored in the same row order but compressed separately. Common encodings like bit packing, dictionary, run-length encoding, and LZ4 are supported for column compression. The same column can use a different encoding in each segment optimized for the data specific to that segment. The segment metadata is stored in a durable in-memory rowstore table (described in section 2.1.1), containing information including the file locations, the encodings, and the min/max values for each column.

Additionally, a bit vector is stored within the segment metadata to represent the deleted rows in the segment.

This representation is mostly optimized for OLAP, however, several considerations were made to speed up point read operations commonly found in OLTP workloads. The column encodings are each implemented to be seekable to allow efficient reads at a specific row offset without decoding all the rows. Storing min/max values allows segment elimination to be performed using in-memory metadata to skip fetching segments with no row matched.

A sort key can be specified on each columnstore table to allow more efficient segment elimination. If specified, rows are fully sorted by the sort key within each segment. The sort order across segments is maintained similar to LSM trees [8, 5] by building up sorted runs of segments. A background merger process is used to merge the segments incrementally to maintain a logarithmic number of sorted runs.

For each columnstore table, S2DB creates a rowstore table as a write-optimized store to store small writes and avoid creating small sorted runs across many files. This corresponds to the level 0 storage in other LSM trees, like MemTable in RocksDB [10]. A background flusher process periodically deletes rows from the rowstore and converts those rows into a columnstore segment in a transaction. For read performance, this write-optimized store is kept small relative to the table size. Since the background merger and flusher processes can move rows between the rowstore and different segments, reads need to use partition-local snapshot isolation to guarantee a consistent view of the table.

Columnstore tables support vectorized execution [9], and, for some filter, group-by and hash join operations, encoded execution [7]. S2DB vectorized execution uses late materialization, only decoding columns if data in them qualifies based on filters on other columns. Encoded execution can achieve large speedups on filtering and aggregation operations by operating directly on compressed data and using SIMD instructions when appropriate.

3 Separation of storage and compute

S2DB can run with and without access to a blob store for separated storage. When running without access to blob storage, S2DB behaves like a typical shared-nothing distributed database, where the nodes in the cluster are the source of truth. When running with a blob store, S2DB is atypical in that it doesn't store all persistent data on the blob store and only transient data on local storage. Instead, newly written data is only persisted on the local storage of the cluster and later moved to the blob store asynchronously. This design allows S2DB to commit transactions without the latency penalty of needing to write all the transaction data to blob storage to make it durable. By treating blob storage truly as cold storage, S2DB is able to support low-latency writes while still getting many of the benefits of separated storage (faster provisioning and scaling, storing datasets bigger than local disk, cheaper historical storage for point in time restores etc.). So in order to describe how S2DB separates storage, it's important to understand how S2DB's local or integrated durability and compute functions, as that local durability mechanism works in tandem with blob storage to maintain durability of committed transactions.

Durability is managed by the cluster on each partition using replication. The in-cluster replication is fast and log pages can be replicated out-of-order and replicated early without waiting for transaction commit. Replicating out-of-order allows small transactions to commit without waiting for big transactions, guaranteeing that commits have low and predictable latency. By default, data is considered committed when it is replicated in-memory to at least one replica partition for every master partition involved in a transaction. This means loss of a single node will never lose data, and if replication is configured across availability zones, loss of an entire availability zone will never lose data. If all copies of a partition are lost due to concurrent node failures, before any new HA replicas are provisioned, recently written data that was only present in-memory will be lost, but any data synced to local disk is recoverable as long as the local disk survived the failure (i.e., a database process crash). While SingleStore supports synchronously committing to local disk as well, this tradeoff often doesn't make sense in cloud environments where loss of a host often implies loss of the local storage attached to that host. For this reason, S2DB doesn't synchronously commit transactions to local disk by default.

Section 2 showed that the data for a partition of a columnstore table is stored in a LSM tree, where the top level is an in-memory rowstore, and lower levels are HTAP-optimized columnstore data files. To explain how the integrated durability

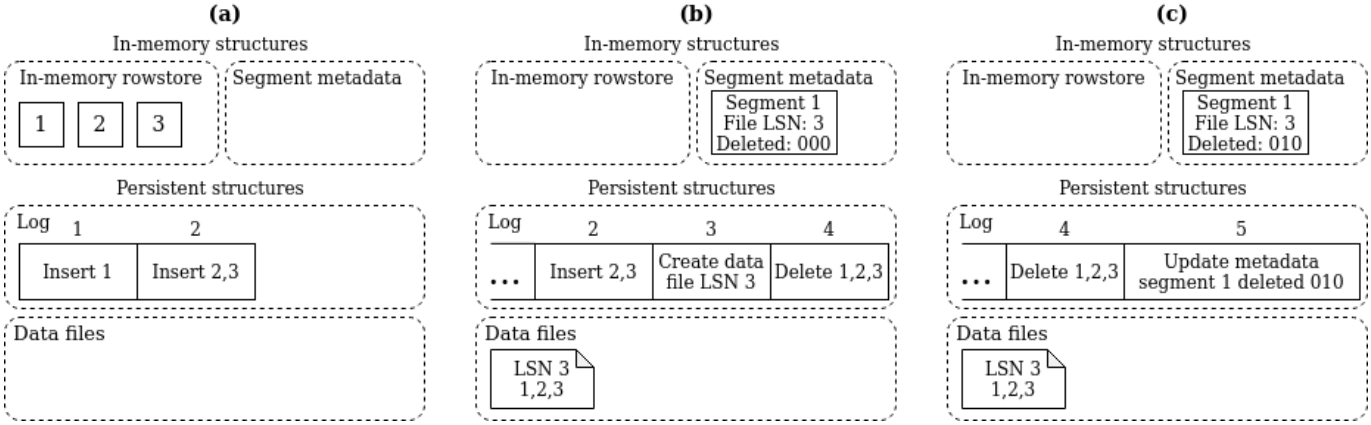

Figure 1: Example of a sequence of writes in S2, showing the state of the in memory and persisted structures in each step. (a) Inserting rows 1,2,3 in two transactions (b) Converting in-memory rows 1,2,3 to segment 1 (c) Deleting row 2 from segment 1.

mechanism works with the table storage, figure 1 presents the database state after a few example write operations. The bottom of figure 1 shows the persisted structures stored on disk, including the log and the physical columnstore data files, while the top shows the corresponding in-memory state at each step. Figure 1(a) shows two insert transactions made durable in the log and the rows inserted to the in-memory rowstore.

When enough rows are amassed in the in-memory rowstore, they will be converted into a columnstore segment by the flushing process described in section 2.1.2. As illustrated in figure 1(b), the flushing process creates data files to store the rows in a segment, and deletes the converted rows from the in-memory rowstore in the same transaction. Each data file is named after the log page at which it was created, so that data files can be considered as logically existing in the log stream, while physically being separate files. For larger write transactions that involve the creation of multiple data files, each file is replicated as soon as it's written on the master without need to wait for the transaction to commit.

Data files are immutable - to delete a row from a segment, only the segment metadata is updated to mark the row as deleted in the deleted bit vector. Figure 1(c) shows how the metadata change is logged for a delete (omitting for simplicity the move transaction described in section 4.2). The key observation is that the data file itself is immutable, but the metadata changes are logged, which will lend itself well to the separation of storage and compute.

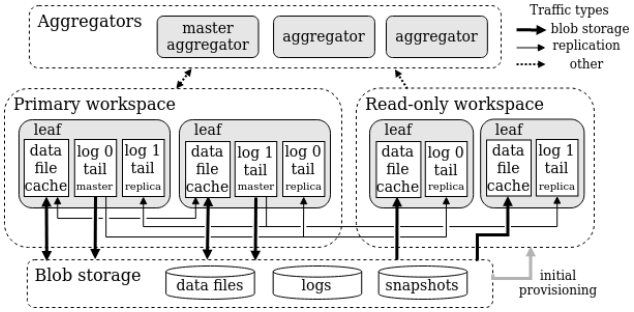

Figure 2: Cluster architecture with separated storage. The left side shows how the master workspace uploads data files, logs, and snapshots to blob storage asynchronously, while using replication to ensure durability of the log tails. The right side shows a read-only workspace provisioned from blob storage

3.1 Staging Data from Local to Remote Storage

As shown in the previous section, S2DB has durable, low-latency data storage in which all data for a given partition is recorded in a single log. The log is the only file which is ever updated (via appends). The data files containing columnstore data are immutable once written. This immutability is important for making use of blob storage because cloud blob stores typically don't support efficient file updates.

S2DB's separation of storage and compute design is shown in Figure 2 and can be summarized as follows:

- Transactions are committed to the tail of the log and replicated to other nodes just as when S2DB runs without blob storage available. Since the tail of the log is stored on the local disk and memory of the leaf nodes, no blob store writes are required to commit a transaction
- Newly committed columnstore data files are uploaded asynchronously to blob storage as quickly as possible after being committed. Hot data files are kept in a cache locally on disk for use by queries and cold data files are removed from local disk once uploaded.
- Transaction logs are uploaded to blob storage in chunks below a position in the log known to contain only fully durable and replicated data. The tail of the log newer than this position is still receiving active writes, thus these newer log pages are never uploaded to blob storage until replication advances the fully durable and replicated log position past them.
- Snapshots of rowstore data are taken only on master partitions and written directly to the blob store reducing local disk IO compared to S2DB running without blob storage. Replicas don't need to take their own snapshots. If a replica ever needs a snapshot (say because it was disconnected for a long period of time), it can get the snapshot from blob storage.
- To add more compute to the cluster, new replica databases get the snapshots and logs they need from blob storage and replicate the tail of the log (not yet in blob storage) from the master databases. Columnstore data files are pulled from the blob store on demand and stored in the data file cache as needed. This design allows new replicas to be provisioned quickly, as they don't need to download all data files before they can start acknowledging transactions or servicing read queries. This fast provisioning process allows pools of compute called workspaces to be created over the same set of databases as shown in Figure 2. These are discussed in more detail in section 3.2.

The biggest advantage of this design compared to the designs used by cloud data warehouses is that blob store writes are not needed to commit a transaction, so write latency is low and predictable. Since data files are uploaded to the blob store asynchronously and the working set of recently used data files are kept cached on local disks, short periods of unavailability in the blob store doesn't affect the steady-state workload, as long as reads happen within the cached working set. This allows S2DB to not be limited by the availability guarantee of the underlying blob store, which is usually much less than its durability guarantee. For example, Amazon S3 guarantees 11 nines of durability but only 3 nines of availability [45]. S2DBs design contrasts with many cloud data warehouses that need data to be written to the blobstore for it to be considered durable [26, 27, 30]. On the other hand, persisting to local disk and then moving the data to blob storage asynchronously does have some drawbacks compared to keeping all persistent data in the blob store. Our approach is more complex as keeping persistent state on local disk/memory requires a local high performance replication and recovery protocol. It's also less elastic. Adding or removing hosts requires moving the local data not yet in blob storage carefully to maintain durability and availability guarantees. In the event of multiple concurrent

failures, for instance loss of all nodes which have a copy of a single partition, committed data could be lost. This can be mitigated by having sync replicas spread across multiple availability zones, so losing data would require concurrent failures between availability zones or the loss of an entire region.

Cloud operational databases such as Amazon Aurora [28] don't use blob storage for system-of-record data at all, instead using their own separated storage or log services to make data durable and available. Blob storage is only used for backups. As a result, the maximum database size that Aurora can support is limited by what its storage service can support, currently 128 TB (it's not unlimited). It also means the expense associated with storing and accessing data is higher (Aurora storage is about 4 times as expensive as S3).This trade-off makes sense for a database targeting OTLP workloads, as the data sizes they deal with are typically smaller and prioritizing efficient availability and durability features is more important for OLTP.

3.2 Capabilities Enabled by Separated Storage

S2DB's separated storage design gives it many of the benefits expected of systems using shared remote storage. Even though S2DB's data separation relies on storing the tail of the log on local disk/memory, the typical capabilities of having the bulk of the database's data on remote storage still apply. Some example capabilities enabled by remote storage are:

- S2DB uses faster ephemeral SSDs for local storage instead of more expensive and slower network block storage (EBS) often used by other cloud databases [21]. Most of the data stored on local disks is cached, frequently-accessed data that is persisted in the blob store. The local disks are not responsible for persisting this data. The local disks are only responsible for persisting the tail of the log not yet in blob storage as described in section 3 and 3.1. Ephemeral SSDs can have multiple orders of magnitude higher IOPS than network block storage depending on the particular disks used, so this is a considerable boost in performance for workloads doing a lot of concurrent reads and writes.
- S2DB can keep months of history since it is cheap to store data at rest in blob storage. This history is used by a point in time restore (PITR) command to restore a database back to the state it was in at a given time in the past without the need to have taken an explicit backup at that time. The blob store acts as a continuous backup of the database. A PITR to a target time in the past runs by inspecting the versioning metadata stored in the log files in blob storage to find a transactionally consistent point in the log (called LP) for each partition that maps as closely as possible to the given PITR target wall clock time. It then drops the existing local state of the database, and does a restore up until the log position LP for each partition in the same fashion as when recovering data from blob storage on a process restart. That is, it fetches and replays the data from the first snapshot file before LP in the log stream and then fetches and replays any logs after the snapshot until LP is reached. Note that today S2DB doesn't support querying at a specific point in time, sometimes called time travel querying, only a full restore of the database state via PITR is supported.
- S2DB supports the creation of read-only workspaces which are a set of hosts that replicate recently written data asynchronous from the primary writable workspace, but which don't participate in acking commits for durability, as shown in figure 2. Read-only workspaces can be used to scale out the cluster on demand to handle an increase in read query concurrency by directing some of the read workload to the new workspace. Depending on the number of hosts in the workspace and the amount of data being actively queried, they can often be created within minutes. They also create an isolated environment to run heavy analytical workloads without impacting a more mission-critical read/write workload running on the primary workspace. Data files other than the recently written ones that are replicated to the workspace are read from the blob store directly rather than from the primary workspace, so that each workspace can cache its own set of data independently.

In conclusion, S2DB's design for separation of storage and compute gives it many of the durability and elasticity benefits of traditional cloud data warehouse designs. Specifically, flexible options for pausing, resuming and scaling compute, as well as access to practically unlimited durable storage that scales independently of compute and that can be used for consistent point in time restores. However, S2DB's integrated durability and compute means it does not sacrifice write latency, making our storage design suitable for both analytic workloads and transactional workloads.

4 Unified table storage

As a storage engine built for HTAP, S2DB table storage needs to work well in a wide range of workloads. In many situations, we observed that choosing between rowstore and columnstore storage formats was a hard decision for users. It required the users to identify whether the access patterns of each table lean OLTP or OLAP, creating friction when developing applications. This was especially true for workloads having both OLTP and OLAP aspects on the same tables, like real-time analytic use cases running analytics concurrently with high-concurrency point reads and writes. Therefore, we designed the unified table storage with the foremost goal of providing a unified table type that works well both for OLTP and OLAP access. This eliminates the burden on users to choose the data layout suitable for their particular workload. Furthermore, it allows demanding HTAP workloads to work efficiently without the complexity of managing data movements across tables serving different parts of the workloads.

To quantify some of the benefits we wanted to achieve with unified table storage, we sought to allow customers who were using a UNION ALL view of a rowstore (storing recent data for uniqueness enforcement) and a columnstore (storing older data for efficient analytics) to just use one unified storage table. This replaces three DDL statements with just one, and eliminates application code to move data from rowstore to columnstore as it ages. Analytical query performance also improves, often by several

times, because UNION ALL plans are harder to optimize and execute than plans over a regular table.

To achieve this, unified table storage extends the columnstore storage described in 2.1.2. S2DB columnstore storage is LSM-tree based, which writes data in large consecutive chunks and works well with tiered storage. The column-oriented data format is well known to be ideal for OLAP workloads [31]. Therefore, the main problem to solve here is making it work efficiently for OLTP use cases without sacrificing its OLAP performance. In particular, S2DB unified table storage maintains these properties crucial to the performance of the columnstore storage:

- **No merge-based reconciliation during reads**

Common LSM tree implementations, such as RocksDB [10], Cassandra [33], and BigTable [34], use tombstone entries to represent deletes in the LSM tree. Under this representation, reads need to reconcile results from all LSM levels to read the latest data. S2DB columnstore storage avoids this reconciliation process for reads, since merging would introduce a significant per-row overhead on analytical queries (for comparison, the total processing time for TPCH query 1 can be as low as 8.6 clock cycles per-row using vectorized execution on encoded data [7]). Instead, S2DB represents deletes using a bit vector stored as part of the segment metadata, which is cheaper to apply on the data files for the segment to filter out deleted rows compared to merging all LSM tree levels.

- **Minimize disk access and blocking during writes**

Similar to common LSM tree implementations, S2DB columnstore storage performs streaming inserts on an in-memory write-optimized store to achieve low latency writes. Different from other LSM trees, for update and delete operations S2DB needs to modify the segment metadata in addition to the in-memory store, due to the design decision to avoid tombstone records. While update and delete operations still minimize disk writes, extra care (section 4.2) is required to avoid blocking from concurrent modifications on the same segment.

4.1 Secondary indexes

Secondary indexes are important for efficient point access in transactional workloads. Some commonly adopted indexing approaches for LSM tree implementations are:

Per-segment filtering structure - Building bloom filters or inverted indexes for each on-disk segment to allow skipping the individual segments when there isn't a match [12], like RocksDB [10], Procella [11], Cassandra SASI index [13], and many other LSM tree implementations [37].

External index structure - Having an index structure (e.g. a B-tree or a separate LSM tree) outside of the LSM tree used for table storage, mapping the secondary index columns to the values of the primary index columns, like Spanner [32] and WiredTiger [36]. Note that this approach is used for non-LSM tree storage engines as well, such as InnoDB [15]. SQL server [14] uses the position of the row in the columnstore instead of the primary index columns, but it also requires an external mapping index to find the current position of the row from its original position.

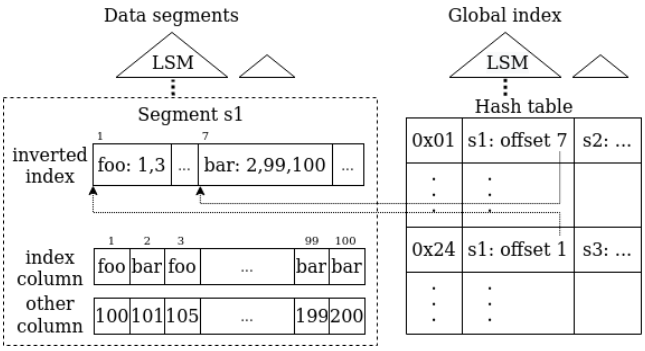

Figure 3: Two-level secondary index structure. A segment and a global hash table from the corresponding LSM trees are shown here

Differing from the common approaches, S2DB secondary indexes use a two-level structure integrated with the LSM tree storage, as illustrated in figure 3.

- For each segment, an inverted index is built to map values of the indexed column to a postings list, which stores row offsets in the segment with that value.
- Across segments in the table, a global index is used to map values of the indexed column to the ids of the segments with that value, along with the starting location of the corresponding postings list in the inverted index for each segment.

The per-segment inverted index is similar to other LSM tree implementations. Since the content of each segment is immutable, the per-segment inverted index gets built once when the segment is created and stays unchanged. The global index allows reads to query the inverted indexes only on segments with the target value. Since the starting location of the posting list is stored inline in the global index, this doesn't introduce an extra indirection during the lookup.

The global index can be implemented with different data structures to support different index types. Currently S2DB supports unordered secondary indexes, where the global index uses a special LSM tree storing an immutable hash table at each level. When a segment is created, a corresponding hash table gets created for the secondary index, covering values from the segment. Over time, the hash tables for different segments get merged together using the LSM tree merging algorithm, creating larger hash tables covering multiple segments. Segment deletions are handled lazily in the global index -- reads simply skip the references to deleted segments in the global hash tables, and the LSM tree merging process later rewrites the hash table if at least half of the segments it covers are deleted. Lazy segment deletion allows the index maintenance to happen independently from the data segments, so that secondary indexes won't become a point of contention on writes. In the future, we plan to support ordered secondary indexes by using a regular ordered LSM tree as the global index structure.

Since the global index is implemented as a LSM tree, it introduces an extra O(log(N)) factor in write amplifications due to the merges. An optimization to minimize the write cost is that

S2DB stores only the hashes, not the column values in the global hash tables. The column values are instead stored in the per-segment inverted indexes. This reduces the write cost significantly in cases with wide columns (e.g. when indexing on strings) since the per-segment inverted indexes don't go through the merges on the global index. Furthermore, the global index only stores information about the unique values in each segment, so its write cost is minimal when the index column contains only a few distinct values.

Compared with the per-segment filtering structure approach, this implementation has significant advantage on point reads, since the number of lookups required is O(log(N)) (checking each hash table in the global index) instead of O(N) (checking the index or bloom filter per segment). The drawback is having an extra O(log(N)) factor in the write cost, which we found to be an acceptable tradeoff for efficient index lookups.

Compared with the external index structure approach, this implementation has advantage on reads, since it avoids the cost of performing a LSM tree lookup per matched row for finding the row in the primary LSM tree storage. The main difference here is that this implementation stores the physical row offsets instead of the primary key value. This advantage is particularly significant when there are many matched rows for the same secondary key value. The drawback is having extra write cost when merging happens on the primary LSM tree, since merges change the physical row offsets, which then creates a new hash table in the global index. On the other hand, this extra write cost is minimal when the primary LSM tree rarely needs to perform merges, e.g. when the table has no sort key, or the rows are inserted in the sort key order.

The per-segment inverted index in this design allows the simultaneous use of multiple indexes when filtering on a boolean expression of multiple indexed columns. Lookup results from different indexes can be combined efficiently by merging the postings lists [43], so that only the exact set of rows passing all index filters gets scanned. S2DB's postings list format supports forward seeking, so that sections in a long postings list can be skipped during the merge, if postings lists from the other indexes already guarantee that no match is present in the section.

4.1.1 Multi-column secondary index

To support multi-column secondary indexes while minimizing storage costs, S2DB builds a secondary index for each indexed column, and allows the single-column indexes to be shared across multiple indexes referring to the same columns. For example, a secondary index on columns (a, b, c) builds the following data structures:

1. Per-segment inverted indexes on each of the columns a, b, c
2. Global indexes on each of the columns a, b, c.
3. A global index on the tuple of indexed columns (a, b, c), mapping from the hash of each tuple (value_a, value_b, value_c) to the starting locations of the corresponding per-column postings lists for value_a, value_b, and value_c.

The per-column data structures (1) and (2) are the same as single-column secondary indexes. Building inverted indexes per-column allows S2DB to answer queries on any subset of indexed columns by merging the postings lists from the individual per-column indexes.

Since the most selective filtering on a multi-column index happens on queries filtering on all indexed columns, we build an extra global index (3) to speed up those queries by skipping segments without a row matching all indexed columns. This is also important for uniqueness enforcement (section 4.1.2), since the unique key check is by definition always matching all columns in the unique key.

The need to merge per-column postings lists in this design introduces a higher index lookup cost compared to the alternative design of building a postings list specific to each unique tuple of indexed columns. This difference is more significant if each individual column has a large number of matches, since the merging cost increases with the length of the postings lists, while the seeking cost remains constant. Despite the higher index lookup cost in non-selective cases, we believe that the flexibility of filtering on a partial index match is more important. Note that the total cost of the read includes also the cost of decoding the matched rows, which is similarly proportional to the number of matched rows, and it often outweighs the index lookup cost in non-selective cases.

4.1.2 Uniqueness enforcement

Most columnstore implementations don't support the enforcement of uniqueness constraints. For the few that do, it's usually done by either making the LSM tree sort on the unique key [16], or duplicating the data into a rowstore table or index [14]. Using the secondary index structure described above, S2DB columnstore supports uniqueness constraint enforcement without forcing the sort key to be the unique key columns or duplicating the data. The idea is simple - each newly inserted row checks the secondary index for duplicates before inserting into the table. As an optimization, each batch of ingested rows is checked together to amortize the metadata access cost of the global indexes. The following procedure is used

1. Take locks on the unique key values for each row in the batch. An in-memory lock manager is used here to avoid concurrent inserts of the same unique key value.
2. Perform secondary index lookups on the unique key values.
3. When there are duplicated values, depending on the user-specified unique-key handling option, either report an error (default), skip the new row (SKIP DUPLICATE KEY ERRORS option), delete and then replace the conflicting rows (REPLACE command), or update the conflicting rows (ON DUPLICATE KEY UPDATE option).

In the typical case when there's no duplicate value found during the secondary index lookup, the secondary index lookup only needs to access the global hash tables (and rarely the per-segment inverted indexes on hash collision). When there are duplicates, the data segments would need to be accessed at the row offsets matched by the index in the REPLACE and ON DUPLICATE KEY UPDATE cases.

4.2 Row-level locking

S2DB columnstore storage represents deleted rows in a segment as a bit vector in the segment metadata. While this representation is optimized for vectorized access during analytical queries, a naive implementation would introduce a source of contention when modifying the segment metadata: a user transaction running update or delete operations would acquire the lock on the metadata row of a modified segment to install a new version of the deleted bit vector, blocking other modifications on the same segment (1 million rows) until the user transaction commits or rolls back. Furthermore, the background merging process described in section 2.1.2 runs segment merge transactions, which can also block modifications on the segments being merged.

Instead of the naive implementation of having update and delete queries update the bit vector directly, S2DB implements a row-level locking mechanism to avoid blocking during transactional workloads. Rows to be updated or deleted are first moved to the in-memory rowstore part of the table in an autonomous transaction, which we refer to as a "move transaction". Since moving the row doesn't change the logical table content, the move transaction can be committed immediately, so that the user transaction only needs to lock and modify the in-memory row. With this approach, the primary key of the in-memory rowstore acts as the lock manager, where inserting a copy of the row locks the row preventing concurrent modifications. To ensure that the locked rows aren't modified before inserting their copies, an extra scanning pass on newly created segments is performed after locking to find the latest versions of the locked rows.

Since a move transaction doesn't change the logical table content, it can be reordered with other move or segment merge transactions. Reordering move and segment merge transactions allows segment merges to happen without blocking update or delete queries. Furthermore, concurrent move transactions are combined and committed as a single transaction as an optimization. To make sure that deleted bits set by move transactions reflect the latest segment metadata, the commit process applies all segment merges between the scan timestamp and the commit timestamp of the move transaction to the deleted bits modified as part of the move.

5 Adaptive query execution

Unified table storage supports multiple data access methods for transaction and analytical processing. Since hybrid workloads blur the boundary between transaction and analytical processing, for those workloads it becomes important for the query execution engine to combine different access methods and apply them in the optimal order. For example, a query may be able to use a secondary index for one filter and encoded execution for another. In which case, the optimal order of applying those filters would depend on the selectivity and the evaluation cost of each filter. Static decisions made by the query optimizer don't always work well for selecting data access methods, since the cost depends highly on the query parameters and the encodings used. Instead, S2DB adopts adaptive query execution to make the data access decisions dynamically.

Data access on S2DB unified table storage has 3 high-level steps: (1) finding the list of segments to read, (2) running filters to find the rows to read from each segment, and (3) selectively decoding and outputting the rows. Each step outputs in a consistent format, which serves as the common interface to be used across different data access methods. This section focuses on the first two steps to discuss how they incorporate dynamic decisions to work efficiently in HTAP workloads.

5.1 Segment skipping

Segments can be skipped using either the global secondary index structures or the min/max values stored in the segment metadata. The secondary index check is done first, because it only requires probing O(log(N)) times, and its result can reduce the number of segments to check for the min/max values. On the other hand, there can be multiple keys to look up when the index is used in cases like an IN-list or multiple filters connected by OR, which increases the index probing cost proportionally. Therefore, S2DB dynamically disables the use of a secondary index if the number of keys to look up is too high relative to the table size.

Using secondary indexes adaptively is important when running joins, since the number of keys used for join probing can have large variations. Instead of the typical representation of a nested loop join on the index, S2DB models a secondary index join as a "join index filter": similar to bloom filters used in hash joins, it filters the larger table using the smaller of the joined tables. Compared to a bloom filter, the join index filter has no false positives, and it runs much faster (with a small joined table) by performing index probes instead of a table scan. This model allows the join index filter to be dynamically disabled, in which case the execution falls back to a hash join, scanning the larger table and probing the hash table built from the smaller table.

5.2 Filtering

For each clause (e.g. col1 = val1) in the filter condition, there are up to four different ways to evaluate the filter, each with different tradeoffs:

Regular filter selectively decodes col1 for rows that passed previous filters, then executes filter on the decoded values.

Encoded filter executes directly on the compressed values. For example, when dictionary encoding is used, it evaluates the filter on all possible values in the dictionary for col1, then looks up the results based on the dictionary index without decoding the column. Compared with a regular filter, this strategy is ideal with a small set of possible values, but it can be worse if the dictionary size is greater than the number of rows that passed the previous filters.

Group filter decodes all filtered columns and runs the entire filter condition, instead of running the filter clauses separately. Compared with a regular filter, running a group filter is better if most rows pass each individual filter clause since it avoids the cost of combining results from individual clauses. On the other

hand, a regular filter is better if some clauses can filter out and skip further filter evaluation on most of the rows.

Secondary index filter reads the postings list for val1 stored in the index to find the filtered row offsets. Using a secondary index is usually better compared to a regular filter. However, it can still be worse if the other clauses already filtered the result down to a few rows.

To select the optimal evaluation strategy, S2DB costs each different method of filter evaluation by timing it on a small batch of data at the beginning of each segment. Doing the costing per-segment ensures that the cost is aware of the data encoding and the data correlation with the sort key. For filters using an index, costing is done using the postings list size stored in the index, since there's no per-row evaluation cost beyond reading the postings list. Costing is skipped if the filter condition is a conjunction with a selective index filter, since costing in this case would be more expensive than running the filters on rows output by the index.

Furthermore, S2DB dynamically reorders filter clauses using estimated per-row evaluation costs and filter selectivities. Consider a filtering condition A AND B with two clauses. Let cost(X) be the cost of evaluating clause X, and P(X) be its selectivity. It's better to evaluate A first if the following inequality holds:

$$cost(A) + P(A) * cost(B) \leq cost(B) + P(B) * cost(A)$$

The above inequality is equivalent to the following (by dividing both sides by cost(A) * cost(B) and rearranging the terms):

$$\frac{1 - P(B)}{cost(B)} \leq \frac{1 - P(A)}{cost(A)}$$

Therefore, the optimal evaluation order can be found by sorting the clauses by (1 - P(X)) / cost(X), under the assumption that filter clauses are independent. Similar reordering can be done for clauses connected by OR by tracking the ratio of rows not selected by the filter clause instead of the selected rows. S2DB represents the filter condition as a tree and reorders each intermediate AND/OR node in the tree separately. The ordering decision is made per-block using the selectivities from previous blocks, to ensure that the selectivity estimates reflect the data distribution in the nearby blocks.

6 Experimental Results

We used benchmarks derived from the industry-standard TPC-H and TPC-C benchmarks to evaluate S2DB compared to other leading cloud databases and data warehouses. The results below show S2DB achieves leading-edge performance on both TPC-H, an OLAP benchmark, and TPC-C, an OLTP benchmark. We also ran CH-BenCHmark against S2DB which runs a mixed workload derived from running TPC-C and TPC-H simultaneously.

We compared S2DB with three other products: two cloud data warehouses we refer to as CDW1 and CDW2, and a cloud operational database we refer to as CDB. As the results below show, S2DB had good performance and cost-performance on the analytic benchmark TPC-H compared to the cloud data warehouses, and on the transactional benchmark TPC-C compared to CDB. On the other hand, CDW1 and CDW2 only support data warehousing and cannot run TPC-C. CDB can run both benchmarks, but our results as well as previous results [47] show it performs orders of magnitude worse than the cloud data warehouses on TPC-H.

We ran both benchmarks on S2DB's unified table storage described in section 4. That is, both benchmarks were run using the same underlying table storage (which is the default out-of-the-box configuration; we did not force rowstore or columnstore table storage). We used indexes, sort keys, and shard keys appropriate for each benchmark, and used similar features across all products where those options were available. The schemas, data loading commands, and queries used for testing are published online[17].

We used the schemas, data, and queries of the benchmarks as defined by the TPC. However, this is not an official TPC benchmark.

We compared the products on TPC-H at the 1TB scale factor, and TPC-C at 1,000 warehouses. Note that we have previously published results on these benchmarks as well as TPC-DS at much larger scale factors [19, 50], demonstrating that S2DB scales well. Here, we chose to run TPC-H instead of TPC-DS because it required fewer modifications to run the same benchmark on CDW1 and CDW2.

Product	vCPU	Size (warehouses)	Throughput (tpmC)	Throughput (% of max)
CDB	32	1000	12,582	97.8%
S2DB	32	1000	12,556	97.7%
S2DB	256	10000	121,432	94.4%

Table 1: TPC-C results (higher is better, up to the limit of 12.86 tpmC/warehouse)

Product	Cluster price per hour	TPC-H geomean (sec)	TPC-H geomean (cents)	TPC-H throughput (QPS)
S2DB	$16.50	8.57 s	3.92 ¢	0.078
CDW1	$16.00	10.31 s	4.58 ¢	0.069
CDW2	$16.30	10.06 s	4.55 ¢	0.082
CDB	$13.92	Did not finish within 24 hours		

Table 2: Summary of TPC-H (1TB) results

We did comparisons on Amazon EC2 on cluster sizes that were chosen to be as similar in price as possible. Information about the cluster configurations we used are in Table 1 and 2.

For TPC-H, we measured the runtime of each query, and computed the cost of each query by multiplying the runtime by the price per second of the product configuration. We then computed the geomean of the results across the queries. The results are shown in Table 2 and Figures 4. We performed one cold run of each query to allow for query compilation and data caching, and then measured the average runtime of multiple warm runs of each query, with results caching disabled. S2DB, CDW1, and CDW2 all have competitive performance. We also tested CDB on the largest available size ($13.92 per hour), and it performed orders of magnitude worse: most queries failed to complete within 1 hour, and running all the benchmark queries

once failed to complete within 24 hours (compared to about 5 minutes for the cloud data warehouses).

For TPC-C, we compared S2DB against CDB, as shown in Table 1. Note all S2DB results were on our columnar-based unified table storage, and were competitive with CDB which is a rowstore-based operational database. CDW1 and CDW2 do not support running TPC-C. We measured the throughput (tpmC), as defined by the TPC-C benchmark. We compared against results previously published by CDB. Note that TPC-C specifies a maximum possible tpmC of 12.86 per warehouse, and both S2DB and CDB are essentially reaching this maximum at 1,000 warehouses, with similarly priced clusters. We also tested S2DB on TPC-C at 10,000 warehouses and it continues to scale linearly.

These results, summarized in Figure 5, demonstrate that S2DB's unified table storage is able to achieve state-of-the-art performance competitive with leading operational databases as well as analytical databases on benchmarks specific to each workload. In contrast, cloud operational databases like CDB have orders of magnitude worse performance on TPC-H, because of the use of a row-oriented storage format and single-host query execution on complex query operations; cloud warehouses like CDW1 and CDW2 are unable to support TPC-C, due to the lack of enforced unique constraints, granular locking, and efficient seeks under high concurrency. S2DB can meet workload requirements that previously required using multiple specialized database systems.

Test Case	vCPU	Configuration TW=Transaction Worker AW=Analytic Worker	Transactional Throughput (TpmC)	Analytical Throughput (QPS)
1	16	50 TWs and 0 AWs	7530	-
2	16	0 TWs and 2 AWs	-	0.076
3	16	50 TWs and 2 AWs sharing one workspace	3950	0.039
4	32	50 TWs and 2 AWs each in own workspace	7454	0.062
5	32	50 TWs and 2 AWs each in own workspace, no blob store	7545	0.065

Table 3: Summary of S2DB CH-BenCHmark results (1000 warehouses, 20 minute test executions)

To demonstrate how S2DB performs on mixed workloads we ran CH-BenCHmark in several different configurations by varying the number of transactional workers (TWs) running a TPC-C workload and analytical workers (AWs) running TPC-H queries over the same tables. The results are shown in Table 3. Test cases 1 to 3 were run with a single writable workspace with 2 leaves in it. 50 TWs resulted in the highest TpmC when they were run in isolation with no AWs (test case 1). 2 AWs results in the highest queries per second (QPS) from TPC-H when run in isolation (test case 2). When 50 TWs and 2 AWs are run together in the same workspace each slows down by about 50% compared to when each is run in isolation (test case 3). This result demonstrates that TWs and AWs share resources roughly equally when running together without an outsized impact on each other. Test case 4 introduces a read-only workspace with 2 leaves in it that is used to run AWs. This new workspace replicates the workload from the primary writable workspace that runs TWs as described in section 3.1, effectively doubling the compute available to the cluster. This new configuration doesn't impact TWs throughput when compared to test case 1 without the read-only workspace. AWs throughput is dramatically improved vs case 3 where it shared a single workspace with TWs. This is not too surprising as the AWs have their own dedicated compute resources in test case 4. The AWs QPS was impacted by ~20% compared to running the AWs workload without any TWs at all (test case 2) as S2DB needed to do some extra work to replicate the live TWs transactions in this case which used up some CPU. Regarding the replication lag, the AWs workspace had on average less than 1 ms of lag, being only a handful of transactions behind the TWs workspace. Test case 5 was run with blob storage disabled (all data is stored on local disks) and the performance was very close to the equivalent test case with blob storage enabled (test case 4). This shows that asynchronously uploading to blob storage doesn't use up noticeable hardware resources.

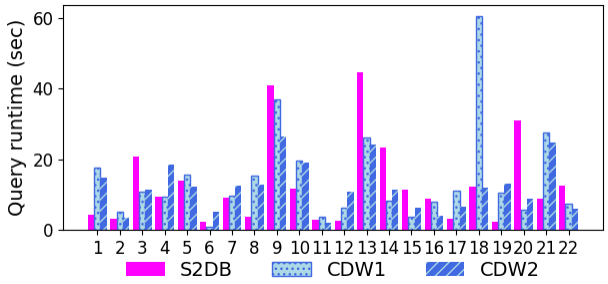

Figure 4: TPC-H 1TB query runtimes (lower is better)

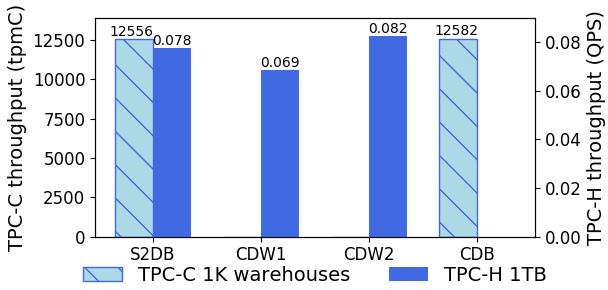

Figure 5: Summary of TPC-C and TPC-H throughputs (higher is better)

7 Related Work

Most HTAP databases on the market today are systems evolved from existing OLTP engines. Oracle [22] and SQL Server [14] have both augmented their popular rowstore engines by adding columnstore storage to them. Oracle's columnstore is in-memory only and designed to speed up scans over top of its on-disk rowstore. SQL Server's columnstore is a traditional on-disk columnstore and can be created alongside secondary B-tree rowstore indexes, which is comparable to S2DB's unified table design. However, SQL Server's secondary indexes are less

integrated into their columnstore than S2DB. For example, SQL Server requires a mapping index to map between secondary key rows and the position of the row in the clustered columnstore. S2DB stores offsets directly in its secondary keys avoiding this indirection and improving performance of secondary key lookups. SQL Server also needs to do secondary B-tree index maintenance during bulk loading. S2DB's secondary indexes are broken into segments similar to how columns are stored, and are merged in the background which improves load performance by moving most of the index maintenance work out of the bulk data loading code path.

TiDB[23] was initially built as a distributed, highly available rowstore and later added the capability to transparently replicate data into columnstore format to improve the performance of analytical queries. This design allows for OLTP queries to target the rowstore and OLAP queries to target the columnstore replicas at the cost of having to store the data twice in two different formats. Using OLAP replicas in this fashion has some limitations that S2DB's unified table design doesn't have. The replica design can't support both OLTP writes and OLAP reads within the same transaction because the OLTP writes won't be replicated to the OLAP store yet. Forcing all writes through an OLTP optimized store also means TiDB is unable to gain the bulk data loading performance benefits of keeping the data only in highly compressed columnstore format. TiDB has separate storage and query nodes within a cluster, but it doesn't make use of blob storage as a shared disk which means it misses out on the durability, elasticity and cost benefits mentioned in section 3.3.

Janus[48] also uses a write optimized rowstore for OLTP and read optimized column store for OLAP with a transactionally consistent data movement pipeline moving data from the rowstore to the columnstore. Janus is unique in that it allows the read and write optimized stores to have different partitioning schemes and its data movement pipeline batches up transactions so they can more efficiently be applied on the columnstore. This design picks up most of the same limitations mentioned above for TiDB as far as requiring data to be stored in two different formats.

There are several databases built from scratch to support HTAP. Hyper [20, 25] supports a high performance in-memory hybrid rowstore and columnstore engine. It supports running OLAP queries on a snapshot of the OLTP data, but only operates on datasets that fit into main memory. SAP HANA[24] supports in-memory rowstores and in-memory columnstore tables among other storage engines. Developers choose which table type they prefer for each table. It also supports replicating data from a rowstore into a columnstore so the same data can be stored in both formats. This has similar disadvantages to the TiDB OLAP replica design mentioned above. Neither Hyper nor SAP HANA support using blob storage as a shared disk.

Most cloud data warehouses today use a blob store for persistent storage and keep only frequently queried data cached on the hosts they use to run queries. Snowflake [26], Redshift [27] and Databricks [30] all follow this pattern. These systems force new data to be written to blob storage before it is considered durable which limits their ability to support low latency, high throughput transactional write workloads. These systems also don't support the fine-grained indexing and seekable compression schemes of S2DB's unified table storage that are needed to run low latency point queries for OLTP.

Wildfire [29] is a database that adds HTAP capabilities to Apache Spark. It commits transactions on local SSDs before converting the data into columnstore format and moving it to blob storage asynchronously. Unlike S2DB, it doesn't ship its transaction logs to blob storage so it doesn't support point in time restores from blob storage. Wildfire also builds an LSM Tree during data ingestion to allow index lookups and can spill this index to the blob store if needed. The performance of wildfire on point updates and deletes was not evaluated. The WiSER project later added stronger transactional guarantees to Wildfire [38].

Procella [11] is a database system built by Google that powers the real-time analytical features of YouTube. It has many similar design goals to S2DB as far as support for low-latency selective queries over columnstore data via inverted indexes and seekable compression schemes. It also supports separated storage by making use of Google's internal blob storage service. Although Procella is designed for low latency analytics and streaming ingestion, it doesn't support OLTP. It has special APIs for data ingestion but no support for low latency point read and write queries with millisecond latencies.

8 Conclusion

S2DB was designed to handle transactional and analytical workloads with strong performance. Its use of blob storage enables the cost, durability and elasticity benefits of shared-disk databases such as cloud data warehouses without impacting its ability to run low-latency, high-throughput write transactions. S2DB only stores cold data in blob storage. It never writes to the blob store to commit transactions. S2DB's unified table storage uses a combination of an in-memory rowstore and an on-disk columnstore that supports secondary and unique keys via inverted indexes. This design has the fast scan performance of a traditional columnstore while enabling efficient point queries via indexing. The set of design trade-offs we have chosen has been validated by the successful use of S2DB for a varied set of workloads by our customers, often meeting application requirements that previously required using multiple specialized databases.

ACKNOWLEDGMENTS

We would like to thank the SingleStore engineering team for their efforts over the years in making the various ideas in this paper a reality. Their ingenuity and hard work are a key part of the success of S2DB. We also need to call out our customers who took the time to share their feedback, expectations, and uses cases with us. They played a large role in molding the product into what it is today.

REFERENCES

[1] AWS Cloud Databases (2021). https://aws.amazon.com/products/databases/

[2] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In Proceedings of the 21st International Conference on Data Engineering (ICDE '05). IEEE Computer Society, USA, 2–11. DOI:https://doi.org/10.1109/ICDE.2005.1

[3] Amazon S3 (2021). https://aws.amazon.com/s3/

[4] Amazon EC2 (2021). https://aws.amazon.com/ec2

[5] A. Skidanov, A. J. Papito and A. Prout, "A column store engine for real-time streaming analytics," 2016 IEEE 32nd International Conference on Data Engineering (ICDE), 2016, pp. 1287-1297, doi: 10.1109/ICDE.2016.7498332.

[6] A. Prout, The Story Behind SingleStore's Skiplist Indexes (2019). https://www.singlestore.com/blog/what-is-skiplist-why-skiplist-index-for-memsql/

[7] Michal Nowakiewicz, Eric Boutin, Eric Hanson, Robert Walzer, and Akash Katipally. 2018. BIPie: Fast Selection and Aggregation on Encoded Data using Operator Specialization. In Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18). Association for Computing Machinery, New York, NY, USA, 1447–1459. DOI:https://doi.org/10.1145/3183713.3190658

[8] O'Neil, P., Cheng, E., Gawlick, D. et al. The log-structured merge-tree (LSM-tree). Acta Informatica 33, 351–385 (1996). https://doi.org/10.1007/s002360050048

[9] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution, Proc. of the 2005 CIDR Conf.

[10] Dong, S., Callaghan, M.D., Galanis, L., Borthakur, D., Savor, T., & Strum, M. (2017). Optimizing Space Amplification in RocksDB. CIDR.

[11] Chattopadhyay, B., Dutta, P., Liu, W., Tinn, O., McCormick, A., Mokashi, A., Harvey, P., Gonzalez, H., Lomax, D., Mittal, S., Ebenstein, R., Mikhaylin, N., Lee, H., Zhao, X., Xu, T., Perez, L., Shahmohammadi, F., Bui, T., Mckay, N., Aya, S., Lychagina, V., & Elliott, B. (2019). Procella: Unifying serving and analytical data at YouTube. Proc. VLDB Endow., 12, 2022-2034.

[12] Luo, C., & Carey, M.J. (2019). LSM-based storage techniques: a survey. The VLDB Journal, 29, 393-418.

[13] Indexing with SSTable attached secondary indexes (SASI). https://docs.datastax.com/en/dse/5.1/cql/cql/cql_using/useSASIIndexConcept.html

[14] 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.

[15] InnoDB Clustered and Secondary Indexes https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html

[16] Lipcon, Todd et al. "Kudu : Storage for Fast Analytics on Fast Data ∗." (2016).

[17] Bench marking code. https://github.com/memsql/benchmarks-tpc

[18] DB-Engines Ranking. https://db-engines.com/en/ranking

[19] Singlestore Unofficial TPC Benchmarking. https://www.singlestore.com/blog/memsql-tpc-benchmarks /

[20] Kemper, A., & Neumann, T. (2011). HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. 2011 IEEE 27th International Conference on Data Engineering, 195-206.

[21] Amazon RDS DB instance storage. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html

[22] 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.

[23] Huang, D., Liu, Q., Cui, Q., Fang, Z., Ma, X., Xu, F., Shen, L., Tang, L., Zhou, Y., Huang, M., Wei, W., Liu, C., Zhang, J., Li, J., Wu, X., Song, L., Sun, R., Yu, S., Zhao, L., Cameron, N., Pei, L., & Tang, X. (2020). TiDB: A Raft-based HTAP Database. Proc. VLDB Endow., 13, 3072-3084.

[24] 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.

[25] Lang, H., Mühlbauer, T., Funke, F., Boncz, P.A., Neumann, T., & Kemper, A. (2016). Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. Proceedings of the 2016 International Conference on Management of Data.

[26] Dageville, B., Cruanes, T., Zukowski, M., Antonov, V.N., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., Lee, A.W., Motivala, A., Munir, A., Pelley, S., Povinec, P., Rahn, G., Triantafyllis, S., & Unterbrunner, P. (2016). The Snowflake Elastic Data Warehouse. Proceedings of the 2016 International Conference on Management of Data.

[27] Ippokratis Pandis: The evolution of Amazon Redshift. Proc. VLDB Endow. 14(12): 3162-3163 (2021)

[28] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K.K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., & Bao, X. (2017). Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. Proceedings of the 2017 ACM International Conference on Management of Data.

[29] Shekar, K. & Bhoomeshwar, B.. (2020). Evolving Database for New Generation Big Data Applications. 10.1007/978-981-15-1632-0_26.

[30] Armbrust, M., Das, T., Paranjpye, S., Xin, R., Zhu, S., Ghodsi, A., Yavuz, B., Murthy, M., Torres, J., Sun, L., Boncz, P.A., Mokhtar, M., Hovell, H.V., Ionescu, A., Luszczak, A., Switakowski, M., Ueshin, T., Li, X., Szafranski, M., Senster, P., & Zaharia, M. (2020).Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. Proceedings of the VLDB Endowment, 13, 3411 - 3424.

[31] Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E.J., O'Neil, P.E., Rasin, A., Tran, N., & Zdonik, S.B. (2005). C-Store: A Column-oriented DBMS. VLDB.

[32] Optimizing Schema Design for Cloud Spanner. https://cloud.google.com/spanner/docs/whitepapers/optimizing-schema-design

[33] Avinash Lakshman, Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. 44(2): 35-40 (2010)

[34] Chang, F.W., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R.E. (2008). Bigtable: A Distributed Storage System for Structured Data. TOCS.

[35] Stonebraker, M. (1985). The Case for Shared Nothing. IEEE Database Eng. Bull..

[36] WiredTiger: Schema, Columns, Column Groups, Indices and Projections. https://source.wiredtiger.com/2.5.2/schema.html

[37] Luo, C., & Carey, M.J. (2019). LSM-based storage techniques: a survey. The VLDB Journal, 29, 393-418.

[38] Barber, Ronald & Garcia-Arellano, Christian & Grosman, Ronen & Lohman, Guy & Mohan, C. & Mueller, Rene & Pirahesh, Hamid & Raman, Vijayshankar & Sidle, Richard & Storm, Adam & Tian, Yuanyuan & Tozun, Pinar & Wu, Yingjun. (2019). WiSer: A Highly Available HTAP DBMS for IoT Applications.

[39] E. Hanson, SingleStore's Patented Universal Storage, 2021. https://www.singlestore.com/blog/singlestore-universal-storage-episode-4/

[40] Lattner, C., & Adve, V.S. (2004). LLVM: a compilation framework for lifelong program analysis & transformation. International Symposium on Code Generation and Optimization, 2004. CGO 2004., 75-86.

[41] Neumann, T. (2011). Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. VLDB Endow., 4, 539-550.

[42] Özcan, F., Tian, Y., & Tözün, P. (2017). Hybrid Transactional/Analytical Processing: A Survey. Proceedings of the 2017 ACM International Conference on Management of Data.

[43] Sanders, P., & Transier, F. (2007). Intersection in Integer Inverted Indices. ALENEX.

[44] Amazon Elastic Block Store (EBS). https://aws.amazon.com/ebs/

[45] Amazon S3 FAQs. https://aws.amazon.com/s3/faqs/

[46] Chen, Jack & Jindel, Samir & Walzer, Robert & Sen, Rajkumar & Jimsheleishvilli, Nika & Andrews, Michael. (2016). The MemSQL query optimizer: a modern optimizer for real-time analytics in a distributed database. Proceedings of the VLDB Endowment. 9. 1401-1412. 10.14778/3007263.3007277.

[47] Performance comparison of HeatWave with Snowflake, Amazon Redshift, Amazon Aurora, and Amazon RDS for MySQL. https://www.oracle.com/mysql/heatwave/performance

[48] Arora, Vaibhav & Nawab, Faisal & Agrawal, Divyakant & Abbadi, Amr. (2017). Janus: A Hybrid Scalable Multi-Representation Cloud Datastore. IEEE Transactions on Knowledge and Data Engineering. PP. 1-1. 10.1109/TKDE.2017.2773607

[49] Eric Boutin, How Careful Engineering Led to Processing Over a Trillion Rows Per Second (2018). https://www.singlestore.com/blog/how-to-process-trillion-rows-per-second-ad-hoc-analytic-queries/

[50] G. LaLonde, J. Cheng, S. Wang, TPC Benchmarking Results (2021). https://www.singlestore.com/blog/tpc-benchmarking-results/