Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing

Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, Jihong Ma, Li Su, Kai Zeng
Alibaba Group
{xiaowei.jxw, yuejun.huyj, yu.xiangy, guangran.jianggr, xiaojun.jinxj, chen.xiac, guobei.jwh, bob.yj, haitao.w, yuan.jiang, jihong.ma, lisu.sl, zengkai.zk}@alibaba-inc.com

ABSTRACT

In existing big data stacks, the processes of analytical processing and knowledge serving are usually separated in different systems. In Alibaba, we observed a new trend where these two processes are fused: knowledge serving incurs generation of new data, and these data are fed into the process of analytical processing which further fine tunes the knowledge base used in the serving process. Splitting this fused processing paradigm into separate systems incurs overhead such as extra data duplication, discrepant application development and expensive system maintenance.

In this work, we propose `Hologres`, which is a cloud native service for hybrid serving and analytical processing (HSAP). `Hologres` decouples the computation and storage layers, allowing flexible scaling in each layer. Tables are partitioned into self-managed shards. Each shard processes its read and write requests concurrently independent of each other. `Hologres` leverages hybrid row/column storage to optimize operations such as point lookup, column scan and data ingestion used in HSAP. We propose *Execution Context* as a resource abstraction between system threads and user tasks. Execution contexts can be cooperatively scheduled with little context switching overhead. Queries are parallelized and mapped to execution contexts for concurrent execution. The scheduling framework enforces resource isolation among different queries and supports customizable schedule policy. We conducted experiments comparing `Hologres` with existing systems specifically designed for analytical processing and serving workloads. The results show that `Hologres` consistently outperforms other systems in both system throughput and end-to-end query latency.

PVLDB Reference Format:
Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, Jihong Ma, Li Su, Kai Zeng. Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing. *PVLDB*, 13(12): 3272 - 3284, 2020.
DOI: https://doi.org/10.14778/3415478.3415550

1. INTRODUCTION

Modern business is pervasively driven by deriving business insights from huge amounts of data. From the experience of running Alibaba internal big data service stacks as well as public cloud offerings, we have observed new patterns on how modern business uses big data. For instance, to support real-time learning and decision making, the big data stack behind modern e-commerce services usually aggregate real-time signals like purchase transactions and user click logs to continuously derive fresh product and user statistics. These statistics are heavily used in both online and offline manners, e.g.: (1) They are served immediately online as important features. Incoming user events are joined with these features to generate samples for real-time model training in search and recommendation systems. (2) They are also used by data scientists in complex interactive analysis to derive insights for model tuning and marketing operations. These usage patterns clearly demonstrate a host of new trends which the traditional concept of *Online Analytical Processing* (OLAP) can no longer accurately cover:

Fusion of Analytical Processing and Serving. Traditional OLAP systems usually play a rather static role in the whole business stack. They analyze large quantities of data and derive knowledge (e.g., precomputed views, learned models, etc.) off-line, but hand over the derived knowledge to another system for serving online applications. Differently, modern business decision-making is a constantly-tuned online process. The derived knowledge is not only served but also participates in complex analytics. The need for analytical processing and serving on big data is fused together.

Fusion of Online and Offline Analysis. Modern business needs to quickly transform freshly obtained data into insights. Written data has to be available to read within seconds. A lengthy offline ETL process is no longer tolerable. Furthermore, among all the data collected, the traditional way of synchronizing data from an OLTP system only accounts for a very small portion. Orders of magnitudes more data is from less transactional scenarios such as user click logs. The systems have to handle high volume data ingestion with very low latency while processing queries.

Existing big data solutions usually host the hybrid serving and analytical processing workloads using a combination of different systems. For instance, the ingested data is pre-aggregated in real time using systems like Flink [4] populated in systems like Druid [36] that handles multi-dimensional analytics, and served in systems like Cassandra [26]. This inevitably causes excessive data duplication and complex data synchronization across systems, inhibits an application's ability to act on data immediately, and incurs non-trivial development and administrative overheads.

In this paper, we argue that hybrid serving/analytical processing (HSAP) should be unified and handled in a single system. In Alibaba, we build a cloud-native HSAP service called `Hologres`. As a new service paradigm, HSAP has challenges that are very different from existing big data stacks (See Section 2.2 for a detailed discussion): *(1) The system needs to handle query workloads much higher*

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415550

than traditional OLAP systems. These workloads are hybrid, with very different latency and throughput trade-offs. (2) While handling high-concurrency query workloads, the system also needs to keep up with high-throughput data ingestion. The ingested data needs to be available to reads within seconds, in order to meet the stringent freshness requirements of serving and analysis jobs. (3) The mixed workloads are highly dynamic, usually subject to sudden bursts. The system has to be highly elastic and scalable, reacting to these bursts promptly.

In order to tackle these challenges, Hologres is built with a complete rethinking of the system design:

Storage Design. Hologres adopts an architecture that decouples storage from computation. Data is remotely persisted in cloud storage. Hologres manages tables in table groups, and partitions a table group into multiple shards. Each shard is self-contained, and manages reads and writes independently. Decoupled from the physical worker nodes, data shards can be flexibly migrated between workers. With data shard as the basic data management unit in Hologres, processes such as failure recovery, load balancing and cluster scaling out can be implemented using shard migration efficiently.

To support low-latency queries with high-throughput writes at the same time, shards are designed to be versioned. The critical paths of reads and writes on each table group shard are separated. Hologres uses a tablet structure to uniformly store tables. Tablets can be in row or columnar formats, and are both managed in a LSM-like way to maximize the write throughput, and minimizes the freshness delay for data ingestion.

Concurrent Query Execution. We build a service-oriented resource management and scheduling framework, named HOS. HOS uses *execution context* as the resource abstraction on top of system threads. Execution contexts are cooperatively scheduled with little context switching overhead. HOS parallelizes query execution by dividing queries into fine-grained work units and mapping work units to execution contexts. This architecture can fully exploit the potential of high hardware parallelism, allowing us to multiplex a huge number of queries concurrently. Execution context also facilitates the enforcement of resource isolation, such that low-latency serving workload can coexist with the analytical workload in the same system without being stalled. HOS makes the system easily scalable according to practical workload.

In retrospect, we make the following list of contributions:

1. We introduce a new paradigm of big data service for hybrid serving/analytical processing (HSAP), and identify the new challenges under this new paradigm.
2. We design and implement a cloud-native HSAP service called Hologres. Hologres has a novel storage design, and a highly efficient resource management and scheduling layer named HOS. These novel designs in combination help Hologres achieve real-time ingestion, low-latency serving, interactive analytical processing, and also support federated query execution with other systems such as PostgreSQL [12].
3. we have deployed Hologres in Alibaba's internal big data stack as well as public cloud offerings, and conducted a thorough performance study under real-life workloads. Our results show that Hologres achieves superior performance even compared with specialized serving systems and OLAP engines.

The paper is organized as follows: The key design considerations and system overview of Hologres are presented in Section 2. In Section 3, we explain the data model and storage framework. Next, we introduce the scheduling mechanism and details of query processing in Section 4. Experimental results are presented and discussed in Section 5. Lastly, we discuss the related research in Section 6 and conclude this work.

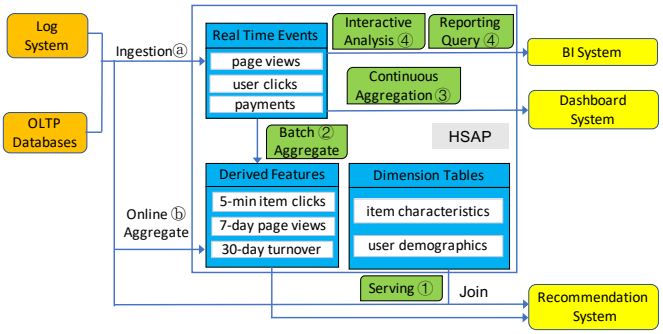

Figure 1: An example HSAP scenario: the big data stack behind a recommendation service

2. KEY DESIGN CONSIDERATIONS

Big data systems in modern enterprises are facing an increasing request of hybrid serving and analytical processing. In this section, we use the recommendation service in Alibaba to demonstrate a typical HSAP scenario, and summarize the new challenges posed by HSAP to system design. Then we provide a system overview of how Hologres addresses these challenges.

2.1 HSAP in Action

Modern recommendation services put great emphasis on reflecting real-time user trends and provide personalized recommendations. In order to achieve these goals, the backend big data stack has evolved into a state with extreme complexity and diverse data processing patterns. Figure 1 presents an illustrative picture of the big data stack backing the recommendation service in Alibaba e-commerce platforms.

To capture personalized real-time behaviors, the recommendation service heavily relies on real-time features and continuously updated models. There are usually two types of real-time features:

1. The platform aggressively collects a large number of real-time events, including log events (e.g., page views, user clicks), as well as transactions (e.g., payments synced from the OLTP databases). As we observed from production, these events are of extremely high volume, and the majority of them are less transactional log data, e.g., 10^7 events/s. These events are immediately ingested into the data stack (ⓐ) for future use, but more importantly they are joined with various dimension data on the fly to derive useful features (①), and these features are fed into the recommendation system at real-time. This real-time join needs point lookup of dimension data with extremely low latency and high throughput, in order to keep up with the ingestion.
2. The platform also derives many features by aggregating the real-time events in sliding windows, along a variety of dimensions and time granularities, e.g., 5-min item clicks, 7-day page views, and 30-day turnover. These aggregations are carried out in either batch (②) or streaming fashion depending on the sliding window granularity, and ingested into the data stack (ⓑ).

These real-time data are also used in generating training data to continuously update the recommendation models, through both online and offline training.

Despite of its importance, the above process is only a small portion of the entire pipeline. There is a whole stack of monitoring, validation, analysis and refinement process supporting a recommendation system. These include but not limited to continuous dashboard queries (③) on the collected events to monitor the key performance metrics and conduct A/B testing, and periodic batch queries (④) to

generate BI reports. Besides, data scientists are constantly performing complex interactive analysis over the collected data to derive real-time insights for business decisions, to do causal analysis and refinement of the models. For instance, on the double-11 shopping festival, the incoming OLAP query requests can go up to hundreds of queries per second.

The above demonstrates a highly complex HSAP scenario, ranging from real-time ingestion (ⓐ) to bulk load (ⓑ), from serving workload (①), continuous aggregation (③), to interactive analysis (④), all the way to batch analysis (②). Without a unified system, the above scenario has to be jointly served by multiple isolated systems, e.g., batch analysis by systems like Hive; serving workload by systems like Cassandra; continuous aggregation by systems like Druid; interactive analysis by systems like Impala or Greenplum.

2.2 Challenges of a HSAP Service

As a new big data service paradigm, HSAP service proposes challenges that were not as prominent just a few years ago.

High-Concurrency Hybrid Query Workload. HSAP systems usually face high query concurrency that is unprecedented in traditional OLAP systems. In practice, compared to OLAP query workload, the concurrency of serving query workload is usually much higher. For instance, we have observed in real-life applications that serving queries could arrive at a rate as high as 10^7 *queries per second* (QPS), which is five orders of magnitude higher than the QPS of OLAP queries. Furthermore, serving queries have a much more stringent latency requirement than OLAP queries. How to fulfill these different query SLOs while multiplexing them to fully utilize the computation resource is really challenging.

Existing OLAP systems generally use a process/thread-based concurrency model, i.e., use a separate process [5] or thread [6] to handle a query, and rely on the operating system to schedule concurrent queries. The expensive context switching caused by this design puts a hard limit on the system concurrency, and thus is no longer suitable for HSAP systems. And it prevents the system to have enough scheduling control to meet different query SLOs.

High-Throughput Real-Time Data Ingestion. While handling high-concurrency query workloads, HSAP systems also need to handle high-throughput data ingestion. Among all the data ingested, the traditional way of synchronizing data from an OLTP system only accounts for a very small portion, while the majority of data comes from various data sources such as real-time log data that do not have a strong transaction semantics. The ingestion volume can be much higher than observed in a hybrid transaction-analytical processing (HTAP) system. For instance, in the above scenario the ingestion rate goes up to tens of millions of tuples per second. What is more, different from traditional OLAP systems, HSAP systems require real-time data ingestion—written data has to be visible within subsecond—to guarantee the data freshness of analysis.

High Elasticity and Scalability. The ingestion and query workload can undergo sudden bursts, and thus require the system to be elastic and scalable, and react promptly. We have observed in real-world applications that the peak ingestion throughput reaches $2.5X$ of the average, and the peak query throughput reaches $3X$ of the average. Also, the bursts in ingestion and query workload do not necessarily coincide, which requires the system to scale the storage and computation independently.

2.3 Data Storage

In this subsection, we discuss the high-level design of data storage in `Hologres`.

Decoupling of Storage/Computation. `Hologres` takes a cloud-native design where the computation and storage layers are decoupled. All the data files and logs of `Hologres` are persisted in Pangu by default, which is a high performance distributed file system in Alibaba Cloud. We also support open-source distributed file systems such as HDFS [3]. With this design, both the computation and storage layers can be independently scaled out according to the workload and resource availability.

Tablet-based Data Layout. In `Hologres`, both tables and indexes are partitioned into fine-grained tablets. A write request is decomposed into many small tasks each of which handles the updates to a single tablet. Tablets for correlated tables and indexes are further grouped into shards, to provide efficient consistency guarantees. To reduce contention, we use a latch-free design that each tablet is managed by a single writer, but can have arbitrary number of readers. We can configure a very high read parallelism for query workloads, which hides the latency incurred by reading from a remote storage.

Separation of Reads/Writes. `Hologres` separates the read and write paths, to support both high-concurrency reads and high-throughput writes at the same time. The writer of a tablet uses an LSM-like approach to maintain the tablet image, where the records are properly versioned. Fresh writes can be visible for reads with subsecond-level latency. Concurrent reads can request a specific version of the tablet image, and thus are not blocked by the writes.

2.4 Concurrent Query Execution

In this subsection, we discuss the high-level design of the scheduling mechanism used by `Hologres`.

Execution Context. `Hologres` builds a scheduling framework, referred to as `HOS`, which provides a user-space thread called *execution context* to abstract the system thread. Execution contexts are super light weight and can be created and destroyed with negligible cost. `HOS` cooperatively schedules execution contexts on top the system thread pools with little context switching overhead. An execution context provides an asynchronous task interface. `HOS` divides users' write and read queries into fine-grained work units, and maps the work units onto execution contexts for scheduling. This design also enables `Hologres` to promptly react to sudden workload bursts. The system can be elastically scaled up and down at runtime.

Customizable Scheduling Policy. `HOS` decouples the scheduling policy from the execution context based scheduling mechanism. `HOS` groups execution contexts from different queries into scheduling groups, each with their own resource share. `HOS` is in charge of monitoring the consumed share of each scheduling group, and enforcing resource isolation and fairness between scheduling groups.

2.5 System Overview

Figure 2 presents the system overview of `Hologres`. The *frond-end nodes (FEs)* receive queries submitted from clients and return the query results. For each query, the *query optimizer* in the FE node generates a query plan, which is parallelized into a DAG of fragment instances. The coordinator dispatches fragment instances in a query plan to the worker nodes, each of which map the fragment instances into work units (Section 4.1). A *worker node* is a unit of physical resources, i.e., CPU cores and memory. Each worker node can hold the memory tables for multiple table group shards (Section 3.2) for a database. In a worker node, work units are executed as execution contexts in the *EC pool* (Section 4.2). The *HOS scheduler* schedules the EC pool on top of the system threads (Section 4.3), following the pre-configured scheduling policy (Section 4.5).

The *resource manager* manages the distribution of table group shards among worker nodes: resources in a worker node are logically split into *slots*, each of which can only be assigned to one ta-

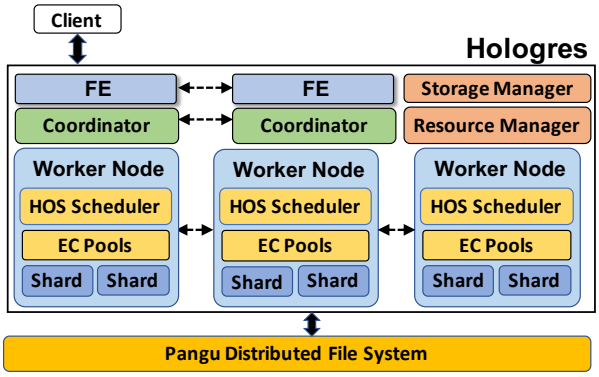

Figure 2: Architecture of Hologres

ble group shard. The resource manager is also responsible for the adding/removal of worker nodes in a `Hologres` cluster. Worker nodes periodically send heartbeats to the resource manager. Upon a worker node failure or a workload burst in the cluster, the resource manager dynamically adds new worker nodes into the cluster.

The *storage manager* maintains a directory of table group shards (see Section 3.1), and their meta data such as the physical locations and key ranges. Each coordinator caches a local copy of these meta data to facilitate the dispatching of query requests.

`Hologres` allows the execution of a single query to span `Hologres` and other query engines (Section 4.2.3). For instance, when fragment instances need to access data not stored in `Hologres`, the coordinator distributes them to other systems storing the required data. We designed and implemented a set of unified APIs for query processing, such that work units executed in `Hologres` can communicate with other execution engines such as PostgreSQL [12]. Non-`Hologres` execution engines have their own query processing and scheduling mechanisms independent of `Hologres`.

3. STORAGE

`Hologres` supports a hybrid row-column storage layout tailored for HSAP scenarios. The row storage is optimized for low-latency point lookups, and the column storage is designed to perform high-throughput column scans. In this section, we present the detailed design of the hybrid storage in `Hologres`. We start by introducing the data model and defining some preliminary concepts. Next, we introduce the internal structure of table group shards, and explain in details how to perform writes and reads. Lastly, we present the layouts of the row and column storage, followed by a brief introduction to the caching mechanism in `Hologres`.

3.1 Data Model

In `Hologres`, each table has a user-specified clustering key (empty if not specified), and a unique *row locator*. If the clustering key is unique, it is directly used as the row locator; otherwise, a *uniquifier* is appended to the clustering key to make a row locator, i.e., $\langle clustering\_key, uniquifier \rangle$.

All the tables of a database are grouped into *table groups*. A table group is sharded into a number of *table group shards* (TGSs), where each TGS contains for each table a partition of the base data and a partition of all the related indexes. We treat the base-data partition as well as an index partition uniformly as a *tablet*. Tablets have two storage formats: *row tablet* and *column tablet*, optimized for point lookup and sequential scan respectively. The base data and indexes can be stored in a row tablet, a column tablet, or both. A tablet is required to have a unique key. Therefore, the key of a base-data tablet is the row locator. Whereas for tablets of secondary indexes, if the index is unique, the indexed columns are used as the key of the tablet; otherwise, the key is defined by adding the row locator to the indexed columns. For instance, consider a TGS with a single table and two secondary indexes—a unique secondary index $(k_1 \rightarrow v_1)$ and a non-unique secondary index $(k_2 \rightarrow v_2)$—and the base data is stored in both row and column tablets. As explained above, the key of the base-data (row and column) tablets are $\langle row\_locator \rangle$, the key of the unique-index tablet is $\langle k_1 \rangle$ and the key of the non-unique-index tablet is $\langle k_2, row\_locator \rangle$.

We observed that majorities of writes in a database access a few closely-related tables, also writes to a single table update the base data and related indexes simultaneously. By grouping tables into table groups, we can treat related writes to different tablets in a TGS as an atomic write operation, and only persist one log entry in the file system. This mechanism helps improve the write efficiency by reducing the number of log flushes. Besides, grouping tables which are frequently joined helps eliminate unnecessary data shuffling.

3.2 Table Group Shard

TGS is the basic unit of data management in `Hologres`. A TGS mainly comprises a WAL manager and multiple tablets belonging to the table shards in this TGS, as exampled in Figure 3.

Tablets are uniformly managed as an LSM tree: Each tablet consists of a memory table in the memory of the worker node, and a set of immutable shard files persisted in the distributed file system. The memory table is periodically flushed as a shard file. The shard files are organized into multiple levels, $Level_0$, $Level_1$, ..., $Level_N$. In $Level_0$, each shard file corresponds to a flushed memory table. Starting from $Level_1$, all the records in this level are sorted and partitioned into different shard files by the key, and thus the key ranges of different shard files at the same level are non-overlapping. $Level_{i+1}$ can hold K times more shard files than $Level_i$, and each shard file is of max size M. More details of the row and column tablets are explained in Section 3.3 and 3.4, respectively.

A tablet also maintains a metadata file storing the status of its shard files. The metadata file is maintained following a similar approach as RocksDB [13], and persisted in the file system.

As records are versioned, reads and writes in TGSs are completely decoupled. On top of that, we take a lock-free approach by only allowing a single writer for the WAL but any number of readers concurrently on a TGS. As HSAP scenarios have a weaker consistency requirement than HTAP, `Hologres` chooses to only support *atomic write* and *read-your-writes read* to achieve high throughput and low latency for both reads and writes. Next, we explain in details how reads and writes are performed.

3.2.1 Writes in TGSs

`Hologres` supports two types of writes: *single-shard write* and *distributed batch write*. Both types of writes are atomic, i.e., writes either commit or rollback. Single-shard write updates one TGS at a time, and can be performed at an extremely high rate. On the other hand, distributed batch write is used to dump a large amount of data into multiple TGSs as a single transaction, and is usually performed with a much lower frequency.

Single-shard Write. As illustrated in Figure 3, on receiving a single-shard ingestion, the WAL manager (1) assigns the write request an LSN, which consists of the timestamp and an increasing sequence number, and (2) creates a new log entry and persists the log entry in the file system. The log entry contains the necessary information to replay the logged write. The write is committed after its log entry is completely persisted. After that, (3) operations in the write request

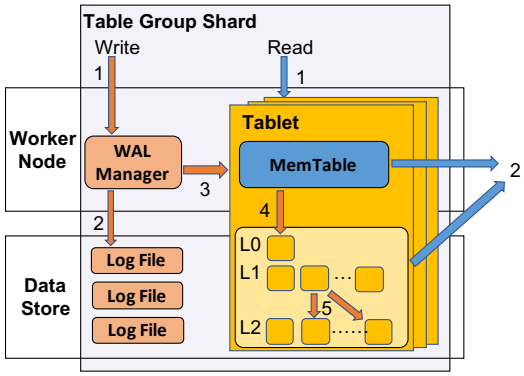

Figure 3: Internals of TGS

are applied in the memory tables of the corresponding tablets and made visible to new read requests. It is worth noting that updates on different tablets can be parallelized (see Section 4.1). Once the memory table is full, (4) it is flushed as a shard file in the file system and a new one is initialized. Lastly, (5) shard files are asynchronously compacted in the background. At the end of a compaction or memory table flush, the metadata file of the tablet is updated accordingly.

Distributed Batch Write. We adopt a two-phase commit mechanism to guarantee write atomicity for distributed batch write. The FE node which receives the bath write request locks all the accessed tablets in the involved TGSs. Then each TGS: (1) assigns an LSN for this batch write, (2) flushes the memory tables of the involved tablets and (3) loads the data as in the process of single-shard ingestion and flushes them as shard files. Note that, step (3) can be further optimized by building multiple memory tables and flushing them into the file system in parallel. Once finished, each TGS votes to the FE node. When the FE node collects all the votes from participating TGSs, it acknowledges them the final commit or abort decision. On receiving the commit decision, each TGS persists a log indicating this batch write is committed; otherwise, all the newly generated files during this batch write are removed. When the two-phase commit is done, locks on involved tablets are released.

3.2.2 Reads in TGSs

`Hologres` supports multi-version reads in both row and column tablets. The consistency level of read requests is *read-your-writes*, i.e., a client will always see the latest committed write by itself. Each read request contains a read timestamp, which is used to construct an LSN_{read}. This LSN_{read} is used to filter out invisible records to this read, i.e., records whose LSNs are larger than LSN_{read}.

To facilitate multi-version read, a TGS maintains for each table a LSN_{ref}, which stores the LSN of the oldest version maintained for tablets in this table. LSN_{ref} is periodically updated according to a user-specified retaining period. During the memory table flush and file compaction, for a given key: (1) records whose LSNs are equal to or smaller than LSN_{ref} are merged; (2) records whose LSNs are larger than LSN_{ref} are kept intact.

3.2.3 Distributed TGS Management

In our current implementation, the writer and all the readers of a TGS are co-located in the same worker node to share the memory tables of this TGS. If the worker node is undergoing workload bursts, `Hologres` supports migrating some TGSs off the overloaded worker nodes (see Section 4.4).

We are working on a solution that maintains for a TGS read-only replicas remote to the corresponding writer, to futher balance concurrent reads. We plan to support two types of read-only replicas: (1) a *fully-synced replica* maintains the up-to-date copy of both the memory table and metadata file of the TGS, and can serve all read requests; (2) a *partially-synced replica* only maintains an up-to-date copy of the metadata file, and can only serve reads over the data flushed into file system. Reads to a TGS can be dispatched to different replicas according to their read versions. Note that, both read-only replicas do not need to replicate the shard files, which are loaded from the distributed file system if requested.

If a TGS is failed, the storage manager requests an available slot from the resource manager, and at the same time broadcasts a *TGS-fail* message to all the coordinators. When recovering a TGS, we replay the WAL logs from the latest flushed LSN to rebuild its memory tables. The recovery is done once all the memory tables are completely rebuilt. After that, the storage manager is acknowledged and then broadcasts a *TGS-recovery* message containing the new location to all the coordinators. The coordinators temporarily hold requests to the failed TGS until it is recovered.

3.3 Row Tablet

Row tablets are optimized to support efficient point lookups for the given keys. Figure 4(a) illustrates the structure of a row tablet: We maintain the memory table as a Masstree [30], within which we sort the records by their keys. Differently, the shard files are of a block-wise structure. A shard file consists of two types of blocks: data block and index block. Records in a shard file are sorted by the key. Consecutive records are grouped as a data block. To help look up records by their keys, we further keep track of the starting key of each data block and its offset in the shard file as a pair of $\langle key, block\_offset \rangle$ in the index block.

To support multi-versioned data, the value stored in a row tablet is extended as $\langle value\_cols, del\_bit, LSN \rangle$: (1) the $value\_cols$ are the non-key column values; (2) del_bit indicates if this is a delete record; (3) LSN is the corresponding write LSN. Given a key, both the memory table and the shard files could have multiple records with different LSNs.

Reads in Row Tablets. Every read in row tablets consists of a key and an LSN_{read}. The result is obtained by searching in the memory table and shard files of the tablet in parallel. Only the shard files whose key ranges overlaps with the given key are searched. During the search, a record is marked as a *candidate* if it contains the given key and has an LSN equal to or smaller than LSN_{read}. The candidate records are merged in the order of their LSNs as the result record. If the del_bit in the result record is equal to 1, or no candidate record is found, there is no record for the given key exists in the version of LSN_{read}. Otherwise, the result record is returned.

Writes in Row Tablets. In row tablets, an insert or update consists of the key, column values and an LSN_{write}. A delete contains a key, a special deletion mark and an LSN_{write}. Each write is transformed into a key-value pair of row tablets. For insert and update, the del_bit is set as 0. For delete, the column fields are empty and del_bit is set as 1. The key-value pairs are first appended into the memory table. Once the memory table is full, it is flushed into the file system as a shard file in $Level_0$. This could further trigger a cascading compaction from $Level_i$ to $Level_{i+1}$ if $Level_i$ is full.

3.4 Column Tablet

Column tablets are designed to facilitate column scans. As depicted in Figure 4(b), different from row tablets, a column tablet consists of two components, a *column LSM tree* and a *delete map*.

The value stored in a column LSM tree is extended in the format of: $\langle value\_cols, LSN \rangle$, where $value\_cols$ are the non-key columns and LSN is the corresponding write LSN. In a column LSM tree, the

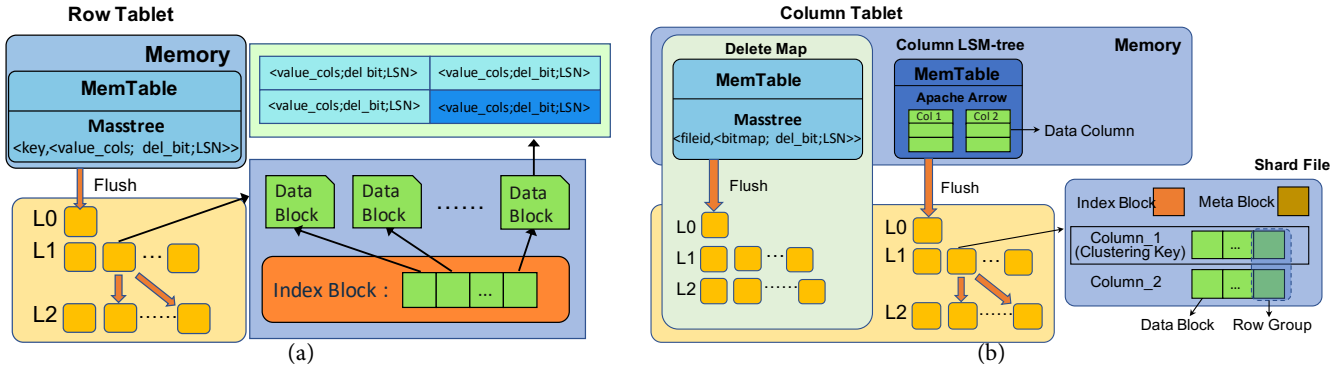

Figure 4: (a) The structures of a row tablet, and (b) the structures of a column tablet

memory table stores the records in the format of Apache Arrow [2]. Records are continuously appended into the memory table in their arriving order. In a shard file, records are sorted by the key and logically split into row groups. Each column in a row group is stored as a separate data block. Data blocks of the same column are continuously stored in the shard file to facilitate sequential scan. We maintain the meta data for each column and the entire shard file in the meta block to speed up large-scale data retrieving. The meta block stores: (1) for each column, the offsets of data blocks, the value ranges of each data block and the encoding scheme, and (2) for the shard file, the compression scheme, the total row count, the LSN and key range. To quickly locate the row according to a given key, we store the sorted first keys of row groups in the index block.

The delete map is a row tablet, where the key is the ID of a shard file (with the memory table treated as a special shard file) in the column LSM tree, and the value is a bitmap indicating which records are newly deleted at the corresponding LSN in the shard file. With the help of the delete map, column tablets can massively parallelize sequential scan as explained below.

Reads in Column Tablets. A read operation to a column tablet comprises of the target columns and an LSN_{read}. The read results are obtained by scanning the memory table and all the shard files. Before scanning a shard file, we compare its LSN range with LSN_{read}: (1) if its minimum LSN is larger than LSN_{read}, this file is skipped; (2) if its maximum LSN is equal to or smaller than LSN_{read}, the entire shard file is visible in the read version; (3) otherwise, only a subset of records in this file are visible in the read version. In the third case, we scan the LSN column of this file and generate an LSN bitmap indicating which rows are visible in the read version. To filter out the deleted rows in a shard file, we perform a read in the delete map (as explained in Section 3.3) with the ID of the shard file as the key at version LSN_{read}, where the merge operation unions all the candidate bitmaps. The obtained bitmap is intersected with the LSN bitmap, and joined with the target data blocks to filter out the deleted and invisible rows at the read version. Note that different from row tablets, in a column tablet each shard file can be read independently without the need of consolidating with shard files in other levels, as the delete map can efficiently tell all the deleted rows up to LSN_{read} in a shard file.

Writes in Column Tablets. In column tablets, an insert operation consists of a key, a set of column values and an LSN_{write}. A *delete* operation specifies the key of the row to be deleted, with which we can quickly find out the file ID containing this row and its row number in this file. We perform an insert at version LSN_{write} in the delete map, where the key is the file ID and the value is the row number of the deleted row. The *update* operation is implemented as *delete* followed by *insert*. Insertions to the column LSM tree and the delete map can trigger memory table flush and shard file compaction.

3.5 Hierarchical Cache

`Hologres` adopts a hierarchical caching mechanism to reduce both the I/O and computation costs. There are in total three layers of caches, which are the *local disk cache*, *block cache* and *row cache*. Every tablet corresponds to a set of shard files stored in the distributed file system. The local disk cache is used to cache shard files in local disks (SSD) to reduce the frequency of expensive I/O operations in the file system. On top of the SSD cache, an in-memory block cache is used to store the blocks recently read from the shard files. As the serving and analytic workloads have very different data access patterns, we physically isolate the block caches of row and column tablets. On top of the block cache, we further maintain an in-memory row cache to store the merged results of recent point lookups in row tablets.

4. QUERY PROCESSING & SCHEDULING

In this section, we present the parallel query execution paradigm of `Hologres` and the `HOS` scheduling framework.

4.1 Highly Parallel Query Execution

Figure 5 illustrates the query-processing workflow in `Hologres`. On receiving a query, the query optimizer in the FE node generates a query plan represented as a DAG, and divides the DAG at *shuffle* boundaries into *fragments*. There are three types of fragments: *read/write/query fragments*. A read/write fragment contains a read-/write operator accessing a table, whereas a query fragment only contains non-read/write operators. Each fragment is then parallelized into multiple *fragment instances* in a data parallel way, e.g., each read/write fragment instance processes one TGS.

The FE node forwards the query plan to a coordinator. The coordinator then dispatches the fragment instances to worker nodes. Read/write fragment instances are always dispatched to the worker nodes hosting the accessed TGSs. Query fragment instances can be executed in any worker node, and are dispatched taking into account the existing workloads of worker nodes to achieve load balancing. The locality and workload information are synced with the storage manager and resource manager, respectively.

In a worker node, fragment instances are mapped into *work units (WUs)*, which are the basic units of query execution in `Hologres`. A WU can dynamically spawn WUs at run time. The mapping is described as follows:

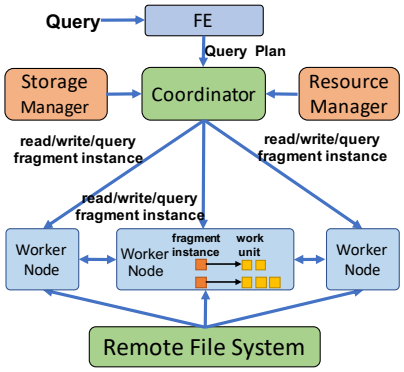

Figure 5: Workflow of Query Parallelization

- A read fragment instance is initially mapped to a *read-sync WU*, which fetches the current version of the tablet from the metadata file, including a read-only snapshot of the memory table and a list of shard files. Next, the read-sync WU spawns multiple *read-apply WUs* to read the memory table and shard files in parallel, as well as to execute downstream operators on the read data. This mechanism exploits high intra-operator parallelism to make better use of the network and I/O bandwidth.
- A write fragment instance maps all non-write operators into a *query WU*, followed by a *write-sync WU* persisting the log entry in WAL for the written data. The write-sync WU then spawns multiple *write-apply WUs*, each updating one tablet in parallel.
- A query fragment instance is mapped to a *query WU*.

4.2 Execution Context

As a HSAP service, `Hologres` is designed to execute multiple queries submitted by different users concurrently. The overhead of context switching among WUs of concurrent queries could become a bottleneck for concurrency. To solve this problem, `Hologres` proposes a user-space thread, named as *execution context (EC)* , as the resource abstraction for WU. Different from threads which are preemptively scheduled, ECs are cooperatively scheduled without using any system call or synchronization primitive. Thus the cost of switching between ECs is almost negligible. `HOS` uses EC as the basic scheduling unit. Computation resources are allocated in the granularity of EC, which further schedules its internal tasks. An EC will be executed on the thread which it is assigned to.

4.2.1 EC Pools

In a worker node, we group ECs into different pools to allow isolation and prioritization. EC pools can be categorized into three types: *data-bound EC pool*, *query EC pool* and *background EC pool*.

- A data-bound EC pool has two types of ECs: *WAL EC* and *tablet EC*. Within a TGS, there is one WAL EC and multiple tablets ECs, one for each tablet. The WAL EC executes the write-sync WUs, while the tablet EC executes the write-apply WUs and read-sync WUs on the corresponding tablet. The WAL/tablet ECs process WUs in a single-threaded way, which eliminates the necessity of synchronization between concurrent WUs.
- In a query EC pool, each query WU or read-apply WU is mapped to a query EC.
- In a background EC pool, ECs are used to offload expensive work from data-bound ECs and improve the write throughput. This includes memory table flush and shard file compaction, etc. With this design, the data-bound ECs are reserved mainly for operations on the WAL and writes to memory tables, and thus the system can achieve a very high write throughput without the overhead of locking.

To limit the resource consumption of background ECs, we physically isolate background ECs from the data-bound and query ECs in different thread pools, and execute the background ECs in a thread pool with lower priority.

4.2.2 Internals of Execution Context

Next, we introduce the internal structure of an EC.

Task Queue. There are two task queues in an EC: (1) a lock-free internal queue which stores tasks submitted by the EC itself, (2) a thread-safe submit queue which stores tasks submitted by other ECs. Once scheduled, tasks in the submit queue are relocated to the internal queue to facilitate lock-free scheduling. Tasks in the internal queue are scheduled in FIFO order.

State. During the lifetime of an EC, it switches between three states: *runnable*, *blocking* and *suspended*. Being suspended means the EC cannot be scheduled, as its task queues are empty. Submitting task to an EC switches its state as runnable, which indicates the EC can be scheduled. If all the tasks in an EC are blocked, e.g., by I/O stall, the EC switches out and its state is set as blocking. Once receiving new task or the blocked task returns, a blocking EC becomes runnable again. ECs can be externally cancelled or joined. Cancelling an EC will fail the incompleted tasks and suspend it. After an EC is joined, it cannot receive new tasks and suspends itself after its current tasks are completed. ECs are cooperatively scheduled on top the system thread pools, and thus the overhead of context switching is almost negligible.

4.2.3 Federated Query Execution

`Hologres` supports federated query execution to interact with the rich services available from the open source world (e.g., Hive [7] and HBase [6]). We allow a single query spanning `Hologres` and other query systems which are physically isolated in different processes. During query compilation, operators to be executed in different systems are compiled as separate fragments, which are then dispatches to their destination systems by coordinators in `Hologres`. Other systems interacting with `Hologres` are abstracted as special stub WUs, each of which is mapped to an EC uniformly managed in `Hologres`. This stub WU handles pull requests submitted by WUs in `Hologres`. Besides functionality considerations such as accessing data in other systems, this abstraction also serves as an isolation sandbox for system security reasons. For instance, users can submit queries with possibly-insecure user-defined functions. `Hologres` disseminates the execution of these functions to PostgreSQL processes, which execute them in a context physically isolated from other users in `Hologres`.

4.3 Scheduling Mechanism

In this subsection, we introduce details about how WUs of a query are scheduled to produce the query outputs.

Asynchronous Pull-based Query Execution. Queries are executed asynchronously following a pull-based paradigm in `Hologres`. In a query plan, the leaf fragments consume external inputs, i.e., shard files, and the sink fragment produces query outputs. The pull-based query execution starts from the coordinator, which sends pull requests to the WUs of the sink fragments. When processing a pull request, the receiver WU further sends pull requests to its dependent WUs. Once the WU of a read operator, i.e., *column scan*, receives a pull request, it reads a batch of data from the corresponding shard file and returns the results in the format of $\langle record\_batch, EOS \rangle$, where $record\_batch$ is a batch of the result records and EOS is a bool indicating if the producer WU has completed its work. On

receiving results for the previous pull request, the coordinator determines if the query has completed by checking the returned *EOS*. If the query has not completed, it sends out another round of pull requests. A WU depending on multiple upstream WUs needs to pull from multiple inputs concurrently to improve the parallelism of query execution and the utilization of computation/network resource. Hologres supports concurrent pulls by sending multiple asynchronous pull requests. This approach is more natural and efficient compared with traditional concurrency model which requires multiple threads to cooperate.

Intra-worker pull request is implemented as a function call, which inserts a pull task into the task queue of EC hosting the receiver WU. An inter-worker pull request is encapsulated as an RPC call between the source and destination worker nodes. An RPC call contains ID of the receiver WU, according to which the destination worker node inserts a pull task into the task queue of the corresponding EC.

Backpressure. Based on the above paradigm, we implemented a pull-based backpressure mechanism to prevent a WU from being overwhelmed by receiving too many pull requests. First of all, we constrain the number of concurrent pull requests that a WU can issue at a time. Secondly, in a WU which produces outputs for multiple downstream WUs, processing a pull request may results in the production of new outputs for multiple downstream WUs. These outputs are buffered waiting for the pull requests from the corresponding WUs. To prevent the output buffer in a WU growing too fast, the downstream WU that pulls more frequently than others will temporarily slow down sending new pull requests to this WU.

Prefetch. HOS supports prefetching results for future pull requests to reduce the query latency. In such cases, a set of prefetch tasks are enqueued. The results of prefetch tasks are queued in a prefetch buffer. When processing a pull request, results in the prefetch buffer can be immediately returned and a new prefetch task is created.

4.4 Load Balancing

The load balancing mechanism in Hologres are of two folds: (1) migrating TGSs across worker nodes, and (2) redistributing ECs among intra-worker threads.

Migration of TGSs: In our current implementation, read/write fragment instances are always dispatched to the worker nodes hosting the TGS. If one TGS becomes a hotspot, or a worker node is overloaded, Hologres supports migrating some TGSs from the overloaded worker nodes to others with more available resources. To migrate a TGS, we mark the TGS as failed in the storage manager, and then recover it in a new worker node following the standard TGS recovery procedure (see Section 3.2.3). As discussed in Section 3.2.3, we are implementing read-only replicas for TGSs, which enables balancing the read fragment instances to a TGS's read-only replicas located in multiple worker nodes.

Redistribution of ECs: In a worker node, HOS redistributes ECs among threads within each EC pool to balance the workload. HOS performs three types of redistribution: (1) a newly created EC is always assigned to the thread with minimum number of ECs in the thread pool; (2) HOS periodically reassigns ECs between threads such that the difference of the numbers of ECs among threads is minimized; (3) HOS also supports *workload stealing*. Once a thread has no EC to schedule, it "steals" one from the thread which has the maximum number of ECs in the same thread pool. The reassignment of an EC is conducted only when it is not running any task.

4.5 Scheduling Policy

A critical challenge for HOS is to guarantee the query-level SLO in multi-tenant scenarios, e.g., large-scale analytic queries should not block the latency-sensitive serving queries. To solve this problem, we propose *Scheduling Group (SG)* as a virtual resource abstraction for the data-bound and query ECs in a worker node. More specifically, HOS assigns each SG a *share*, whose value is proportional to the amount of resources assigned to this SG. The resources of an SG are further split among its ECs, and an EC can only consume resources allocated to its own SG.

In order to separate the ingestion workloads from query workloads, we isolate data-bound ECs and query ECs into different SGs. Data-bound ECs handle critical operations that need synchronization shared by all queries, and are mainly dedicated to ingestion workload (read-sync WU are usually very light-weight), we group all the data-bound ECs in a single *data-bound SG*. On the contrary, we put query ECs of different queries into separate *query SGs*. We assign the data-bound SG a large enough share to handle all ingestion workload. By default, all the query SGs are assigned of the same share to enforce fair resource allocation. SG shares are configurable.

Given a SG, the amount of CPU time assigned to its ECs in a time interval is impacted by two factors: (1) its share, (2) the amount of CPU time it has occupied in the last time interval. The share of an SG is adjusted according to the status of its ECs in the last time interval, as explained below:

An EC can only be scheduled when it is *runnable*. Denoting the share of EC_i as EC_share_i, we calculate $EC\_share\_avg_i$ to represent the practical share of EC_i in a time interval, while the practical share of SG_i is the sum of the shares of its ECs:

$$EC\_share\_avg_i = EC\_share_i * \frac{\Delta T_{run}}{\Delta T_{run} + \Delta T_{spd} + \Delta T_{blk}}$$

$$SG\_share\_avg_i = \sum_{j=1}^{N} EC\_share\_avg_j$$

ΔT_{run}, ΔT_{spd} and ΔT_{blk} represent the time intervals while EC_i is in the status of *runnable*, *suspend* and *blocking*.

For EC_i in SG_j, we maintain a *Virtual Runtime* reflecting the state of its historical resource allocation. Denoting the CPU time that EC_i is assigned of during the last time interval as ΔCPU_time_i, the increment on EC_i's *Virtual Runtime*, $\Delta vruntime_i$, during the last time interval is calculated as follows:

$$EC\_vshare_i = \frac{EC\_share_i * SG\_share_j}{SG\_share\_avg_j};$$

$$\Delta vruntime_i = \frac{\Delta CPU\_time_i}{EC\_vshare_i}$$

When selecting the next EC to be scheduled, the thread scheduler always selects the one with the minimum *vruntime*.

5. EXPERIMENTS

In this section, we conduct experiments to evaluate the performance of Hologres. We first study the performance of Hologres on OLAP workloads and serving workloads respectively, by comparing it with state-of-the-art OLAP systems and serving systems (Section 5.2). We show that Hologres has superior performance even compared with these specialized systems. Then we present experiment results on various performance aspects of Hologres handling hybrid serving and analytical processing workloads:

- We study in isolation how well the design of Hologres can parallelize and scale when handling analytical workloads or serving workloads alone. We experiment with increasing the workload and the computation resource (Section 5.3).

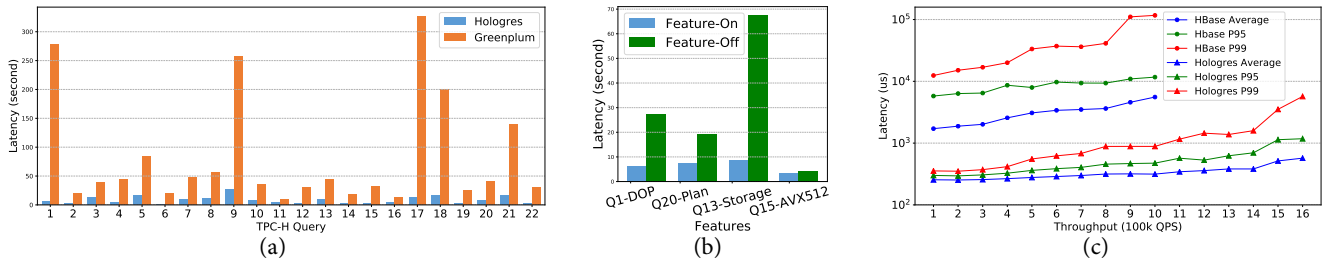

Figure 6: (a) Analytical query latencies of `Hologres` and Greenplum on the TPC-H benchmark. (b) A breakdown study on the effects of `Hologres`'s performance-critical features. (c) Serving query throughputs/latencies of `Hologres` and HBase on the YCSB benchmark.

- We study two aspects of `HOS`'s performance: (1) whether `HOS` can enforce resource isolation and fair scheduling when handling hybrid serving and analytical workloads; (2) whether `HOS` can react in a prompt way to sudden workload bursts (Section 5.4).
- We study the efficiency of `Hologres`'s storage design: (1) the impact of high-speed data ingestion on read performance, and (2) the write latency and write throughput under the maintenance of multiple indexes (Section 5.5).

5.1 Experiment Setup

Workloads. We use the TPC-H benchmark [15] (1TB) to simulate a typical analytical workload, and the YCSB benchmark [17] to simulate a typical serving workload, which contains a table of 100 million records, each record has 11 fields, and each field is 100 bytes. When testing on a hybrid serving and analytical workload on the same data (Sectioin 5.4.1), we use the TPC-H dataset and mix the TPC-H queries with synthetic serving queries (point lookup) on the `lineitem` table. To study under mixed read/write requests, we simulate a production workload in Alibaba, referred to as PW. PW has a shopping cart table that consists of 600 million rows, and has 10^6 updates per second. Each record has 16 fields, and the size of a record is 500 bytes. We replay the updates during the experiment.

System Configurations. We use a cluster consisting of 8 physical machines, each with 24 virtual cores (via hyper-threading), 192GB memory and 6T SSD. Unless explicitly specified, we use this default setting in the experiments on the TPC-H and YCSB benchmarks.

To the best of our knowledge, there is no existing HSAP system. In order to study the performance of `Hologres`, we compared it with specialized systems for analytical processing and serving respectively. For analytical processing, we compared against Greenplum 6.0.0 [5]; for serving, we compared against HBase 2.2.4 [6]. The detailed configurations of each system are explained as follows: (1) The Greenplum cluster has in total 48 segments, which are evenly allocated among 8 physical machines. Each segment is assigned 4 cores. This is the recommended setting from Greenplum's official documentation [11], in consideration of both intra-query (multiple plan fragments in a query) and inter-query concurrency during query execution. Greenplum uses the local disks to store the data files, and the data is stored in column format. (2) The HBase cluster has 8 region servers, each of which is deployed on a physical machine. HBase stores the data files in HDFS, configured using the local disks. HBase stores the data in row format. (3) The `Hologres` cluster has 8 worker nodes, each worker node occupying one physical machine exclusively. To make a fair comparison with Greenplum and HBase, `Hologres` is also configured to use the local disks. The data is stored in both row and column formats in `Hologres`.

The experiments on the PW workload are conducted in a cloud environment with 1,985 cores and 7,785 GB memory. We use Pangu—the remote distributed file system in Alibaba Cloud to store the data. The base data of the shopping cart table is stored in column format. This table also has an index stored in row format.

Experiment Methodology. All the experiments start with a warm-up period of 20 minutes. For every reported data point, we repeat the experiment for 5 times and report the average value.

In the experiments, we use the standard YCSB client for all the experiments on the YCSB data. For experiments on TPC-H and PW data, we implemented a client similar to YCSB. More specifically, the client connections submit query requests asynchronously. We can configure the maximal number of concurrent queries a single connection can submit (denoted as W). Multiple client connections submit query requests concurrently. Unless explicitly specified, we set $W = 100$ throughout the experiments.

5.2 Overall System Performance

In this set of experiments, we study the performance of `Hologres` on analytical workloads and serving workloads respectively, compared against specialized OLAP and serving systems.

Analytical Workloads. In this experiment, we compare `Hologres` and Greenplum using the TPC-H dataset. To accurately measure the query latency, we use a single client and set W to 1. Figure 6(a) reports the average end-to-end latency of the 22 queries.

As shown in the figure, `Hologres` outperforms Greenplum on all the TPC-H queries: the query latency in `Hologres` is on average only 9.8% of that in Greenplum. For Q_1, `Hologres` is 42X faster than Greenplum. The reasons are as follows: (1) HOS enables flexible high intra-operator parallelism for query execution. The read parallelism can go as high as the number of shard files in the tables. The flexibility allows Hologres to have the right parallelism for all queries. On the other hand, GreenPlum's parallelism is determined by the number of segments and cannot make full use of CPU for all queries (e.g., Q1). (2) The layout of column tablets support efficient encoding and indexes. These storage layout optimizations can greatly improve the performance, if the query has filters that can be pushed down to data scan (e.g., Q13). (3) `Hologres` adopts efficient vectorized execution, and can support the *AVX-512* instruction set [8], which can further speed up queries that benefit from vectorized execution[1] (e.g., Q_{15}). (4) `Hologres` can generate better plans, making use of optimizations such as dynamic filters for joins (e.g., Q20). These optimizations together contribute to the improved performance of analytic processing in `Hologres`.

To verify the effect of the above performance-critical techniques (1)-(4), we conduct a breakdown experiment using the TPC-H bench-

[1]We use *AVX-512* mainly in: (1) arithmetic expressions (e.g., addition, subtraction, multiplication, division, equals, not-equals); (2) filtering; (3) bitmap operations; (4) hash value computation; and (5) batch copy.

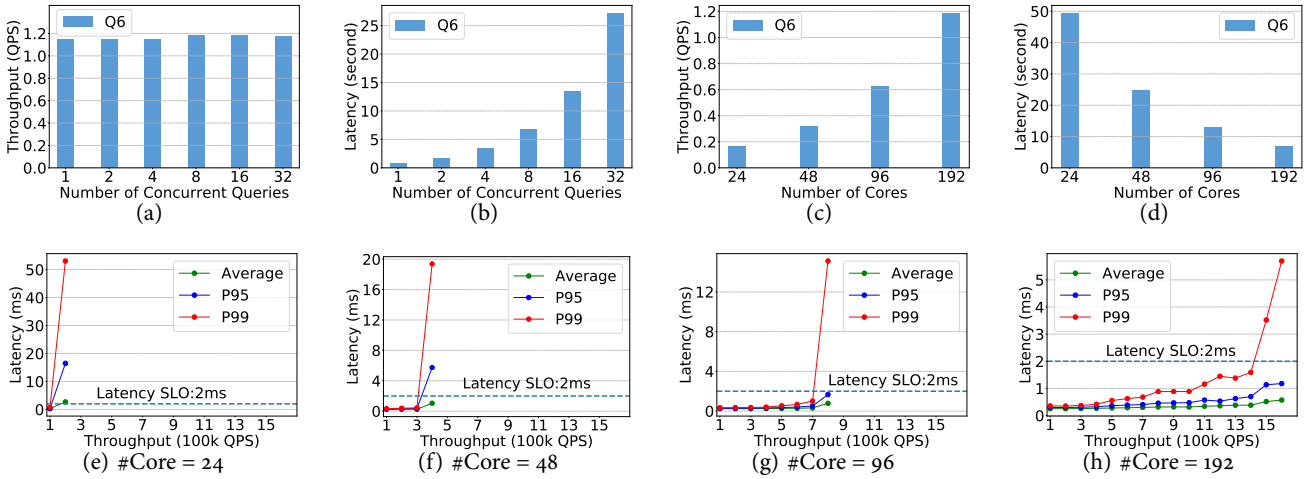

Figure 7: The throughput and latency of analytical workloads under (a)(b) different numbers of concurrent queries and (c)(d) different numbers of cores. (e)(f)(g)(h) The throughput/latency curves of serving workloads under different numbers of cores.

mark. For each technique we choose a representative query, and compare the query latency in `Hologres` with the technique turned on and off. Specifically: For (1), we use Q1, and to turn the feature off we set the parallelism to the number of segments in Greenplum. For (2), we use Q13, and to turn the feature off we disable the dictionary encoding. For (3), we use Q15, and to turn the feature off we use a build without AVX-512. For (4), we use Q20, and to turn the feature off we disable the dynamic filter optimization. The results are reported in Figure 6(b), where (1) (2) (3) (4) are denoted as *Q1-DOP*, *Q13-Storage*, *Q15-AVX512*, and *Q20-Plan* respectively. As we can see, these techniques brings a performance boost from 1.2X to 7.6X.

We also conduct a micro-benchmark on the single-machine performance by comparing `Hologres` with Vectorwise (Actian Vector 5.1 [1]) using the TPC-H benchmark (100GB). The experiment is conducted on a single machine with 32 cores and 128GB memory. It takes `Hologres` 84s to run all the 22 TPC-H queries, while 27s for Vectorwise. This result shows that `Hologres` still has room for performance improvements. However, the optimization techniques in Vectorwise are applicable to `Hologres` and in future work we will integrate them into `Hologres`.

Serving Workloads. In this experiment, we compare `Hologres` and HBase in terms of the throughput and latency using the YCSB benchmark. We gradually increase the query throughput from 100K QPS to 1600K QPS. For each throughput, we report the corresponding *average*, 95% and 99% percentile of query latencies of both systems in Figure 6(c). We set the 99% latency SLO to 100ms, and do not report the data points exceeding the SLO.

First to note that, HBase does not scale to throughputs larger than 1000K QPS, as the query latency exceeds the latency SLO. Whereas, even at 1600K QPS, the 99% latency of `Hologres` is still under 6ms, and the 95% latency is even below 1.18ms. For throughputs under 1000K QPS, the average, 95% and 99% latencies of `Hologres` on average are better than HBase by 10X, 22X and 57X respectively. This is because the thread-based concurrency model in HBase incurs significant context switching overhead when facing highly concurrent serving workload. On the contrary, execution contexts in `Hologres` are very light-weight and can be cooperatively scheduled with little context switching overhead. This design also makes the scheduling well under control, guaranteeing the stability of query latencies. For instance, at throughput = 800K QPS, the 99% latency of HBase is 10.5X higher than its average latency; on the contrary, this difference in `Hologres` is only 1.8X.

The above experiments clearly demonstrate that with the new storage and scheduling design, `Hologres` consistently outperforms state-of-the-art specialized analytical systems and serving systems.

5.3 Parallelism and Scalability of Hologres

Next, we study the parallelism and scalability of `Hologres` when handling analytical workloads and serving workloads respectively.

Analytical Workloads. For analytical workloads, we study two aspects: (1) how well `Hologres` can parallelize analytical queries, and (2) how scalable `Hologres` is with more computation resources. We choose TPC-H Q6 as a representative OLAP query of sequential scans over a large amount of data.

In the first experiment, we use the default cluster setting (8 worker nodes each with 24 cores). We use a single client to submit the queries, but gradually increase the number of concurrent queries W from 1 to 32. The results are reported in Figure 7(a) and 7(b). As we can see, with the number of concurrent queries increasing, the throughput keeps stable. This result clearly shows that even with a single analytical query, `Hologres` can fully utilize the parallelism in the hardware. The latency increases linearly as the resources are evenly shared by all the concurrent queries.

In the second experiment, we fix the number of concurrent queries $W = 8$, but scale out the resources. Specifically, we use 8 worker nodes, and gradually increase the number of cores in each worker node from 3 to 24. The results are presented in Figure 7(c) and 7(d), which show that the throughput increases linearly, and meanwhile the query latency decreases as the number of cores increases. Again, this shows that the high intra-operator parallelism mechanism of `Hologres` can automatically saturate the hardware parallelism.

Serving Workloads. In this set of experiments, we evaluate the throughput and latency of `Hologres` on serving workloads by varying the amount of resources. Again, we use 8 worker nodes, and gradually increase the number of cores from 3 to 24 in each worker node. For each cluster setting, we increase the throughput until the 99% latency exceeds a latency SLO of 2ms. We use 8 clients to continuously submit the queries. We report the corresponding query latencies for each throughput.

The results are presented in Figure 7(e)-7(h) respectively. We have two observations from these figures. First, the maximum through-

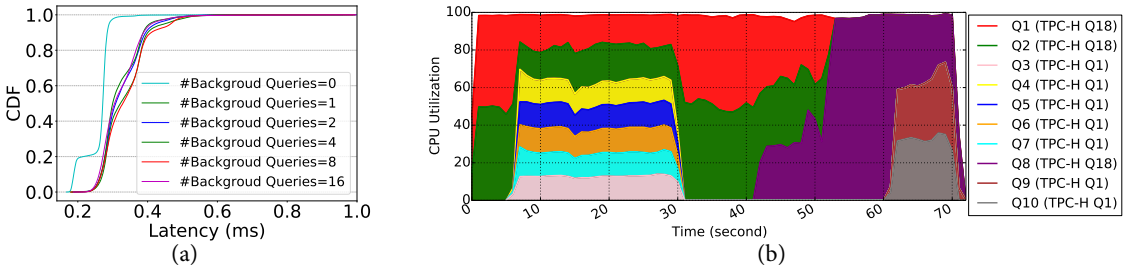

Figure 8: (a) Hybrid workload: the latency CDF of the foreground serving queries under different background analytical workloads. (b) The dynamic shares of CPU time HOS assigned to concurrent queries.

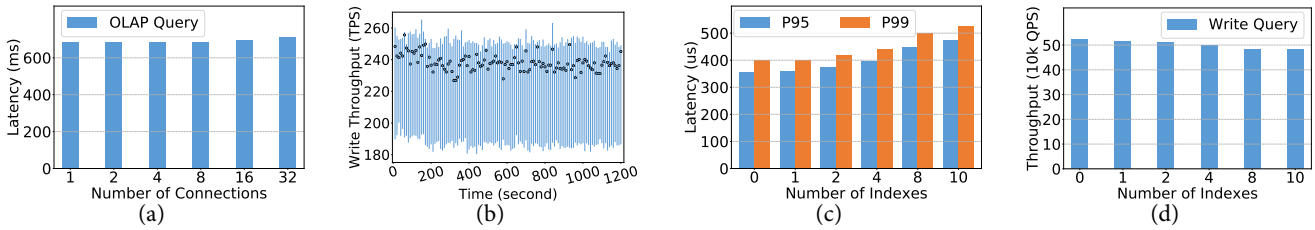

Figure 9: (a) The foreground latency of read queries under different background write workloads. (b) The distribution of per-TGS write throughput over time in the PW workload. (c)(d) The write latency/throughput when maintaining varied numbers of secondary indexes.

put that Hologres can achieve increases linearly as the number of cores increases. For instance, we can see that the maximum throughput at #core=192 is 8 times of the maximum throughput at #core=24. Second, before the system reaches its maximum throughput, the query latencies remain at a stable level. Taking #core=192 as an example, the average, 95% and 99% latencies increase very slowly as the throughput grows. This is due to the fact that Hologres can fully control the scheduling of execution contexts in user space.

5.4 Performance of HOS

In this subsection, we study two performance aspects of HOS: (1) resource isolation under hybrid serving and analytical workloads, and (2) scheduling elasticity under sudden workload bursts.

5.4.1 *Resource Isolation under Hybrid Workloads*

A key scheduling requirement in HSAP services is that the latency-sensitive serving queries are not affected by resource-consuming analytical queries. To study this, we generate a hybrid serving/analytical workload that has two parts: (1) **background**: We continuously submit analytical queries (TPC-H $Q6$ with different predicates) in the background. We vary the background workloads by increasing the number of concurrent queries W from 0 to 16. (2) **foreground**: We submit serving queries in the foregound and measure the query latency. To accurately test the latency, we set the number of concurrent queries $W = 1$. For each setting of the background workloads, we collect 50K data points, and plot their CDF.

Figure 8(a) presents the results. We can see that: by increasing the number of background queries from 0 to 1, there is a small increment on the latency of serving query; but further increasing the background worklaods (from 1 to 16) brings no increment.

It clearly shows that resources allocated to different queries are well isolated by HOS, because execution contexts of different queries are grouped into separate scheduling groups. Therefore, analytical queries and serving queries can coexist in the same system while both their latency SLOs can still be fulfilled.

5.4.2 *Scheduling Elasticity under Sudden Bursts*

In this experiment, we demonstrate how well HOS can react to suddern workload bursts. The experiment is started by concurrently issuing $Q1$ and $Q2$ at time 0. At time 5, we issue 5 new queries ($Q3$-$Q7$). $Q3$-$Q7$ finish roughly at time 30. At time 40, query $Q8$ enters the system. $Q1$ and $Q2$ finish roughly at time 50. In the end, at time 60, we submit $Q9$ and $Q10$, and leave $Q8$-$Q10$ run to completion. All the queries are assigned with equal priorities. Figure 8(b) shows the fraction of CPU used by each query along the timeline.

Note that at time 5, HOS quickly adjusts the resource assignment so that all the seven queries have an equal share of CPU. At time 30, after $Q3$-$Q7$ finish execution, HOS immediately adjusts the scheduling and reassigns CPU equally between $Q1$ and $Q2$ that are still running. Similar behaviors can be observed at time 40, 50 and 60. This experiment highlights that HOS can dynamically and promptly adjust its scheduling behaviors according to the real-time concurrent workloads in the system, always guaranteeing fair sharing.

5.5 Performance of Hologres Storage

In this set of experiments, we evaluate the effects of read/write separation on query latency and study the write performance under index maintenance in Hologres .

5.5.1 *Separating Read/Write Operations*

To study the impacts of writes on query latency, we generate a mixed read/write workloads on the PW workload consisting of two parts: (1) **background:** We replay the tuple writes in PW to simulate a 20-minute background workloads. We vary the write throughput by increasing the number of write clients from 1 to 32. The writes are uniformly distributed across TGSs. E.g., for the case that the number of write clients is 32, we sample the write throughput every 10 seconds, and report the average/min/max write throughputs among all the TGSs in Figure 9(b). (2) **foreground:** We use 16 clients to submit OLAP queries as the foregound workloads. To accurately measure the query latency, each client has its W set to 1. We re-

port the average query latency at each throughput setting in Figure 9(a). As shown, the latency of the OLAP queries is stable despite of the increase on write throughputs. This result evidences that high-throughput writes has little impact on query latencies. This is because of the read/write separation in Hologres. The versioned tablets guarantee that reads are not blocked by writes.

5.5.2 Write Performance

Next, we study the write performance of Hologres under index maintenance using the YCSB benchmark, where we create a number of secondary indexes for the YCSB table. We vary the number of secondary indexes from 0 to 10. For each setting, we push the system to its maximum write throughput and report the 95% and 99% percentile of the write latencies.

As shown in Figure 9(c) and 9(d), as the number of indexes increases, the write latency and the write throughput keep rather stable, and only change slightly. Compared to the case with no secondary index, maintaining 10 secondary indexes only incurs a 25% increment on the write latency and a 8% decrement on the write throughput. This result shows that index maintenance in Hologres is very efficient and has very limited impact on write performance. The main reasons are three folds: (1) Hologres optimizes the write performance by sharing a WAL among all the index tablets in a TGS. Therefore, adding more indexes does not incur additional log flushes. (2) For each write to a TGS, each index is updated by a separate write-apply WU in parallel. (3) Hologres aggressively parallelizes operations such as memory table flushes and file compactions by offloading them to the background EC pool. With enough computation resources, this design removes performance bottleneck.

6. RELATED WORK

OLTP and OLAP Systems. OLTP systems [10, 12, 35] adopt row store to support quick transactions which frequently perform point lookups over a small number of rows. OLAP systems [34, 37, 14, 27, 24, 22, 36] utilize column store to achieve efficient column scans, which is the typical data access pattern in analytic queries. Unlike the above OLTP/OLAP systems, Hologres supports hybrid row-column storage. A table can be stored in both the row and column storage formats to efficiently support both point lookup and column scans required by HSAP workloads.

MPP databases like Greenplum [5] usually partition data into large segments, and co-locate the data segments with the computing nodes. When scaling the system, MPP databases usually need to reshard the data. Conversely, Hologres manages data in TGSs which is a much smaller unit segments. Hologres maps TGSs dynamically to worker nodes, and can flexibly migrate between worker nodes without resharding the data. Also, the worker nodes only need to keep the memory tables of the hosted TGSs in memory, but fetch TGS's shard files from the remote file system on demand. In terms of multi-tenant scheduling, [5] handles different requests in different processes and relies on the OS to schedule concurrent queries, easily putting a hard limit on the query concurrency. Instead Hologres multiplexes concurrent queries on a set of user space threads, achieving much better query concurrency.

[31, 29] study the highly parallel query processing mechanisms for analytical workloads. They decompose query execution into small tasks and schedules tasks across a set of threads pinned in physical cores. Hologres takes a similar high parallel approach. But Hologres uses a hierarchical scheduling framework, and the abstraction of work units reduces the complexity and overheads when scheduling a large number of tasks in a multi-tenant scenario. Execution contexts and scheduling groups provide a powerful mechanism to ensure resource isolation across different tenants. [19] discusses a CPU sharing technique for performance isolation in multi-tenant databases. It emphasizes an absolute CPU reservation that is required in Database-as-a-Service environments. While, Hologres only requires relative CPU reservation, which is enough to prevent analytical queries form delaying serving queries.

HTAP Systems. In recent years, with the fast increasing needs for more real-time analysis, we have seen a lot of research interest on providing Hybrid Transactional/Analytical Processing (HTAP) solutions over big data sets. [33] studies how the hybrid row and column format helps improve the databases' performance for queries with various data access patterns. Follow-up systems such as SAP HANA [21], MemSQL [9], HyPer [23], Oracle Database [25] and SQL Server [20, 28] support both transactional and analytical processing. They usually use row formats for OLTP and column formats for OLAP, but require converting the data between row and column formats. Due to these conversions, newly committed data might not be reflected in the column stores immediately. On the contrary, Hologres can store tables in both row and column tablets, and each write into a table updates both types of tablets at the same time. Hologres parallelizes writes to all tablets at the same time to achieve high write throughput. In addition, HSAP scenarios have much higher ingestion rates than transaction rates in HTAP scenario (e.g., users usually generate tens of page view events before making a purchase transaction), but usually with a weaker consistency requirement. Hologres deliberately only supports atomic write and read-your-write read, which achieves a much higher read-/write throughput by avoiding the complex concurrency control.

[32] studies task scheduling for highly concurrent workloads in HTAP systems. For OLTP workloads, it adapts concurrency level to saturate CPU as OLTP tasks include heavy usage of synchronization. However, Hologres adopts a latch-free approach and avoids frequent blocking. For OLAP workloads, it uses a concurrency hint to adjust task granularity for analytical workloads, which can be integrated into Hologres to schedule execution contexts.

NewSQL. The sharding mechanism adopted in Hologres is similar to BigTable [16] and Spanner [18]. BigTable uses the abstraction of table tablet to facilitate range search over sorted data. Spanner is a globally-distributed key-value store supporting strong consistency. The data shard in Spanner is used as the basic unit for maintaining data consistency with the existence of distributed data replication. Unlike Spanner which is mainly used as an OLTP solution, Hologres deliberately chooses to support a weaker consistency model for HSAP scenarios to chase for better performance.

7. CONCLUSION & FUTURE WORK

There are a host of new trends towards a fusion of serving and analytical processing (HSAP) in modern big data processing. In Alibaba, we design and implement Hologres, a cloud-native HSAP service. Hologres adopts a novel tablet-based storage design, an execution context-based scheduling mechanism, as well as a clear decoupling of storage/computation and reads/writes. This enables Hologres to deliver high-throughput data ingestion in real-time and superior query performance for the hybrid serving and analytical processing. We present a comprehensive experimental study of Hologres and a number of big data systems. Our results show that Hologres outperforms even state-of-the-art systems that are specialized for analytical or serving scenarios.

There are a number of open challenges for even higher performance in HSAP. These challenges include better scale-out mechanism for read-heavy hotspots, better resource isolation of memory subsystem and network bandwidth, and absolute resource reservation in distributed environments. We plan on exploring these issues as part of future work.

8. REFERENCES

[1] Actian vector. https://www.actian.com.

[2] Apache arrow. https://arrow.apache.org.

[3] Apache hdfs. https://hadoop.apache.org.

[4] Flink. https://flink.apache.org.

[5] Greenplum. https://greenplum.org.

[6] Hbase. https://hbase.apache.org.

[7] Hive. https://hive.apache.org.

[8] Intel avx-512 instruction set. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[9] Memsql. http://www.memsql.com/.

[10] Mysql. https://www.mysql.com.

[11] Pivotal greenplum. https://gpdb.docs.pivotal.io/6-0/admin_guide/workload_mgmt.html.

[12] Postgresql. https://www.postgresql.org.

[13] Rocksdb. https://github.com/facebook/rocksdb/wiki.

[14] Teradata. http://www.teradata.com.

[15] Tpc-h benchmark. http://www.tpc.org/tpch.

[16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC 2010, New York, NY, USA, 2010. Association for Computing Machinery.

[18] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, and et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug. 2013.

[19] S. Das, V. R. Narasayya, F. Li, and M. Syamala. CPU sharing techniques for performance isolation in multitenant relational database-as-a-service. *PVLDB*, 7(1):37–48, 2013.

[20] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[21] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[22] J.-F. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li, and et al. Pinot: Realtime olap for 530 million users. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[23] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011.

[24] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[25] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1253–1258, 2015.

[26] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), Apr. 2010.

[27] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[28] P.-r. 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.

[29] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2014, New York, NY, USA, 2014. Association for Computing Machinery.

[30] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys 2012, New York, NY, USA, 2012. Association for Computing Machinery.

[31] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

[32] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task scheduling for highly concurrent analytical and transactional main-memory workloads. In *Proceedings of the Fourth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2013)*, number CONF, 2013.

[33] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *Proceedings of the 28th International Conference on Very Large Data Bases*, page 430–441. VLDB Endowment, 2002.

[34] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[35] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[36] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2014, New York, NY, USA, 2014. Association for Computing Machinery.

[37] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35:21–27, 2012.