KRYPTON: Real-time Serving and Analytical SQL Engine at ByteDance

Jianjun Chen*, Rui Shi, Heng Chen, Li Zhang*, Ruidong Li, Wei Ding*, Liya Fan, Hao Wang*, Mu Xiong, Yuxiang Chen*, Benchao Dong, Kuankuan Guo, Yuanjin Lin, Xiao Liu*, Haiyang Shi*, Peipei Wang*, Zikang Wang, Yemeng Yang, Junda Zhao, Dongyan Zhou, Zhikai Zuo, Yuming Liang
*ByteDance US Infrastructure System Lab, ByteDance, Inc
jianjun.chen@bytedance.com

ABSTRACT

In recent years, at ByteDance, we have started seeing more and more business scenarios that require performing real-time data serving besides complex Ad Hoc analysis over large amounts of freshly imported data. The serving workload requires performing complex queries over massive newly added data items with minimal delay. These systems are often used in mission-critical scenarios, whereas traditional OLAP systems cannot handle such use cases. To work around the problem, ByteDance products often have to use multiple systems together in production, forcing the same data to be ETLed into multiple systems, causing data consistency problems, wasting resources, and increasing learning and maintenance costs.

To solve the above problem, we built a single Hybrid Serving and Analytical Processing (HSAP) system to handle both workload types. HSAP is still in its early stage, and very few systems are yet on the market. This paper demonstrates how to build KRYPTON, a competitive cloud-native HSAP system that provides both excellent elasticity and query performance by utilizing many previously known query processing techniques, a hierarchical cache with persistent memory, and a native columnar storage format. KRYPTON can support high data freshness, high data ingestion rates, and strong data consistency. We also discuss lessons and best practices we learned in developing and operating KRYPTON in production.

PVLDB Reference Format:
Jianjun Chen, Rui Shi, Heng Chen, Li Zhang, Ruidong Li, Wei Ding, Liya Fan, Hao Wang, Mu Xiong, Yuxiang Chen, Benchao Dong, Kuankuan Guo, Yuanjin Lin, Xiao Liu, Haiyang Shi, Peipei Wang, Zikang Wang, Yemeng Yang, Junda Zhao, Dongyan Zhou, Zhikai Zuo, Yuming Liang. KRYPTON: Real-time Serving and Analytical SQL Engine at ByteDance. PVLDB, 16(12): 3528 - 3542, 2023.
doi:10.14778/3611540.3611545

1 INTRODUCTION

In recent years, at ByteDance, we have started seeing more and more business scenarios that require performing real-time data serving besides complex Ad Hoc analysis over enormous amounts of freshly imported data. For example, ByteDance's Ads AB Testing team maintains a high-dimension dashboard that needs to perform aggregation and filtering over millions of newly ingested rows per second in a real-time fashion. Feature serving in *ZhuXiaoBang* (a popular mobile App for providing house renovation information in China [40]) provides a real-time feature extraction service for the upper-layer recommendation system, which needs to perform real-time aggregation on user-specified features with millisecond-level latency over arbitrary time windows.

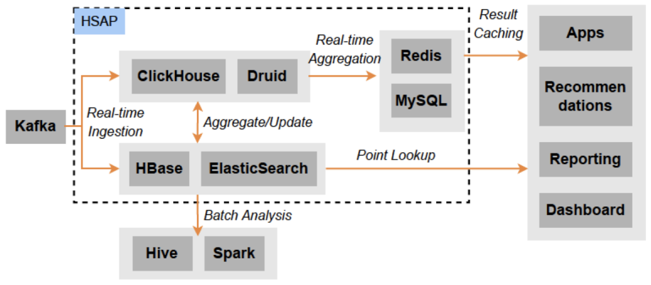

Figure 1: An illustration of a typical Ads Backend architecture with open source solutions.

Typical data serving use cases in ByteDance include:

- **Real-time statistics** They can be the number of likes or shares displayed in real-time or statistics of content to a third party through APIs.
- **Real-time user profiling** Our systems continuously build short-term and long-term user profiles. Short-term user profiles can be top-N clicked items in the last ten refreshes, while long-term user profiles update in batches on a daily basis.
- **Real-time model training/learning** In support of online learning and decision-making, we aggregate real-time signals (e.g., payments and user clicks) to continuously derive user statistics, which are heavily used in online and offline training. The derived metrics are also used in complex interactive analysis to derive insights for model tuning and marketing.

Traditional OLAP systems perform complex data analytics over relatively static data, where data freshness is usually measured by hours or even days. The serving workload, however, requires performing complex queries over a large amount of newly added data items with sub-second delays. Such systems are often used in mission-critical scenarios that traditional OLAP systems cannot handle. To work around the above problem, ByteDance often uses multiple systems together in production. Figure 1 shows a production setup of an Ads backend system, typically including a streaming ingestion pipeline and a batch analysis system. The former ingests and aggregates data into systems such as Druid [85] or

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611545

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

ElasticSearch (ES) [30] for online analysis. The latter uses Spark [78] or Hive [37] to perform batch analysis over large amounts of batch data, then loads the results into the data warehouse for offline analysis. To support high QPS and low latency serving workloads, systems like Redis [71] or MySQL [63] are often used as the front caching layer.

Such an approach forces the same data to be ETLed into multiple systems, causes data consistency problems, wastes resources, and incurs higher learning and maintenance costs. In addition, each system has its own strengths and drawbacks. For example, ES performs well with point lookup queries but needs to improve on complex query logic. In contrast, ClickHouse [19] performs well on aggregation queries but has limited data ingestion and query throughput support. As a result, optimizing these systems requires addressing their limitations individually, which incurs duplicated efforts and does not scale well.

To address the above challenges in a unified manner, we aim to build one single Hybrid Serving and Analytical Processing (HSAP) system to handle both workloads. In particular, these two workloads are expressible in SQL queries with some extensions. Compared to the OLAP workload, serving workload queries are usually much simpler. They are often known with some fixed patterns in advance, providing great opportunities for performance optimization. However, the serving workload usually has significantly higher QPS and lower query latency requirements than the OLAP workload. Another huge challenge is that data are ingested into the system continuously at high throughput while performing the analytical workload above.

Specifically, we have built a large-scale real-time HSAP system that supports both serving and OLAP workloads with the following design goals:

- **Large scale**. Several ByteDance's popular apps, such as TikTok, Douyin, and Toutiao, have hundreds of millions of daily active users. Hence, we want to build a distributed real-time analytic system that can scale up to hundreds of petabytes of data.
- **Low query latency and high QPS**. The serving workload has very different characteristics from the traditional OLAP workload. Some of our serving use cases require up to millisecond-level query latency and millions of QPS.
- **Strong data consistency**. Our customers want consistent snapshot reads and atomic writes when importing data across partitions.
- **High data freshness**. Some of our customers want to query newly ingested data under sub-second delays.
- **High ingestion rate**. Some of our customers want to ingest millions of rows per second.
- **Standard SQL support**. It is important to support common SQL standards so customers can readily migrate their applications to our HSAP system.

Hybrid Transactional and Analytical Processing (HTAP) systems [7–10, 17, 38, 46, 47, 52, 56, 57, 60, 65, 70, 74] also provide real-time data analytics with high data freshness. Unlike HTAP systems, KRYPTON focuses on efficiently handling various ingested data sources and providing millisecond-level low query latency over many serving queries without strong OLTP capabilities such as transactions and general DML support in HTAP systems.

HSAP is still in its early stage, and very few systems are yet on the market. To the best of our knowledge, Procella [15] and Hologres [42] are the only two systems falling into this category. In this paper, we will showcase our journey of building the KRYPTON system that can support both real-time serving and analytical workloads. Even though many of the techniques we used in KRYPTON can be found in various database and data processing systems, it is nevertheless a challenging and exciting exploration to build a new HSAP system. While some of our techniques such as Lightweight API and Dirty Read are specifically designed for serving queries, most of Krypton's techniques such as caching and pre-computation, adaptive statistics with dynamic sampling, resource isolation, and fair scheduling are useful for both serving and analytical queries. We started the design and development at the beginning of 2021 and released version 1.0 in early 2022. By now, we already have multiple internal customers using KRYPTON in production, and we expect many more customers will adopt it in 2023. After migrating to KRYPTON, our customers see great benefits over their original systems. For example, ByteDance's Webcast team performs real-time data analysis that involve joins and aggregations to promote the interaction between the anchor and the audience. After migration from Doris (0.14 version) to Krypton, the P99 query latency is reduced from approximately 5 seconds during the peak business period to less than 1 second. We have learned tremendously from this journey and hope that our story can be helpful to readers with similar needs.

In summary, our key contributions are as follows:

- We demonstrate how to build a competitive cloud-native HSAP system with disaggregated storage, separate data ingestion and query processing, and a hierarchical local cache with Persistent Memory (PMem) at query processing nodes to provide excellent elasticity and query performance.
- KRYPTON can provide high data freshness with sub-second delays. Moreover, during data ingestion, it can perform pre-computations to generate aggregated tables and materialized views that are critical for serving workloads.
- KRYPTON adopts efficient and adaptive query processing techniques for diversified HSAP workloads. It uses incremental statistics and dynamic sampling to obtain up-to-date lightweight statistics information for query optimization. In addition, it heavily relies on caching to improve query performance whenever possible. Lastly, it adopts a vectorized query executor with asynchronous schedulers for efficient query execution over high concurrent query workloads.
- KRYPTON has its own columnar storage format that provides efficient storage and fast query processing for HSAP workloads.
- We talk about lessons we have learned and our best practices in developing and running the KRYPTON system in production.

The rest of the paper is organized as follows: Section 2 describes the overall architecture of KRYPTON and the implementations of key components. Section 3 focuses on how KRYPTON performs pre-aggregations at ingestion time. Section 4 describes the efficient and adaptive query processing in KRYPTON. Section 5 shows the hierarchical cache with PMem, and Section 6 presents the native data format inside KRYPTON. Section 7 provides some empirical measurements of KRYPTON. Section 8 lists some of the major lessons

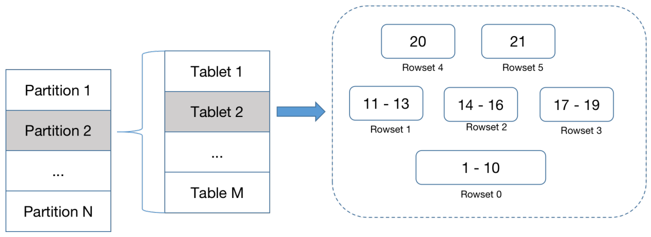

Figure 2: Krypton's Data Model

we have learned in production. Section 9 gives an overview of related work. Finally, Section 10 concludes our work.

2 DATA MODELS AND SYSTEM ARCHITECTURE

In this section, we give an overview of Krypton system architecture after describing its data models.

2.1 Data Models

As Figure 2 shows, Krypton tables support standard two-level data partitioning: *Partitions*, and *Tablets*. Both levels support partitioning policies of hash, range, and list. *Tablets* are stored in different directories in the cloud storage. Each tablet can contain multiple rowsets, in which rows are sorted by keys defined in the table schema. A rowset with a committed version is generated when a memtable flushing happens. Once a rowset is written, it becomes immutable. In addition, data compaction can merge multiple rowsets into a single rowset with a contiguous version range that consists of the minimum and maximum versions of the merged rowsets. A tablet's committed version is the greatest version of all rowsets within the tablet. In the example tablet shown in Figure 2, tablet-2 contains six rowsets, and its committed version is 21, which is the version of the rowset 5.

Each query in Krypton carries a version obtained from its Metadata Server that defines its read snapshot. Multiple rowsets in a tablet are merged when a tablet scan happens. The merge algorithm varies according to the table model. In support of different use cases, Krypton supports the following table models:

- *Duplicate Table*: The same row can be inserted multiple times and saved duplicated with this table model.
- *Unique Table*: The system needs to define Primary Key (PK) in DDL with this table model.
- *Aggregate Table*: Similar to Unique Table, Aggregate Table also needs a PK. Rows with the same PK are merged according to aggregate functions defined on their own column. Note that only aggregated data is stored inside an aggregated table.

2.2 System Architecture

Figure 3 shows the architecture diagram of Krypton. Krypton adopts a typical cloud system architecture by decoupling computation from its cloud storage. Both data and metadata are stored remotely. The Cloud Store uses ByteDance's HDFS system that can scale almost infinitely. Compute modules are stateless and can elastically scale based on workload independently from storage.

Krypton separates data loading from query execution to isolate reads from writes. For the write path, streaming input data commit

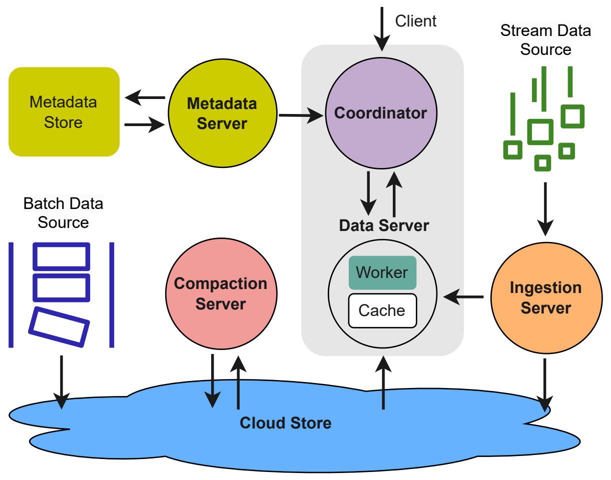

Figure 3: Krypton System Architecture

after corresponding Write-Ahead Logs (WALs) are flushed successfully by Ingestion Servers (IgSs) to Cloud Store. Each IgS contains an in-memory delta store, which periodically flushes its data to Cloud Store in columnar format. It can also serve direct read from Data Servers (DSs) for improved data freshness. Data tablets are partitioned across IgSs to improve write scalability. The Compaction Servers (CSs) periodically merge column files in Cloud Store and write the merged files back to Cloud Store. In Krypton, clients send read-only queries to Coordinators, which generate query plans and send them to Data Servers for query execution. Krypton's query processor adopts a Massive Parallel Processing (MPP) architecture for easy scale-out. IgSs and DSs can scale independently based on read and write workloads.

Cache and pre-computations are crucial to achieve great performance and scalability to support stringent serving workload requirements. In Krypton, we adopt the principle of caching everything, including data, query results, query plans, and metadata. Each DS has a local hierarchical cache that utilizes multiple storage media to cache data blocks, including DRAM, PMem, and SSD. When a DS processes a scan request, if the data is not in the cache, it will fetch from the remote Cloud Store using RDMA. In order to process serving workloads, we expect almost all data to be hit in the cache in Krypton. Krypton also allows users to specify preload options for their tables that guarantee data in those tables are loaded into the cache when the system starts. Krypton's hierarchical cache provides an excellent performance/price ratio that fits various business scenarios. In addition, Krypton allows users to define pre-aggregated tables and materialized views, which perform pre-computation at data ingestion time, to improve query performance. Both Data Servers and Ingestion Servers use affinity partitioning for better cache utilization.

An efficient and adaptive query processor is vital for satisfying Krypton's query performance requirements. Krypton's query optimizer (QO) adopts a rule-based approach with some cost-based extension as needed. It dynamically collects up-to-date statistics through incremental statistics and dynamic sampling. Plan caching and query hints are very useful for serving workloads. Krypton's query execution engine adopts a vectorized query execution with two-level asynchronous scheduling using coroutines. In addition, it provides adaptive parallelism, special lightweight APIs for serving queries, and resource isolation to support the HSAP workload.

Krypton also has a native storage format for column files to support efficient encoding for nested data types. It contains secondary data structures for both fast lookup and scan operations. In addition, it supports various encoding/decoding and compression/decompression algorithms and is deeply integrated with query engine.

Finally, customized configuration is critical to the Krypton's design since it enables customers to have trade-offs among data freshness, query performance, and costs. For example, some cost-based query optimization rules only apply to OLAP but are disabled in serving workloads.

3 INGESTION AND PRE-COMPUTATION

Krypton supports user data loading in two ways: streaming mode and batch mode. For better data freshness, rows are normally inserted in streaming mode. A write is considered a success after it persists in WAL. Meanwhile, WAL is consumed asynchronously and applied to delta store in memory. The delta store is converted to column format and flushed to the Cloud Store when its size reaches the predefined threshold. Versions of newly flushed tablets are increased in Metadata Server before they become visible to users. Regarding batch writes, each batch is split into multiple parts according to the data partition policy defined in the table schema, then flushed into the Cloud Store. All tablet-committed versions of this batch are modified atomically in Metadata Server.

In addition, Krypton can import external columnar files in a bulk-load fashion. Krypton maintains two kinds of indexes during ingestion: 1) built-in indexes, such as prefix index and ZoneMap index; 2) user-defined secondary indexes, such as BloomFilter index and BitMap index. In addition, incremental stats, such as row count and the number of distinct values, are collected during ingestion and stored in Metadata Server.

Data preloading can be configured at the table level to preload data into the cache of Data Servers before being queried. For such tables, once new data is flushed to Cloud Store successfully, Ingestion Server notifies Metadata Server to invoke corresponding Data Servers to load data into their cache as part of the data registration step. When the registration request finishes, the data is visible to users and can be accessed from DS cache module. Enabling cache preloading on one production cluster has observed the P99 query latency drops from 19 ms to 2.6 ms and the average latency reduces from 3.6 ms to 1 ms.

Materialized View (MV) plays an essential role in serving workloads. It is like a base table with its own schema, such as distribution keys and data models. Currently, Krypton supports MVs defined on single tables and maintains them in real time during data ingestion.

Automatic Data Model Derivation. The data model is important for MVs as base tables in Krypton. An improper data model could adversely impact the performance of related queries and may even lead to wrong query results. Take the following MV as an example:

```
CREATE MATERIALIZED VIEW test_mv AS
  SELECT a, COUNT(DISTINCT b) FROM base_table
  GROUP BY a;
```

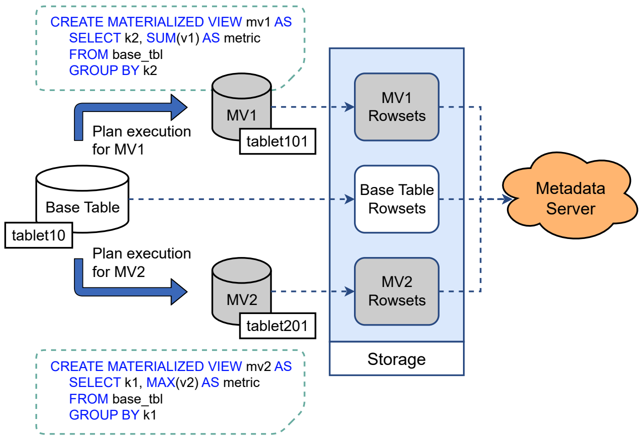

Figure 4: Materialized View Maintenance

Because the aggregation function used here is *count distinct*, Krypton does not yet support pre-aggregation. The query results will be wrong if an aggregate table is used as the data model of *test_mv*. Instead, we should choose a duplicated table model for *test_mv* to retain the original data from the base table.

It is challenging for users to manually define the data model over MVs when they create MVs. To address this, in Krypton, the optimizer automatically deduces the optimal data model for an MV based on the base table data model and the aggregation function (if any) used in the MV query definition. To be specific, we classify aggregation functions into two types: 1) splittable aggregation functions: for these functions (e.g., sum, min, and max), the final result is obtained by splitting the data into disjoint partitions, aggregating each partition separately, and merging partial results to the final result; 2) un-splittable aggregation functions, for these functions, we cannot easily obtain the final result from partial results in our current implementation. Therefore, we can only use duplicate table as the table model for MVs with un-splittable functions. Due to the space limit, we omit the detailed rules for the data model derivation for MVs in this paper.

Materialized View Maintenance. In Krypton, the real-time data consistency between base tables and related MVs is maintained by Ingestion Server on the tablet level. An execution plan for each MV generated by the query optimizer during MV creation is stored inside Metadata Server. Ingestion Server maintains one active memtable for each base table tablet. Once a memtable is ready for flush, the Ingestion Server verifies whether the related table has any associated MVs. If yes, it retrieves the execution plans from Metadata Server and executes them over the tablet.

Figure 4 shows the case that a base table has two materialized views *mv1* and *mv2*. After transformation, the data are written into two materialized view memtables: memtable of tablet 101 from *mv1*, and memtable of tablet 201 from *mv2*. All three memtables will be flushed into the Cloud Store as new rowsets without particular order. If all rowsets persist successfully, the Ingestion Server requests Metadata Server to register the three new rowsets together. If failure happens during the flush process, no changes will happen in Metadata Server, and the persisted rowsets will eventually

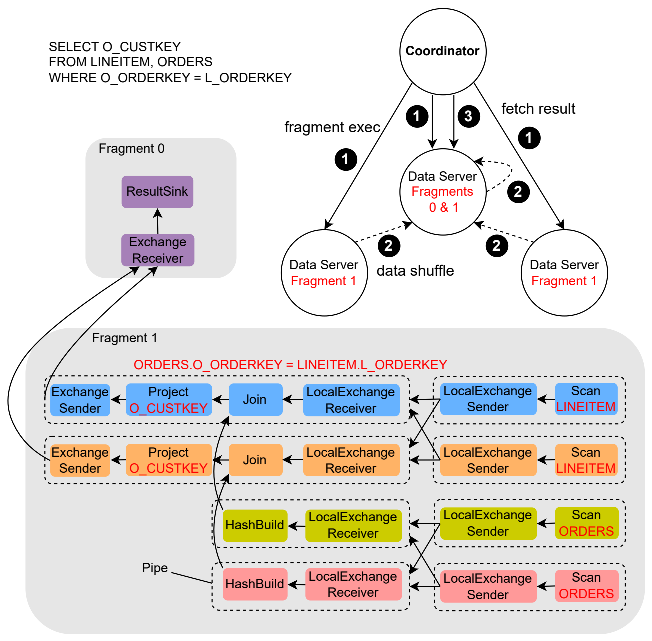

Figure 5: Krypton's Query Execution Workflow

be garbage collected. The metadata of the three tablets will be updated simultaneously in Metadata Server atomically to maintain the data consistency between the base table and materialized views. To give readers a glimpse of the overhead in maintaining materialized views, the impact on data freshness is within 10% in a scenario of one base table with five materialized views.

4 EFFICIENT AND ADAPTIVE QUERY PROCESSING

Figure 5 illustrates how Krypton executes a query. The query optimizer and scheduler colocate at a designated node called *Coordinator*. After optimization, the query scheduler divides the plan into a set of *fragments* in a distributed fashion and sends them to a set of Data Servers for execution. Whenever a *block* (i.e., a batch of rows) is produced by its local fragment executor, it will be shuffled to its destination DS(s) for future processing. The final result blocks will be buffered at a dedicated DS and fetched by the Coordinator in time. In the example shown in Figure 5, the query plan is divided into two query fragments, *fragment 0* and *fragment 1*, which are responsible for gathering all the result blocks and performing the co-located join, respectively. In addition, *fragment 1* consists of a set of *pipes*, each of which contains a set of operators that can execute without any stalls [64]. To repartition/shuffle blocks between different parallel executing pipes of the same type, we introduce a particular operator called the *local exchange*. This operator enables us to exploit parallelism among different pipes with significantly less overhead than the *exchanger* operator, which usually brings extra network and serialization costs.

The Krypton query processor is designed with the HSAP workload in mind from the start. We aim to minimize planning time and generate query plans towards high QPS for serving workloads while considering more plan alternatives and maximizing parallelism to reduce latency for OLAP queries. Krypton employs a push-based and vectorized execution engine with a coroutine-based asynchronous scheduler (i.e. *coro-scheduler*) as its core to achieve optimal execution performance for HSAP workloads.

The rest of this section describes several critical techniques Krypton adopted for efficient and adaptive query processing.

4.1 Query Cache and Adaptive Statistics with Dynamic Sampling

Krypton uses cache extensively to avoid repetitive query computation to save planning and execution time. It has a unified cache framework, defined as a hashmap, for storing query plans, query fragment selectivity estimation, and query results. We include the data version into part of the cache key entry to avoid using the stale cache entry when the data version changes. Regarding cache efficiency, we have a background cache maintenance manager which periodically compares the data version with Metadata Server and proactively removes stale cache entries. Many of our production workloads contain repetitive queries and we observed some workloads have benefited from this feature with an average of above 50% plan cache hit ratio in production.

Besides caching query plans which is common in database systems, Krypton also maintains a *stats cache* for adaptive selectivity estimation. For a cache entry of *stats cache*, the key is a query fragment with a filter condition, and the value is its selectivity. Additionally, we have a query result cache to further reduce execution time. Note that the query result cache is shared among sessions, so clients issuing the same query share the same result set. The query result cache fits comfortably the environments with many identical queries over the rarely updated tables, a typical situation for Web servers that generate many dynamic pages based on database content. Krypton's query result cache relieves customers from implementing their cache at the application level. We have observed a 15% QPS increment on average for our *ZhuXiaoBang* customer workload.

Krypton dynamically maintains up-to-date statistics, including table row counts and the number of distinct values (NDV) in a column (i.e., column NDV), incrementally without any hassle from the users. As mentioned in Section 3, Krypton Ingestion Server can pre-process the data during the data ingestion phase, which Krypton utilizes to maintain our statistics incrementally. The table row count is trivial to compute by summing the newly ingested rows with previous row counts stored in our metadata. For column NDV, we use the HyperLogLog (HLL) algorithm [31, 36] to build approximate column NDV incrementally during the flush process, which is much more accurate and efficient than some other DBMSs' sampling-based NDV estimation algorithms.

Krypton does not rely on static tablet stats to estimate filter selectivity, which is often hard to be accurate. Instead, it issues a sample query plan fragment with the actual filter condition during query planning time to collect the count information. In Krypton, the sampling granularity is configurable. Testing on the TPC-H 1T dataset shows that with dynamic sampling our stats estimation is within 1% difference from the actual runtime stats but the overhead is less than 2% of the total running time.

Still, the sampling approach has its own limitations. First, for the skewed dataset, the sampled filter selectivity may not be adequate to represent the whole table selectivity. Second, the sampling approach does incur additional overhead in executing the sampling query, which may not be negligible in some serving workloads.

We address the issues above by leveraging the lightweight runtime profile provided by our execution engine and *stats cache*. Our query execution engine collects lightweight runtime stats after query execution and sends this information back to the optimizer along with query results. The profile includes the actual output row count with the filter condition, which helps the optimizer to know the actual table-level filter selectivity afterward. The actual filter selectivity will be used to evaluate the accuracy of our estimated filter selectivity stored in the *stats cache*. A background cache maintenance thread continuously monitors the accuracy of the estimated stats against the runtime selectivity. It updates the cache if the difference exceeds a pre-defined threshold.

4.2 Query Rewrite with Materialized Views

Besides pre-aggregated tables, Krypton also supports materialized views to perform pre-computation in serving workload. As mentioned in Section 3, the Ingestion Server can synchronously maintain user-defined materialized views during data ingestion time. During the query planning, the query optimizer automatically selects the appropriate materialized views for query rewriting for query acceleration. Our general query rewriting algorithm follows the algorithm proposed by Jonathan and Per-Åke [34]. Due to the space limit, we skip the details in this paper.

Many serving queries involve analyzing data that fall into time ranges with arbitrary boundaries. For a large time range, the to-be-processed data volume is also likely to be large. The following query represents a typical serving query for many of our use cases:

```
SELECT a, SUM(b) FROM tbl
WHERE t BETWEEN '2022-05-01 00:00:00'
  AND '2022-05-09 14:12:15'
GROUP BY a;
```

In this query, t is a column of timestamp type, and the time range of the filter condition spans more than eight days. Processing such a query on the fly can be time-consuming. Materialized view (MV) is an effective solution to speed up the large time range analysis and to significantly reduce the amount of data to process. For the example query above, we define multiple MVs that separately aggregate data by day and hour. Then the original query can be optimized by splitting the time range into three non-overlapping parts: 1) 2022-05-01 00:00:00 - 2022-05-09 00:00:00, 2) 2022-05-09 00:00:00 - 2022-05-09 14:00:00, and 3) 2022-05-09 14:00:00 - 2022-05-09 14:12:15. We can compute the results of each part separately, and merge the partial results to generate the final result. The first two parts can utilize the daily MV and hourly MV. The last part cannot make use of any MV, so it just uses data from the base table. The above process is performed automatically by our query optimizer at query planning time.

4.3 Adaptive Parallelism Control

Parallelism control in query execution has been studied extensively in the past decades [11, 18, 33, 51, 66, 82], which can be either static or dynamic in general. In static query parallelization, a given query plan can be divided into multiple sub-plans to exploit parallelism at the planning stage by utilizing either heuristic rules and/or statistical information at query compile time [4, 32, 44, 69], or statistical execution information collected at runtime [20, 43, 73]. On the other hand, dynamic query parallelization introduces techniques to parallelize given query work during execution. Leis et al. [48] have proposed morsel-driven query processing, which divides input data into small fragments (morsels) to achieve better load balancing among all the processor cores. The elastic iterator model, proposed by Wang et al. [81] for in-memory database clusters, has introduced a dynamic scheduler to improve processor utilization efficiency.

Both static and dynamic parallelism control mechanisms have their pros and cons. Generally speaking, parallelism assignment at compile time is hard to be optimal, as the workload in each compute node is affected by numerous factors and can be highly dynamic during the query execution. On the other hand, although determining the parallelism at runtime sounds more promising, it suffers from the overhead of locking/pausing the entire pipeline execution for changing the query plan structure, such as adding or updating operators.

In contrast, Krypton's query processor attempts to benefit from both approaches by statically deciding the inter-fragment and inter-pipe parallelisms while dynamically adjusting the intra-pipe parallelism using its coro-scheduler.

- At the planning time, the query optimizer statically determines the parallelism at both inter-fragment and inter-pipe levels (i.e., the number of Data Servers(DSs) a fragment utilizes and the number of threads used to execute a fragment). Deciding the parallelism at planning time avoids the overhead of modifying the plan on-the-fly and provides the opportunity to generate a more cost-efficient plan which eliminates costly operators such as local exchange and repartition whenever possible.
- At the execution time, the execution engine utilizes the coro-scheduler to exploit the *intra-pipe parallelism*, which dynamically parallelizes the computation of incoming blocks within each pipe. Specifically, the execution engine creates an execution task in a coroutine thread (*coro-thread*) and hands blocks over to the underlying coro-scheduler for execution whenever a block is pushed into the pipe. Because it dispatches execution tasks at the block granularity, the coro-scheduler reduces the working coro-threads for these in-flight tasks when queries with higher priority contest the CPU resources. Furthermore, because the coro-thread has a low overhead for the context switch (compared to the thread) and is asynchronous, scheduling the execution using coro-threads dramatically reduces the CPU idle time and improves query performance from both throughput and latency perspectives.

4.4 Resource Isolation and Fair Scheduling

Resource isolation in Krypton is required for serving different business requirements in a multi-tenant environment. For example, for paid clients, we must ensure their resources will not be contested by others. In addition, we need to ensure that the analytic queries will not block the latency-sensitive serving queries.

First, Krypton allows users to create dedicated resource groups at the DS instance level to achieve complete resource isolation. Each resource group is allocated to a set of dedicated DS instances at

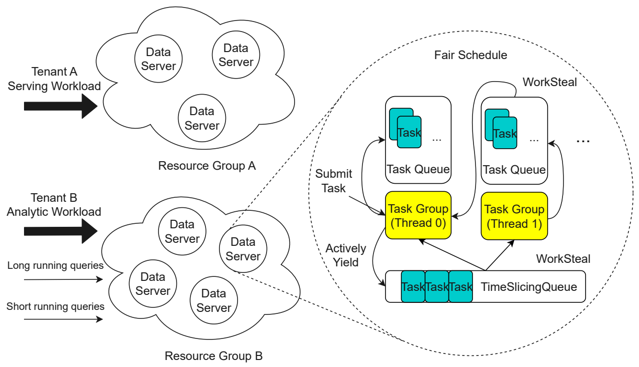

Figure 6: Resource Isolation and Fair Scheduling

execution time. Because Krypton uses a decoupled storage and compute architecture, the resource group can quickly scale out without data rebalancing in the underlying storage as traditional MPP databases. In addition, Krypton can quickly bring up a temporary resource group for short-term needs, e.g., processing nightly ETL queries and releasing the resources afterward.

Second, Krypton provides fair scheduling to solve resource contention within a resource group for the cases where a resource group is shared by multiple clients or by a single client with multiple applications through its coro-scheduler with a priority-based global time-slicing queue. As shown in Figure 6, each core/thread bonds to a different *task group*, which manages all the coro-threads assigned to it. During the execution, tasks of computing incoming blocks are submitted to its local task queue for execution. After a time slice *t*, a non-finished local task is pushed (actively yield) to the global time-slicing queue for future execution. When a local task queue is empty, the corresponding task group will fetch tasks from the global queue for execution, whose priority is based on the elapsed CPU time using some existing fairness-aware scheduling algorithms [42]. Because tasks are dispatched in a block-at-a-time fashion in Krypton, it enables us to minimize the scheduling overheads by placing the yield checkpoints at the time when the task is dispatched, as the execution time for each task is typically comparable with the time slice *t* (typically less than 10 ms).

In summary, Krypton can fully utilize the underlying asynchronous scheduler in a vectorized query engine to achieve both adaptive parallelism control and fairness schedule.

4.5 Lightweight API for Single Node Serving Queries

As illustrated in Figure 5, the query execution in Krypton generally requires two remote calls from the Coordinator, which first sends the fragments to a set of Data Servers (DSs) for execution, and then asynchronously fetches the cached query results from a dedicated DS. For complex queries with extensive query results, this send-and-fetch style execution is appropriate since the performance bottleneck usually comes from the computation instead of the network. However, for serving queries that only need a single DS for execution, the network overheads can be dominant, which prevents us from achieving desired query latencies.

To solve this problem, Krypton explicitly provides a lightweight API for simple serving queries. When the Coordinator detects that a single DS is enough for execution and the number of result rows is within a pre-defined threshold, it will use this API to execute the query and fetch the query results synchronously. In this way, these serving queries can be executed with a single remote call, and therefore the network overheads can be reduced by half.

4.6 Dirty Read for Sub-second Data Freshness

Some customers of Krypton require reading the most recent data changes that can be under sub-second delay. To meet such requirements, Krypton query engine can directly read uncommitted data (a.k.a. *dirty read*) from in-memory delta stores of Ingestion Servers (IgSs). Specifically, we introduce a dedicated dirty read operator in the query executor. The general workflow is as follows. First, the Coordinator obtains IgS addresses and the *committed_version* numbers corresponding to those tables from the Metadata Server and sends them to Data Servers as part of the distributed query plan so that the dirty read operator is constructed prior to the execution. The query executor will then use this operator to fetch uncommitted rows with versions larger than the committed_version through remote calls to the IgSs. These uncommitted rows will be merged with the committed/persisted ones before being returned to the clients. Delta stores normally hold recently flushed data for a small period of time to increase the chance of successful dirty reads.

It is worth noting that Data Servers do not cache data returned by the dirty read. Besides, multiple dirty read requests to the same IgS from a Data Server can consolidate into one request to reduce network latencies and overhead.

5 HIERARCHICAL CACHE WITH PMEM

Krypton leverages a caching component at Data Servers to overcome the performance penalty of decoupled computing and storage and to accelerate computations. This caching layer utilizes the strengths of different storage media to accommodate the diverse requirements of real-time serving and analytical queries cost-effectively. For real-time serving, Krypton recommends working sets fitting in DRAM and PMem.

Figure 7 illustrates the modular design of Krypton cache. A cache instance consists of three modules: cache index, replacement policy, and storage engine. The replacement policy module is pluggable and has several built-in policies. The storage engine module provides an abstract interface that hides the implementation details of different storage media from the index and replacement policy modules. Cache items stored in DRAM and PMem storage engines can be accessed directly through a zero-copy interface.

In the rest of this section, we describe some of the key techniques utilized in Krypton hierarchical cache to optimize its performance by making full use of hardware potentials.

5.1 Replacement Policy

HSAP workloads exhibit a wide variety of access patterns: analytical queries may require scanning substantial amounts of data, whereas serving queries tend to access a much smaller set of data with much higher frequency. As such, cache replacement policies in

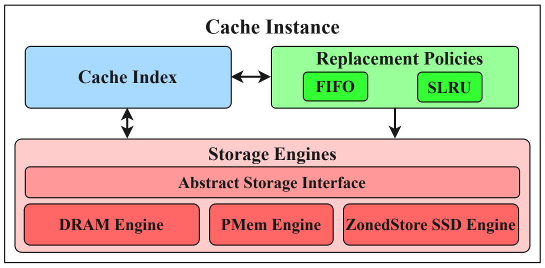

Figure 7: Modular Design of Krypton Cache

Table 1: Query Latencies under Different Replacement Policies

Latency (s)	FIFO		SLRU	
	Analytical	Serving	Analytical	Serving
P50	28.088	4.792	28.108	4.141
P99	36.437	7.274	33.523	5.679

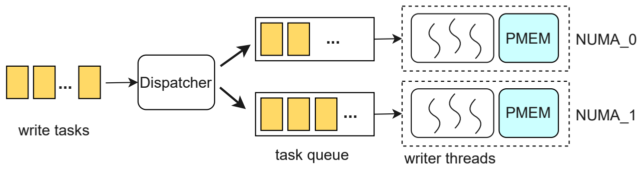

Figure 8: NUMA-Aware Async PMem Write

HSAP systems should be *scan resistant* to ensure a high hit ratio for serving queries.

We choose Segment LRU (SLRU) [59] as the replacement policy for the DRAM and PMem tiers. Besides being scan resistant, this policy also has low concurrency overheads because fetching items already in the cache never requires waiting for an LRU lock, which makes it a good candidate for the DRAM and PMem tiers. Unlike the SLRU used in [59], we use a lock-free concurrent hash table as the cache index to further reduce concurrency overhead.

We evaluate Krypton hierarchical cache replacement policy using a workload of mixed serving and analytical queries. We use TPC-H Q6 as the serving query and Q21 as the analytical query, with dedicated workers for each. The experiment setup is described in Section 7. We warm up the cache by running Q6 alone before starting Q21. As shown in Table 1, the median and P99 latencies of the serving query (Q6) under Krypton cache replacement policy are ~15% and ~28% lower, respectively, than those under a simple first-in-first-out (FIFO) cache replacement policy.

5.2 NUMA-Aware Async PMem Write

Despite its read latency and throughput advantages, the asymmetric write bandwidth of PMem is a performance bottleneck [83, 86]. Specifically, PMem write bandwidth is only about one-sixth of DRAM write bandwidth [41], saturates at a concurrent access level lower than read bandwidth, and deteriorates when accessed from remote non-uniform memory access (NUMA) node [68].

Krypton uses a NUMA-aware async-write mechanism to optimize PMem write performance. As illustrated in Figure 8, each PMem device is assigned a dedicated writer thread pool, bound to the same NUMA node, and solely responsible for all writes to the PMem device. Other threads read directly from the PMem device but write only indirectly by dispatching asynchronous write tasks to the writer thread pool. The number of threads in each pool is set through experiments to yield maximum write throughput.

We evaluate the NUMA-aware async-write mechanism on a machine with two NUMA nodes, each with four 128GB PMem DIMMs. With three writer threads per pool, Krypton PMem cache achieves a maximum throughput of 3.7GB/s with this mechanism, which is about 23% higher than the 3.0GB/s throughput without it.

Table 2: Mixed Random Read/Write Workload

R:W Ratio		Avg Latency (us)		Throughput (MB/s)	
		ZonedStore	RocksDB	ZonedStore	RocksDB
20:80	Read	619	1,412	153	20
	Write	10,880	27,061	1,515	184
80:20	Read	470	891	1,640	365
	Write	632	30,261	485	97

5.3 ZonedStore Based SSD Cache

SSD brings Krypton affordable, large cache capacity, and fast warm-up after the system restarts, but it also raises new challenges. In particular, most existing cache systems at ByteDance manage content on SSD using Log-Structured Merge-Tree (LSM Tree) key-value stores (e.g., RocksDB [28]). These LSM tree-based storage solutions are not specifically designed for an SSD caching system and suffer from excessive space and read/write amplifications when used as one. To address these shortcomings, we design and implement ZonedStore, a low-cost and high-performance SSD storage.

Zone-based Sequential Write. ZonedStore partitions each SSD into multiple *zones* of equal size (e.g., 10GB). Only one zone is writable at a time. Newly inserted cache items always append to the current writable zone sequentially, which reduces SSD internal write amplification [61]. Since in ZonedStore, most items have a size larger than 4KB, it is feasible to keep an in-memory index of all items cached in ZonedStore, which eliminates read amplification during lookup. In order to speed up index recovery upon restart, a summary segment describing all cache items in a zone is written at the end of it before the writes switch to a new writable zone.

Cache Optimized Garbage Collection (GC). ZonedStore reclaims storage space at zone granularity. The garbage ratio and access frequency of each zone are tracked in an in-memory zone metadata table, and the GC policy always reclaims zones with a high garbage ratio and a low access frequency. Unlike RocksDB, ZonedStore does not write tombstones to SSD. Instead, evicted items are marked as *soft-deleted* in the in-memory index. Soft-deleted but not yet reclaimed items can still be used to serve lookups and inserts, exploiting the fact that all data in Krypton hierarchical cache are immutable. Furthermore, ZonedStore limits GC-induced write amplification by simply discarding valid data in the reclaimed zone when the write bandwidth is limited.

We evaluate ZonedStore on an Intel P4510 SSD using mixed read-write workloads issued by 16 concurrent threads with a 50KB item size. As shown in Table 2, ZonedStore significantly outperforms RocksDB in both scenarios.

6 EFFICIENT COLUMNAR DATA FORMAT

Krypton implements its own file format to efficiently support serving and analytical workloads. This format is a typical Partition Attributes Across (PAX) format [3] that organizes user data in different regions by columns and encodes and compresses at the granularity of the data page (usually 1MB). A data page is the unit of data reading. In regular OLAP query processing, Krypton first accesses indexes before the actual data pages to filter out the ones that do not need to be read. Krypton format divides the whole file into user data region, index region, and metadata footer, where data and indexes are completely separated and placed in different regions of the file. To be cache friendly, each region is partitioned by columns to maximizes the access locality.

Encoding and Index Algorithms. The Krypton format uses lightweight encoding and rich indexes to achieve efficient scans and seeks while reducing storage costs. The Krypton format supports various encodings, such as Run-Length-Encoding and Frame-of-Reference encoding compression [49]. Krypton format implements vectorized reads and writes for all encoding methods to provide efficient scanning and combines various indexes to provide $O(1)$ seek.

In order to support hybrid workloads, Krypton utilizes a set of index methods to speed up scans and seeks. To quickly locate the requested data's physical position in serving scenarios, users can choose suitable indexes on particular columns in the DDL statement. A brief description of each index method and its intended usage scenario is listed as follows:

- Ordinal index: to find the data page with the target ordinal number.
- Sparse index, min/max index, bloom filter, and ribbon filter [27]: to skip data pages that do not need reading.
- Short-key index: a particular sparse index that takes the first 36 bytes of the sorted key column as the index key.
- Equality and range bitmap index: to quickly filter for the row ordinal according to the predicate.
- Skip index: to quickly locate the physical position within the page. Under some conditions, using a skip index can avoid decoding the entire data page.

Nested Type Handling. The Krypton format proposes an efficient algorithm to represent nested and repeated types, which differs from the popular Google Dremel algorithm [58] implemented in Parquet [6]. Unlike the layout in Google Dremel which only stores leaf fields, our approach organizes a type's schema as a B-tree of fields and the data of all fields are placed continuously and independently. The format saves the *occurrence* and *validity* information of child fields in non-leaf fields and stores the data in leaf fields. *Occurrence* represents the prefix sum of the subfield occurrences, resulting in $O(1)$ complexity for getting the offset and length of the repeated data. Hence we can achieve $O(m)$ seek even on nested and repeated data, where m is the depth of the schema tree. In contrast, Google Dremel must iterate multiple repetition levels to determine the offsets, slowing down seeking operations. *Validity* distinguishes whether the field is empty or null. By not saving any data for a null field, Krypton format gains high efficiency for storing sparse data. Comparing to Google Dremel, our algorithm has the following advantages: 1) Higher storage efficiency for sparse fields, and 2) faster seeking for nested and repeated types.

Table 3: File Format Benchmark on TPC-H and Magnus

Dataset	Read Speed (rows / sec)		Relative File Size	
	Parquet	Krypton	Parquet	Krypton
TPC-H	2.54M	3.07M	100.00	113.68
Magnus	0.15M	0.21M	100.00	91.99

Query Engine Integration. The Krypton format is deeply integrated with the query engine, as *late materialization* and *push-down computation* are used as much as possible to reduce the I/O overhead during the query process. Predicate filtering and column pruning are pushed down to the format layer, along with push-down runtime filter predicates and file indexes. During the read process, it first reads and uses the file index referenced by the push-down predicate and the runtime filter predicate to init the ordinal selection vector, which stores all selected ordinal numbers. Second, we materialize the predicate column data for expression computation to narrow the range of the ordinal selection vector, then perform column pruning according to the push-down projection. Finally, we materialize non-predicate column data according to a selection vector. In addition to providing APIs to read raw user data, Krypton format also provides APIs to read encoded data that allows Query Engine to directly calculate on encoded data (e.g., dictionary-encoded data), thereby reducing decoding overhead.

We evaluate the performance of single-threaded reads on the TPC-H dataset (a typical scalar type dataset) and the Magnus dataset (a nested and repeated ML training dataset in production). The testbed is equipped with Intel Xeon Platinum 8260 CPU (2.40GHz, 48 Cores, 96 vCPUs), 128GB DRAM, 512GB PMem, and 2TB NVMe SSD. Table 3 shows the results. Compared to the Parquet file, the read performance of Krypton file improves by about 21% on the TPC-H dataset and about 40% on the Magnus dataset. Compared to the Parquet file, the size of Krypton file increases by about 13% on the TPC-H dataset. These extra file spaces store additional indexes not provided in the Parquet file to provide efficient seeking operations. Note that on the Magnus dataset, the size of Krypton file decreases by about 8%, because the benefits Krypton format gained on compressing data of nested and repeated types outweigh the overhead of keeping additional indexes.

7 PERFORMANCE STUDY

In this section, we conduct experiments to evaluate the performance of Krypton. We present our experiment results using benchmark workloads followed by results using ByteDance real production workloads.

7.1 Workloads and Experiment Setup

Since we cannot find an HSAP benchmark ready to use, we choose YCSB [24] Workload C [21] to simulate serving workloads and use TPC-H [80] benchmark with a 1TB dataset to simulate analytical workloads. All YCSB and TPC-H workload experiments are conducted with the *benchbase* benchmarking framework [25, 26].

Table 4: Serving Query Throughputs and Latencies (YCSB 100GB)

	Throughput (QPS)	Latency (ms)				
		Avg.	P75	P90	P95	P99
Separate	62272.22	3.52	3.63	4.61	5.98	10.49
Resource Group	62183.40	3.55	3.63	4.65	6.08	10.52

Regarding the production workload, we choose ByteDance's *ZhuXiaoBang* workloads. *ZhuXiaoBang* represents a common feature engineering use case that requires performing aggregate calculations on specific user features within a time window ranging from minutes to months. These features are continuously ingested and queried in real time with high QPS and low query latency.

All our experiments are conducted on a cluster of eight machines equipped with Intel Xeon Platinum 8260 CPUs (2.40GHz, 48 Cores, 96 vCPUs), 128GB DRAM, 512GB PMem, and 2TB NVMe SSD. The cluster is interconnected with 25Gbps Ethernet NICs. Unless otherwise stated, all our experiments are conducted with the following setup: two machines for Coordinators and three machines for Data Servers. In addition, one machine is assigned for each of the following roles: Compaction Server, Ingestion Server, and Metadata Server. All experiments start with warm-up runs where the result cache is disabled, and the reported values of each experiment are the averages of three repeated runs.

7.2 Hybrid Performance

In this set of experiments, we choose a hybrid workload comprising YCSB and TPC-H queries to evaluate the effectiveness of Krypton's two-level resource isolation. Specifically, we create two dedicated resource groups: a serving resource group for executing YCSB and an analytical resource group for executing TPC-H 1T workloads separately. We allocate one-third of DSs to the serving resource group and the remaining DSs to the analytical resource group.

Resource Group Isolation. We first run TPC-H 1T and YCSB workloads in their corresponding resource group to study the performance of Krypton on analytical and serving workloads, respectively. Then we run a mixed workload of TPC-H 1T and YCSB to demonstrate the effectiveness of resource group isolation. As illustrated in Figure 9 and Table 4, Krypton delivers a competitive performance for both analytical and serving workloads. Compared with separately running TPC-H and YCSB workloads, the hybrid running with resource group isolation does not incur noticeable performance degradation for either analytical or serving workloads.

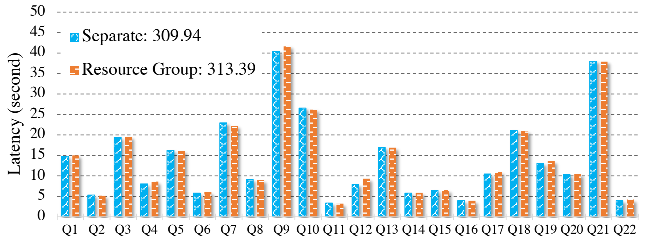

Figure 9: Analytical Query Latencies (TPC-H 1TB)

Fair Scheduling. Fair scheduling (FS) is designed to solve resource contention within a resource group. Therefore, in this experiment, we construct a workload of TPC-H Q6 and Q21, which are representatives of short- and long-running queries running in the analytical resource group. Both queries start with one client, and the number of clients for Q6 increases from 1 to 2, 4, and 8, gradually. Figure 10 shows that, without fair scheduling, the increased resource contention from Q6 causes a significant performance degradation in Q21.

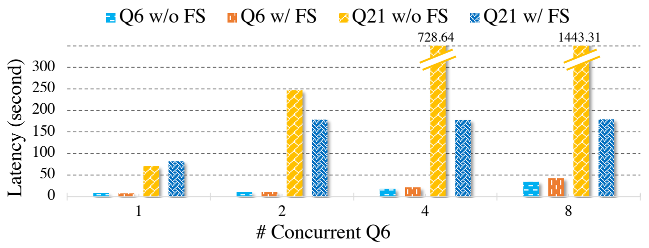

Figure 10: Effectiveness of Fair Scheduling

In contrast, fair scheduling can specify the portion of resources (e.g., CPU, memory) allocated to a workload. It, therefore, prevents the performance of one workload from being dramatically affected by other concurrent workloads. As shown in Figure 10, Q21's latency is inversely proportional to the number of clients without fair scheduling. When the CPU resources (in percentage) for running Q21 and Q6 are configured as 20% and 80%, respectively, the latency of Q21 only slightly increases as the number of Q6 clients increases. In the meantime, we also observe only slight performance degradation in Q6.

In addition, we collected 10 CPU time samples for each experiment with fair scheduling enabled. As shown in Figure 11, when running a single Q6 query, which is not able to fully utilize the configured resources (i.e., 80%), it only consumes approximately 53% of the total resources, and the adaptivity of the fair scheduling mechanism allows Q21 to use an extra 27% CPU time. As the number of Q6 clients increases, both Q6 and Q21 can use up the allocated resources, and therefore the accumulated CPU time is stably shared between them as the configured percentages.

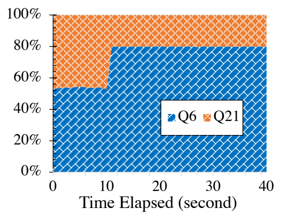

Figure 11: Fairness of Fair Scheduling

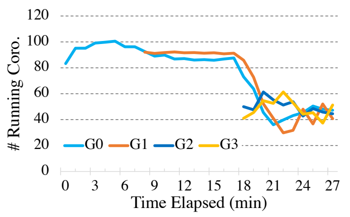

Figure 12: Adaptive Parallelism Control

Adaptive Parallelism Control. To demonstrate the effectiveness of adaptive parallelism control, we use four clients (G_0 - G_3), each of which repetitively runs Q6 with the maximum degree of intra-fragment parallelism of 48 in the analytical resource group. As seen in Figure 12, with only G_0, these queries can run with their

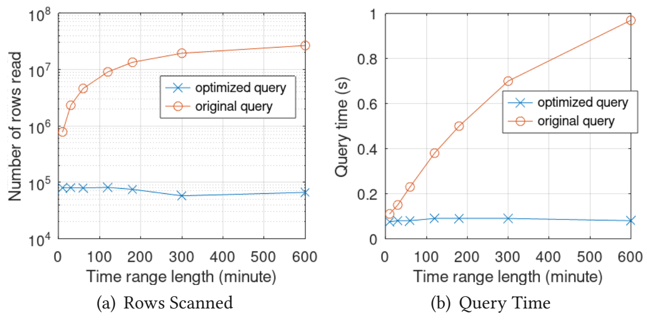

Figure 13: Effects of Optimizing Time Range Queries

maximum degree of intra-fragment parallelism due to sufficient CPU resources. As we start more clients G_1 - G_3, hence the CPU resource becomes more contested, the number of running corothreads starts to change adaptively for each client as expected.

7.3 Production Performance

We show query performance results and production metrics from the *ZhuXiaoBang* production environment.

Effects of Optimizing Time Range Queries. To evaluate the effects of optimizing time range queries with MVs, we use the following queries from *ZhuXiaoBang* workload. We use a fixed end time for each time range and vary the time range by changing the starting time. We use MVs to aggregate data for each minute, hour, and day. In our experiments, the time range varies from 10 minutes to 10 hours.

```
SELECT feature_index, SUM(feature_value)
FROM feature_table
WHERE event_time BETWEEN $start_time
  AND '2023-02-16 11:00:30'
GROUP BY feature_index;
```

Figure 13(a) shows the number of rows read by queries with and without the optimization. We observe that the data volume without temporal optimization increases significantly as the time range increases. With the optimization, however, the data volume is almost the same. Similarly, the query time remain stable for optimized queries as the time range increases, as shown in Figure 13(b). For queries with time ranges that span a few days, the impact of the optimization is even more significant.

Effects of Lightweight API. We compare the query latency with around 10,000 QPS to evaluate the effects of Lightweight API. Figure 14 shows the P99 latency is reduced by about 45% after Lightweight API is enabled.

Data Freshness of Streaming Ingestion. Data freshness in KRYPTON is defined as the time interval between the new rows that can be queried after corresponding WALs are committed successfully. Figure 15 shows the P99 latency of streaming load data freshness for *ZhuXiaoBang*. As the data input rate increases, the data freshness becomes worse, but remains pretty low (i.e. around 15 ms).

Read/Write Scenario in Production. *ZhuXiaoBang* is a typical read-write hybrid business scenario. Real-time data is imported into KRYPTON through the Flink streaming job. On a normal day, the ingestion rate (rows/sec) and query QPS started to grow after 18:00, and the peak period of business is around 22:00. The peak ingestion rate is about 460% of the daily average, and the peak query TPS is about 300% of the daily average. Since KRYPTON adopts the read-write separation architecture, query performance is not significantly affected even at the peak time of data inputs. As shown in Figure 16, the p99 query latency for *ZhuXiaoBang* during rush hours is relatively stable and remains under 60 ms.

8 PRODUCTION EXPERIENCES AND LESSONS LEARNED

This section briefly summarizes some lessons we have learned from developing and running KRYPTON in production since early 2022. As KRYPTON serves as a general serving and analytical SQL engine, we have been working with various teams in the company, such as the Machine Learning platform, Ads, and Shopping. Those KRYPTON users provided us with a lot of valuable feedback, which significantly improved the quality and expedited the iterations of the system. Below is a representative but by no means an exhaustive list.

Interface Compatibility with Legacy Systems. Prior to KRYPTON, multiple customers at ByteDance used Apache Doris [29] and its related tools. From the beginning of the project, we set the goals for KRYPTON to be compatible with Apache Doris (MySQL), including the SQL dialect, data model, data ingestion interface, and clients. As a result, the existing Doris users can readily migrate to KRYPTON with minimal efforts. Based on our production experience, most users can smoothly switch their underlying engines to KRYPTON without interrupting their services.

Always Explore New Opportunities to Improve Performance. There is no silver bullet for HSAP systems to provide much higher QPS and lower latency than traditional OLAP systems. Instead, we constantly look for ways to help customers to improve their performance by closely studying their needs. For example, an online service may send hundreds of millions of queries to KRYPTON per day, but only tens of thousands are distinct except for filtering conditions. In this case, both result and plan caches are greatly helpful in reducing resource usage in the back end. Some other key techniques we used include optimizing the size of WAL and asynchronous writes, which proved high effectiveness in ensuring the stability of critical write paths while ingesting a large amount of data.

Live Traffic Testing is Critical. KRYPTON is a large, complex, and newly developed system. When we promote KRYPTON to customers, they usually do not have enough trust in its features and stability in the beginning. Even though standard testing benchmarks are used in development, they can not cover the real use cases in production. To address this challenge, we adopt the approach of replicating real online traffic to the KRYPTON testing clusters. The query results returned from KRYPTON are logged asynchronously and compared with those of the legacy system offline without jeopardizing the critical path. With this approach, we could discover and fix various issues, effectively improving the system's overall quality. In summary, live testing makes the process for our customers to migrate to KRYPTON much smoother.

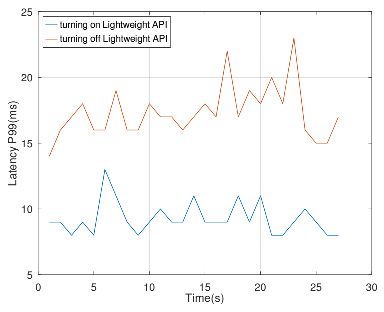

Figure 14: Query Latency for Lightweight API Turning On vs. Turning Off

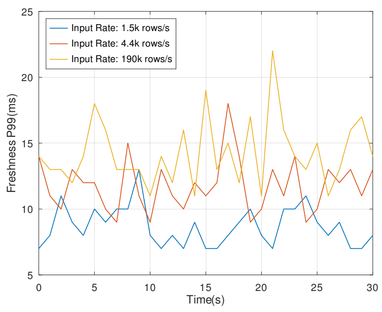

Figure 15: Data Freshness of Different Ingestion Rate for *ZhuXiaoBang*

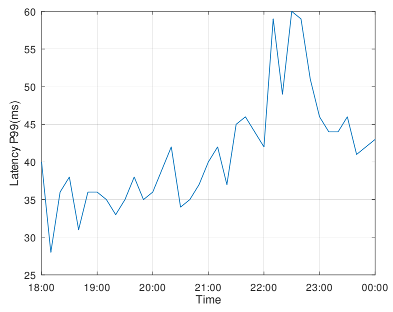

Figure 16: Query Latency During Rush Hours for *ZhuXiaoBang*

Krypton is still in early adoption at ByteDance, and we are glad to share more experiences and lessons learned in the future.

9 RELATED WORK

Procella [15] is an HSAP system most similar to Krypton. They share many essential design principles, such as disaggregated storage, separate read and write paths. Both systems heavily utilize cache and pre-computation to accelerate serving query performance. Krypton uses a hierarchical local cache that includes DRAM, PMem, and SSD, while Procella only caches data in DRAM. In addition, they are also different in terms of dirty read handling: Procella pushes data from Ingestion Servers to Data Servers, while Krypton queries Ingestion Servers as external tables for dirty reads.

Hologres [42] is an HSAP system from Alibaba. Hologres also adopts disaggregated storage and separate read and write paths architecture. However, in Hologres, read and write queries run on the same location where a Table Group Shard resides, while Krypton and Procella's read and write queries run on different nodes. Hologres utilizes a hierarchical cache but does not utilize PMem as Krypton.

Data warehouse and OLAP systems [13, 53, 54, 58, 75, 87] perform complex data analytics over relatively static data, which cannot handle serving queries with millions of QPS and data ingestion rates required by HSAP workloads. Snowflake [23] is a cloud data warehouse that utilizes the cache to improve query performance in a decoupled computing and storage architecture. Krypton also keeps a hierarchical cache at its computing layer to speed up query processing, which is even more critical for serving workload than analytical workload. In addition, some previous work [5, 50, 62, 84, 87] to support real-time analytic queries also shed light on designing efficient OLAP engines. Krypton adopts many previously prevailing query processing techniques to build a high-performant HSAP system.

Druid [85] and Pinot [39] are two open-source systems used for low-latency data ingestion and real-time OLAP analytics on large datasets. However, their capabilities to support general SQL queries are more limited than Krypton.

Time-series databases [1, 14, 67, 72, 77] are widely used in IoT and compute system metrics analysis over a large number of ordered data logs. Similar to Krypton, they also focus on the data ingestion throughput, data freshness, and query latency. However, they usually have much more limited SQL capabilities than Krypton.

Kraken [35] is a real-time monitoring and analytics system providing strong data consistency by a deterministic partitioning scheme with a single truth source. Similar to Krypton, Kraken also supports different types of data stream and ingestion. However, unlike Krypton, Kraken does not allow queries on the data until it is persisted to the durable storage.

Napa [2] maintains materialized views at data ingestion time to speed up querying. Krypton adopts a similar approach to maintain aggregated tables and materialized views at data injection time.

The emergence of new storage hardware such as Intel Optane Persistent Memory has motivated the evolution of caching systems [16, 22, 45, 55, 76, 79]. CacheLib [12] is a popular open-source C++ library for accessing and managing cache data. We note that Krypton hierarchical cache shares several important design choices as CacheLib, such as zero-copy interface, full in-memory index of items cached on SSD, and region/zone-based sequential SSD writes. We plan to compare the performance of the two in future work.

10 CONCLUSIONS

In this paper, we demonstrate how to build Krypton, a single Hybrid Serving and Analytical Processing (HSAP) system, to handle both types of workloads. Krypton adopts disaggregated storage, separate data injection and query processing, a hierarchical local cache at query processing nodes, and a native columnar storage format to provide excellent elasticity and query performance. Krypton can provide high data freshness, high data ingestion rate, and strong data consistency support by utilizing many previously-known query processing techniques. Lastly, we briefly overview the lessons and best practices we have learned in developing and running Krypton in the production environment.

ACKNOWLEDGMENTS

We would like to extend our thanks to the anonymous reviewers for their valuable comments. We heartily thank all people who made contributions to the design and development of Krypton system: Wei Chen, David Gong, Liming Fang, Jianfeng Qian, Zixiong Liu, Ron Hu, Dongwei Wang, Jianhua Cao, Jin Li, Kai Wu, Hao Tan, Guowei Zhang, Qianling Li, Yize Li, Yang Liu, Hongkai Wei, Yue Gong.

REFERENCES

[1] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George Talbot, Adam Tart, and Nick Taylor. 2020. Monarch: Google's Planet-Scale in-Memory Time Series Database. *Proc. VLDB Endow.* 13, 12 (sep 2020), 3181–3194. https://doi.org/10.14778/3181-3194

[2] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Jim Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh S R, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Junichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divyakanth Agrawal, Jeff Naughton, Sujata Sunil Kosalge, , and Hakan Hacıgümüş. 2021. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *Proc. VLDB Endow.* 12 (2021), 2986–2998.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 169–180.

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proc. VLDB Endow.* 5, 10 (jun 2012), 1064–1075. https://doi.org/10.14778/2336664.2336678

[5] Rana Alotaibi, Bogdan Cautis, Alin Deutsch, and Ioana Manolescu. 2021. HADAD: A lightweight approach for optimizing hybrid complex analytics queries. In *Proceedings of the 2021 International Conference on Management of Data.* 23–35.

[6] Apache. 2022 (Accessed on 2023-03-02). Apache Parquet. https://parquet.apache.org/.

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

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

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

[10] Ronald Barber, Vijayshankar Raman, Richard Sidle, Yuanyuan Tian, and Pinar Tözün. 2019. Wildfire: HTAP for big data. In *Encyclopedia of Big Data Technologies*. Springer.

[11] Srikanth Bellamkonda, Hua-Gang Li, Unmesh Jagtap, Yali Zhu, Vince Liang, and Thierry Cruanes. 2013. Adaptive and Big Data Scale Parallel Execution in Oracle. *Proc. VLDB Endow.* 6, 11 (aug 2013), 1102–1113. https://doi.org/10.14778/2536222.2536235

[12] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

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

[14] Wei Cao, Yusong Gao, Feifei Li, Sheng Wang, Bingchen Lin, Ke Xu, Xiaojie Feng, Yucong Wang, Zhenjun Liu, and Gejin Zhang. 2020. Timon: A Timestamped Event Database for Efficient Telemetry Data Processing and Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 739–753. https://doi.org/10.1145/3318464.3386136 event-place: Portland, OR, USA.

[15] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *PVLDB* 12(12) (2019), 2022–2034. https://dl.acm.org/citation.cfm?id=3360438

[16] Cheng Chen, Jun Yang, Mian Lu, Taize Wang, Zhao Zheng, Yuqiang Chen, Wenyuan Dai, Bingsheng He, Weng-Fai Wong, Guoan Wu, Yuping Zhao, and Andy Rudoff. 2021. Optimizing In-Memory Database Engine for AI-Powered on-Line Decision Augmentation Using Persistent Memory. *Proc. VLDB Endow.* 14, 5 (mar 2021), 799–812. https://doi.org/10.14778/3446095.3446102

[17] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. 2022. ByteHTAP: Bytedance's HTAP System with High Data Freshness and Strong Data Consistency. *Proc. VLDB Endow.* 15, 12 (sep 2022), 3411–3424. https://doi.org/10.14778/3554821.3554832

[18] Ming-Syan Chen, Ming-Ling Lo, Philip S. Yu, and Honesty C. Young. 1992. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 15–26.

[19] ClickHouse. 2023. ClickHouse: Fast Open-Source OLAP DBMS. https://clickhouse.com/. [Online; accessed 24-January-2023].

[20] Richard L. Cole and Goetz Graefe. 1994. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 150–160. https://doi.org/10.1145/191839.191872

[21] Brian Cooper. 2023. YCSB/workloadc. https://github.com/brianfrankcooper/YCSB/blob/master/workloads/workloadc/. [Online; accessed 13-FEbuary-2023].

[22] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 339–351. https://doi.org/10.1145/3448016.3457292

[23] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[24] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.

[25] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[26] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2023 (Accessed on 2023-02-28). cmu-db/benchbase: Multi-DBMS SQL Benchmarking Framework via JDBC. https://github.com/cmu-db/benchbase.

[27] Peter C Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *arXiv preprint arXiv:2103.02515* (2021).

[28] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (oct 2021), 32 pages. https://doi.org/10.1145/3483840

[29] Doris. 2023. Apache Doris. https://doris.apache.org/. [Online; accessed 24-January-2023].

[30] ElasticSearch. 2023. What is Elasticsearch? https://www.elastic.co/what-is/elasticsearch. [Online; accessed 24-January-2023].

[31] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 137–156.

[32] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. 1992. Query Optimization for Parallel Execution. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD '92)*. Association for Computing Machinery, New York, NY, USA, 9–18. https://doi.org/10.1145/130283.130291

[33] Kazuo Goda, Yuto Hayamizu, Hiroyuki Yamada, and Masaru Kitsuregawa. 2020. Out-of-Order Execution of Database Queries. *Proc. VLDB Endow.* 13, 12 (sep 2020), 3489–3501. https://doi.org/10.14778/3415478.3415571

[34] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. *ACM SIGMOD Record* 30, 2 (2001), 331–342.

[35] Stavros Harizopoulos, Taylor Hopper, Morton Mo, Shyam Sundar Chandrasekaran, Tongguang Chen, Yan Cui, Nandini Ganesh, Gary Helmling, Hieu Pham, and Sebastian Wong. 2022. Meta's next-Generation Realtime Monitoring and Analytics Platform. *Proc. VLDB Endow.* 15, 12 (sep 2022), 3522–3534. https://doi.org/10.14778/3554821.3554841

[36] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 683–692. https://doi.org/10.1145/2452376.2452456

[37] Hive. 2023. Apache Hive. https://hive.apache.org/. [Online; accessed 24-January-2023].

[38] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (sep 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[39] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, and Ravi Aringunram. 2018. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3183713.3190661 event-place: Houston, TX, USA.

[40] ByteDance Inc. 2023. Zhuxiaobang App. https://www.zhuxiaobang.com/.

[41] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[42] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, Jihong Ma, Li Su, and Kai Zeng. 2020. Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing. *Proc. VLDB Endow.* 13, 12 (sep 2020), 3272–3284. https://doi.org/10.14778/3415478.3415550

[43] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/276304.276315

[44] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[45] Dimitrios Koutsoukos, Raghav Bhartia, Ana Klimovic, and Gustavo Alonso. 2021. How to use Persistent Memory in your Database. https://doi.org/10.48550/ARXIV.2112.00425

[46] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1740–1751. https://doi.org/10.14778/2824032.2824071

[47] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. 2017. Parallel Replication across Formats in SAP HANA for Scaling out Mixed OLTP/OLAP Workloads. *Proc. VLDB Endow.* 10, 12 (aug 2017), 1598–1609. https://doi.org/10.14778/3137765.3137767

[48] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. 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* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[49] Daniel Lemire. 2023. Effective compression using frame-of-reference and delta coding. https://lemire.me/blog/2012/02/08/effective-compression-using-frame-of-reference-and-delta-coding/. [Online; accessed 02-March-2023].

[50] Feng Li, M. Tamer Özsu, Gang Chen, and Beng Chin Ooi. 2014. R-Store: A scalable distributed system for supporting real-time analytics. In *2014 IEEE 30th International Conference on Data Engineering*. 40–51. https://doi.org/10.1109/ICDE.2014.6816638

[51] Bin Liu and Elke A. Rundensteiner. 2005. Revisiting Pipelined Parallelism in Multi-Join Query Processing. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 829–840.

[52] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified multi-zone indexing for large-scale HTAP. In *Advances in Database Technology-22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. OpenProceedings. org, 1–12.

[53] Zhenxiao Luo, Lu Niu, Venki Korukanti, Yutian Sun, Masha Basmanova, Yi He, Beinan Wang, Devesh Agrawal, Hao Luo, Chunxu Tang, Ashish Singh, Yao Li, Peng Du, Girish Baliga, and Maosong Fu. 2022. From Batch Processing to Real Time Analytics: Running Presto® at Scale. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1598–1609. https://doi.org/10.1109/ICDE53745.2022.00165

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

[55] Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, and King Lv. 2022. HiEngine: How to Architect a Cloud-Native Memory-Optimized Database Engine. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2177–2190. https://doi.org/10.1145/3514221.3526043

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

[57] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. Sap hana–the evolution of an in-memory dbms from pure olap processing towards mixed workloads. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)* (2017).

[58] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (sep 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[59] memcached. 2018. memcached - a distributed memory object caching system. https://memcached.org/blog/modern-lru/.

[60] Elena Milkai, Yannis Chronis, Kevin P Gaffney, Zhihan Guo, Jignesh M Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *Proceedings of the 2022 International Conference on Management of Data*. 1810–1824.

[61] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random Write Considered Harmful in Solid State Drives. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast12/sfs-random-write-considered-harmful-solid-state-drives

[62] Niloy Mukherjee, Shasank Chavan, Maria Colgan, Mike Gleeson, Xiaoming He, Allison Holloway, Jesse Kamp, Kartik Kulkarni, Tirthankar Lahiri, Juan Loaiza, Neil Macnaughton, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, and Raunak Rungta. 2016. Fault-tolerant real-time analytics with distributed Oracle Database In-memory. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1298–1309. https://doi.org/10.1109/ICDE.2016.7498333

[63] MySQL. 2023. MySQL. https://www.mysql.com/. [Online; accessed 24-January-2023].

[64] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (jun 2011), 539–550. https://doi.org/10.14778/2002938.2002940

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

[66] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. 2001. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proceedings of the 17th International Conference on Data Engineering*. IEEE Computer Society, USA, 567–574.

[67] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[68] Ivy B Peng, Maya B Gokhale, and Eric W Green. 2019. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*. 304–315.

[69] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-Aware Data Placement and Task Scheduling for Analytical Workloads in Main-Memory Column-Stores. *Proc. VLDB Endow.* 10, 2 (oct 2016), 37–48. https://doi.org/10.14778/3015274.3015275

[70] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. 2015. Scaling up mixed workloads: a battle of data freshness, flexibility, and scheduling. In *Performance Characterization and Benchmarking. Traditional to Big Data: 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1–5, 2014. Revised Selected Papers 6*. Springer, 97–112.

[71] Redis. 2023. Redis. https://redis.io//. [Online; accessed 24-January-2023].

[72] Sean Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. 2017. LittleTable: A Time-Series Database and Its Uses. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 125–138. https://doi.org/10.1145/3035918.3056102 event-place: Chicago, Illinois, USA.

[73] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro Adaptivity in Vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1231–1242. https://doi.org/10.1145/2463676.2465292

[74] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do,

Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. 2018. F1 Query: Declarative Querying at Scale. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1835–1848. https://doi.org/10.14778/3229863.3229871

[75] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[76] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. 2020. Large-Scale in-Memory Analytics on Intel(®) Optane(™) DC Persistent Memory. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3399666.3399933

[77] Xuanhua Shi, Zezhao Feng, Kaixi Li, Yongluan Zhou, Hai Jin, Yan Jiang, Bingsheng He, Zhijun Ling, and Xin Li. 2020. ByteSeries: An in-Memory Time Series Database for Large-Scale Monitoring Systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 60–73. https://doi.org/10.1145/3419111.3421289

[78] Spark. 2023. Apache Spark. https://spark.apache.org/. [Online; accessed 24-January-2023].

[79] Jason Sun, Haoxiang Ma, Li Zhang, Huicong Liu, Haiyang Shi, Shangyu Luo, Kai Wu, Kevin Bruhwiler, Cheng Zhu, Yuanyuan Nie, Jianjun Chen, Lei Zhang, and Yuming Liang. 2023. Accelerating Cloud-Native Databases with Distributed PMem Stores. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 3043–3057. https://doi.org/10.1109/ICDE55515.2023.00233

[80] TPC. 2023. TPC-H Homepage. https://www.tpc.org/tpch/. [Online; accessed 24-January-2023].

[81] Li Wang, Minqi Zhou, Zhenjie Zhang, Yin Yang, Aoying Zhou, and Dina Bitton. 2016. Elastic Pipelining in an In-Memory Database Cluster. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1279–1294. https://doi.org/10.1145/2882903.2882904

[82] Yun Wang. 1995. DB2 Query Parallelism: Staging and Implementation. In *Proceedings of the 21th International Conference on Very Large Data Bases (VLDB '95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 686–691.

[83] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508.

[84] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[85] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A Real-Time Analytical Data Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/2588555.2595631 event-place: Snowbird, Utah, USA.

[86] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory.. In *FAST*, Vol. 20. 169–182.

[87] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2059–2070.