Separation Is for Better Reunion: Data Lake Storage at Huawei

Xin Tang\*, Dawei Zhao\*, Haohai Ma, Jiaquan Zhang, Yong Zheng, Zhenyong Fan, Yue Zhao, Yi He, Keji Huang†,

Zhongsheng Xu, Yidong Wang, Tao Tao‡, Zhuo Chen‡, Guoliang Li§, Jiwu Shu§, Chengliang Chai§

streamlake@huawei.com  
 Huawei Technologies Co., Ltd. China Mobile Communications Group Co., Ltd.‡ Tsinghua University§

ABSTRACT

Huawei works with some of the Chinese largest business companies to store and process exabytes of nationwide operational data in data lake storage to provide business insights. Nation-scale streams of updates and static server resources require sub-systems in the analytic platform to use and share massive data cost-efficiently. However, existing solutions are sub-optimal due to inevitable data isolation and copies in compute-engine-oriented designs. To address this problem, we have developed an experimental data lake storage system, StreamLake. It introduces a novel design to support log message streaming and ETL processing acceleration in distributed storage. Data intensive operations such as message streaming, query operator pushdown, transaction and query time travel are managed inside the centralized disaggregated storage cluster to minimize data copies and shorten time windows in analytic pipelines. Storage features such as multi-level caches and erasure coding along with algorithms like reinforcement learning and probabilistic network are also applied in StreamLake to further optimize query time and resource usage. We have experimented with this framework using production data from China Mobile, the world’s largest mobile network operator, and the results demonstrate improvements of 30% to 4x in terms of performance and over 39% in terms of server resource gains.

ACM Reference Format:

Xin Tang, Dawei Zhao, Haohai Ma, Jiaquan Zhang, Yong Zheng, Zhenyong Fan, Yue Zhao, Yi He, Keji Huang, Zhongsheng Xu, Yidong Wang, Tao Tao, Zhuo Chen, Guoliang Li, Jiwu Shu, Chengliang Chai. 2023. Separation Is for Better Reunion: Data Lake Storage at Huawei. In Proceedings of the 2023 International Conference on Management of Data (SIGMOD ’23), June 18–23, 2023, Seattle, WA, USA. ACM, New York, NY, USA, 13 pages. https://doi.org/XXX/XXX

**1** INTRODUCTION

\* Xin Tang and Dawei Zhao contributed equally to the paper.

† Keji Huang is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD ’23, June 18–23, 2023, Seattle, WA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN XXX/23/06. . . $15.00

https://doi.org/XXX/XXX

As Internet of Things (IoT) and 5G communication technologies become mature and widely commercialized, massive data are collected, stored and analyzed. The traditional architecture of data infrastructure at data centers has been challenged by cloud-native designs. Compute and storage resources are pooled to be able to serve massive structured and unstructured data in an elastic and cost-efficient manner. Analytical systems such as data warehouses and big data platforms have also evolved from siloed construction to disaggregated storage and compute architecture with horizontal integration of resource pools. Thanks to its 10x better price, availability and persistence compared to traditional storage formats, data lake storage [4] becomes a de facto cost-friendly storage layer in this architecture, storing massive online and offline data captured for large scale data analysis. Enterprise data engineers and analysts can easily use data lake storage implementation such as AWS S3 [4] or Huawei OceanStor Pacific [21] as an affordable and reliable centralized pool of storage to save full data. On top of it, they build data preparation and analytics pipelines to assist business decision making, serve customers and meet compliance requirements.

However, as large enterprises further digitalize their business, the data to be stored and analyzed explode. The management costs rise rapidly and become a heavy burden. For instance, 4.8 petabytes fresh data flow into data lakes daily for storage and analysis in China Mobile, the [world's largest mobile network operator](https://en.wikipedia.org/wiki/List_of_mobile_network_operators" \o "List of mobile network operators), and the existing architectures of analytic systems are inapt to scale and process petabytes of data gracefully. Hence, expensive additional operations have to be introduced to support analytic needs. Similarly, we have observed the same situation in other large cooperate customers which Huawei closely works with. A large amount of resources and costs have to be added as the analytic system scales. Although storage increase is inevitable as data grow, a large portion of the cost increases in the analytic systems is because compute engines in data pipelines fail to share data management properly. Hence, massive server resources are wasted in data transfers, re-computing and re-storing intermediate states across engines even though these engines may use same data inputs.

To address this common problem of big data platforms in enterprise production environments, researchers and engineers in our data management community have evolved existing compute engines or designed new system components to converge data pipelines and enhance data re-usage. For example, first, unified engines [6, 13, 20, 38] for stream and batch processing as well as lakehouse technologies [7, 8] with ACID-compliant transactions are developed. These approaches are closely tied to a specific compute engine, optimizing an end-to-end process with a point view. However, in order to cover various business analytical needs, enterprise production data pipelines need close collaboration between multiple analytic tools. Hence, rather optimizing for a single step or module, it requires us to take an end-to-end view to make effective collaborative optimizations in real world. Second, in the modern cloud-native storage-disaggregated architecture, network overhead is expensive if we maintain a shared state layer in the compute cluster which requires frequent and massive data accesses to the storage. Third, separating data management from the storage layer misses the opportunities to apply advanced storage technologies [21, 27, 33] which are critical to process massive data cost-effectively. Finally, data layout management strategies such as compaction and partitioning are key to ensure overall storage utilization and query performance. Existing approaches [7, 8] normally apply static or manual methods to compact small files or manage data partitions. This is sub-optimal compared to dynamic self-learning algorithms in production with complex data pipelines and massive data. We argue that it is reasonable to dedicate the data management component to the storage side and intelligently optimize it on top of the data lake persistent storage to enable shortest dataflows and to maximize the potential of data re-usage across engines in a cost-efficient and collaborative manner.

In this paper, we present a data lake storage framework, StreamLake, with its novel design to support massive message stream ingestion and data pipeline co-processing. This framework provides community compatible API for message stream processing, allowing inflow messages to bypass the compute layer and to be injected to the data lake storage cluster directly. It also supports ACID-compliant transactions, query time-travel, and query operator pushdowns to minimize data transfer and maximize data sharing. A new data lake optimizer is introduced where reinforcement learning and probabilistic network algorithms are applied to data layout management to optimize query time and server resource usage. All these features are built on top of the Huawei OceanStor Pacific storage and the system applies advanced capabilities in the Huawei storage such as in-cluster RDMA network, tiered storage, erasure coding and instant snapshot [21] to achieve superior system resource utilization with reasonable costs. With this centralized stateful storage, analytic engine instances can be more elastic and hence server resources can be highly utilized. We have experimented this framework in a China Mobile data lake with 20 petabytes of data. The system has demonstrated significant server usage gains. We conclude our contributions as follows:

* We propose a novel design of a data lake storage co-processing framework to unify log message streaming, processing and querying acceleration in disaggregated data lake storage, improving both the resource utilization and the speed of data processing pipelines in scale.
* We design a scalable message stream storage with separated control and data planes. It offers ecosystem-compatible messaging system APIs and is highly scalable and reliable, being able to directly consume billions of records with cost-efficient tiered persistent storages.
* We extend the message streaming service to support lakehouse data formats for query concurrent processing. It provides lakehouse tabular abstraction to computing engines to concurrently access millions of records as ACID-compliant tables, improving the granularity of the data usage significantly.
* We introduce a new data lake optimizer in storage. It applies reinforcement learning and probabilistic network algorithms with storage characteristics and query history to optimize the data layout dynamically, optimizing both the query time and the storage block utilization.
* We conduct a use case study with the China mobile IT team to evaluate the StreamLake implementation. Compared to the current system, the experiments show that the new design brings in 39% to 4x resource usage and performance gains.

The rest of this paper is organized as follows. Section 2-6 detail the motivation, design and implementation of key components such like the stream storage, the lakehouse framework and the data lake optimizer. Section 7 discusses the experiments. Section 8 compares related work. Conclusions are drawn in Section 9.

2  MOTIVATION

The design of StreamLake is motivated by Huawei storage customers. In the past several years, we have been closely working with over 200 enterprise customers in 16 industries and the key statistics of their big data processing are summarized as follows:

* **Petabytes of data**. Big data teams in 49% customers have processed data between one terabyte and 10 petabytes (PB). 29% process more than 10 PB and 8% process over 100 PB.
* **Data store at least one year**. 43% customers are required to store data between 1 and 5 years. 22% store between 5 and 10 years and 27% store at least 10 years, according to regulations and practices in different industries.
* **Log data**. 81% customers report that the primary data formats they work with are log messages.
* **Stream and batch processing**. 69% customers use batch processing and 65% customers use stream processing actively. 39% customers use both stream and batch processing.

It has become a widely adopted practice to store petabytes of log messages for more than a year, using stream and batch data processing techniques to support business applications in industries. After drawing these big data user profiles, we have further explored the key factors to consider for future big data project deployment with our customers. The top priorities we have identified are:

* **Processing efficiency**. More than 71% customers report that their platforms are required to support interactive queries for end user mobile apps and interactive business applications. Hence, highly efficient data processing is anticipated. Furthermore, enterprise customers working with petabytes of data report that it often requires weeks to months to extend the business data pipelines to support new analytic requirements due to the complexity of tools in the big data stack. Complete and flexible big data pipeline enterprise solutions are needed in order to shorten launch time in new analytic use cases.
* **Storage scalability**. The data volumes are exparchiected to continue growing rapidly. For instance, as 5G users rise, log data collected by our communication carrier customers increase 5x. The storage systems should be able to scale gracefully to store petabytes of new data.
* **Cost**. Flexible and cost-effective end-to-end solutions are important in real world deployment. For instance, some data centers of our customers still use 1 GE network although 10 GE network has already been a standard practice. When we deploy a new system to these data centers, the system should be able to remain its SLA even though facing network constraints. As many customers’ IT budget growths are often slower than the exponential growth of their data volumes and application requirements, our system design should take customer’s previous IT investment into accounts and provide flexibilities to deploy in various datacenter infrastructures.
* **Reliability**. Data reliability and security remains a top priority in our customers’ big data usage as any loss or leak of data could lead to serious damage to the company business and brand. An enterprise data pipeline should provide built-in data reliability and security to provide full protection to the data.

3  ARCHITECTURE

To satisfy enterprise customers’ needs in the next generation big data solutions, we have designed and implemented StreamLake, a data lake storage system based on the Huawei OceanStor Pacific storage. The system aims at optimizing the end-to-end processing of massive log messages in big data pipelines. Figure 1 shows the architecture of StreamLake which consists of store, data service and access three layers from a high-level perspective.

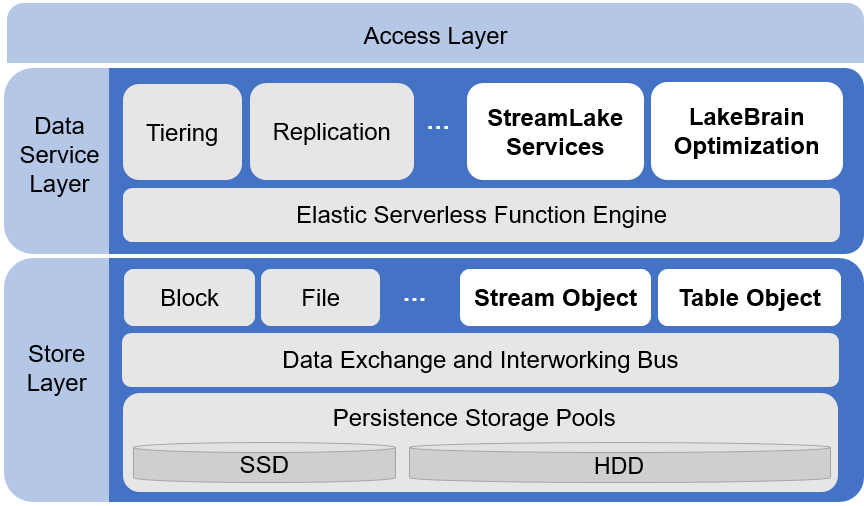


Figure 1: StreamLake Storage Architecture.

The store layer is responsible for data persistence. It consists of SSD and HDD data storage pools, a high-speed data exchange and interworking bus as well as block, file and other storage semantic abstraction. 1) The SSD and HDD data storage pools provide reliable data storage management. Physical storage space of SSD and HDD disks in the storage cluster is divided into slices. Slices from different servers and disks are organized as logical units to provide data redundancy and load balance. The storage pools also implement storage space features such as garbage collection, data reconstruction, snapshot, clone, WORM and thin provision, etc. 2) The data exchange and interworking bus offers high-speed data transfer and interworking of different storage abstraction. It supports RDMA to bypass CPU and L1 cache and leverages intelligent stripe aggregation, I/O priority scheduling and other technologies. All nodes are interconnected by the data bus to enable high IOPS, large bandwidth and low latency data exchanges. The data bus also supports interworking of storage abstraction. One piece of data can be shared and accessed by different interfaces, eliminating data migration and saving storage space. 3) The block, file and other storage abstraction implements access interfaces to the underlying storage in different semantics. We introduce two new abstraction, stream object and table object, to manage messaging streams and tabular data efficiently. Their implementation is discussed in section 4.

The data service layer provides a rich set of features to perform various data management tasks at enterprise scale. For instance, the tiering service offers static and dynamic data migration and eviction between the SSD and HDD storage pools based on tiering policies. The replication service provides periodical replications to remote sites for backup and recovery. We extend the data service layer to provide services and optimizations for log message processing operations (StreamLake services and LakeBrain optimization namely). These services and optimizations are elaborated in section 5 and 6. The elastic serverless function engine is a component that we introduce to support near data processing of the StreamLake services. Its design is discussed in section 5.3.

Finally, the access layer implements storage access protocols to handle user requests. It supports a block service via standard iSCSI access, NAS services via NFS and SMB protocols as well as an object service via S3 protocol, etc. The new StreamLake services utilize the OceanStor distributed Parallel Client (DPC) which is a universal protocol-agnostic client providing shorter but superfast IO path. The access layer is also responsible for authentication and ACL permission control. Only valid user requests will be translated to internal requests and forwarded to other modules to be processed.

4  STREAM AND TABLE STORAGE OBJECT

In this section, we introduce stream object and table object in the store layer. They are purpose-built semantic storage abstraction to store and access stream and table data efficiently.

**4.1 Stream object**

Stream object is storage abstraction in the store layer to support key-value message streaming in scale. It stores a partition of key-value pair records for a continuous message stream. Data in a stream object are organized as a collection of data slices. As shown in Figure 3, a slice can contain up to 256 records. Every incoming message record is appended to a specific slice in a stream object according to its record topic, key and offset.

1. int32\_t **CreateServerStreamObject** (

2. IN CREATE\_OPTIONS\_S \*option,

3. OUT object\_id\_t \*objectId);

4.

5. int32\_t **DestroyServerStreamObject** (

6. IN object\_id\_t \*objectId);

7.

8. int32\_t **AppendServerStreamObject**(

9. IN object\_id\_t \*objectId,

10. IN IO\_CONTENT\_S \*io,

11. OUT uint64\_t \*offset);

12.

13. int32\_t **ReadServerStreamObject(**

14. IN object\_id\_t \*objectId,

15. IN uint64\_t offset,

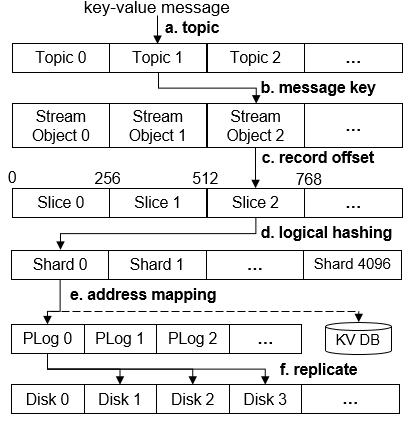
16. IN READ\_CTRL\_S \*readCtrl,

17. INOUT IO\_CONTENT\_S \*io);

**Figure 2: Stream Object Operations.**

Similar to the block and file storage abstraction in the store layer, the stream object implements and encapsulates read/write operations for the stream storage. Figure 2 lists some core operations supported by the stream object. CreateServerStream-Object (line 1-3) and DestroyServerStreamObject (line 5-6) are functions to create and destroy a stream object. The option field (line 2) sets storage configurations such as the data redundancy method (replicate vs erasure code) and the I/O quota to ensure enterprise class reliability and performance. An objectId (line 3) is assigned when the stream object is successfully created. It is used as a unique identifier to operate the stream object. Function AppendServerStreamObject (line 8-11) appends incoming records as IO\_CONTENT\_S to the stream object and returns the beginning offset of the appended records for future reads. Function ReadServerStreamObject (line 13-17) reads the stream object starting from the given offset. Field readCtrl sets control conditions such as the length to read. As the messaging service is designed to support real time streaming, it is by default configured to return all the following messages unless specified or reaching quota limits. IO\_CONTENT\_S (line 10 and 17) is a data structure that offers non-blocking I/O, applying buffers to speed up writes and reads.

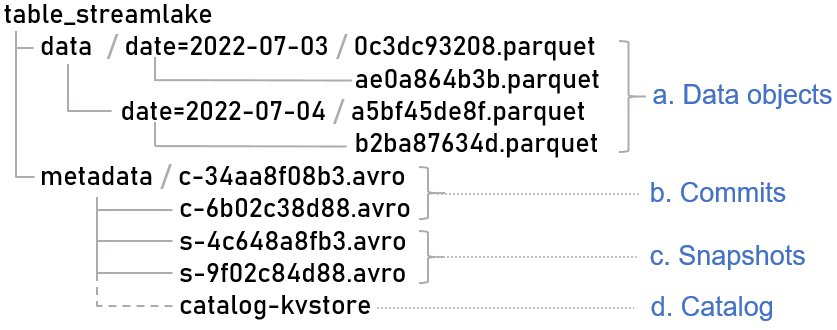
The underlying SSD and HDD storage pools provide enterprise class load-balanced and redundant persistence for the stream objects. Figure 3 illustrates the persistence process in the StreamLake distributed storage. Messages are assigned to stream object slices based on their topics, keys and offsets. A distributed hash table is applied to ensure dynamic even data distribution. Data slices of all stream objects are distributed to 4096 logical shards evenly. Each shard is assigned with storage space managed by persistence logs (PLog). PLog is a collection of persistence services in OceanStor [21]. Each PLog unit controls some fixed size of storage space in multiple disks, providing 128 MB of addresses to a shard. When receiving a message, the PLog unit replicates it to different disks and realizes the redundancy strategy. For fast record lookup, key-value database serves as indexes for PLogs.



**Figure 3: Writing Messages to StreamLake.**

**4.2 Table object**

We extend the storage object abstraction layer in StreamLake to support table-like operations, providing fine-grained data storage and management similar to lakehouse [7, 8]. The storage of the table abstraction adopts an open lakehouse format with optimizations to accelerate metadata accesses. It is logically defined by a directory of data and metadata files shown in Figure 4.



**Figure 4: File Organization in a StreamLake Table Object.**

Parquet files in the data directory stores the table data objects. In this example, the table is partitioned by the date column hence data objects are separated in different sub-directory by date. The name of each sub-directory denotes its partition range. Data objects in each parquet file are organized as row-groups and stored in a columnar format for fast analytic data access. Footers in the parquet files record statistics to support in file data skipping.

Metadata in the table abstraction records table schemas, file addresses for the table and its partitions as well as transaction commits. It is organized in commit, snapshot and catalog three levels as illustrated in Figure 4 - b, c, d. Commits are arvo files that contain file-level metadata and statistics such as file paths, record counts and value ranges for data objects. In each data insert, update and delete operation, a new commit file is generated to record changes of data object files.

Snapshots are index files pointing to valid commit files in the given period of time. Commit statistics such as current file and row count, added file and row count and removed file and row count are documented in snapshots as data operation logs. Snapshots and commits together provide snapshot level isolation to support optimistic concurrency control. Readers read from the valid commit files. Changes made by a writer will not been seen by the readers until they are committed and recorded in a snapshot. Therefore, one writer and multiple readers can access the data simultaneously without needing a lock. As snapshots monitor expiration time of all commits, they are also essential to support time travel. A time travel query returns data as it appeared at a specific time. As the table object keeps old commits and snapshots as long as needed, it allows us to use a timestamp to lookup the corresponding snapshot and commits to access historical data.

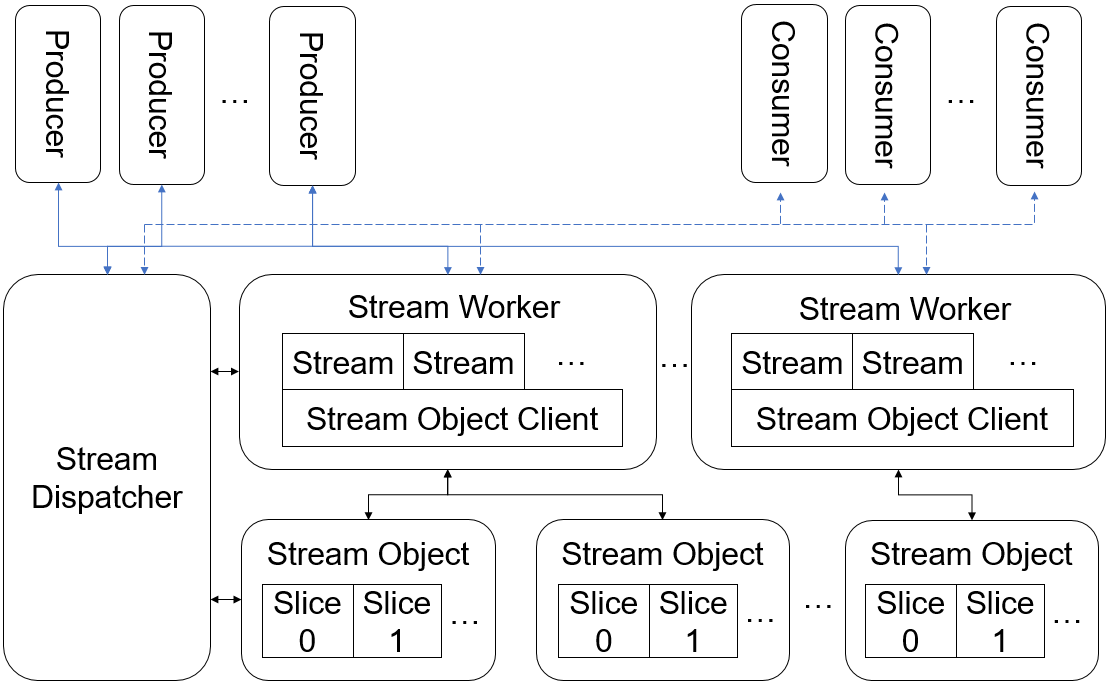
Finally, the catalog describes the table object, comprised of table profile data such as table IDs, table directory paths, table schemas, snapshot descriptions and modification timestamps, etc. Data and metadata files are stored in the table directory except the catalog. The data and metadata files are converted to PLogs in the underlying storage for redundant persistence. The catalog is stored in a distributed key-value engine optimized for RDMA and SCM to ensure high performance metadata access.

5  STREAMLAKE DATA PROCESSING

In this section, we introduce data processing services in the data layer. Motivated by real world use cases summarized in Section 2, these services provide an end-to-end enterprise class data lake storage solution to store and process log messages in scale. The StreamLake services include a stream storage for message streaming, lakehouse-format read/write for tabular data processing as well as query operator computation pushdown supports.

**5.1 Message Streaming**

We implement a distributed stream storage engine to provide message streaming in scale. This engine is built on top of the stream object storage abstraction to provide enterprise class reliability and scalability. Figure 5 shows the high-level design. The stream storage consists of producers, consumers, stream workers, stream objects and a stream dispatcher.



**Figure 5: Message Streaming Service.**

Producers are upstream processes that publish messages to a topic. Consumers in the downstream subscribe to topics and processes feeds of published messages. As our customers have used open-source message streaming services in some production environments, the producer/consumer message APIs are designed to be compatible with the open-source de facto standard to maximize connections to the ecosystem. Users can migrate their applications to StreamLake with minimal costs. Figure 6 shows sample code snippets to write and read messages. We create a producer and write a new message “Hello world” to a topic named “topic\_streamlake\_test” as a key-value pair. A consumer subscribes to this topic and processes published messages.

1. // Sample producer code

2. Producer producer = new Producer(...);

3. Message msg = new Message("Hello world");

4. producer.send("topic\_streamlake\_test", msg);

5.

6. // Sample consumer code

7. Consumer consumer = new Consumer(...);

8. consumer.subscribe("topic\_streamlake\_test");

9. while(true) {

10. // poll for new data

11. }

**Figure 6. Sample Producer and Consumer Code.**

Stream workers and stream objects process streams and store messages. Stream workers are created along with the message streaming service. Its number is determined by configurations and physical resources allocated to the stream storage. Each stream worker can contain multiple streams and one stream object client. When a topic is created, a stream is added to one of the stream workers in a round robin fashion so that streams evenly spread across the cluster to balance workloads. Each stream is mapped to a unique stream object in the store layer. As we have introduced in section 3.1, stream object is purposed built storage abstraction to support key-value message streaming. It provides efficient interfaces and implementation to write / read streams to and from the storage pools. The persistence process has been illustrated in details in Figure 3. Stream object clients monitor stream objects and implement message delivery. They unwrap messages from clients, encapsulate them in the stream object data format and redirect them to the corresponding stream objects via RDMA. To guarantee message delivery, stream object clients actively monitor the health of stream objects to which they connect and periodically exchange critical service data with the dispatcher service. The synchronization includes reporting the health of the stream object connections and refreshing stream objects a client connects to.

The stream dispatcher maintains the messaging service metadata and configurations, handling external and internal requests to dispatch message streams with proper resources. The topology of topics, streams, stream workers and stream objects are stored as key-value pairs at a fault tolerant key-value store in the stream dispatcher. Once there is a component status change such as a stream worker or topic being added or removed, metadata records in the key-value store will be updated immediately to refresh the topology tracking. This topology tracking helps the stream dispatcher handle requests to dispatch message streams. When a producer or consumer connection request arrives at the stream storage, the stream dispatcher routes the request to a stream worker according to the stream topic, setting up a direct message exchange channel between the producer, the stream worker and the consumer.

The stream dispatcher also sets configurations for the messaging service in the unit of topic. Figure 7 shows an example of some enterprise class feature configurations.

* *stream\_num* sets the topic parallelism. This configuration needs to be provided during the topic declaration. In our example, 3 streams are created for the topic. They are evenly assigned to stream workers, processing messages in parallel.
* *quota* sets the maximum processing rate. Each stream in our example can process up to 1000000 messages per second.
* *scm\_cache* enables use of SCM caches.
* *convert\_2\_table* enables auto-conversion from stream object messages to table object records. Once this configuration is set, a background process will apply *table\_schema* to convert messages to table object records periodically, saving them in the table object directory *table\_path*. The conversion is triggered by conditions set in *split\_offset* and *split\_time*, whichever comes first. In our example, the dispatcher does a stream to table conversion when the topic accumulates 10000000 messages or the time passes 36000 seconds.
* *archive* automates historical data archiving to meet business and regulation requirements. The data can be saved in the cost-friendly archive storage pool in StreamLake which is pre-configured or can be archived to an external storage system specified at *external\_archive\_url*. *archive\_size* denotes the data volume in MB to trigger archiving. *row\_2\_col* sets whether archiving in a columnar format.

1. { "stream\_num" : 3,
2. "quota" : 1000000,
3. "scm\_cache" : true,
4. "convert\_2\_table" : {
5. "table\_schema" : { … },
6. "table\_path” : …,
7. "split\_offset" : 10000000,
8. "split\_time" : 36000,
9. "delete\_msg" : false,
10. "enabled" : true }
11. "archive" : {
12. "external\_archive\_url” : null,
13. "archive\_size" : 262144,
14. "row\_2\_col" : true,
15. "enabled" : true } }

**Figure 7. Stream Storage Configuration Example.**

As an enterprise class stream storage, StreamLake supports delivery guarantee, efficient transfer and high elasticity.

**Delivery Guarantee.** Message delivery could be interrupted by network failures, ACK delays and other accidents. Our system ensures consistent message delivery through the four following aspects: 1) Data in the same stream object is strictly ordered. That means messages arrive first will be consumed first. 2) Message writing is idempotent. In the case of a network failure, the producer may send duplicate messages. We identify the producer ID and message sequence number to determine whether the message is a duplicate. 3) Strong data consistency. The entire system does not depend on file system, page cache, or any unreliable components. The persistence of all the data is built based on the stream object which can tolerate the failures from node, network and disk.4) The system supports exact-once semantics. There are two typical scenarios. The first is that a producer writes a batch of messages to multiple streams. All messages are either successfully written or failed. The second scenario is that the application reads the message, processes the message, and writes the message to a new stream and the whole process are either successful or failed. To provide exactly-once semantics for both scenarios, we introduce a transaction manager at the stream dispatcher. This manager implements the two-Phase Commit Protocol, marks and logs all participant (consumer and producer) actions to make sure all results in a transaction are visible or invisible at the same time.

**Efficient Transfer**. We implement three mechanisms to transfer data efficiently. 1) Stream workers and stream objects are connected using the data bus with RDMA. It reduces switch overheads between the user space and the kernel space in the TCP/IP protocol stack. 2) We implement an I/O aggregation mechanism to aggregate small I/O requests and increase throughput. This function is enabled by default and can be disabled in latency-sensitive scenarios. 3) We introduce a local cache at the stream object client to accelerate message consumption. A small memory buffer is allocated to each stream as a read cache.

**High Elasticity.** Decoupling data storage at the stream objects and data serving at the stream workers enables high elasticity to cope with service peaks and toughs. The message streaming service can scale up and down without data migration. When receiving a scaling request, the service acquires or releases system resources to update the number of stream workers and rewrites the mapping relations between the stream workers and the underlying stream objects. The whole process involves no data migration, so it is very efficient and can be completed within seconds.

**5.2 Lakehouse Read and Write**

The StreamLake services support concurrent tabular data reads and writes similar to lakehouse. In this section, we first introduce the storage native conversion from stream messages to tabular records. Then we discuss the lakehouse operations in details.

Log messages in the stream storage can be converted from stream messages to table records natively. The conversion is performed by a background service triggered by the *convert\_2\_table* conditions in Figure 7, which includes the table schema and bounds for data freshness in the downstream processing. The table schema needs to be specified at the topic declaration as it sets expectations to field types and values for all messages. Once the conversion conditions are satisfied, the conversion process will automatically convert records in stream objects to table objects, signals new arriving data for downstream processing. To best preserve the storage volume, users can reserve only messages in critical topics as stream objects to support real time applications while converting most data to table objects. When data needs to be played back, we support a reversed conversion from table records to stream messages. As shown in Figure 13 and Table 1, this design provides a balance between the system cost and the processing efficiency. It also helps decoupling the data processing from the business logics.

The StreamLake services implement lakehouse read/write operations. Our implementation is based on the table object and applies high performance caches and computation pushdowns which eliminate unnecessary data transmission and accelerate concurrent data reads and writes. The rest of this section will introduce the implementation of key read/write operations in details.

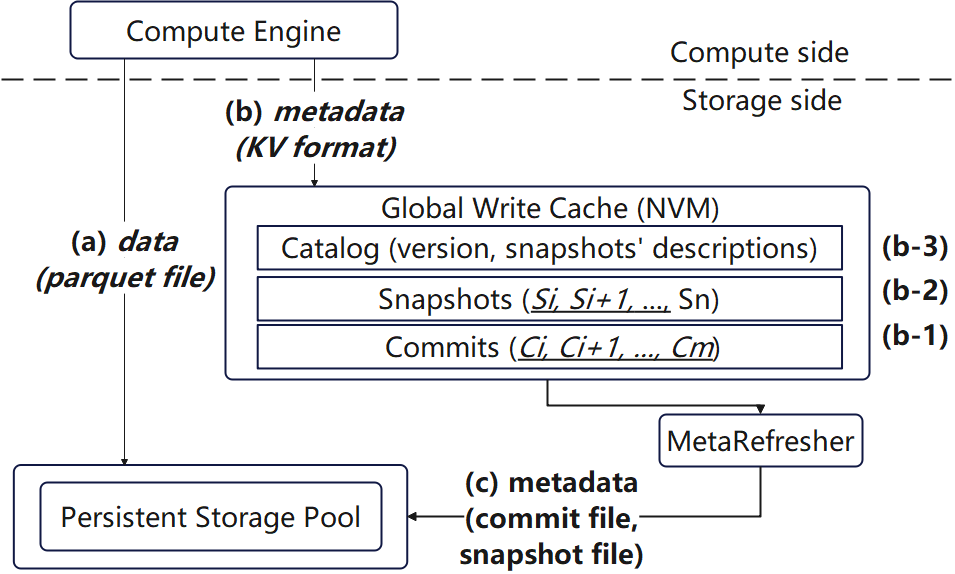
**CREATE TABLE**. The create table operation first registers the table information (schema, path, database, table name, etc) in the catalog, and creates the */data* and */metadata* folders under the table path. Then table configurations (schema, partition spec, target file size, etc) are written to the metadata directory for persistence.

**INSERT RECORD**. The insert record operation includes persisting data, caching metadata to the non-volatile memory (NVM) and persisting metadata. The NVM cache is introduced to combine small I/O accesses to the underlying storage pools.

a) Data Persistence: Records are directly written to the persistent layer as parquet files in the corresponding partition path (*/data/partition*) under the table root directory.

b) Metadata Caching: Metadata updates are mostly small I/O operations. To avoid generating significant number of small files, we leverage a global write cache to aggregate the metadata updates. b-1) Each added parquet file generates a commit record, containing file-level metadata and descriptions. When a commit is made, all the new commit records are written to the write cache as key-value pairs. b-2) The latest snapshot will be read from the persisting layer to the cache (if not cached yet) and its commit data will be updated. b-3) Finally, the snapshot descriptions and the version history in the catalog are also read from the persistence layer and are overwritten by adding the latest snapshot description.

c) Metadata Persistence: Metadata in the NVM write cache is asynchronously flushed to the persistent storage pool when the buffer is full. A metadata management process (MetaRefresher, namely) transforms the commits and snapshots from key-value pairs to files and writes them to the *table/metadata* directory.



**Figure 8. Write Cache Acceleration in Lakehouse Read/Write.**

**SELECT RECORD**. The select operation first reads the catalog to retrieve the table profile for collecting the list of snapshot files needed for this query, such as the metadata version and snapshot descriptions. Then the corresponding snapshots and commit metadata are read from both the NVM cache and the persistent storage pool to generate the latest complete snapshots and commit metadata. When all the record file addresses are confirmed, data are read from the persistence pool by read tasks.

**DELETE RECORD**. The delete operation first executes a select operation to find files containing matched records, in which there are two different cases. If the filtering conditions match all data in several partitions, only the metadata will be updated, i.e., a new commit version is generated directly by eliminating the information of deleted partitions. Else, if the filtering conditions only match some files, then these files will be read and the data matching the filtering condition will be deleted. Specifically, with computation pushdowns, the file read and write process is performed without data transmission to/from the compute engines.

**UPDATE RECORD**. Similar to the delete operation, update does a select to find files that contain records matching the filtering conditions. Then pushdowns are applied to process file reads and writes, reducing data movements.

**DROP TABLE**. There are two kinds of drop table operations: drop table soft and drop table hard. 1) Drop table soft only unregisters the table from the catalog, retaining the table's metadata and data in the persistent layer, which can be used to restore the table later. To restore a soft deleted table, one can create a new table and link it to the original table path, which actually reregisters a deleted table back to the catalog. 2) Drop table hard removes the metadata (all files under */metadata*) and data (all files under */data*) of the table at the same time, after which it also clears the table in catalog. Since part of the metadata may have been written to the acceleration cache during a drop table hard operation, which will be flushed to the persistent layer asynchronously in the background, the operation to delete the metadata will first clear the metadata in the cache and then delete the metadata on the disk.

**5.3 Query Operator Computation Pushdown**

This section introduces a query operator computation pushdown feature. In our disaggregated-storage scenario, it reduces data transferring from the StreamLake data lake storage engine to the query engines. This feature is built on top of an elastic serverless engine in the data service layer. We choose serverless computing as our computing execution model because its lightweight design gives us flexibilities to use and release server resources upon requests. We can start a large number of instances right away to execute computation tasks near the data sources. When the tasks are completed, resources can be free without effort. This elasticity is critical as CPU resources is relatively scarce while the system is actively performing critical data management jobs. Figure 9 presents the main components of the serverless function engine.

* A function dispatcher is responsible for job scheduling and workflow controls.
* Worker instances execute the jobs.
* A worker manager manages server resources and the life cycle of worker instances.
* A function repository registers and stores function images.

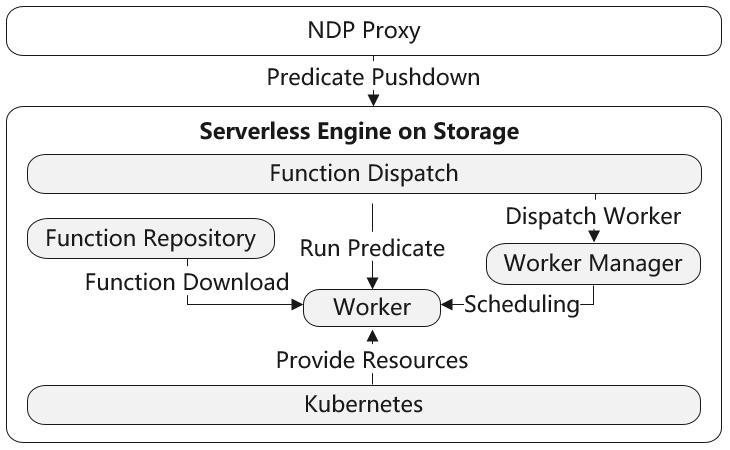
These modules interact with each other to support elastic serverless computing. When receiving job requests, the function dispatcher obtains data locations from the storage devices, selecting appropriate storage nodes based on data distribution and available node resources. Then it requests the worker manager to deploy worker instances to the selected node. Worker instances download functions from the function repository and execute the jobs with data read from the storage infrastructure. When the jobs are terminated, a callback message is sent to the caller to notify the results. Elastic scaling policies are used to adjust the number of nodes dynamically based on the workload to guarantee service quality. For instance, a load balancing method is used to balance scheduling among instances.

To maximize the pushdown benefits, three categories of query operators are supported.

* Projection Pushdown: Only selected columns will be returned.
* Filter Pushdown: Only rows meeting the filtering conditions will be returned.
* Aggregate Pushdown: The results of aggregate functions such as count, max and avg will be returned.

These three types of operators are chosen because the sizes of their outputs could be significantly smaller than the inputs, thereby a large optimization opportunity can be found. The operators are wrapped as different functions, registered and run in the serverless engine service. As a generalized implementation, different query engines can share and run the same query operator function once its image is register in the serverless engine.

It requires close collaboration between the query engines and StreamLake to pushdown query operator from the compute layer to the storage cluster. Therefore, we design a dedicated entity, NDP Proxy, to coordinate the interactions. During query planning, the query engine optimizer applies predicate pushdown rules to identify operators that can be pushed down. Then in execution query plan fragments for these predicates are sent to the NDP proxy. The NDP Proxy inserts these requests into a block queue as a traffic control. When appropriate, it send these function requests to execute in the serverless engine as illustrated in Figure 9. Immediately upon the results are ready, the NDP Proxy sends the data back to the compute engine. StreamLake’s high-speed data exchange bus is used to ensure efficient data transfers.



**Figure 9: Serverless Function Engine.**

6  LAKEBRAIN OPTIMIZATION

Optimizers in data warehouse and big data systems play a critical role to optimize large-scale query processing and data management [17, 29, 30, 35, 36, 39]. However, in a complex storage-disaggregated analytic platform with multiple compute engines, it is prohibitively difficult for a single query engine optimizer to make ideal decisions to improve the end-to-end data pipeline performance and resource consumption. This task is challenging due to the lack of the knowledge on both the compute and storage cluster environments as well as queries executed by other engines simultaneously. Even though all these statistics were given, the optimization would be still painful since the search space would be humongous considering the large number of tunable and interdependent variables [23].

In respond to this challenge, we introduce a novel data lake storage optimizer, LakeBrain, to complement the end-to-end data pipeline optimization. Unlike optimizers in query engines focusing on join ordering and cardinality estimation [42], LakeBrain intends to optimize data usage in storage during query execution which is key to improve both query execution performance and storage resource utilization in a storage-disaggregated design. For instance, data ingestion and transaction in a streaming application scenario typically result in an enormous number of small files, giving rise to low query performance on merge-on-read (MOR) tables. LakeBrain can employ a compaction strategy to combine numerous small files into few large ones to improve inter-cluster storage and network usage as well as query performance.

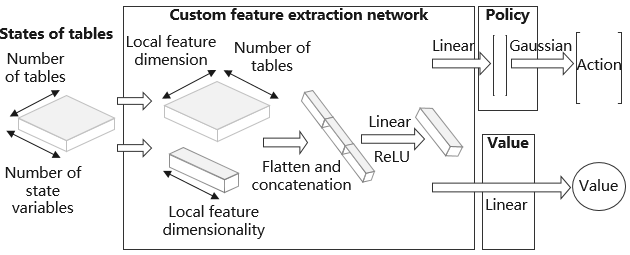
We intentionally keep the logical design of LakeBrain simple so that it is easy to extend and support different applications. Conceptually LakeBrain consists of three components. First is a statistics collector to gather system configurations, environment variables and workload history. Second is the core optimization logic to suggest best strategy candidates. Although rule-based configuration driven by domain knowledge may appear effective to certain extent, it is still highly difficult to achieve the desired optimal performance considering the high-dimensional search space in the data lake storage environment. Thereby our optimization combines heuristic rules, probabilistic models and machine learning algorithms to achieve the best possible results. Finally, there is an executor entity to deploy the optimization strategy to the system. Statistics of its effects are collected as feedback by the statistics collector to improve future optimization.

To illustrate the value of a data lake storage optimizer, we have developed two LakeBrain applications. One is auto compaction and the other is predicate-aware fine-grained partitioning. Details of these two use cases are elaborated in this section.

**6.1 Automatic Compaction**

File compaction intends to search a compaction strategy to reward query execution such as better runtime or higher block utilization in storage. Different algorithms are used in the interactive process of optimizing the compaction strategy. First, particle swarm optimization (PSO) is employed to search for the global optimum, which is a population-based metaheuristic method with no need to assume the mapping from the tunable parameters to the query performance. The goal of applying this efficient global optimization is to obtain an approximately optimal solution within a limited amount of time. Second, a reinforcement learning (RL) algorithm is utilized to find a more sophisticated policy which embodies the rules governing the execution of individual compactions based on the states of the data lake environment.

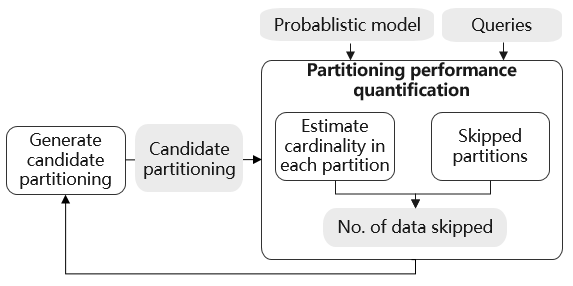
During the optimization, the action space contains a set of discrete compaction configurations. Given the continuous nature of the state space, a function approximation method is preferred. Taking into account the high degree of variability associated with the query performance in a distributed environment, training stability is critical. Therefore, the proximal policy optimization (PPO) is applied, which not only directly updates the policy by maximizing the expected return but also maintains an appropriate policy update in each iteration. A deep neural network (DNN) is designed approximate the policy and the value function, a.k.a. the actor and the critic. They share the feature backbone network, of which the output covers both global and local characteristics of the states. Following the feature network, two multi-layer fully connected networks are attached to compute the policy output and the action value. The actor and critic are alternatively updated after collecting the new trajectories using the latest policy during training. When a desired result is obtained, the abstract numerical output of the compaction strategy is translated by the data lake connector into actionable operations with respect to a certain data lake engine, thereby facilitating the optimization.



**Figure 10: The Network Design for the RL Algorithm.**

**6.2 Predicate-Aware Fine-Grained Partitioning**

Optimizing data partitioning aims to best assign records to storage blocks thereby minimizing the number of blocks accessed by queries. Our partitioning method is based on the query-tree framework proposed in [43]. Considering tractability and fast inference speed of the data in LakeBrain, instead of repeatedly scanning the datasets, we select a probabilistic model, sum-product network (SPN) [19, 26, 31], to model the distribution of the data. Furthermore, we apply time series prediction models to ensure that data accesses are optimized even during data re-partitioning.



**Figure 11: Probabilistic Model-Based Cardinality Estimation for Partitioning Optimization.**

The query-tree framework generates a tree-like partitioning policy using pushdown predicates. Each leaf node represents a partition whose column ranges can be derived from the pushdown predicates used to split its parents. We utilize probabilistic models to characterize the dataset to select the candidate partitioning policy. As shown in Figure 11, the probabilistic model-based cardinality estimation is used to estimate the number of data records in each partition instead of scanning the original data, saving tremendous amount of data scanning time. This technique significantly speeds up the partitioning optimization algorithm and make it applicable for large scale systems. Another benefit of probabilistic models is that we can represent a sequence of dataset with a sequence of probabilistic modes that have a fixed structure but vary in parameters. Through this way, we are able to represent a sequence of datasets as a sequence of multi-dimensional vectors representing all the learnable variables in the probabilistic model with fixed length, i.e. a time series. After that, we can predict the future probabilistic model with time series prediction methods and then use the predicted probabilistic model to estimate the number of data records in a partition during partitioning optimization. This enables our system to provide optimized partitioning for incoming datasets.

To realize the optimized data layout, we introduce a partitioning mechanism which saves the data in fine-grained partitions based on the partitioning strategy. We also implement an evaluator to skip irrelevant partitions by checking the overlaps between the pushdown predicates and the ranges of columns in each partition. For numerical columns, the range can be represented as lower bounds and upper bounds. Data skipping based on lower bounds and upper bounds has been well handled by many data formats. For categorical columns, we either record its range or its complement using “IN” or “NOT IN” predicates. This predicate-aware partitioning approach is evaluated in section 7.2 in which the test results demonstrate excellent performance.

7 EVALUATIONS

To illustrate the end-to-end usage of StreamLake, we examine a simplified real-world use case which compares the StreamLake framework with an open-source storage solution to construct a big data processing pipeline to support business analyses. The business scenario is that a mobile financial application company would like to understand its app usage to prevent frauds as well as improve its product experience and hence it partners with a mobile carrier to collect and analyze its app usage data. The carrier provides this analytic service through an end-to-end big data processing pipeline which consists of data collection, normalization, labeling and querying jobs as illustrated in Figure 13.

(a) **Collection**: The network carrier collects mobile app data packets at its data centers across the nation via deep packet inspection (DPI) and transfers them to a centralized storage pool.

(b) **Normalization**: At the centralized storage pool, the data packets are normalized as records in a unified schema. Data are validated to ensure accuracy and quality. Sensitive data are shielded to protect individual privacy.

(c) **Labeling**: Labels from knowledge bases are added when appropriate to classify the records and identify actionable insights.

(d) **Query**: When the normalization and labeling jobs are completed, the records are inserted into tables and are ready to be invoked by query engines. The app company uses secure API calls to query these data and perform analyses to monitor and improve its business operations. For instance, Figure 12 shows a single SQL query to count the daily active users (DAU) in different provinces. More sophisticated analyses, such as hidden Markov and Gaussian mixture models can also be applied as pattern recognition algorithms to draw user profiles and identify abnormal activities.

1. SELECT COUNT(\*) AS DAU

2. FROM TB\_DPI\_LOG\_HOURS

3. WHERE url = ‘http://streamlake\_fin\_app.com’

4. AND start\_time >= 1656806400 -- July 3rd, 2022

5. AND start\_time < 1656892800 -- July 4th, 2022

6. GROUP BY province;

**Figure 12: Query Calculates DAU**

To support both full data and real time analyses, the network carrier builds two data flows in the data analytic processing pipeline. One flow runs full data in batch every two hours and the other streams messages constantly to deliver time-sensitive logs such as new logins, payments and password modifications.

This use case is evaluated in a commodity cluster using different sizes of input data packets and the results are compared with open-source storage solution Hadoop Filesystem (HDFS) and Kafka. These two storage systems are chosen because of their popularity in real world and are relatively easy to provide context to illustrate the usage of StreamLake. The cluster hardware contains 3 nodes, each of which has 24 2.30 GHz cores and 256 GB RAM. It is configured as a 3-node StreamLake storage cluster when we measure StreamLake. While running the open-source solution, it is configured to host a 3-node HDFS storage and a 3-node Kafka cluster simultaneously. The number of input data packets are 10 million, 50 million, 100 million, 500 million and 1 billion. Since a data packet is 1.2 KB on average, the data volumes we measure are 12 GB, 60 GB, 120 GB, 600 GB, 1.2 TB, correspondingly.

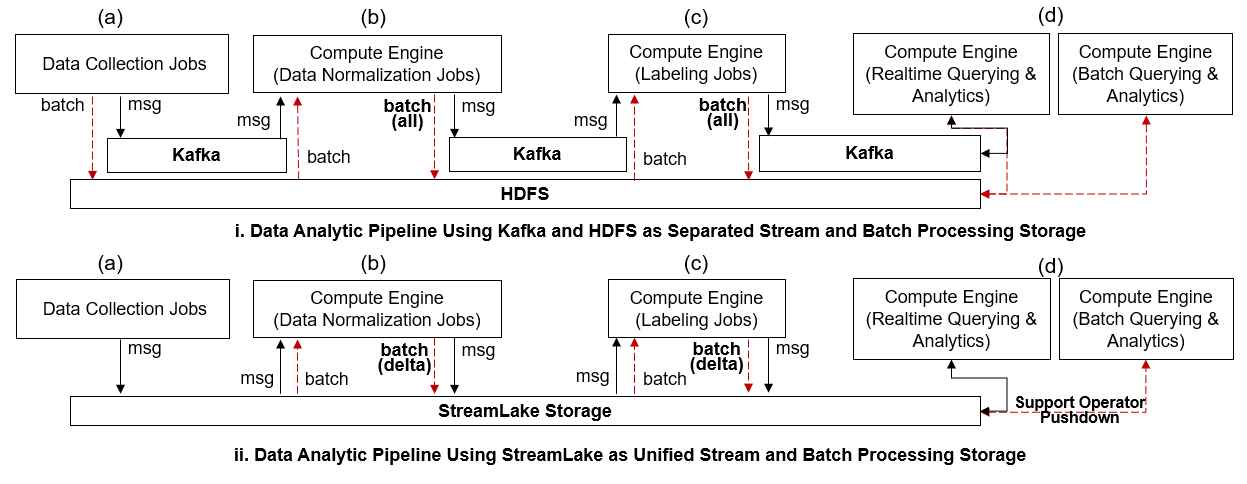
Figure 13 shows the data processing process. In the open-source approach, Kafka and HDFS serves as independent stream storage and batch storage respectively to pass data between collection, normalization, labeling and query jobs. As a standard ETL practice, a new copy of all data is written to HDFS and Kafka after each job. In case it fails accidentally, a job can read its input data to reproduce the results. In the StreamLake solution, StreamLake serves as unified stream and batch processing storage. It reads messages from the data collection jobs and passes messages and aggregated batches to the same stream and batch processing engines in the normalization, labeling and query jobs. As StreamLake supports time travel, only updated rows are written to storage. When a job needs to re-run, it can use time travel to retrieve its input data. During the query jobs, the three filters in the WHERE clause and the COUNT aggregate in Figure 12 are pushed down to compute in StreamLake, accelerating querying.

Table 1 shows the results of the experiments. The numbers of input data packets are in the top row. The storage usage and processing time for StreamLake (S), HDFS (H), Kafka (K) are in the following rows. HK denotes the sum of the storage usage in HDFS and Kafka. The ratio of storage usage (HK/S) and processing time (K/S, H/S) are calculated to compare the results.

The experiment demonstrates that StreamLake significantly improves the total storage usage and the batch processing time. The storage usage in the HDFS and Kafka approach is 4 times as much as StreamLake. In the HDFS and Kafka approach, full data are written into the storage when each ETL job is finished. This is a common practice to support downstream jobs restart after unexpected failures in large-scale production ETL pipelines. As a result, 6 copies of full data are written into the storage. In the StreamLake approach, since the storage natively supports time travel, we only save 1 copy of full data plus updates in each ETL job, saving about 75% storage usage.

The batch processing speed in StreamLake is better than HDFS when the workload is 50 million records or more. As the workload grows, the advantage to skip irrelevant partitions becomes significant. StreamLake is 50% faster than HDFS when the workloads are 500 million and 1 billion records. On the other hand, StreamLake is less ideal for small workload. When the workload is 10 million records, StreamLake is 20% slower than HDFS as it performs extra metadata managements.

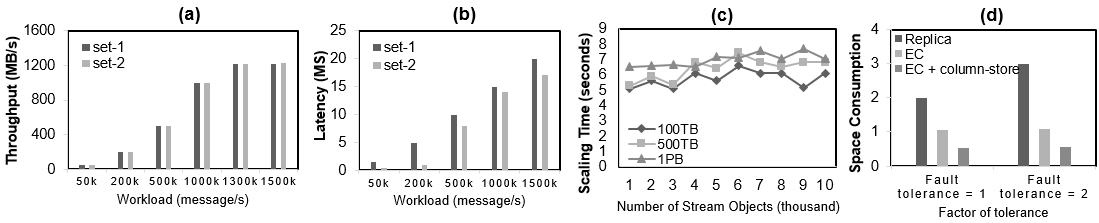
The message stream processing speed in StreamLake is similar to Kafka. StreamLake and Kafka process about 300 thousand messages per second when the workload is 10 million records. Both systems scale to process about 500 thousand messages per second when the workloads are 100 million and more.



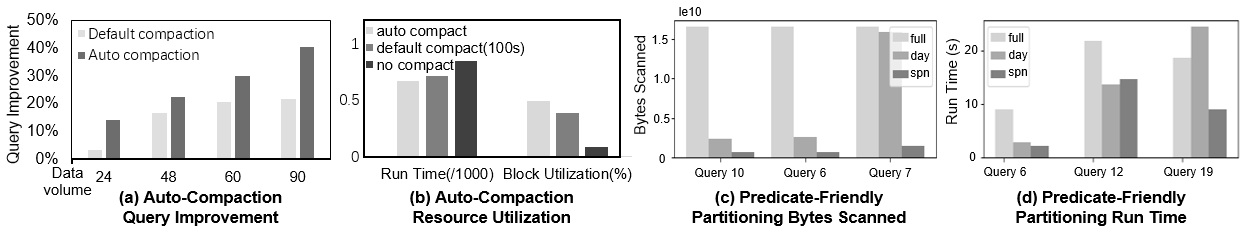
**Figure 13: Data Analytic Pipelines for a Simplified Real-World Use Case.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Data Packet # | 10,000,000 | 50,000,000 | 100,000,000 | 500,000,000 | 1,000,000,000 |
| Storage Space Usage (GB) | S | 34 | 166 | 329 | 1,659 | 3,289 |
| HK | 145 | 729 | 1451 | 6,901 | 13,816 |
| HK / S | 4.33 | 4.38 | 4.40 | 4.16 | 4.20 |
| Stream Processing Speed (Messages/Second) | S | 301,552 | 417,303 | 518,065 | 530,077 | 546,987 |
| K | 302,611 | 413,613 | 527,826 | 531,021 | 539,893 |
| K / S | 1.00 | 0.99 | 1.02 | 1.00 | 0.99 |
| Batch Processing Total Time (Second) | S | 259 | 664 | 1173 | 4868 | 9646 |
| H | 212 | 795 | 1548 | 7535 | 14771 |
| H / S | 0.82 | 1.19 | 1.32 | 1.55 | 1.53 |

**Table 1: StreamLake vs HDFS and Kafka.**



**Figure 14: Stream Engine Throughput, Latency, Scalability and Space Consumption.**



**Figure 15: LakeBrain Compaction and Partitioning Performance.**

**7.1 Message Streaming**

To quantitively measure the message streaming service as an independent stream storage, we conduct an experiment to evaluate its throughput, latency, elasticity and volume. We select OpenMessaging as our benchmark framework as it is widely used to compare messaging platforms. A cluster with three nodes is used in this experiment for its ease of reproduction. To help better understand the impact of tiered storage, two sets of hardware configurations are tested. In the first set of hardware (Set-1), each node has 10 CPU cores, 128 GB RAM and 800 GB NVMe SSD, 3 PB SAS HDD and all the nodes are connected with 10 Gb ethernet. In the second set of hardware (Set-2), all the configurations are the same except that each node has additional 16 GB persistent memory to serve as an extra cache. Messages are sent from producers to consumers in a fixed size of 1 KB. The data volume we examine are 100 TB, 500 TB and 1 PB.

Figure 14 shows the results of the experiment. As the messages to process increase from 50000 per second to 1.5 million per second, the system throughput increases linearly, reaching a peak of 1.2 GB/s with a workload of 1.3 million message per second. Set-1 and set-2 achieve the same throughputs, indicating that it does not improve the throughput to add persistent memory as a cache. However, as shown in Figure 14(b), persist memory reduced the latency as we expect, especially when the workload is 200k messages per second or less. Figure 14(c) shows the high elasticity of the stream storage. The service gracefully scales from 1000 to 10000 partitions in less than 10 seconds. The instant scalability demonstrates a significant advantage of the data centric and disaggregated storage design. Finally, Figure 14(d) compares the volumes of different storage strategies. Without scarifying the reliability, StreamLake provides the option to use erasure coding and column-store which can offer 3 to 5 times of volume compared to standard storage with 1 or more replicas.

**7.2 LakeBrain Compaction and Partitioning**

This section discusses the evaluations of LakeBrain’s two applications, auto-compaction and predicate-aware partitioning.

**Auto-Compaction:** To precisely evaluate the effectiveness of the optimization-based automatic compaction strategy, a TPC-H based test bed was set up to ingest data from the message streaming platform to the data lake storage, during which a compaction strategy was tested. We ran the experiment with 24 GB to 90 GB data and three compaction strategies are deployed: 1) no compaction, 2) a static strategy which simply compacts data files in a 30 second interval, and 3) auto-compaction, respectively. During the ingestion, multiple rounds of TPC-H queries are executed in parallel on top of the data sets to get their end to end performance. As shown in Figure 15(a), the results depict how much percentage both compaction strategies can improve the query performance over the baseline which was the one without compaction. It is observed that the auto-compaction strategy outperforms the static one in all volume tests. As the data volume increases, its advantage becomes more significant.

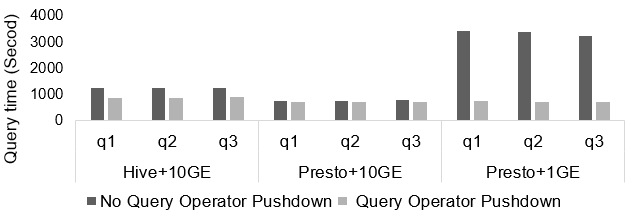
In addition to the query performance, we also evaluate the block utilization of the auto compaction. In this experiment, we control the file ingestion speed so that we can generate different number of files to measure both the run time and the block utilization in different workloads. The run time is evaluated along with the block utilization because an ideal strategy should improve the utilization without scarifying the performance. Similar to the previous experiment, we deploy three test groups: 1) no compaction, 2) a static strategy compacts data files in a fixed time interval, and 3) auto-compaction. It is revealed in the test results that the auto-compaction outperforms the static strategy in term of block utilization. When we deploy the auto-compaction, the system is able to identify good compaction opportunities in which there are many small files and both the file ingestion speed and the block utilization are relatively low. File ingestion speed is important because compaction commits will fail if there are file access conflicts. As a comparison, it is hardly to avoid unnecessary or unsuccessful compactions in the static compaction strategy hence its performance is less ideal. Figure 15(b) summarizes the results of all three test groups. Compared with no compaction and the static compaction strategy, the auto-compaction performs better in term of both block utilization and query run time.

**Predicate-Aware Partitioning:** We also tested the partitioning method on the TPC-H test bed with different scale factors. We train the probabilistic model with 3% of the data randomly sampled from the *lineitem* table in a dataset generated with a scale factor of 2. After that, we obtain the optimized partitioning policy with the proposed method and evaluate our system on the full dataset with scale factors of 2, 5, 10 and 100. To evaluate the performance, we compare the resulting bytes skipped for *lineitem* table with 1) no partition (full), 2) partitioning by the day of *l\_shipdate* (day), and 3) our proposed method using sum-product networks (spn). We compare the results with partitioning by the day of *l\_shipdate* considering it appears frequently in the pushdown predicates for *lineitem* table. The workload includes TPC-H query 6, 12 and 19 which involve *lineitem* table and include predicates other than *l\_shipdate*. We skipped the other TPC-H queries because their performance is driven mainly by multiple tables joining performance which is beyond our purpose.

The results presented in Figure 15(c,d) shows that the proposed method obtains non-marginal performance gains in terms of both bytes scanned and the runtime. The fine-grained partitioning is superior on the queries in terms of data skipping compared to partitioning by the day of *l\_shipdate* because the optimized partitioning policy split the data based on other predicates except *l\_shipdate*. Even though the runtime for the queries are dominated by table joining, the optimized partitioning also demonstrated some improvements for query 6 and query 19, considering we only optimize the partition of the *lineitem* table.

**7.3 Query Operator Computation Pushdown**

In this experiment we evaluate the query operator pushdown method which we believe can provide stable query runtime regardless of network conditions. This is significant in the real-world deployment as it is not always an option to upgrade the data center network. In fact, many customers who we have worked with did the opposite to ask us to reuse their existing network to reduce the overall upgrade costs. Hence, it is a critical design that the query operator pushdown method ensures the query processing time and the application service level agreements even with a constraint network bandwidth.

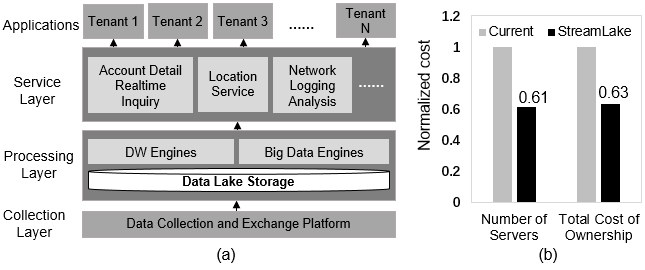


**Figure 16: Computation Pushdown Query Time.**

Two groups of clusters with different network bandwidths are used in this experiment, one with 10 Gb bandwidth and the other with 1 Gb. To precisely assess the benefits, we carefully select three live queries with data intensive operations and 4.8 TB data from a China Mobile production environment. Two different query engines, Hive and Presto, are deployed to process the SQL queries for generalization. It is observed that in the test group without query operator pushdown, the query performance varies widely across engines. When the queries are executed in the 10 Gb ethernet, Presto completes the jobs in about 900 seconds while Hive takes around 1200 seconds. When network bandwidth drops to 1 Gb, all the execution time soars to over 3000 seconds. As a comparison, we applied query operator pushdown in the second test group. The runtime of all the queries is close to 900 seconds with less than 10% difference, regardless of compute engines and network bandwidths. This means a 4 times performance advantage when the engine process queries in a 1 Gb bandwidth network. In summary, we can conclude that our generalized query operator pushdown method introduces stable and high performance to query processing in a storage-disaggregated architecture.

**7.4 China Mobile Use Case**

An application of StreamLake in China Mobile data lakes with production data demonstrates a solid optimization in term of resource utilization. As the world’s largest mobile network operator, China Mobile manages one of the largest data analytic platforms in China. Over 4.8 petabytes of fresh business operation and network logging data flows from business branches and edge devices scattered across over 30 provinces and regions to several centralized data centers. This fresh data first lands on a collection and exchange platform where the team can perform data exchanges across data centers. Then it is loaded into the analytic platform. Data warehouse and big data engines run billions of jobs and analytic models with the data to provide location services, network logging analyses and many other applications to support internal and external business and users. As its data grows to the exabyte scale, the platform starts to experience high skews of resource utilization. For instance, the utilizations of CPU, memory and storage are 26%, 41% and 66% on average based on a 14-day measurement in a data center.



**Figure 17: China Mobile a) Data Analytic Platform; b) StreamLake Resource Utilization Increases.**

We deployed StreamLake to a China Mobile data center with 20 petabytes of production data to evaluate its impacts. The existing data analytic architecture was replaced by a disaggregated-storage architecture in which its storage is powered by Huawei OceanStor Pacific with the StreamLake framework. Moderate changes are applied to connect the analytic engines to StreamLake.

The evaluation shows a significant improvement of resource utilization. The new system runs the same number of analytic jobs with 39% less servers that thanks to high utilization of data and server resources in StreamLake. Besides this, it also introduces additional benefits in term of performance and service flexibility. For instance, some batch queries can speed up 4 times when the query operator pushdown and the LakeBrain features are enabled. Another example is the message streaming. The platform originally had to maintain 300+ Kafka servers. The expansion of partitions and nodes posed a big challenge to the China Mobile IT team. With the stream storage in StreamLake, the team no longer needs to manually manage the Kafka servers. In addition, minimum data migration is required to scale the system, and maintenance costs are thus greatly reduced.

8 RELATED WORK

Several open-source projects and research work are related to StreamLake in the domain of message streaming platforms, lakehouse data management framework, query computation pushdown and automatic database tuning systems. 1) Kafka, Pulsar and Pravega [9, 11, 12] are widely used open-source event streaming platforms in the industry. Unlike StreamLake which builds the messaging service on top of the stream object and PLogs and integrates its stream storage with a lakehouse framework, these solutions are file based and have to manually connect to compute engines and external storage such like HDFS [22] or S3 [4] for downstream processing or cost friendly archiving. They increase both the complexity and costs of the data pipeline managements. 2) Iceberg, Hudi and Delta Lake [7, 8] are popular open-source lakehouse data management framework. These solutions rely on statistic file or object storage. Massive data transmission between the storage and the compute engines are inventible in many scenarios. StreamLake builds its lakehouse framework on top of the table object and PLogs, leveraging the enterprise class data redundancy, high performance cache and query computation pushdown to provide reliable and high speed concurrent lakehouse reads/writes. 3) NetApp [27] supports Hadoop to use its storage devices through NFS-based connector docking, through S3A docking to its object storage, and through SAS/iSCSI/FC building native HDFS [22] on its block/lun devices. AWS EMRFS [3] is an enhancement introduced to address the inconsistency of object storage in metadata operations, with official information showing that it has made computation pushdown related optimizations for the engine. Alibaba EMR is based on object storage, and the JindoFS [1] solves the performance problem of object storage by introducing local data caching. These solutions improve data access to the persistent storage in computation pushdown while StreamLake offers built-in computation pushdown operations directly. 4) In term of automatic database tuning systems, OtterTune [40] is a classic ML-based framework, recommending knob configuration using Gaussian process (GP). To address the limitation of traditional ML-based approaches, RL has been adopted in CDBTune [44]. Investigated in [41] shows the impact of the performance variation in production environments, indicating that GP tends to converge faster but is frequently trapped in local optima, whereas RL or deep learning (DL) generally needs a longer training process and achieves better performance. [37] is the first approach that tries to maximize data skipping for a partitioning using pushdown predicates with a bottom-up approach. QDTree [43] proposed a greedy algorithm and a reinforcement learning based algorithm to solve the data skipping maximization problem to solve the suboptimal limitation. However, these algorithms need to quantify the performance of each candidate partitioning. In addition, the partitioning layout is sub-optimal when new data comes, as it is optimized based on existing data.

9 CONCLUSIONS

We have presented the StreamLake system, an experimental data lake storage system optimized to stream and process log messages concurrently in massive scale. The system extends Huawei OceanStor Pacific’s interworking capabilities to support stream and table objects, tightly integrated with the persistent storage pools and multi-level caches to provide end-to-end intelligent data storage and management for log message streaming and processing.

ACKNOWLEDGMENTS

StreamLake is the result of efforts of many people, supported by funds of Huawei storage research and joint innovation between Huawei and China Mobile. We thank Zhiwei Guo, Sha Dai, Feng Wang, Mao Ye, Jiacheng Liu, Anwei Chen, Chenming Chen, Jianzhuang Ge, Rui Zhang, Duanshun Li, Hong Li, Ziqin Zhou, Qi Yuan, Changchen Li, Yuxiang Yu, Yang He, Shiqiu Zhao, Shuncun Zhao, Jiangbo Lu, Yafei Li, Zesheng Yang, Jingbin Cheng, Yiding Huang, Hongliang Tang, Rob Foley, Peter Puhov, Hui Lei, Xin Wu and Wei Zha for their contributions. We are grateful to Yuefeng Zhou, Xin Pang, Haiyong Xu, Meng Yang, Guangbin Meng, Sheng Chang, Lingxiang Sun, Fei Xiang and Weijie Wang for their strategic vision, leadership and supports. Jeff Naughton and Remzi Arpaci-Dusseau reviewed the paper.

REFERENCES

[1] Alibaba Cloud Jindodata. 2022. https://github.com/aliyun/alibabacloud-jindodata.

[2] Amazon Aurora. 2022. Amazon Aurora | MySQL PostgreSQL Relational Database | Amazon Web Services. https://aws.amazon.com/rds/aurora.

[3] Amazon EMR. 2022. Big Data Platform – Amazon EMR – Amazon Web Services. https://aws.amazon.com/emr.

[4] Amazon S3. 2022. Cloud Object Storage – Amazon S3 – Amazon Web Services. Retrieved Feb 28, 2022 from https://aws.amazon.com/s3.

[5] Apache Bookkeeper. 2022. Pravega – A Reliable Stream Storage System. Retrieved Feb 28, 2022 from https://cncf.pravega.io.

[6] Apache Flink. 2022. Apache Flink: Stateful Computations over Data Streams. Retrieved Feb 28, 2022 from https://flink.apache.org.

[7] Apache Hudi. 2022. https://hudi.apache.org.

[8] Apache Iceberg. 2022. https: //iceberg.apache.org.

[9] Apache Kafka. 2022. <https://kafka>. apache.org.

[10] Apache Parquet. 2022. https: //parquet.apache.org.

[11] Apache Pravega. 2022. Pravega – A Reliable Stream Storage System. Retrieved Feb 28, 2022 from <https://cncf.pravega.io/>.

[12] Apache Pulsar. 2022. https://pulsar.apache.org/.

[13] Apache Spark. 2022. Apache Spark™ - Unified Engine for large-scale data analytics. Retrieved Feb 28, 2022 from https://spark.apache.org.

[14] Robert Foley, Peter Puhov. Futurewei. Caerus NDP. 2021.

        Retrieved Feb 28, 2022 from [https://github.com/open-infrastructure-labs/caerus-dike](https://nam11.safelinks.protection.outlook.com/?url=https%3A%2F%2Fgithub.com%2Fopen-infrastructure-labs%2Fcaerus-dike&data=04%7C01%7Chongliang.tang%40futurewei.com%7C86420ecbadf44da4ed6c08d9fbb84b87%7C0fee8ff2a3b240189c753a1d5591fedc%7C1%7C0%7C637817591239608229%7CUnknown%7CTWFpbGZsb3d8eyJWIjoiMC4wLjAwMDAiLCJQIjoiV2luMzIiLCJBTiI6Ik1haWwiLCJXVCI6Mn0%3D%7C3000&sdata=zzvtqm0wS5tbyXLz6fBbFpwx%2FLlHm9E0M%2FJMhkoTCSw%3D&reserved=0" \o ).

[15] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instanceoptimized data layouts for cloud analytics workloads. In Proceedings of the 2021 International Conference on Management of Data. 418–431.

[16] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: a unified cache storage system for analytical databases. Proceedings of the VLDB Endowment 14, 11 (2021), 2432–2444.

[17] Mohamed Eltabakh, Anantha Subramanian, Awny Al-Omari, Mohammed Al Kateb, Sanjay Nair, Mahbub Hasan, Wellington Cabrera, Charles Zhang, Amit Kishore, and Snigdha Prasad. 2021. Not Black-Box Anymore! Enabling Analytics-Aware Optimizations in Teradata Vantage. Proceedings of the VLDB Endowment 14, 12 (2021), 2959–2971.

[18] Exactly-once Semantics is Possible. 2022. Exactly-once Semantics is Possible: Here’s How Apache Kafka Does it. Retrieved Feb 28, 2022 from https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/.

[19] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! Proceedings of the VLDB Endowment 13, 7, 992–1005.

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

[21] Huawei Data Storage Systems. 2022. https://e.huawei.com/en/products/storage.

[22] ASF Infrabot. 2019. Filesystem Compatibility with Apache Hadoop. Apache Software Foundation. Retrieved July 9, 2019 from https://cwiki.apache.org/confluence/display/HADOOP2/HCFS.

[23] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. 2021. Microlearner: A fine-grained Learning Optimizer for Big Data Workloads at Microsoft. In 2021 IEEE 37th International Conference on Data Engineering (ICDE). IEEE, 2423–2434.

[24] K. R. Krish, Bharti Wadhwa, M. Safdar Iqbal, M. Mustafa Rafique, and Ali R. Butt. 2016. On Efficient Hierarchical Storage for Big Data Processing. In 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). 403–408. https://doi.org/10.1109/CCGrid.2016.61.

[25] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. Proceedings of the VLDB Endowment 12, 12 (2019), 2118–2130.

[26] Alejandro Molina, Antonio Vergari, Karl Stelzner, Robert Peharz, Pranav Subramani, Nicola Di Mauro, Pascal Poupart, and Kristian Kersting. 2019. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks. arXiv preprint arXiv:1901.03704 (2019).

[27] NetApp Data Storage. 2022. Data Management Solutions for the Cloud | NetApp. Retrieved Feb 28, 2022 from https://www.netapp.com/data-storage.

[28] OceanStor Pacific Series. 2021. Introduction to DPC. Retrieved September 8, 2021 from https://support.huawei.com/enterprise/en/doc/EDOC1100194277/326d9def/introduction-to-dpc.

[29] Oracle Documentation. 2022. https://docs.oracle.com.

[30] Aditi Pandit, Derrick Kondo, David Simmen, Anjali Norwood, and Tongxin Bai. 2015. Accelerating big data analytics with collaborative planning in Teradata Aster 6. In 2015 IEEE 31st International Conference on Data Engineering. IEEE, 1304–1315.

[31] Hoifung Poon and Pedro Domingos. 2011. Sum-product networks: A new deep architecture. In 2011 IEEE International Conference on Computer VisionWorkshops(ICCV Workshops). IEEE, 689–690.

[32] Presto. 2022. https://prestodb.io/.

[33] Pure Storage. 2022. A Modern Data Experience | Data Storage Company | Pure Storage. Retrieved Feb 28, 2022 from https://www.purestorage.com.

[34] Rabbit MQ. 2022. Messaging that just works — RabbitMQ. https://www.rabbitmq.com.

[35] David Simmen, Karl Schnaitter, Jeff Davis, Yingjie He, Sangeet Lohariwala, Ajay Mysore, Vinayak Shenoi, Mingfeng Tan, and Yu Xiao. 2014. Large-scale graph analytics in aster 6: bringing context to big data discovery. Proceedings of the VLDB Endowment 7, 13 (2014), 1405–1416.

[36] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (Melbourne, Victoria, Australia) (SIGMOD ’15). Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797.

[37] Liwen Sun, Michael J Franklin, Sanjay Krishnan, and Reynold S Xin. 2014. Fine-grained partitioning for aggressive data skipping. In Proceedings of the 2014 ACM SIGMOD international conference on Management of data. 1115–1126.

[38] Xin Tang, Robert Wehrmeister, James Shau, Abhirup Chakraborty, Daley Alex, Awny Al Omari, Feven Atnafu, Jeff Davis, Litao Deng, Deepak Jaiswal, et al. 2016. SQL-SA for big data discovery polymorphic and parallelizable SQL user-defined scalar and aggregate infrastructure in Teradata Aster 6.20. In 2016 IEEE 32nd international conference on data engineering (ICDE). IEEE, 1182–1193.

[39] Teradata Documentation. 2022. https://docs.teradata.com/.

[40] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In Proceedings of the 2017 ACM international conference on management of data. 1009–1024.

[41] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An inquiry into machine learningbased automatic configuration tuning services on real-world database management systems. Proceedings of the VLDB Endowment 14, 7 (2021), 1241–1253.

[42] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. Proceedings of the VLDB Endowment 12, 3 (2018), 210–222.

[43] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-tree: Learning data layouts for big data analytics. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. 193–208.

[44] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In Proceedings of the 2019 International Conference on Management of Data. 415–432.

[45] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. Restune: Resource oriented tuning boosted by metalearning for cloud databases. In Proceedings of the 2021 International Conference on Management of Data. 2102–2114.