Separation Is for Better Reunion: Data Lake Storage at Huawei

Xin Tang, Chengliang Chai[§], Dawei Zhao, Haohai Ma, Yong Zheng, Zhenyong Fan,
Xin Wu, Jiaquan Zhang, Rui Zhang, Duanshun Li, Yi He, Keji Huang, Guangbin Meng, Yidong Wang,
Yuefeng Zhou, Tao Tao[†], Lirong Jian[γ], Jiwu Shu[‡], Yuping Wang[§], Ye Yuan[§], Guoren Wang[§], Guoliang Li[‡]
Beijing Institute of Technology[§], Huawei, China Mobile Communications[†], Hashdata[γ], Tsinghua University[‡]

***Abstract*—Huawei collaborates with some Chinese large business companies to store and process exabytes of nationwide operational data in data lake storage to provide business insights. Specifically, our customers will ask to store and process massive log message data to support their real-time and decision-making applications. Thus, we need computation and storage components in the analytic platform to process and store these data cost-efficiently.**

To meet these user requirements, we have designed a storage system in data lake, `StreamLake`, which introduces a novel design to serve log message streaming and batch data processing in distributed storage, with high scalability, efficiency, reliability and low cost. Specifically, we introduce a stream (storage) object as a storage abstraction for message streaming data to achieve the storage-disaggregated architecture with high scalability and reliability. Moreover, we utilize the erasure coding and tiered storage to save the storage cost, and furthermore, the stream object can be automatically converted to a table object such that cost-effective stream and batch data processing can be achieved. For tabular data, we implement the lakehouse functionality to support ACID via the table object, with a metadata acceleration to improve the efficiency of data access between the compute and storage engines. Also, we design a `LakeBrain` optimizer at the storage side to optimize the query performance and resource utilization under the storage-disaggregated architecture. Finally, we have also deployed `StreamLake` in China Mobile, the world's largest mobile network operator to serve over 20`PB` production data, and the results demonstrate improvements of 30% to $4\times$ in terms of query performance and over 37% in terms of cost saving.

I. Introduction

As the Internet of Things and 5G technologies become prevalent, massive amounts of data are being collected, stored, and analyzed. The traditional architecture of data infrastructure has been challenged by cloud-native designs, where compute and storage resources are pooled 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 constructions to compute-and-storage disaggregated architectures. For example, data lake storage (*e.g.*, AWS S3 [1], Huawei OceanStor Pacific [2]), with its $10\times$ better price, availability, and persistence compared to traditional storage formats, has been popular for storing massive various data, so as to support large-scale data analysis.

[§]Chengliang Chai and [§]Yuping Wang are the corresponding authors. Work done when Xin Tang, Zhenyong Fan and Xin Wu were in Huawei.

However, as large enterprises further digitalize their business, the data to be stored and analyzed explode. Over the past several years, we have collaborated closely with over 200 enterprise customers from 16 different industries to better understand their big data processing requirements. Our analysis of key statistics has revealed the following insights:
Petabytes of data. Nearly half of our customers (49%) have processed data ranging from one terabyte to 10 petabytes (PB). A significant percentage of customers (29%) handle more than 10 PB, while 8% manage more than 100 PB of data.
Log data. A large majority (81%) of our customers primarily work with log message data.
Stream and batch processing. Both stream and batch processing play a critical role in the big data processing. 69% of customers actively use batch processing, and 65% use stream processing. Nearly 40% care about both. Also, when processing data through data pipelines, in many cases, customers have to continuously update the datasets.
Data retention. In practice, 43% of customers are required to store data for 1-5 years. 22% store 5-10 years and 27% store at least 10 years, according to regulations and practices in different industries.

To satisfy the above users' requirements, we aim to design a data lake storage system to support stream and batch data processing with high efficiency, persistence, scalability and low total cost of ownership (TCO). To this end, the system has several significant aspects to be considered. (1) As users always face petabytes of log streaming data, it is challenging to store the data persistently at low cost, while keeping high elasticity and processing efficiency. For example, as streaming data needs real-time processing, some typical system like Kafka uses local file system as the storage, which lacks of elasticity because the computation and storage are tightly coupled. Also, in practice, users may conduct stream or batch processing for different applications over the same data, thus storing two copies for different processes is costly. (2) In a complex data analysis pipeline, there are likely to be multiple copies of data to support different tasks. If these copies are updated individually, data could be inconsistent. Hence, it is significant to support atomic writes to achieve high-quality data. (3) In data lake storage, it is challenging to perform optimization (like the optimizer in

databases) because the computing engine is always decoupled with the storage. Hence, it is critical to consider how to incorporate an optimizer in the storage engine to optimize system performance.

To address these issues, we deploy our StreamLake storage system with its novel design to serve enterprise-level stream and batch data processing.

First, in terms of streaming storage, we implement the compute-and-storage disaggregated architecture to achieve high scalability and reliability. To be specific, we introduce the *stream object* to read/write the streaming messages, based on which our streaming service becomes elastic, $i.e.$, the number of instances for processing the messages can be efficiently adjusted without data migration. The object has buffers to support real-time stream processing, and load-balanced storage and redundant persistence are also achieved. Stream object is a novel design that directly stores messages as stream in the underlying storage system rather than storing via files like Kafka. In this way, we can better leverage the characteristics of stream data to well store, compress, transfer, serve and backup the data. Besides, to better reduce the storage cost, StreamLake is built on the tiering storage Huawei OceanStor, which can automatically migrate data between SSD and HDD. To better achieve cost-effective stream and batch processing, the *stream object* can be automatically converted to *table object*, and vice versa. In this way, data can be maintained for just one copy rather than stored for two copies separately, and thus the storage cost is reduced. Existing systems like the widely-used Kafka [3] use the local file system to persist data, which is less elastic than disaggregated storage. Pravega [4] and Pulsar [5] adopt the disaggregated architecture, but still store messages in files, while we have a native stream object that can achieve highly efficient and reliable storage. In addition, we can automatically manage cold data based on our built-in tiering storage (or conduct table-stream conversion to save the cost), they have to migrate the data to other cost-friendly storage systems like HDFS [6] or S3 [1].

Second, in terms of supporting updates, the Lakehouse system [7]–[9] can eliminate multiple copies by achieving concurrent read and write in an ACID manner over one copy. We also implement the lakehouse functionality in StreamLake to support ACID via the table object. Particularly, we design a metadata acceleration that combines small I/O accesses, so as to improve the efficiency of compute engines visiting data in the remote disaggregated storage.

Third, we build an intelligent data lake optimizer LakeBrain at the storage-side that focuses on optimizing the data layout in the storage, so as to improve the resource utilization as well as query performance. Many recent works [10]–[13] have focused on using machine learning techniques to optimize database systems, including the knob tuner, query optimizer, etc. For the data lake with disaggregated storage, we think that it is a promising direction to design an optimizer and we conduct the following two attempts. We design a reinforcement learning based automatic compaction module to decide whether to compact small files considering the system state at a certain system status, so as to improve the block utilization while keeping the system running smoothly. We also build a predicate-aware partitioning model that is used to judiciously distribute data to storage blocks to reduce the number of tuples to be visited, so as to improve the query efficiency.

Overall, our StreamLake has the following characteristics.

High storage scalability. Leveraging the stream object and table object, StreamLake adopts the compute-and-storage disaggregated architecture that allows for elastically adjusting computing and storage resources according to dynamic workloads for both stream and batch data. In this way, our system can scale gracefully to store petabytes of new data.

High processing efficiency. The stream object in StreamLake provides efficient read/write APIs to support real-time stream processing. Also, our stream object provides load-balanced stream storage, which can also help improve the efficiency. Besides, query computation pushdown is applied to reduce the data transfer between the storage and query engine. Furthermore, the LakeBrain optimizer improves the query performance by optimizing the data layout.

High reliability. StreamLake is built on top of the Huawei OceanStor Pacific [2], which has built-in data reliability and security to provide full protection to the data.

Low TCO. Overall, StreamLake can save the users' cost a lot by leveraging our novel designs as well as the ability of our Huawei OceanStor storage. The cost mostly includes the cost of storage and computing servers. On the one hand, we use erasure coding [2] as the data redundancy strategy, which stores fewer copies of data than other systems such as HDFS (improving the disk utilization rate from 33% to 91%), and we also use built-in tiering storage and compression techniques to save storage cost. Besides, we can just store one copy to serve both stream and batch data processing, which further saves the cost. On the other hand, the LakeBrain optimizer and pushdown also save the compute resource by improving the query efficiency. Moreover, the disaggragated architecture makes the users require their compute or storage resources as needed.

Use case. We deployed StreamLake in China Mobile data lakes with production data, resulting in significant optimization of resource utilization and performance. China Mobile manages one of the largest data analytic platforms in China. Over 4.8 petabytes per day of fresh data flow from business branches and edge devices scattered across over 30 provinces to several centralized data centers. As shown in Figure 1(a), the fresh data first lands on a collection and exchange platform where data exchanges across data centers. Then it is loaded into the analytic platform. Data warehouses and big data engines run billions of jobs over the data to provide location services, network logging analysis and many other applications to serve users. As the platform grew to the exabyte scale, resource utilization became increasingly

skewed, with average CPU, memory, and storage utilization at 26%, 41%, and 66% respectively. Previously, China Mobile handled these jobs between independent Kafka and HDFS servers, which could be expensive and error-prone.

To overcome this, we deployed StreamLake in a China Mobile data center with 20 petabytes of production data, replacing the existing analytic architecture with a disaggregated-storage architecture powered by Huawei OceanStor Pacific with StreamLake framework. Figure 1(b) shows some general evaluation results of our deployment. With StreamLake, compared with the Kafka and HDFS solution, our customers can run the same number of analytical jobs with 39% fewer servers, due to the high utilization of server resources in StreamLake, and leading to 37% cost savings (TCO). Here TCO refers to the number of servers to support the jobs. Also, a number of queries can achieve performance improvement from 30% to 4×. Moreover, minimum data migration is required to scale the system, and thus maintenance cost is thus greatly reduced. More detailed evaluation is shown in Section VII.

II. Related Work

Data lake storage system. Dell EMC [14] and NetApp [15] support HDFS protocol via connectors to NFS or SAS/iSCSI/FC implementation on top of block/LUN devices [16], [17]. While these storage products provide enterprise-level scalability and high reliability to customers, their supports to analytic efficiency primarily rely on the analytical engine partners [18]. AWS [19], Azure [20], Google Cloud [21] and Alibaba Cloud [22] provide a rich portfolio of storage services to build a data lake in the cloud. These cloud storage services are loosely connected to support messaging and batch processing while StreamLake tightly integrates message streaming, lakehouse and persistent storage in a single system which is more efficient and cost-effective.

Message Streaming. Kafka, Pulsar and Pravega [3]–[5] are popular open-source streaming platforms in industry. Unlike StreamLake, which builds its messaging service with stream object and PLogs, and integrates its stream storage with a lakehouse framework, these solutions are file-based and require manual connections to compute engines and external storage, such as HDFS [6] or S3 [1] for downstream processing or cost-friendly archiving. This increases both the complexity and cost of data pipeline management.

Lakehouse. Iceberg, Hudi and Delta Lake [7]–[9] are top lakehouse data management frameworks which store data in popular file formats for analytics [23]. StreamLake builds the lakehouse framework on top of the table object storage and PLogs [2]. This integration enables us to fully leverage Huawei Storage's enterprise-class features [2] to provide high scalability and reliability. In addition, metadata acceleration and dynamic data layout optimizations facilitate concurrent lakehouse operations with improved speed and reliability.

Automatic database tuning. Recently, AI is widely-used inside the database system to improve the performance [10]–[13], [24]–[30], [30]–[34]. For instance, OtterTune [10] is a classic ML-based framework, recommending knob configuration using the Gaussian process. Moreover, RL has been adopted in CDBTune [11] to iteratively explore the optimal configuration. Sun et.al [35] is the first approach that tries to maximize data skipping for partitioning using pushdown predicates with a bottom-up approach. QD-tree [28] proposed a greedy algorithm and a reinforcement learning based algorithm to further optimize the partitioning strategy, but they need to quantify partition cardinalities by sampling or scanning, which is not accurate and efficient enough.

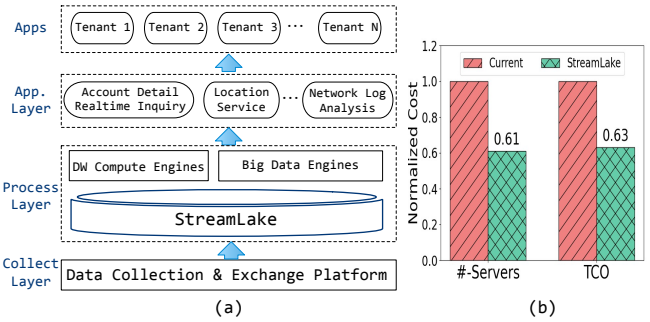

Fig. 1. China Mobile Use Case.

III. Architecture

In order to meet the demands of enterprise customers for next-generation big data solutions, StreamLake aims at optimizing the processing efficiency and resource utilization of massive log messages in big data pipelines. At a high level, StreamLake is composed of three layers: storage, data service, and data access, as depicted in Figure 2, which is the expansion of the StreamLake module in Figure 1.

This architecture comprises two tiers of routes (as detailed in the rest of this section, section IV and Section IV.A–Write stream messages): a) one at the data service layer to distribute processing capabilities across nodes and b) another at the store layer to balance storage space and facilitate rapid data duplication and reconstruction. This symmetric architecture, combined with Huawei proprietary technologies, ensures the system's ability to deliver high performance, fault-tolerance and high scalability.

Store layer is responsible for data persistence, which consists of SSD and HDD data storage pools, a high-speed data exchange and interworking bus as well as multiple types of storage semantic abstractions (including block, file, stream, table, etc.).

(1) The data storage pools comprised of SSD and HDD offer reliable management of stored data. The physical storage space on the disks in the storage cluster is divided into slices, which are then organized as logical units across disks in various servers to ensure data redundancy and load balancing. The storage pools also implement storage space features such as garbage collection, data reconstruction, snapshot, clone, write-once-read-many mechanism, thin provision, etc.

(2) The data exchange and interworking bus [2] offers high-speed data transfer and interworking of different storage abstractions.Its advanced features include support for Remote

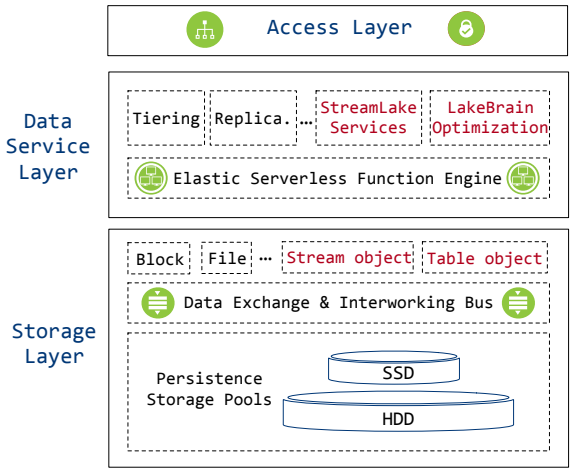

Fig. 2. StreamLake Storage Architecture.

Direct Memory Access (RDMA), which bypasses the CPU and L1 cache to accelerate data transfer speeds. Additionally, the bus leverages intelligent stripe aggregation, I/O priority scheduling, etc., to optimize data transfer and processing. All nodes are interconnected by the data bus to enable high Input/Output Operations per Second (IOPS), large bandwidth and low latency data exchanges. Furthermore, the bus supports the interworking of different storage abstractions, allowing for the sharing and access of a single data piece by different interfaces, which eliminates the need for data migration and significantly saves storage space.

(3) The storage abstractions such as block and file implement access interfaces to the underlying storage in different semantics. We introduce two new abstractions, stream object and table object, to manage messaging streams and tabular data efficiently. Their implementation will be discussed in Section IV.

Data service layer provides a rich set of features to enable efficient data management at the 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, which saves a lot of storage costs. The replication service provides periodical replications to remote sites for backup and recovery.

Particularly, to further enhance the capabilities of the layer, we have extended it to include specialized services and optimizations for log message processing operations, which include the `StreamLake` services (Section V) to support real-time streaming and lakehouse functionality, and `LakeBrain` (Section VI) to improve the resource utilization and query efficiency. The elastic serverless function engine can be regarded as a lightweight computation platform to serve the above components.

Data 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 also plays a crucial role in managing authentication and access control lists, which ensure that only valid user requests are translated into internal requests for further processing, so as to achieve the security and integrity.

```
1  int32_t CreateServerStreamObject(
2    IN  CREATE_OPTIONS_S  *option,
3    OUT object_id_t  *objectId);
4  int32_t DestroyServerStreamObject(
5    IN  object_id_t  *objectId);
6  int32_t AppendServerStreamObject(
7    IN  object_id_t  *objectId,
8    IN  IO_CONTENT_S  *io,
9    OUT uint64_t  *offset);
10 int32_t ReadServerStreamObject(
11   IN  object_id_t *objectId,
12   IN  uint64_t offset,
13   IN  EAD_CTRL_S *readCtrl,
14   INOUT  IO_CONTENT_S *io);
```

Fig. 3. Stream Object Operations.

IV. STREAM AND TABLE STORAGE OBJECT

In this section, we introduce the stream object and table object, purpose-built storage abstractions designed for efficient storage and access of stream and table data in the storage layer.

A. Stream Object

The stream object is a storage abstraction in the store layer that efficiently supports key-value message streaming at scale. It stores a partition of key-value pairs for message streams, organized as a collection of data slices. Each slice contains up to 256 records as depicted in Figure 4. Incoming message records are appended to a specific slice in a stream object based on its topic, key, and offset.

Stream objects operations. The stream object operates similarly to the block and file storage abstractions, providing read and write functionality for stream storage. Figure 3 outlines key operations supported by the stream object, including creating and destroying a stream object with functions `CreateServerStreamObject` (line 1-3) and `DestroyServerStreamObject` (line 4-5) respectively. The `*option` field (line 2) sets storage configurations, such as data redundancy methods (replicate or erasure code) and I/O quotas, so as to ensure enterprise-level reliability and performance. The assigned `objectId` (line 3) serves as a unique identifier to operate the stream object. The `AppendServerStreamObject` function appends incoming records to the stream object and returns the starting offset of the appended records. The `ReadServerStreamObject` function reads the stream object starting from a specified offset, with control conditions such as the length of the read specified in the `readCtrl` field. Since the message service is designed to support real-time streaming, it is set to respond to all subsequent messages unless specified limits by the user. `IO_CONTENT_S` (line 8 and 14) is a data structure that provides non-blocking I/O by using buffers to enhance the performance of both writing and reading operations.

Write stream messages. We discuss how to write messages into StreamLake and endure enterprise-level load-balanced and redundant persistence for the stream objects, which is achieved based on SSD and HDD storage pools. As shown in Figure 4, the messages are first assigned to stream object slices based on topics, keys, and offsets (Figure 4-a,b,c). Then, a distributed hash table is leveraged to ensure even data distribution for load-balance storage (Figure 4-d). Specifically, data slices will be distributed evenly to 4096 logical shards, each of which has the storage space managed by persistence logs (PLog, Figure 4-e). Each PLog unit is a collection of persistence services in OceanStor [2] that controls a fixed amount of storage space on multiple disks and provides 128 MB of addresses per shard. When a message is received, the PLog unit replicates it to multiple disks for redundancy (Figure 4-f). We use key-value databases to serve as indexes for PLogs for fast record lookup.

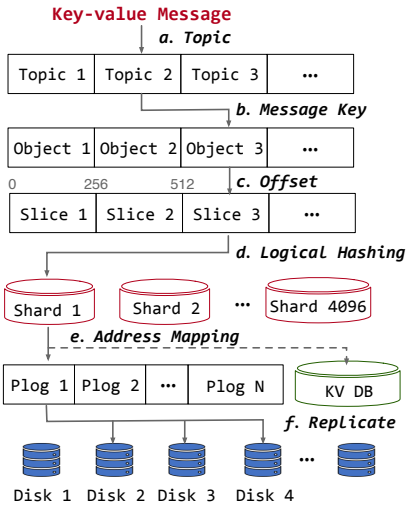

Fig. 4. Write Message to StreamLake.

B. Table Object

We also extend the storage object layer in StreamLake to support operations over tables for more effective data storage and management, like lakehouse systems [7]–[9]. The table storage uses an open lakehouse format with putting catalog in the KV store for faster metadata access. The table abstraction is logically defined by a directory of data and metadata files, as shown in Figure 5.

Data directory. Table objects are stored in Parquet files of the data directory. In this example, data objects are separated to different sub-directories by the location column. Each sub-directory name represents its partition range. Data objects in each Parquet file are organized as row-groups and stored in a columnar format for efficient analysis. Footers in the Parquet files contain statistics to support data skipping within the file.

Metadata directory keeps track of the file paths of the table, schema and transaction commits etc., which are organized into three levels: commit, snapshot, and catalog, in Figure 5-(b, c, d).

Commits are Arvo files that contain file-level metadata and statistics such as file paths, record counts, and value ranges for the data objects. Each data insert, update, and delete operation will generate a new commit file to record changes of the data object files.

Snapshots are index files that index valid commit files for a specified time period. These snapshots commit statistics such as current files, row count and added/removed files/rows as data operation logs. Along with commits, snapshots provide snapshot-level isolation to support optimistic concurrency control. Readers can access the data by reading from the valid commit files, while changes made by a writer will not be visible to readers until they are committed and recorded in a snapshot. This allows multiple readers and one writer to access the data simultaneously without locks.

Snapshots also monitor the expiration of all commits, making them essential for supporting time travel, which allows data to be viewed as it appeared at a specific time. By keeping old commits and snapshots, table objects use a timestamp to look up the corresponding snapshot and commit to access historical data.

Catalog describes the table object, including the profile data such as the table ID, directory paths, schema, snapshot descriptions, modification timestamps, etc. The data and metadata files are stored in the table directory, except for the catalog stored in a distributed key-value engine optimized for RDMA and Storage Class Memory (SCM) to ensure fast metadata access. The data and metadata files are converted to PLogs in the storage for redundant persistence as discussed above.

Fig. 5. File Organization of StreamLake Table Objects.

V. StreamLake Data Processing

In this Section, we present the data processing services in the data layer. Driven by practical application scenarios discussed in Section I, these services provide a comprehensive, enterprise-level data lake storage solution to efficiently store and process log messages at scale. The StreamLake services encompass a stream storage system for message streaming (Section V-A) and lakehouse-format read/write capabilities for efficient tabular data processing (Section V-B).

A. Message Streaming

We design a stream storage engine to facilitate message streaming at large scale, which leverages stream objects to ensure enterprise-level scalability via the disaggregated storage architecture.

Overall architecture of streaming service. The high-level design of the stream service is in Figure 6, which comprises producers, consumers, stream workers, stream objects, and a stream dispatcher. They work together to provide seamless message streaming. The stream objects are located in the store layer, while the stream workers and dispatchers are in the data services layer of `StreamLake`.

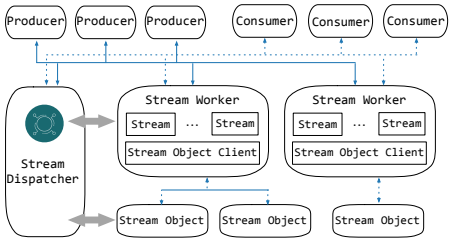

Fig. 6. Write Message to `StreamLake`.

Producers and Consumers. Producers are responsible for publishing messages to topics, which are named resources to categorize streaming messages. Consumers, located downstream, subscribe to these topics to receive and process the published messages. To ensure seamless integration with existing open-source message streaming services used by our customers in production environments, the producer and consumer message APIs are designed to be compatible with the open-source de facto standard. This maximizes connectivity with the ecosystem, allowing users to easily migrate their applications to `StreamLake` with minimum costs. Figure 7 demonstrates the process of writing and reading messages using the producer and consumer APIs. In this example, a producer writes a new message "Hello World" as a key-value pair to a topic named "`topic_streamlake_test`". The consumer then subscribes to this topic and processes published messages.

```
/*Sample producer code*/
Producer producer = new Producer();
Message msg = new Message("Hello world");
producer.send("topic_streamlake_test", msg);
 /*Sample consumer code*/
Consumer consumer = new Consumer();
consumer.subscribe("topic_streamlake_test");
While (true) {
 /*Poll for new data*/ }
```

Fig. 7. Sample code of Producer and Consumer.

Stream workers work together with stream objects discussed in Section IV-A to tackle stream processing and message storage. The number of stream workers is determined by the configurations and the physical resources allocated to stream storage. Each stream worker is capable of handling multiple streams and a single stream object client. When a topic is created, streams are added to the stream workers in a round-robin manner to ensure even distribution and workload balancing across the cluster.

Each stream is mapped to a unique stream object in the storage layer, which is a storage abstraction customized to key-value message streaming. The stream object offers efficient interfaces and implementations for writing and reading streams from the storage pools. The persistence process is detailed in Figure 4.

Message delivery is carried out by stream object clients, which monitor the stream objects. These clients 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, clients actively monitor the health of the stream objects to which they are connected and regularly exchange critical service data with the dispatcher service. This synchronization process includes reporting the health of the stream object connections and refreshing the stream objects connected to by the client.

Stream dispatcher. The stream dispatcher is responsible for managing the metadata and configurations of the messaging service, and directing external/internal requests to the appropriate resources for message dispatch. The relationships among topics, streams, stream workers, and stream objects are stored as key-value pairs in a fault-tolerant key-value store within the stream dispatcher. When there is a status change ($e.g.$, a stream worker or topic is added or removed), the metadata in the key-value store is updated immediately to refresh the topology tracking, which aids the stream dispatcher in directing requests for message stream dispatch. When there is a producer or consumer connection request, the stream dispatcher will route the request to the appropriate stream worker based on the associated stream topic, establishing a direct message exchange channel between the producer, the stream worker, and the consumer.

```
{ "stream_num" : 3,
  "quota" : 10^6,
  "scm_cache" : true,
  "convert_2_table" :{
      "table_schema" : { … },
      "table_path" : …,
      "split_offset" : 10^7,
      "split_time" : 36000,
      "delete_msg" : false,
      "enabled" : true }
  "archive" : {
      "external_archive_url" : null,
      "archive_size" : 262144,
      "row_2_col" : true,
      "enabled" : true }}
```

Fig. 8. Stream Storage Configuration Example..

The stream dispatcher also sets configurations for the messaging service in the unit of the topic. An example is shown in Figure 8.

- *The `stream_num` configuration* sets the parallelism of a topic, which should be provided during topic declaration. In the example, three streams are created for the topic and they are evenly distributed among stream workers to process messages in parallel.
- *The `quota` configuration* sets the maximum processing rate for each stream. In the example, each stream can process up to 10^6 messages per second.
- *The `scm_cache` configuration* enables the use of storage class memory (SCM) caches.
- *The `convert_2_table` configuration* enables the automatic conversion of stream object messages to table

object records, and it can also be converted back. When it is set, a background process will apply the `table_schema` to convert messages to table object records periodically and save them in `table_path`, *i.e.*, the table object directory. The conversion is triggered by either an accumulation of 10^7 messages or the passing of 36000 seconds. The advantage of this configuration will be illustrated in Section V-B.

• *The `archive` configuration* automates the archiving of historical data to meet business and regulatory requirements. Data can be stored in the cost-effective `StreamLake` archive storage pool or automatically exported to an external storage system specified in the *`external_archive_url` configuration*. The *`archive_size` configuration* denotes the data volume in MB that triggers archiving, and the *`row_2_col` configuration* determines whether the data is archived in a columnar format.

Overall, the `StreamLake` stream storage provides guaranteed delivery, efficient transfer, and high elasticity for enterprise use.

Delivery Guarantee: Our system ensures consistent message delivery through several measures. (1) Data within a stream object is strictly ordered, ensuring that messages are consumed in the order in which they are received. (2) Message writing is idempotent, which means that for network failure, duplicate messages sent by the producer can be identified. (3) Strong data consistency is achieved by eliminating unreliable components like file systems and page caches, and storing data in stream objects that can tolerate node, network, and disk failures. (4) The system provides exactly-once semantics through a transaction manager and the two-phase commit protocol. This tracks participant actions and ensures that all results in a transaction are visible or invisible at the same time.

Efficient Transfer: Our system implements several mechanisms to efficiently transfer data. First, Stream workers and stream objects are connected through a data bus with RDMA, which reduces the switching overhead in the TCP/IP protocol stack. Second, an I/O aggregation mechanism is used to aggregate small I/O requests and increase throughput. This function can be disabled for latency-sensitive scenarios. Finally, a local cache is implemented at the stream object client to speed up message consumption.

High Scalability: Our system provides high elasticity by decoupling data storage and data serving to achieve high scalability. The number of stream workers can be adjusted without data migration, and the mapping between stream workers and stream objects can be updated to reflect the changes in a matter of seconds. This allows the message streaming service to easily scale up or down to accommodate changes in service demand.

B. Lakehouse Operations

`StreamLake` also provides support for concurrent reading and writing of tabular data, similar to the architecture of the lakehouse [7]–[9]. Besides directly inserting tabular data, we can also get it from the conversion of streaming data. In this section, we first describe the storage conversion from stream messages to tabular records, and then the implementation of key lakehouse operations.

Stream-to-table conversion. This process is performed by a background service and results in the conversion of records from stream objects to table objects, allowing efficient downstream processing, which is triggered by the `convert_2_table` configuration in Figure 8, which includes the table schema and the time for data freshness in downstream processing. The table schema must be specified in the topic declaration, as it determines the expectations for field types and values across all messages. To effectively leverage the storage, users can choose to keep messages in crucial topics as stream objects to support real-time applications while converting most stream data to table objects. The reverse conversion, from table records to stream messages, is also supported for data playback. This conversion helps to reduce the storage cost because we can just store one copy to achieve both stream and batch processing. Also, this design can reduce unnecessary data movement between storage and compute clusters for data conversion.

For tabular data processing, our `StreamLake` services implement lakehouse read/write operations using a table object and high-performance caches to accelerate concurrent data reads and writes. In the rest of this subsection, we will introduce the implementation of key read/write operations in detail.

`CREATE TABLE`: This operation begins by registering the table information, such as the schema, path, database, and table name, in the catalog. The `/data` and `/metadata` directories are then created under the table path. Then table configurations (schema, partition specification, target file size, etc.) are written to the metadata directory for persistence.

`INSERT`: This operation includes the persistence of data and metadata, as well as caching of metadata, which is introduced to combine small I/O accesses to the underlying storage pools.

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

(b) Metadata caching: Metadata updates are mostly small I/O operations. To avoid generating a significant number of small files, we leverage a write cache to aggregate the metadata updates, which is achieved through the following steps: (b-1) Each added parquet file generates a commit record containing file-level metadata and descriptions. All new commit records are written to the write cache as key-value pairs when a commit is made. (b-2) The latest snapshot will be read from the persistence layer to the cache and its commit data will be updated. (b-3) The snapshot descriptions and version history in the catalog are also read from the persistence layer and overwritten by adding the latest snapshot description.

(c) Metadata persistence: Metadata in the write cache

is asynchronously flushed to the persistent storage pool when the buffer is full. A metadata management process (`MetaFresher`) transforms the commits and snapshots from key-value pairs to files and writes them to the `table/metadata` directory.

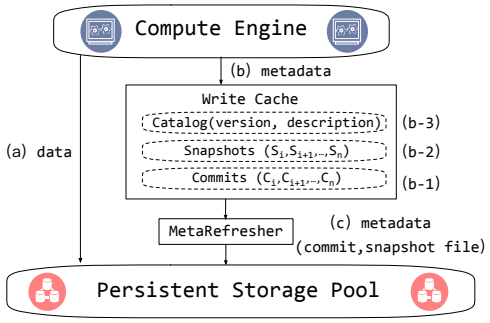

Fig. 9. Metadata Acceleration in Lakehouse Read/Write.

`SELECT:` 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 cache and the persistent storage pool to generate the latest complete snapshots and commit metadata. When all the record file addresses are confirmed, data is read from the persistence pool by read tasks.

`DELETE:` The delete operation begins with a select operation to find files containing records that match the filtering conditions. There are two cases to consider: If the filtering conditions match all data in several partitions, only the metadata will be updated, and a new commit version will be generated by eliminating the information of deleted partitions. If the filtering conditions only match some files, these files will be read, and the data matching the filtering condition will be deleted. Computation pushdowns are applied to process file reading and writing to reduce data transmission to/from the compute engines (we do not illustrate this in detail due to the space limitation).

`UPDATE:` Similar to the delete operation, the update operation also uses a select statement to identify records that match the specified conditions. Optimizations, such as pushdowns, are applied to reduce data movements during the file read and write processes.

`Drop Table:` There are two types of drop table operations: (1) Drop table soft unregisters the table from the catalog but retains the table's metadata and data in the persistent layer for potential future restoration. To restore a soft-deleted table, a new table can be created and linked to the original table path, effectively registering the deleted table back to the catalog. (2) Drop table hard removes both the metadata (files under `/metadata`) and data (files under `/data`) of the table and clears the table from the catalog. Note that some of the metadata may have been written to the acceleration cache during the drop table hard operation and will be flushed to the persistent layer asynchronously in the background. The operation to delete the metadata will first clear it from the cache, and then delete it from the disk.

VI. LakeBrain Optimization

Optimizing query processing over large-scale data is significant in data warehouse and big data systems, as discussed in [24], [36]–[41]. However, designing an optimizer like in a database is challenging in `StreamLake` because of the complicated compute-and-storage disaggregated architecture. Moreover, there exist a large number of tunable and interdependent variables, making it hard to optimize because of the large search space [42].

To address this, we present `LakeBrain`, a novel data lake storage optimizer that aims to optimize the data layout at storage-side, so as to improve resource utilization and query performance. Unlike query engine optimizers that focus on join order and cardinality estimation [12], [13], [43], data layout is key to improve both query performance and storage resource utilization in a storage-disaggregated design. Here we mainly focus on two cases, $i.e.$, automatic compacting small files and judiciously partitioning tables to improve resource utilization and performance.

A. Automatic Compaction

In a streaming application, data ingestion and transactions often result in numerous small files, leading to low query performance on merge-on-read tables. A typical method is to compact files statically using rule-based methods such as setting a time window or a data size threshold [7], [8]. In this part, `LakeBrain` designs the automatic compaction to combine these small files into fewer and larger ones, so as to improve the block utilization as well as query performance. The block utilization at a certain state t is defined by $\frac{\sum_{i=1}^{n} f_t^i}{K \times \sum_{i=1}^{n} \lceil \frac{f_t^i}{K} \rceil}$, where n_t denotes the number of files at the state, f_t^i denotes the size of each file and K is the block size. As streaming data is continuously ingested, we are likely to frequently determine whether to merge small files in each partition.

However, considering a certain state in the system, we cannot simply compact files when the block utilization is low because both compaction and data ingestion require commits, which may have conflicts, leading to compaction failure. On top of that, compaction consumes a relatively large amount of computing resources. Moreover, there exist a number of parameters, $e.g.$, file ingestion speed, target file size (the maximum size of a file after compaction), number of small files, number of concurrent queries, etc., that influence the system. The action ($i.e.$, whether to merge files) at each state will change some parameters, but it does not purely influence the current system situation, but also future states. Hence, compaction aims to achieve long-term rewards, $i.e.$, co-optimizing the query performance and storage utilization at the end. Therefore, we propose a reinforcement learning framework that can well capture the relationship between the system parameters of each state and the long-term benefit

(considering the future system states) of conducting the compaction or not for each table partition. To be specific, we show the details in Figure 10.

Agent can be taken as our automatic compaction model that receives the `reward` (resource utilization) and `state` (system/partition parameters) from the `environment` (the storage system). Then it updates the policy network (*e.g.*, Deep Q-Network [44], [45]) to guide whether to conduct the compaction operation for each partition so as to maximize the long-term reward.

State denotes the current state of the storage system, described by a number of features (parameters) as discussed above. The features can be categorized into two sets *i.e.*, one for the entire storage system and the other for individual partitions. The former one includes global features like target file size, ingestion speed, query patterns, global block utilization, etc. The latter one includes partition features like data access frequency, data access ordering, block utilization of the partition. The two features will be concatenated as the input of the policy network.

Reward reflects whether the compaction has a positive or negative effect. Specifically, if the compaction succeeds, the reward is computed by the improvement of the block utilization of the partition. If it fails, the reward is the minus of (1 - the expected improvement of the block utilization). The negative reward indicates that if the compaction tends to fail estimated by the policy network at current state, and the expected block utilization improvement is small, we tend to not conduct the compaction.

Action denotes whether we compact for each partition at each state, which is the output of the policy network. If we decide to compact, we will use the binpack strategy [7] to efficiently merge small files to the target file size.

Overall, the training process is that given each state in the system, when acting the compaction or files are keeping ingested, the state will change and we can observe the reward provided by the environment. We will store these experiences (previous states, action and reward triples) and allow the agent to reuse them to train the policy network of the agent. This process repeats until the model converges. For inference, as the streaming data comes continuously, we can trigger the trained RL model every few moments to determine whether to compact the files.

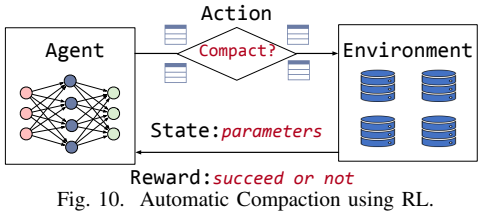

Fig. 10. Automatic Compaction using RL.

B. Predicate-aware Partitioning

With data increasing, we have to partition the data into different storage blocks such that the query efficiency can be much improved. In practice, users always select a single (or multiple) column as the partition key, apply a hash function to the values of the key, and then distribute the data to different blocks based on the partition values. This method is sub-optimal *w.r.t.* the latency because it may lead to imbalanced data distribution. `LakeBrain` designs predicate-aware method to partition the data in a fine-grained way such that given a query, the number of tuples to be assessed is minimized, and thus the efficiency is improved.

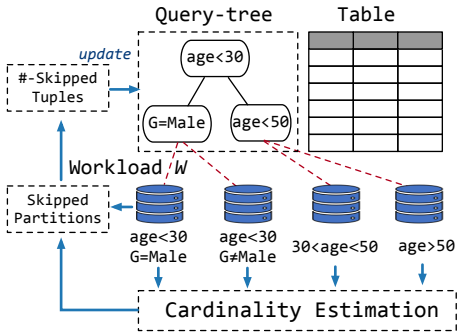

Fig. 11. Predicate-aware Partitioning.

Specifically, our partition method is based on the query-tree framework [28], and additionally leverage the machine learning based cardinality estimation method to optimize the query tree, so as to find a fine-grained data partition with high query efficiency. As shown in Figure 11, given a table T and a query workload W consisting of the pushdown predicates, we will build a query tree, similar to a decision tree where each inner node denotes a predicate in the form of (attribute, operator, literal), where operator includes $\{\leq, \geq, <, >, =, IN\}$. Each leaf node refers to a partition such that when executing W, we can skip as many tuples as possible. For example, the leftmost partition contains tuples satisfying `age` < 30 `and G=Male`. Given W and the partitions, we can compute how many partitions that we can skip. But in order to compute the number of skipped tuples, we have to know the cardinality of each partition. We can either directly compute the cardinality, or sample for estimation, which is time-consuming or not accurate enough. Hence, we can use AI-driven cardinality estimation methods [12], [13], [25], [26] to estimate the cardinality accurately and efficiently via learning the data distribution. In practice, we use the sum-product network [12] as the estimator.

VII. Experiment

A. Experimental Settings

Our Experimental Scenario. We employ a real-world use case simplified from the case in Section I. A mobile financial company collaborates with China Mobile to collect and analyze its app usage data. The company aims to understand its usage patterns to prevent frauds and enhance its product experience. China Mobile provides this analytic service through an end-to-end big data processing pipeline, which includes several jobs such as data collection, normalization, labeling, and querying, as depicted in Figure 12. We compare

StreamLake framework with their current solution to build a pipeline that can facilitate business analysis.

(a) Collection: The network carrier collects mobile app data packets in data centers and transfers them to a centralized storage pool.

(b) Normalization: In the storage pool, data packets are unified into records with validated accuracy and quality, while sensitive information is shielded for privacy protection.

(c) Labeling: Labels from knowledge bases are added, so as to classify the records and identify useful insights.

(d) Query: Following normalization and labeling, records are inserted into tables for query engine access. The app company utilizes secure API calls for data queries. Figure 13 demonstrates an SQL query example, counting daily active users (DAU) across provinces.

The network carrier establishes two data flows: one handles batch processing of full data every two hours, while the other continuously processes stream messages for time-sensitive logs, such as new logins and payments. This setup enables comprehensive analysis of both historical data and real-time events for accurate decision-making.

Settings. This use case is evaluated in a cluster using different sizes of input data packets and the results are compared with open-source storage solution Hadoop Distributed File System (HDFS) [6] and Kafka [3]. The reason of why we choose the two storage systems is that in reality, China Mobile has been using them for many years, which have shown stable and good performance. Hence, it is reasonable to directly compare with the systems that our customer (China Mobile) is using. Also, in practice, many of our customers also use HDFS and Kafka to cope with similar application scenarios.

To be specific, the cluster consists of 3 nodes, each with 24 2.30 GHz cores and 256 GB RAM. The cluster is configured as a 3-node StreamLake when we measure it. While running the open-source solution, it is configured to host a 3-node HDFS storage and a 3-node Kafka cluster simultaneously. Spark is employed as the compute engines for data processing because of the extensive usage across customer environments. The number of input data packets varies: 10 million, 50 million, 100 million, 500 million, and 1 billion packets. Each packet has an average size of 1.2 KB, resulting in corresponding data volumes of 12 GB, 60 GB, 120 GB, 600 GB, and 1.2 TB, respectively.

Overall, Figure 12 shows the data processing process. Kafka and HDFS serve as independent stream storage and batch storage respectively to pass data across collection, normalization, labeling and query jobs. As a typical ETL practice, a new copy of all data is written to HDFS and Kafka after each job. In case failing accidentally, a job can read its input data to reproduce the results.

In our solution, StreamLake serves as a unified stream and batch processing storage. We replace Kafka and HDFS with StreamLake, which handles the message streaming and data storage. Only minimal changes are made to the compute engines, so the business logic remains unchanged. As StreamLake supports time travel, only updated rows are written to the storage. When a job needs to re-run, it can use time travel to retrieve its input data. During the query jobs, for example, the three filters in the WHERE clause and the COUNT aggregate in Figure 13 are pushed down to compute in StreamLake, so as to accelerate the query.

B. Overall Comparison

Table 1 shows the results. 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. The "Ratio" represents that the ratio between HDFS (Kafka) and StreamLake with respect to the storage usage or time. Note that HK denotes the sum of the storage usage in HDFS and Kafka.

The experiment demonstrates that StreamLake significantly improves the total storage cost and the batch processing time. The storage cost in the HDFS and Kafka is 4 times as much as StreamLake. The reason is that in HDFS and Kafka, full data is written into the storage when each ETL job is finished, which is a common practice to support downstream jobs restart after unexpected failures. As a result, six copies of full data are written into the storage. While for our StreamLake, we save 75% storage cost by saving one copy of full data plus updates in each ETL job via the stream-to-table conversion and lakehouse functionality.

The batch processing speed in StreamLake is better than HDFS when the workload is 50 million records or more. As the workload grows, StreamLake is 50% faster than HDFS when the workloads are 500 million and 1 billion records because we use the LakeBrain optimizer and metadata acceleration to improve the efficiency. On the other hand, StreamLake may not be the best choice for small workloads. When the workload is 10 million records, StreamLake is 20% slower than HDFS as it performs extra metadata management. The message stream processing speed in StreamLake is competitive 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.

C. Evaluation of 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 [46] as our benchmark because it is widely used to compare messaging platforms. A cluster with three nodes is used in this experiment for ease of reproduction. To help better understand the impact of tiering 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

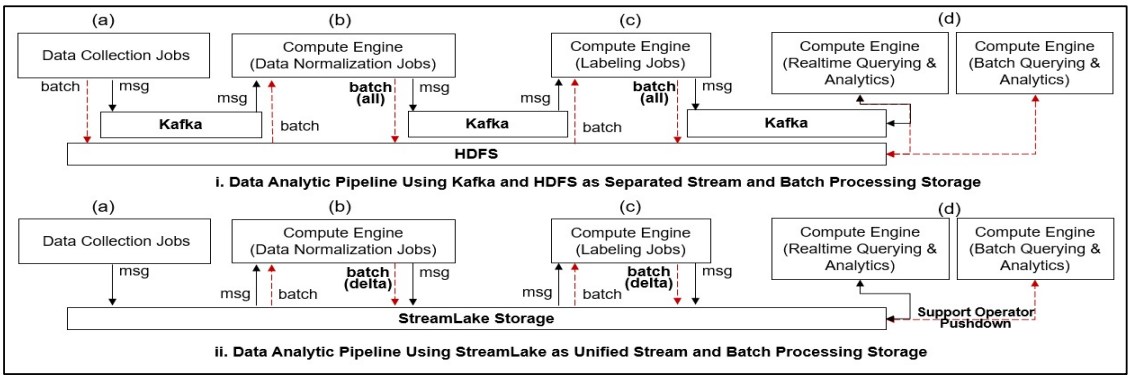

Fig. 12. Data Analytic Pipelines for a Real-world Use Case.

	#-Data Packet	10,000,000	50,000,000	100,000,000	500,000,000	1,000,000,000
Storage Space Usage (GB)	StreamLake	34	166	329	1,659	3,289
	HDFS + Kafka	145	729	1451	6,901	13,816
	Ratio (HK/S)	4.33	4.38	4.40	4.16	4.20
Stream Processing Speed (Messages/Second)	StreamLake	301,522	417,303	518,065	530,077	546,987
	Kafka	302,611	413,613	527,826	531,021	539,893
	Ratio (K/S)	1.00	0.99	1.02	1.00	0.99
Batch Processing Total Time (Second)	StreamLake	259	664	1173	4868	9646
	HDFS	212	795	1548	7535	14771
	Ratio (H/S)	0.82	1.19	1.32	1.55	1.53

TABLE 1
STREAMLAKE V.S. HDFS AND KAFKA.

```
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;
```

Fig. 13. Query Example of Computing DAU.

sent from producers to consumers in a fixed size of 1 KB. The data volumes are 100 TB, 500 TB and 1 PB respectively.

Figure 14 shows the results in terms of latency, throughput, scaling time, and space consumption. As shown in Figure 14(a), persistent memory reduces the latency as we expect, especially when the workload is 200k messages per second or less. When it comes to the throughput (Figure 14(b)), as the messages to process increase from 50000 per second to 1.5 million per second, the system throughput increases linearly. `Set-1` and `Set-2` achieve almost the same throughputs, indicating that it does not improve the throughput to add persistent memory as a cache. 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 good scalability demonstrates a significant advantage of the disaggregated storage architecture. Finally, Figure 14(d) compares the space consumption different storage strategies (`Replication` refers to saving data in its original format using multiple copies, `EC` refers to using erasure coding to store the data, `EC+Col-store` refers to first converting the data to columnar format and then applying erasure coding). The X-axis, *e.g.*, `Fault Tolerance(FT)`=1 means that a storage cluster with the redundancy strategies can tolerate one node failure, and no data is lost. The Y-axis means the times of its original data size using these redundancy strategies. Without scarifying the reliability, `StreamLake` provides the options (`EC` and `EC+Col-store`) to use erasure coding and column-store which can save three to five times of storage cost compared to `Replication`.

D. Metadata Acceleration in Lakehouse

We assess Lakehouse metadata acceleration by comparing it with file-based catalog systems, focusing on how different metadata structures affect metadata operations and query execution. We execute 100 real queries, akin to those in Fig.13, using `Where` clause conditions to utilize metadata for data filtering. Two scenarios are examined.

We first use data of real production environment which partitions the data in the unit of hours. i.e., files generated in each hour are put into the same file. The number of files (the X-axis in Figure 15(a)) we use in this experiment is generally as follows: 489,000 files/960 partitions (40 hours), 865,000 files/1920 partitions (80 hours), 2,1204,000 files/3840 partitions (160 hours), 3,947,000 files/7680 partitions (320 hours), 4,409,000 files/9600 partitions (400 hours), respectively. The Y-axis denotes the metadata operation time. We can observe that as the number of partitions increases, the latency of the method without metadata acceleration increases linearly, while our method with acceleration increases moderately. When the partition number increases 10 $\times$, the difference becomes significant. The reason is that, when we use key-value cache to accelerate the metadata, the lookup cost is constant instead of linear to the partition number.

Second, we apply different sizes of allocated memory on the compute side to observe the relation between memory size and query time. In Figure 15(b), the X-axis denotes the allocated memory and the Y-axis denotes the query time. It is observed that the query performs faster and more stable when the metadata acceleration is applied. For example, when the memory is 1GB, the method without acceleration runs out of memory (OOM). Our solution is more efficient and stable because the metadata acceleration partially complements the allocated memory for the compute engine.

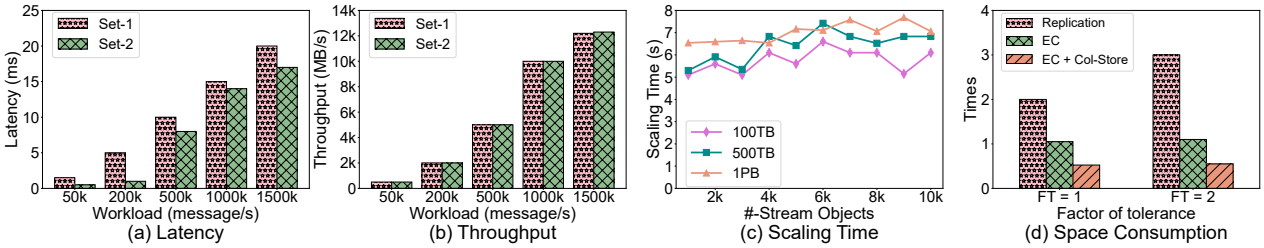

Fig. 14. Evaluation of Message Streaming.

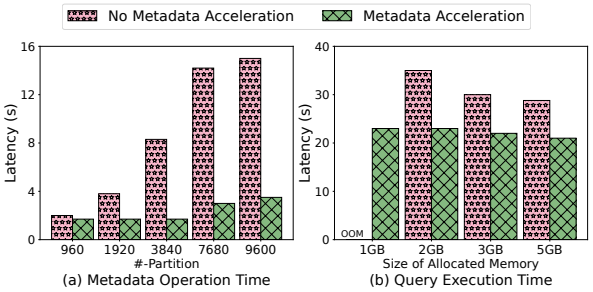

Fig. 15. Evaluation of Metadata Acceleration

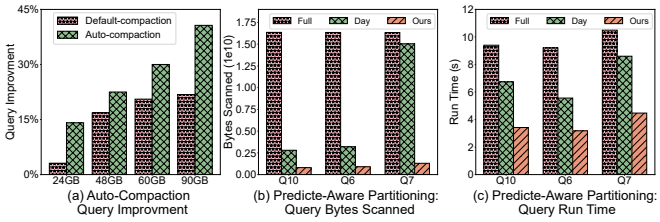

Fig. 16. `LakeBrain` Compaction and Partition Performance.

E. Evaluation of `LakeBrain`

Auto-Compaction: To precisely evaluate the effectiveness of our automatic compaction strategy, a `TPC-H` based test bed is set up to ingest data from the message streaming platform to the data lake storage, during which a compaction strategy is tested. We run the experiment with 24 GB to 90 GB data and compare our `Auto-compaction` in `StreamLake` with `Default-compaction strategy`, *i.e.*, a static strategy which simply compacts data files in a 30-second interval. We follow the method in [47] to randomly generate 5,000 queries based on the schema of `TPC-H`, and multiple rounds of there queries are executed in parallel to obtain their end-to-end performance, serving as the train data. The training time takes 3.5 hours. As shown in Figure 16(a), the results depict how much improvement of query performance that the compaction strategies can make, compared with the baseline. We can observe that the auto-compaction strategy outperforms the static one for all data volumes. As the data volume increases, the advantage becomes more significant because the number of blocks to be visited is reduced.

We evaluate auto-compaction block utilization by varying file ingestion speed to generate different file numbers. Auto-compaction consistently outperforms default-compaction, achieving approximately 50% higher block utilization on average during system operation. The system employs reinforcement learning to identify optimal compaction opportunities, prioritizing scenarios with numerous small files and low file ingestion speed and block utilization.

Predicate-Aware Partitioning: We test the partitioning method on `TPC-H` with different scale factors. Initially, we train a probabilistic model on 3% randomly sampled data from the `lineitem` table in a dataset of scale factor 2, requiring 1.5 hours for training. Subsequently, we optimize the partitioning policy with our proposed method and evaluate it across scale factors of 2, 5, 10, and 100. To gauge performance, we compare bytes skipped for the `lineitem` table under different partitioning strategies: (1) No partition (`Full`), (2) Partition by the day of `l_shipdate` (`Day`), and (3) Our predicate-aware partitioning method (`Ours`). Notably, our approach outperforms partitioning by the day of `l_shipdate`, particularly evident in finer data skipping and query runtime improvements, as depicted in Figure 16(b,c).

VIII. Conclusion

We develop `StreamLake`, combining stream and batch data processing with high elasticity, reliability, scalability, and efficiency through a disaggregated architecture. This system incorporates lakehouse functionality to ensure ACID compliance for tabular data and deploys `LakeBrain` for query and resource optimization.

Acknowledgment

This paper is supported by the NSFC(62102215, U23B2019,61925205, 62232009), CCF-Huawei Populus Grove Fund (CCF-HuaweiDB202306), National Key R&D Program of China (2023YFB4503600). Yuping Wang is supported by the NSFC (U23A20297). Ye Yuan is supported by the National Key R&D Program of China(2022YFB2702100), the NSFC (61932004, 62225203, U21A20516) and the DITDP (JCKY2021211B017). Guoren Wang is supported by the NSFC (U2001211). StreamLake is the result of the efforts of many people, supported by funds of Huawei storage research and joint innovation between Huawei and China Mobile. We thank Zhuo Chen, Zhiwei Guo, Sha Dai, Hong Li, Ziqin Zhou, Qi Yuan, Changchen Li, Xiaomin Xia, Chao Ma, Yang He, Shiqiu Zhao, Feng Wang, Jiacheng Liu, Anwei Chen, Chunming Chen, Jianzhuang Ge, Mao Ye, Shuncun Zhao, Jiangbo Lu, Yafei Li, Jingbin Cheng, Zesheng Yang, Zhiwei Chen, Liming Xie, Xuesong Wang, Hongliang Tang, Robert Foley, Peter Puhov, Hui Lei, Meng Guo, Banghong Liu, Hao Pan, and Wei Zha for their contributions. We are grateful to Haiyong Xu, Meng Yang, Jibin Wang, Xin Pang, Sheng Chang, Lingxiang Sun, Fei Xiang, Weijie Wang, and Weifeng Fang for their strategic vision and supports. Jeff Naughton and Remzi Arpaci-Dusseau reviewed the paper.

REFERENCES

[1] "Cloud object storage - amazon s3 - amazon web services." https://aws.amazon.com/s3, Amazon S3.

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

[3] https://kafka. apache.org, Apache Kafka.

[4] "Pravega - a reliable stream storage system." https://cncf.pravega.io/, Pravega.

[5] https://pulsar.apache.org/, Pulsa.

[6] "Filesystem compatibility with apache hadoop. apache software foundation," https://cwiki.apache.org/confluence/display/HADOOP2/HCFS, ASF Infrabot. 2019.

[7] https: //iceberg.apache.org, Apache Iceberg.

[8] https://hudi.apache.org, Apache Hudi.

[9] M. Armbrust, T. Das, S. Paranjpye, R. Xin, S. Zhu, A. Ghodsi, B. Yavuz, M. Murthy, J. Torres, L. Sun, P. A. Boncz, M. Mokhtar, H. V. Hovell, A. Ionescu, A. Luszczak, M. Switakowski, T. Ueshin, X. Li, M. Szafranski, P. Senster, and M. Zaharia, "Delta lake: High-performance ACID table storage over cloud object stores," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3411–3424, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3411-armbrust.pdf

[10] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 2017, pp. 1009–1024. [Online]. Available: https://doi.org/10.1145/3035918.3064029

[11] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 2019, pp. 415–432. [Online]. Available: https://doi.org/10.1145/3299869.3300085

[12] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "Deepdb: Learn from data, not from queries!" *VLDB*, vol. 13, no. 7, pp. 992–1005, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf

[13] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, P. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica, "Deep unsupervised cardinality estimation," *VLDB*, vol. 13, no. 3, pp. 279–292, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol13/p279-yang.pdf

[14] https://www.dell.com/en-uk/dt/big-data/data-lake/index.htm, Dell EMC.

[15] "Data management solutions for the cloud — netapp," https://www.netapp.com/data-storage, NetApp Data Storage.

[16] https://www.netapp.com/media/19868-wp-7217.pdf, NetApp.

[17] https://blog.netapp.com/optimize-data-management-and-analytics-with-netapp-solutions-for-hadoop, NetApp Hadoop.

[18] https://www.netapp.com/artificial-intelligence/big-data-analytics/, NetApp Bigdata.

[19] https://aws.amazon.com/big-data/datalakes-and-analytics/datalakes, AWS.

[20] https://azure.microsoft.com/en-us/solutions/data-lake, Azure.

[21] https://cloud.google.com/solutions/data-lake , Google Cloud.

[22] https://www.alibabacloud.com/zh/product/data-lake-analytics , Alibaba Cloud.

[23] https: //parquet.apache.org, Apache Parquet.

[24] X. Zhou, C. Chai, G. Li, and J. Sun, "Database meets artificial intelligence: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 3, pp. 1096–1116, 2022.

[25] J. Wang, C. Chai, J. Liu, and G. Li, "FACE: A normalizing flow based cardinality estimator," *Proc. VLDB Endow.*, vol. 15, no. 1, pp. 72–84, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol15/p72-li.pdf

[26] J. Sun, G. Li, and N. Tang, "Learned cardinality estimation for similarity queries," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds. ACM, 2021, pp. 1745–1757. [Online]. Available: https://doi.org/10.1145/3448016.3452790

[27] D. V. Aken, D. Yang, S. Brillard, A. Fiorino, B. Zhang, C. Billian, and A. Pavlo, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1241–1253, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1241-aken.pdf

[28] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P. Larson, D. Kossmann, and R. Acharya, "Qd-tree: Learning data layouts for big data analytics," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2020, pp. 193–208. [Online]. Available: https://doi.org/10.1145/3318464.3389770

[29] X. Yu, G. Li, C. Chai, and N. Tang, "Reinforcement learning with tree-lstm for join order selection," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1297–1308. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00116

[30] G. Li, C. Chai, J. Fan, X. Weng, J. Li, Y. Zheng, Y. Li, X. Yu, X. Zhang, and H. Yuan, "CDB: optimizing queries with crowd-based selections and joins," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, 2017, pp. 1463–1478. [Online]. Available: https://doi.org/10.1145/3035918.3064036

[31] X. Yu, C. Chai, G. Li, and J. Liu, "Cost-based or learning-based? A hybrid query optimizer for query plan selection," *Proc. VLDB Endow.*, vol. 15, no. 13, pp. 3924–3936, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3924-li.pdf

[32] J. Wang, C. Chai, J. Liu, and G. Li, "Cardinality estimation using normalizing flow," *VLDB J.*, vol. 33, no. 2, pp. 323–348, 2024. [Online]. Available: https://doi.org/10.1007/s00778-023-00808-x

[33] H. Dong, C. Chai, Y. Luo, J. Liu, J. Feng, and C. Zhan, "Rw-tree: A learned workload-aware framework for r-tree construction," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 2073–2085. [Online]. Available: https://doi.org/10.1109/ICDE53745.2022.00201

[34] R. Zhu, Z. Wu, C. Chai, A. Pfadler, B. Ding, G. Li, and J. Zhou, "Learned query optimizer: At the forefront of ai-driven databases," in *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, J. Stoyanovich, J. Teubner, P. Guagliardo, M. Nikolic, A. Pieris, J. Mühlig, F. Özcan, S. Schelter, H. V. Jagadish, and M. Zhang, Eds. OpenProceedings.org, 2022, pp. 1–4. [Online]. Available: https://doi.org/10.48786/edbt.2022.56

[35] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin, "Fine-grained partitioning for aggressive data skipping," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 2014, pp. 1115–1126. [Online]. Available: https://doi.org/10.1145/2588555.2610515

[36] https://docs.oracle.com, Oracle Documentation.

[37] https://docs.teradata.com/, Teradata Documentation.

[38] M. Eltabakh, A. Subramanian, A. Al-Omari, M. Al-Kateb, S. Nair, M. Hasan, W. Cabrera, C. Zhang, A. Kishore, and S. Prasad, "Not black-box anymore! enabling analytics-aware optimizations in teradata vantage," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 2959–2971, 2021.

[39] A. Pandit, D. Kondo, D. Simmen, A. Norwood, and T. Bai, "Accelerating big data analytics with collaborative planning in teradata aster 6," in *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 1304–1315.

[40] X. Tang, R. M. Wehrmeister, J. Shau, A. Chakraborty, D. Alex, A. A. Omari, F. Atnafu, J. Davis, L. Deng, D. Jaiswal, C. Keswani, Y. Lu, C. Ren, T. Reyes, K. Siddiqui, D. E. Simmen, D. Vidhani, L. Wang, S. Yang, and D. Yu, "SQL-SA for big data discovery polymorphic and parallelizable SQL user-defined scalar and aggregate infrastructure in teradata aster 6.20," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 2016, pp. 1182–1193. [Online]. Available: https://doi.org/10.1109/ICDE.2016.7498323

[41] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394.

[42] A. Jindal, S. Qiao, R. Sen, and H. Patel, "Microlearner: A fine-grained learning optimizer for big data workloads at microsoft," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 2423–2434.

[43] X. Yu, G. Li, C. Chai, and N. Tang, "Reinforcement learning with tree-lstm for join order selection," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1297–1308. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00116

[44] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nat.*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[45] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[46] https://github.com/openmessaging/benchmark, Openmessaging.

[47] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica, "Deep unsupervised cardinality estimation," *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 279–292, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol13/p279-yang.pdf