Separation Is for Better Reunion: Data Lake Storage at Huawei

ABSTRACT

Abstract here.

1 INTRODUCTION

As the Internet of Things (IoT) and 5G communication technologies become increasingly 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 disaggregated storage and compute architectures. For example, data lake storage (*e.g.*, AWS S3 [], Huawei OceanStor Pacific []), with its 10× better price, availability, and persistence compared to traditional storage formats, has been very popular for storing massive various data, so as to support large-scale data analysis.

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 (29%) handle more than 10 PB, while 8% manage over 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 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% 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.

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, typical system like Kafka uses local file system as the storage, which lacks of elasticity capacity because the computation and storage are tightly coupled. Also, in practice, given the same data, over which users may conduct stream or batch processing for different applications, thus storing two copies for different processes is costly. (2) In a data analysis pipeline, there are likely to be multiple copies when data goes through the pipeline. 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 in a database 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, so as 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 the 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. 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 storing for two copies separately, and thus the storage cost is reduced. Existing systems like the widely-used Kafka [] uses the local file system to persist data, which is less elastic than the disaggregated storage. Pravega [] and Pulsar [] adopt the disaggregated architecture, but unlike we can automatically mange 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 [] or S3 [].

Second, in terms of supporting updates, Lakehouse system [] can address this by achieving concurrent read and write in an ACID manner. We also implement the lakehouse functionality in `StreamLake` to support ACID via the table object. Particularly, we design a global Non-Volatile Memory (NVM) cache that combines small I/O accesses $w.r.t.$ the metadata, so as to improve the efficiency of compute engines visiting 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 [] have focused on using machine learning techniques to optimize database systems, including the knob tuner, query optimizer etc. For the data lake system with the 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 [CC] **[improve the block utilization while keeping the system running smoothly.]** We

also build a predicate-aware partitioning model that is utilized to judiciously distribute data to [CC] **[storage blocks]** to reduce the number of blocks 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 also provides load-balanced stream storage, which can also help improve the efficiency. In addition, we also adopt the query computation pushdown to reduce the data transfer between the storage and query engine. Furthermore, the LakeBrain optimizer can improve the query performance by optimizing the data layout.

High reliability. StreamLake is built on top of the Huawei OceanStor Pacific [], 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 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 as the data redundancy strategy, which stores less copies of data than other systems like HDFS (improving the disk utilization rate from 33% to 91%), and we also use built-in tiering storage and compression techniques to save the 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 warehouse 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 utilizations at 26%, 41%, and 66% respectively.

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 result of our deployment. With StreamLake, our customers can run the same number of analytic jobs with 39% less servers, due to the high utilization of server resources in StreamLake, and leading to

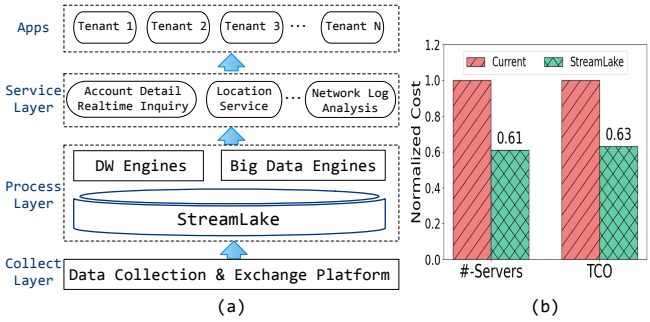

Figure 1: China Mobile Use Case.

37% cost saving (TCO). In addition, our customers no longer need to maintain data management between independent Kafka and HDFS servers, which could be expensive and error-prone. 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 6.

2 ARCHITECTURE

In order to meet the demands of enterprise customers for next-generation big data solutions, StreamLake aims at optimizing the processing 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.

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 [] offers high-speed data transfer and interworking of different storage abstractions.Its advanced features include support for Remote 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 3.

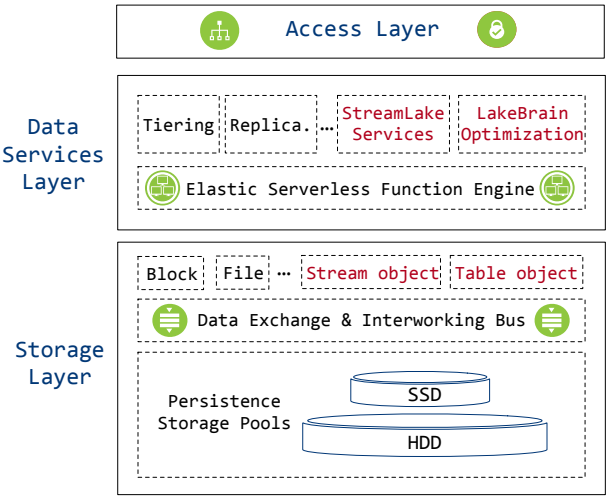

Figure 2: StreamLake Storage Architecture.

Data service layer provides a rich set of features to enable efficient data management 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, which saves the storage cost a lot. 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 4) to support real-time streaming and lakehouse functionality, and `LakeBrain` (Section 5) 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.

3 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.

3.1 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 continuous message streams, organized as a collection of data slices. Each slice can contain up to 256 records as depicted in Figure 4. Incoming message records are

```
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);
```

Figure 3: Stream Object Operations.

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 for operating 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 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 on the basis of 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 [] 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.

3.2 Table Object

We also extend the storage object layer in `StreamLake` to support operations over tables for more effective data storage and management, like the lakehouse systems []. The table storage uses an open

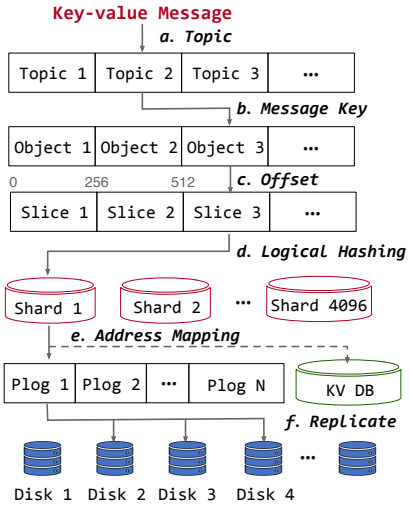

Figure 4: Write Message to StreamLake.

Figure 5: File Organization of StreamLake Table Objects.

lakehouse format with putting catalog in KV store for faster metadata access. The table abstraction is logically defined by a directory of data and metadata files, as shown an example in Figure 5.

Data directory. Table objects are stored in Parquet files of the data directory. In this example, the table is partitioned based on the date column, so the data objects are separated into different sub-directories by the location. Each sub-directory name represents its partition range. The data objects in each Parquet file are organized as row-groups and stored in a columnar format for efficient data 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, table schema, and transaction commits etc., which are organized into three levels: commit, snapshot, and catalog, as shown 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 document 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 for multiple readers and one writer to access the data simultaneously without the need for locks.

Snapshots also monitor the expiration of all commits, making them essential for supporting time travel. Time travel queries allow data to be viewed as it appeared at a specific time. By keeping old commits and snapshots, the table object enables the use of a timestamp to look up the corresponding snapshot and commits, so as to access to 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, which is stored in a distributed key-value engine optimized for RDMA and SCM to ensure fast metadata access. The data and metadata files are converted to PLogs in the underlying storage for redundant persistence as discussed above.

4 STREAMLAKE DATA PROCESSING

In this Section, we present the data processing services in the data layer. Driven by practical application scenarios discussed in Section 1, 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 4.1), lakehouse-format read/write capabilities for efficient tabular data processing (Section 4.2), and support for query operator computation pushdown (Section 4.3).

4.1 Message Streaming

We develop a distributed stream storage engine that facilitates message streaming at large scale. Our engine leverages the stream object storage abstraction to ensure enterprise-level scalability via the disaggregated storage architecture.

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

Producers and Consumers. Producers are responsible for publishing messages to topics, which are named resources for categorizing 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

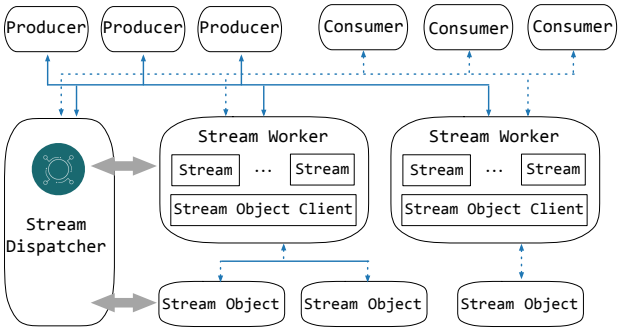

Figure 6: Write Message to StreamLake.

"topic_streamlake_tes". 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*/ }
```

Figure 7: Sample code of Producer and Consumer.

Stream workers work together with stream objects discussed in Section 3.1 to tackle stream processing and message storage. The number of stream workers is determined by configurations and the physical resources allocated to the 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.

The task of 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, the 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. This topology tracking 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 }}
```

Figure 8: Stream Storage Configuration Example..

The stream dispatcher also sets configurations for the messaging service in the unit of topic. An example of configurations 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 4.2.
- *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 tolerant node, network, and disk failures. (4) The system provides exactly-once semantics through the use of 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 switch 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.

4.2 Lakehouse Read and Write

`StreamLake` also provides support for concurrent tabular data reads and writes, similar to the architecture of lakehouse []. Besides directly inserting tabular data, we can also get it from the conversion from 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 in stream objects to table objects, allowing for efficient downstream processing, which is triggered by the `convert_2_table` configuration in Figure 8, which include the table schema and time for data freshness in the downstream processing. The table schema must be specified at 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 just retain 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 the 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 details.

`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 to the non-volatile memory (NVM), 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 significant number of small files, we leverage a global 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 NVM 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.

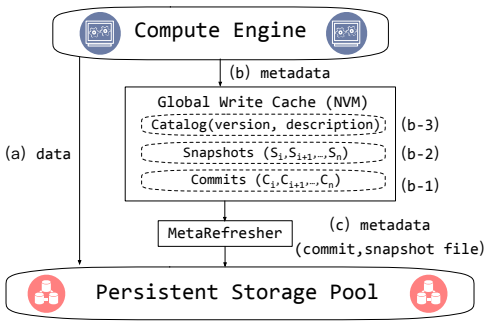

Figure 9: Write Cache Acceleration in Lakehouse Read/Write.

`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 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 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 reads and writes to reduce data transmission to/from the compute engines.

UPDATE: Similar to the delete operation, 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.

4.3 Query Operator Computation Push Down

In this subsection, we introduce the computation pushdown (similar to S3 Select []) to reduce the amount of data transfer between the storage engine of `StreamLake` and the query engines. Our pushdown method is built on top of an elastic, serverless engine in the data service layer. Here, we choose serverless computing as our execution model because it is lightweight and flexible, which allows us to quickly start a large number of instances for computation tasks near the data sources, and free the resources as soon as the tasks are completed. The elasticity is important since CPU resources can be scarce during critical data management jobs.

The main components of the serverless function engine are shown in Figure 10, which include function dispatcher, worker instances, a worker manager, and a function repository. The function dispatcher schedules jobs and manages workflows, and worker instances execute the tasks. The worker manager oversees server resources and the life cycle of worker instances, and the function repository registers and stores function images. These modules work together to support elastic serverless computing.

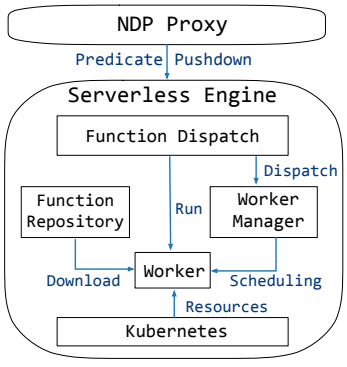

Figure 10: Serverless Function Engine.

To be specific, when a job request is received, the function dispatcher obtains the data location from the storage devices and selects the appropriate storage nodes based on data distribution and available resources. The worker manager is then requested to deploy worker instances to the selected node. The worker instances download the necessary functions from the repository and execute the jobs using data from the storage infrastructure. Upon completion, a callback message is sent to the caller to notify them of the results. To ensure service quality, elastic scaling policies are used to dynamically adjust the number of nodes based on workload. For example, a load-balanced method[1] is used to balance scheduling among instances.

To achieve maximum pushdown benefits, three categories of query operators are supported,

- `Projection Pushdown:` Only selected columns will be returned.
- `Filter Pushdown:` Only rows satisfying the filtering conditions will be returned.
- `Aggregate Pushdown:` The results of aggregate functions such as `Count`, `MAX` and `AVG` will be returned.

These operators are selected because the size of their output could be significantly smaller than the input, and thereby a large optimization opportunity can be achieved. These query operators are implemented as separate functions, which are registered and executed in the serverless engine service. The implementation allows for sharing and reuse of the same query operator function by different query engines, as long as its image is registered in the serverless engine.

To facilitate the pushdown of query operators from the compute layer to the storage cluster in `StreamLake`, we have introduced the near data processing (NDP) Proxy and compute engine plugins for popular big data engines such as Hive, Spark, and Presto [] to invoke query operator pushdown. During query planning, the optimizer of query identifies operators that can be pushed down and sends the information to the NDP Proxy. Then the NDP Proxy inserts these requests into a queue for traffic control and then sends them to the serverless engine for execution as shown in Figure 10. As soon as the query results are ready, the NDP Proxy transfers them back to the compute engine using the high-speed data exchange bus of `StreamLake`, so as to ensure efficient data transfer. Overall, this process leverages the flexibility and elasticity of serverless computing, which allows for efficient use and release of server resources as required, resulting in better overall performance and resource utilization. In addition, inspired by methods proposed by [**? ?**], query operator pushdown can be used together with materialized views as cache to further optimize query performance. This feature is on the roadmap to add to `StreamLake`.

5 LAKEBRAIN OPTIMIZATION

Optimizing query processing over large-scale data is significant in data warehouse and big data systems, as discussed in []. However, for the `StreamLake` system with complicated storage-disaggregated architecture, it is challenging to optimize the query performance and resource usage. The reasons are two-fold. First, it is hard to capture the entire environment about the compute and storage cluster as well as the queries executed by other engines simultaneously. Second, even though all the environment data is available, it is still

[1] not specific

hard to optimize because of the large search space due to the large number of tunable and interdependent variables [].

To address the challenges, we present LakeBrain, a novel data lake storage optimizer that aims to optimize the the data layout at the storage-side, so as to improve the resource utilization as well as query performance. Unlike query engine optimizers that focus on join order and cardinality estimation [], LakeBrain aims to optimize data layout in storage, which is key to improve both query performance and storage resource utilization in a storage-disaggregated design. In this Section, we mainly focus on two cases, *i.e.*, automatic compacting small files and judiciously partitioning tables to improve the resource utilization improve the query performance. Next, we will illustrate the above aspects in detail.

5.1 Automatic Compaction

In a streaming application scenario, data ingestion and transactions often result in numerous small files, leading to low query performance on merge-on-read (MOR) tables. A typical method is to compact files statically using rule-based methods such as setting a time window or a data size threshold. 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 is defined by CC[XX]. As streaming data is continuously ingested, we are likely to frequently determine whether to merge small files. 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 or ingestion slow down. On top of that, compaction consumes relatively large amount of computing resource.

Moreover, at each state, a number of system parameters CC[(*e.g.*,) total?] will influence the system, and the action (*i.e.*, whether to merge files) does not purely influence current system situation, but also future states. For example, CC[XX.]

Therefore, we propose a reinforcement learning framework that can well capture the relationship between system parameters of each state and the long-term benefit (considering the future system states) of conducting the compaction or not. To be specific, as shown in Figure 11,

Agent can be taken as our automatic compaction module that receives the reward (resource utilization) and state (system parameters) from the environment (the storage system). Then it updates the policy network (*e.g.*, DQN []) to guide whether to conduct the compaction operation so as to maximize the long-term reward.

State denotes the current state of the storage system, described by CC[many] features, for example, among which CC[XX] are significant ones closely related to the compaction problem. These features will be encoded as the input of the policy network.

Reward reflects whether the compaction has a positive or negative impact on the system and the extent of the impact. Specifically, if the compaction succeeds, the reward is computed by the improvement of the block utilization. If it fails, the reward is the minus of the number of partitions that participant this compaction.

Action denotes whether we conduct the compaction at each state, which is the output of the policy network.

CC[how to compaction??]

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. This 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.

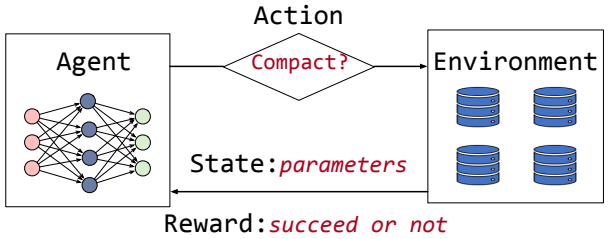

Figure 11: Automatic Compaction using RL.

5.2 Predicate-aware Partitioning

With the data volume 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 CC[number of blocks or tuples?] to be assessed is minimized, and thus the efficiency is improved.

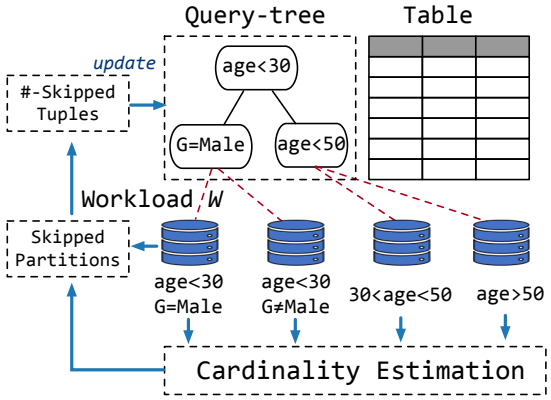

Figure 12: Predicate-aware Partitioning.

Specifically, our partition method is based on the query-tree framework [], and additionally leverage the machine learning based cardinality estimation method for optimizing the query tree, so as to find a fine-grained data partition with high query efficiency. As shown in Figure 12, 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. Hence, we can use data-driven cardinality estimation methods [] to estimate the cardinality accurately and efficiently via learning the data distribution. In practice, we use the sum-product network [] as the estimator.

6 EXPERIMENT

6.1 Experimental Settings

Our Experimental Scenario. To demonstrate the performance of the StreamLake framework, we analyze a simplified[2] real-world use case. The case compares StreamLake framework with an open-source storage solution to build a big data processing pipeline that can facilitate business analysis. Specifically, a mobile financial application company collaborates with a mobile carrier to collect and analyze its app usage data. The company aims to understand its app usage patterns to prevent frauds and enhance its product experience. The mobile carrier provides this analytic service through an end-to-end big data processing pipeline. This pipeline includes several jobs such as data collection, normalization, labeling, and querying, as depicted in Figure 14.

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

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

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

(d) Query: After the normalization and labeling processes are completed, the records are inserted into tables and are available for query engines. To perform analyses, the app company employs secure API calls to query the data. Figure 13 illustrates an example SQL query that counts the daily active users (DAU) in different provinces. More complicated analysis, like hidden Markov and Gaussian Mixture Models[3], can also be applied to draw user profiles and identify abnormal activities.

To support both full data and real time analyses, the network carrier builds two data flows in the pipeline. One flow processes full data in batch every two hours and the other processes stream messages constantly to deliver time-sensitive logs such as new logins, payments and password modifications. This ensures that the network carrier can effectively analyze both historical data and real-time events to make accurate and timely decisions.

Settings. This use case is evaluated in a commodity[4] cluster using different sizes of input data packets and the results are compared with open-source storage solution Hadoop Distributed File System (HDFS) [] and Kafka []. The reason of why we choose the two

[2] why simplified?
[3] why mention models?
[4] @

```
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 consumer = province;
```

Figure 13: Query Example of Computing DAU.

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, as we know, many other companies also use HDFS and Kafka to cope with similar application scenarios. CC **[For above, do we need more justification? like why hdfs and kafka fit? or common sense?]**

To be specific, the cluster hardware[5] 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. 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 14 shows the data processing process. Kafka and HDFS serves 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 it[6] fails 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. CC **[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.]**[7] 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.

6.2 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 usage and the batch processing time. The storage usage in the HDFS and Kafka is 4 times as much as StreamLake. The reason is that in HDFS and Kafka, full data is

[5] cluster hardware?
[6] who?
[7] too many and

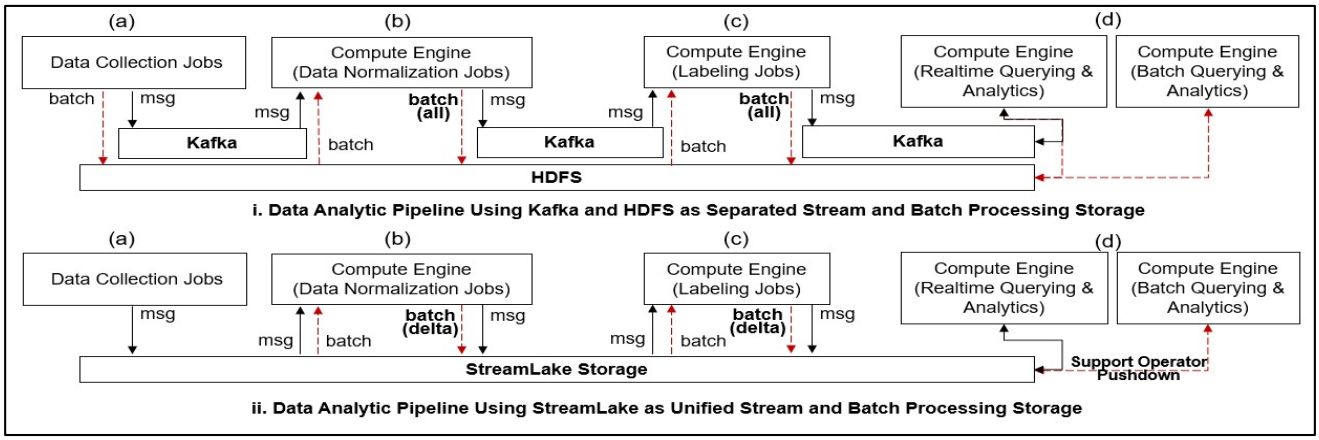

Figure 14: Data Analytic Pipelines for a Real-world Use Case.

	#-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.

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, since the storage natively supports time travel, we only save one 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 of skipping 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 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.

CC **[For the above, can we correspond to the designs with respect to previous sections as the reasons of improvement]**

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.

6.3 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 [] as our benchmark framework as 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 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 CC **[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 volumes we process are 100 TB, 500 TB and 1 PB respectively.

Figure **??** shows the results. 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 almost the same throughputs, indicating that it does not improve the throughput to add persistent memory as a cache. However, as shown in Figure **??**(b), persist memory reduces the latency as we expect, especially when the workload is 200k messages per second

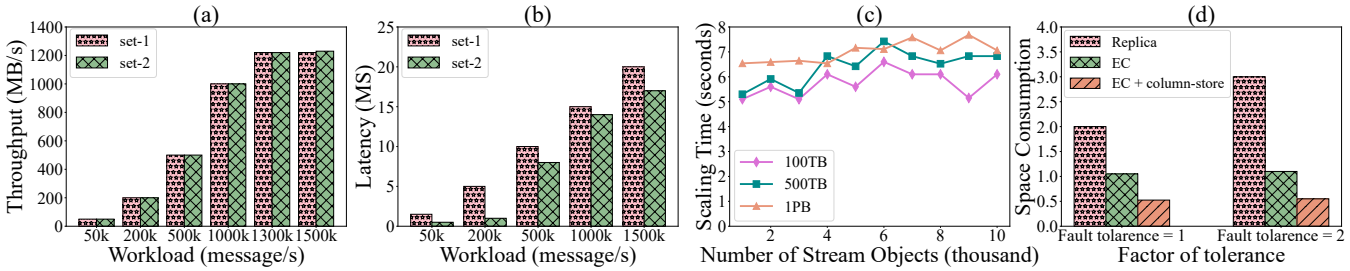

Figure 15: Stream Engine Throughput, Latency, Scalability and Space Consumption.

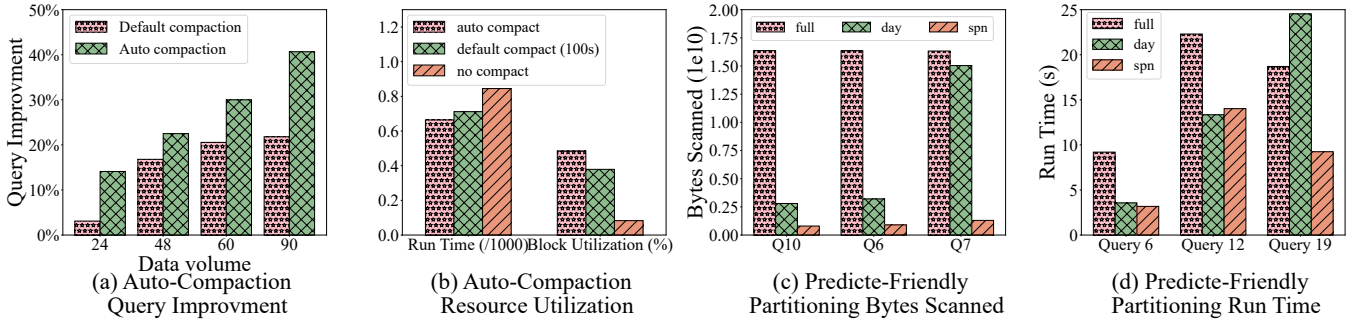

Figure 16: LakeBrain Compaction and Partitioning Performance.

or less. Figure **??**(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 data centric and disaggregated storage design[8]. CC **[Below is hard to follow!]** Finally, Figure **??**(d) compares the volumes of different storage strategies. Without scarifying the reliability, `StreamLake` provides the option to use erasure coding[9] and column-store which can offer three to five times of volume compared to standard storage with one or more replicas.

6.4 Evaluation of `LakeBrain`

In this part, we evaluate the two components in `LakeBrain`, *i.e.*, auto-compaction and predicate-aware partitioning.

Auto-Compaction: To precisely evaluate the effectiveness of our automatic compaction strategy, a `TPC-H` based test bed[10] 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 three compaction strategies are deployed: (1) No compaction. (2) A static strategy which simply compacts data files in a 30 second interval. (3) Auto-compaction, respectively. During the ingestion, multiple rounds of `TPC-H` queries are executed in parallel to obtain their end-to-end performance. As shown in Figure **??**(a), the results depict how much improvement of query performance that the compaction strategies can make, compared with baselines. We ca observe that the auto-compaction strategy outperforms the static one for all data volumes. As the data volume increases, the advantage becomes more significant. CC **[Why?]**

In addition to the query performance, we also evaluate the block utilization of the auto compaction. Specifically, we control the file ingestion speed such 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 above, we deploy three methods: (1) No compaction, (2) The static strategy compacts data files in a fixed time interval[11], and (3) Auto-compaction. We can observe that the auto-compaction outperforms the static strategy in term of block utilization.

CC **[why?]**

CC **[Ingestion speed? Below is hard to follow!]**

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 **??**(b) summarizes the results of all three test groups. Compared with no compaction and the static strategy, our method performs better in term of both block utilization and query run time.

[8] term?
[9] term?
[10] @ a term?
[11] *no need to repeat, which interval?

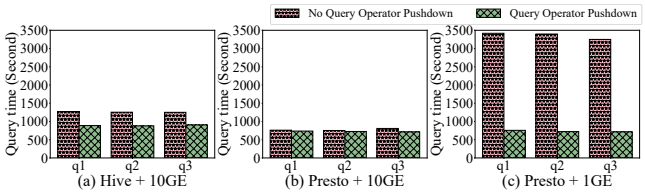

Figure 17: Computation Pushdown Query Time.

Predicate-Aware Partitioning: We also test the partitioning method on TPC-H with different scale factors. We train the probabilistic model with 3% of the data randomly sampled from the `lineitem`[12] 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) Partition by the day of `l_shipdate` (`day`). (3) Our 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. The workload includes TPC-H query 6, 12 and 19 which involve `lineitem` table and include predicates other than `l_shipdate`. We skip other TPC-H queries because their performance is mainly dominated by the joins of multiple tables, which is beyond our purpose.

CC **[Below is not clear enough!]**

The results in Figure **??**(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 is dominated by table joining, the optimized partitioning also demonstrates some improvement for query 6 and query 19, considering we only optimize the partition of the `lineitem` table.

CC **[Why not reverse as previous sections?]**

6.5 Evaluation of Query Pushdown

In this part, we evaluate the query operator pushdown which we believe can provide stable query runtime regardless of network conditions. This is significant in the real-world deployment as it is always expensive to upgrade the data center network.

In our experiment, we use two groups of clusters with different network bandwidths: one with 10 GB bandwidth and the other with 1 GB. To accurately assess the benefits of the query operator pushdown method, we carefully select three live queries with data-intensive operations and 4.8 TB of data from a China Mobile production environment. We deployed two different query engines, `Hive` [] and `Presto` [], so as to process the SQL queries for generalization. We observe that without query operator pushdown, query performance of baselines varies across engines. When 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 reduces to 1 GB, all execution times increases to over 3000 seconds. In comparison, when we apply query operator pushdown, CC **[the runtime of all the queries is close to 900 seconds with less than 10 difference]**, regardless of compute engines and network bandwidths, which indicates a four times performance advantage when the engine processes queries in an 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.

[12] @

7 RELATED WORK

In this section, we will discuss relevant open-source projects and systems related to `StreamLake` *w.r.t.* CC **[streaming platforms]**, lakehouse data management, query computation pushdown and automatic database tuning.

CC **[Data lake storage system]**

Streaming platforms. Kafka, Pulsar and Pravega [] are widely-used open-source streaming platforms in industry. Unlike `StreamLake`, which builds its 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 require manual connections to compute engines and external storage, such as `HDFS` [] or `S3` [], for downstream processing or cost-friendly archiving. This increases both the complexity and cost of data pipeline management.

Lakehouse. Iceberg, Hudi and Delta Lake [] are popular lakehouse data management framework, which rely on statistic file or object storage. Massive data transmission between the storage and the compute engines are inevitable in many scenarios. CC **[Not coherent!!!]** `StreamLake` builds the lakehouse framework on top of the table object and PLogs, leveraging the enterprise-level data redundancy, high performance cache and query computation pushdown to provide reliable and high speed concurrent lakehouse reads/writes.

Automatic database tuning . Recently, AI is widely-used inside the database system to improve the performance. 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 sub-optimal 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.

8 CONCLUSION

REFERENCES