Separation Is for Better Reunion: Data Lake Storage at Huawei

ABSTRACT

Abstract here.

1 INTRODUCTION

Intro here.

2 MOTIVATION

3 ARCHITECTURE

4 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.

4.1 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 2. 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 1 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 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 on the basis of SSD and HDD storage pools. As shown in Figure 2, the messages are first assigned to stream object slices based on topics, keys, and offsets (Figure 2-a,b,c). Then, a distributed hash table is leveraged to ensure even data distribution for load-balance storage (Figure 2-d). Specifically, data slices will be distributed evenly to

```
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 1: Stream Object Operations.

4096 logical shards, each of which has the storage space managed by persistence logs (PLog, Figure 2-e)[1]. 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 2-f). Key-value databases[2] serve as indexes for PLogs for fast record lookup.

4.2 Table Object

We also extend the storage object layer in StreamLake to support table-like[3] operations for more effective data storage and management, similar to lakehouses [][4]. The table storage uses an open lakehouse format[5] with optimizations[6] for faster metadata access. The table abstraction is logically defined by a directory of data and metadata files, as shown an example in Figure 3.

`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 date. Each sub-directory name represents its partition range[7]. The data objects in each Parquet file are organized as row-groups[8] 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 3-(b, c, d).

Commits are `Arvo` files that contain file-level metadata and statistics such as file paths[9], record counts, and value ranges for the data

[1] –Plog 4096
[2] –can we name it specifically?
[3] – a term?
[4] * "similar to lakehouse" is weird
[5] –a term? what does it mean?
[6] –what? a part in Fig4? catalog
[7] –partition key?
[8] –a term?
[9] @directory path

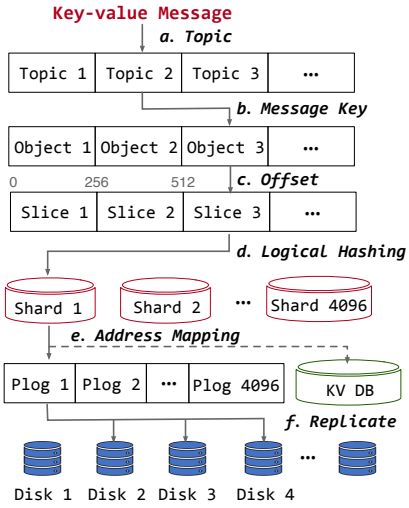

Figure 2: Write Message to StreamLake.

objects. Each data insert, update, and delete operation will generate a new commit file to record changes to 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 [10] 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.

5 STREAMLAKE DATA PROCESSING

In this Section, we present the data processing services in the data layer. Driven by practical application scenarios discussed in Section 2, 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 5.1), lakehouse-format read/write

[10] –can we name it? or cite

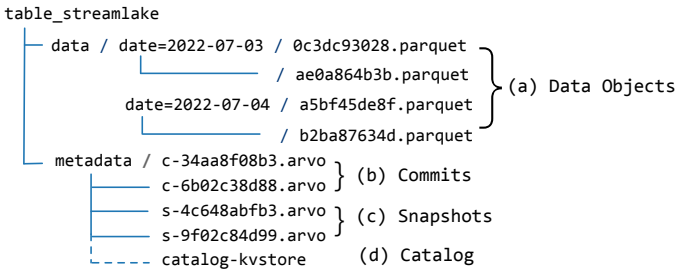

Figure 3: File Organization of StreamLake Table Objects.

capabilities for efficient tabular data processing (Section 5.2), and support for query operator computation pushdown(Section 5.3).

5.1 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 reliability and scalability.

Overall architecture of streaming service. The high-level design of the stream service is shown in Figure 4. The stream storage system comprises of producers, consumers, stream workers, stream objects, and a stream dispatcher, which work together to provide seamless message streaming.

CC **[which techniques to ensure reliability and scalability? Here, can we summarize something different (our characteristic) ?]**

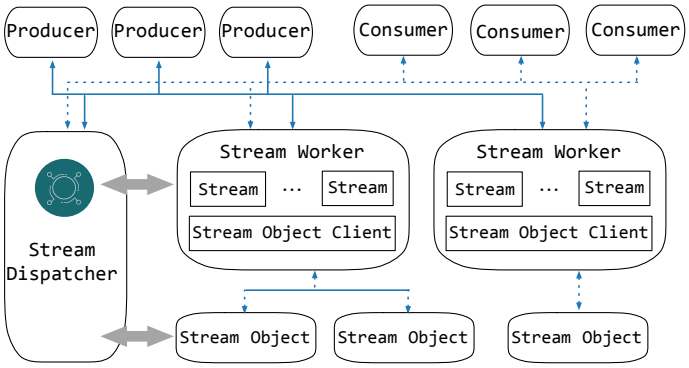

Figure 4: Write Message to StreamLake.

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 5 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_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 5: Sample code of Producer and Consumer.

Stream workers work together with stream objects discussed in Section 4.1 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[11] 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 2.

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 and 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.

The stream dispatcher also sets configurations for the messaging service in the unit of topic[12]. An example of configurations is shown in Figure 6.

[11] streams? Above we talk about messages

[12] @ what is the connection with the above? I want to ask the relationship among topic, stream, messages

```
{ "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 6: Stream Storage Configuration Example..

• *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 SCM[13] caches.

• *The* `convert_2_table` *configuration* enables the automatic conversion of stream object messages to table object records. 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* `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 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.

CC **[The StreamLake stream storage provides guaranteed delivery, efficient transfer, and high elasticity for enterprise use.[14]]**

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[15] 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.

[13] no full name

[14] We should build connection with the above designs.

[15] @

Efficient Transfer: Our system implements several mechanisms to efficiently transfer data. First, Stream workers and stream objects are connected through a data bus[16] 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[17] scenarios. Finally, a local cache is implemented at the stream object client to speed up message consumption.

High Elasticity: Our system provides high elasticity by decoupling data storage and data serving. 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.

5.2 Lakehouse Read and Write

5.3 Query Operator Computation Push Down

[16] *reviewer suggest to extend

[17] low latency?