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 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.

```
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.

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).

<u>*Commits*</u> 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

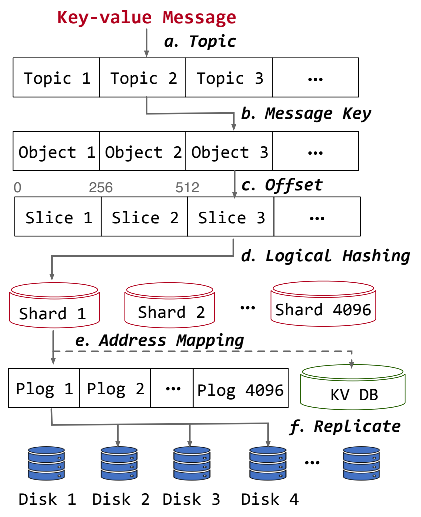

Figure 2: Write Message to `StreamLake`.

objects. Each data insert, update, and delete operation will generate a new commit file to record changes to the data object files.

<u>*Snapshots*</u> 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.

<u>*Catalog*</u> 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

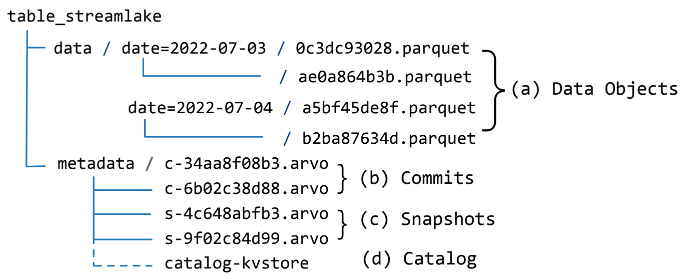

Figure 3: File Organization of `StreamLake` Table Objects.

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) ?]**

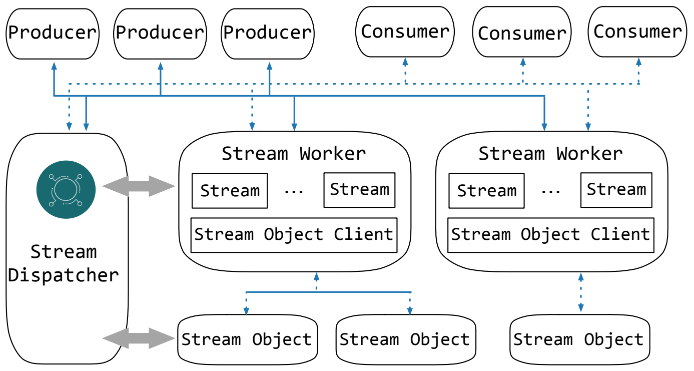

Figure 4: Write Message to `StreamLake`.

<u>*Producers and Consumers.*</u> 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

`StreamLake` also provides support for concurrent tabular data reads and writes, similar to the architecture of lakehouse []. This section describes the storage conversion from stream messages to tabular records[18], as well as the implementation of key lakehouse operations.

^{CC}**[What is the advantage of this conversion?]**

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 6, which include the table schema and bounds for data freshness [19] 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 [20] can choose to just retain messages in crucial topics as stream objects to support real time applications while converting most ^{CC}**[stream]** data to table objects. The reverse conversion, from table records to stream messages, is also supported for data playback. As shown in Figure **??** and Table **??**, this design provides a good trade-off between the system cost and latency, which also helps to decouple the data processing from the business logics.

^{CC}**[What is the connection between the two paragraphs?]**

Our `StreamLake` services implement lakehouse read/write operations using a table object and high-performance caches and computation pushdowns, which eliminate unnecessary data transmission and accelerate concurrent data reads and writes.[21] 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 spec[22], 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. [23] 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[24] 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[25]. (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.

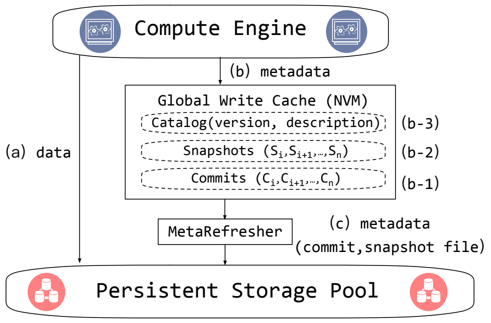

Figure 7: Write Cache Acceleration in Lakehouse Read/Write.

`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

[16] *reviewer suggest to extend
[17] low latency?
[18] why discuss the conversion here
[19] bounds?
[20] *why not automatic?
[21] who optimize who?
[22] ?
[23] *There seems a lack of summarization like why, bennefit and connection.
[24] a commit includes multiples records?
[25] is there an arrow from storage to b2?

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[26] are applied to process file reads and writes without[27] 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[28], are applied to reduce[29] 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.[30] (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.

5.3 Query Operator Computation Push Down

In this subsection, we introduce the computation pushdown to reduce the amount of data transfer between the storage engine of StreamLake and the query engines [31]. It is built on top of an elastic, serverless engine[32] in the data service layer. Here, we choose serverless computing as our execution model due to it is lightweight and flexible, which allow 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 9, 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.

CC **[pushdown what?]**

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 balancing method[33] 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[34] is registered in the serverless engine.

CC **[need connection]**

To facilitate the pushdown of query operators from the compute layer to the storage cluster in StreamLake, we have introduced the NDP Proxy[35]. 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 9. 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.

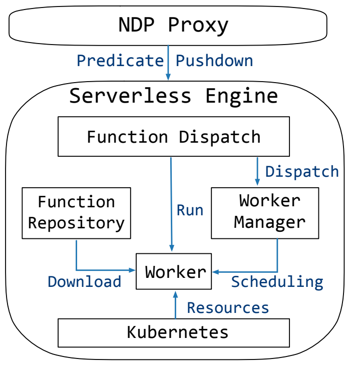

Figure 8: Serverless Function Engine.

[26]next section, s?
[27]without?
[28]same
[29]reduce or without?
[30]@
[31]we can support many query engines? Will we mention it?
[32]@
[33]not specific
[34]@
[35]why need ndp, the full name?

6 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 with multiple compute engines, it is challenging to optimize the end-to-end 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 hard to optimize because of the large search space due to the large number of tunable and interdependent variables [].

To address this challenge, we present `StreamLake`, a novel data lake storage optimizer that complements end-to-end data pipeline optimization. Unlike query engine optimizers, which focus on join ordering and cardinality estimation [], `StreamLake` aims to optimize data usage during query execution, which is key to improving both query performance and storage resource utilization in a storage-disaggregated design. For instance, 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. LakeBrain can use compaction to combine these small files into fewer, larger ones, improving inter-cluster storage and network usage as well as query performance.

LakeBrain's design is kept simple for ease of extension and support for different applications. It consists of three components: a statistics collector, the core optimization logic, and an executor. The statistics collector gathers system configurations, environment variables, and workload history, while the core optimization logic employs heuristic rules, probabilistic models, and machine learning algorithms to suggest the best strategy candidates. The executor then deploys the chosen strategy, with its effects being collected as feedback by the statistics collector for future optimization.

To demonstrate the value of a data lake storage optimizer, we have developed two LakeBrain applications: auto compaction and predicate-aware fine-grained partitioning. These use cases will be explained in detail in the following section.

6.1 Automatic Compaction

File compaction aims to find the optimal strategy for compacting files that result in improved query execution time or increased block utilization in storage. To achieve this goal, the optimization process employs two algorithms: particle swarm optimization (PSO) and reinforcement learning (RL).

PSO is used to search for the global optimum, a population-based method that doesn't require assumptions about the relationship between tunable parameters and query performance. The goal is to obtain an approximately optimal solution within a limited time. RL, on the other hand, finds a more sophisticated policy based on the states of the data lake environment.

The optimization process involves considering a set of discrete compaction configurations within the action space. Since the state space is continuous, a function approximation method is preferred. The stability of the training process is critical due to the high degree of variability in query performance in a distributed environment, so proximal policy optimization (PPO) is applied.

A deep neural network (DNN) is used to approximate the policy and the value function, with a shared feature backbone network that covers both global and local characteristics of the states. The output from the feature network is processed by two fully connected networks to compute the policy output and the action value. The actor and critic are alternatively updated after collecting new trajectories using the latest policy during training.

Once a desired result is obtained, the numerical output of the compaction strategy is translated into actionable operations by the data lake connector for a specific data lake engine, facilitating the optimization process.

6.2 Predicate-aware Fine-grained Partitioning

Optimizing data partitioning involves assigning records to storage blocks in the most efficient manner possible, thereby reducing the number of blocks accessed during queries. Our partitioning approach is based on the query-tree framework [43], and utilizes a sum-product network (SPN) probabilistic model [19, 26, 31] to model the distribution of the data in LakeBrain. This is done in order to ensure fast inference speed and avoid repeatedly scanning the datasets.

The query-tree framework creates a tree-based partitioning strategy using pushdown predicates. Each leaf node represents a partition, and its column ranges are derived from the pushdown predicates used to split its parent nodes. By using probabilistic models to characterize the dataset, we can identify the most suitable partitioning policy. The probabilistic model-based cardinality estimation, as demonstrated in Figure 11, is used to estimate the number of records in each partition, instead of scanning the original data, thereby saving a significant amount of time. This greatly improves the speed of the partitioning optimization algorithm, making it suitable for large-scale systems.

Additionally, probabilistic models allow us to represent a sequence of datasets with a series of probabilistic models that have a fixed structure but varying parameters. This is achieved by representing a sequence of datasets as a series of multi-dimensional vectors, each representing the learnable variables in the probabilistic model with a fixed length, i.e. a time series. We can then use time series prediction methods to predict future probabilistic models, and use these predictions to estimate the number of records in a partition during partitioning optimization.

To implement the optimized data layout, we introduce a partitioning mechanism that saves data in fine-grained partitions based on the partitioning strategy. Additionally, we have implemented an evaluator that skips irrelevant partitions by checking the overlap between pushdown predicates and the column ranges in each partition. For numerical columns, the range can be represented as lower and upper bounds, which are well-handled by many data formats. For categorical columns, we either record its range or its complement using "IN" or "NOT IN" predicates. The effectiveness of this predicate-aware partitioning approach is evaluated in section 7.2, where the test results show exceptional performance.

7 EXPERIMENT

7.1 Settings

7.2 Evaluation of Message Streaming

7.3 Evaluation of LakeBrain

7.4 Evaluation of Query Pushdown

7.5 China Mobile Use Case

8 RELATED WORK

9 CONCLUSION

REFERENCES