VIETTEL GROUP

Mini Project Report

APPLYING APACHE DORIS IN REAL-TIME DATA ANALYSIS

Dinh Viet Thanh

dvthanh19@gmail.com

Viettel Digital Talent 2024 Program

Major: Data Engineering

Mentor: Mr. Pham Cong Minh

Department: Viettel Solution

HO CHI MINH CITY, AUGUST 2024

Preface

This report delves into the application of Apache Doris in real-time data analysis, highlighting its efficacy in handling high-pressure data synchronization scenarios. The focus is on optimizing compaction scores to enhance performance and stability. The significance of real-time data processing in contemporary data-driven environments necessitates robust solutions capable of sustaining intensive loads while ensuring minimal latency. Apache Doris, with its advanced capabilities, emerges as a pivotal tool in achieving these objectives. This study provides insights into the methodologies employed, the results obtained, and the implications for future implementations.

I extend my heartfelt gratitude to my mentor, **Pham Cong Minh**, for his invaluable guidance and support throughout this research. His expertise and encouragement were instrumental in the successful completion of this study.

Summary of Content and Contributions

Introduction: This mini-project explores the application of Apache Doris in real-time data analysis. The primary focus is on optimizing data synchronization under extreme high-pressure conditions. The study aims to demonstrate the capabilities of Apache Doris in maintaining real-time data integrity and performance.

Methodology: The research employed a top-down approach to study the application of Apache Doris in real-time data analysis. Initially, the study began with a broad overview of real-time data processing requirements and the capabilities of various data processing platforms. After identifying Apache Doris as a suitable candidate, the focus narrowed to its specific features and functionalities.

Results: The study achieved significant improvements in the compaction score stability, with marked reductions in peak values and enhanced overall performance. The optimized compaction process resulted in a more consistent and lower compaction score, indicating improved efficiency in handling real-time data loads.

Personal Contributions: My contributions to the mini-project included configuring the Apache Doris environment, designing the performance tests, and conducting the data analysis. I played a key role in identifying the optimization parameters and implementing the changes. Additionally, I analyzed the results and documented the findings, highlighting the improvements achieved through optimization. This project extends existing research by providing practical insights into the application of Apache Doris for real-time data analysis under high-pressure scenarios.

Skills and Knowledge Acquired: Through this mini-project, I gained valuable skills in real-time data processing and analysis using Apache Doris. I enhanced my knowledge of data synchronization techniques and performance optimization. Additionally, I developed practical skills in configuring and deploying data processing environments, conducting performance tests, and analyzing large datasets.

Contents

1 General

1.1 Overview

Apache Doris is a real-time analytical database characterized by its high performance and MPP architecture. Renowned for its exceptional speed and user-friendly interface, it boasts sub-second response times for query results even with large datasets. Moreover, it is proficient in handling high-concurrency point queries and complex analytical scenarios, making it suitable for applications such as report analysis, ad-hoc querying, unified data warehousing, and data lake query acceleration. Users can leverage Apache Doris to develop various applications, including user behavior analysis, AB testing platforms, log retrieval analysis, user profiling, and order analysis.

1.2 History and Development

Apache Doris, initially developed as Palo for Baidu's ad reporting, was open-sourced in 2017 and donated to the Apache Software Foundation in July 2018. It was managed by the incubator project management committee under Apache advisors. In June 2022, Apache Doris graduated to a Top-Level Project. By 2024, the community grew to over 600 contributors from various industries, with more than 120 active contributors each month.

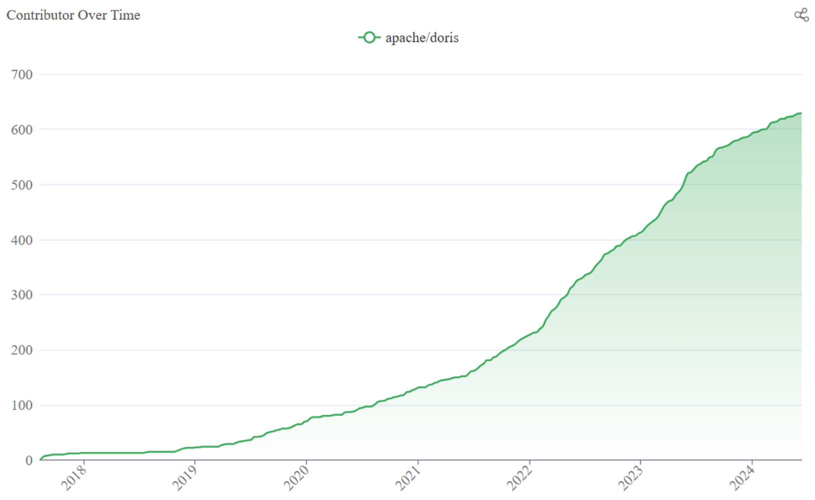


Figure 1: *The number of contributors overtime*

Apache Doris is widely used, with over 4,000 companies using it in production, including TikTok, Baidu, Cisco, Tencent, and NetEase. It serves diverse sectors such as finance, retail, telecommunications, energy, manufacturing, and healthcare. The recent release of Apache Doris 2.0 introduces many significant features, which will be explored in detail later. The Doris Community is rapidly growing, becoming one of the largest open-source projects in the data field by the number of monthly active contributors.

1.3 Usage scenarios

The diagram below showcases Apache Doris's role in a data pipeline. Data sources are integrated and processed, then ingested into the Apache Doris real-time data warehouse and offline data lakehouses like Hive, Iceberg, and Hudi. Apache Doris can be used for the following purposes:

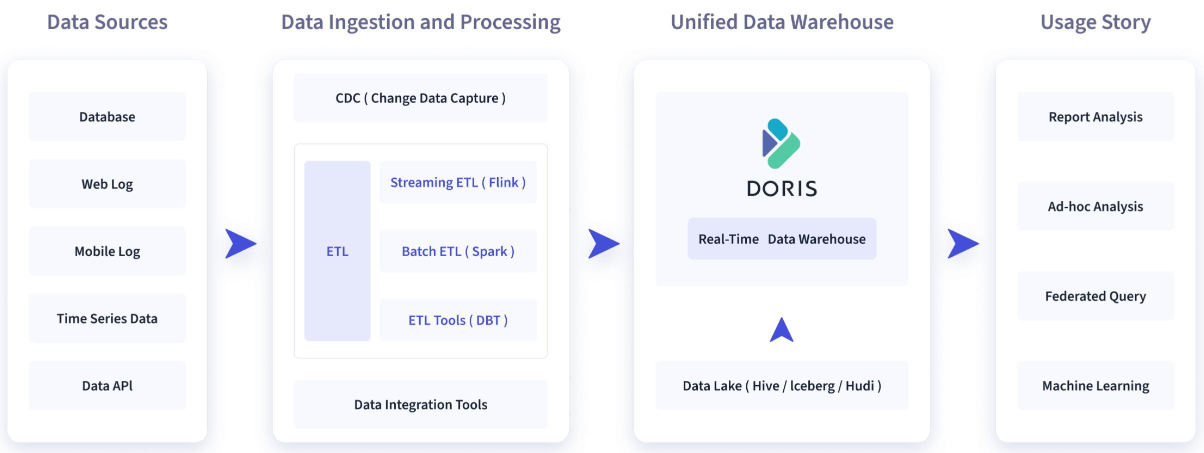


Figure 2: *Apache Doris Query Engine*

- **Report analysis**: Real-time dashboards and reports for internal analysts and managers. Moreover, Apache Doris also handles high concurrency (thousands of QPS) and low query latency (milliseconds).
- **Ad-hoc query**: Supports self-service analytics with irregular patterns and high throughput.
- **Data lake analytics**: Performs federated queries on external tables in offline data lakehouses (Hive, Hudi, Iceberg) without data copying, enhancing query performance.
- **Log analysis**: Supports inverted index and full-text search (since version 2.0), offering 10 times higher cost-effectiveness than typical log analytic solutions.
- **Unified data warehouse**: Serves as a unified platform for various analytic workloads, simplifying tech stacks. For instance, Haidilao replaced its architecture of Spark, Hive, Kudu, HBase, and Phoenix with Apache Doris.

2 Apache Doris Features

2.1 Doris Architecture

Apache Doris boasts a straightforward yet elegant architecture, characterized by its simplicity and efficiency, which is achieved with only two types of processes. This design choice ensures ease of management and scalability while maintaining robust performance.

- **Frontend (FE):** The FE process is the system's brain, handling user requests, query parsing, and query planning. It manages metadata to ensure accurate schema, table, and other element synchronization across the cluster. Additionally, the FE oversees node management, coordinating cluster nodes for smooth, efficient operation.
- **Backend (BE):** The BE process, on the other hand, is the muscle of the architecture. It is tasked with data storage, ensuring that data is securely and efficiently stored across the distributed system. The BE also handles query execution, processing the queries parsed and planned by the FE and retrieving the required data for the user.

Apache Doris is scalable, reliable, and efficient. It supports hundreds of machines and petabytes of data, maintaining performance and reliability. Its design simplifies operation and is cost-effective for large-scale data management.

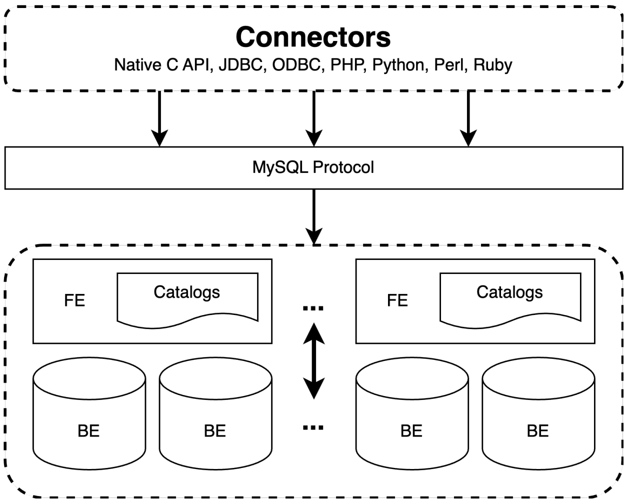


Figure 3: *Apache Doris Architecture*

Query Engine: Doris features an MPP-based query engine designed for parallel execution both between and within nodes. It supports distributed shuffle joins for large tables, enabling more efficient handling of complex queries.

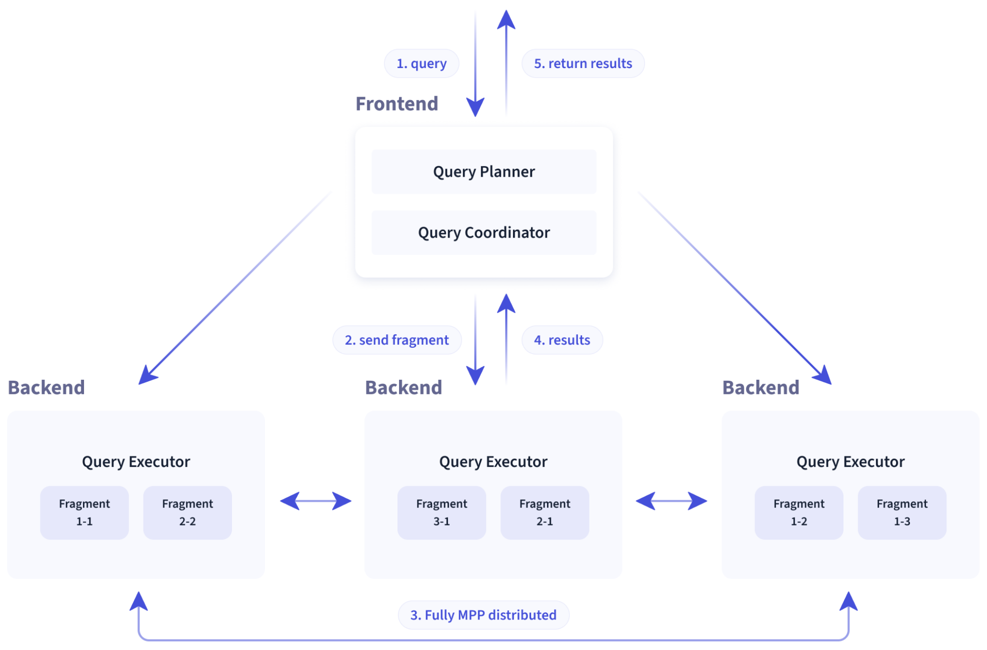


Figure 4: *Apache Doris Query Engine*

- The Doris query engine is fully vectorized, with all memory structures in a columnar format, reducing virtual function calls, increasing cache hit rates, and efficiently using SIMD instructions. This results in a 5 to 10 times performance boost in wide table aggregation scenarios compared to non-vectorized engines.
- Doris uses adaptive query execution to dynamically adjust execution plans based on runtime statistics, generating runtime filters that reduce processed data and enhance join performance. These filters, including In, Min/Max, and Bloom Filter, are pushed to the lowest-level scan node on the probe side.
- The Doris query optimizer combines Cost-Based Optimization (CBO) and Rule-Based Optimization (RBO). RBO handles constant folding, subquery rewriting, and predicate push-down, while CBO manages join reordering. Continuous improvements in the CBO enhance the accuracy of statistics collection, inference, and the cost model.

2.2 Cost-based Optimizer

Structure

The new optimizer called **Nereids** - a brand new planner. This new optimizer in Apache Doris introduces an advanced approach to query optimization by using both *Rule-Based Optimization*

(RBO) and *Cost-Based Optimization (CBO)*.

- Rule-Based Optimization (RBO):
  - RPO applies a predefined set of rules to transform and optimize queries. These rules are based on heuristics and do not consider the data distribution or query cost
  - Using scenarios: simple queries, known patterns, performance consistency priority, resource constraints.
- Cost-Based Optimization (CBO):
  - CBO use statistical information about the data (table sizes, data distribution, etc) to estimate the cost of different query execution plans. Then, select the plan with the lowest estimated cost.
  - Using scenarios: complex queries, large dataset, dynamic query pattern, performance optimization priority.

Furthermore, the CBO of the new optimizer is based on the advanced cascades framework, uses richer data statistics, and applies a cost model with more scientific dimensions. This makes the new optimizer more handy when faced with multi-table join queries. The workflow of combining RBO and CBO in Apache Doris:

- Step 1: Initial Parsing and RBO
- Step 2: Pattern matching
- Step 3: Transition to CBO
- Step 4: Cost Estimation and Plan Selection
- Step 5: Plan Refinement

This combined approach leverages RBO for quick optimizations and CBO for detailed analysis, ensuring efficient and scalable query execution.

In addition, collecting statistics helps the optimizer understand data distribution characteristics. When performing Cost-Based Optimization (CBO), the optimizer uses these statistics to calculate the selectivity of predicates and estimate the cost of each execution plan. This allows for the selection of more optimal plans, significantly improving query efficiency. Currently, the following information is collected for each column: `row_count` , `data_size` , `avg_size_byte` , `ndv` , `min` , `max` , `null_count`

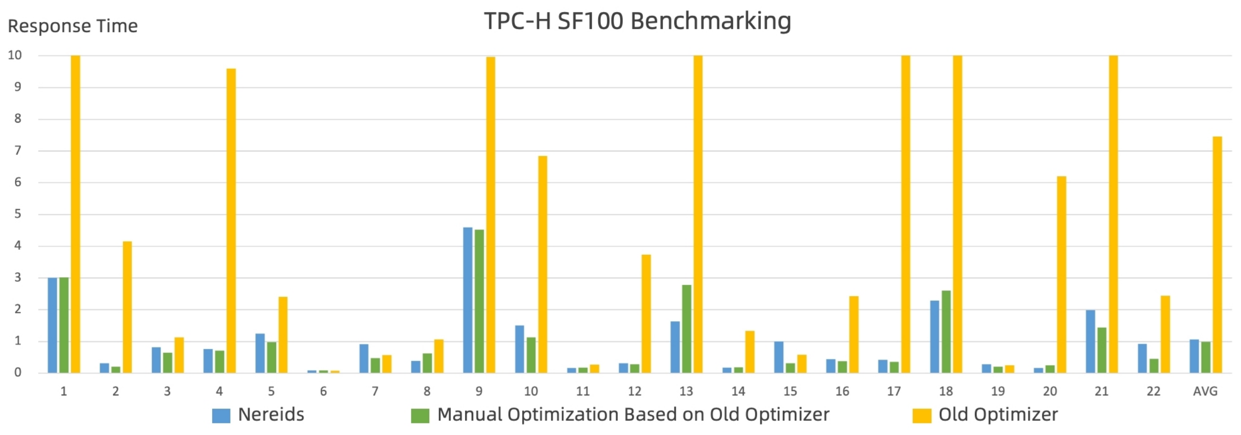


Figure 5: *Apache Doris Query Engine*

Advantages of the new optimizer

- Faster: TPC-H SF100 query speed comparison shows the new optimizer using original SQL and collected statistics performs similarly to the old optimizer with hand-tuned SQL, without needing manual optimization.
- Robust: The new optimizer completes all optimization rules on the logical execution plan tree, making it more reasonable and unified compared to the old optimizer's internal data structure. This reduces logic errors, particularly in subquery processing.
- Flexible: The new optimizer's modern architectural design allows for easily extendable optimization rules and processing stages, enabling quicker responses to user needs.

2.3 MPP-based Architecture

Massively Parallel Processing (MPP) architecture is a distributed computing approach where a large task is divided into smaller tasks that can be executed in parallel across multiple processing units or nodes. Each node typically operates independently, processing a portion of the data, and then the results are combined to produce the final output. MPP architectures are commonly used in data warehousing and analytics systems to handle large-scale data processing tasks efficiently.

In general, Apache Doris follows an MPP-based architecture follows several principles listed below:

- Distributed Storage: Doris stores data across multiple nodes in a distributed manner, ensuring data locality and enabling parallel processing. Data is partitioned and replicated across nodes to achieve fault tolerance and load balancing.
- Parallel Query Execution: Queries are parallelized across multiple nodes, with each node

processing a portion of the data in parallel. This parallelism allows Doris to handle complex analytical queries efficiently, even on large datasets.

- Query Coordination: Doris employs a distributed query coordination mechanism to distribute query execution plans to individual nodes and coordinate their execution. The query coordinator optimizes query plans and ensures that data is processed in parallel across the cluster.
- Scalability and Fault Tolerance: Doris scales horizontally by adding more nodes to the cluster, allowing it to handle increasing data volumes and user concurrency. Fault tolerance is achieved through data replication and distribution, ensuring data reliability and system resilience.
- Data Consistency: Doris maintains data consistency across nodes through mechanisms such as distributed transactions and replication consistency protocols. This ensures that query results are accurate and reliable, even in distributed environments.

By leveraging MPP principles, Apache Doris can efficiently process large-scale analytical workloads, provide real-time insights, and scale to meet the growing demands of modern data analytics applications.

2.4 Fully vectorized Query engine

The Doris query engine is fully vectorized, with all memory structures laid out in a columnar format. Doris delivers a 5-10 times higher performance in wide table aggregation scenarios than non-vectorized engines.

Vectorized execution refers to processing data in batches or vectors rather than row by row. This approach allows the CPU to take advantage of modern hardware features like *SIMD (Single Instruction, Multiple Data*) to perform the same operation on multiple data points simultaneously.

In Doris, query engine processes data in columnar batches, applying operations like filtering, aggregation, and joins to entire vectors at once. With only vectorized query engine, it can be achieved with a certain number of benefits:

- This approach optimizes CPU usage, maximize CPU throughput and speeds up query execution, particularly for complex analytical queries.
- Reduced overhead: Handling data in larger chunks reduces the overhead of function calls and loop controls, leading to faster execution.
- Cache and Processing efficiency: Takes advantage of these contiguous memory blocks to perform operations on data loaded into CPU caches, reducing cache misses and improving overall execution speed.

2.5 Columnar-format & Hybrid memory structure

Apache Doris, being a column-oriented database, allows for efficient data compression and sharding. However, this may not be ideal for high-concurrency point queries in customer-facing services, as it can amplify I/O operations.

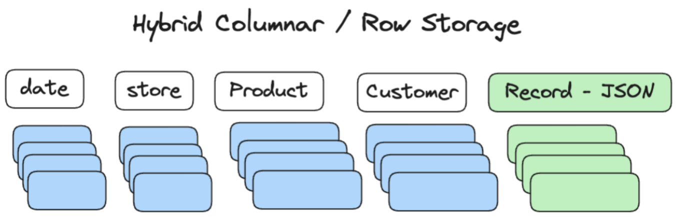
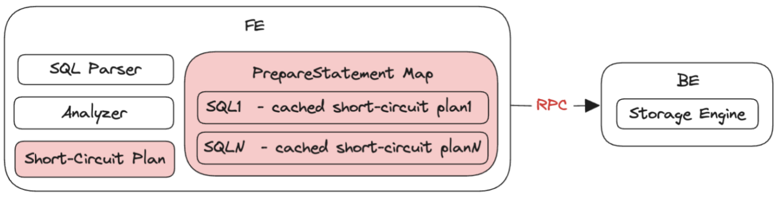


Figure 6: *Column-oriented database*

To address this, Apache Doris supports hybrid storage, combining row and columnar storage. For simple point queries, it uses a short circuit plan to avoid the overhead of the query planner. Additionally, prepared statements pre-compute and cache SQL statements, reducing the overhead of SQL parsing for high-concurrency point queries.



2.6 Short Circuit

In the context of query execution, a short circuit plan is a performance optimization technique. Short circuit is particularly useful for simple queries, such as point queries. When a simple query is executed, instead of going through the entire query planning process, Doris directly executes the query without involving the query planner. This bypasses unnecessary overhead and improves query execution speed.

2.7 Pipeline Execution

The pipeline execution engine, introduced experimentally in Doris version 2.0, aims to replace the current volcano model-based execution engine. It fully utilizes multi-core CPU capabilities and limits query threads to address thread bloat.

The traditional Doris execution engine has limitations in multi-core scenarios, such as:

- Inadequate use of multi-core power, requiring manual parallelism settings.

- Each query instance uses a thread, leading to thread pool overload, pseudo-deadlocks, and logical deadlocks.
- Blocking operations occupy thread resources and depend on OS thread scheduling, causing high overhead.

The new engine transforms from a pull-based logic-driven process to a push-based data-driven one. Asynchronous blocking reduces thread-switching overhead, optimizing CPU use and enhancing query performance in mixed-load scenarios as shown in the figure below.

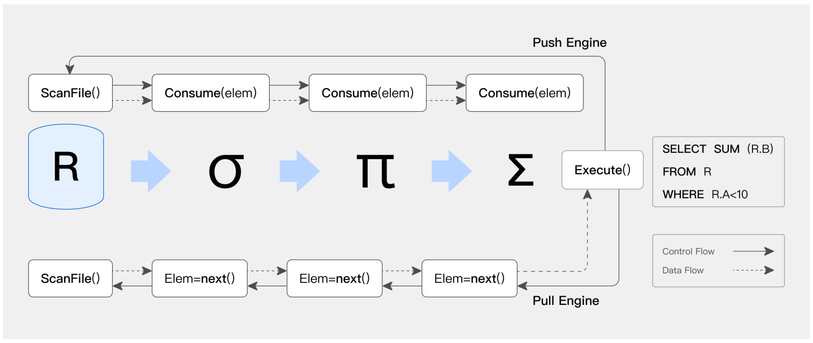


Figure 7: *Pipeline Execution Engine*

2.8 Index

Indexes in Doris enhance the speed of data filtering and retrieval. Doris supports two main types of indexes:

1. **Built-in Smart Indexes**: includes Prefix indexes and ZoneMap indexes.
2. **User-Created Secondary Indexes**: includes Bloomfilter indexes and Bitmap indexes.

ZoneMap index is automatically maintained for each column in the column storage format, containing information like Min/Max values and the number of `Null` values. This index is transparent to the user.

Prefix Index

- Doris does not support creating indexes on arbitrary columns as traditional databases do. Designed as an OLAP database with MPP architecture, Doris handles large data volumes by increasing concurrency.
- Doris stores data in a structure similar to an *Sorted String Table (SSTable)*, which is an ordered data structure that can be sorted by specified columns. This allows efficient lookup using sorted columns as conditions.
- In the Aggregate, Unique, and Duplicate data models, data is sorted and stored according to columns specified in the table building statements (`AGGREGATE KEY` , `UNIQUE KEY` ,

`DUPLICATE KEY`). The prefix index leverages this sorting to quickly query data based on a given prefix column.

BloomFilter Index, introduced by Bloom in 1970, is a fast search algorithm using multi-hash function mapping. It's ideal for quickly checking if an element is part of a set, though it doesn't guarantee 100% accuracy. Key features include:

- Space-Efficient: It uses minimal space to determine if an element is in a set.
- Probabilistic Results: It indicates if an element may exist or definitely does not exist.
- False Positives: It may incorrectly indicate an element exists.

A BloomFilter consists of a long binary bit array, initially all zeros, and several hash functions. When querying an element, it is hashed into the bit array, and all corresponding positions are set to 1. For example, with a BloomFilter of size $m = 18$ and $k = 3$ hash functions, elements x, y, and z are hashed into the bit array. When querying element w, if any bit is 0, w is not in the set.

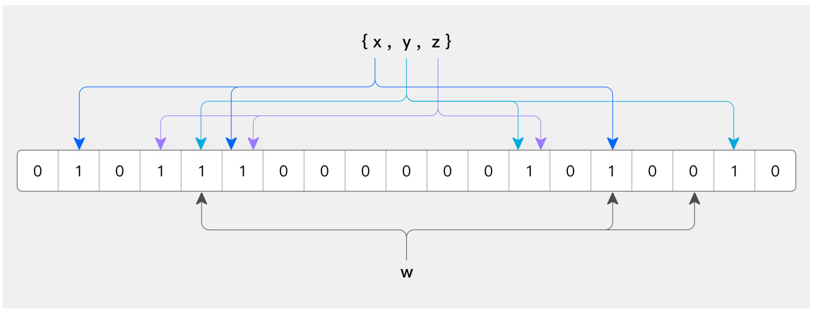


Figure 8: *BloomFilter Index Idea*

Bitmap Index: users can speed up queries by creating a bitmap index This document focuses on how to create an index job, as well as some considerations and frequently asked questions when creating an index.

2.9 Materialized view

In Apache Doris, a **materialized view** is a database object that stores the result set of a query as a physically persistent table. It allows users to pre-compute and store the results of complex queries, thereby improving query performance and reducing the overhead of executing the query each time it is needed.

Advantages

- Materialized views in Apache Doris significantly enhance query performance for frequently used sub-query results.
- Doris automatically maintains the materialized view data, ensuring consistency with the base

table regardless of new imports or delete operations, without requiring additional manual maintenance efforts.

- During querying, Doris intelligently selects the optimal materialized view and retrieves data directly from it. This streamlined process eliminates the need for manual intervention and ensures efficient data access, contributing to improved overall system performance.

Limitations

- Materialized views, aggregate function parameters only support single columns (multi-column support added in 2.0).
- Deleting data requires the conditional column to exist in the materialized view; otherwise, the view must be deleted first.
- Having too many materialized views on a single table can slow down data imports due to synchronous updates.
- Materialized views cannot have the same column with different aggregate functions simultaneously (supported in 2.0).
- Materialized views in the Unique Key data model can only reorder columns, not perform aggregation.

2.10 Projection

A projection defines a subset of table columns for queries, allowing users to include or exclude columns. Each projection has its own schema, specifying data types and structures. Users can customize schemas for better query performance and storage efficiency.

Projections maintain statistics (min, max, distinct count) to help Doris's query optimizer create efficient plans. Types of projections in Apache Doris include Primary Key, Covering, Aggregation, and Partitioned projections.

Benefits of Projections in Doris:

- Query Performance: By selecting only the necessary columns for a projection, users can improve query performance by reducing data transfer and processing overhead.
- Storage Efficiency: Projections allow users to define custom schemas and column subsets, optimizing storage efficiency by storing only the required columns on disk.
- Schema Evolution: Projections support schema evolution by allowing users to define different schemas for different use cases or query patterns. This flexibility enables seamless adaptation to changing business requirements.
- Query Flexibility: Users can create multiple projections with different column subsets to

support various query patterns and analytical use cases. This flexibility enables efficient data access and analysis across diverse workloads.

Overall, projections in Apache Doris define subsets of columns from a table that are available for querying. By selecting appropriate column subsets and defining custom schemas, users can optimize query performance, storage efficiency, and query flexibility to meet diverse analytical requirements. Projections play a crucial role in optimizing data access and analysis in Doris-based analytical systems.

2.11 Join optimization

Doris supports two physical join operators: *Hash Join* and *Nested Loop Join*.

- Hash Join: creates a hash table on the right table based on the join column, allowing the left table to perform join calculations in a streaming manner. It is limited to equivalent joins.
- Nested Loop Join: uses two for-loops, suitable for non-equivalent joins (e.g., greater than or less than) but has poor performance.

Doris Shuffle Methods

- Broadcast Join:
  - Sends the entire right table to each node of the left table.
  - Supports both Hash Join and Nested Loop Join.
  - Network overhead: N * T(R).
- Shuffle Join:
  - Splits and sends data based on hash values of the join column.
  - Supports only Hash Join.
  - Network overhead: T(S) + T(R).
- Bucket Shuffle Join:
  - Utilizes pre-bucketed data for joining, minimizing data movement.
  - Supports Hash Join.
  - Network overhead: T(R).
- Colocate Join:
  - Joins pre-partitioned data directly without shuffling.
  - Supports Hash Join.

- Network overhead: 0.

Runtime Filter Join Optimization

- Runtime Filter Types:
  - IN: Pushes a hashset to the scanning node. Effective but limited to small right tables.
  - BloomFilter: Constructs and pushes a BloomFilter to the scanning node. Suitable for various types.
  - MinMax: Uses range-based filtering. Effective for numeric columns.
- Usage: Effective when the right table is small and the join filters out most data from the left table. Enhances performance by reducing the data processed at the join layer.

Join Reorder

- Principle: Prioritizes joins that generate smaller intermediate results by joining larger tables with smaller ones and applying conditional joins early.
- Optimization: Use session variables to enable cost-based join reorder for improved join performance.
- `set enable_cost_based_join_reorder = true`

Join Optimization Tips

- Use Same Type Columns: reduces data casting and speeds up calculations.
- Choose Key Columns: enhances delayed materialization and filtering efficiency.
- Prefer Co-location for Large Tables: reduces network overhead.
- Use Runtime Filters Judiciously: effective for high join filtering rates but has overhead costs.
- Ensure Join Order Rationality: aim for large left tables and small right tables; prefer Hash Join over Nested Loop Join. Use SQL Rewrite and Join Hints if necessary.

Case Practices

- Case 1: Enabling cost-based join reorder reduced join time from 14 seconds to 4 seconds by correctly adjusting table order.
- Case 2: Switching from IN to BloomFilter reduced query time from 44 seconds to 13 seconds by filtering 99% of data.
- Case 3: Using a Join Hint for Shuffle Join reduced time from 3 minutes to 7 seconds by correcting an unreasonable join order.

2.12 Smart caching

Apache Doris offers an advanced caching mechanism tailored to optimize query performance for read-heavy data analysis scenarios. This smart caching strategy ensures data consistency, minimizes additional system complexity, and improves query efficiency without external cache components.

Demand Scenarios

Smart caching is particularly useful in the following scenarios:

- High Concurrency: Supports a high number of simultaneous queries, which a single server might struggle with.
- Complex Dashboards: Handles complex charts and large-screen applications with multiple queries per page.
- Trend Analysis: Efficiently processes queries over large date ranges, such as daily user trends.
- Repeated Queries: Manages scenarios where users repeatedly refresh pages, generating redundant queries.

Traditional application-layer solutions, like using Redis for caching, often suffer from data inconsistency, low hit rates, and additional costs. Doris' built-in smart caching addresses these issues with a partition cache strategy that prioritizes data consistency and refines cache granularity to improve hit rates.

Solution Features

- Data Consistency: Cache invalidation is controlled via versioning, ensuring cached data matches the queried data.
- No Additional Costs: Cache results are stored in BE's memory, with adjustable cache memory size.
- Efficient Caching Strategy: Utilizes SQL Cache and consistent hashing with an improved LRU algorithm to handle BE node changes.

SQL Cache Usage

The SQL Cache supports internal OlapTables and external Hive tables, caching only completely consistent SQL statements.

Configuration Parameters:

- Maximum Rows per Query Result:

```
vim fe/conf/fe.conf
cache_result_max_row_count=3000
```

- Maximum Data Size per Query Result (default 30M):

```
vim fe/conf/fe.conf
cache_result_max_data_size=31457280
```

- Cache Validity Interval (default 30 seconds):

```
vim fe/conf/fe.conf
cache_last_version_interval_second=30
```

- Cache Memory Limits:

```
vim be/conf/be.conf
query_cache_max_size_mb=256
query_cache_elasticity_size_mb=128
```

The `query_cache_max_size_mb` sets the upper memory limit, while `query_cache_elasticity_size` allows for additional memory stretch. When the cache size exceeds the combined limit, cleanup processes maintain memory usage within the specified bounds.

Monitoring and Optimization

Doris provides several metrics to monitor and optimize caching:

- FE Monitoring:
  - Number of tables in query
  - Number of Olap tables in query
  - Cache mode hits and misses
- BE Monitoring:
  - Cache memory size
  - Number of cached SQL queries

These metrics, viewable in Grafana, help in adjusting cache parameters to meet business goals. Apache Doris' smart caching strategy enhances performance in read-heavy scenarios by ensuring data consistency, reducing query overhead, and eliminating the need for external cache components. By finely tuning cache parameters and monitoring performance metrics, users can achieve significant improvements in query efficiency and system resource utilization.

3 Core capabilities

3.1 High concurrency Data services

Doris uses a columnar storage engine, which can cause high random read IO in high-concurrency scenarios when retrieving entire rows, especially from wide tables. Simple queries, like point queries, suffer from the heavy query engine and planner. To address these issues, Doris introduces row storage, a short query path in the FE's query plan, and `PreparedStatement`. The FE, written in Java, acts as the SQL query access layer, and parsing SQL can lead to high CPU overhead for high-concurrency queries.

Use cases:

- Customer-facing (e.g. e-commerce delivery checking): A substantial volume of users concurrently seek information from the system, with each user seeking only a one or two data rows.
- Machine-facing: real-time risk control, IoT, etc.

Solutions: Hybrid storage, Short circuit plan, PreparedStatment.

3.2 Service availability & Data reliability

3.2.1 High availability

Architecturally, Doris has two processes which are front-end and back-end shown in below figure.

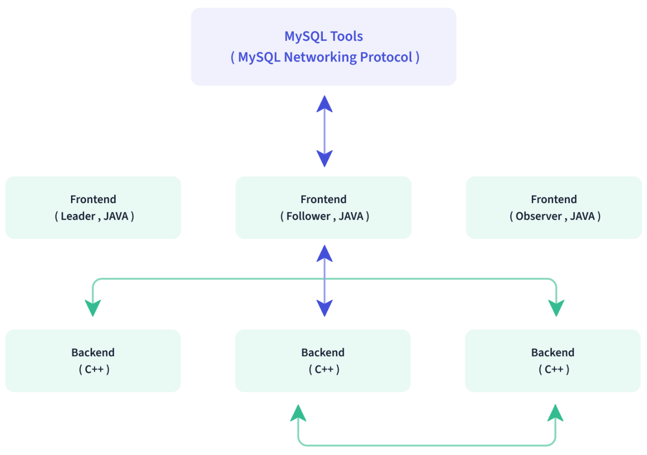


Figure 9: *Architecture: Front-end and Back-end*

- The front-end nodes manage the metadata
- The back end nodes execute the queries, do auto data balancing and auto restoration.
- Both the front-end and back-end processes are scalable, therefore it totally allow upgrading or scaling cluster without any interruptions for the whole services and business.

3.2.2 Data Backup & Recovery

For data reliability, Apache Doris supports snapshot backup and snapshot restoration both at the table level and partition level (data backup to HDFS or object storage).

3.2.3 Cross Cluster Replication

For Enterprise users, especially those in finance or e-commerce, they would need to back up their clusters or or even their entire Data Center and just in case of Force Major so in DS 2.0 we have *Cross cluster Replication (CCR).*

Cross-Cluster Replication (CCR) is a mechanism that enables the replication of data across multiple geographically or logically separated database clusters. This ensures that data is duplicated across different clusters to achieve high availability, fault tolerance, disaster recovery, and load balancing.

There are 3 main capabilities that CCR bring to us:

- **Disaster recovery** is the ability to quickly restoration of data and services. In the event of a failure in one region, the other regions can provide uninterrupted service.
- **Read-write separation** is a strategy that enhances the performance and scalability of distributed database systems like Apache Doris by segregating read and write operations across different clusters. There are 2 kinds of clusters:
  - Primary Cluster (Write Cluster): is designated for handling all write operations, including inserts, updates, and deletes. It ensures that all data modifications are initially processed and stored in a central, authoritative location.
  - Secondary Clusters (Read Clusters): are designated for handling read operations. Data from the primary cluster is replicated to these clusters, providing multiple points for querying and data retrieval without impacting the write operations
- **Separated cluster upgrading**: when you need to upgrade your clusters, Doris allows you to pre-create a backup cluster so we can run things on the backup cluster to for preventing incompatibility issues or bugs.

With the separation in read and write, it can:

- Improved performance: Read and write operations are handled by separate clusters, reducing contention and enhancing efficiency.
- Scalability: Add more read clusters to support increasing query loads and concurrent users.
- Reduced latency: Direct read requests to the nearest read cluster for faster response times.
- High Availability: Read clusters can continue serving requests even if the primary cluster is down.
- Load Balancing: Distribute read requests across multiple clusters to prevent bottlenecks and optimize resource use.

3.3 High-performance data ingestion

Data ingestion in Apache Doris involves the process of importing and integrating data from various sources into the Doris system for storage, analysis, and querying. Apache Doris supports multiple data ingestion methods to accommodate different types of data and ingestion scenarios.

Apache Doris provides a wide range of methods for you to choose:

- Broker Load: Suitable for loading large datasets from external storage systems like HDFS, S3, or other distributed file systems. It supports parallel loading and data transformation during the loading process.
- Stream Load: Designed for real-time data ingestion, typically from data streams or message queues like Kafka. It uses HTTP protocol to push data into Doris and supports continuous data loading with minimal latency.
- Routine Load: Automatically and continuously loads data from streaming sources like Kafka. It periodically pulls data and ensures data consistency and reliability.
- Insert Load: Uses SQL INSERT statements to load data directly into Doris tables. Suitable for small to moderate datasets or real-time data inserts from applications.
- Flink-Doris Connector: Integrates with Apache Flink for real-time data processing and ingestion. The connector allows Flink to write processed data directly into Doris, ensuring low-latency data availability for analytics.
- Spark-Doris Connector: Integrates with Apache Spark for batch and stream processing. The connector facilitates writing data from Spark jobs into Doris.

Batch writing in Apache Doris is designed for efficiently ingesting large volumes of data at once, typically from offline sources such as distributed file systems, databases, or data lakes. This method is well-suited for scenarios where data doesn't need to be available in real-time but rather in periodic intervals, such as daily or hourly updates:

- Spark Load: leverage Spark resources to pre-process data from HDFS and object storage
- Broker Load: No need to deploy broker, supports HDFS and S3 protocol
- Data integration for data lakehouse: :
  - `INSERT INTO <external_table> SELECT FROM <external _table>`
  - Storage systems (S3, HDFS, local files)
  - Data lakes (Iceberg, Hudi, Hive)
  - Databases (MySQL, Oracle, Elasticsearch, etc)

Besides, Apache Doris is working with some data tools, which are very diverse: DataX, X2Doris, etc. Read-write separation is also used in data ingesting process.

3.4 Data Update

Use cases & requirements: E-commerce order analysis, user profile update, data deletion, data overwrite.

Update in Primary Key (Unique) Model: starting from Doris 2.0, Doris primary key (unique) model supports both Merge-on-Read (MoR) and Merge-on-Write (MoW) storage modes.

- Merge-on-Read: is optimized for write operations with low-frequency batch updates.
- Merge-on-Write: is suitable for real-time writing optimized for faster analysis performance. In actual tests, the analysis performance of MoW storage can be 5-10 times faster than MoR with light merge upon writing.

Apache Doris support most updates: upsert, partial column update, conditional update/deletion, partition overwrite, etc. Upsert is supported in all data ingestion methods. In concurrent updates, the updating order is decided by order of transaction commits or the Sequence column.

Non-primary key tables (duplicate tables, aggregate tables)

- Aggregate tables implements partial column update by `replace_if_not_null`.
- All table model supports data deletion based on specified predicated and some date expressions (`curdate()`).

  `DELETE FROM mytable PARTITION t1 WHERE k1 = 3;`

  `DELETE FROM mytable PARTITIONS(p1,p2) WHERE k1 >= 3 AND k2 = "abc", AND t = curdate()`

3.5 Semi-structured Data analysis

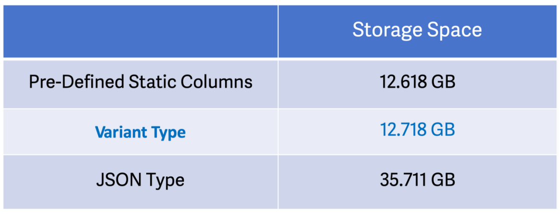
Semi-structured data, characterized by flexible formats such as XML, JSON, and log files, is essential in various industry scenarios:

- **E-commerce:** Stores user reviews as semi-structured data for sentiment analysis and behavior pattern mining.
- **Telecommunications:** Requires schemaless support for network data and nested JSON data.
- **Mobile Applications:** Records user behavior, adapting easily to changes in user attributes without frequent schema modifications.
- **IoT and IoV Platforms:** Receives real-time data from sensors for monitoring, fault alerting, and route planning.

Apache Doris, an open-source real-time data warehouse, improves semi-structured data processing with version 2.1.0, introducing the Variant data type. Unlike JSON, Variant fields can hold integers, strings, boolean values, and combinations without pre-defined schemas. This Schema-on-Write method supports both Variant and static columns, enhancing storage and query flexibility.

Advantages of Variant over JSON: Test environment consists of 16-core, 64GB AWS EC2 instance, 1TB ESSD.

1. **Storage Efficiency:** Variant columns require 65% less storage space compared to JSON.

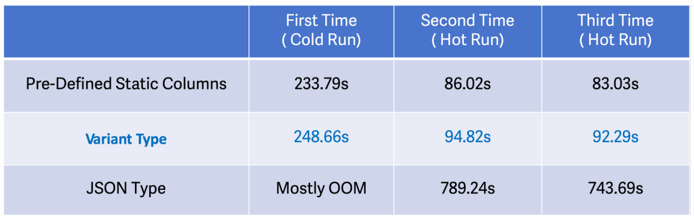


	Storage Space
Pre-Defined Static Columns	12.618 GB
Variant Type	12.718 GB
JSON Type	35.711 GB

Figure 10: *Storage space comparison*

Variant uses one-third the space of JSON, similar to static columns.

2. **Query Performance:** Queries on Variant columns are 8 times faster than those on JSON columns, though slightly slower (10%) than pre-defined static columns.



	First Time (Cold Run)	Second Time (Hot Run)	Third Time (Hot Run)
Pre-Defined Static Columns	233.79s	86.02s	83.03s
Variant Type	248.66s	94.82s	92.29s
JSON Type	Mostly OOM	789.24s	743.69s

Figure 11: *Query performance comparison*

Variant columns perform significantly better than JSON, with most JSON queries failing due to out-of-memory errors in cold runs.

Apache Doris 2.1.0's Variant data type provides an efficient and flexible solution for handling semi-structured data, offering substantial improvements in both storage and query performance.

4 Comparison with other tools

4.1 Why and When to Use Apache Doris?

Apache Doris is a next-generation real-time data warehouse designed to handle both analytical and transactional processing workloads efficiently. It excels in scenarios where low-latency query performance and high concurrency are crucial. Ideal use cases for Apache Doris include:

- Real-Time Analytics: Doris is optimized for real-time data ingestion and analysis, making it suitable for environments requiring up-to-the-minute insights.
- High Concurrency: Its architecture supports high concurrency, making it ideal for use cases where numerous users or applications need to access data simultaneously.
- Hybrid Workloads: Doris can efficiently handle hybrid transactional/analytical processing (HTAP) workloads, allowing businesses to perform complex queries on live transactional data without performance degradation.

4.2 Comparison Doris with Similar Tools

Clickhouse

- Architecture: Clickhouse is a columnar database management system (DBMS) designed for online analytical processing (OLAP). It uses a distributed architecture optimized for query speed and efficiency.
- Performance: Clickhouse offers impressive read performance and is particularly strong in environments with heavy read and analytical query demands.
- Use Cases: Ideal for applications requiring fast query speeds over large datasets, such as web analytics, log analysis, and real-time reporting.
- ***Advantages:*** Superior query performance for large datasets, highly efficient data compression.
- ***Disadvantages:*** Limited support for complex transactions, less suited for write-intensive workloads compared to Doris.

Presto (Trino)

- Architecture: Presto, now known as Trino, is a distributed SQL query engine designed for running interactive analytic queries against data sources of all sizes.
- Performance: Trino is designed for fast analytics across large-scale data warehouses and data lakes. It excels in federating queries across various data sources.
- Use Cases: Suitable for querying data from multiple heterogeneous data sources, including

HDFS, AWS S3, and traditional databases.

- ***Advantages:*** High flexibility in querying multiple data sources, strong community support, and extensive plugin ecosystem.
- ***Disadvantages:*** Not a standalone data store; relies on underlying storage systems, which may introduce performance bottlenecks depending on the storage system used.

4.3 Other Advantages and Disadvantages of Apache Doris

Advantages	Disadvantages
- **Unified Storage and Processing:** Doris integrates storage and processing, simplifying data management and reducing the need for complex data pipelines. - **Efficient Data Compression:** Advanced data compression techniques lead to reduced storage costs and improved query performance. - **Scalability:** Easily scalable architecture allows for handling increasing data volumes and user demands. - **Ease of Use:** User-friendly interface and SQL compatibility make it accessible for users familiar with traditional relational databases. - **Open Source:** Being an open-source project, Doris benefits from community contributions and continuous improvements.	- **Maturity:** As a relatively new project compared to Clickhouse and Trino, Doris may have fewer features and integrations. - **Ecosystem:** The surrounding ecosystem and community support, while growing, may not be as extensive as more established tools. - **Complexity in Setup:** Initial setup and configuration can be complex, requiring expertise in distributed systems.

In conclusion, Apache Doris is a powerful tool for real-time data warehousing needs, particularly when low-latency and high concurrency are critical. It offers a unified approach to handling both analytical and transactional workloads but may require careful consideration of its relative maturity and ecosystem compared to other tools like Clickhouse and Trino.

5 Typical Case-study - Real-time writes from Flink to Apache Doris

5.1 Background

With the growing demand for real-time analysis, timely data is crucial for enterprise operations. Real-time data warehouses are vital for extracting valuable information, enabling quick data feedback, faster decision-making, and better product iterations.

In this context, Apache Doris excels as a high-performance, user-friendly real-time MPP analytic database supporting various data import methods. Integrated with Apache Flink, it allows rapid import of unstructured data from Kafka and CDC from databases like MySQL. Apache Doris offers sub-second analytic queries, ideal for real-time scenarios such as multi-dimensional analysis, dashboards, and data serving.

5.2 Challenge

Ensuring high end-to-end concurrency and low latency in real-time data warehouses presents several challenges, such as:

- How to maintain second-level end-to-end data synchronization?
- How to promptly ensure data visibility?
- How to address the issue of writing small files under high concurrency conditions?
- How to guarantee end-to-end Exactly-Once semantics?

To address these challenges, an in-depth study of business scenarios using Flink and Doris for real-time data warehouses was conducted. By identifying users' pain points, targeted optimizations were implemented in Doris version 1.1, enhancing user experience, system stability, and resource consumption efficiency.

5.3 Optimization

5.3.1 Streamming Write

The initial practice of Flink Doris Connector is to cache the data into the memory batch after receiving data.The method of data writing is saving batches, and using parameters such as `batch.size` and `batch.interval` to control the timing of Stream Load writing at the same time.

Traditional Batch-Based Stream Load

Characteristics: Data is accumulated into batches before being sent to Doris. Parameters like `batch.size` determine how much data is accumulated before initiating a Stream Load request. This approach will run:

- Stable with reasonable parameters: When parameters are well-tuned (e.g., batch size, frequency of loads), the system runs stably.
- Problems with unreasonable parameters:
  - Frequent Stream Load: If parameters are set too low, it results in frequent Stream Load requests. This can overwhelm the system, causing excessive version errors (-235).
  - Untimely Compaction: Frequent loads can lead to inefficient data compaction, further degrading performance.
  - Out-of-Memory (OOM) Errors: Setting the batch.size too large in an attempt to reduce the frequency of loads can cause OOM errors due to excessive memory consumption.

Streaming write is introduced to solve this problem:

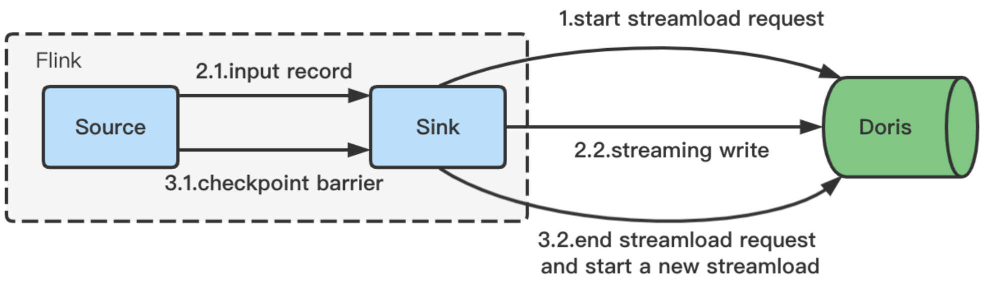

1.start streamload request
Flink
2.1.input record
Source
Sink
Doris
2.2.streaming write
3.1.checkpoint barrier
3.2.end streamload request and start a new streamload

Figure 12: *Streaming write with Flink-Doris connector*

- After the Flink task starts, the Stream Load HTTP request will be asynchronously initiated.
- When the data is received, it will be continuously transmitted to Doris through the Chunked transfer encoding of HTTP.
- HTTP request will end at Checkpoint and complete the Stream Load writing . The next Stream Load request will be asynchronously initiated at the same time.
- The data will continue to be received and the follow-up process is the same as above.

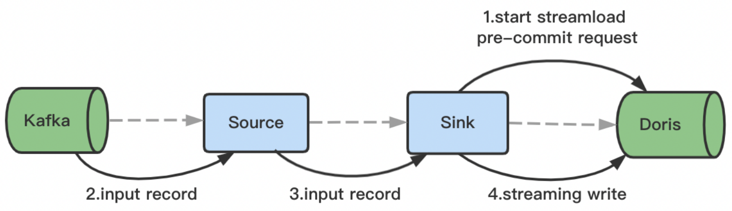
Chunked transfer encoding allows data to be sent in a series of chunks without defined total size of the data beforehand. This mechanism ensures that data can start being processed by Doris even as more data is being transmitted, enabling efficient real-time data ingestion.

Flink's checkpointing mechanism ensures fault tolerance and data consistency. Each checkpoint marks a safe point in the data stream, ensuring that all prior data is securely written and acknowledged by Doris before proceeding.

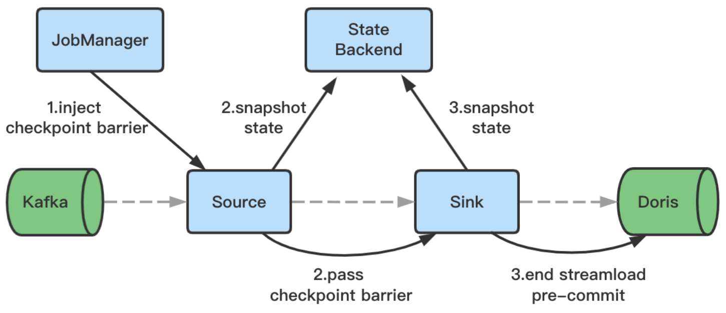
5.3.2 Exactly-Once

Exactly-Once means that data will not be reprocessed or lost, even machine or application failure. Flink supports the End-to-End's Exactly-Once scenario a long time ago, mainly through the two-phase commit protocol to realize the Exactly-Once semantics of the Sink operator.

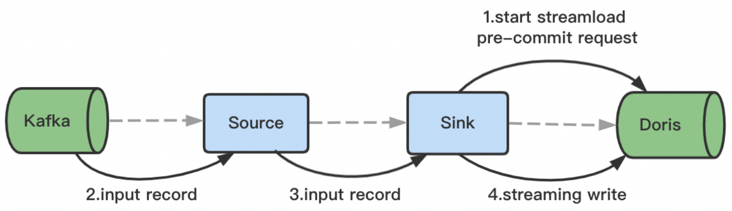
On the basis of Flink's two-stage submission, with the help of Doris 1.0's Stream Load two-stage submission, Flink Doris Connector implements Exactly Once semantics:



- **Step 1**: Stream Load PreCommit: When the Flink task starts, it initiates a Stream Load PreCommit request, opening a transaction. Data is sent continuously to Doris via HTTP chunked transfer.


JobManager
State Backend
1.inject checkpoint barrier
2.snapshot state
3.snapshot state
Kafka
Source
Sink
Doris
2.pass checkpoint barrier
3.end streamload pre–commit

- **Step 2**: Checkpoint: At each checkpoint, the HTTP request completes, and the transaction is set to preCommitted. The data is written to BE but remains invisible to users.



- **Step 3**: Commit: After the checkpoint, a Commit request is made, changing the transaction status to Committed, making the data visible to users.
- **Step 4**: Recovery: If the Flink application fails and restarts from a checkpoint, and the last transaction was preCommitted, a rollback request sets the transaction to Aborted.

Based on the above , Flink Doris Connector can be used to realize real-time data storage without loss or weight.

5.3.3 Second- Level Data Synchronization

Second-Level Data Synchronization in Apache Doris refers to the system's capability to synchronize data changes across the cluster within a second, ensuring that data remains consistent and up-to-date nearly in real-time. This is crucial for applications requiring low-latency data updates and real-time analytics.

End-to-end second-level data sync and real-time visibility of data in high concurrent write scenarios require Doris to have the following capabilities:

- Transaction processing capability: ensure the basic ACID characteristics, and support Flink's second-level data sync in high concurrency scenarios.
- Rapid aggregation capability of data versions: the continuous high-concurrency small file writing scenario extremely tests the real-time ability anddata merging performance.

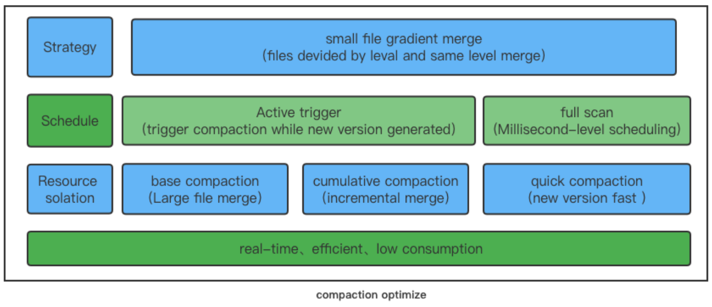

Strategy
small file gradient merge
(files devided by leval and same level merge)
Schedule
Active trigger
(trigger compaction while new version generated)
full scan
(Millisecond-level scheduling)
Resource solation
base compaction
(Large file merge)
cumulative compaction
(incremental merge)
quick compaction
(new version fast)
real-time、efficient、low consumption
compaction optimize

Figure 13: *Compaction Optimizer*

In Doris 1.1, *QuickCompaction* was introduced to proactively trigger compaction as data versions increase. Enhanced fragment metadata scanning swiftly identifies and initiates compaction for necessary fragments, effectively resolving real-time data merging issues. To handle high-frequency small file cumulative compaction, a scheduling and isolation mechanism ensures that resource-intensive base compaction does not interfere with new data merging. Additionally, the merging strategy was optimized with a *gradient merge method*, which merges files of similar sizes hierarchically and progressively. This reduces the frequency of file merges, saving CPU resources and enhancing system efficiency.

5.4 Effect

5.4.1 General Flink High Concurrency Scenarios

Specifically reflected in the following aspects:

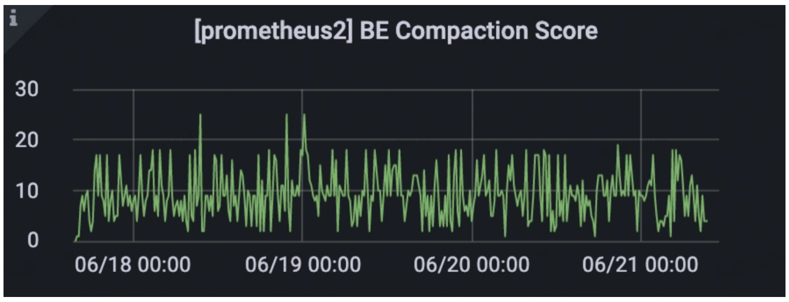


Figure 14: *Back-end Compaction score*

- Compaction Real-time: Data can be merged quickly, the number of tablet data versions is kept below 50, and the compaction score is stable.
- PU Resource Consumption: Doris version 1.1 has optimized the strategy for compaction of small files. In high-concurrency import scenarios, CPU resource consumption is reduced by 25
- QPS Query Delay is Stable: By reducing the CPU usage and the number of data versions, the overall order of data has been improved, and the delay of SQL queries will be reduced.

5.4.2 Second-Level Data Synchronization Scenario

In single bet and single tablet with 30 concurrent limit stream load pressure test on the client side, data in real-time $<$ 1s, the comparison before and after compaction score optimization as below:

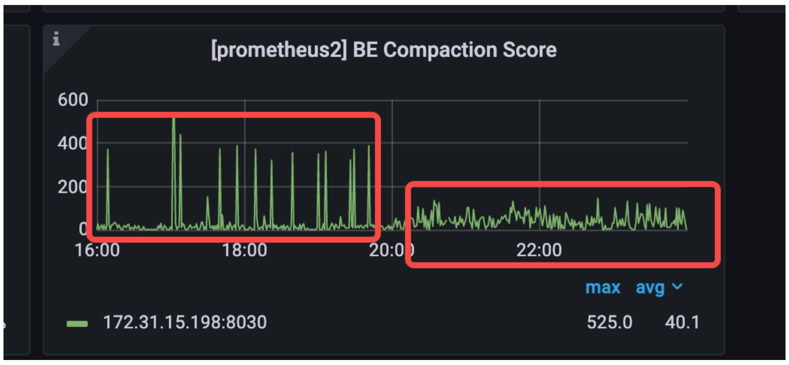


Figure 15: *Comparision before and after compaction*

6 Demonstration

6.1 Prerequisites

- Apache Flink (v1.19): Ensure you have Apache Flink installed and configured.
- Apache Doris (v2.0.11): Ensure you have Apache Doris installed and configured.
- Flink-Doris connector (v1.6.1): Ensure you have Apache Doris installed and configured.
- Java Development Kit (JDK): Ensure JDK is installed on your system.
- Maven: Ensure Maven is installed to manage project dependencies.
- MySQL (v8.4.0): Ensure MySQL is installed on your system.

Flink-Doris connector: 1.6.1

6.2 Guildline

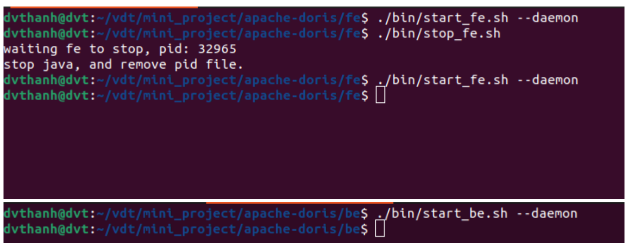
Step 1: Set Up Apache Flink and Doris

Apache Flink: download and install from flink.apache.org/downloads/.

Apache Doris: download and install from doris.apache.org.quick-start

Step 2: Run the Apache Doris on After install and extract the file, now we are going to configure the *fe.conf* anf *be.conf* with ports, JAVA_HOME directory and network to connect. After that, we start to run both front-end and back-end up with 2 command:

```
./bin/start_fe.sh --daemon

sudo sysctl -w vm.max_map_count=2000000
ulimit -n 60000
./bin/start_be.sh --daemon
```



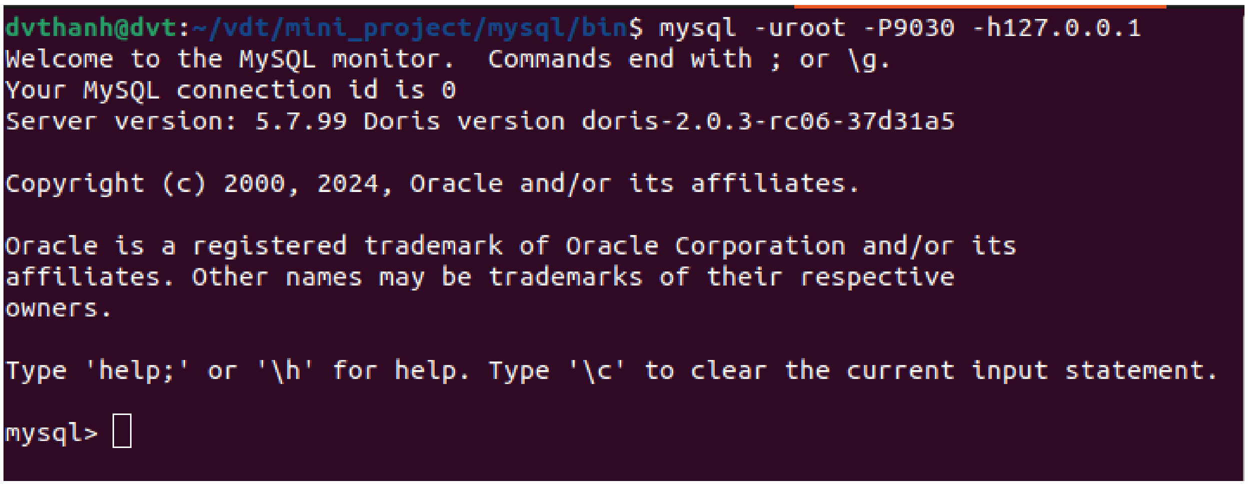
```
dvthanh@dvt:~/vdt/mini_project/apache-doris/fe$ ./bin/start_fe.sh --daemon
dvthanh@dvt:~/vdt/mini_project/apache-doris/fe$ ./bin/stop_fe.sh
waiting fe to stop, pid: 32965
stop java, and remove pid file.
dvthanh@dvt:~/vdt/mini_project/apache-doris/fe$ ./bin/start_fe.sh --daemon
dvthanh@dvt:~/vdt/mini_project/apache-doris/fe$
```

```
dvthanh@dvt:~/vdt/mini_project/apache-doris/be$ ./bin/start_be.sh --daemon
dvthanh@dvt:~/vdt/mini_project/apache-doris/be$
```

Figure 16: *Start front-end and back-end*

After that, we will run up the MySQL and connect it to our front end.

```
mysql -uroot -P9030 -h127.0.0.1
```



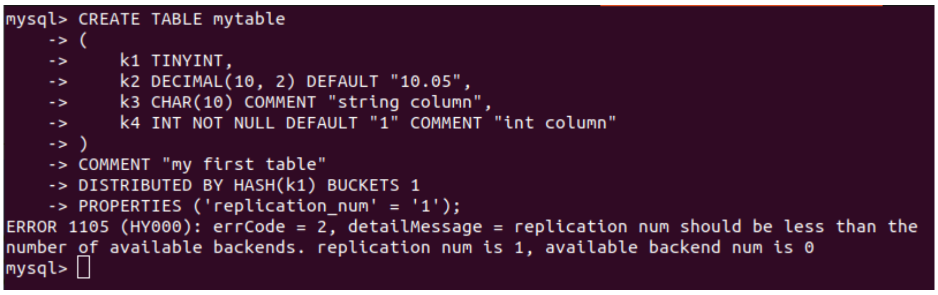
```
dvthanh@dvt:~/vdt/mini_project/mysql/bin$ mysql -uroot -P9030 -h127.0.0.1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 0
Server version: 5.7.99 Doris version doris-2.0.3-rc06-37d31a5

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```

Figure 17: *Connect MySQL to Doris's front-end*

Now we will go to check if the back-end is running or not by add a demo table.

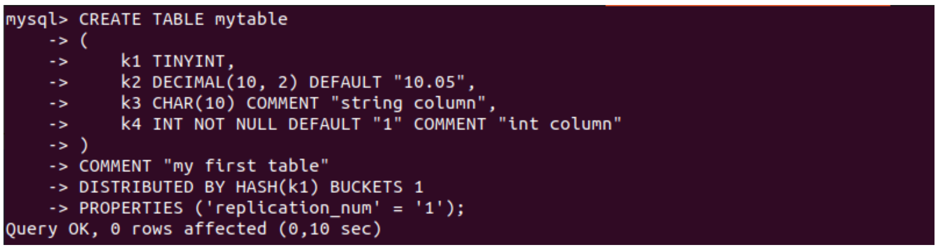
```
show databases
create database demo
use demo
```

```sql
CREATE TABLE mytable
(
    k1 TINYINT,
    k2 DECIMAL(10, 2) DEFAULT "10.05",
    k3 CHAR(10) COMMENT "string column",
    k4 INT NOT NULL DEFAULT "1" COMMENT "int column"
)
COMMENT "my first table"
DISTRIBUTED BY HASH(k1) BUCKETS 1
PROPERTIES ('replication_num' = '1');
```



```
mysql> CREATE TABLE mytable
    -> (
    ->     k1 TINYINT,
    ->     k2 DECIMAL(10, 2) DEFAULT "10.05",
    ->     k3 CHAR(10) COMMENT "string column",
    ->     k4 INT NOT NULL DEFAULT "1" COMMENT "int column"
    -> )
    -> COMMENT "my first table"
    -> DISTRIBUTED BY HASH(k1) BUCKETS 1
    -> PROPERTIES ('replication_num' = '1');
ERROR 1105 (HY000): errCode = 2, detailMessage = replication num should be less than the
number of available backends. replication num is 1, available backend num is 0
mysql> 
```

Figure 18: *Create table with unavailable back-end*



```
mysql> CREATE TABLE mytable
    -> (
    ->     k1 TINYINT,
    ->     k2 DECIMAL(10, 2) DEFAULT "10.05",
    ->     k3 CHAR(10) COMMENT "string column",
    ->     k4 INT NOT NULL DEFAULT "1" COMMENT "int column"
    -> )
    -> COMMENT "my first table"
    -> DISTRIBUTED BY HASH(k1) BUCKETS 1
    -> PROPERTIES ('replication_num' = '1');
Query OK, 0 rows affected (0,10 sec)
```

Figure 19: *Create table with available back-end*

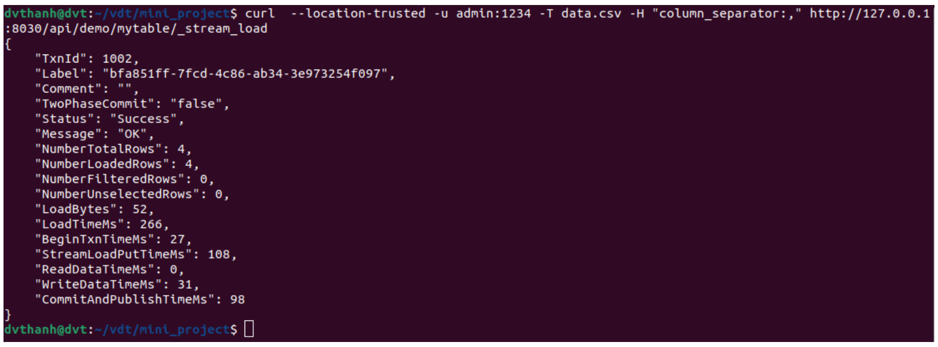
Besides, for the first time we must configure some information such as admin password, root password, connect front-end with back-end, etc. You can see detail in the above provided link.

After successfully creating table, we try to add data from *data.csv* file into the create tables. This is the content of csv file:

```
1,0.14,a1,20
2,1.04,b2,21
3,3.14,c3,22
4,4.35,d4,23
```

Use this command:

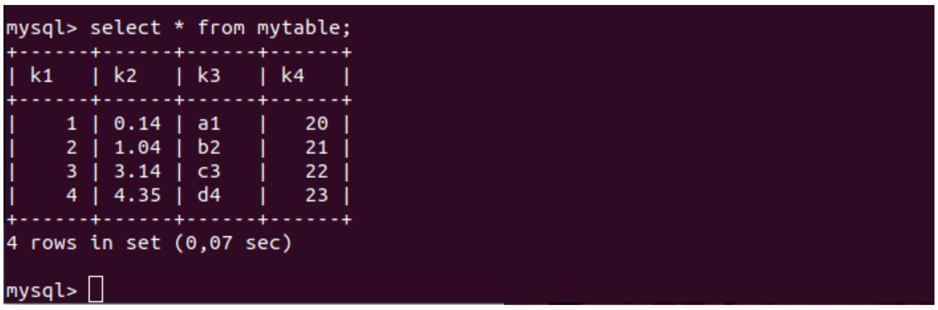
```
curl  --location-trusted -u admin:1234 -T data.csv -H "column_separator:," http
    ://127.0.0.1:8030/api/demo/mytable/_stream_load
```



```
dvthanh@dvt:~/vdt/mini_project$ curl  --location-trusted -u admin:1234 -T data.csv -H "column_separator:," http://127.0.0.1
:8030/api/demo/mytable/_stream_load
{
    "TxnId": 1002,
    "Label": "bfa851ff-7fcd-4c86-ab34-3e973254f097",
    "Comment": "",
    "TwoPhaseCommit": "false",
    "Status": "Success",
    "Message": "OK",
    "NumberTotalRows": 4,
    "NumberLoadedRows": 4,
    "NumberFilteredRows": 0,
    "NumberUnselectedRows": 0,
    "LoadBytes": 52,
    "LoadTimeMs": 266,
    "BeginTxnTimeMs": 27,
    "StreamLoadPutTimeMs": 108,
    "ReadDataTimeMs": 0,
    "WriteDataTimeMs": 31,
    "CommitAndPublishTimeMs": 98
}
dvthanh@dvt:~/vdt/mini_project$
```

Figure 20: *Add data from csv file to Apache Doris*

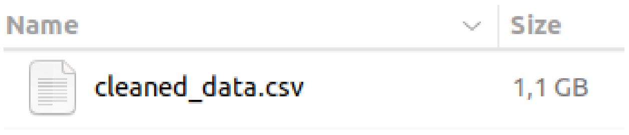
We will back to MySQL and try to query the data from it.



```
mysql> select * from mytable;
+------+------+------+------+
| k1   | k2   | k3   | k4   |
+------+------+------+------+
|    1 | 0.14 | a1   |   20 |
|    2 | 1.04 | b2   |   21 |
|    3 | 3.14 | c3   |   22 |
|    4 | 4.35 | d4   |   23 |
+------+------+------+------+
4 rows in set (0,07 sec)

mysql> 
```

Figure 21: *Query the inserted data in MySQL*

Step 3: Test the query speed with a larger data file The file *cleaned_ data.csv* with 1.1 GB volume. is chosen to add to test



Name	Size
cleaned_data.csv	1,1 GB

Figure 22: *Clean_ data.csv*

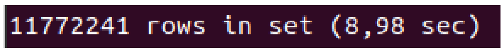
This SQL command create a table including 15 fields with datatypes. Moreover, the partition for age in 3 ranges <50, <70, <100. The bloom filter indexing is added into the column cp to speed up the query performance.

```sql
CREATE TABLE demotable (
    age INT(11) NULL,
    sex CHAR(10) NULL,
    dataset CHAR(50) NULL,
    cp CHAR(70) NULL,
    trestbps INT(11) NULL,
    chol INT(11) NULL,
    fbs CHAR(10) NULL,
    restecg CHAR(70) NULL,
    thalch INT(11) NULL,
    exang CHAR(10) NULL,
    oldpeak DOUBLE NULL,
    slope CHAR(20) NULL,
    ca INT(11) NULL,
    thal CHAR(70) NULL,
    num INT(11) NULL
) ENGINE=OLAP
DUPLICATE KEY(age, sex)
PARTITION BY RANGE(age) (
    PARTITION p1 VALUES LESS THAN (50),
    PARTITION p2 VALUES LESS THAN (70),
    PARTITION p3 VALUES LESS THAN (100)
)
DISTRIBUTED BY HASH(age) BUCKETS 1
PROPERTIES (
    "replication_num" = "1",
    "light_schema_change" = "true",
    "disable_auto_compaction" = "false",
    "bloom_filter_columns"="cp",
    "bloom_filter_fpp"="0.05"
);
```

```
curl  --location-trusted -u admin:1234 -T cleaned_data.csv -H "column_separator:,"
    http://127.0.0.1:8030/api/demo/demotable/_stream_load
```

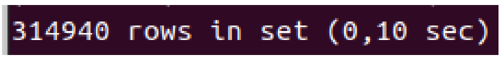
After that, we will back to MySQL to test query the data.

```
-- Query 1
SELECT * FROM demotable

-- Query 2
SELECT sex, dataset, cp, trestbps FROM demotable
WHERE age = 41;

-- Query 3
SELECT * FROM demotable
WHERE age = 58 and sex = "Female" and cp = "non-anginal";

-- Query 4
SELECT * FROM demotable d
JOIN age_person a ON d.age = a.age
WHERE d.age BETWEEN 50 AND 70 AND d.cp = "non-anginal";
```



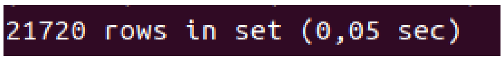
```
11772241 rows in set (8,98 sec)
```

Figure 23: *Query 1's execution time*



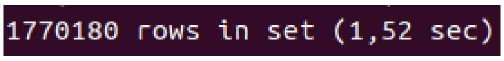
```
314940 rows in set (0,10 sec)
```

Figure 24: *Query 2's execution time*



```
21720 rows in set (0,05 sec)
```

Figure 25: *Query 3's execution time*



```
1770180 rows in set (1,52 sec)
```

Figure 26: *Query 4's execution time*

Evaluation:

- Query 1: Select all of things in the database with 11,772,241 rows (8.98 seconds).
- Query 2: Query with the `WHERE` phrase so the time decrease significantly, especially that *age* column is partitioned (0.1 seconds).
- Query 3: Query with the `WHERE` phrase so the time decrease significantly, especially that *age* column is partitioned and *cp* column is indexed (0.05 seconds).

- Query 4: This query start to be more complex with the `WHERE` (having `BETWEEN`) and `JOIN` phrases. Moreover, the partitioned *age* column and indexed *cp* column. This data querried this time is much bigger that th query 3. (1.52 seconds).

However, it can be seen that because of the dataset with many duplicated value, Apache Doris cannot perform their total capability in indexing. In general, we still can see that Apache Doris support really quick data query with 4 given queries served by their own optimizing features (join optimization, indexing, and partition) and several hidden capability mentioned in previous parts.

7 Conclusion

7.1 Conclusion

This mini-project successfully demonstrated the application of Apache Doris in real-time data analysis, particularly under extreme high-pressure conditions. By employing a top-down approach, we systematically explored the capabilities and performance of Apache Doris in handling intensive data synchronization tasks. The optimization of compaction scores led to significant improvements in stability and performance, reducing peak values and ensuring more consistent data processing.

However, while Apache Doris shows great potential, it also has certain disadvantages in real-time data analysis. One notable challenge is its relatively complex configuration and deployment process, which can be time-consuming and require specialized knowledge. Additionally, the optimization of compaction scores, while beneficial, may not always be straightforward and could necessitate fine-tuning based on specific use cases. Another limitation is the potential for increased resource consumption during peak loads, which might necessitate additional hardware investments.

The findings of this study underscore the potential of Apache Doris as a robust solution for real-time data analysis, capable of sustaining high loads with minimal latency. The successful optimization of compaction scores highlights the effectiveness of the implemented methodologies and provides a valuable reference for similar future applications.

7.2 Development Direction

Improve demonstration: Future work will shift from quick query performance to true real-time data processing. Integrating Apache Doris with real-time ingestion frameworks like Apache Kafka or Apache Flink will enable continuous data streaming and real-time updates.

Study other relevant features: Further research will explore other advanced features of Apache Doris. This deeper study will enhance its application in various data processing scenarios.

Study about combination capability: Future studies will explore combining Apache Doris with other technologies, such as machine learning frameworks, data visualization tools, and big data platforms. This will create a versatile and powerful data analysis ecosystem, enhancing robustness and efficiency.

8 Reference

[1] Apache Doris. (n.d.). Apache Doris Documentation, https://doris.apache.org/

[2] Apache Doris. (n.d.). Apache Doris: An MPP-based interactive SQL data warehousing for reporting and analysis. Retrieved from https://github.com/apache/doris

[3] VedLob (2023). Introduction to Apache Doris: A Next Generation Real-Time Data Warehouse, https://www.youtube.com/watch?v=w_hreEaiDQ4&t=808s&pp=ygUMYXBhY2hlIGRvcmlz

[4] Apache Doris (2022), How Flink's real-time writes to Apache Doris ensure both high throughput and low latency, https://doris.apache.org/blog/Flink-realtime-write/

[5] Li Zhe (2022), JD.com's exploration and practice with Apache Doris in real time OLAP, https://doris.apache.org/blog/JD_OLAP

[6] CIGNA & CMB (2023), Less components, higher performance: Apache Doris instead of ClickHouse, MySQL, Presto, and HBase, https://doris.apache.org/blog/less-components-higher-performanc

[7] Apache Flink. (n.d.). Apache Flink Documentation, https://flink.apache.org/documentation/