MapReduce Service (MRS)
3.3.1-LTS
Introduction

Issue	03
Date	2024-10-30

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


and other Huawei trademarks are the property of Huawei Technologies Co., Ltd.
All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei Cloud and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Cloud Computing Technologies Co., Ltd.

Address: Huawei Cloud Data Center Jiaoxinggong Road
Qianzhong Avenue
Gui'an New District
Gui Zhou 550029
People's Republic of China

Website: https://www.huaweicloud.com/intl/en-us/

Contents

1 What Is MRS?

MapReduce Service (MRS) is a data processing and analysis service built based on a cloud computing platform.

It builds a reliable, secure, and easy-to-use operation and maintenance (O&M) platform and provides storage and analysis capabilities for massive data, helping address user data storage and processing demands. You can apply for and host components, such as Hadoop, Spark, HBase, and Hive, to quickly create clusters on hosts and provide batch storage and computing capabilities for massive amount of data that has low requirements on real-time processing. You can terminate the clusters as soon as completing data storage and computing.

MRS clusters are classified into the following types: Elastic Cloud Server (ECS) and Bare Metal Server (BMS) clusters installed using images, and physical machine clusters managed by ManageOne.

Table 1-1 MRS cluster types

Cluster Type	Cluster Version	Cluster Provisioning Mode
ECS cluster	MRS 3.3.1-LTS	After the MRS console and the corresponding MRS image are installed, create an MRS cluster based on ECSs on the console.
BMS cluster	MRS 3.3.1-LTS	After the MRS console and the corresponding MRS image are installed, create an MRS cluster based on BMSs on the console.
Physical machine cluster	MRS 3.3.1-LTS_offline	After MRS and an independent physical machine cluster of MRS are installed, manage the MRS cluster on the MRS console in a unified manner.

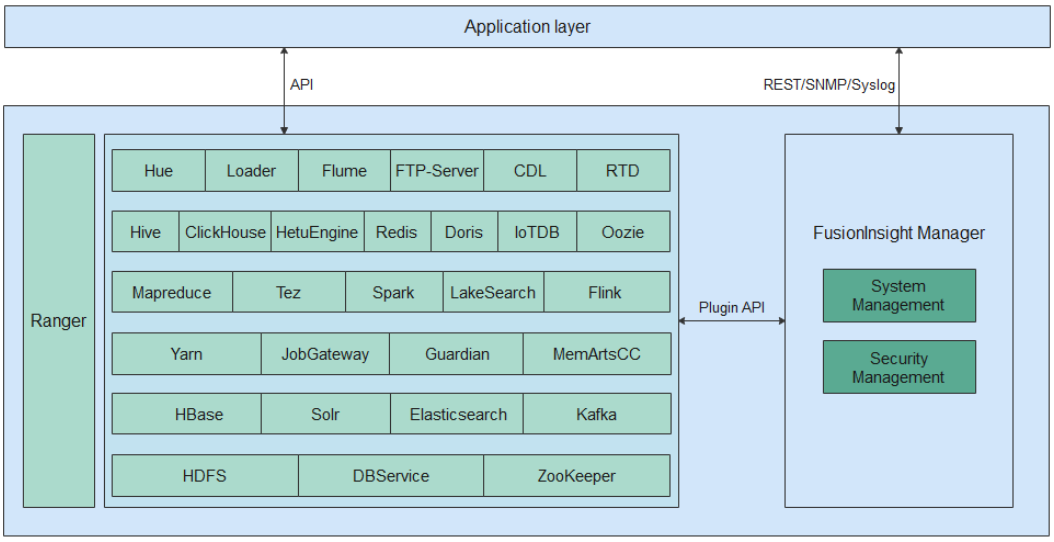
NOTE

- Service configuration parameters in the ECS/BMS cluster cannot be modified during cluster creation.
- ECS/BMS clusters do not support functions such as HDFS federation and cross-AZ HA of a single cluster.
- ECS/BMS cluster capacity expansion is based on node groups with different specifications. One node group supports only one specification.
- On the MRS console, operations performed on an ECS cluster are basically the same as those performed on a BMS cluster. This document describes operations on an ECS cluster. If operations on the two clusters differ, the operations will be described separately.

System Architecture

Figure 1-1 shows the logical architecture of an MRS cluster.

Figure 1-1 Logical architecture



MRS encapsulates and enhances open-source components. The following components are included:

- CDL

  A simple, efficient, and real-time data integration service.
- ClickHouse

  A column-based Database Management System (DBMS) for On-Line Analytical Processing (OLAP).
- DBService

  A conventional, high-reliability, relational database. It provides metadata storage service for Hive, Hue, Oozie, Loader, and Redis.
- Doris

  An easy-to-use, high-performance, and real-time analytical database.
- Elasticsearch

  A distributed and open-source system based on JAVA/Lucene. It integrates the search engine and NoSQL database functions, and supports RESTful requests.

- Flink

  A unified computing framework that supports both batch processing and stream processing. It provides a stream data processing engine that supports data distribution and parallel computing.

- Flume

  A distributed, reliable, and HA massive log aggregation system that supports customized data transmitters for collecting data. It also provides simple processing of data and writes the data to customizable data receivers.

- FTP-Server

  Enables basic operations on the HDFS through an FTP client. The basic operations include uploading or downloading files, viewing, creating, or deleting directories, and modifying file access permissions.

- Guardian

  Guardian provides temporary authentication credentials for accessing OBS.

- GraphBase (supported only by physical machine clusters)

  A distributed graph database based on HBase and Elasticsearch. It builds a property graph model for storage and provides powerful graph query, analysis, and traversal capabilities.

- HBase

  A distributed, column-oriented storage system built on the HDFS. It stores massive data.

- HDFS

  A Hadoop Distributed File System (HDFS) that supports high-throughput data access and is suitable for applications with large-scale data sets.

- HetuEngine

  HetuEngine is a high-performance, interactive SQL analysis and data virtualization engine developed by Huawei. It seamlessly integrates with the big data ecosystem to implement interactive query of massive amounts of data within seconds, and supports cross-source and cross-domain unified data access to enable one-stop SQL convergence analysis in the data lake, between lakes, and between lakehouses.

- Hive

  An open-source data warehouse built on Hadoop. It stores structured data and implements basic data analysis using the Hive Query Language (HQL), a SQL-like language.

- Hue

  Provides a graphical web user interface (WebUI) for MRS applications. It supports HDFS, Hive, Yarn/MapReduce, Oozie, Solr, and ZooKeeper.

- IoTDB

  A software system that collects, stores, manages, and analyzes IoT time series data.

- JobGateway

  A REST API service that allows you to submit Spark, Hive, MapReduce, and Flink jobs.

- Kafka

A distributed, real-time message publishing and subscription system with partitions and replicas. It provides scalable, high-throughput, low-latency, and highly reliable message dispatching services.

- KMS

  A key management server compiled based on the KeyProvider API.

- LakeSearch

  A semantic-based, multi-modal smart search platform. LakeSearch provides out-of-the-box search services with improved accuracy.

- Loader

  An enhanced open-source tool based on Sqoop. It loads and implements data exchange between MRS and relational databases. It provides representational state transfer (REST) application programming interfaces (APIs) for third-party scheduling platforms.

- Manager

  As an O&M system, Manager implements highly reliable and secure cluster management for MRS. It supports installation and deployment, monitoring, alarm management, user management, permission management, audit, service management, and health check of large clusters.

- MapReduce

  A distributed data processing framework. It implements rapid, parallel processing of massive data.

- MemArtsCC

  MemArtsCC is a distributed cache system on compute nodes.

- Oozie

  Orchestrates and executes jobs for open-source Hadoop components. It runs in a Java servlet container (for example, Tomcat) as a Java web application and uses a database to store workflow definitions and running workflow instances (including the status and variables of the instances).

- Ranger

  A centralized framework based on the Hadoop platform. It provides permission control APIs such as monitoring, operation, and management APIs for complex data.

- Redis

  An open-source and high-performance key-value distributed storage database. It supports a variety of data types, supplementing the key-value storage such as memcached and meeting the real-time and high-concurrency requirements.

- RTD
  - Containers

    Provides physical environments for the running of Business Logic Unit (BLU) instances and controls the start and stop of the BLUs.

    Provides Access Load Balance (ALB) to connect to load balancers. ALB implements socket access. Specifically, it distributes requests of different projects to service instances on the platform based on different processing policies and implements conversion between protocol interfaces. ALB is not provided as an independent service but integrated in Containers.

  - MOTService

    Provides fast and large-throughput access capabilities and uses stored procedures to quickly process service logic at the database layer. It is deployed in active/standby mode.

  - RTDService

    Functions as the unified web definition entry of RTD and allows users to define tenants, event sources, dimensions, variables, models, and rules.

- Solr

  A high-performance, full-text search server based on Apache Lucene. It extends Lucene and provides a query language richer than that provided by Lucene. The configurable and scalable Solr optimizes the query performance and provides a comprehensive function management GUI, which makes it an excellent full-text search engine.

- Spark

  A distributed in-memory computing framework.

- Tez

  Supports the distributed computing framework of directed acyclic graphs (DAGs).

- Yarn

  A general resource module that functions as a resource management system, which manages and schedules resources for various applications.

- ZooKeeper

  Enables highly reliable distributed coordination. It helps prevent single point of failures (SPOFs) and provides reliable services for applications.

2 Applicable Objects and Scenarios of MRS

MRS applies to massive data processing and storage in various industries.

- Analyzing and processing massive sets of data

  Usage: analysis and processing of massive sets of data, online and offline analysis, and business intelligence

  Characteristics: massive sets of data, heavy computing, time-consuming data analysis, and numerous computers working simultaneously

  Scenarios: log analysis, online and offline analysis, simulation calculations in scientific research, and biometric analysis
- Large-scale data storage

  Usage: storage and retrieval of massive sets of data and data warehouse

  Characteristics: storage, retrieval, and disaster recovery of massive sets of data and zero data loss

  Scenarios: log storage, file storage, simulation data storage in scientific research, biological characteristic information storage, and genetic engineering data storage
- Stream processing for massive sets of data

  Usage: real-time analysis of massive sets of data, continuous computing, as well as online and offline message consumption

  Characteristics: massive sets of data, high throughput, high reliability, easy scalability, and distributed real-time computing framework

  Scenarios: streaming data collection, web-based tracking, data monitoring, distributed ETL, and risk control

3 Basic Concepts

Region and AZ

A region is a geographic area where MRS is located.

Availability zones (AZs) in the same region can communicate with each other over the intranet, but different regions are not connected over intranet.

MRS can be used in data centers of different regions. You can subscribe to MRS in different regions and design applications to better meet customer requirements or comply with local laws and other demands.

Each region contains many AZs where power resources and networks are physically isolated. AZs in the same region can communicate with each other over the intranet, but those in different regions cannot. Each AZ provides cost-effective and low-latency network connections that are unaffected by faults which may occur in other AZs. Therefore, provisioning MRS in separate AZs protects your applications against local faults that occur in a specific location.

Hadoop

Hadoop is a distributed system framework. It allows users to develop distributed applications using high-speed computing and storage provided by clusters without knowing the underlying details of the distributed system. It can also reliably and efficiently process massive data in scalable, distributed mode. Hadoop is reliable because it maintains multiple work data duplicates, enabling distributed processing for failed nodes. Hadoop is highly efficient because it processes data in parallel mode. Hadoop is scalable because it processes data at the PB level. Hadoop consists of the Hadoop distributed file system (HDFS), MapReduce, HBase, and Hive.

4 Node Types

An MRS cluster consists of multiple nodes. Based on the component roles deployed on nodes, nodes in a cluster are classified as management, controller, and data nodes.

- Management node (MN): installs FusionInsight Manager (the management system of the MRS cluster). It provides a unified access entry. FusionInsight Manager centrally manages nodes and services deployed in the cluster.
- Control node (CN): controls and monitors how data nodes store and receive data, and send process status, and provides other public functions.
- Data node (DN): executes the instructions sent by the management node, reports task status, stores data, and provides other public functions.

NOTE

For an ECS/BMS cluster, the system groups nodes based on node specifications for easier management. Different node groups use different VM specifications.

- Both management and controller nodes are master nodes. By default, management and controller nodes form a master node group when an ECS/BMS cluster is created.
- Data nodes either belong to the core node group or the task node group. You can scale the storage space or computing capabilities of MRS by adding Core nodes or Task nodes without modifying the system architecture. The scaling reduces O&M costs. Deployment instances in a data node group are typically of the same type.

Table 4-1 Node types and groups

Node Type	Node Group Type	Function	Auto Scaling
Management node	Master node group	Nodes on which Manager and other control roles are deployed in a cluster. A master node group is created by default when a BMS/ECS cluster is created.	Not supported.
Control node			

Node Type	Node Group Type	Function	Auto Scaling
Data node	Core node group	Nodes used to process and store data. You can manually add Core nodes to the cluster to handle the peak load. After the cluster is expanded, you need to update the client.	
	Task node group	Nodes used to process data but not store persistent data. After a cluster is created, you can configure auto scaling policies to implement auto scaling. After the cluster is scaled out, you do not need to update the client.	Elastic scaling with high flexibility. Because there is no data storage, the scaling speed is fast.

NOTE

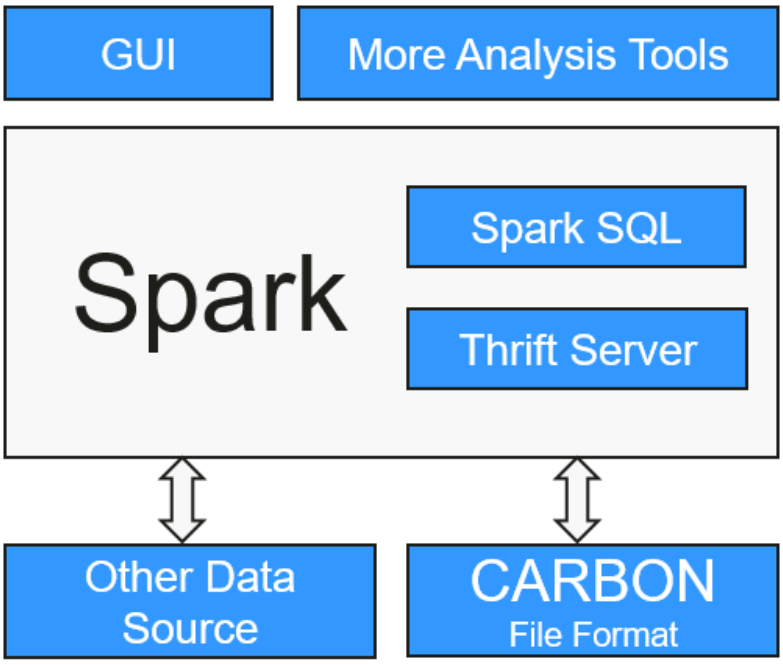
A task node group is a node group whose type is set to **Task** when a cluster is created or a node group is added. Only the NodeManager role (except mandatory roles) can be deployed in this node group.

5 Components

5.1 CarbonData

CarbonData is a new Apache Hadoop native data-store format. CarbonData allows faster interactive queries over PetaBytes of data using advanced columnar storage, index, compression, and encoding techniques to improve computing efficiency. In addition, CarbonData is also a high-performance analysis engine that integrates data sources with Spark.

Figure 5-1 Basic architecture of CarbonData



The purpose of using CarbonData is to provide quick response to ad hoc queries of big data. Essentially, CarbonData is an Online Analytical Processing (OLAP) engine, which stores data using tables similar to those in Relational Database Management System (RDBMS). You can import more than 10 TB data to tables created in CarbonData format, and CarbonData automatically organizes and

stores data using the compressed multi-dimensional indexes. After data is loaded to CarbonData, CarbonData responds to ad hoc queries in seconds.

CarbonData integrates data sources into the Spark ecosystem. You can use Spark SQL to query and analyze data, or use the third-party tool ThriftServer provided by Spark to connect to Spark SQL.

CarbonData features

- SQL: CarbonData is compatible with Spark SQL and supports SQL query operations performed on Spark SQL.
- Simple Table dataset definition: CarbonData allows you to define and create datasets by using user-friendly Data Definition Language (DDL) statements. CarbonData DDL is flexible and easy to use, and can define complex tables.
- Easy data management: CarbonData provides various data management functions for data loading and maintenance. It can load historical data and incrementally load new data. The loaded data can be deleted according to the loading time and specific data loading operations can be canceled.
- CarbonData file format is a columnar store in HDFS. It has many features that a modern columnar format has, such as splittable and compression schema.

Unique features of CarbonData

- Stores data along with index: Significantly accelerates query performance and reduces the I/O scans and CPU resources, when there are filters in the query. CarbonData index consists of multiple levels of indices. A processing framework can leverage this index to reduce the task it needs to schedule and process, and it can also perform skip scan in more finer grain unit (called blocklet) in task side scanning instead of scanning the whole file.
- Operable encoded data: Through supporting efficient compression and global encoding schemes, CarbonData can query on compressed/encoded data. The data can be converted just before returning the results to the users, which is "late materialized".
- Supports various use cases with one single data format: like interactive OLAP-style query, Sequential Access (big scan), and Random Access (narrow scan).

Key technologies and advantages of CarbonData

- Quick query response: CarbonData features high-performance query. The query speed of CarbonData is 10 times of that of Spark SQL. It uses dedicated data formats and applies multiple index technologies, global dictionary code, and multiple push-down optimizations, providing quick response to TB-level data queries.
- Efficient data compression: CarbonData compresses data by combining the lightweight and heavyweight compression algorithms. This significantly saves 60% to 80% data storage space and the hardware storage cost.

For details about CarbonData architecture and principles, see **https://carbondata.apache.org/**.

5.2 CDL

5.2.1 CDL Basic Principles

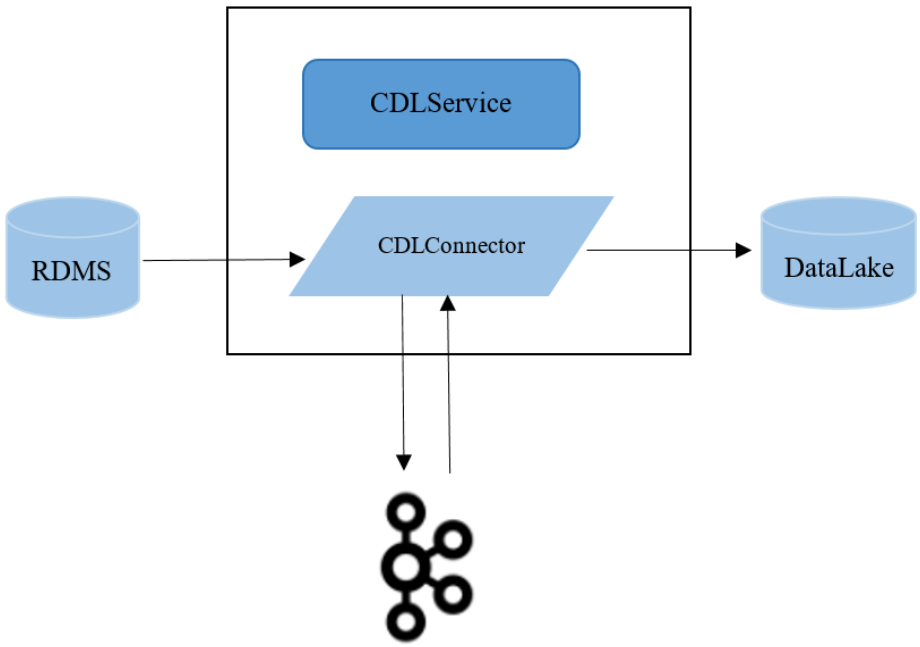
Overview

Change Data Loader (CDL) is a real-time data integration service based on Kafka Connect. The CDL service captures data change events from various OLTP databases and push them to Kafka. Then, Sink Connector pushes the events to the big data ecosystem.

Currently, CDL supports the following data sources: MySQL, PostgreSQL, Hudi, openGauss, and ThirdParty-Kafka (including Oracle, openGauss, and IoT). Data can be written to Kafka, Hudi, GaussDB(DWS), and ClickHouse data at the target end.

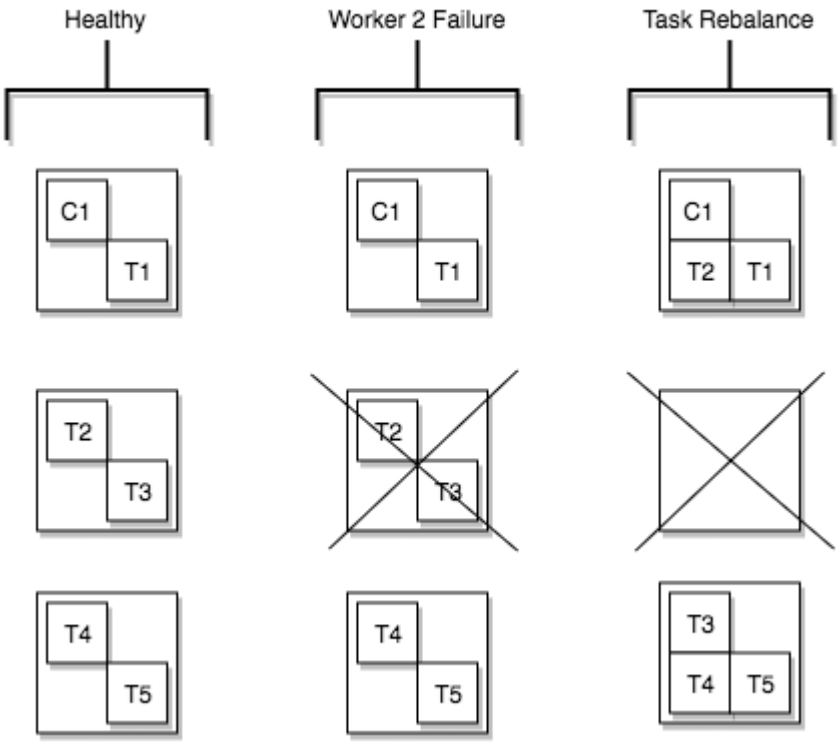
CDL structure

The CDL service contains two important roles: CDLConnector and CDLService. CDLConnector, including Source Connector and Sink Connector, is the instance for executing data capture jobs. CDLService is the instance for managing and creating jobs.



There are two active CDLService instances. Both can perform service operations. The CDLConnector instances work in distributed mode and provide HA and rebalance capabilities. When tasks are created, the number of tasks specified is balanced among CDLConnector instances in a cluster to ensure that the number of tasks running on each instance is similar. If a CDLConnector instance is abnormal or a node breaks down, the number of tasks are rebalanced on other nodes.

Figure 5-2 Rebalance of a task



5.2.2 Relationship Between CDL and Other Components

The CDL component works on top of the Kafka Connect framework. Captured data is forwarded using Kafka topics and written into Hudi, GaussDB(DWS) and ClickHouse through Spark. So, CDL component depends on the Kafka component. CDL stores task metadata and monitoring information that are also stored in a database, so it also depends on the DBService component.

5.3 ClickHouse

5.3.1 Basic Principle

Introduction to ClickHouse

ClickHouse is an open-source columnar database oriented to online analysis and processing. It is independent of the Hadoop big data system and features ultimate compression rate and fast query performance. In addition, ClickHouse supports SQL query and provides good query performance, especially the aggregation analysis and query performance based on large and wide tables. The query speed is one order of magnitude faster than that of other analytical databases.

The core functions of ClickHouse are as follows:

Comprehensive DBMS functions

ClickHouse is a database management system (DBMS) that provides the following basic functions:

- Data Definition Language (DDL): allows databases, tables, and views to be dynamically created, modified, or deleted without restarting services.
- Data Manipulation Language (DML): allows data to be queried, inserted, modified, or deleted dynamically.
- Permission control: supports user-based database or table operation permission settings to ensure data security.
- Data backup and restoration: supports data backup, export, import, and restoration to meet the requirements of the production environment.
- Distributed management: provides the cluster mode to automatically manage multiple database nodes.

Column-based storage and data compression

ClickHouse is a database that uses column-based storage. Data is organized by column. Data in the same column is stored together, and data in different columns is stored in different files.

During data query, columnar storage can reduce the data scanning range and data transmission size, thereby improving data query efficiency.

In a traditional row-based database system, data is stored in the sequence in **Table 5-1**:

Table 5-1 Row-based database

row	ID	Flag	Name	Event	Time
0	12345678901	0	name1	1	2020/1/11 15:19
1	32345678901	1	name2	1	2020/5/12 18:10
2	42345678901	1	name3	1	2020/6/13 17:38
N	...	...	...	...	...

In a row-based database, data in the same row is physically stored together. In a column-based database system, data is stored in the sequence in **Table 5-2**:

Table 5-2 Columnar database

row:	0	1	2	N
ID:	12345678901	32345678901	42345678901	...
Flag:	0	1	1	...
Name:	name1	name2	name3	...
Event:	1	1	1	...

Time:	2020/1/11 15:19	2020/5/12 18:10	2020/6/13 17:38	...

This example shows only the arrangement of data in a columnar database. Columnar databases store data in the same column together and data in different columns separately. Columnar databases are more suitable for online analytical processing (OLAP) scenarios.

Vectorized executor

ClickHouse uses CPU's Single Instruction Multiple Data (SIMD) to implement vectorized execution. SIMD is an implementation mode that uses a single instruction to operate multiple pieces of data and improves performance with data parallelism (other methods include instruction-level parallelism and thread-level parallelism). The principle of SIMD is to implement parallel data operations at the CPU register level.

Relational model and SQL query

ClickHouse uses SQL as the query language and provides standard SQL query APIs for existing third-party analysis visualization systems to easily integrate with ClickHouse.

In addition, ClickHouse uses a relational model. Therefore, the cost of migrating the system built on a traditional relational database or data warehouse to ClickHouse is lower.

Data sharding and distributed query

The ClickHouse cluster consists of one or more shards, and each shard corresponds to one ClickHouse service node. The maximum number of shards depends on the number of nodes (one shard corresponds to only one service node).

ClickHouse introduces the concepts of local table and distributed table. A local table is equivalent to a data shard. A distributed table itself does not store any data. It is an access proxy of the local table and functions as the sharding middleware. With the help of distributed tables, multiple data shards can be accessed by using the proxy, thereby implementing distributed query.

ClickHouse Applications

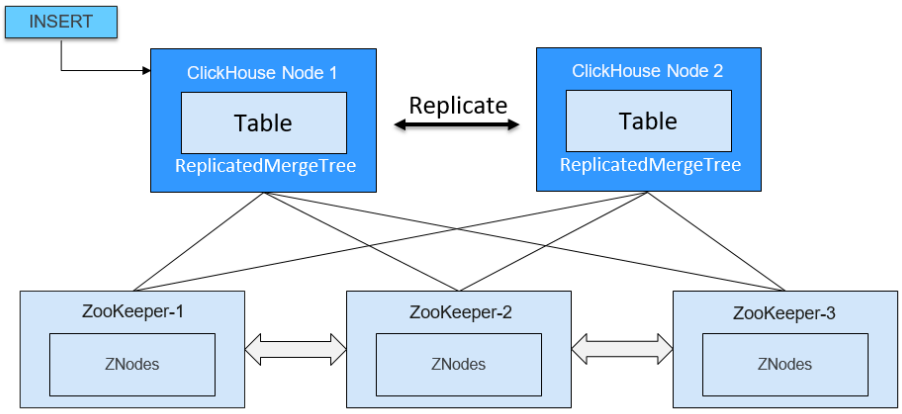
ClickHouse is short for Click Stream and Data Warehouse. It is initially applied to a web traffic analysis tool to perform OLAP analysis for data warehouses based on page click event flows. Currently, ClickHouse is widely used in Internet advertising, app and web traffic analysis, telecommunications, finance, and Internet of Things (IoT) fields. It is applicable to business intelligence application scenarios and has a large number of applications and practices worldwide. For details, visit **https://clickhouse.tech/docs/en/introduction/adopters/**.

5.3.2 Key Features

Replica Mechanism

ClickHouse uses Zookeeper to implement the replica mechanism through the ReplicatedMergeTree engine. The replica mechanism is a multi-master

architecture. An **INSERT** statement can be sent to any replica, and other replicas perform asynchronous data replication.



Replica mechanism functions:

- The design of the ClickHouse replica mechanism minimizes network data transmission, synchronizes data in different data centers, and builds a multi-data center and multi-active remote cluster architecture.
- The Replica mechanism is the basis for implementing HA, load balance, and migration/upgrade functions.
- High availability: The system monitors the synchronization status of replica data, identifies faulty nodes, and performs fault recovery when the nodes recover, ensuring overall high availability of services.

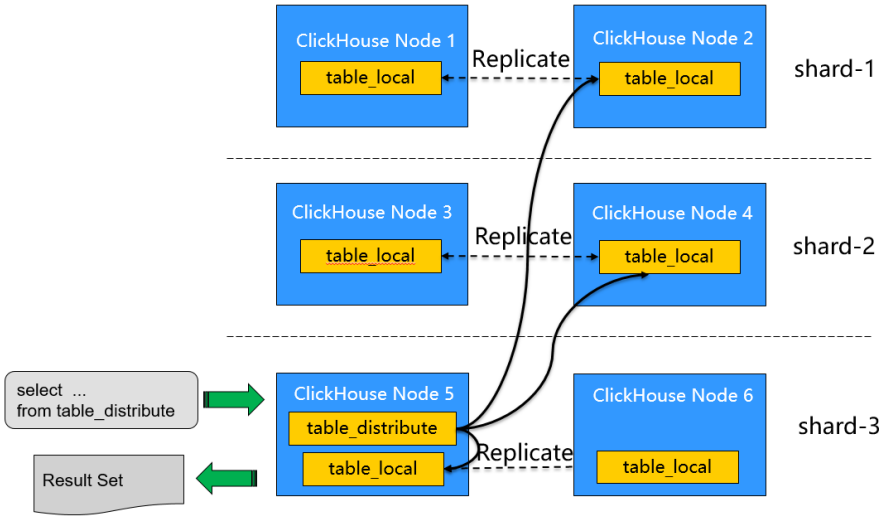
Distributed query

ClickHouse provides linear scaling through sharding and distributed table mechanisms.

- The sharding mechanism is used to solve the performance bottleneck of a single node. Data in a table is split horizontally to multiple nodes. Data on different nodes is not duplicated. In this way, ClickHouse can be linearly expanded by adding shards.
- Distributed table: When querying sharded data, a distributed table is used for query. The distributed table engine does not store any data. It is only a layer-1 proxy and can automatically route data to each shard node in the cluster to obtain data. That is, a distributed table needs to work together with other data tables.

As shown in the following figure **Figure 5-3**, the distributed table table_distributed needs to be queried. The distributed table automatically routes query requests to shard nodes and aggregates results.

Figure 5-3 Distributed query



MergeTree Engine

MergeTree and its family (*MergeTree) are ClickHouse's most powerful storage engine, designed to insert large amounts of data into a single table. Data is quickly written as data blocks. Data blocks are asynchronously merged in the background to ensure efficient insertion and query performance.

The following functions are supported:

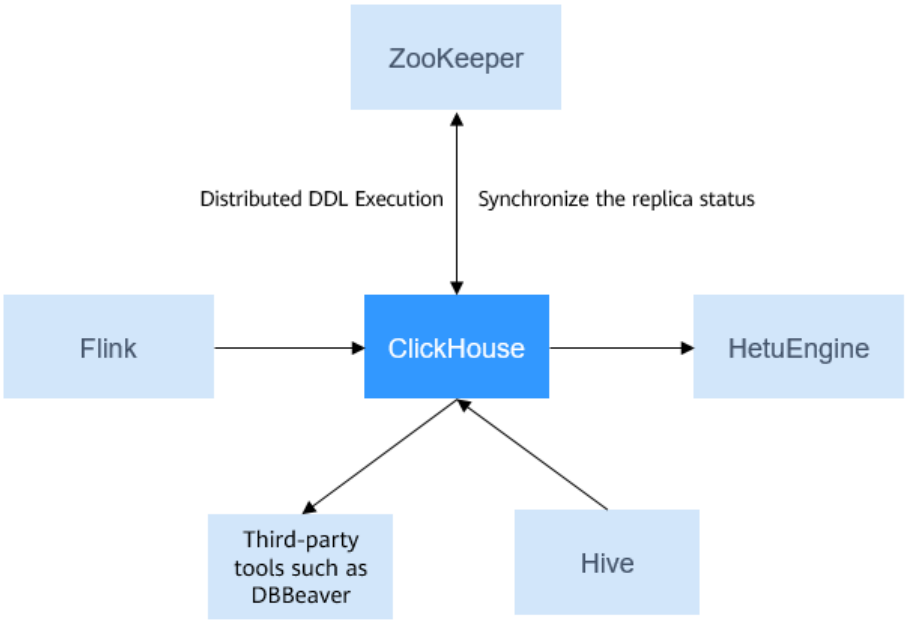
- Primary key sorting, sparse indexing
- Data partitioning
- Replica mechanism (ReplicatedMergeTree series)
- Data sampling
- Concurrent data access
- Supports TTL
- Secondary index (Data skipping index)

5.3.3 Relationship with Other Components

The following figure **Figure 5-4** shows the relationship between ClickHouse and other components.

- Flink supports ClickHouse Sink.
- Hive/SparkSQL data can be imported in batches. ClickHouse.
- The HetuEngine supports the ClickHouse data source.
- Common third-party tools, such as DBeaver, support ClickHouse interconnection.
- ClickHouse depends on ZooKeeper to implement distributed DDL execution as well as status synchronization between the active and standby nodes of the ReplicatedMergeTree table.

Figure 5-4 Relationships between ClickHouse and Other Components



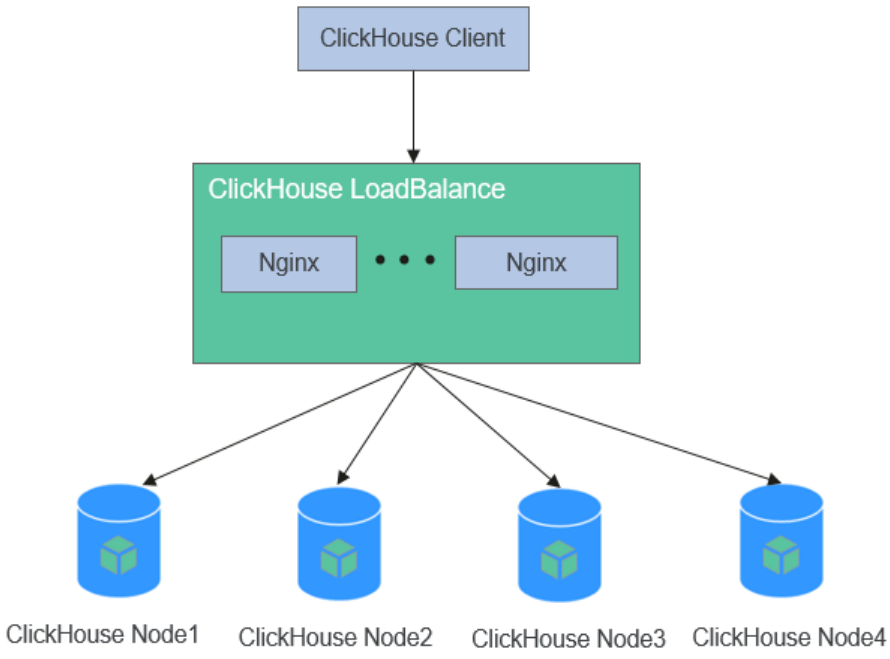
5.3.4 ClickHouse Enhanced Open Source Features

LoadBalance

ClickHouse uses the LoadBalance-based deployment architecture to automatically distribute user access traffic to multiple backend nodes, expanding service capabilities to external systems and improving fault tolerance.

When a client application requests a cluster, a Nginx-based ClickHouse controller node is used to distribute traffic. In this way, data read/write load and high availability of application access are guaranteed.

Figure 5-5 Working principle of the LoadBalance



5.4 Containers

5.4.1 ALB Basic Principles

Overview

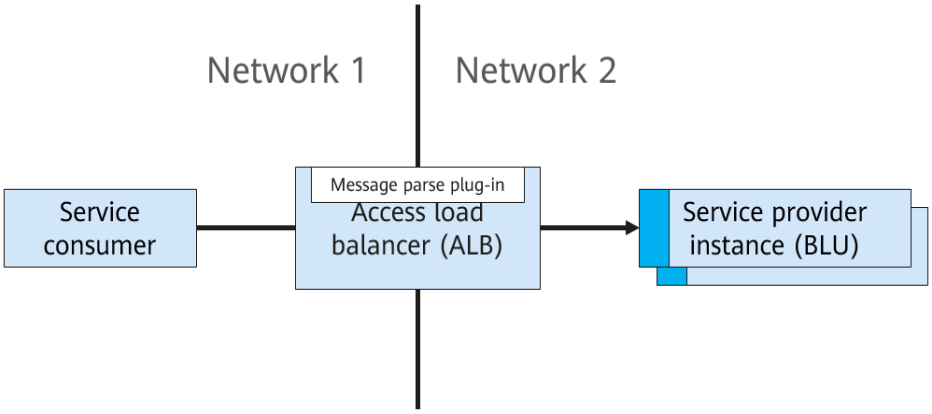
Access Load Balance (ALB) allows external systems to access clusters through HTTP or sockets. After requests are received, ALB forwards them to BLUs in the cluster for conversion between interfaces of different protocols. BLUs are developed based on the service consumer specifications and provide RESTful APIs for external systems.

NOTE

In FusionInsight RTD, ALB is not provided as an independent service but integrated with Containers.

Structure

Figure 5-6 ALB structure



- ALB is used between service consumers and service providers.
- ALB provides access channels for untrusted networks, shields internal topology details, and implements internal load balancing and service routing.

Principles

ALB provides multi-protocol access, which improves the networking adaptability of FusionInsight RTD. In a complex network where the FusionInsight RTD client and cluster are not in the same network segment, ALB can be used as the gateway to process messages, distribute requests to service instances, and control distribution policies.

After FusionInsight RTD is installed, the system administrator can deploy ALB on the platform. Physically, ALB is a preset BLU in FusionInsight RTD.

Relationship with Other Components

- **ALB and BLU**

  ALB is a load balancer that hides BLU's multiple instances. Customers can use ALB to access BLUs.

- **ALB and ZooKeeper**

  ZooKeeper provides the service registration center. ALB subscribes to services from the registration center as a service consumer.

5.4.2 Containers Basic Principles

Overview

Based on the open source Apache Tomcat 8, Containers is a lightweight application container that supports standard functions of the community edition and incorporates enhancements for enterprise applications. It provides running environment resources for and manages BLUs deployed on the FusionInsight RTD platform and supports heterogeneous underlying platforms.

Tomcat Server is an open source and lightweight web application server for small- and medium-sized systems and scenarios with few concurrent access requests. It provides the following functions:

- Supports Servlet Spec 3.0 and JSP Spec 2.2.
- Prevents cross-site script attacks using random numbers.
- Prevents session attacks by changing the jessionid mechanism in security authentication.
- Records asynchronous logs.

Structure

After an event source is brought online, its BLU is deployed in a container. In FusionInsight RTD, a maximum of five containers can be installed on each host.

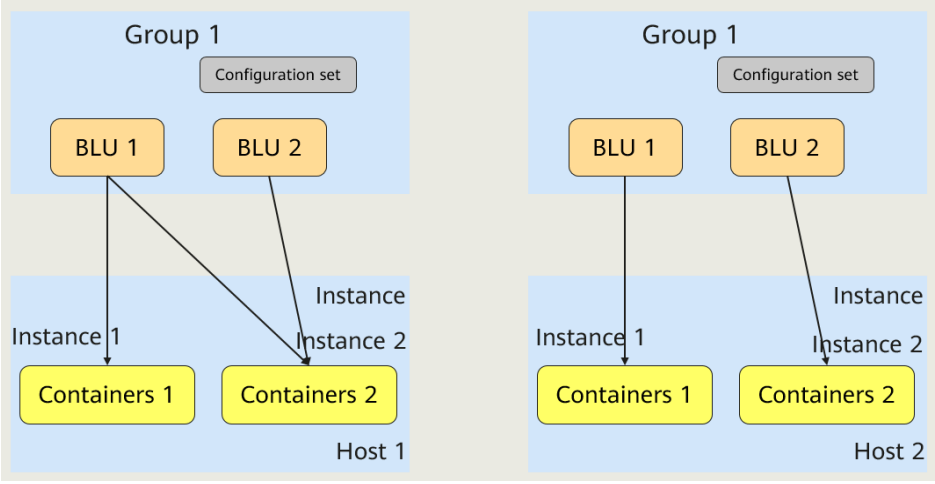
Principles

FusionInsight RTD manages applications in groups. Containers in a cluster can belong to only one group at a time, but different BLUs in one group can be deployed in the same container at the same time. Each BLU creates a BLU instance in a container. See **Figure 5-7**.

The Containers component monitors and manages BLUs. System administrators can deploy, start, stop, and delete BLUs on FusionInsight Manager.

The FusionInsight Manager platform provides the function of directly uploading configuration files for each BLU so that users can update BLUs based on service environments. BLUs in a group can use the same configuration file set.

Figure 5-7 BLU deployment



Relationship with Other Components

- **Containers and RTDService**

  After the event source defined by RTDService is brought online, the generated BLU application is deployed in Containers.

- **Containers and ALB**

ALB is a special BLU application deployed in Containers. It is a load balancer for accessing BLUs.

- **Containers and ZooKeeper**

  ZooKeeper provides a service registration center for service providers in BLUs and service address lists for service consumers.

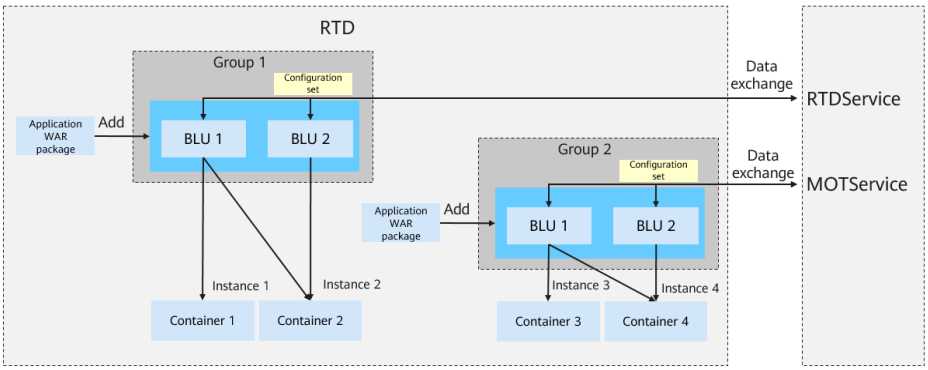
5.4.3 Containers Enhanced Features

Application Management

FusionInsight RTD deploys Tomcat clusters and distributes BLUs, and also monitors the Tomcat clusters and BLUs.

After obtaining an application developed by a service developer, the system administrator can easily and quickly deploy the application to a cluster with the UI.

Figure 5-8 FusionInsight RTD platform application deployment

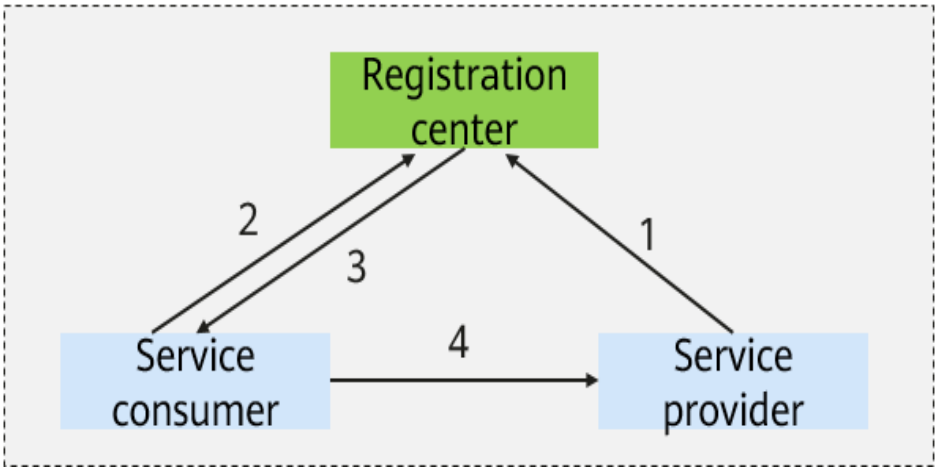


Service Governance

Developers can quickly develop RESTful services in BLUs. FusionInsight RTD manages services provided by BLUs, including controlling service access, managing load balancing policy, and performing grayscale release. FusionInsight RTD also monitors calling latency and TPS of the services.

Figure 5-9 shows the basic service invoking process in FusionInsight RTD.

Figure 5-9 Basic service invoking process



1. After the service provider instance is started, it registers its services with the registration center.

   ZooKeeper provides the registration center and manages the list containing the mappings between services and service addresses.
2. The service consumer subscribes to the specific service address list from the registration center when the consumer starts.
3. The registration pushes the changes in the service address list to the clients of related services.
4. The service consumer selects a service address based on service management policies and accesses the service.

5.5 DBService

5.5.1 DBService Basic Principles

Overview

DBService is a HA storage system for relational databases, which is applicable to the scenario where a small amount of data (about 10 GB) needs to be stored, for example, component metadata. DBService can only be used by internal components of a cluster and provides data storage, query, and deletion functions.

DBService is a basic component of a cluster. Components such as Hive, Hue, Oozie, Loader, Redis, CDL, Flink, HuteEngine, Kafka, and Ranger store their metadata in DBService, and provide the metadata backup and restoration functions using DBService.

DBService Architecture

DBService in the cluster works in active/standby mode. Two DBServer instances are deployed and each instance contains three modules: HA, Database, and FloatIP.

Figure 5-10 shows the DBService logical architecture.

Figure 5-10 DBService architecture

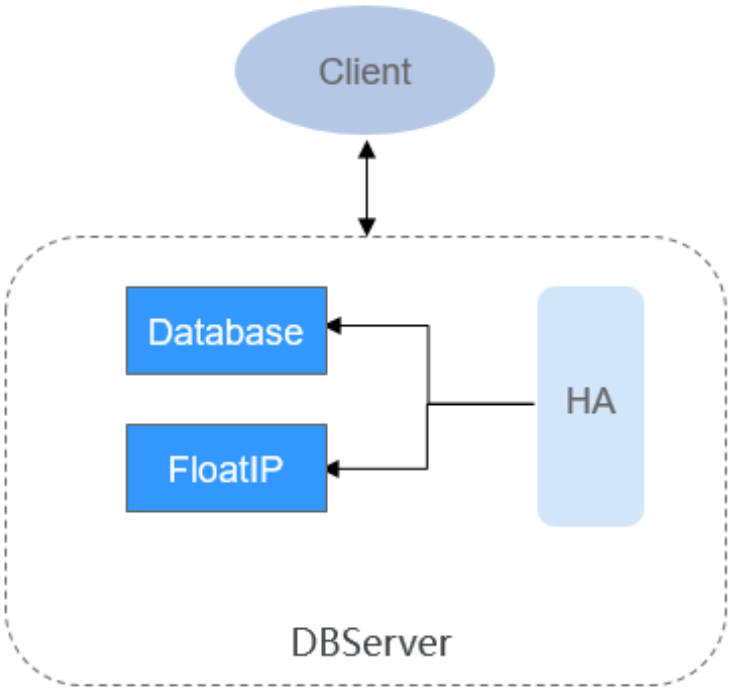


Table 5-3 describes the modules shown in **Figure 5-10**

Table 5-3 Module description

Name	Description
HA	HA management module. The active/standby DBServer uses the HA module for management.
Database	Database module. This module stores the metadata of the Client module.
FloatIP	Floating IP address that provides the access function externally. It is enabled only on the active DBServer instance and is used by the Client module to access Database.
Client	Client using the DBService component, which is deployed on the component instance node. The client connects to the database by using FloatIP and then performs metadata adding, deleting, and modifying operations.

5.5.2 Relationship Between DBService and Other Components

DBService is a basic component of a cluster. Components such as Hive, Hue, Oozie, Loader, and Redis, and Loader store their metadata in DBService, and provide the metadata backup and restoration functions by using DBService.

5.6 Apache Doris

5.6.1 Basic Principles

Introduction to Doris

Doris is a high-performance, real-time analytical database based on the MPP architecture, known for its extreme speed and ease of use. It returns query results of large-scale data in sub-seconds and supports high-concurrency point queries and high-throughput complex analysis. All this makes Doris an ideal tool for report analysis, ad-hoc query, unified data warehouse, and data lake query acceleration. On Doris, users can build various applications, such as user behavior analysis, AB test platform, log retrieval analysis, user portrait analysis, and order analysis. For more information, see **Apache Doris**.

Doris Architecture

The following figure shows the overall architecture of Doris. Both frontend (FE) and backend (BE) nodes are horizontally scalable.

Figure 5-11 Doris architecture

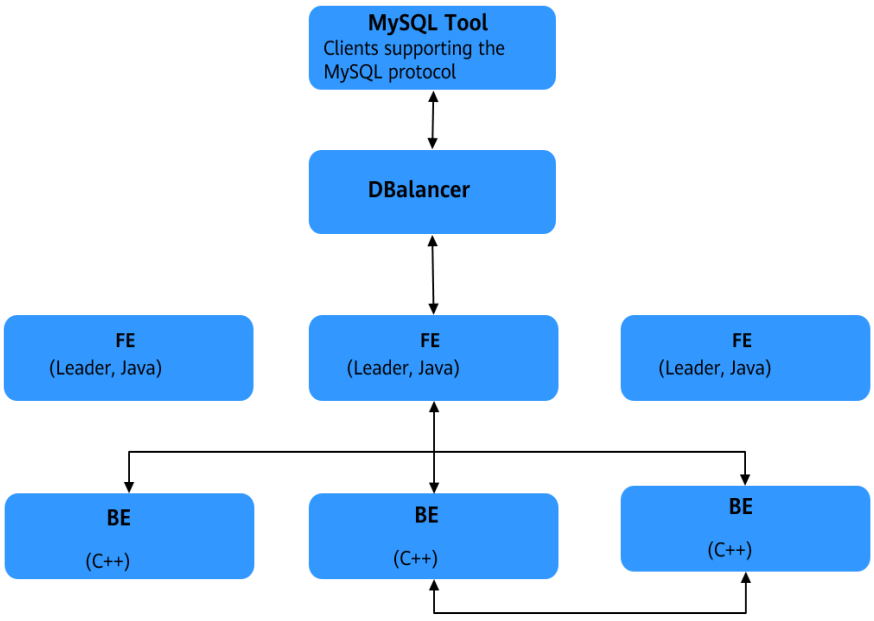


Table 5-4 Description

Parameter	Description
MySQL Tools	Doris is fully compatible with MySQL dialect and can be accessed by various client tools. It also supports standard SQL and can seamlessly connect to BI tools.
FE	Frontend nodes process user access requests, plan query parsing, and manage metadata and nodes.
BE	Backend nodes store data, execute query plans, and balance load among copies.
Leader	Leader is a role elected from Follower nodes.
Follower	Follower nodes receive metadata logs, which must be written successfully in most nodes.
DBalancer	DBalancer forwards TCP and HTTPS requests to FE nodes based on configured policies to balance data connections between FE nodes. When you need to run client commands to connect to FE nodes or access the web UI of an FE node, you can use the DBalancer instance to access FE nodes instead. The method is the same as directly accessing the FE node. You only need to change the port to that of the DBalancer, the DBalancer instance forwards requests to FE nodes and balances loads between FE nodes.

Doris uses the MPP model for inter-node and intra-node parallel execution, making it suitable for distributed joins of large tables.

It also supports vectorized query execution engines, adaptive query execution (AQE) technology, optimization strategies that combine CBO and RBO, and hot data cache queries.

Basic Concepts

In Doris, data is logically described in the form of tables.

- **Rows and Columns**
  A table consists of rows and columns.

  - Row: a row of user data.
  - Column: different fields in a row of data.

  Columns can be classified into two types: keys and values. From the service perspective, Key and Value correspond to dimension columns and metric columns, respectively. In the aggregation model, rows with the same Key column are aggregated into one row. How Value columns are aggregated as specified by a user when the table is created.

- **Tablets and Partitions**

  In the Doris storage engine, user data is horizontally divided into several tablets (also called data buckets). Each tablet contains several rows of data. The data between the individual tablets does not intersect and is physically stored independently.

  Multiple tablets logically belong to different partitions. A tablet belongs to only one partition, but a partition can contain multiple tablets. Since the tablets are physically stored independently, the partitions can be seen as physically independent, too. Tablet is the smallest physical storage unit for data operations such as movement and replication.

  Multiple partitions form a table. A partition can be regarded as the smallest logical unit for management. Data can be imported or deleted only for one partition.

- **Data Models**

  Doris data models are classified into three types: Aggregate, Unique, and Duplicate.

  - **Aggregate Model**

    When data is imported, rows with the same Key column are aggregated, and the Value columns are aggregated based on the AggregationType configured by users. AggregationType has the following modes:

    - SUM: Sum up the values in multiple rows.
    - REPLACE: Replace the previous value with the newly imported value.
    - MAX: Keep the maximum value.
    - MIN: Keep the minimum value.

  - **Unique Model**

    In some multi-dimensional analysis scenarios, users are highly concerned about how to create uniqueness constraints for the Primary Key. The Unique model is introduced to solve this problem.

    - **Merge on Read**

      The merge on read implementation in the Unique model is equivalent to Replace implementation in the Aggregate model. The internal implementation and data storage method are the same.

    - **Merge on Write**

      The Merge on Write implementation of the Unique model is completely different from that of the Aggregate model. It can deliver better performance (almost like that of the Duplicate model) in aggregation queries with primary key limitations. This

implementation is particularly suitable for aggregation queries and those using indexes to filter out large scale data.

In a Unique table where Merge on Write is enabled, overwritten and updated data is marked and deleted during data import, and new data is written to a new file. During a query, all data marked for deletion is filtered out at the file level, and the read data is the latest data. This eliminates the data aggregation process in Merge on Read and supports pushdown of multiple predicates in many cases. Performance can be greatly improved in many scenarios, especially in the case of aggregation queries.

- **Duplicate Model**

  In some multi-dimensional analysis scenarios, primary keys and data aggregation are not required. Duplicate models can be introduced to meet such requirements.

  Different from the Aggregate and Unique models, the Duplicate model stores the data as they are and executes no aggregation. Even if there are two identical rows of data, they will both be retained. The **DUPLICATE KEY** in the **CREATE TABLE** statement is only used to specify based on which columns the data are sorted.

- **Data Model Selection**

  The data model is established when the table is created and cannot be modified. Therefore, it is important to select a proper data model.

  - The Aggregate model aggregates data in advance, greatly reducing data scanning and calculation workload. Therefore, it is suitable for reporting query business, which has fixed schema. However, this model is not user-friendly for **count(*)** queries. Since the aggregation method on the Value column is fixed, semantic correctness should be considered in other types of aggregation queries.

  - The Unique model ensures that the primary key is unique when it is required. However, pre-aggregation such as Rollup cannot be used in this case.

    The Unique model supports only the update of an entire row. If you need to update both the unique primary key constraint and some columns (for example, importing multiple source tables to one Doris table), you can use the Aggregate model and set the aggregation type of non-primary key columns to **REPLACE_IF_NOT_NULL**.

  - Duplicate is suitable for ad-hoc queries in any dimension. Although pre-aggregation cannot be used, Duplicate is not restricted by the aggregation model and can make full use of the advantages of the column-store model, that is, only related columns are read, and not all key columns need to be read.

5.6.2 Relationship with Other Components

HDFS

Doris can import and export HDFS data and directly query HDFS data sources.

Hudi

Doris can directly query Hudi data sources.

Spark

Spark Doris Connector allows Spark to read data stored in Doris and write data to Doris.

Flink

Flink Doris Connector allows you to perform operations (read, insert, modify, and delete) on data stored in Doris through Flink.

Hive

Doris can directly query Hive data sources.

Kafka

Doris can import Kafka data.

5.7 Elasticsearch

5.7.1 Elasticsearch Basic Principles

Elasticsearch Architecture

The Elasticsearch cluster solution consists of the EsMaster and EsClient, EsNode1, EsNode2, EsNode3, EsNode4, EsNode5, EsNode6, EsNode7, EsNode8, and EsNode9 processes, as shown in **Figure 5-12**. **Table 5-5** describes the modules.

Figure 5-12 Elasticsearch architecture

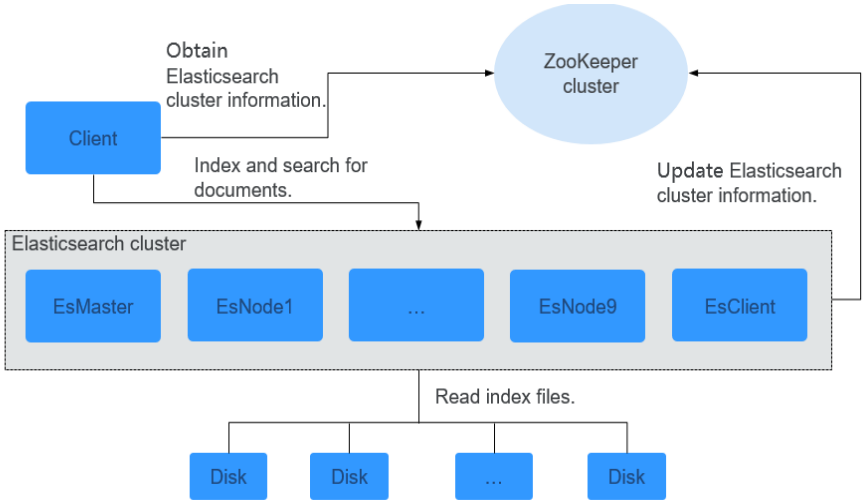


Table 5-5 Module description

Module	Description
Client	Client communicates with the EsClient and EsNode instance processes in the Elasticsearch cluster over HTTP or HTTPS to perform distributed collection and search.
EsMaster	EsMaster is the master node of Elasticsearch. It manages the cluster, such as determining shard allocation and tracing cluster nodes.
EsNode1-9	EsNodes 1-9 are data nodes of Elasticsearch. They store index data, and add, delete, modify, query, and aggregate documents.
EsClient	EsClient is the coordinator node of Elasticsearch. It processes routing requests, searches for data, and dispatches indexes. EsClient neither store data or manage clusters.
ZooKeeper cluster	ZooKeeper provides functions such as storage of security authentication information for Elasticsearch.

Basic Concepts

- Index: An index is a logical namespace in Elasticsearch, consisting of one or multiple shards. Apache Lucene is used to read and write data in the index. It is similar to a relational table instance. One Elasticsearch instance can contain multiple indexes.
- Document: A document is a basic unit of information that can be indexed. This document refers to JSON data at the top-level structure or obtained by serializing the root object. The document is similar to a row in the database. An index contains multiple documents.
- Mapping: A mapping is used to restrict the type of a field and can be automatically created based on data. It is similar to the schema in the database.
- Field: A field is the minimum unit of a document, which is similar to a column in the database. Each document contains multiple fields.
- EsMaster: The master node that temporarily manages some cluster-level changes, such as creating or deleting indexes, and adding or removing nodes. The master node does not participate in document-level change or search. When traffic increases, the master node does not become the bottleneck of the cluster.
- EsNode: an Elasticsearch node. A node is an Elasticsearch instance.
- EsClient: an Elasticsearch node. It processes routing requests, searches for data, and dispatches indexes. It does not store data or manage a cluster.
- Shard: A shard is the smallest work unit in Elasticsearch. It stores documents that can be referenced in the shard.
- Primary shard: Each document in the index belongs to a primary shard. The number of primary shards determines the maximum data that can be stored in the index.

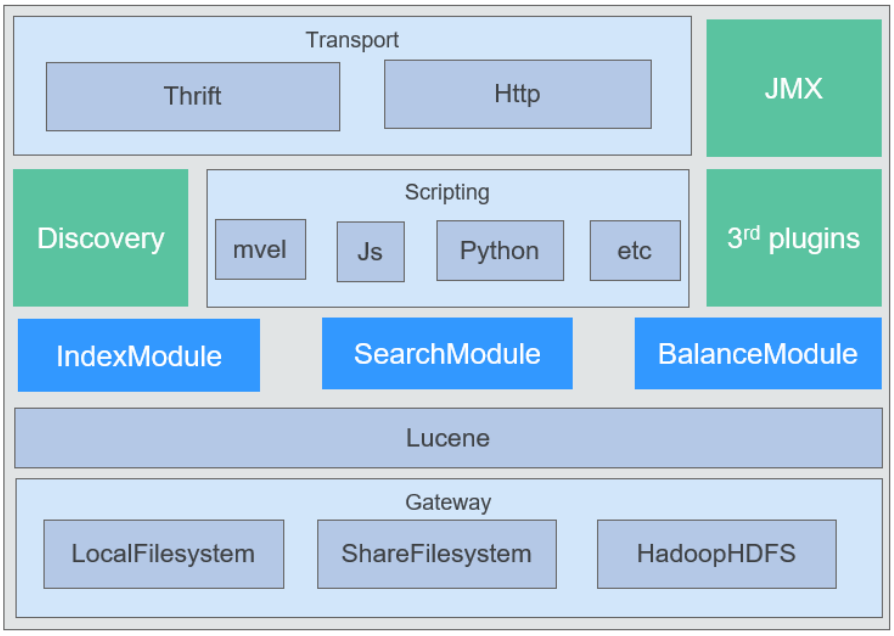
- Replica shard: A replica shard is a copy of the primary shard. It prevents data loss caused by hardware faults and provides read requests, such as searching for or retrieving documents from other shards.
- Recovery: Indicates data restoration or data redistribution. When a node is added or deleted, Elasticsearch redistributes index shards based on the load of the corresponding physical server. When a faulty node is restarted, data recovery is also performed.
- Gateway: Indicates the storage mode of an Elasticsearch index snapshot. By default, Elasticsearch stores an index in the memory. When the memory is full, Elasticsearch saves the index to the local hard disk. A gateway stores index snapshots. When the corresponding Elasticsearch cluster is stopped and then restarted, the index backup data is read from the gateway. Elasticsearch supports multiple types of gateways, including local file systems (default), distributed file systems, and Hadoop HDFS.
- Transport: Indicates the interaction mode between Elasticsearch internal nodes or clusters and the Elasticsearch client. By default, Transmission Control Protocol (TCP) is used for interaction. In addition, HTTP (JSON format), Thrift, Servlet, Memcached, and ZeroMQ transmission protocols (integrated through plug-ins) are supported.
- ZooKeeper cluster: It is mandatory in Elasticsearch and provides functions such as storage of security authentication information.

Elasticsearch Principles

- **Elasticsearch internal architecture**

  Elasticsearch provides various access APIs through RESTful APIs or other languages (such as Java), uses the cluster discovery mechanism, and supports script languages and various plug-ins. The underlying layer is based on Lucene, with absolute independence of Lucene, and stores indexes through local files, shared files, and HDFS, as shown in **Figure 5-13**.

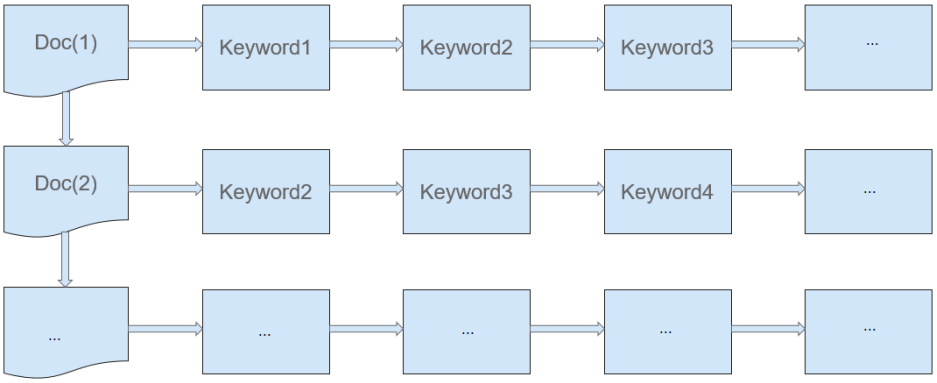
  Figure 5-13 Internal architecture



- **Inverted indexing**

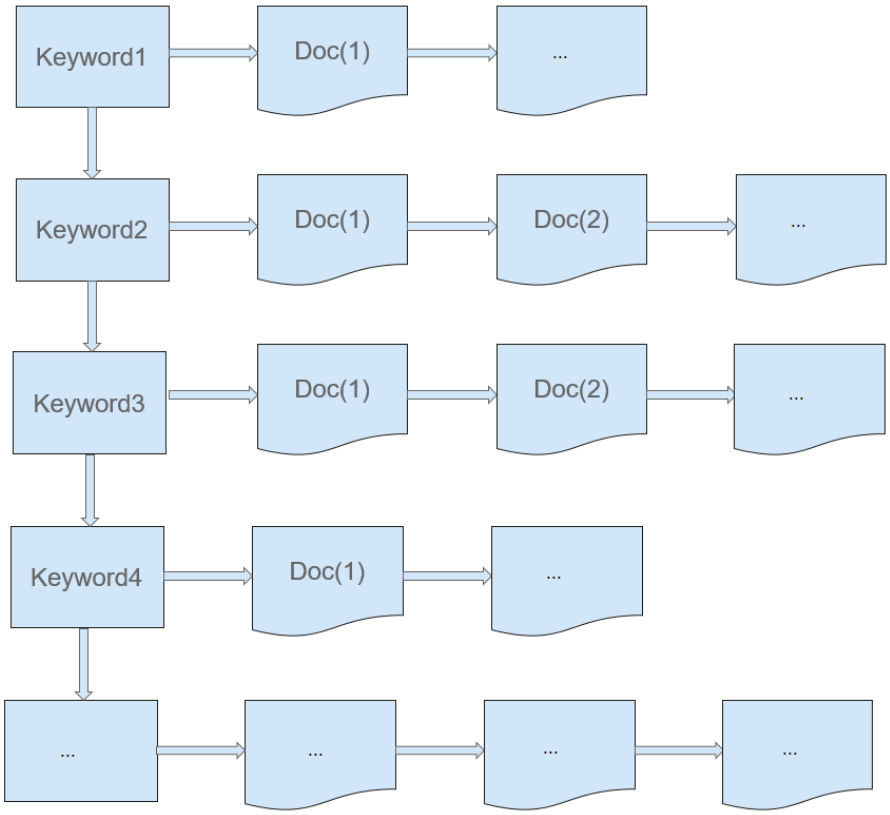
  In the traditional search mode (forward indexing, as shown in **Figure 5-14**), documents are searched based on their IDs. During the search, keywords of each document are scanned to find the keywords that meet the search criteria. Forward indexing is easy to maintain but is time consuming.

  Figure 5-14 Forward indexing

  Elasticsearch (Lucene) uses the inverted indexing mode, as shown in **Figure 5-15**. A table consisting of different keywords is called a dictionary, which contains various keywords and statistics of the keywords (including the ID of the document where a keyword is located, the location of the keyword in the document, and the frequency of the keyword). In this search mode, Elasticsearch searches for the document ID and location based on a keyword and then finds the document, which is similar to the method of looking for a word in a dictionary or finding the content on a specific book page according to the table of contents of the book. Inverted indexing is time consuming for constructing indexes and costly for maintenance, but it is efficient in search.

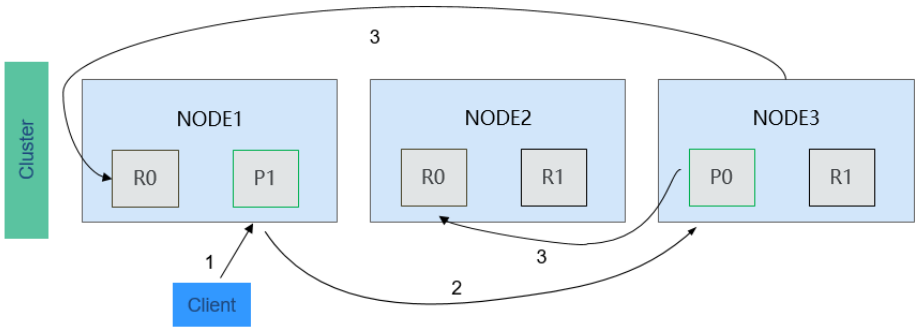
Figure 5-15 Inverted indexing



- **Elasticsearch Distributed Indexing**

  Figure 5-16 shows the process of Elasticsearch distributed indexing flow.

  Figure 5-16 Distributed indexing flow



  The procedure is as follows:

  Phase 1: The client sends an index request to any node, for example, Node 1.

  Phase 2: Node 1 determines the shard (for example, shard 0) to store the file based on the request. Node 1 then forwards the request to Node 3 where primary shard P0 of shard 0 exists.

  Phase 3: Node 3 executes the request on primary shard P0 of shard 0. If the request is successfully executed, Node 3 sends the request to all the replica shard R0 in Node 1 and Node 2 concurrently. If all the replica shards successfully execute the request, a verification message is returned to Node 3.

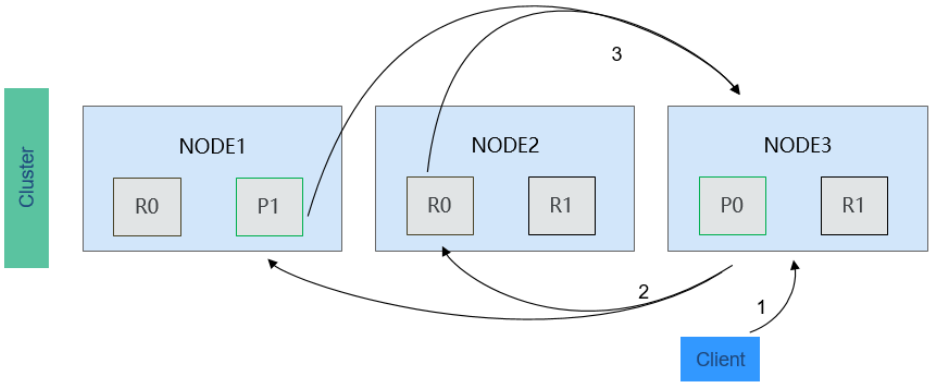
After receiving the verification messages from all the replica shards, Node 3 returns a success message to the user.

- **Elasticsearch Distributed Searching**

  The Elasticsearch distributed searching flow consists of query and acquisition.

  Figure 5-17 shows the query phase.

  Figure 5-17 Query phase of the distributed searching flow



  The procedure is as follows:

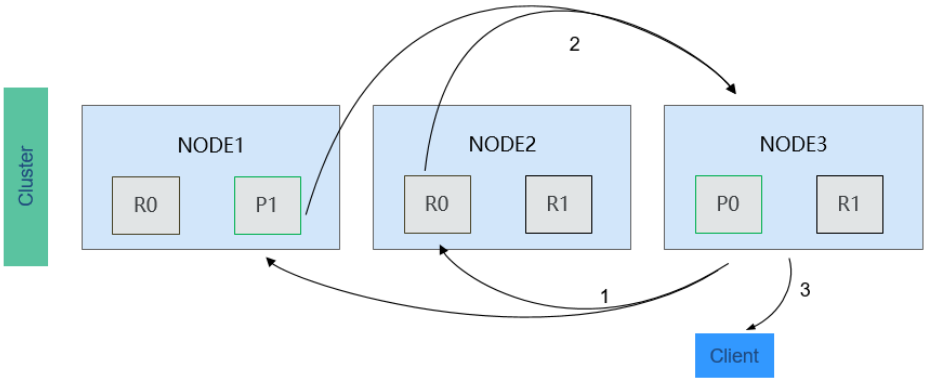
  Phase 1: The client sends a retrieval request to any node, for example, Node 3.

  Phase 2: Node 3 sends the retrieval request to each shard in the index adopting the polling policy. One of the primary shards and all of its replica shards is randomly selected to balance the read request load. Each shard performs retrieval locally and adds the sorting result to the local node.

  Phase 3: Each shard returns the local result to Node 3. Node 3 combines these values and performs global sorting.

  In the query phase, the data to be retrieved is located. In the acquisition phase, these data will be collected and returned to the client. **Figure 5-18** shows the acquisition phase.

  Figure 5-18 Acquisition phase of the distributed searching flow



  The procedure is as follows:

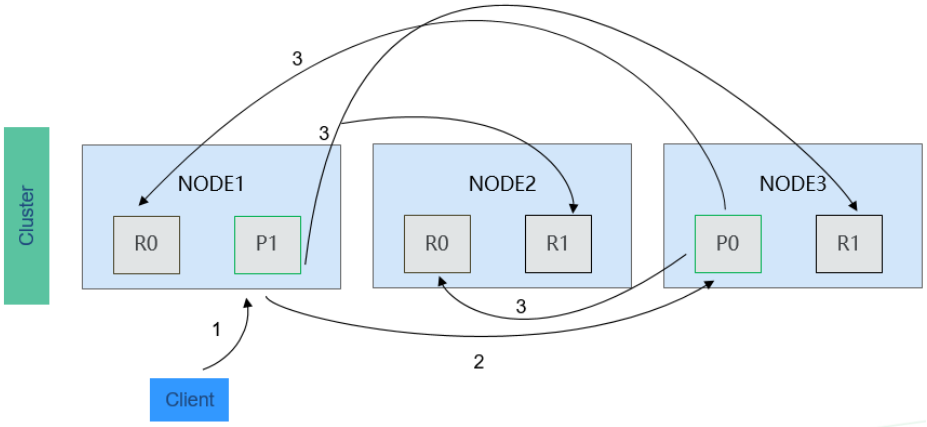
  Phase 1: After all data to be retrieved is located, Node 3 sends a request to related shards.

Phase 2: Each shard that receives the request from Node 3 reads the related files and return them to Node 3.

Phase 3: After obtaining all the files returned by the shards, Node 3 combines them into a summary result and returns it to the client.

- **Elasticsearch Distributed Bulk Indexing**

Figure 5-19 Distributed bulk indexing flow



The procedure is as follows:

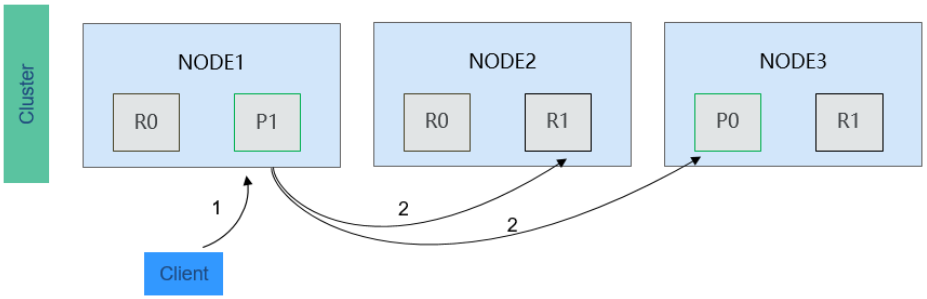
Phase 1: The client sends a bulk request to Node 1.

Phase 2: Node 1 constructs a bulk request for each shard and forwards the requests to the primary shard according to the request.

Phase 3: The primary shard executes the requests one by one. After an operation is complete, the primary shard forwards the new file (or deleted part) to the corresponding replication node and then performs the next operation. Replica nodes report to the request node that all operations are complete. The request node sorts the response and returns it to the client.

- **Elasticsearch Distributed Bulk Searching**

Figure 5-20 Distributed bulk searching flow



The procedure is as follows:

Phase 1: The client sends an **mget** request to Node 1.

Phase 2: Node 1 constructs a retrieval request of multi-piece data records for each shard and forwards the requests to the primary shard or its replica shard based on the requests. When all replies are received, Node 1 constructs a response and returns it to the client.

- **Elasticsearch Routing Algorithm**

  Elasticsearch provides two routing algorithms:

  – Default route: **shard=hash (routing) %number_of_primary_shards**.

  – Custom route: In this routing mode, the routing can be specified to determine the shard to which the file is written, or only the specified routing can be searched.

- **Elasticsearch Balancing Algorithm**

  Elasticsearch provides the automatic balance function for capacity expansion, capacity reduction, and data import scenarios. The algorithm is as follows:

  weight_index(node, index) = indexBalance * (node.numShards(index) - avgShardsPerNode(index))

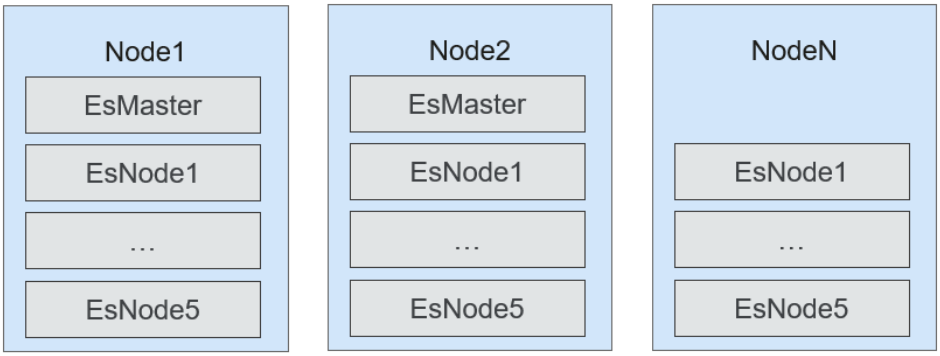
  Weight_node(node, index) = shardBalance * (node.numShards() - avgShardsPerNode)

  weight(node, index) = weight_index(node, index) + weight_node(node, index)

- **Elasticsearch Multi-Instance Deployment on a Node**

  Multiple Elasticsearch instances can be deployed on the same node, and differentiated from each other based on the IP address and port number. This method increases the usage of the single-node CPU, memory, and disk, and improves the Elasticsearch indexing and searching capability.

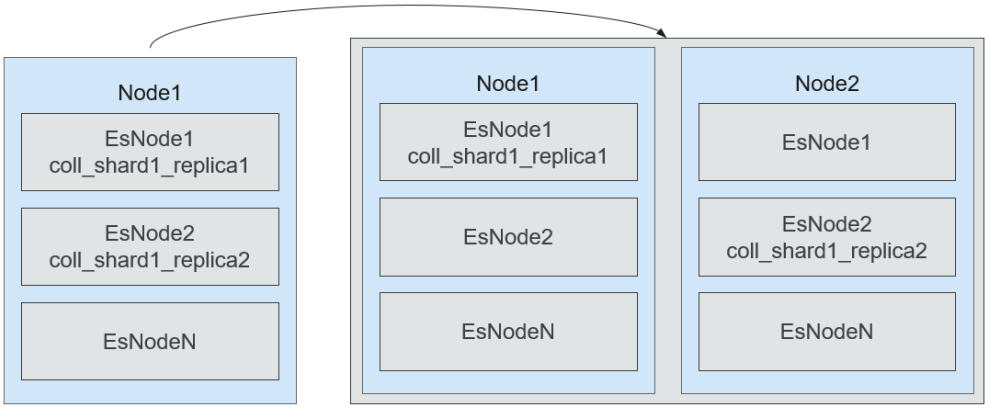
  Figure 5-21 Multi-instance deployment on a node



- **Elasticsearch Cross-Node Replica Allocation Policy**

  When multiple instances are deployed on a node and multiple replicas exist, replicas can only be allocated across instances. However, SPOFs may occur. To solve this problem, set **cluster.routing.allocation.same_shard.host** to **true**.

Figure 5-22 Automatic replica distribution across nodes



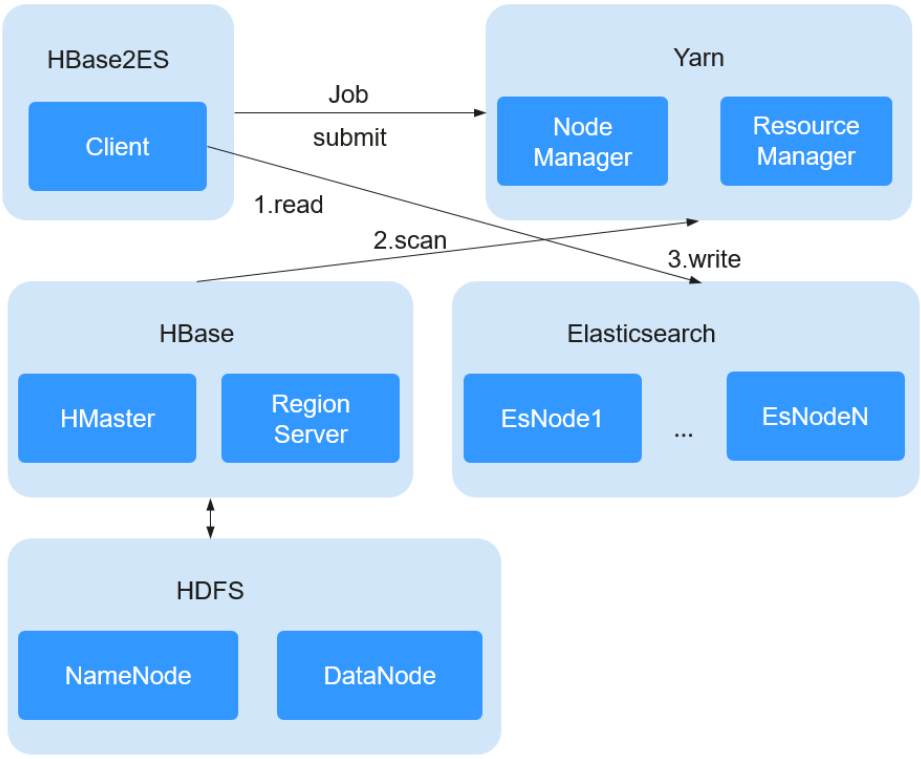
5.7.2 Relationship with Other Components

Elasticsearch Indexing HBase Data

When Elasticsearch indexes the HBase data, the HBase data is written to the HDFS and meanwhile Elasticsearch creates the corresponding HBase index data. The index ID is mapped to the rowkey of the HBase data, which ensures the unique mapping between each index data record and HBase data and implements full-text searching of the HBase data.

Batch indexing: For data already existing in HBase, an MR task is submitted to read all data in HBase, and then indexes are created in Elasticsearch. **Figure 5-23** shows the indexing process.

Figure 5-23 Elasticsearch indexing HBase data



5.7.3 Elasticsearch Enhanced Open Source Features

Elasticsearch Enhanced Open Source Features

- Enhanced Usability, Security, and Reliability
  - Monitors the memory, CPU, disk I/O, as well as index and shard status of Elasticsearch instances and manages alarms.
  - Provides the index permission control based on user/role in security mode.
  - Provides the Kerberos authentication to ensure the index data security.
- **Multi-Instance Deployment**: A maximum of 11 Elasticsearch instances can be deployed on each node.
- **IK analyzer integration**: This version integrates the IK analyzer that can be directly used. Provides the dynamic dictionary validation function.
- Data Import and Export Tool
  - HBase data can be imported to Elasticsearch using the HBase2ES tool.
  - Data between two Elasticsearch clusters can be migrated using the ES2ES tool.
  - Use the HDFS 2ES tool to import the formatted data from HDFS to Elasticsearch.

5.8 Apache Flink

5.8.1 Flink Basic Principles

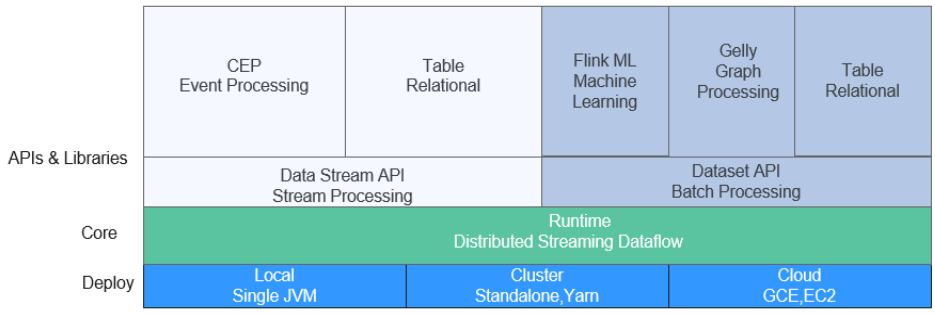
Overview

Flink is a unified computing framework that supports both batch processing and stream processing. It provides a stream data processing engine that supports data distribution and parallel computing. Flink features stream processing and is a top open source stream processing engine in the industry.

Flink provides high-concurrency pipeline data processing, millisecond-level latency, and high reliability, making it extremely suitable for low-latency data processing.

Figure 5-24 shows the technology stack of Flink.

Figure 5-24 Technology stack of Flink



Flink provides the following features in the current version:

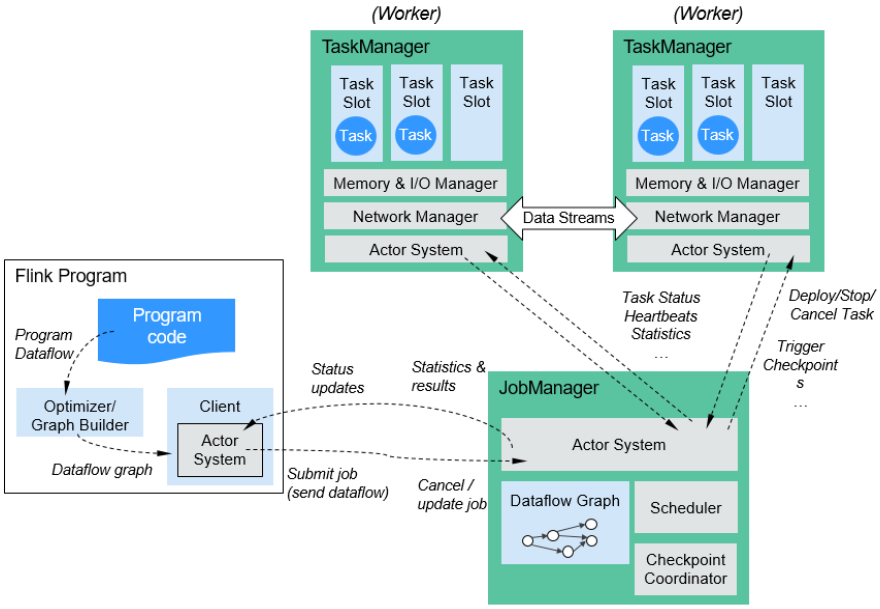
- DataStream
- Checkpoint
- Window
- Job Pipeline
- Configuration Table

Other features are inherited from the open source community and are not enhanced. For details, visit **https://ci.apache.org/projects/flink/flink-docs-release-1.12/**.

Flink Architecture

Figure 5-25 shows the Flink architecture.

Figure 5-25 Flink architecture



As shown in the above figure, the entire Flink system consists of three parts:

- Client

  Flink client is used to submit jobs (streaming jobs) to Flink.

- TaskManager

  TaskManager is a service execution node of Flink. It executes specific tasks. A Flink system can have multiple TaskManagers. These TaskManagers are equivalent to each other.

- JobManager

  JobManager is a management node of Flink. It manages all TaskManagers and schedules tasks submitted by users to specific TaskManagers. In high-availability (HA) mode, multiple JobManagers are deployed. Among these JobManagers, one is selected as the active JobManager, and the others are standby.

For more information about the Flink architecture, visit **https://ci.apache.org/projects/flink/flink-docs-master/docs/concepts/flink-architecture/**.

Flink Principles

- **Stream, transformation, and operators**

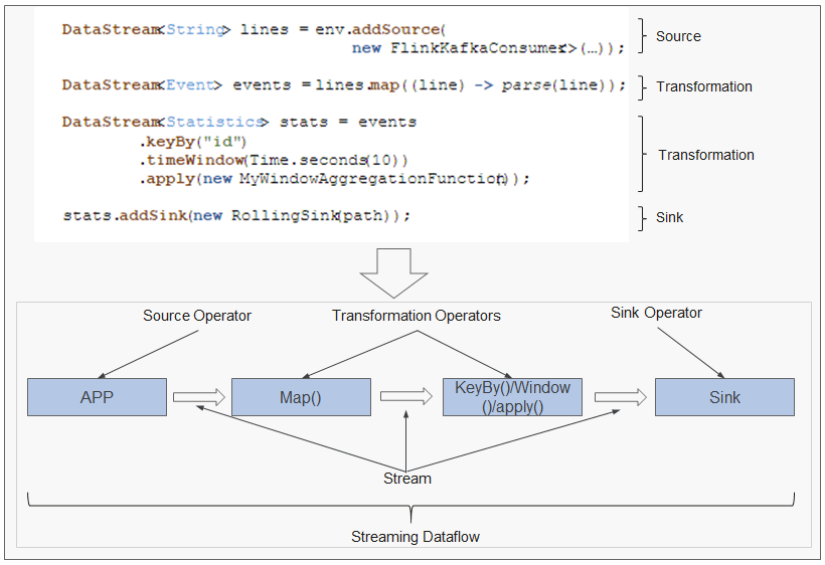
  A Flink program consists of two building blocks: stream and transformation.

  a. Conceptually, a stream is a (potentially never-ending) flow of data records, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result.

  b. When a Flink program is executed, it is mapped to a streaming dataflow. A streaming dataflow consists of a group of streams and transformation operators. Each dataflow starts with one or more source operators and ends in one or more sink operators. A dataflow resembles a directed acyclic graph (DAG).

  Figure 5-26 shows the streaming dataflow to which a Flink program is mapped.

  Figure 5-26 Example of Flink DataStream



  As shown in **Figure 5-26**, **FlinkKafkaConsumer** is a source operator; Map, KeyBy, TimeWindow, and Apply are transformation operators; RollingSink is a sink operator.

- **Pipeline dataflow**

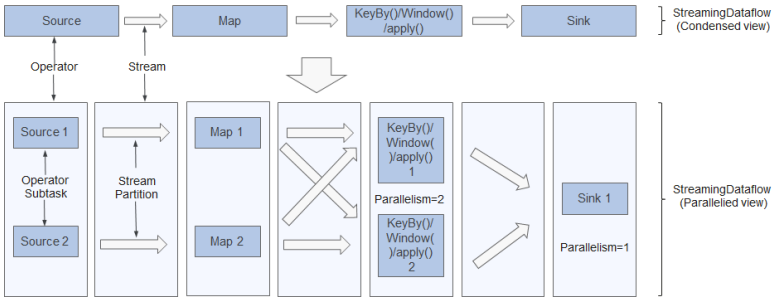
  Applications in Flink can be executed in parallel or distributed modes. A stream can be divided into one or more stream partitions, and an operator can be divided into multiple operator subtasks.

  The executor of streams and operators are automatically optimized based on the density of upstream and downstream operators.

  – Operators with low density cannot be optimized. Each operator subtask is separately executed in different threads. The number of operator subtasks is the parallelism of that particular operator. The parallelism (the total number of partitions) of a stream is that of its producing operator.

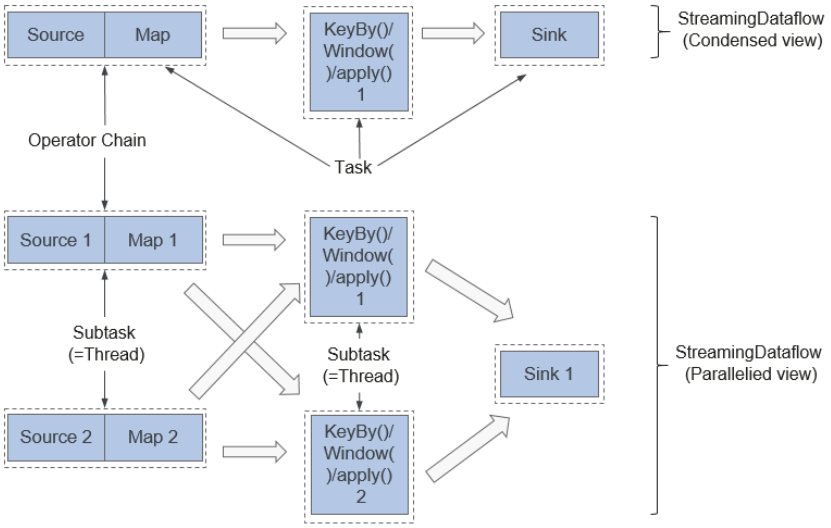
Different operators of the same program may have different levels of parallelism, as shown in **Figure 5-27**.

Figure 5-27 Operator



- Operators with high density can be optimized. Flink chains operator subtasks together into a task, that is, an operator chain. Each operator chain is executed by one thread on TaskManager, as shown in **Figure 5-28**.

Figure 5-28 Operator chain



  - In the upper part of **Figure 5-28**, the condensed Source and Map operators are chained into an Operator Chain, that is, a larger operator. The Operator Chain, KeyBy, and Sink all represent an operator respectively and are connected with each other through streams. Each operator corresponds to one task during the running. Namely, there are three tasks in the upper part.
  - In the lower part of **Figure 5-28**, each task, except Sink, is paralleled into two subtasks. The parallelism of the Sink operator is one.

Key Features

- Stream processing

The real-time stream processing engine features high throughput, high performance, and low latency, which can provide processing capability within milliseconds.

- Various status management

  The stream processing application needs to store the received events or intermediate result in a certain period of time for subsequent access and processing at a certain time point. Flink provides diverse features for status management, including:

  - Multiple basic status types: Flink provides various states for data structures, such as ValueState, ListState, and MapState. Users can select the most efficient and suitable status type based on the service model.
  - Rich State Backend: State Backend manages the status of applications and performs Checkpoint operations as required. Flink provides different State Backends. State can be stored in the memory or RocksDB, and supports the asynchronous and incremental Checkpoint mechanism.
  - Exactly-once state consistency: The Checkpoint and fault recovery capabilities of Flink ensure that the application status of tasks is consistent before and after a fault occurs. Flink supports transactional output for some specific storage devices. In this way, exactly-once output can be ensured even when a fault occurs.

- Various time semantics

  Time is an important part of stream processing applications. For real-time stream processing applications, operations such as window aggregation, detection, and matching based on time semantics are quite common. Flink provides various time semantics.

  - Event-time: The timestamp provided by the event is used for calculation, making it easier to process the events that arrive at a random sequence or arrive late.
  - Watermark: Flink introduces the concept of Watermark to measure the development of event time. Watermark also provides flexible assurance for balancing processing latency and data integrity. When processing event streams with Watermark, Flink provides multiple processing options if data arrives after the calculation, for example, redirecting data (side output) or updating the calculation result.
  - Processing-time and Ingestion-time are supported.
  - Highly flexible streaming window: Flink supports the time window, count window, session window, and data-driven custom window. You can customize the triggering conditions to implement the complex streaming calculation mode.

- Fault tolerance mechanism

  In a distributed system, if a single task or node breaks down or becomes faulty, the entire task may fail. Flink provides a task-level fault tolerance mechanism, which ensures that user data is not lost when an exception occurs in a task and can be automatically restored.

  - Checkpoint: Flink implements fault tolerance based on checkpoint. Users can customize the checkpoint policy for the entire task. When a task fails, the task can be restored to the status of the latest checkpoint and data after the snapshot is resent from the data source.

  - Savepoint: A savepoint is a consistent snapshot of application status. The savepoint mechanism is similar to that of checkpoint. However, the savepoint mechanism needs to be manually triggered. The savepoint mechanism ensures that the status information of the current stream application is not lost during task upgrade or migration, facilitating task suspension and recovery at any time point.

- Flink SQL

  Table APIs and SQL use Apache Calcite to parse, verify, and optimize queries. Table APIs and SQL can be seamlessly integrated with DataStream and DataSet APIs, and support user-defined scalar functions, aggregation functions, and table value functions. The definition of applications such as data analysis and ETL is simplified. The following code example shows how to use Flink SQL statements to define a counting application that records session times.

  ```
  SELECT userId, COUNT(*)
  FROM clicks
  GROUP BY SESSION(clicktime, INTERVAL '30' MINUTE), userId
  ```

  For more information about Flink SQL, see **https://ci.apache.org/projects/flink/flink-docs-master/dev/table/sqlClient.html**.

- CEP in SQL

  Flink allows users to represent complex event processing (CEP) query results in SQL for pattern matching and evaluate event streams on Flink.

  CEP SQL is implemented through the **MATCH_RECOGNIZE** SQL syntax. The **MATCH_RECOGNIZE** clause is supported by Oracle SQL since Oracle Database 12c and is used to indicate event pattern matching in SQL. The following is an example of CEP SQL:

  ```
  SELECT T.aid, T.bid, T.cid
  FROM MyTable
    MATCH_RECOGNIZE (
      PARTITION BY userid
      ORDER BY proctime
      MEASURES
        A.id AS aid,
        B.id AS bid,
        C.id AS cid
      PATTERN (A B C)
      DEFINE
        A AS name = 'a',
        B AS name = 'b',
        C AS name = 'c'
    ) AS T
  ```

5.8.2 Flink HA Solution

Overview

A Flink cluster has only one JobManager. This has the risks of single point of failures (SPOFs). There are three modes of Flink: Flink On Yarn, Flink Standalone, and Flink Local. Flink On Yarn and Flink Standalone modes are based on clusters and Flink Local mode is based on a single node. Flink On Yarn and Flink Standalone provide an HA mechanism. With such a mechanism, you can recover the JobManager from failures and thereby eliminate SPOF risks. This section describes the HA mechanism of the Flink On Yarn.

Flink supports the HA mode and job exception recovery that highly depend on ZooKeeper. If you want to enable the two functions, configure ZooKeeper in the **flink-conf.yaml** file in advance as follows:

```
high-availability: zookeeper
high-availability.zookeeper.quorum:  ZooKeeper IP address:24002
high-availability.storageDir: hdfs:///flink/recovery
```

Flink On Yarn

Flink JobManager and Yarn ApplicationMaster are in the same process. Yarn ResourceManager monitors ApplicationMaster. If ApplicationMaster is abnormal, Yarn restarts it and restores all JobManager metadata from HDFS. During the recovery, existing tasks cannot run and new tasks cannot be submitted. ZooKeeper stores JobManager metadata, such as information about jobs, to be used by the new JobManager. A TaskManager failure is listened and processed by the DeathWatch mechanism of Akka on JobManager. When a TaskManager fails, a container is requested again from Yarn and a TaskManager is created.

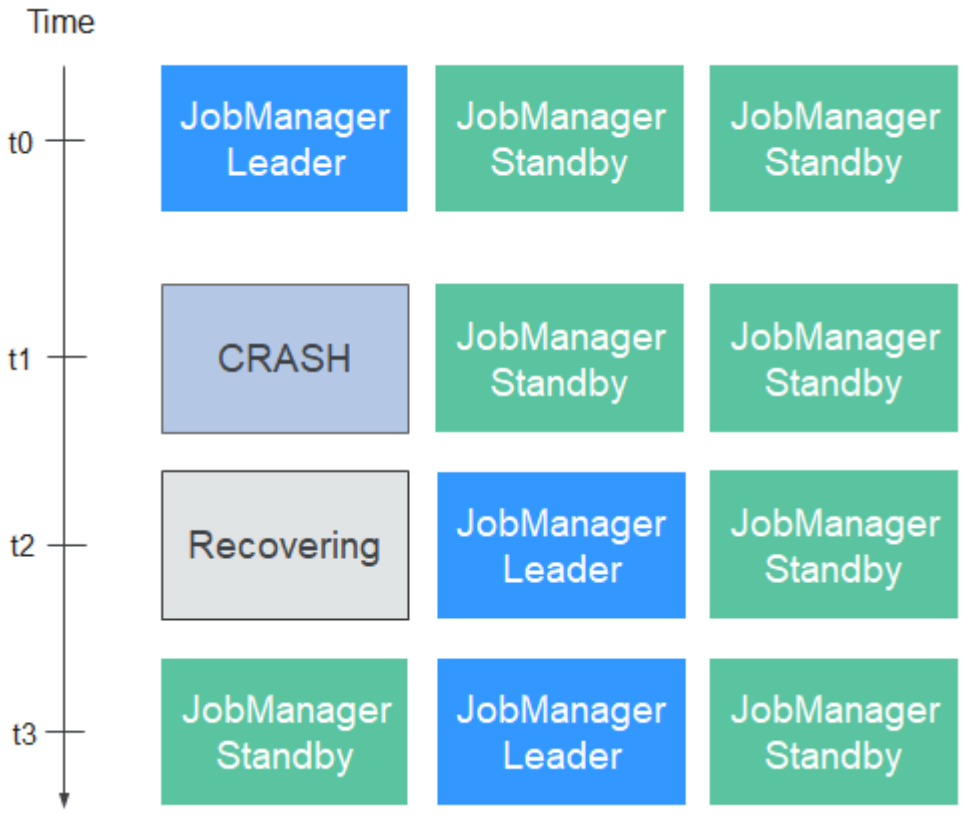
For more information about the HA solution of Flink on YARN, visit:

http://hadoop.apache.org/docs/r3.3.1/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html

Standalone

In the standalone mode, multiple JobManagers can be started and ZooKeeper elects one as the Leader JobManager. In this mode, there is a leader JobManager and multiple standby JobManagers. If the leader JobManager fails, a standby JobManager takes over the leadership. **Figure 5-29** shows the process of a leader/standby JobManager switchover.

Figure 5-29 Switchover process



Restoring TaskManager

A TaskManager failure is listened and processed by the DeathWatch mechanism of Akka on JobManager. If the TaskManager fails, the JobManager creates a TaskManager and migrates services to the created TaskManager.

Restoring JobManager

Flink JobManager and Yarn ApplicationMaster are in the same process. Yarn ResourceManager monitors ApplicationMaster. If ApplicationMaster is abnormal, Yarn restarts it and restores all JobManager metadata from HDFS. During the recovery, existing tasks cannot run and new tasks cannot be submitted.

Restoring jobs

To restore jobs, configure a restart policy in the Flink configuration file. Supported restart policies are **fixed-delay**, **failure-rate**, and **none**. Jobs can be restored only when the policy is configured to **fixed-delay** or **failure-rate**. If the restart policy is configured to **none** and **Checkpoint** is configured for **Job**, the restart policy is automatically configured to **fixed-delay** and the value of **restart-strategy.fixed-delay.attempts** specifies the number of retry times.

For details about the three strategies, visit the Flink official website at **https://ci.apache.org/projects/flink/flink-docs-release-1.17/dev/task_failure_recovery.html**. The configuration strategies are as follows:

```
restart-strategy: fixed-delay
restart-strategy.fixed-delay.attempts: 3
restart-strategy.fixed-delay.delay: 10 s
```


Jobs will be restored in the following scenarios:

- If a JobManager fails, all its jobs are stopped, and will be recovered after another JobManager is created and running.
- If a TaskManager fails, all tasks on the TaskManager are stopped, and will be started until there are available resources.
- When a task of a job fails, the job is restarted.

NOTE

For details about how to configure job restart strategies, visit **https://ci.apache.org/projects/flink/flink-docs-release-1.17/ops/jobmanager_high_availability.html**.

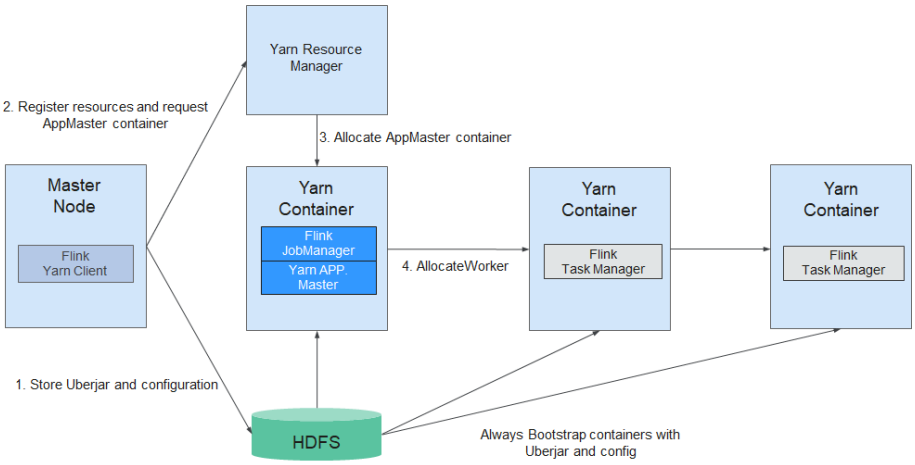
5.8.3 Relationship Between Flink and Other Components

Relationship Between Flink and YARN

Flink supports YARN-based cluster management mode. In this mode, Flink serves as an application of YARN and runs on YARN.

Figure 5-30 shows the YARN-based Flink cluster deployment.

Figure 5-30 YARN-based Flink cluster deployment



1. The Flink YARN Client first checks whether there are sufficient resources for starting the YARN cluster. If yes, the Flink YARN client uploads JAR files and configuration files to HDFS.
2. Flink YARN client communicates with YARN ResourceManager to request a container for starting ApplicationMaster. After all YARN NodeManagers finish downloading the JAR file and configuration files, the ApplicationMaster is started.
3. During the startup, the ApplicationMaster interacts with the YARN ResourceManager to request the container for starting a TaskManager. After the container is ready, the TaskManager process is started.
4. In the Flink YARN cluster, the ApplicationMaster and Flink JobManager are running in the same container. The ApplicationMaster informs each TaskManager of the RPC address of the JobManager. After TaskManagers are started, they register with the JobManager.

5. After all TaskManagers has registered with the JobManager, Flink starts up in the YARN cluster. Then, the Flink YARN client can submit Flink jobs to the JobManager, and Flink can perform mapping, scheduling, and computing for the jobs.

5.8.4 Flink Enhanced Open Source Features

5.8.4.1 Window

Enhanced Open Source Feature: Window

This section describes the sliding window of Flink and provides the sliding window optimization method. For details about windows, visit the official website at **https://ci.apache.org/projects/flink/flink-docs-release-1.17/dev/stream/operators/windows.html**.

Introduction to Window

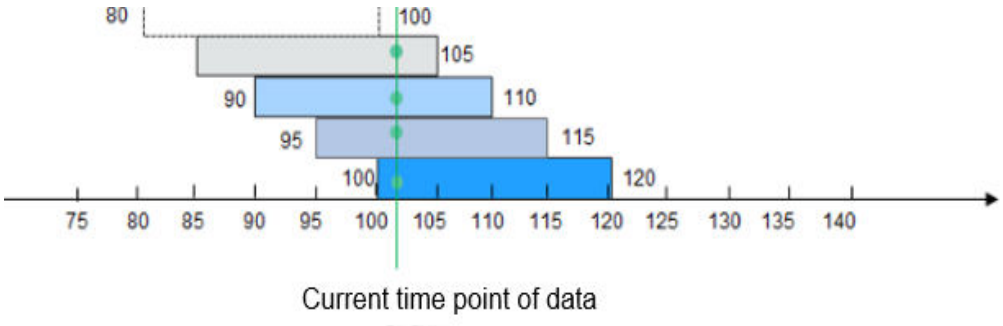
Data in a window is saved as intermediate results or original data. If you perform a sum operation (**window(SlidingEventTimeWindows.of(Time.seconds(20), Time.seconds(5))).sum**) on data in the window, only the intermediate result will be retained. If a custom window (**window(SlidingEventTimeWindows.of(Time.seconds(20), Time.seconds(5))).apply(new UDF)**) is used, all original data in the window will be saved.

If custom windows **SlidingEventTimeWindow** and **SlidingProcessingTimeWindow** are used, data is saved as multiple backups. Assume that the window is defined as follows:

```
window(SlidingEventTimeWindows.of(Time.seconds(20), Time.seconds(5))).apply(new UDFWindowFunction)
```

If a block of data arrives, it is assigned to four different windows (20/5 = 4). That is, the data is saved as four copies in the memory. When the window size or sliding period is set to a large value, data will be saved as excessive copies, causing redundancy.

Figure 5-31 Original structure of a window



If a data block arrives at the 102nd second, it is assigned to windows [85, 105), [90, 110), [95, 115), and [100, 120).

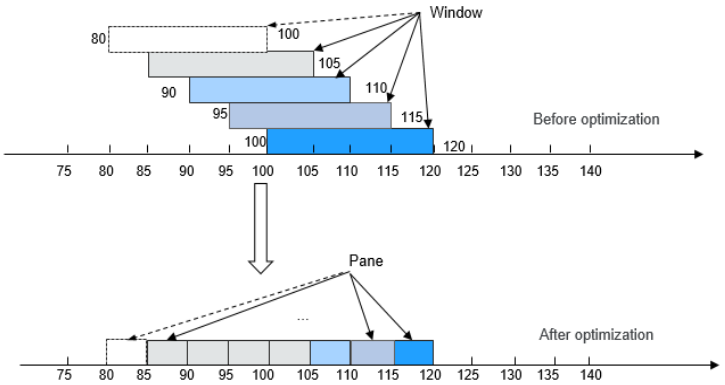
Window Optimization

As mentioned in the preceding, there are excessive data copies when original data is saved in SlidingEventTimeWindow and SlidingProcessingTimeWindow. To resolve this problem, the window that stores the original data is restructured, which optimizes the storage and greatly lowers the storage space. The window optimization scheme is as follows:

1. Use the sliding period as a unit to divide a window into different panes.

   A window consists of one or multiple panes. A pane is essentially a sliding period. For example, the sliding period (namely, the pane) of **window(SlidingEventTimeWindows.of(Time.seconds(20), Time.seconds.of(5)))** lasts for 5 seconds. If this window ranges from [100, 120), this window can be divided into panes [100, 105), [105, 110), [110, 115), and [115, 120).

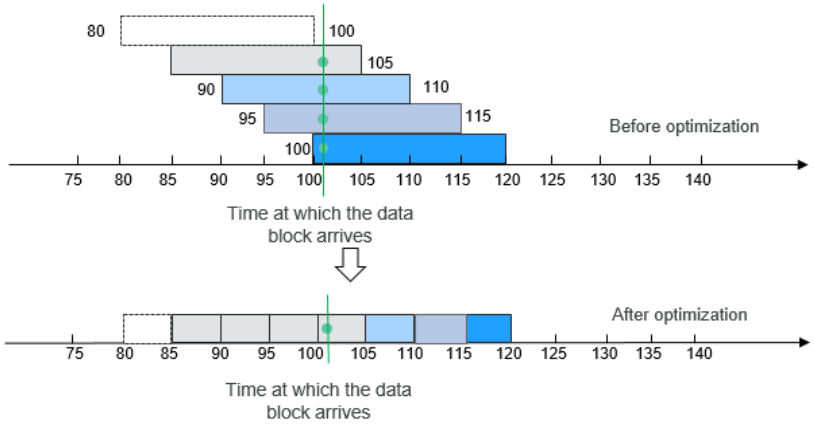
   Figure 5-32 Window optimization



2. When a data block arrives, it is not assigned to a specific window. Instead, Flink determines the pane to which the data block belongs based on the timestamp of the data block, and saves the data block into the pane.

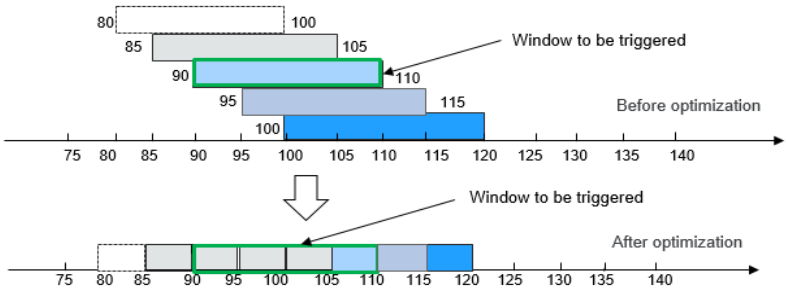
   A data block is saved only in one pane. In this case, only a data copy exists in the memory.

   Figure 5-33 Saving data in a window



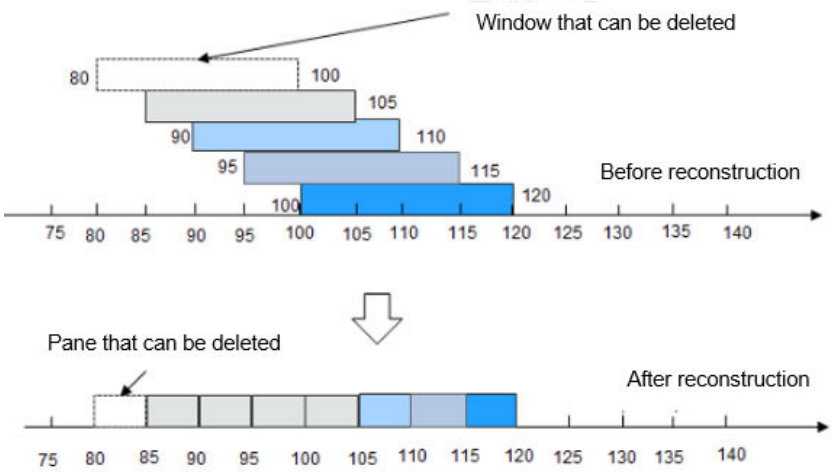
3. To trigger a window, compute all panes contained in the window, and combine all these panes into a complete window.

Figure 5-34 Triggering a window



4. If a pane is not required, you can delete it from the memory.

Figure 5-35 Deleting a window



After optimization, the quantity of data copies in the memory and snapshot is greatly reduced.

5.8.4.2 Job Pipeline

Enhanced Open Source Feature: Job Pipeline

Generally, logic code related to a service is stored in a large JAR package, which is called Fat JAR. Disadvantages of Fat JAR are as follows:

- When service logic becomes more and more complex, the size of the Fat JAR increases.
- Fat Jar makes coordination complex. Developers of all services are working with the same service logic. Even though the service logic can be divided into several modules, all modules are tightly coupled with each other. If the requirement needs to be changed, the entire flow diagram needs to be replanned.

Splitting of jobs is facing the following problems:

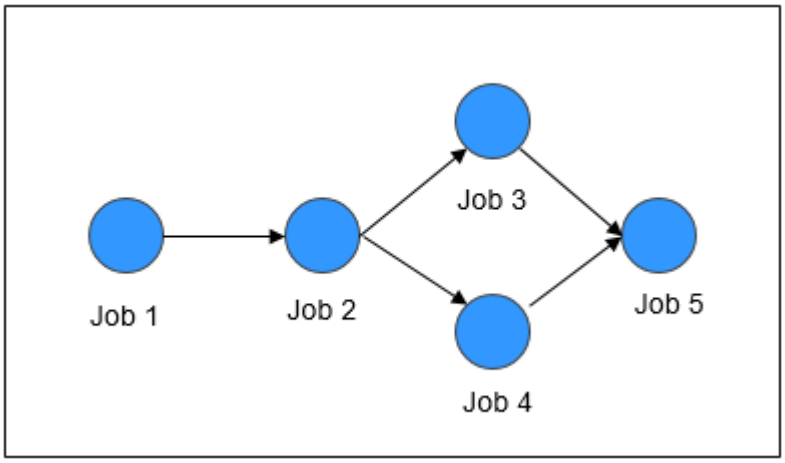
- Data transmission between jobs can be achieved using Kafka. For example, job A transmits data to the topic A in Kafka, and then job B and job C read data from the topic A in Kafka. This solution is easy to implement, but the latency is longer than 100 ms.

- Operators are connected using the TCP protocol. In distributed environment, operators can be scheduled to any node and upstream and downstream services cannot detect the scheduling.

Job Pipeline

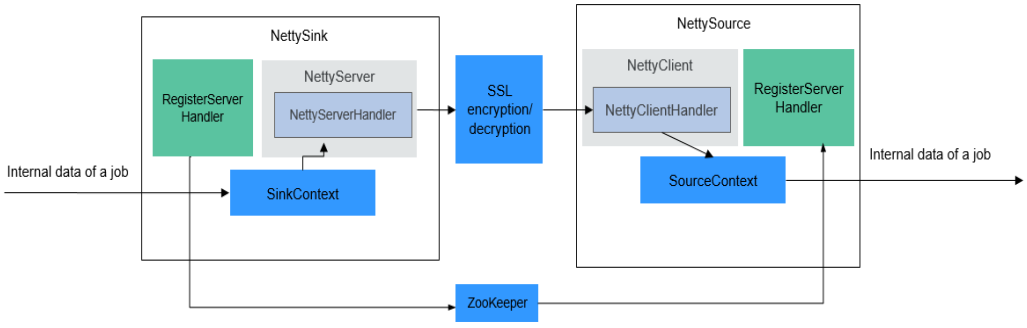
A pipeline consists of multiple Flink jobs connected through TCP. Upstream jobs can send data to downstream jobs. The flow diagram about data transmission is called a job pipeline, as shown in **Figure 5-36**.

Figure 5-36 Job pipeline



Job Pipeline Principles

Figure 5-37 Job pipeline principles



- NettySink and NettySource

  In a pipeline, upstream jobs and downstream jobs communicate with each other through Netty. The Sink operator of the upstream job works as a server and the Source operator of the downstream job works as a client. The Sink operator of the upstream job is called NettySink, and the Source operator of the downstream job is called NettySource.

- NettyServer and NettyClient

  NettySink functions as the server of Netty. In NettySink, NettyServer achieves the function of a server. NettySource functions as the client of Netty. In NettySource, NettyClient achieves the function of a client.

- Publisher

  The job that sends data to downstream jobs through NettySink is called a publisher.

- Subscriber

  The job that receives data from upstream jobs through NettySource is called a subscriber.

- RegisterServer

  RegisterServer is the third-party memory that stores the IP address, port number, and concurrency information about NettyServer.

- The general outside-in architecture is as follows:
  - NettySink->NettyServer->NettyServerHandler
  - NettySource->NettyClient->NettyClientHandler

Job Pipeline Functions

- **NettySink**

  NettySink consists of the following major modules:

  - RichParallelSinkFunction

    NettySink inherits RichParallelSinkFunction and attributes of Sink operators. The RichParallelSinkFunction API implements following functions:

    - Starts the NettySink operator.
    - Runs the NettySink operator and receives data from the upstream operator.
    - Cancels the running of NettySink operators.

    Following information can be obtained using the attribute of RichParallelSinkFunction:

    - subtaskIndex about the concurrency of each NettySink operator.
    - Concurrency of the NettySink operator.

  - RegisterServerHandler

    RegisterServerHandler interacts with the component of RegisterServer and defines following APIs:

    - **start();**: Starts the RegisterServerHandler and establishes a contact with the third-party RegisterServer.
    - **createTopicNode();**: Creates a topic node.
    - **register();**: Registers information such as the IP address, port number, and concurrency to the topic node.
    - **deleteTopicNode();**: Deletes a topic node.
    - **unregister();**: Deletes registration information.
    - **query();**: Queries registration information.

- **isExist();**: Verifies that a specific piece of information exists.
- **shutdown();**: Disables the RegisterServerHandler and disconnects from the third-party RegisterServer.

NOTE

- RegisterServerHandler API enables ZooKeeper to work as the handler of RegisterServer. You can customize your handler as required. Information is stored in ZooKeeper in the following form:

```
Namespace
|---Topic-1
  |---parallel-1
  |---parallel-2
  |....
  |---parallel-n
|---Topic-2
  |---parallel-1
  |---parallel-2
  |....
  |---parallel-m
|...
```

- Information about NameSpace can be obtained from the following parameters of the **flink-conf.yaml** file:

```
nettyconnector.registerserver.topic.storage: /flink/nettyconnector
```

- The simple authentication and security layer (SASL) authentication between ZookeeperRegisterServerHandler and ZooKeeper is implemented through the Flink framework.
- Ensure that each job has a unique topic. Otherwise, the subscription relationship may be unclear.
- When calling **shutdown()**, ZookeeperRegisterServerHandler deletes the registration information about the current concurrency, and then attempts to delete the topic node. If the topic node is not empty, deletion will be canceled, because not all concurrency has exited.

– NettyServer

NettyServer is the core of the NettySink operator, whose main function is to create a NettyServer and receive connection requests from NettyClient. Use NettyServerHandler to send data received from upstream operators of a same job. The port number and subnet of NettyServer needs to be configured in the **flink-conf.yaml** file.

- Port range

```
nettyconnector.sinkserver.port.range: 28444-28943
```

- Subnet

```
nettyconnector.sinkserver.subnet: 10.162.222.123/24
```

NOTE

The **nettyconnector.sinkserver.subnet** parameter is set to the subnet (service IP address) of the Flink client by default. If the client and TaskManager are not in the same subnet, an error may occur. Therefore, you need to manually set this parameter to the subnet (service IP address) of TaskManager.

– NettyServerHandler

The handler enables the interaction between NettySink and subscribers. After NettySink receives messages, the handler sends these messages out. To ensure data transmission security, this channel is encrypted using SSL. The **nettyconnector.ssl.enabled** configures whether to enable SSL

encryption. The SSL encryption is enabled only when **nettyconnector.ssl.enabled** is set to **true**.

- **NettySource**

  NettySource consists of the following major modules:

  – RichParallelSourceFunction

    NettySource inherits RichParallelSinkFunction and attributes of Source operators. The RichParallelSourceFunction API implements following functions:

    - Starts the NettySink operator.
    - Runs the NettySink operator, receives data from subscribers, and injects the data to jobs.
    - Cancels the running of Source operators.

    Following information can be obtained using the attribute of RichParallelSourceFunction:

    - subtaskIndex about the concurrency of each NettySource operator.
    - Concurrency of the NettySource operator.

    When the NettySource operator enters the running stage, the NettyClient status is monitored. Once abnormality occurs, NettyClient is restarted and reconnected to NettyServer, preventing data confusion.

  – RegisterServerHandler

    RegisterServerHandler of NettySource has similar function as the RegisterServerHandler of NettySink. It obtains the IP address, port number, and information of concurrent operators of each subscribed job obtained in the NettySource operator.

  – NettyClient

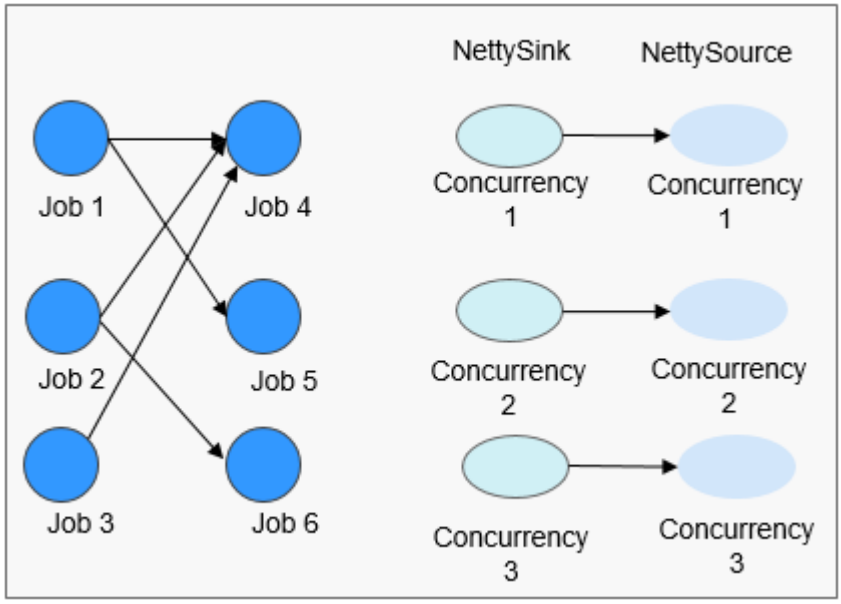
    NettyClient establishes a connection with NettyServer and uses NettyClientHandler to receive data. Each NettySource operator must have a unique name (specified by the user). NettyServer determines whether each client comes from different NettySources based on unique names. When a connection is established between NettyClient and NettyServer, NettyClient is registered with NettyServer and the NettySource name of NettyClient is transferred to NettyServer.

  – NettyClientHandler

    The NettyClientHandler enables the interaction with publishers and other operators of the job. When messages are received, NettyClientHandler transfers these messages to the job. To ensure secure data transmission, SSL encryption is enabled for the communication with NettySink. The SSL encryption is enabled only when SSL is enabled and **nettyconnector.ssl.enabled** is set to **true**.

The relationship between the jobs may be many-to-many. The concurrency between each NettySink and NettySource operator is one-to-many, as shown in **Figure 5-38**.

Figure 5-38 Relationship diagram



5.8.4.3 Stream SQL Join

Enhanced Open Source Feature: Stream SQL Join

Flink's Table API&SQL is an integrated query API for Scala and Java that allows the composition of queries from relational operators such as selection, filter, and join in an intuitive way. For details about Table API & SQL, visit the official website at **https://ci.apache.org/projects/flink/flink-docs-release-1.17/dev/table/index.html**.

Introduction to Stream SQL Join

SQL Join is used to query data based on the relationship between columns in two or more tables. Flink Stream SQL Join allows you to join two streaming tables and query results from them. Queries similar to the following are supported:

```
SELECT o.proctime, o.productId, o.orderId, s.proctime AS shipTime
FROM Orders AS o
JOIN Shipments AS s
ON o.orderId = s.orderId
AND o.proctime BETWEEN s.proctime AND s.proctime + INTERVAL '1' HOUR;
```

Currently, Stream SQL Join needs to be performed within a specified window. The join operation for data within the window requires at least one equi-join predicate and a join condition that bounds the time on both sides. Such a condition can be defined by two appropriate range predicates (<, <=, >=, >), a **BETWEEN** predicate, or a single equality predicate that compares the same type of time attributes (such as processing time or event time) of both input tables.

The following example will join all orders with their corresponding shipments if the order was shipped four hours after the order was received.

```
SELECT *
FROM Orders o, Shipments s
WHERE o.id = s.orderId AND
o.ordertime BETWEEN s.shiptime - INTERVAL '4' HOUR AND s.shiptime
```


NOTE

1. Stream SQL Join supports only inner join.
2. The **ON** clause should include an equal join condition.
3. Time attributes support only the processing time and event time.
4. The window condition supports only the bounded time range, for example, **o.proctime BETWEEN s.proctime - INTERVAL '1' HOUR AND s.proctime + INTERVAL '1' HOUR**. The unbounded range such as **o. proctime > s.proctime** is not supported. The **proctime** attribute of two streams must be included. **o.proctime BETWEEN proctime () AND proctime () + 1** is not supported.

5.8.4.4 Flink CEP in SQL

Flink CEP in SQL

Flink allows users to represent complex event processing (CEP) query results in SQL for pattern matching and evaluate event streams on Flink engines.

SQL Query Syntax

CEP SQL is implemented through the **MATCH_RECOGNIZE** SQL syntax. The **MATCH_RECOGNIZE** clause is supported by Oracle SQL since Oracle Database 12c and is used to indicate event pattern matching in SQL. Apache Calcite also supports the **MATCH_RECOGNIZE** clause.

Flink uses Calcite to analyze SQL query results. Therefore, this operation complies with the Apache Calcite syntax.

```
MATCH_RECOGNIZE (
    [ PARTITION BY expression [, expression ]* ]
    [ ORDER BY orderItem [, orderItem ]* ]
    [ MEASURES measureColumn [, measureColumn ]* ]
    [ ONE ROW PER MATCH | ALL ROWS PER MATCH ]
    [ AFTER MATCH
        ( SKIP TO NEXT ROW
        | SKIP PAST LAST ROW
        | SKIP TO FIRST variable
        | SKIP TO LAST variable
        | SKIP TO variable )
    ]
    PATTERN ( pattern )
    [ WITHIN intervalLiteral ]
    [ SUBSET subsetItem [, subsetItem ]* ]
    DEFINE variable AS condition [, variable AS condition ]*
    )
```

The syntax elements of the **MATCH_RECOGNIZE** clause are defined as follows:

(Optional) **-PARTITION BY**: defines partition columns. This clause is optional. If this parameter is not defined, the parallelism 1 is used.

(Optional) **-ORDER BY**: defines the sequence of events in a data flow. The **ORDER BY** clause is optional. If it is ignored, non-deterministic sorting is used. Since the order of events is important in pattern matching, this clause should be specified in most cases.

(Optional) **-MEASURES**: specifies the attribute value of the successfully matched event.

(Optional) **-ONE ROW PER MATCH | ALL ROWS PER MATCH**: defines how to output the result. **ONE ROW PER MATCH** indicates that only one row is output for each matching. **ALL ROWS PER MATCH** indicates that one row is output for each matching event.

(Optional) **-AFTER MATCH**: specifies the start position for processing after the next pattern is successfully matched.

-PATTERN: defines the matching pattern as a regular expression. The following operators can be used in the **PATTERN** clause: join operators, quantifier operators (*, +, ?, {n}, {n,}, {n,m}, and {,m}), branch operators (vertical bar |), and differential operators ('{- -}').

(Optional) **-WITHIN**: outputs a pattern clause match only when the match occurs within the specified time.

(Optional) **-SUBSET**: combines one or more associated variables defined in the **DEFINE** clause.

-DEFINE: specifies the Boolean condition, which defines the variables used in the **PATTERN** clause.

In addition, the **MATCH_RECOGNIZE** clause supports the following functions:

-MATCH_NUMBER(): Used in the **MEASURES** clause to allocate the same number to each row that is successfully matched.

-CLASSIFIER(): Used in the **MEASURES** clause to indicate the mapping between matched rows and variables.

-FIRST() and **LAST()**: Used in the **MEASURES** clause to return the value of the expression evaluated in the first or last row of the row set mapped to the schema variable.

-NEXT() and **PREV()**: Used in the **DEFINE** clause to evaluate an expression using the previous or next row in a partition.

-RUNNING and **FINAL** keywords: Used to determine the semantics required for aggregation. **RUNNING** can be used in the **MEASURES** and **DEFINE** clauses, whereas **FINAL** can be used only in the **MEASURES** clause.

- Aggregate functions (**COUNT**, **SUM**, **AVG**, **MAX**, **MIN**): Used in the **MEASURES** and **DEFINE** clauses.

Query Example

The following query finds the V-shaped pattern in the stock price data flow.

```
SELECT *
   FROM MyTable
   MATCH_RECOGNIZE (
     ORDER BY rowtime
     MEASURES
       STRT.name as s_name,
       LAST(DOWN.name) as down_name,
       LAST(UP.name) as up_name
     ONE ROW PER MATCH
     PATTERN (STRT DOWN+ UP+)
     DEFINE
       DOWN AS DOWN.v < PREV(DOWN.v),
       UP AS UP.v > PREV(UP.v)
   )
```


In the following query, the aggregate function **AVG** is used in the **MEASURES** clause of **SUBSET E** consisting of variables related to A and C.

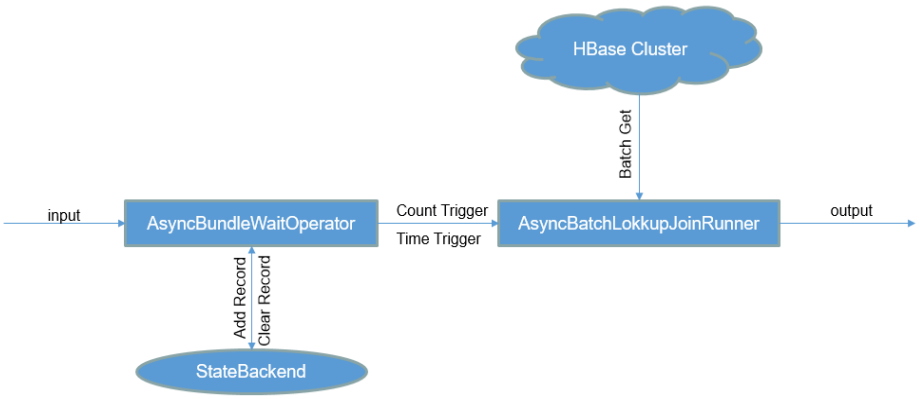
```
SELECT *
   FROM Ticker
   MATCH_RECOGNIZE (
     MEASURES
       AVG(E.price) AS avgPrice
     ONE ROW PER MATCH
     AFTER MATCH SKIP PAST LAST ROW
     PATTERN (A B+ C)
     SUBSET E = (A,C)
     DEFINE
       A AS A.price < 30,
       B AS B.price < 20,
       C AS C.price < 30
   )
```

5.8.4.5 Batch Read of HBase Connector Dimension Tables

HBase Connector supports Flink SQL dimension table query. However, in heavy-traffic service scenarios, each piece of data accesses the HBase cluster in real time. Excessive remote procedure calls (RPCs) affect job performance. HBase dimension tables support batch read, improving dimension table query performance.

Handling process: The AsyncBundleWaitOperator operator caches the received data to the state backend to prevent data loss. The count and time triggers are used to control when data is sent to AsyncBatchLookupJoinRunner. AsyncBatchLookupJoinRunner receives the data construction List<Get>, obtains data in batches from the HBase cluster, and sends the result set returned by HBase to downstream operators. Batch reads can reduce RPCs and improve performance.

For details, visit the Flink official website at **https://nightlies.apache.org/flink/flink-docs-master/docs/dev/table/sql/queries/joins/#lookup-join**.



Example SQL of enabling the function of reading HBase Connector dimension tables in batches:

```
CREATE TABLE Customers (
id INT,
name STRING,
country STRING,
zip STRING ) WITH (
'connector' = 'hbase-2.2',
...
'lookup.batch' = 'true'
);
```


Table 5-6 Parameters

Parameter	Description	Default Value
lookup.batch	Whether to enable batch lookup	false
lookup.batch.interval	The batch interval	1s
lookup.batch.size	The batch size	100

NOTICE

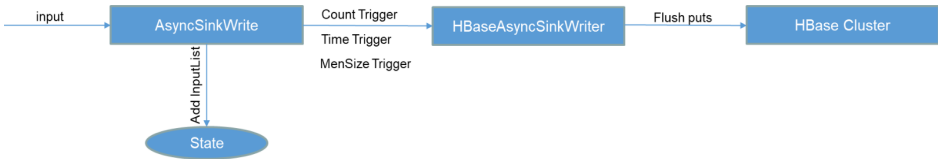
- HBase dimension tables support only stream jobs.
- To enable this feature, set **table.exec.batch-lookup.enabled** to **true** by configuring this parameter in *Client installation path*/**Flink/flink/conf/flink-conf.yaml** or running **-D** (dynamic parameter command).
- This feature is available only in HBase 2.2.

5.8.4.6 Asynchronous Write of HBase Connector Sink Tables

Flink provides asynchronous write.

Handling process: The AsyncSinkWrite operator caches data in the memory and stores the data that is not written to the sink to the state backend and checkpoints during checkpointing to prevent data loss. The count, time, and cache size triggers are used to control when data is written to the sink. HBaseAsyncSinkWrite constructs data as Put or Delete operations and calls HBase's Flush API to send the data to the HBase cluster.

For details, visit the Flink official website at **https://cwiki.apache.org/confluence/display/FLINK/FLIP-171%3A+Async+Sink**.



Example SQL for enabling asynchronous write for HBase Connector sink tables:

```
CREATE TABLE Customers (
id INT,
name STRING,
country STRING,
zip STRING ) WITH (
'connector' = 'hbase-2.2',
...
'sink.async' = 'true'
);
```


Table 5-7 Parameters

Parameter	Description	Default Value
sink.async	Whether to enable asynchronous write	false
sink.batch.max-size	The maximum number of elements that can be transferred to the downstream for writing in a batch	500
sink.requests.max-buffered	The maximum number of records buffered before backpressure	10000
sink.requests.max-inflight	The maximum number of unfinished requests. When the value of this parameter is reached, the operator does not accept new data.	50
sink.flush-buffer.size	The size of the flush buffer, in bytes	4MB
sink.flush-buffer.timeout	The timeout interval of the flush buffer, in milliseconds. After timeout, data is flushed to the connector.	5000
sink.requests.max-retries	The number of retries upon a flush failure	0

NOTE

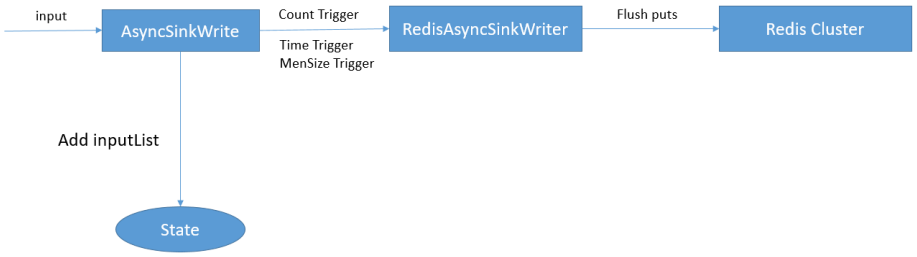
This feature is available only in HBase 2.2.

5.8.4.7 Asynchronous Write of Redis Connector Sink Tables

Flink provides asynchronous write of Redis Connector sink tables.

Handling process: The AsyncSinkWrite operator caches data in the memory and stores the data that is not written to the sink to the state backend and checkpoints during checkpointing to prevent data loss. The Count, Time, and Buffer Size triggers are used to control when data is written to the sink. RedisAsyncSinkWriter constructs data as Put or Delete operations and calls Redis' Flush API to send the data to the Redis cluster.

Figure 5-39 Flink's asynchronous write



Example SQL for enabling asynchronous write for Redis Connector sink tables:

```
CREATE TABLE Customers (
id INT,
name STRING,
country STRING,
zip STRING ) WITH (
'connector' = 'redis',
...
'sink.async' = 'true'
);
```

Table 5-8 Parameters

Parameter	Description	Default Value
sink.async	Whether to enable asynchronous write	false
sink.batch.max-size	The maximum number of elements that can be transferred to the downstream for writing in a batch	500
sink.requests.max-buffered	The maximum number of records buffered before backpressure	10000
sink.requests.max-inflight	The maximum number of unfinished requests. When the value of this parameter is reached, the operator does not accept new data.	50
sink.flush-buffer.size	The size of the flush buffer, in bytes	4MB
sink.flush-buffer.timeout	The timeout interval of the flush buffer, in milliseconds. After timeout, data is flushed to the connector.	5000

5.8.4.8 Flink SQL Enhancement

The following lists the newly added features for Flink SQL enhancement. For details, see "Enhancements to Flink SQL" in *MapReduce Service (MRS) 3.3.1-LTS User Guide (for Huawei Cloud Stack 8.3.1) in the MapReduce Service (MRS) 3.3.1-LTS Usage Guide (for Huawei Cloud Stack 8.3.1)*.

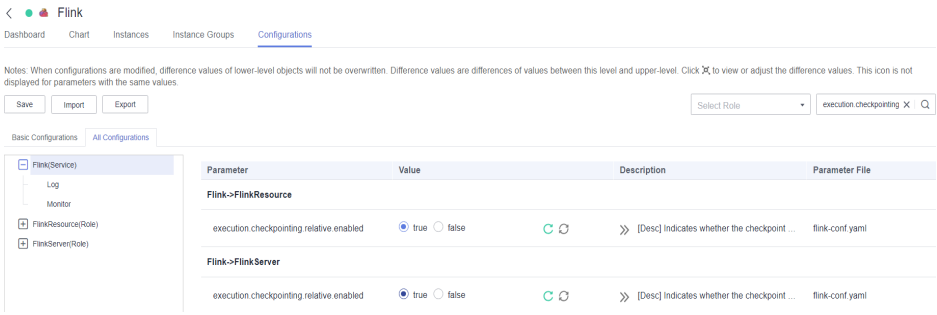
- The DISTRIBUTEBY feature is added to Flink SQL to partition data based on specified fields. A single or multiple fields are supported, solving the problem where only data needs to be partitioned.
- Window functions are added to Flink SQL to support late data processing. Currently, the TUMBLE, HOP, OVER, and CUMULATE window functions support late data. When a window receives late data, the start time and end time of the window can be output by adding **window.start.field** and **window.end.field** to Hint. The fields must be of the timestamp type.
- The function of exiting the Flink SQL OVER window upon data expiration is added. When the existing data expires and no new data arrives, OVER aggregation results are updated and the latest calculation results are sent to the downstream operator. You can use this function by configuring the **over.window.interval** parameters.

5.8.4.9 Relative Directory for Flink Job Checkpoint

The absolute paths of **FileStateHandle** and **ByteStreamStateHandle** are stored in the checkpoint metadata file **_metadata** of a Flink job. As a result, the checkpoint directory becomes unavailable after migration. You can set the **execution.checkpointing.relative.enabled** parameter to set the file path in **_metadata** to a relative path to support checkpoint migration. This function is disabled by default. You can enable it in either of the following ways:

- Enabling the function on FusionInsight Manager
  a. Log in to FusionInsight Manager.
  b. Choose **Cluster** > **Service** > **Flink**, and click **Configuration** and then **All Configurations**. Search for **execution.checkpointing.relative.enabled**, and set all its values to **true**.

  Figure 5-40 Setting the values

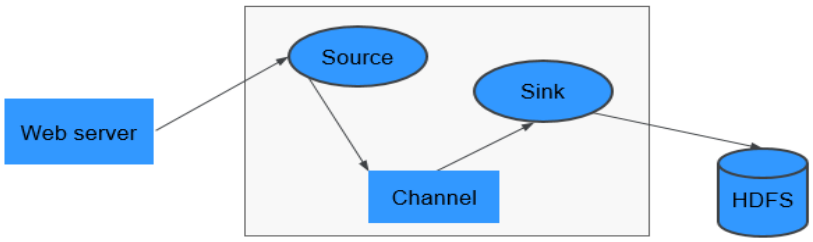
  c. Choose **Dashboard** > **More** > **Restart Service**. Enter the password, and restart the Flink service as prompted.
- Enable the function by adding dynamic parameters when you submit a job.
  If you submit a job in yarn-cluster mode, use the following setting:
  flink run -m yarn-cluster -yD execution.checkpointing.relative.enabled=true

5.9 Apache Flume

5.9.1 Flume Basic Principles

Flume is a distributed, reliable, and HA system that supports massive log collection, aggregation, and transmission. Flume supports customization of various data senders in the log system for data collection. In addition, Flume can roughly process data and write data to various data receivers (customizable). A Flume-NG is a branch of Flume. It is simple, small, and easy to deploy. The following figure shows the basic architecture of the Flume-NG.

Figure 5-41 Flume-NG architecture



A Flume-NG consists of agents. Each agent consists of three components (source, channel, and sink). A source is used for receiving data. A channel is used for transmitting data. A sink is used for sending data to the next end.

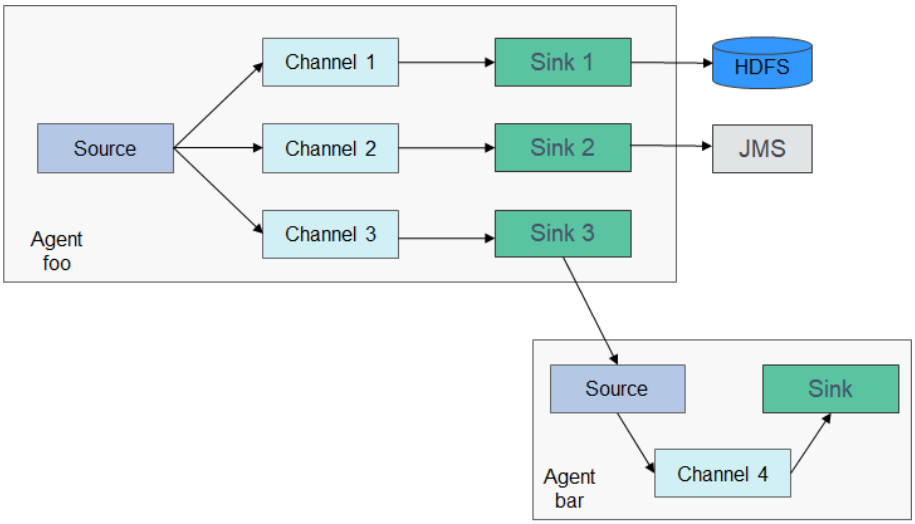
Table 5-9 Module description

Module	Description
Source	A source receives data or generates data by using a special mechanism, and places the data in batches in one or more channels. The source can work in data-driven or polling mode. Typical source types are as follows: • Sources that are integrated with the system, such as Syslog and Netcat • Sources that automatically generate events, such as Exec and SEQ • IPC sources that are used for communication between agents, such as Avro A source must be associated with at least one channel.

Module	Description
Channel	A channel is used to buffer data between a source and a sink. The channel caches data from the source and deletes that data after the sink sends the data to the next-hop channel or final destination. Different channels provide different persistence levels. • Memory channel: non-persistency • File channel: Write-Ahead Logging (WAL)-based persistence • JDBC channel: persistency implemented based on the embedded database The channel supports the transaction feature to ensure simple sequential operations. A channel can work with sources and sinks of any quantity.
Sink	A sink sends data to the next-hop channel or final destination. Once completed, the transmitted data is removed from the channel. Typical sink types are as follows: • Sinks that send storage data to the final destination, such as HDFS and HBase • Sinks that are consumed automatically, such as Null Sink • IPC sinks used for communication between Agents, such as Avro A sink must be associated with a specific channel.

As shown in **Figure 5-42**, a Flume client can have multiple sources, channels, and sinks.

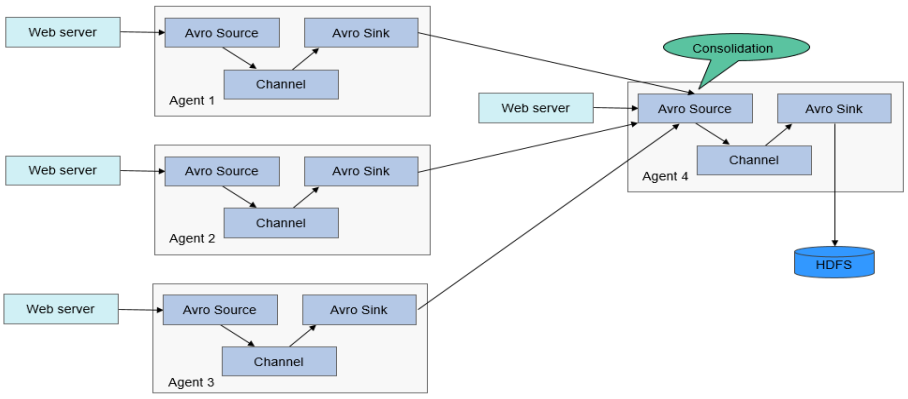
Figure 5-42 Flume structure



The reliability of Flume depends on transaction switchovers between agents. If the next agent breaks down, the channel stores data persistently and transmits data

until the agent recovers. The availability of Flume depends on the built-in load balancing and failover mechanisms. Both the channel and agent can be configured with multiple entities between which they can use load balancing policies. Each agent is a Java Virtual Machine (JVM) process. A server can have multiple agents. Collection nodes (for example, Agents 1, 2, 3) process logs. Aggregation nodes (for example, Agent 4) write the logs into HDFS. The agent of each collection node can select multiple aggregation nodes for load balancing.

Figure 5-43 Flume cascading

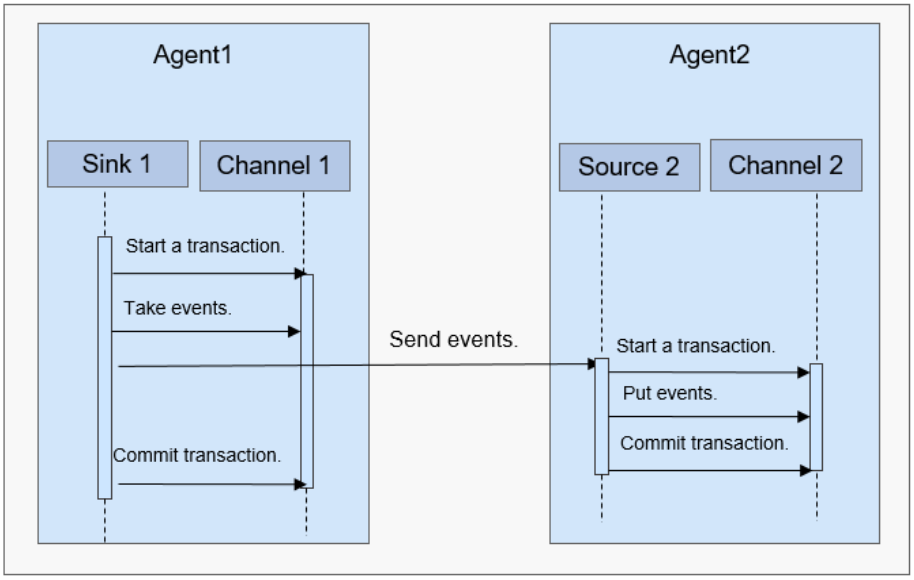


Principle

Reliability between agents

Figure 5-44 shows the data exchange between agents.

Figure 5-44 Data transmission process



1. Flume ensures reliable data transmission based on transactions. When data flows from one agent to another agent, the two transactions take effect. The sink of Agent 1 (agent that sends a message) needs to obtain a message

from a channel and sends the message to Agent 2 (agent that receives the message). If Agent 2 receives and successfully processes the message, Agent 1 will submit a transaction, indicating a successful and reliable data transmission.

2. When Agent 2 receives the message sent by Agent 1 and starts a new transaction, after the data is processed successfully (written to a channel), Agent 2 submits the transaction and sends a success response to Agent 1.
3. Before a commit operation, if the data transmission fails, the last transcription starts and retransmits the data that fails to be transmitted last time. The commit operation has written the transaction into a disk. Therefore, the last transaction can continue after the process fails and restores.

5.9.2 Relationship Between Flume and Other Components

Relationship Between Flume and HDFS

If HDFS is configured as the Flume sink, HDFS functions as the final data storage system of Flume. Flume installs, configures, and writes all transmitted data into HDFS.

Relationship Between Flume and HBase

If HBase is configured as the Flume sink, HBase functions as the final data storage system of Flume. Flume writes all transmitted data into HBase based on configurations.

5.9.3 Flume Enhanced Open Source Features

Flume Enhanced Open Source Features

- Improving transmission speed: Multiple lines instead of only one line of data can be specified as an event. This improves the efficiency of code execution and reduces the times of disk writes.
- Transferring ultra-large binary files: According to the current memory usage, Flume automatically adjusts the memory used for transferring ultra-large binary files to prevent out-of-memory.
- Supporting the customization of preparations before and after transmission: Flume supports customized scripts to be run before or after transmission for making preparations.
- Managing client alarms: Flume receives Flume client alarms through MonitorServer and reports the alarms to the alarm management center on MRS Manager.

5.10 FTP-Server

5.10.1 FTP-Server Basic Principles

Overview

FTP-Server is a pure Java File Transfer Protocol (FTP) service based on the existing open FTP protocol. FTP-Server supports FTP and FTP over SSL (FTPS). Each FTP-Server service supports port and passive data transmission modes. You can perform operations, such as uploading or downloading files, viewing, creating, or deleting directories, and modifying file access permissions, on HDFS through an FTP client.

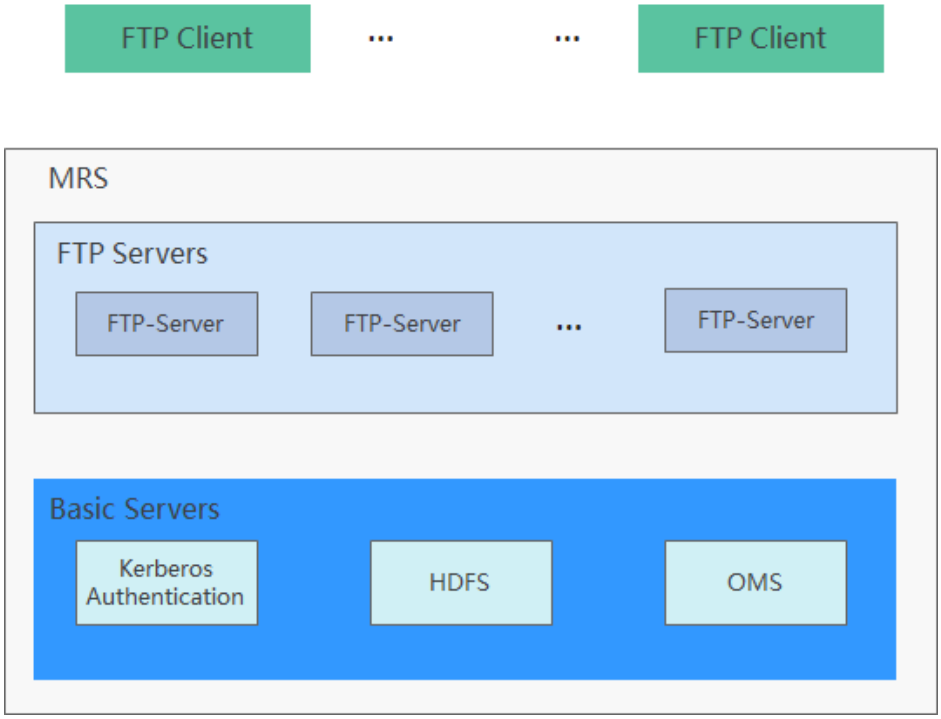
- Supports FTPS. FTPS-based data transmission is encrypted to ensure security. FTP has security risks. It is recommended that FTPS be used.
- Supports port and passive data transmission modes.
- Performs user authentication by using the Kerberos authentication service provided by a cluster.

FTP-Server Architecture

The FTP-Server service consists of multiple FTP-Server or FTPS-Server processes, as shown in **Figure 5-45**.

The FTP-Server service can be deployed on multiple nodes. Each node has only one FTP-Server instance, and each instance has only one FTP-Server process.

Figure 5-45 FTP-Server structure



FTP client

The FTP client is used to access the FTP server to upload and download data. The FTP client is integrated into service applications.

FTP server

The FTP server provides standard FTP APIs externally for FTP clients to access the HDFS system. The FTP server provides most of the FTP commands.

The basic MRS services implement underlying services of FTP servers. That is, the Kerberos security authentication service implements user management, the HDFS service implements data storage, and the OMS service implements service configuration.

Basic servers

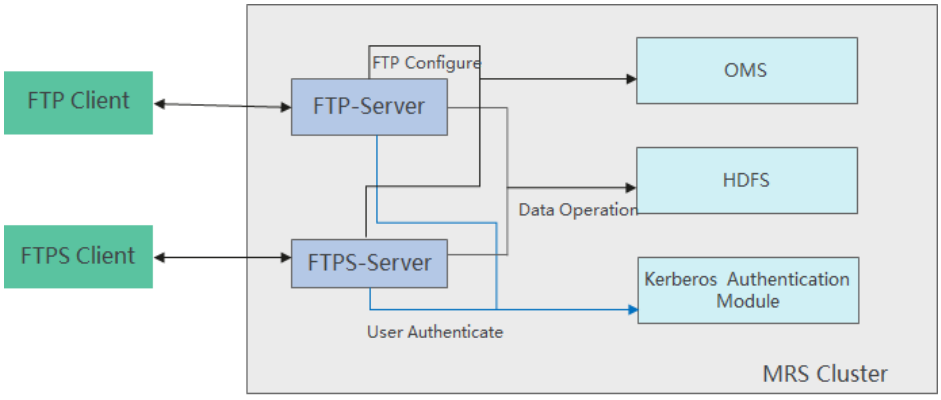
The FTP server provides the following basic services:

- Kerberos security service: supports FTP user management and user login.
- HDFS: implements data storage.
- OMS: configures FTP service parameters and enables or disables FTP services.

Principle

Figure 5-46 shows the FTP-Server data access process.

Figure 5-46 FTP-Server data access process



1. An FTP client connects to the FTP server using the FTP service IP address and port number.
2. The FTP server uses the information to perform user authentication on the Kerberos module.
3. After the authentication succeeds, the FTP server accesses HDFS and returns the file information to the client.
4. The FTP client uses the standard FTP to upload and download files and manage HDFS file directories.

Security

FTP communication is not encrypted, so that the content, username, password, and transmission data are easily stolen. Therefore, FTPS is recommended to be used in untrusted networks. MRS provides FTP-Server to support basic enterprise and financial applications. FTPS allows data to be encrypted during transmission, effectively preventing information leakage. When the client uses FTPS, only the implicit **FTP over TLS** encryption mode is supported.

The FTP-Server process of FTP is disabled by default. The administrator can enable it on the FTP service configuration window. A connection can be created (using the business IP address) only after the service is restarted.

Each node supports 16 FTP/FTPS (user or client) connections by default. To satisfy performance requirements, FTPS is recommended to be used with the command channel encrypted but the data channel not encrypted.

5.10.2 Relationship with Components

Relationship Between FTP-Server and HDFS

HDFS is the storage file system of FTP-Server. All the data uploaded by users is stored on related directories on HDFS. Users perform operations on the files in HDFS by using FTP commands.

Relationship Between FTP-Server and Kerberos

Kerberos Authentication Module is the authentication module of FTP-Server. FTP-Client needs to send the username and password to FTP-Server before connecting to FTP-Server. After receiving the username and password, FTP-Server uses the Kerberos service to check whether the password is correct and whether the user has the rights to access FTP-Server.

5.10.3 FTP-Server Enhanced Open Source Features

Enhanced Open Source Feature: Kerberos Authentication

Apache FTP Server authentication records usernames and passwords in files or databases. In a distributed system, this storage mode has certain defects. The file storage mode is not applicable for distributed systems, while the database storage mode is quite different from user management in HDFS. Therefore, MRS uses the Kerberos service in the cluster for authentication, seamlessly integrating user management, cluster user management, and HDFS user management.

Enhanced Open Source Feature: FTP-based File Transfer to the HDFS File System

As the storage file system of FTP-Server, HDFS stores all data of FTP-Server.

5.11 GraphBase

5.11.1 GraphBase Basic Principles

Overview

With the quick development of network technologies, enterprises in the Internet era are facing massive data. As the number of data sets increases, the query performance of traditional relational databases deteriorates, especially for some special service scenarios. Therefore, a new solution is urgently needed to cope with

this problem. To resolve the complex relationship problem, GraphBase came into being.

In GraphBase, data is stored and queried by graph. A graph contains nodes and relationships. Nodes and relationships can have labels and attributes, and edges can have directions. GraphBase is a distributed graph database. Based on the distributed storage mechanism of HBase, it supports data of tens of billions of nodes and hundreds of billions of relationships, and provides Spark-based data import and Elasticsearch-based index mechanisms. GraphBase is widely used in recommendations, relationship analysis, and financial anti-fraud. GraphBase has the following features:

- Distributed architecture and seamless integration with the Hadoop ecosystem.
- Queries of hundreds of billions of relationships on tens of billions of nodes in just seconds.
- Easy-to-use REST APIs to facilitate data query and analysis.
- Powerful Gremlin graph traversal function to implement complex service logic.
- Offline batch import, real-time stream import, and import performance optimization.

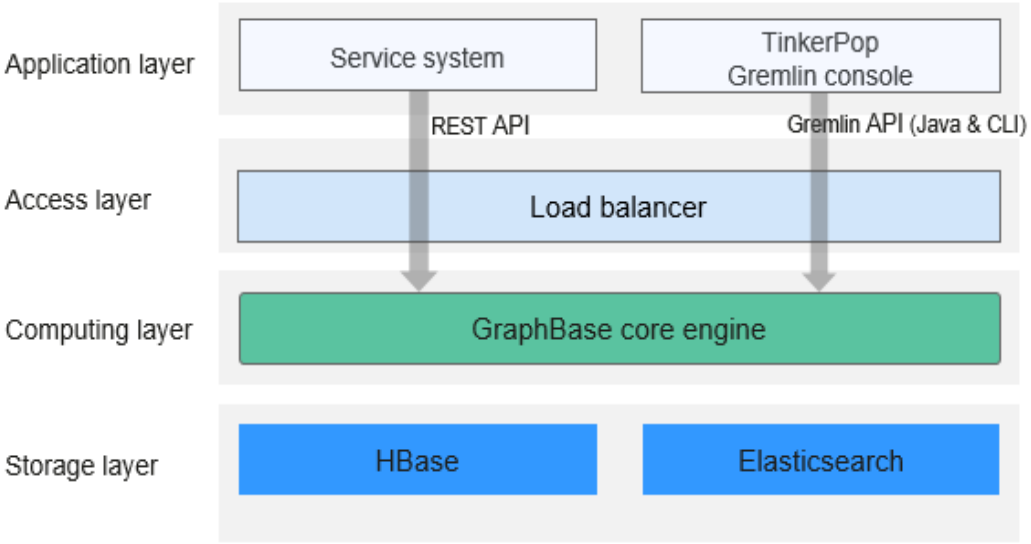
GraphBase architecture

GraphBase contains GraphServer and LoadBalancer.

- GraphServer: includes the GremlinServer and StandardServer services. GremlinServer is used for the graph query using Gremlin, and StandardServer is used for the REST service. When the system is started, the **meta_graph** graph is started first. The **meta_graph** graph is used to store multi-graph metadata and asynchronous tasks. ZooKeeper monitors live instances in services and provides distributed lock services.
- LoadBalancer: balances the load of GraphServer.

Figure 5-47 shows the GraphBase architecture.

Figure 5-47 GraphBase architecture



- Access layer
  - Gremlin API: is an open-source standard language API for graph interactive query based on the Apache TinkerPop Gremlin.
  - REST API: includes APIs for graph query, modification, and management, and graph algorithm enhanced online analysis.
  - Load Balancer: provides load sharing for multi-instance GraphServer.
- Compute layer
  - Provides a core engine of data management and metadata management for GraphBase.
  - Provides API adaptation for backend storage and index.
- Storage layer
  - Distributed KV storage: provides massive graph data storage capabilities.
  - Provides a search engine with secondary index, full-text search, and fuzzy search capabilities.

Typical application scenarios:

- Financial anti-fraud
- Knowledge graph
- Relationship analysis

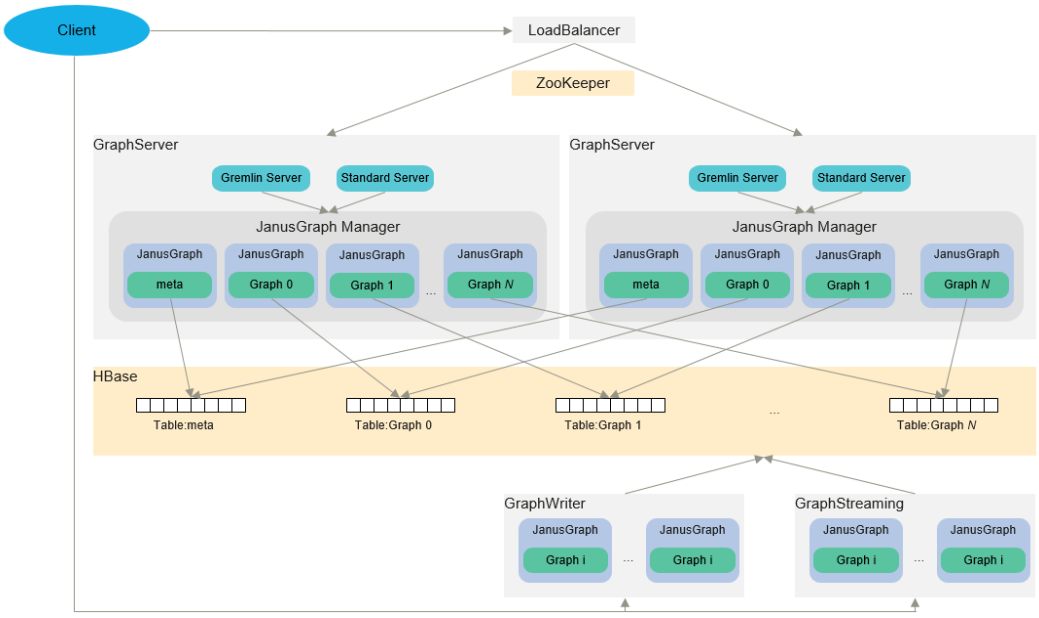
5.11.2 GraphBase Key Features

Key Feature: Multi-Graph

Scenario

- Different service departments can use the same graph database to import different graphs for application development.
- Different applications use different data. Data is not associated, which facilitates service isolation.

Design of Multi-Graph Solution



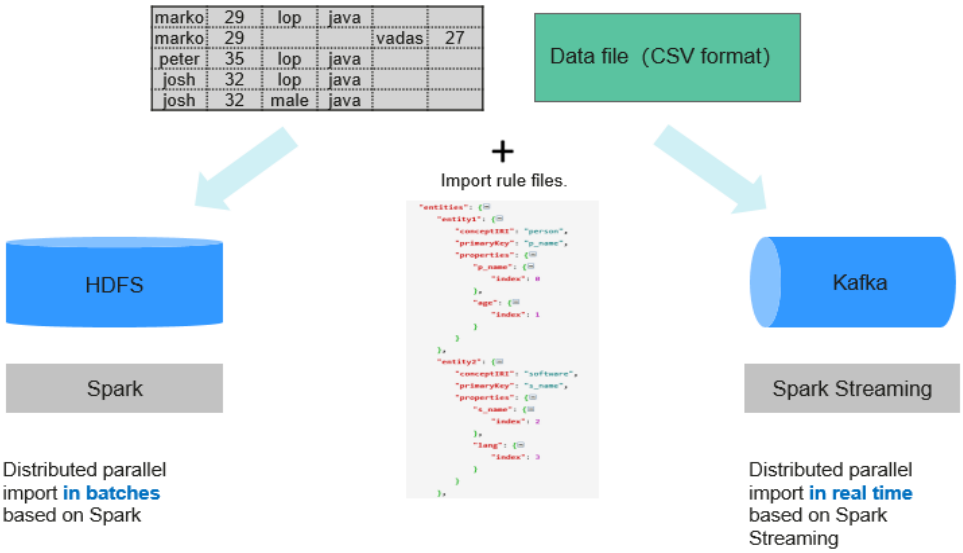
- GraphServer: includes the GremlinServer and StandardServer services. GremlinServer is used for the graph query using Gremlin, and StandardServer is used for the REST service. When the system is started, the **meta_graph** graph is started first. The **meta_graph** graph is used to store multi-graph metadata and asynchronous tasks. ZooKeeper monitors live instances in services and provides distributed lock services.
- LoadBalancer: balances the load of GraphServer.
- GraphWriter: is the module for batch data import.
- GraphStreaming: is used for real-time data import.

Key Feature: Data Import

Batch Import and Real-Time Import

GraphBase supports batch data import and real-time data import. For batch data import, Spark is used to import all historical data stored in HDFS to GraphBase. For real-time data import, Kafka and SparkStreaming are used to import data to GraphBase in real time.

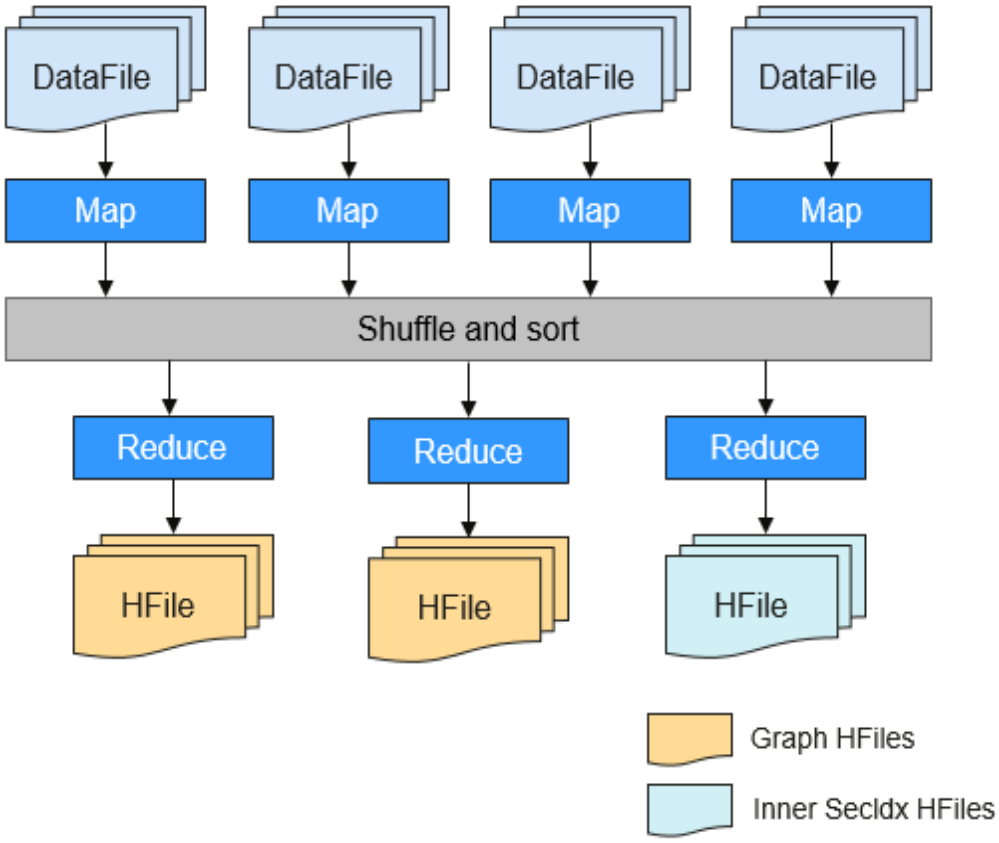
Flexible data mapping rules are provided to map original data to graph models.



BulkLoad Supported in Batch Data Import

The capability of importing data in BulkLoad mode is added to facilitate data import.

During data import, Graph HFiles and Inner secondary index HFiles can be generated in one MapReduce job.

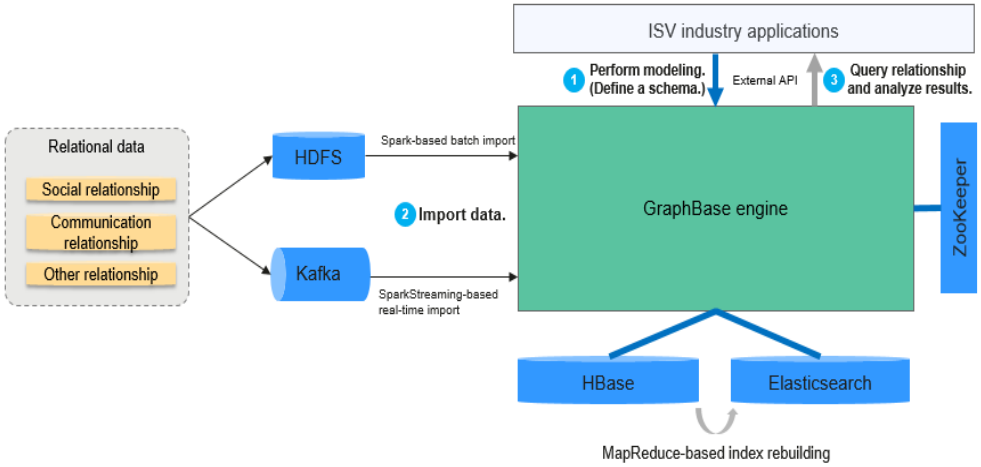


5.11.3 Relationship Between GraphBase and Other Components

Service data and metadata are stored in HBase to support massive data. External index data is stored in Elasticsearch to implement query capabilities such as full-text search and fuzzy match. GraphBase uses Spark to implement batch and real-time data import, uses MapReduce to implement index recreation and batch deletion, and uses ZooKeeper to implement distributed coordination of multiple instances of the compute engine.

Figure 5-48 shows the relationship between GraphBase and other components.

Figure 5-48 Relationship between GraphBase and other components



5.12 Guardian

Guardian Basic Principles

Guardian is a service that provides temporary authentication credentials for services such as HDFS, Hive, Spark, HBase, Loader and HetuEngine to access OBS in decoupled storage and compute scenarios. The Guardian component needs to be installed only when OBS is connected. Typical features of Guardian include:

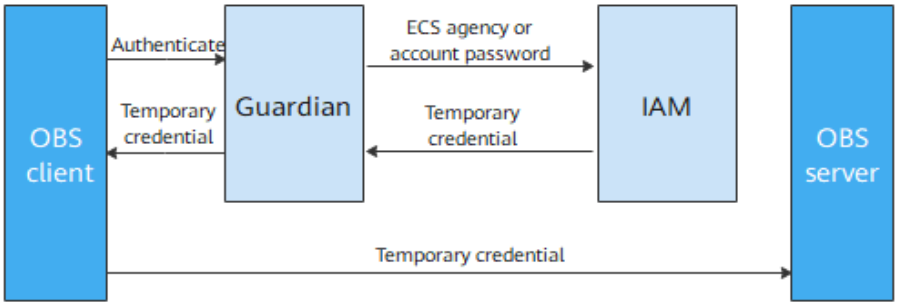
- Provides the capability of obtaining temporary authentication credentials for accessing OBS.
- Provides fine-grained permission control for accessing OBS.
- Provides the unified cache refreshing capability for temporary authentication credentials used to access OBS.

The Guardian server provides functions for the TokenServer role. TokenServer supports multi-instance deployment. Each instance can have the same functions. A single point of failure (SPOF) does not affect service functions. In addition, the Guardian server provides RPC and HTTPS interfaces to obtain temporary authentication credentials for accessing OBS.

Guardian Architecture

Figure 5-49 shows the basic architecture of Guardian.

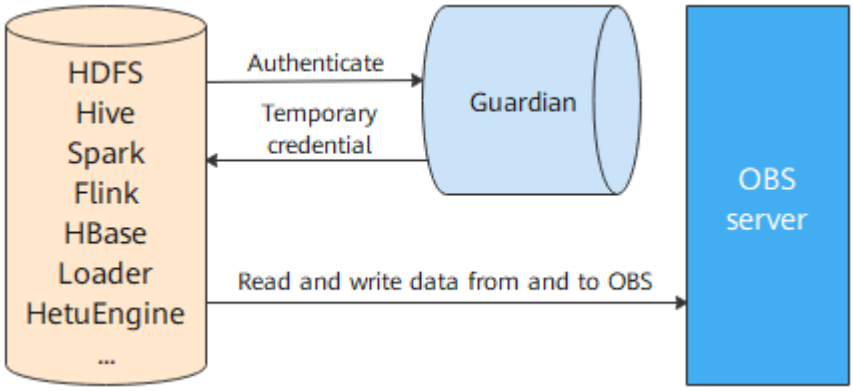
Figure 5-49 Guardian architecture



Relationships Between Guardian and Other Components

Before accessing OBS, HDFS, Hive, Spark, Flink, HBase, Loader, and HetuEngine access Guardian to obtain temporary credentials for the access. Guardian generates a temporary credential with fine-grained authentication content based on the IAM access request of the current login user and returns the credential to the component. The component uses the credential to access OBS. OBS determines whether the current user has the access permission based on the credential.

Figure 5-50 Relationships between Guardian and other components



5.13 Apache HBase

5.13.1 HBase Basic Principles

HBase undertakes data storage. HBase is an open source, column-oriented, distributed storage system that is suitable for storing massive amounts of unstructured or semi-structured data. It features high reliability, high performance, and flexible scalability, and supports real-time data read/write. For more information about HBase, see **https://hbase.apache.org/**.

Typical features of a table stored in HBase are as follows:

- Big table (BigTable): One table contains hundred millions of rows and millions of columns.
- Column-oriented: Column-oriented storage, retrieval, and permission control
- Sparse: Null columns in the table do not occupy any storage space.

MRS HBase supports decoupled storage and compute to allow data to be stored in low-cost cloud storage services (for example, OBS) and allow data to be backed up across AZs. Furthermore, MRS HBase supports secondary indexing to allow indexes to be created for column values so that data can be filtered by column using native HBase APIs.

HBase Architecture

An HBase cluster consists of active and standby HMaster processes and multiple RegionServer processes.

Figure 5-51 HBase architecture

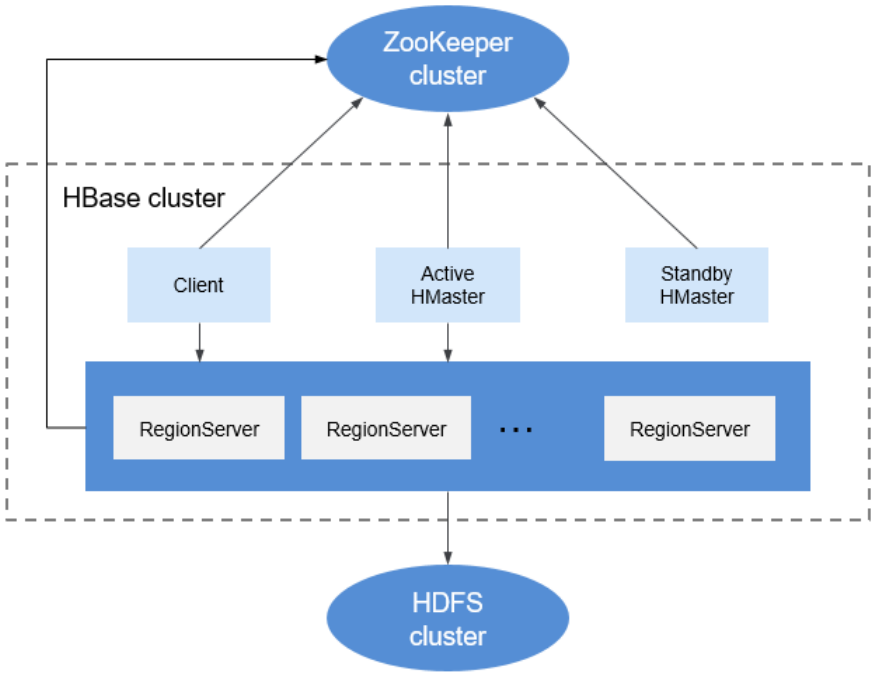


Table 5-10 Module description

Module	Description
Master	Master is also called HMaster. In HA mode, HMaster consists of an active HMaster and a standby HMaster. • Active Master: manages RegionServer in HBase, including the creation, deletion, modification, and query of a table, balances the load of RegionServer, adjusts the distribution of Region, splits Region and distributes Region after it is split, and migrates Region after RegionServer expires. • Standby Master: takes over services when the active HMaster is faulty. The original active HMaster demotes to the standby HMaster after the fault is rectified.
Client	Client communicates with Master for management and with RegionServer for data protection by using the Remote Procedure Call (RPC) mechanism of HBase.
RegionServer	RegionServer provides read and write services of table data as a data processing and computing unit in HBase. RegionServer is deployed with DataNodes of HDFS clusters to store data.
ZooKeeper cluster	ZooKeeper provides distributed coordination services for processes in HBase clusters. Each RegionServer is registered with ZooKeeper so that the active Master can obtain the health status of each RegionServer.

Module	Description
HDFS cluster	HDFS provides highly reliable file storage services for HBase. All HBase data is stored in the HDFS.

HBase Principles

- **HBase Data Model**

  HBase stores data in tables, as shown in **Figure 5-52**. Data in a table is divided into multiple Regions, which are allocated by Master to RegionServers for management.

  Each Region contains data within a RowKey range. An HBase data table contains only one Region at first. As the number of data increases and reaches the upper limit of the Region capacity, the Region is split into two Regions. You can define the RowKey range of a Region when creating a table or define the Region size in the configuration file.

  Figure 5-52 HBase data model

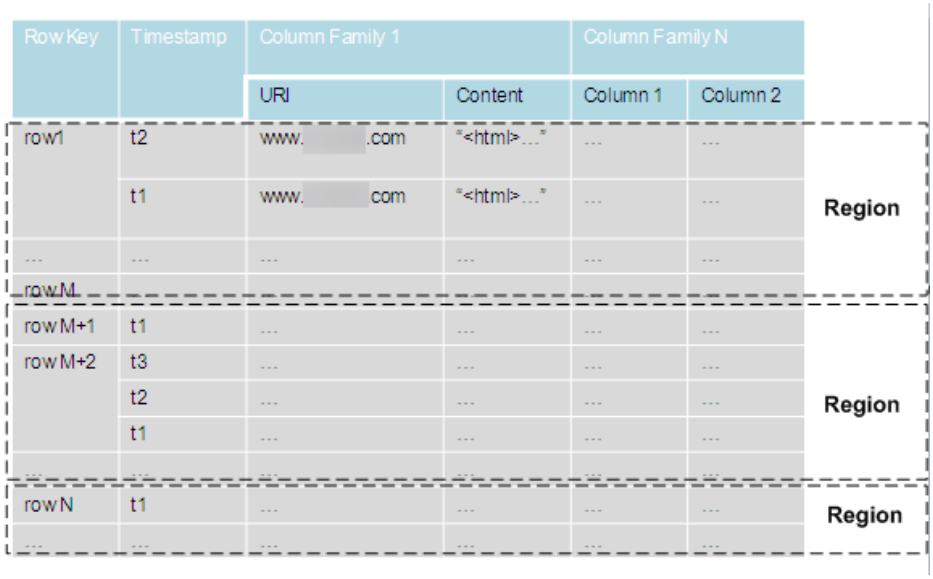


Table 5-11 Concepts

Module	Description
RowKey	Similar to the primary key in a relationship table, which is the unique ID of the data in each row. A RowKey can be a string, integer, or binary string. All records are stored after being sorted by RowKey.
Timestamp	The timestamp of a data operation. Data can be specified with different versions by time stamp. Data of different versions in each cell is stored by time in descending order.

Module	Description
Cell	Minimum storage unit of HBase, consisting of keys and values. A key consists of six fields, namely row, column family, column qualifier, timestamp, type, and MVCC version. Values are the binary data objects.
Column Family	One or multiple horizontal column families form a table. A column family can consist of multiple random columns. A column is a label under a column family, which can be added as required when data is written. The column family supports dynamic expansion so the number and type of columns do not need to be predefined. Columns of a table in HBase are sparsely distributed. The number and type of columns in different rows can be different. Each column family has the independent time to live (TTL). You can lock the row only. Operations on the row in a column family are the same as those on other rows.
Column	Similar to traditional databases, HBase tables also use columns to store data of the same type.

- **RegionServer Data Storage**

  RegionServer manages the regions allocated by HMaster. **Figure 5-53** shows the data storage structure of RegionServer.

  Figure 5-53 RegionServer data storage structure

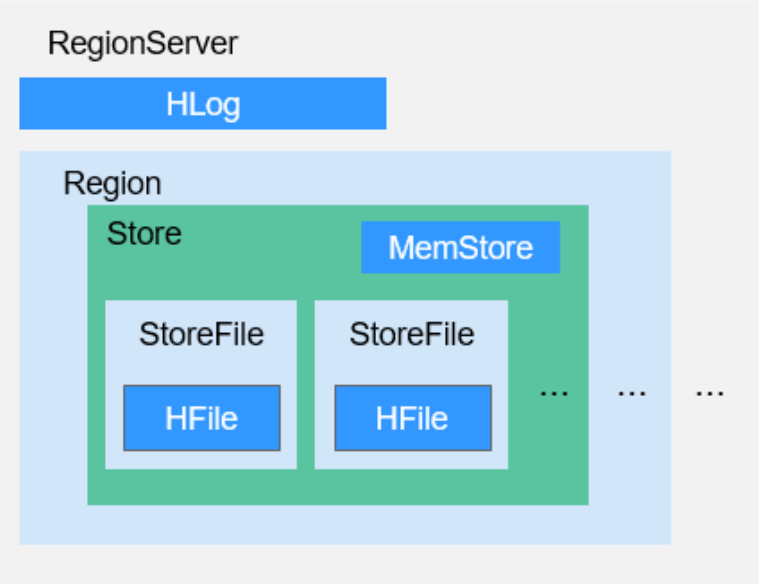


  Table 5-12 lists each component of Region described in **Figure 5-53**.

Table 5-12 Region structure description

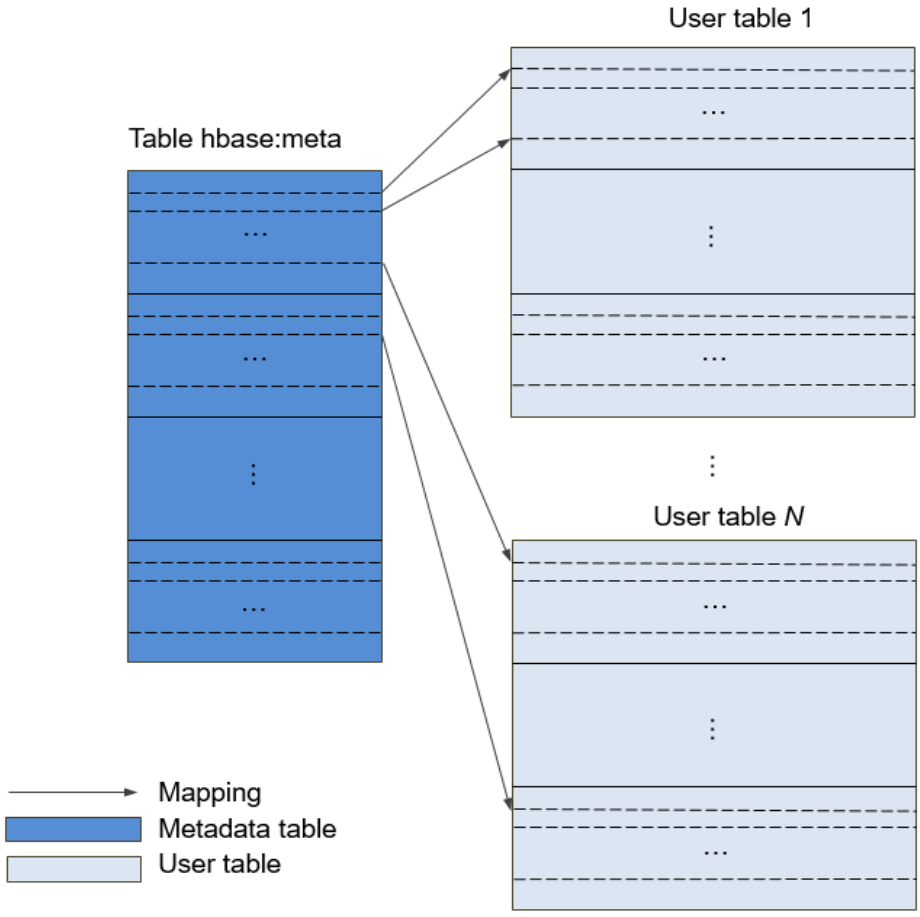
Module	Description
Store	A Region consists of one or multiple Stores. Each Store maps a column family in **Figure 5-52**.
MemStore	A Store contains one MemStore. The MemStore caches data inserted to a Region by the client. When the MemStore capacity reaches the upper limit, RegionServer flushes data in MemStore to the HDFS.
StoreFile	The data flushed to the HDFS is stored as a StoreFile in the HDFS. As more data is inserted, multiple StoreFiles are generated in a Store. When the number of StoreFiles reaches the upper limit, RegionServer merges multiple StoreFiles into a big StoreFile.
HFile	HFile defines the storage format of StoreFiles in a file system. HFile is the underlying implementation of StoreFile.
HLog	HLogs prevent data loss when RegionServer is faulty. Multiple Regions in a RegionServer share the same HLog.

- **Metadata Table**

  The metadata table is a special HBase table, which is used by the client to locate a region. Metadata table includes **hbase:meta** table to record region information of user tables, such as the region location and start and end RowKey.

  Figure 5-54 shows the mapping relationship between metadata tables and user tables.

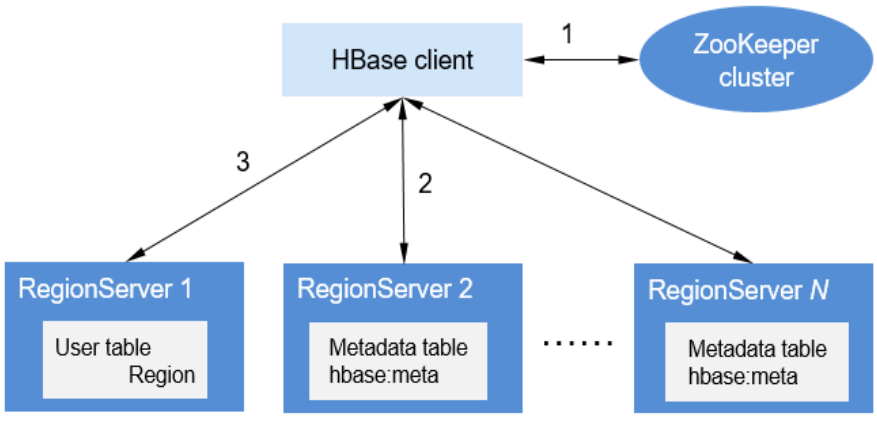
Figure 5-54 Mapping relationships between metadata tables and user tables



- **Data Operation Process**

  Figure 5-55 shows the HBase data operation process.

  Figure 5-55 Data processing



  a. When you add, delete, modify, and query HBase data, the HBase client first connects to ZooKeeper to obtain information about the RegionServer where the **hbase:meta** table is located. If you modify the NameSpace, such as creating and deleting a table, you need to access HMaster to update the meta information.

b. The HBase client connects to the RegionServer where the region of the **hbase:meta** table is located and obtains the RegionServer location where the region of the user table resides.

c. Then the HBase client connects to the RegionServer where the region of the user table is located and issues a data operation command to the RegionServer. The RegionServer executes the command.

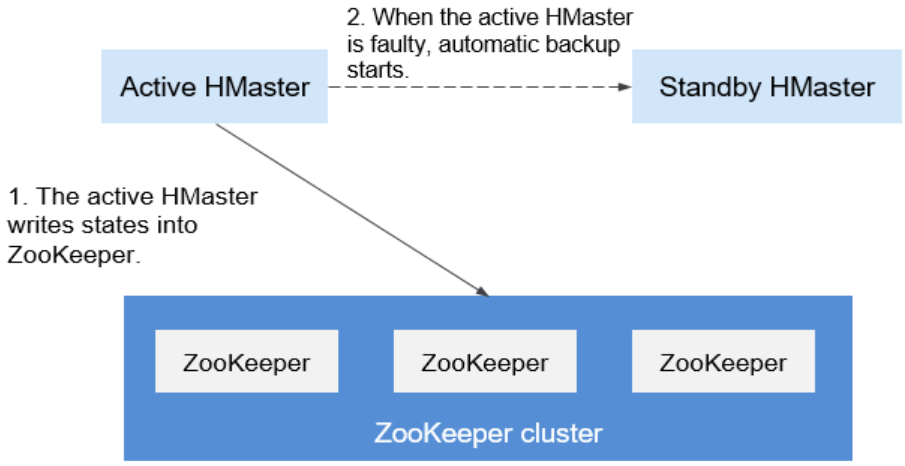
To improve data processing efficiency, the HBase client caches region information of the **hbase:meta** table and user table. When an application initiates a second data operation, the HBase client queries the region information from the memory. If no match is found in the memory, the HBase client performs the preceding operations to obtain region information.

5.13.2 HBase HA Solution

HBase HA

HMaster in HBase allocates Regions. When one RegionServer service is stopped, HMaster migrates the corresponding Region to another RegionServer. The HMaster HA feature is brought in to prevent HBase functions from being affected by the HMaster single point of failure (SPOF).

Figure 5-56 HMaster HA implementation architecture



The HMaster HA architecture is implemented by creating Ephemeral nodes (temporary nodes) in the ZooKeeper cluster.

Upon startup, HMaster nodes try to create a master znode in the ZooKeeper cluster. The HMaster node that creates the master znode first becomes the active HMaster, and the other is the standby HMaster.

It will add watch events to the master node. If the service on the active HMaster is stopped, the active HMaster disconnects from the ZooKeeper cluster. After the session expires, the active HMaster disappears. The standby HMaster detects the disappearance of the active HMaster through watch events and creates a master node to make itself be the active one. Then, the active/standby switchover completes. If the failed node detects existence of the master node after being restarted, it enters the standby state and adds watch events to the master node.

When the client accesses the HBase, it first obtains the HMaster's address based on the master node information on the ZooKeeper and then establishes a connection to the active HMaster.

5.13.3 Relationship with Other Components

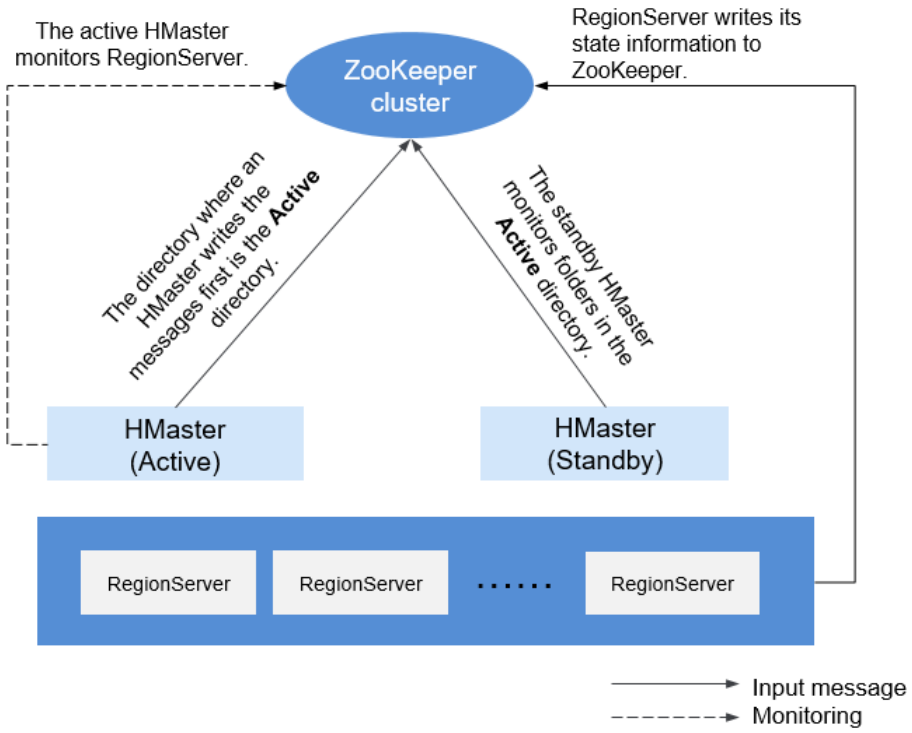
Relationship Between HDFS and HBase

HDFS is the subproject of Apache Hadoop. HBase uses the Hadoop Distributed File System (HDFS) as the file storage system. HBase is located in structured storage layer. The HDFS provides highly reliable support for lower-layer storage of HBase. All the data files of HBase can be stored in the HDFS, except some log files generated by HBase.

Relationship Between ZooKeeper and HBase

Figure 5-57 describes the relationship between ZooKeeper and HBase.

Figure 5-57 Relationship between ZooKeeper and HBase



1. RegionServer registers itself to ZooKeeper in Ephemeral node. ZooKeeper stores the HBase information, including the HBase metadata and HMaster addresses.
2. HMaster detects the health status of each RegionServer using ZooKeeper, and monitors them.
3. HBase can deploy multiple HMasters (like HDFS NameNode). When the active HMatser node is faulty, the standby HMaster node obtains the state information of the entire cluster using ZooKeeper, which means that HBase single point faults can be avoided using ZooKeeper.

5.13.4 HBase Enhanced Open Source Features

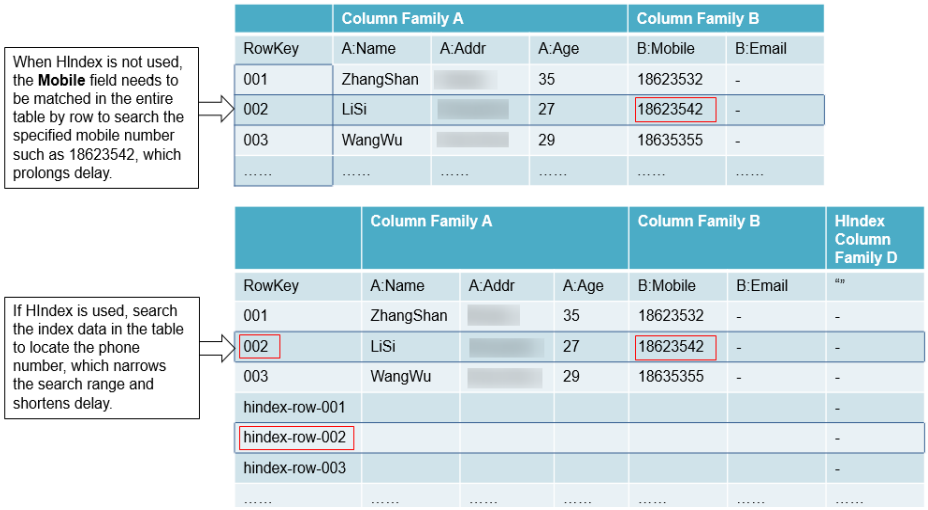
HIndex

HBase is a distributed storage database of the Key-Value type. Data of a table is sorted in the alphabetic order based on row keys. If you query data based on a specified row key or scan data in the scale of a specified row key, HBase can quickly locate the target data, enhancing the efficiency.

However, in most actual scenarios, you need to query the data of which the column value is *XXX*. HBase provides the Filter feature to query data with a specific column value. All data is scanned in the order of row keys, and then the data is matched with the specific column value until the required data is found. The Filter feature scans some unnecessary data to obtain the only required data. Therefore, the Filter feature cannot meet the requirements of frequent queries with high performance standards.

HBase HIndex is designed to address these issues. HBase HIndex enables HBase to query data based on specific column values.

Figure 5-58 HIndex



- Rolling upgrade is not supported for index data.
- Restrictions of combined indexes:
  - All columns involved in combined indexes must be entered or deleted in a single mutation. Otherwise, inconsistency will occur.

    Index: **IDX1=>cf1:[q1->datatype],[q2];cf2:[q2->datatype]**

    Correct write operations:

    ```
    Put put = new Put(Bytes.toBytes("row"));
    put.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q1"), Bytes.toBytes("valueA"));
    put.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q2"), Bytes.toBytes("valueB"));
    put.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q2"), Bytes.toBytes("valueC"));
    table.put(put);
    ```

    Incorrect write operations:

    ```
    Put put1 = new Put(Bytes.toBytes("row"));
    put1.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q1"), Bytes.toBytes("valueA"));
    table.put(put1);
    ```



```
Put put2 = new Put(Bytes.toBytes("row"));
put2.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q2"), Bytes.toBytes("valueB"));
table.put(put2);
Put put3 = new Put(Bytes.toBytes("row"));
put3.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q2"), Bytes.toBytes("valueC"));
table.put(put3);
```

  - The combined conditions-based query is supported only when the combined index column contains filter criteria, or StartRow and StopRow are not specified for some index columns.

    Index: **IDX1=>cf1:[q1->datatype],[q2];cf2:[q1->datatype]**

    Correct query operations:

    ```
    scan 'table', {FILTER=>"SingleColumnValueFilter('cf1','q1',>=,'binary:valueA',true,true) AND
    SingleColumnValueFilter('cf1','q2',>=,'binary:valueB',true,true) AND
    SingleColumnValueFilter('cf2','q1',>=,'binary:valueC',true,true) "}

    scan 'table', {FILTER=>"SingleColumnValueFilter('cf1','q1',=,'binary:valueA',true,true) AND
    SingleColumnValueFilter('cf1','q2',>=,'binary:valueB',true,true)" }

    scan 'table', {FILTER=>"SingleColumnValueFilter('cf1','q1',>=,'binary:valueA',true,true) AND
    SingleColumnValueFilter('cf1','q2',>=,'binary:valueB',true,true) AND
    SingleColumnValueFilter('cf2','q1',>=,'binary:valueC',true,true)",STARTROW=>'row001',STOPROW
    =>'row100'}
    ```

    Incorrect query operations:

    ```
    scan 'table', {FILTER=>"SingleColumnValueFilter('cf1','q1',>=,'binary:valueA',true,true) AND
    SingleColumnValueFilter('cf1','q2',>=,'binary:valueB',true,true) AND
    SingleColumnValueFilter('cf2','q1',>=,'binary:valueC',true,true)  AND
    SingleColumnValueFilter('cf2','q2',>=,'binary:valueD',true,true)"}

    scan 'table', {FILTER=>"SingleColumnValueFilter('cf1','q1',=,'binary:valueA',true,true) AND
    SingleColumnValueFilter('cf2','q1',>=,'binary:valueC',true,true)" }

    scan 'table', {FILTER=>"SingleColumnValueFilter('cf1','q1',=,'binary:valueA',true,true) AND
    SingleColumnValueFilter('cf2','q2',>=,'binary:valueD',true,true)" }

    scan 'table', {FILTER=>"SingleColumnValueFilter('cf1','q1',=,'binary:valueA',true,true) AND
    SingleColumnValueFilter('cf1','q2',>=,'binary:valueB',true,true)" ,STARTROW=>'row001',STOPROW
    =>'row100' }
    ```

- Do not explicitly configure any split policy for tables with index data.
- Other mutation operations, such as **increment** and **append**, are not supported.
- Index of the column with **maxVersions** greater than 1 is not supported.
- The data index column in a row cannot be updated.

  Index 1: **IDX1=>cf1:[q1->datatype],[q2];cf2:[q1->datatype]**

  Index 2: **IDX2=>cf2:[q2->datatype]**

  Correct update operations:

  ```
  Put put1 = new Put(Bytes.toBytes("row"));
  put1.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q1"), Bytes.toBytes("valueA"));
  put1.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q2"), Bytes.toBytes("valueB"));
  put1.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q1"), Bytes.toBytes("valueC"));
  put1.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q2"), Bytes.toBytes("valueD"));
  table.put(put1);

  Put put2 = new Put(Bytes.toBytes("row"));
  put2.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q3"), Bytes.toBytes("valueE"));
  put2.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q3"), Bytes.toBytes("valueF"));
  table.put(put2);
  ```

  Incorrect update operations:

  ```
  Put put1 = new Put(Bytes.toBytes("row"));
  put1.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q1"), Bytes.toBytes("valueA"));
  ```



```
put1.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q2"), Bytes.toBytes("valueB"));
put1.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q1"), Bytes.toBytes("valueC"));
put1.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q2"), Bytes.toBytes("valueD"));
table.put(put1);

Put put2 = new Put(Bytes.toBytes("row"));
put2.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q1"), Bytes.toBytes("valueA_new"));
put2.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("q2"), Bytes.toBytes("valueB_new"));
put2.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q1"), Bytes.toBytes("valueC_new"));
put2.addColumn(Bytes.toBytes("cf2"), Bytes.toBytes("q2"), Bytes.toBytes("valueD_new"));
table.put(put2);
```

- The table to which an index is added cannot contain a value greater than 32 KB.
- If user data is deleted due to the expiration of the column-level TTL, the corresponding index data is not deleted immediately. It will be deleted in the major compaction operation.
- The TTL of the user column family cannot be modified after the index is created.
  - If the TTL of a column family increases after an index is created, delete the index and re-create one. Otherwise, some generated index data will be deleted before user data is deleted.
  - If the TTL value of the column family decreases after an index is created, the index data will be deleted after user data is deleted.
- The index query does not support the reverse operation, and the query results are disordered.
- The index does not support the **clone snapshot** operation.
- The index table must use HIndexWALPlayer to replay logs. WALPlayer cannot be used to replay logs.

```
[xxx@10-10-x-x client]#hbase org.apache.hadoop.hbase.hindex.mapreduce.HIndexWALPlayer
Usage: WALPlayer [options] <WAL inputdir> [<tables> <tableMappings>]
 <WAL inputdir>   directory of WALs to replay.
 <tables>         comma separated list of tables. If no tables specified,
                  all are imported (even hbase:meta if present).
 <tableMappings>  WAL entries can be mapped to a new set of tables by passing
                  <tableMappings>, a comma separated list of target tables.
                  If specified, each table in <tables> must have a mapping.
To generate HFiles to bulk load instead of loading HBase directly, pass:
 -Dwal.bulk.output=/path/for/output
 Only one table can be specified, and no mapping allowed!
To specify a time range, pass:
 -Dwal.start.time=[date|ms]
 -Dwal.end.time=[date|ms]
 The start and the end date of timerange (inclusive). The dates can be
 expressed in milliseconds-since-epoch or yyyy-MM-dd'T'HH:mm:ss.SS format.
 E.g. 1234567890120 or 2009-02-13T23:32:30.12
Other options:
 -Dmapreduce.job.name=jobName
 Use the specified mapreduce job name for the wal player
 -Dwal.input.separator=' '
 Change WAL filename separator (WAL dir names use default ',')
For performance also consider the following options:
  -Dmapreduce.map.speculative=false
  -Dmapreduce.reduce.speculative=false
```

- When the **deleteall** command is executed for the index table, the performance is low.
- The index table does not support HBCK. To use HBCK to repair the index table, delete the index data first.

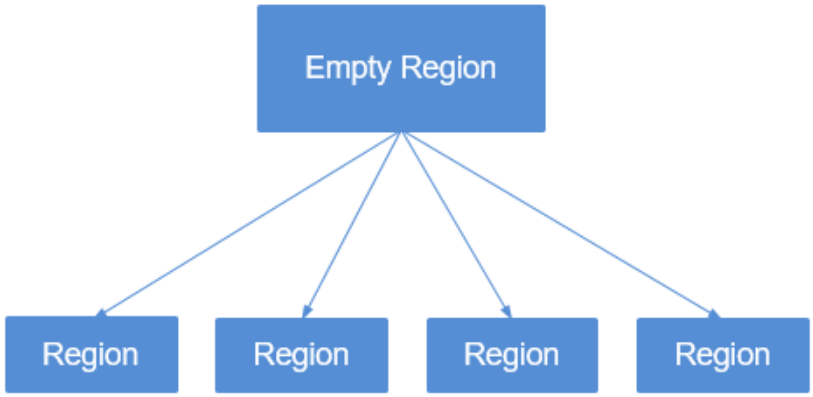
Multi-point Division

When you create tables that are pre-divided by region in HBase, you may not know the data distribution trend so the division by region may be inappropriate. After the system runs for a period, regions need to be divided again to achieve better performance. Only empty regions can be divided.

The region division function delivered with HBase divides regions only when they reach the threshold. This is called "single point division".

To achieve better performance when regions are divided based on user requirements, multi-point division is developed, which is also called "dynamic division". That is, an empty region is pre-divided into multiple regions to prevent performance deterioration caused by insufficient region space.

Figure 5-59 Multi-point division



Connection Limitation

Too many sessions mean that too many queries and MapReduce tasks are running on HBase, which compromises HBase performance and even causes service rejection. You can configure parameters to limit the maximum number of sessions that can be established between the client and the HBase server to achieve HBase overload protection.

Improved Disaster Recovery

The disaster recovery (DR) capabilities between the active and standby clusters can enhance HA of the HBase data. The active cluster provides data services and the standby cluster backs up data. If the active cluster is faulty, the standby cluster takes over data services. Compared with the open source replication function, this function is enhanced as follows:

1. The standby cluster whitelist function is only applicable to pushing data to a specified cluster IP address.
2. In the open source version, replication is synchronized based on WAL, and data backup is implemented by replaying WAL in the standby cluster. For BulkLoad operations, since no WAL is generated, data will not be replicated to the standby cluster. By recording BulkLoad operations on the WAL and

synchronizing them to the standby cluster, the standby cluster can read BulkLoad operation records through WAL and load HFile in the active cluster to the standby cluster to implement data backup.

3. In the open source version, HBase filters ACLs. Therefore, ACL information will not be synchronized to the standby cluster. By adding a filter (**org.apache.hadoop.hbase.replication.SystemTableWALEntryFilterAllowACL**), ACL information can be synchronized to the standby cluster. You can configure **hbase.replication.filter.sytemWALEntryFilter** to enable the filter and implement ACL synchronization.
4. The standby cluster is read-only for HBase clients that are not deployed on the standby cluster. Only the internal administrative user of the nodes in the standby cluster can modify HBase.

HFS

HBase FileStream (HFS) is an independent HBase file storage module. It is used in MRS upper-layer applications by encapsulating HBase and HDFS interfaces to provide these upper-layer applications with functions such as file storage, read, and deletion.

In the Hadoop ecosystem, the HDFS and HBase face tough problems in mass file storage in some scenarios:

- If a large number of small files are stored in HDFS, the NameNode will be under great pressure.
- Some large files cannot be directly stored on HBase due to HBase APIs and internal mechanisms.

HFS is developed for the mixed storage of massive small files and some large files in Hadoop. Simply speaking, massive small files (smaller than 10 MB) and some large files (greater than 10 MB) need to be stored in HBase tables.

For such a scenario, HFS provides unified operation APIs similar to HBase function APIs.

Multiple RegionServers Deployed on the Same Server

Multiple RegionServers can be deployed on one node to improve HBase resource utilization.

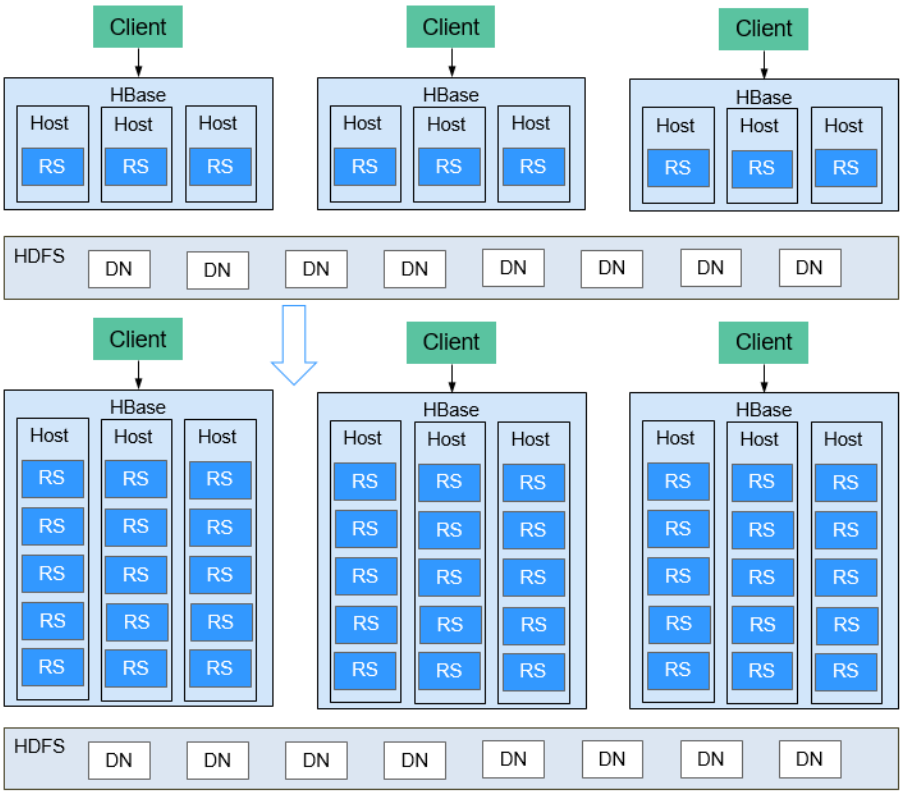
If only one RegionServer is deployed, resource utilization is low due to the following reasons:

1. A RegionServer supports a limited number of regions, and therefore memory and CPU resources cannot be fully used.
2. A single RegionServer occupies up to 20 TB of storage, two replicas require 40 TB, and three replicas require 60 TB. The 96 TB disk space will not be used up.
3. Poor write performance: One RegionServer is deployed on a physical server, and only one HLog exists. Only three disks can be written at the same time.

The HBase resource utilization can be improved when multiple RegionServers are deployed on the same server.

1. A physical server can be configured with a maximum of five RegionServers. The number of RegionServers deployed on each physical server can be configured as required.
2. Resources such as memory, disks, and CPUs can be fully used.
3. A physical server supports a maximum of five HLogs and allows data to be written to 15 disks at the same time, significantly improving write performance.

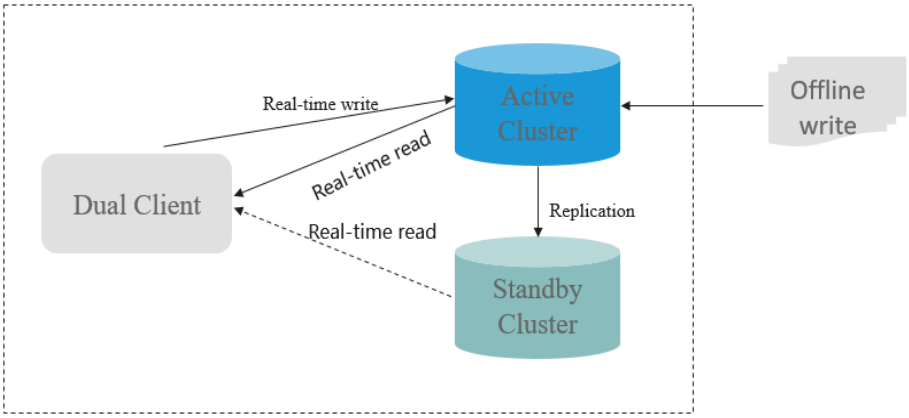
Figure 5-60 Improved HBase resource utilization



HBase Dual-Read

In the HBase storage scenario, it is difficult to ensure 99.9% query stability due to GC, network jitter, and bad sectors of disks. The HBase dual-read feature is added to meet the requirements of low glitches during large-data-volume random read.

The HBase dual-read feature is based on the DR capability of the active and standby clusters. The probability that the two clusters generate glitches at the same time is far less than that of one cluster. The dual-cluster concurrent access mode is used to ensure query stability. When a user initiates a query request, the HBase service of the two clusters is queried at the same time. If the active cluster does not return any result after a period of time (the maximum tolerable glitch time), the data of the cluster with the fastest response can be used. The following figure shows the working principle.



Custom Delimiters Supported on Phoenix CsvBulkLoadTool

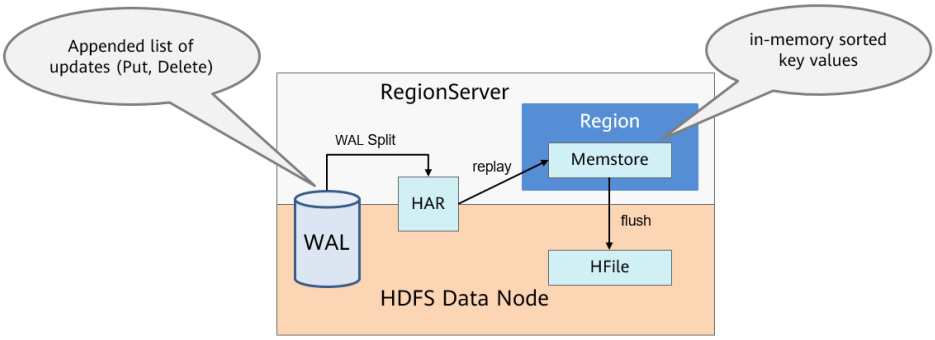
Currently, Phoenix's open source CsvBulkLoadTool supports only a single character as the data delimiter. When a user data file contains any characters, a special string is used as the delimiter. To meet this requirement, custom delimiters are supported so you can use any visible characters within the specified length as delimiters to import data files.

Writing Small Files Generated During WAL File Splitting to the HTTP Archive (HAR) File

When a RegionServer is faulty or restarted, HMaster uses ServerCrashProcedure to restore the services running on the RegionServer. The restoration process involves splitting WAL files. During WAL file splitting, a large number of small files are generated, which may cause HDFS performance bottlenecks. As a result, service restoration takes a long time.

This feature writes small files to the HAR file during WAL file splitting to shorten the RegionServer restoration duration.

For details about HAR, visit **http://hadoop.apache.org/docs/stable/hadoop-archives/HadoopArchives.html**.



Batch TRSP

HBase 2.*x* uses HBase Procedure to rewrite the region assignment logic (AMV2). When each region is opened or closed, a TransitRegionStateProcedure (TRSP) is

associated with it. When services running on a RegionServer need to be restored due to RegionServer faults or restarts, HMaster creates a TRSP for each region to be restored. A large number of TRSPs need to persist data to Proc WAL files and perform an RPC interaction with RegionServer, which may cause HMaster performance bottlenecks. As a result, the service restoration takes a long time.

This feature attaches regions to TRSPs and uses one TRSP to restore all regions of a RegionServer. RegionServer batch opens or closes regions and reports all regions to HMaster at a time.

NOTE

This feature can only restore regions to their original RegionServers. Therefore, the prerequisite for this optimization to take effect is that the faulty or restarted RegionServer has been brought online again when HMaster creates a TRSP. This feature is used to optimize the duration for HBase restart or service fault restoration. If a few RegionServers are faulty, this feature may not take effect because HMaster had created TRSPs before RegionServers were brought online again.

HBase Self-Healing from Hotspotting

HBase is a distributed key-value database. Regions are the smallest units for data management. If table planning and rowkey design are improper, requests are distributed to a few fixed regions, and the service pressure is concentrated on a single node. As a result, the service performance deteriorates or even requests fail.

The MetricController instance is available for the HBase service. The instance is optional. MetricController must be deployed in active/standby mode and share the same node with HMaster. If the MetricController instance is deployed, hotspotting detection is enabled by default and the request traffic of each RegionServer node can be monitored. Through aggregation analysis, the nodes and regions with excessive requests can be identified, helping quickly identify hotspotting. In addition, the self-healing from hotspotting function is provided to transfer workload or perform region splitting. If the self-healing from hotspotting function cannot be used (for issues such as hotspotting on a single rowkey and sequential write hotspotting), the hotspot traffic limiting function is provided instead to minimize the impact on other normal services on this node.

NOTE

To disable automatic recovery for HBase hotspotting, log in to FusionInsight Manager, click **Cluster**, choose **Services** > **HBase**, and click **Configurations**. Search for **hbase.metric.report.provider** and **hbase.rowkey.metric.provider**, leave the two parameters empty, and click **Save**. Click **Instances**, select all MetricController instances, and choose **More** > **Stop Instance**.

5.14 HDFS

5.14.1 HDFS Basic Principles

Hadoop Distributed File System (HDFS) implements reliable and distributed read/write of massive amounts of data. HDFS is applicable to the scenario where data read/write features "write once and read multiple times". However, the write operation is performed in sequence, that is, it is a write operation performed

during file creation or an adding operation performed behind the existing file. HDFS ensures that only one caller can perform write operation on a file but multiple callers can perform read operation on the file at the same time.

HDFS Architecture

HDFS consists of active and standby NameNodes and multiple DataNodes, as shown in **Figure 5-61**.

HDFS works in master/slave architecture. NameNodes run on the master (active) node, and DataNodes run on the slave (standby) node. ZKFC should run along with the NameNodes.

The communication between NameNodes and DataNodes is based on Transmission Control Protocol (TCP)/Internet Protocol (IP). The NameNode, DataNode, ZKFC, and JournalNode can be deployed on Linux servers.

Figure 5-61 HA HDFS architecture

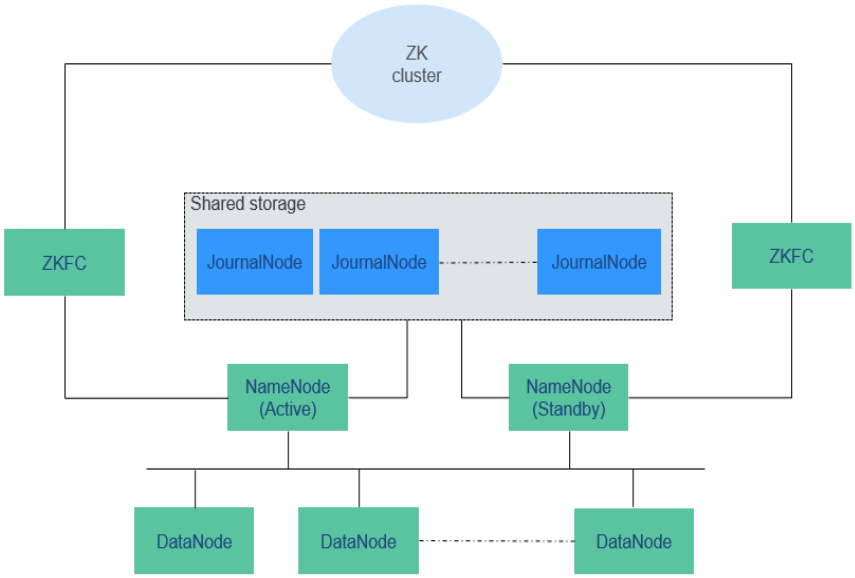


Table 5-13 describes the functions of each module shown in **Figure 5-61**.

Table 5-13 Module description

Module	Description
NameNode	A NameNode is used to manage the namespace, directory structure, and metadata information of a file system and provide the backup mechanism. The NameNode is classified into the following two types: • Active NameNode: manages the namespace, maintains the directory structure and metadata of file systems, and records the mapping relationships between data blocks and files to which the data blocks belong. • Standby NameNode: synchronizes with the data in the active NameNode, and takes over services from the active NameNode when the active NameNode is faulty. • Observer NameNode: synchronizes with the data in the active NameNode, and processes read requests from the client.
DataNode	A DataNode is used to store data blocks of each file and periodically report the storage status to the NameNode.
JournalNode	In HA cluster, synchronizes metadata between the active and standby NameNodes.
ZKFC	ZKFC must be deployed for each NameNode. It monitors NameNode status and writes status information to ZooKeeper. ZKFC also has permissions to select the active NameNode.
ZK Cluster	ZooKeeper is a coordination service which helps the ZKFC to elect the active NameNode.
HttpFS gateway	HttpFS is a single stateless gateway process which provides the WebHDFS REST API for external processes and FileSystem API for the HDFS. HttpFS is used for data transmission between different versions of Hadoop. It is also used as a gateway to access the HDFS behind a firewall.

- **HDFS HA Architecture**

  HA is used to resolve the SPOF problem of NameNode. This feature provides a standby NameNode for the active NameNode. When the active NameNode is faulty, the standby NameNode can quickly take over to continuously provide services for external systems.

  In a typical HDFS HA scenario, there are usually two NameNodes. One is in the active state, and the other in the standby state.

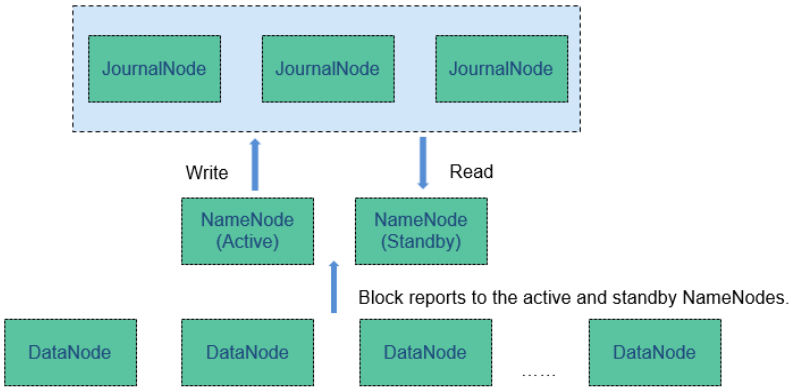
  A shared storage system is required to support metadata synchronization of the active and standby NameNodes. This version provides Quorum Journal Manager (QJM) HA solution, as shown in **Figure 5-62**. A group of JournalNodes are used to synchronize metadata between the active and standby NameNodes.

  Generally, an odd number (2N+1) of JournalNodes are configured, and at least three JournalNodes are required. For one metadata update message,

data writing is considered successful as long as data writing is successful on N +1 JournalNodes. In this case, data writing failure of a maximum of N JournalNodes is allowed. For example, when there are three JournalNodes, data writing failure of one JournalNode is allowed; when there are five JournalNodes, data writing failure of two JournalNodes is allowed.

JournalNode is a lightweight daemon process and shares a host with other services of Hadoop. It is recommended that the JournalNode be deployed on the control node to prevent data writing failure on the JournalNode during massive data transmission.

Figure 5-62 QJM-based HDFS architecture

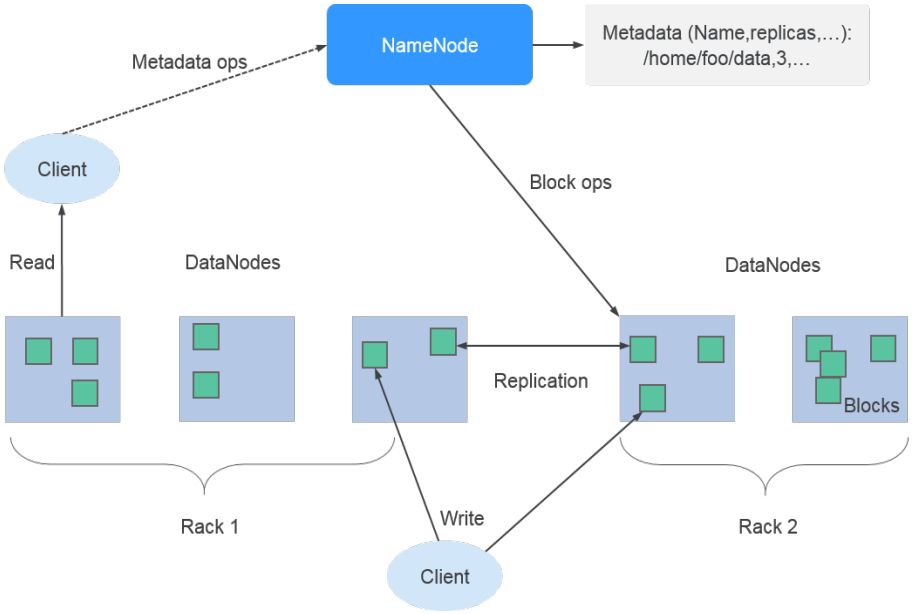


HDFS Reliability

MRS uses the HDFS copy mechanism to ensure data reliability. One backup file is automatically generated for each file saved in HDFS, that is, two copies are generated in total. The number of HDFS copies can be queried using the **dfs.replication** parameter.

- When the Core node specification of the MRS cluster is set to non-local hard disk drive (HDD) and the cluster has only one Core node, the default number of HDFS copies is 1. If the number of Core nodes in the cluster is greater than or equal to 2, the default number of HDFS copies is 2.
- When the Core node specification of the MRS cluster is set to local disk and the cluster has only one Core node, the default number of HDFS copies is 1. If there are two Core nodes in the cluster, the default number of HDFS copies is 2. If the number of Core nodes in the cluster is greater than or equal to 3, the default number of HDFS copies is 3.

Figure 5-63 HDFS architecture



MRS supports balanced node scheduling on HDFS and balanced disk scheduling on a single node, improving HDFS storage performance after node or disk scale-out.

For details about the Hadoop architecture and principles, see **https://hadoop.apache.org/**.

5.14.2 HDFS HA Solution

HDFS HA Background

In versions earlier than Hadoop 2.0.0, SPOF occurs in the HDFS cluster. Each cluster has only one NameNode. If the host where the NameNode is located is faulty, the HDFS cluster cannot be used unless the NameNode is restarted or started on another host. This affects the overall availability of HDFS in the following aspects:

1. In the case of an unplanned event such as host breakdown, the cluster would be unavailable until the NameNode is restarted.
2. Planned maintenance tasks, such as software and hardware upgrade, will cause the cluster stop working.

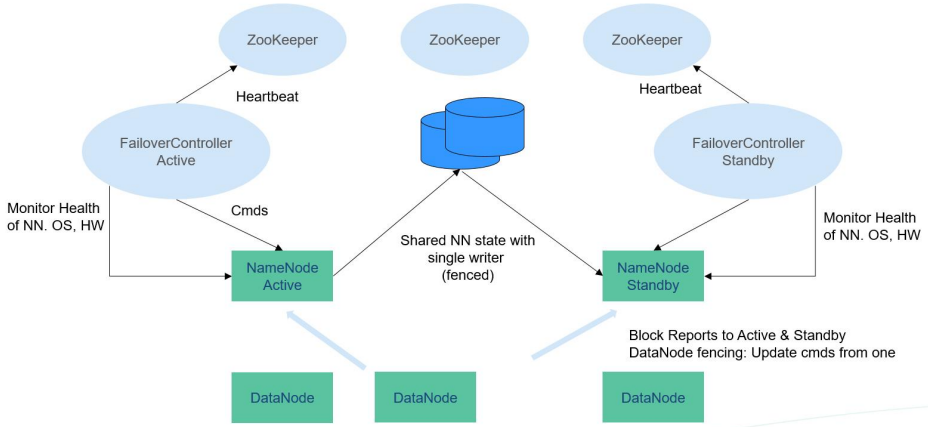
To solve the preceding problems, the HDFS HA solution enables a hot-swap NameNode backup for NameNodes in a cluster in automatic or manual (configurable) mode. When a machine fails (due to hardware failure), the active/standby NameNode switches over automatically in a short time. When the active NameNode needs to be maintained, the MRS cluster administrator can manually perform an active/standby NameNode switchover to ensure cluster availability during maintenance.

For details about HDFS automatic failover, see

https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html#Automatic_Failover

HDFS HA Implementation

Figure 5-64 Typical HA deployment



In a typical HA cluster (as shown in **Figure 5-64**), two NameNodes need to be configured on two independent servers, respectively. At any time point, one NameNode is in the active state, and the other NameNode is in the standby state. The active NameNode is responsible for all client operations in the cluster, while the standby NameNode maintains synchronization with the active node to provide fast switchover if necessary.

To keep the data synchronized with each other, both nodes communicate with a group of JournalNodes. When the active node modifies any file system's metadata, it will store the modification log to a majority of these JournalNodes. For example, if there are three JournalNodes, then the log will be saved on two of them at least. The standby node monitors changes of JournalNodes and synchronizes changes from the active node. Based on the modification log, the standby node applies the changes to the metadata of the local file system. Once a switchover occurs, the standby node can ensure its status is the same as that of the active node. This ensures that the metadata of the file system is synchronized between the active and standby nodes if the switchover is incurred by the failure of the active node.

To ensure fast switchover, the standby node needs to have the latest block information. Therefore, DataNodes send block information and heartbeat messages to two NameNodes at the same time.

It is vital for an HA cluster that only one of the NameNodes be active at any time. Otherwise, the namespace state would split into two parts, risking data loss or other incorrect results. To prevent the so-called "split-brain scenario", the JournalNodes will only ever allow a single NameNode to write data to it at a time. During switchover, the NameNode which is to become active will take over the role of writing data to JournalNodes. This effectively prevents the other NameNodes from being in the active state, allowing the new active node to safely proceed with switchover.

For more information about the HDFS HA solution, visit the following website:

https://hadoop.apache.org/docs/r3.3.1/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html

5.14.3 Relationship Between HDFS and Other Components

Relationship Between HDFS and HBase

HDFS is a subproject of Apache Hadoop, which is used as the file storage system for HBase. HBase is located in the structured storage layer. HDFS provides highly reliable support for lower-layer storage of HBase. All the data files of HBase can be stored in the HDFS, except some log files generated by HBase.

Relationship Between HDFS and MapReduce

- HDFS features high fault tolerance and high throughput, and can be deployed on low-cost hardware for storing data of applications with massive data sets.
- MapReduce is a programming model used for parallel computation of large data sets (larger than 1 TB). Data computed by MapReduce comes from multiple data sources, such as Local FileSystem, HDFS, and databases. Most data comes from the HDFS. The high throughput of HDFS can be used to read massive data. After being computed, data can be stored in HDFS.

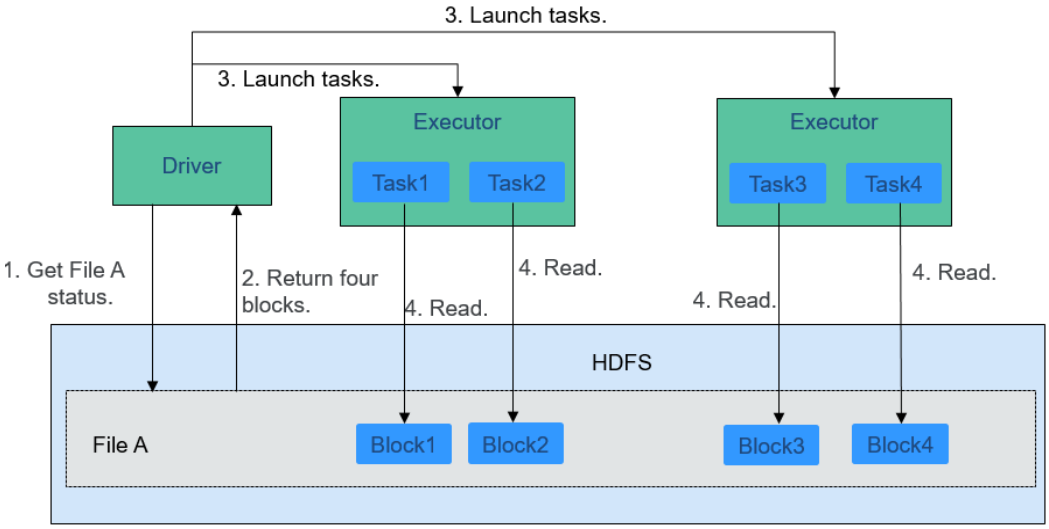
Relationship Between HDFS and Spark

Data computed by Spark comes from multiple data sources, such as local files and HDFS. Most data comes from HDFS which can read data in large scale for parallel computing. After being computed, data can be stored in HDFS.

Spark involves Driver and Executor. Driver schedules tasks and Executor runs tasks.

Figure 5-65 shows how data is read from a file.

Figure 5-65 File reading process



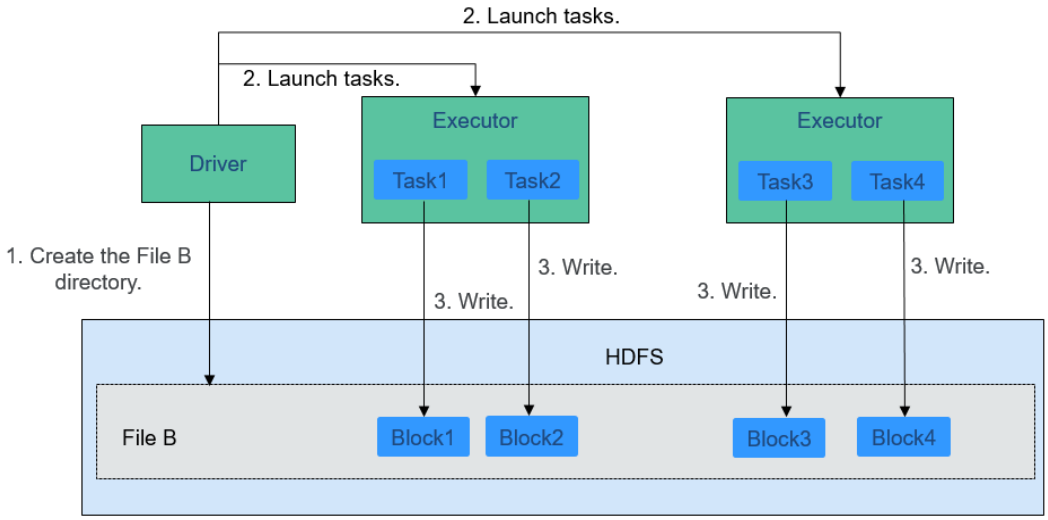
The file reading process is as follows:

1. Driver interconnects with HDFS to obtain the information of File A.
2. The HDFS returns the detailed block information about this file.
3. Driver sets a parallel degree based on the block data amount, and creates multiple tasks to read the blocks of this file.

4. Executor runs the tasks and reads the detailed blocks as part of the Resilient Distributed Dataset (RDD).

Figure 5-66 shows how data is written to a file.

Figure 5-66 File writing process



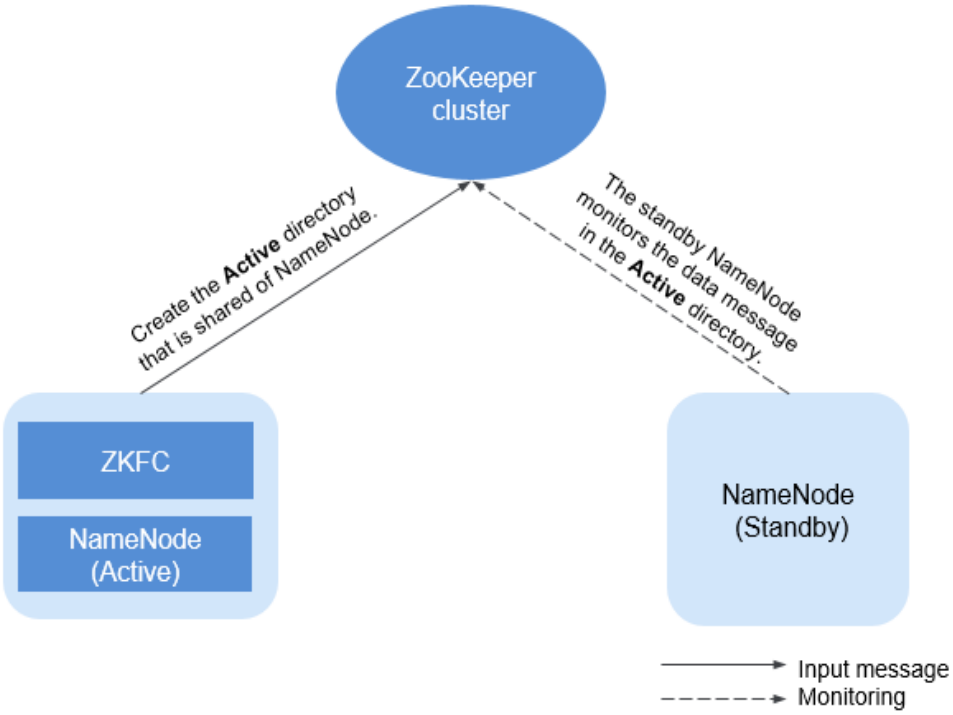
The file writing process is as follows:

1. Driver creates a directory where the file is to be written.
2. Based on the RDD distribution status, the number of tasks related to data writing is computed, and these tasks are sent to Executor.
3. Executor runs these tasks, and writes the computed RDD data to the directory created in 1.

Relationship Between HDFS and ZooKeeper

Figure 5-67 shows the relationship between ZooKeeper and HDFS.

Figure 5-67 Relationship between ZooKeeper and HDFS



As the client of a ZooKeeper cluster, ZKFailoverController (ZKFC) monitors the status of NameNode. ZKFC is deployed only in the node where NameNode resides, and in both the active and standby HDFS NameNodes.

1. The ZKFC connects to ZooKeeper and saves information such as host names to ZooKeeper under the znode directory **/hadoop-ha**. NameNode that creates the directory first is considered as the active node, and the other is the standby node. NameNodes read the NameNode information periodically through ZooKeeper.
2. When the process of the active node ends abnormally, the standby NameNode detects changes in the **/hadoop-ha** directory through ZooKeeper, and then takes over the service of the active NameNode.

5.14.4 HDFS Enhanced Open Source Features

Enhanced Open Source Feature: File Block Colocation

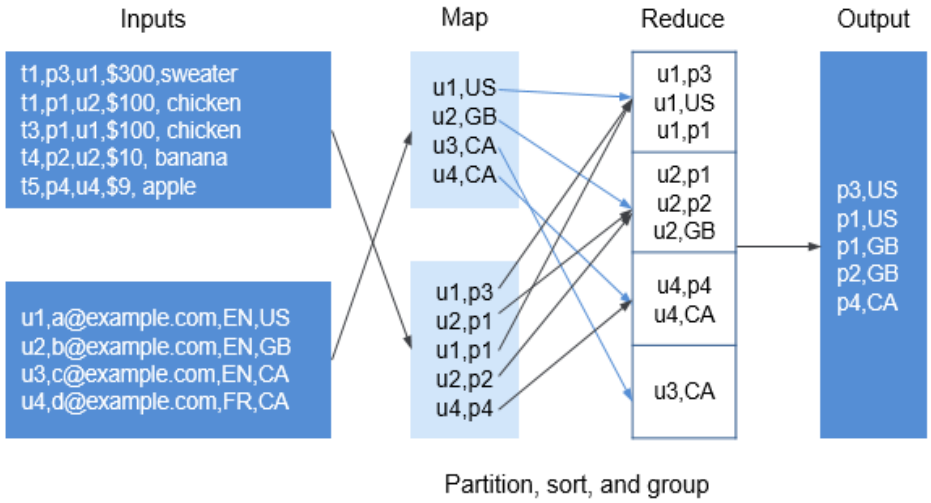
In the offline data summary and statistics scenario, Join is a frequently used computing function, and is implemented in MapReduce as follows:

1. The Map task processes the records in the two table files into Join Key and Value, performs hash partitioning by Join Key, and sends the data to different Reduce tasks for processing.
2. Reduce tasks read data in the left table recursively in the nested loop mode and traverse each line of the right table. If join key values are identical, join results are output.

   The preceding method sharply reduces the performance of the join calculation. Because a large amount of network data transfer is required

when the data stored in different nodes is sent from MAP to Reduce, as shown in **Figure 5-68**.

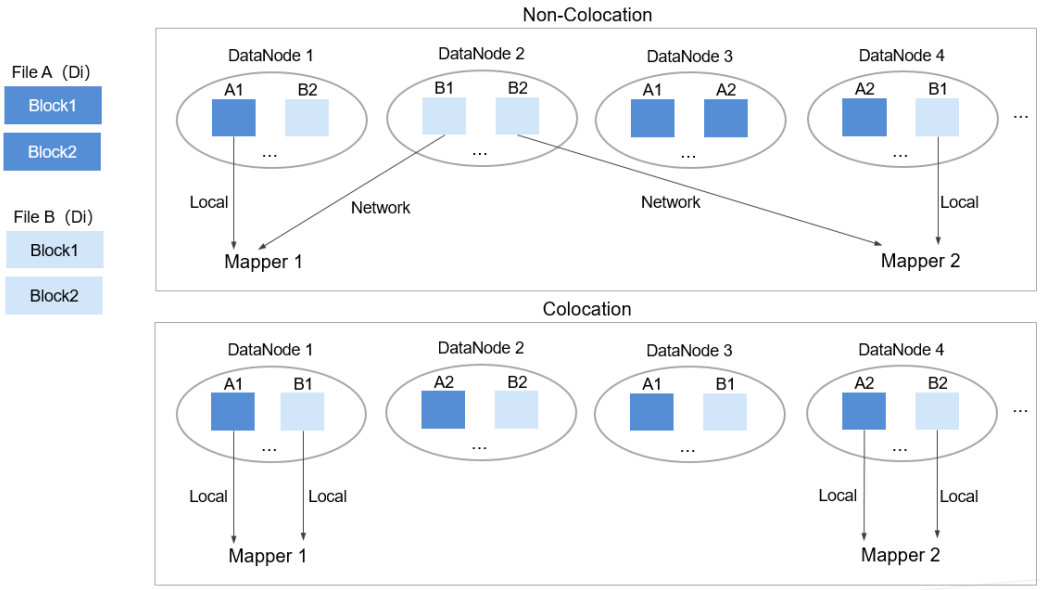
Figure 5-68 Data transmission in the non-colocation scenario



Data tables are stored in physical file system by HDFS block. Therefore, if two to-be-joined blocks are put into the same host accordingly after they are partitioned by join key, you can obtain the results directly from Map join in the local node without any data transfer in the Reduce process of the join calculation. This will greatly improve the performance.

With the identical distribution feature of HDFS data, a same distribution ID is allocated to files, FileA and FileB, on which association and summation calculations need to be performed. In this way, all the blocks are distributed together, and calculation can be performed without retrieving data across nodes, which greatly improves the MapReduce join performance.

Figure 5-69 Data block distribution in colocation and non-colocation scenarios

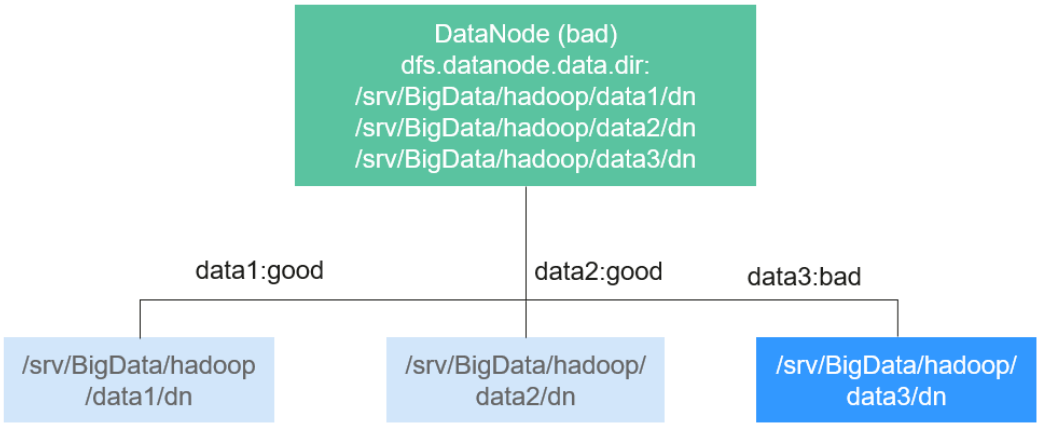


Enhanced Open Source Feature: Damaged Hard Disk Volume Configuration

In the open source version, if multiple data storage volumes are configured for a DataNode, the DataNode stops providing services by default if one of the volumes is damaged. If the configuration item **dfs.datanode.failed.volumes.tolerated** is set to specify the number of damaged volumes that are allowed, DataNode continues to provide services when the number of damaged volumes does not exceed the threshold.

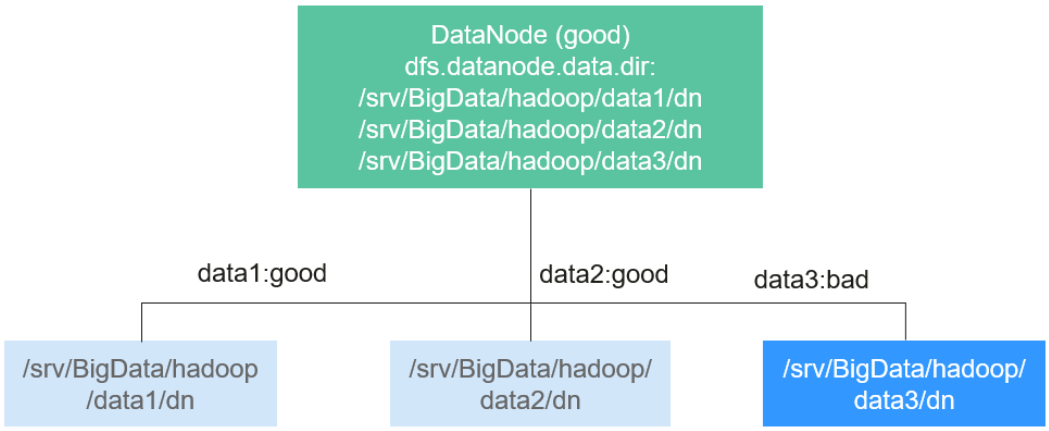
The value of **dfs.datanode.failed.volumes.tolerated** ranges from -1 to the number of disk volumes configured on the DataNode. The default value is **-1**, as shown in **Figure 5-70**.

Figure 5-70 Item being set to 0



For example, three data storage volumes are mounted to a DataNode, and **dfs.datanode.failed.volumes.tolerated** is set to 1. In this case, if one data storage volume of the DataNode is unavailable, this DataNode can still provide services, as shown in **Figure 5-71**.

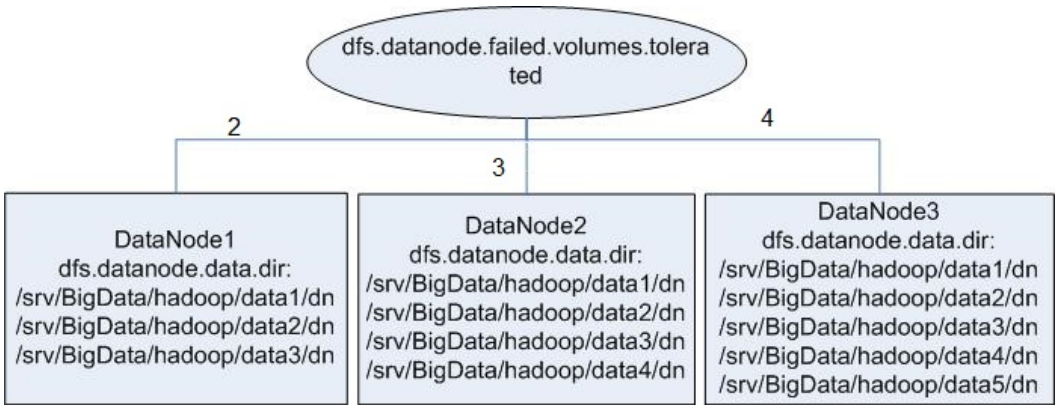
Figure 5-71 Item being set to 1



This native configuration item has some defects. When the number of data storage volumes in each DataNode is inconsistent, you need to configure each DataNode independently instead of generating the unified configuration file for all nodes.

Assume that there are three DataNodes in a cluster. The first node has three data directories, the second node has four, and the third node has five. If you want to ensure that DataNode services are available when only one data directory is available, you need to perform the configuration as shown in **Figure 5-72**.

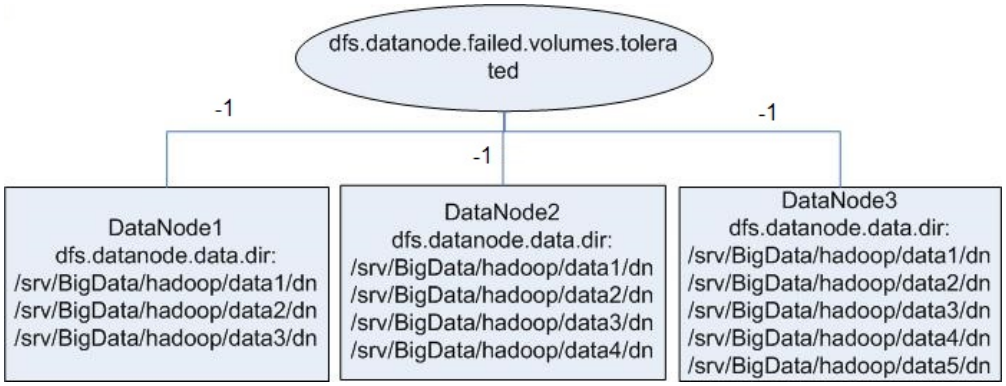
Figure 5-72 Attribute configuration before being enhanced



In self-developed enhanced HDFS, this configuration item is enhanced, with a value **-1** added. When this configuration item is set to **-1**, all DataNodes can provide services as long as one data storage volume in all DataNodes is available.

To resolve the problem in the preceding example, set this configuration to **-1**, as shown in **Figure 5-73**.

Figure 5-73 Attribute configuration after being enhanced



Enhanced Open Source Feature: HDFS Startup Acceleration

In HDFS, when NameNodes start, the metadata file FsImage needs to be loaded. Then, DataNodes will report the data block information after the DataNodes startup. When the data block information reported by DataNodes reaches the preset percentage, NameNodes exits safe mode to complete the startup process. If the number of files stored on the HDFS reaches the million or billion level, the two processes are time-consuming and will lead to a long startup time of the NameNode. Therefore, this version optimizes the process of loading metadata file FsImage.

In the open source HDFS, FsImage stores all types of metadata information. Each type of metadata information (such as file metadata information and folder

metadata information) is stored in a section block, respectively. These section blocks are loaded in serial mode during startup. If a large number of files and folders are stored on the HDFS, loading of the two sections is time-consuming, prolonging the HDFS startup time. HDFS NameNode divides each type of metadata by segments and stores the data in multiple sections when generating the FsImage files. When the NameNodes start, sections are loaded in parallel mode. This accelerates the HDFS startup.

Enhanced Open Source Feature: Label-based Block Placement Policies (HDFS Nodelabel)

You need to configure the nodes for storing HDFS file data blocks based on data features. You can configure a label expression to an HDFS directory or file and assign one or more labels to a DataNode so that file data blocks can be stored on specified DataNodes. If the label-based data block placement policy is used for selecting DataNodes to store the specified files, the DataNode range is specified based on the label expression. Then proper nodes are selected from the specified range.

- You can store the replicas of data blocks to the nodes with different labels accordingly. For example, store two replicas of the data block to the node labeled with L1, and store other replicas of the data block to the nodes labeled with L2.
- You can set the policy in case of block placement failure, for example, select a node from all nodes randomly.

Figure 5-74 gives an example:

- Data in **/HBase** is stored in A, B, and D.
- Data in **/Spark** is stored in A, B, D, E, and F.
- Data in **/user** is stored in C, D, and F.
- Data in **/user/shl** is stored in A, E, and F.

Figure 5-74 Example of label-based block placement policy

Enhanced Open Source Feature: HDFS Load Balance

The current read and write policies of HDFS are mainly for local optimization without considering the actual load of nodes or disks. Based on I/O loads of different nodes, the load balance of HDFS ensures that when read and write operations are performed on the HDFS client, the node with low I/O load is selected to perform such operations to balance I/O load and fully utilize the overall throughput of the cluster.

If HDFS Load Balance is enabled during file writing, the NameNode selects a DataNode (in the order of local node, local rack, and remote rack). If the I/O load of the selected node is heavy, the NameNode will choose another DataNode with lighter load.

If HDFS Load Balance is enabled during file reading, an HDFS client sends a request to the NameNode to provide the list of DataNodes that store the block to be read. The NameNode returns a list of DataNodes sorted by distance in the network topology. With the HDFS Load Balance feature, the DataNodes on the list are also sorted by their I/O load. The DataNodes with heavy load are at the bottom of the list.

Enhanced Open Source Feature: HDFS Auto Data Movement

Hadoop has been used for batch processing of immense data in a long time. The existing HDFS model is used to fit the needs of batch processing applications very well because such applications focus more on throughput than delay.

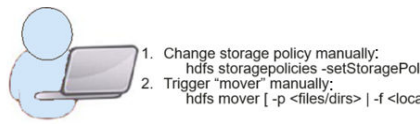
However, as Hadoop is increasingly used for upper-layer applications that demand frequent random I/O access such as Hive and HBase, low latency disks such as solid state disk (SSD) are favored in delay-sensitive scenarios. To cater to the trend, HDFS supports a variety of storage types. Users can choose a storage type according to their needs.

Storage policies vary depending on how frequently data is used. For example, if data that is frequently accessed in the HDFS is marked as **ALL_SSD** or **HOT**, the data that is accessed several times may be marked as **WARM**, and data that is rarely accessed (only once or twice access) can be marked as **COLD**. You can select different data storage policies based on the data access frequency.

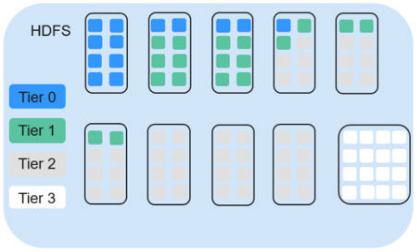
However, low latency disks are far more expensive than spinning disks. Data typically sees heavy initial usage with decline in usage over a period of time. Therefore, it can be useful if data that is no longer used is moved out from expensive disks to cheaper ones storage media.

A typical example is storage of detail records. New detail records are imported into SSD because they are frequently queried by upper-layer applications. As access frequency to these detail records declines, they are moved to cheaper storage.

Before automatic data movement is achieved, you have to manually determine by service type whether data is frequently used, manually set a data storage policy, and manually trigger the HDFS Auto Data Movement Tool, as shown in the figure below.



Policy ID	PolicyName	Block Placement (n replacas)	Fallback storages for creation	Fallback storages for replication
15	Lazy_Persist	RAN_DISK:1 DISK:n-1	DISK	DISK
12	All_SSD	SSD:n	DISK	DISK
10	One_SSD	SSD:1,DISK:n-1	SSD,DISK	SSD,DISK
7	Hot(default)	DISK:n	<none>	ARCHIVE
5	Warm	DISK:1,ARCHIVE:n-1	ARCHIVE, DISK	ARCHIVE, DISK
2	Cold	ARCHIVE:n	<none>	<none>



If aged data can be automatically identified and moved to cheaper storage (such as disk/archive), you will see significant cost cuts and data management efficiency improvement.

The HDFS Auto Data Movement Tool is at the core of HDFS Auto Data Movement. It automatically sets a storage policy depending on how frequently data is used. Specifically, functions of the HDFS Auto Data Movement Tool can:

- Mark a data storage policy as **All_SSD**, **One_SSD**, **Hot**, **Warm**, **Cold**, or **FROZEN** according to age, access time, and manual data movement rules.
- Define rules for distinguishing cold and hot data based on the data age, access time, and manual migration rules.
- Define the action to be taken if age-based rules are met.

  MARK: the action for identifying whether data is frequently or rarely used based on the age rules and setting a data storage policy. **MOVE**: the action for invoking the HDFS Auto Data Movement Tool and moving data based on the age rules to identify whether data is frequently or rarely used after you have determined the corresponding storage policy.
  - **MARK**: identifies whether data is frequently or rarely used and sets the data storage policy.
  - **MOVE**: the action for invoking the HDFS Auto Data Movement Tool and moving data across tiers.
  - **SET_REPL**: the action for setting new replica quantity for a file.
  - **MOVE_TO_FOLDER**: the action for moving files to a target folder.
  - **DELETE**: the action for deleting a file or directory.
  - **SET_NODE_LABEL**: the action for setting node labels of a file.

With the HDFS Auto Data Movement feature, you only need to define age based on access time rules. HDFS Auto Data Movement Tool matches data according to age-based rules, sets storage policies, and moves data. In this way, data management efficiency and cluster resource efficiency are improved.

5.15 HetuEngine

5.15.1 HetuEngine Product Overview

HetuEngine Description

HetuEngine is a self-developed high-performance, interactive SQL analysis and data virtualization engine. It seamlessly integrates with the big data ecosystem to implement interactive query of massive amounts of data within seconds, and supports cross-source and cross-domain unified data access to enable one-stop SQL convergence analysis in the data lake, between lakes, and between lakehouses.

HetuEngine Architecture

HetuEngine consists of different modules. **Figure 5-75** shows the structure of HetuEngine. **Table 5-14** describes the basic concepts of HetuEngine.

Figure 5-75 HetuEngine architecture

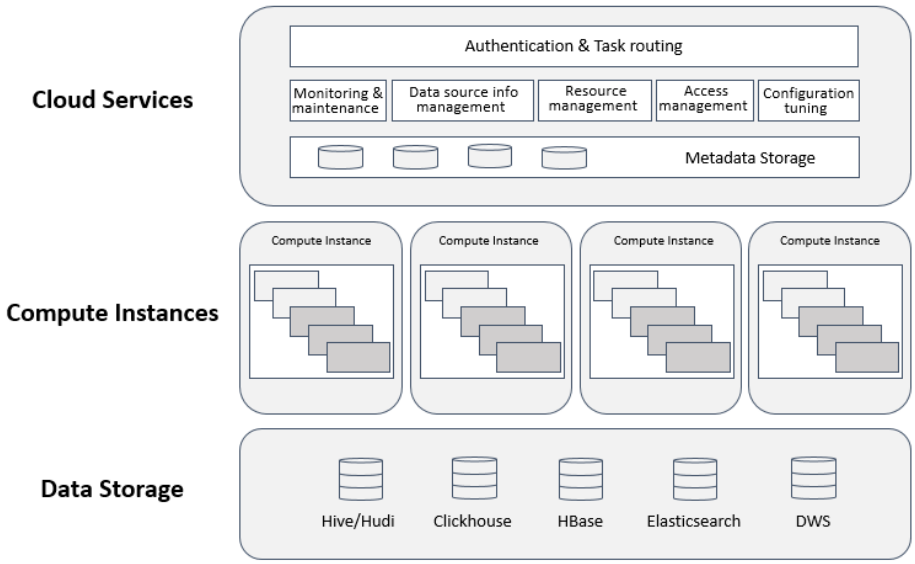


Table 5-14 Module description

Module	Concept	Description
Cloud service layer	HetuEngine CLI/ JDBC	HetuEngine client, through which query requests are submitted and results are returned and displayed.
	HSBroker	Service management component of HetuEngine. It manages and verifies compute instances, monitors health status, and performs automatic maintenance.
	HSConsole	Provides visualized operation GUIs and RESTful APIs for data source information management, compute instance management, and automatic task query.
	HSFabric	Provides a unified SQL access entry to meet the requirements for high-performing and highly secure data transfer across domains (data centers).
	QAS	An in-house module of HetuEngine. It provides automatic detection, learning, and diagnosis of historical SQL execution records for more efficient online SQL O&M and faster online SQL analysis.
Engine layer	Coordinator	Management node of HetuEngine compute instances. It receives and parses SQL statements, generates and optimizes execution plans, assigns tasks, and schedules resources.
	Worker	Work node of HetuEngine compute instances. It provides capabilities such as parallel data pulling from data sources and distributed SQL computing.

HetuEngine Application Scenarios

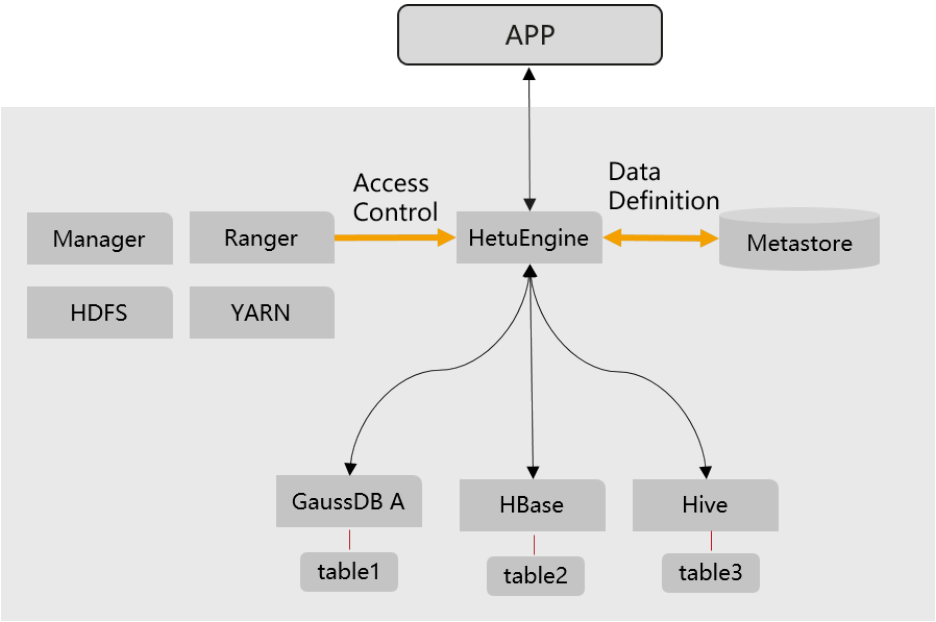
HetuEngine supports cross-source (multiple data sources, such as Hive, HBase, GaussDB(DWS), Elasticsearch, and ClickHouse) and cross-domain (multiple regions or data centers) quick joint query, especially for interactive quick query of Hive and Hudi data in the Hadoop cluster (MRS).

Using the HetuEngine Cross-Source Function

Enterprises usually store massive data, such as from various databases and warehouses, for management and information collection. However, diversified data sources, hybrid dataset structures, and scattered data storage rise the development cost for cross-source query and prolong the cross-source query duration.

HetuEngine provides unified standard SQL statements to implement cross-source collaborative analysis, simplifying cross-source analysis operations.

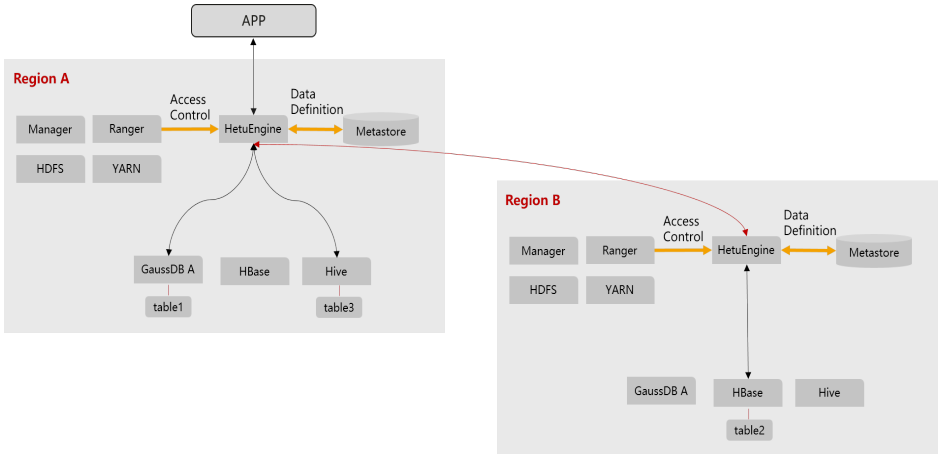
Figure 5-76 HetuEngine cross-source function



Using the HetuEngine Cross-Domain Function

HetuEngine provide unified standard SQL to implement efficient access to multiple data sources distributed in multiple regions (or data centers), shields data differences in the structure, storage, and region, and decouples data and applications.

Figure 5-77 HetuEngine cross-region functions



5.15.2 Relationship Between HetuEngine and Other Components

The HetuEngine installation depends on the MRS cluster. **Table 5-15** lists the components on which the HetuServer installation depends.

Table 5-15 Components on which HetuEngine depends

Name	Description
HDFS	Hadoop Distributed File System, supporting high-throughput data access and suitable for applications with large-scale data sets.
Hive	Open-source data warehouse built on Hadoop. It stores structured data and implements basic data analysis using the Hive Query Language (HQL), a SQL-like language.
ZooKeeper	Enables highly reliable distributed coordination. It helps prevent single point of failures (SPOFs) and provides reliable services for applications.
KrbServer	Key management center that distributes bills.
Yarn	Resource management system, which is a general resource module that manages and schedules resources for various applications.
DBService	DBService is a high-availability relational database storage system that provides metadata backup and restoration functions.

5.16 Apache Hive

5.16.1 Hive Basic Principles

Hive is a data warehouse built on Hadoop. It provides batch computing capability for the big data platform and is able to batch analyze and summarize structured and semi-structured data for data calculation. Hive operates structured data using Hive Query Language (HQL), a SQL-like language. HQL is automatically converted into MapReduce tasks for the query and analysis of massive data in the Hadoop cluster. For more information about Hive tables, see the **Hive tutorial** of the open source community.

Hive provides the following functions:

- Analyzes massive structured data and summarizes analysis results.
- Allows complex MapReduce jobs to be compiled in SQL languages.
- Supports flexible data storage formats, including JavaScript object notation (JSON), comma separated values (CSV), TextFile, RCFile, SequenceFile, and ORC (Optimized Row Columnar).

Hive Architecture

Hive is a single-instance service process that provides services by translating HQL into related MapReduce jobs or HDFS operations. **Figure 5-78** shows how Hive is connected to other components.

Figure 5-78 Hive framework

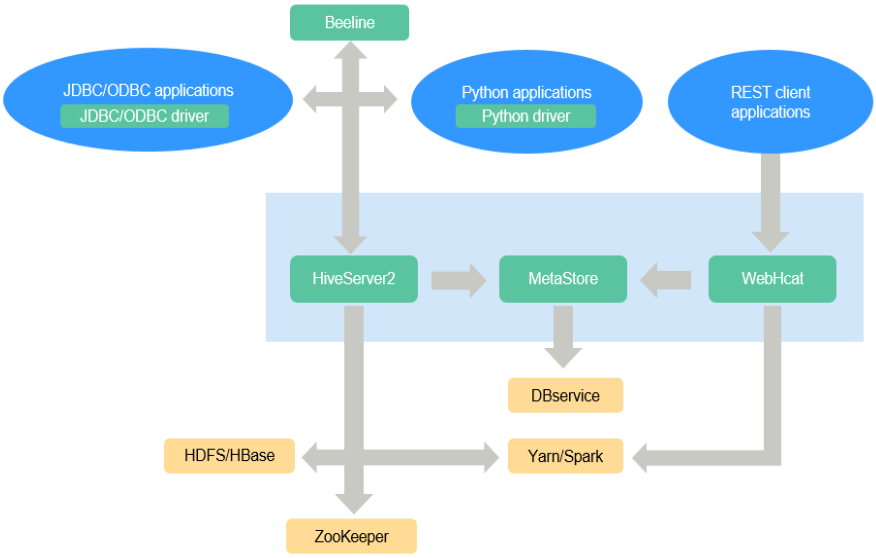
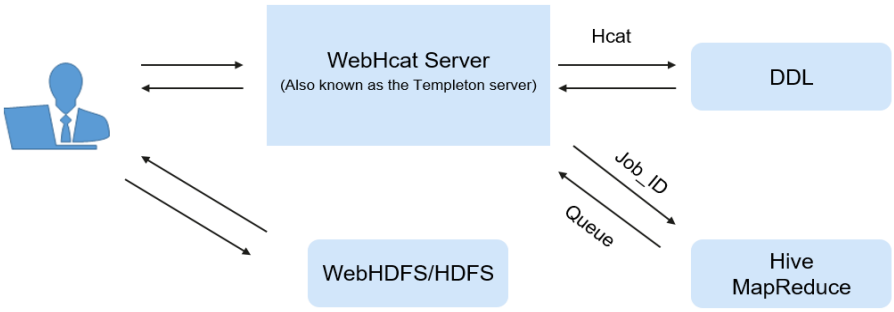


Table 5-16 Module description

Module	Description
HiveServer	Multiple HiveServers can be deployed in a cluster to share loads. HiveServer provides Hive database services externally, translates HQL statements into related YARN tasks or HDFS operations to complete data extraction, conversion, and analysis.
MetaStore	• Multiple MetaStores can be deployed in a cluster to share loads. MetaStore provides Hive metadata services as well as reads, writes, maintains, and modifies the structure and properties of Hive tables. • MetaStore provides Thrift APIs for HiveServer, Spark, WebHCat, and other MetaStore clients to access and operate metadata.
WebHCat	Multiple WebHCats can be deployed in a cluster to share loads. WebHCat provides REST APIs and runs the Hive commands through the REST APIs to submit MapReduce jobs.
Hive client	Hive client includes the human-machine command-line interface (CLI) Beeline, JDBC drive for JDBC applications, Python driver for Python applications, and HCatalog JAR files for MapReduce.
ZooKeeper cluster	As a temporary node, ZooKeeper records the IP address list of each HiveServer instance. The client driver connects to ZooKeeper to obtain the list and selects corresponding HiveServer instances based on the routing mechanism.
HDFS/HBase cluster	The HDFS cluster stores the Hive table data.
MapReduce/ YARN cluster	Provides distributed computing services. Most Hive data operations rely on MapReduce. The main function of HiveServer is to translate HQL statements into MapReduce jobs to process massive data.

HCatalog is built on Hive Metastore and incorporates the DDL capability of Hive. HCatalog is also a Hadoop-based table and storage management layer that enables convenient data read/write on tables of HDFS using different data processing tools such as MapReduce. HCatalog also provides read/write APIs for these tools and uses a Hive CLI to publish commands for defining data and querying metadata. After encapsulating these commands, WebHCat Server can provide RESTful APIs, as shown in **Figure 5-79**.

Figure 5-79 WebHCat logical architecture



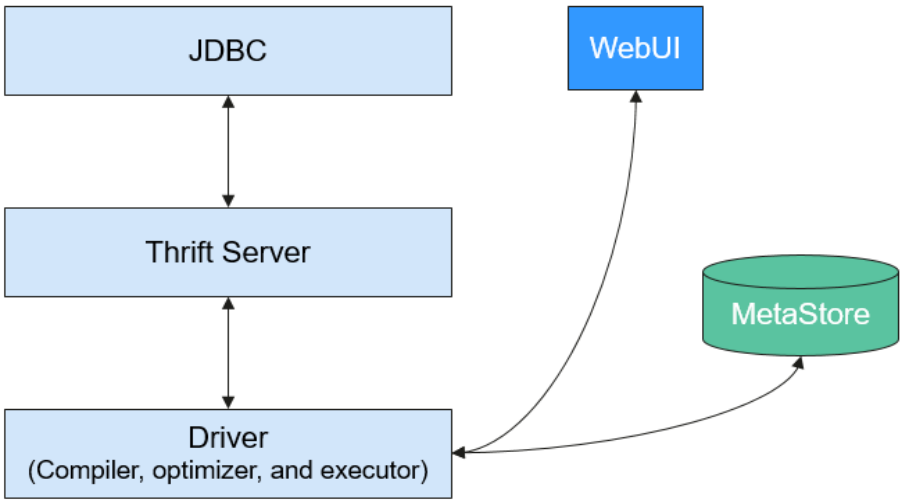
Principles

Hive functions as a data warehouse based on HDFS and MapReduce architecture and translates HQL statements into MapReduce jobs or HDFS operations. For details about Hive and HQL, see **HiveQL Language Manual**.

Figure 5-80 shows the Hive structure.

- **Metastore**: reads, writes, and updates metadata such as tables, columns, and partitions. Its lower layer is relational databases.
- **Driver**: manages the lifecycle of HiveQL execution and participates in the entire Hive job execution.
- **Compiler**: translates HQL statements into a series of interdependent Map or Reduce jobs.
- **Optimizer**: is classified into logical optimizer and physical optimizer to optimize HQL execution plans and MapReduce jobs, respectively.
- **Executor**: runs Map or Reduce jobs based on job dependencies.
- **ThriftServer**: functions as the servers of JDBC, provides Thrift APIs, and integrates with Hive and other applications.
- **Clients**: include the WebUI and JDBC APIs and provides APIs for user access.

Figure 5-80 Hive framework



5.16.2 Hive CBO Principles

Hive CBO Principles

CBO is short for Cost-Based Optimization.

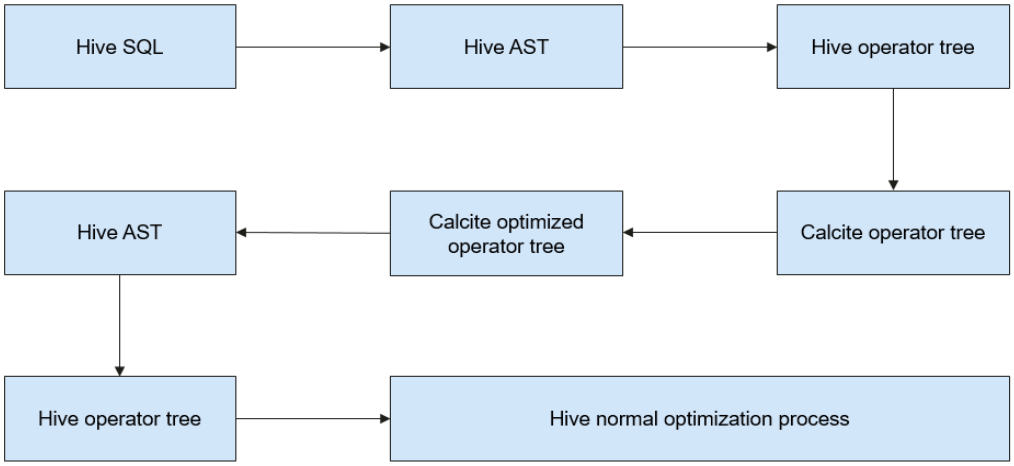
It will optimize the following:

During compilation, the CBO calculates the most efficient join sequence based on tables and query conditions involved in query statements to reduce time and resources required for query.

In Hive, the CBO is implemented as follows:

Hive uses open-source component Apache Calcite to implement the CBO. SQL statements are first converted into Hive Abstract Syntax Trees (ASTs) and then into RelNodes that can be identified by Calcite. After Calcite adjusts the join sequence in RelNodes, RelNodes are converted into ASTs by Hive to continue the logical and physical optimization. **Figure 5-81** shows the working flow.

Figure 5-81 CBO Implementation process



Calcite adjusts the join sequence as follows:

1. A table is selected as the first table from the tables to be joined.
2. The second and third tables are selected based on the cost. In this way, multiple different execution plans are obtained.
3. A plan with the minimum costs is calculated and serves as the final sequence.

The cost calculation method is as follows:

In the current version, costs are measured based on the number of data entries after joining. Fewer data entries mean less cost. The number of joined data entries depends on the selection rate of joined tables. The number of data entries in a table is obtained based on the table-level statistics.

The number of data entries in a table after filtering is estimated based on the column-level statistics, including the maximum values (max), minimum values (min), and Number of Distinct Values (NDV).

For example, there is a table **table_a** whose total number of data records is 1,000,000 and NDV is 50. The query conditions are as follows:

```
Select * from table_a where colum_a='value1';
```

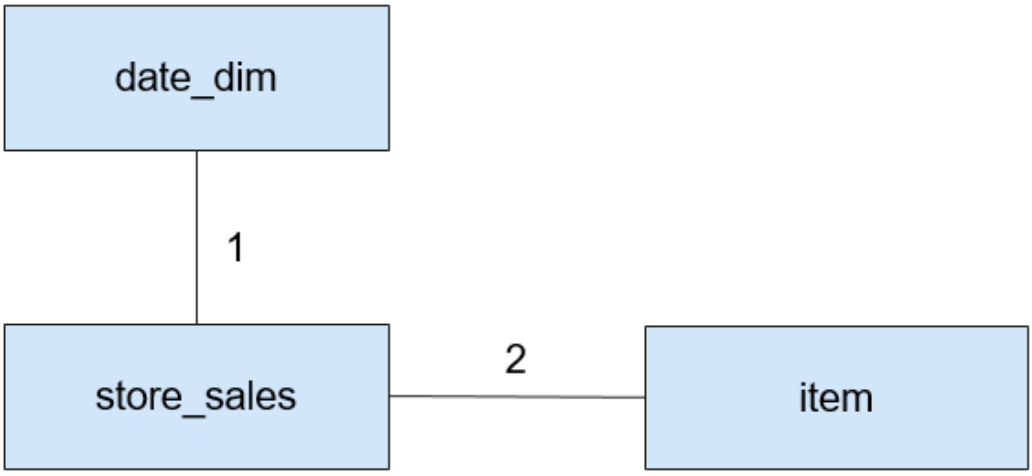
The estimated number of queried data entries is: 1,000,000 x 1/50 = 20,000. The selection rate is 2%.

The following takes the TPC-DS Q3 as an example to describe how the CBO adjusts the join sequence:

```
select
    dt.d_year,
    item.i_brand_id brand_id,
    item.i_brand brand,
    sum(ss_ext_sales_price) sum_agg
from
    date_dim dt,
    store_sales,
    item
where
    dt.d_date_sk = store_sales.ss_sold_date_sk
    and store_sales.ss_item_sk = item.i_item_sk
    and item.i_manufact_id = 436
    and dt.d_moy = 12
group by dt.d_year , item.i_brand , item.i_brand_id
order by dt.d_year , sum_agg desc , brand_id
limit 10;
```

Statement explanation: This statement indicates that inner join is performed for three tables: table **store_sales** is a fact table with about 2,900,000,000 data entries, table **date_dim** is a dimension table with about 73,000 data entries, and table **item** is a dimension table with about 18,000 data entries. Each table has filtering conditions. **Figure 5-82** shows the join relationship.

Figure 5-82 Join relationship



The CBO must first select the tables that bring the best filtering effect for joining.

By analyzing min, max, NDV, and the number of data entries, the CBO estimates the selection rates of different dimension tables, as shown in **Table 5-17**.

Table 5-17 Data filtering

Table	Number of Original Data Entries	Number of Data Entries After Filtering	Selection Rate
date_dim	73,000	6,200	8.5%
item	18,000	19	0.1%

The selection rate can be estimated as follows: Selection rate = Number of data entries after filtering/Number of original data entries

As shown in the preceding table, the **item** table has a better filtering effect. Therefore, the CBO joins the **item** table first before joining the **date_dim** table.

Figure 5-83 shows the join process when the CBO is disabled.

Figure 5-83 Join process when the CBO is disabled

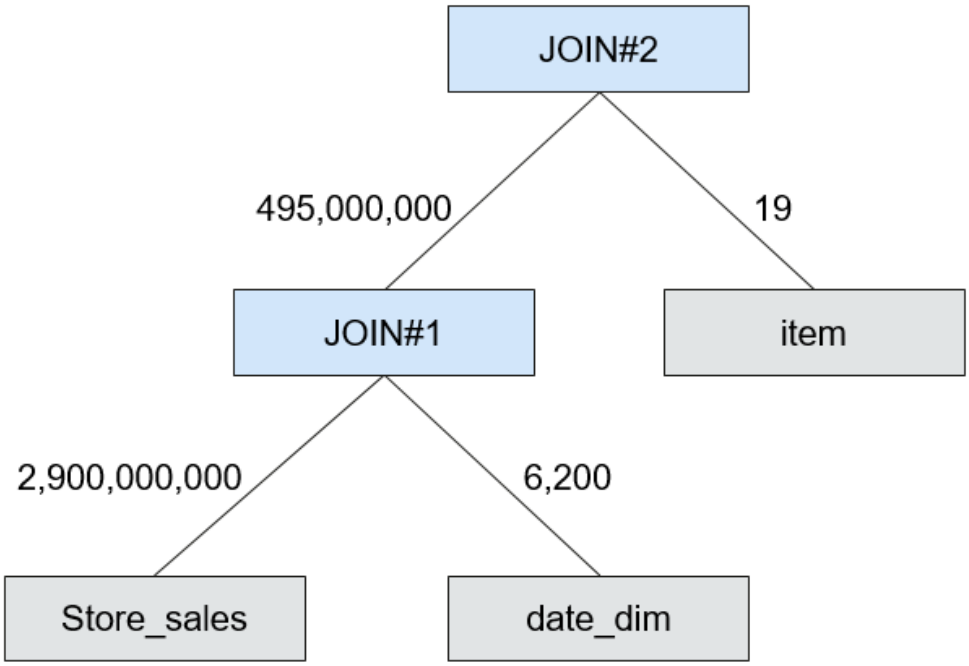
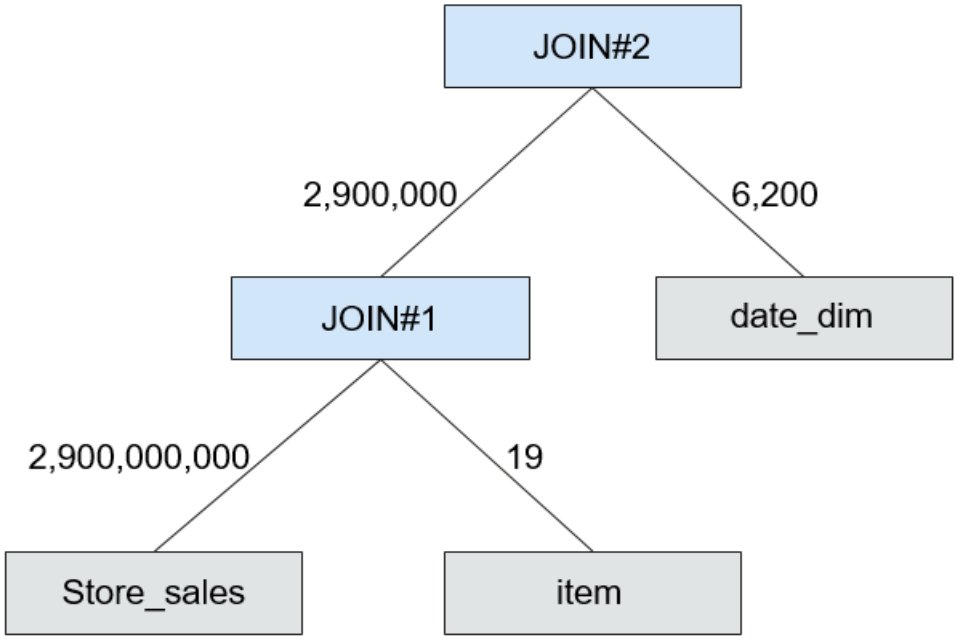


Figure 5-84 shows the join process when the CBO is enabled.

Figure 5-84 Join process when the CBO is enabled



After the CBO is enabled, the number of intermediate data entries is reduced from 495,000,000 to 2,900,000 and thus the execution time can be remarkably reduced.

5.16.3 Relationship Between Hive and Other Components

Relationship Between Hive and HDFS

Hive is a sub-project of Apache Hadoop, which uses HDFS as the file storage system. It parses and processes structured data with highly reliable underlying storage supported by HDFS. All data files in the Hive database are stored in HDFS, and all data operations on Hive are also performed using HDFS APIs.

Relationship Between Hive and MapReduce

Hive data computing depends on MapReduce. MapReduce is also a sub-project of Apache Hadoop and is a parallel computing framework based on HDFS. During data analysis, Hive parses HQL statements submitted by users into MapReduce tasks and submits the tasks for MapReduce to execute.

Relationship Between Hive and Tez

Tez, an open-source project of Apache, is a distributed computing framework that supports directed acyclic graphs (DAGs). When Hive uses the Tez engine to analyze data, it parses HQL statements submitted by users into Tez tasks and submits the tasks to Tez for execution.

Relationship Between Hive and DBService

MetaStore (metadata service) of Hive processes the structure and attribute information of Hive metadata, such as Hive databases, tables, and partitions. The

information needs to be stored in a relational database and is managed and processed by MetaStore. In the product, the metadata of Hive is stored and maintained by the DBService component.

Relationship Between Hive and Elasticsearch

Hive uses Elasticsearch as its extended file storage system. Hive integrates the Elasticsearch-Hadoop plug-in of Elasticsearch, creates a foreign table, and stores table data in Elasticsearch so that Hive can read and write Elasticsearch index data.

Relationship Between Hive and Spark

Spark can be used as the execution engine of Hive. Hive SQL statements delivered by the client are processed at the logical layer on Hive, and physical execution plans are generated and converted into a directed acyclic graph (DAG) of a resilient distributed dataset (RDD), and then submitted to a Spark cluster as a task. This way, Hive query efficiency is improved thanks to the distributed memory computing capability of Spark.

5.16.4 Enhanced Open Source Feature

Enhanced Open Source Feature: HDFS Colocation

HDFS Colocation is the data location control function provided by HDFS. The HDFS Colocation API stores associated data or data on which associated operations are performed on the same storage node.

Hive supports HDFS Colocation. When Hive tables are created, after the locator information is set for table files, the data files of related tables are stored on the same storage node. This ensures convenient and efficient data computing among associated tables.

Enhanced Open Source Feature: Column Encryption

Hive supports encryption of one or more columns. The columns to be encrypted and the encryption algorithm can be specified when a Hive table is created. When data is inserted into the table using the INSERT statement, the related columns are encrypted. The Hive column encryption does not support views and the Hive over HBase scenario.

The Hive column encryption mechanism supports two encryption algorithms that can be selected to meet site requirements during table creation:

- AES (the encryption class is **org.apache.hadoop.hive.serde2.AESRewriter**)
- SMS4 (the encryption class is **org.apache.hadoop.hive.serde2.SMS4Rewriter**)

Enhanced Open Source Feature: HBase Deletion

Due to the limitations of underlying storage systems, Hive does not support the ability to delete a single piece of table data. In Hive on HBase, Hive in the MRS solution supports the ability to delete a single piece of HBase table data. Using a specific syntax, Hive can delete one or more pieces of data from an HBase table.

Enhanced Open Source Feature: Row Delimiter

In most cases, a carriage return character is used as the row delimiter in Hive tables stored in text files, that is, the carriage return character is used as the terminator of a row during queries.

However, some data files are delimited by special characters, and not a carriage return character.

MRS Hive allows you to specify different characters or character combinations as row delimiters for Hive data in text files.

Enhanced Open Source Feature: HTTPS/HTTP-based REST API Switchover

WebHCat provides external REST APIs for Hive. By default, the open source community version uses the HTTP protocol.

MRS Hive supports the HTTPS protocol that is more secure, and enables switchover between the HTTP protocol and the HTTPS protocol.

Enhanced Open Source Feature: Transform Function

The Transform function is not allowed by Hive of the open source version. MRS Hive supports the configuration of the Transform function. The function is disabled by default, which is the same as that of the open source community version.

Users can modify configurations of the Transform function to enable the function. However, security risks exist when the Transform function is enabled.

Enhanced Open Source Feature: Temporary Function Creation Without ADMIN Permission

You must have **ADMIN** permission when creating temporary functions on Hive of the open source community version. MRS Hive supports the configuration of the function for creating temporary functions with **ADMIN** permission. The function is disabled by default, which is the same as that of the open-source community version.

You can modify configurations of this function. After the function is enabled, you can create temporary functions without **ADMIN** permission.

Enhanced Open Source Feature: Database Authorization

In the Hive open source community version, only the database owner can create tables in the database. You can be granted with the **CREATE** and **SELECT** permissions on tables by MRS Hive in a database. After you are granted with the permission to query data in the database, the system automatically associates the query permission on all tables in the database.

Enhanced Open Source Feature: Column Authorization

The Hive open source community version supports only table-level permission control. MRS Hive supports column-level permission control. You can be granted with column-level permissions, such as **SELECT**, **INSERT**, and **UPDATE**.

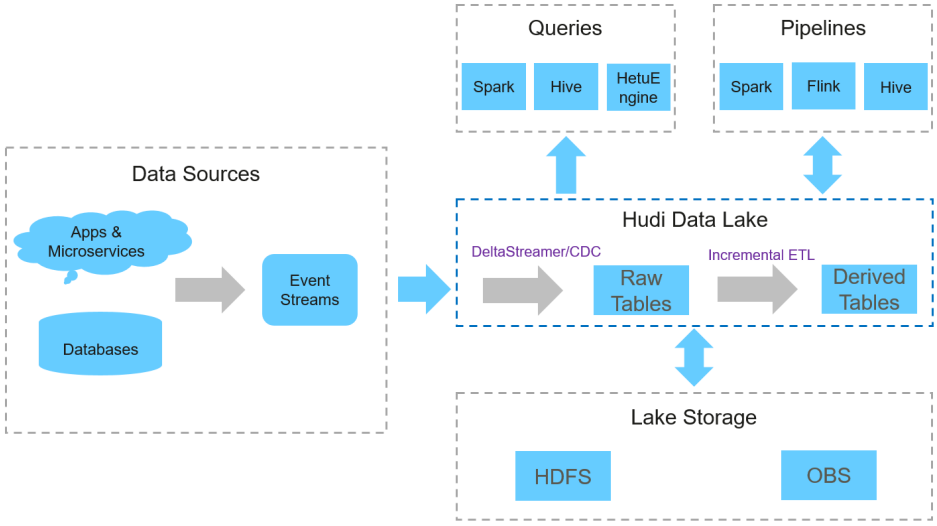
5.17 Apache Hudi

Hudi is a data lake table format that provides the ability to update and delete data as well as consume new data on HDFS. It supports multiple compute engines and provides insert, update, and delete (IUD) interfaces and streaming primitives, including upsert and incremental pull, over datasets on HDFS.

NOTE

To use Hudi, ensure that the Spark service has been installed in the MRS cluster.

Figure 5-85 Basic architecture of Hudi



Features

- The ACID transaction capability supports real-time data import to the lake and batch data import to the data lake.
- Multiple view capabilities (read-optimized view/incremental view/real-time view) enable quick data analysis.
- Multi-version concurrency control (MVCC) design supports data version backtracking.
- Automatic management of file sizes and layouts optimizes query performance and provides quasi-real-time data for queries.
- Concurrent read and write are supported. Data can be read when being written based on snapshot isolation.
- Bootstrapping is supported to convert existing tables into Hudi datasets.

Key Technologies and Advantages

- Pluggable index mechanism: Hudi provides multiple index mechanisms to quickly update and delete massive data.
- Ecosystem support: Hudi supports multiple data engines, including Hive, Spark, HetuEngine, and Flink.

Supported Table Types

- Copy On Write (COW)

  Copy-on-write tables are also called COW tables. Parquet files are used to store data, and internal update operations need to be performed by rewriting the original Parquet files.

  - Advantage: It is efficient because only one data file in the corresponding partition needs to be read.
  - Disadvantage: During data write, a previous copy needs to be copied and then a new data file is generated based on the previous copy. This process is time-consuming. Therefore, the data read by the read request lags behind.

- Merge On Read (MOR)

  Merge-on-read tables are also called MOR tables. The combination of columnar-based Parquet and row-based format Avro is used to store data. Parquet files are used to store base data, and Avro files (also called log files) are used to store incremental data.

  - Advantage: Data is written to the delta log first, and the delta log size is small. Therefore, the write cost is low.
  - Disadvantage: Files need to be compacted periodically. Otherwise, there are a large number of fragment files. The read performance is poor because delta logs and old data files need to be merged.

Three Types of Views to Read Data in Different Scenarios

- Snapshot view

  Provides the latest snapshot data of the current Hudi table. That is, once the latest data is written to the Hudi table, the newly written data can be queried through this view.

  Both COW and MOR tables support this view capability.

- Incremental view

  Provides the incremental query capability. The incremental data after a specified commit can be queried. This view can be used to quickly pull incremental data.

  COW tables support this view capability. MOR tables also support this view capability, but the incremental view capability disappears once the compact operation is performed.

- Read optimized view

  Provides only the data stored in the latest Parquet file.

  This view is different for COW and MOR tables.

  For COW tables, the view capability is the same as the real-time view capability. (COW tables use only Parquet files to store data.)

  For MOR tables, only base files are accessed, and the data in the given file slices since the last compact operation is provided. It can be simply understood that this view provides only the data stored in Parquet files of MOR tables, and the data in log files is ignored. The data provided by this view may not be the latest. However, once the compact operation is performed on MOR tables, the incremental log data is merged into the base data. In this case, this view has the same capability as the real-time view.

5.18 Hue

5.18.1 Hue Basic Principles

Hue is a group of web applications that interact with MRS big data components. It helps you browse HDFS, perform Hive query, and start MapReduce jobs. Hue bears applications that interact with all MRS big data components.

Hue provides the file browser and query editor functions:

- File browser allows you to directly browse and operate different HDFS directories on the GUI.
- Query editor can write simple SQL statements to query data stored on Hadoop, for example, HDFS, HBase, and Hive. With the query editor, you can easily create, manage, and execute SQL statements and download the execution results as an Excel file.

On the WebUI provided by Hue, you can perform the following operations on the components:

- HDFS:
  - View, create, manage, rename, move, and delete files or directories.
  - File upload and download
  - Search for files, directories, file owners, and user groups; change the owners and permissions of the files and directories.
  - Manually configure HDFS directory storage policies and dynamic storage policies.
- Hive:
  - Edit and execute SQL/HQL statements. Save, copy, and edit the SQL/HQL template. Explain SQL/HQL statements. Save the SQL/HQL statement and query it.
  - Database presentation and data table presentation
  - Supporting different types of Hadoop storage
  - Use MetaStore to add, delete, modify, and query databases, tables, and views.

    NOTE

    If Internet Explorer is used to access the Hue page to execute HiveSQL statements, the execution fails, because the browser has functional problems. You are advised to use a compatible browser, for example, Google Chrome.
- MapReduce: Check MapReduce tasks that are being executed or have been finished in the clusters, including their status, start and end time, and run logs.
- Oozie: Hue provides the Oozie job manager function, in this case, you can use Oozie in GUI mode.
- Solr: Hue supports applications searched based on Solr and provides visualized data views.

- ZooKeeper: Hue provides the ZooKeeper browser function for you to use ZooKeeper in GUI mode.

For details about Hue, visit **https://gethue.com/**.

Hue Architecture

Hue, adopting the MTV (Model-Template-View) design, is a web application program running on Django Python. (Django Python is a web application framework that uses open source codes.)

Hue consists of Supervisor Process and WebServer. Supervisor Process is the core Hue process that manages application processes. Supervisor Process and WebServer interact with applications on WebServer through Thrift/REST APIs, as shown in **Figure 5-86**.

Figure 5-86 Hue architecture

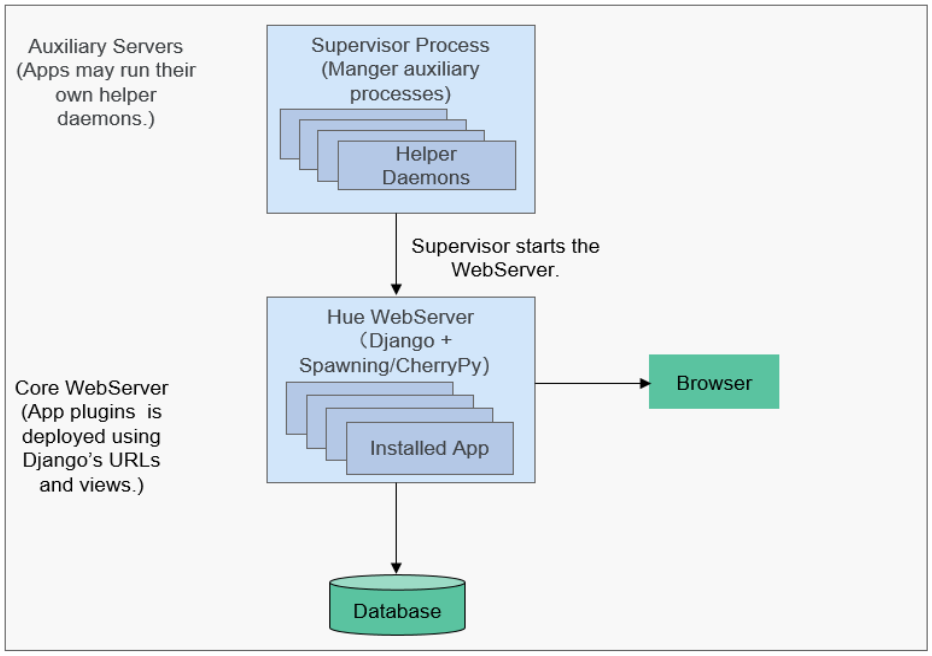


Table 5-18 describes the components shown in **Figure 5-86**.

Table 5-18 Architecture description

Connection Name	Description
Supervisor Process	Manages processes of WebServer applications, such as starting, stopping, and monitoring the processes.

Connection Name	Description
Hue WebServer	Provides the following functions through the Django Python web framework: • Deploys applications. • Provides the GUI. • Connects to databases to store persistent data of applications.

5.18.2 Relationship Between Hue and Other Components

Relationship Between Hue and Hadoop Clusters

Figure 5-87 shows how Hue interacts with Hadoop clusters.

Figure 5-87 Hue and Hadoop clusters

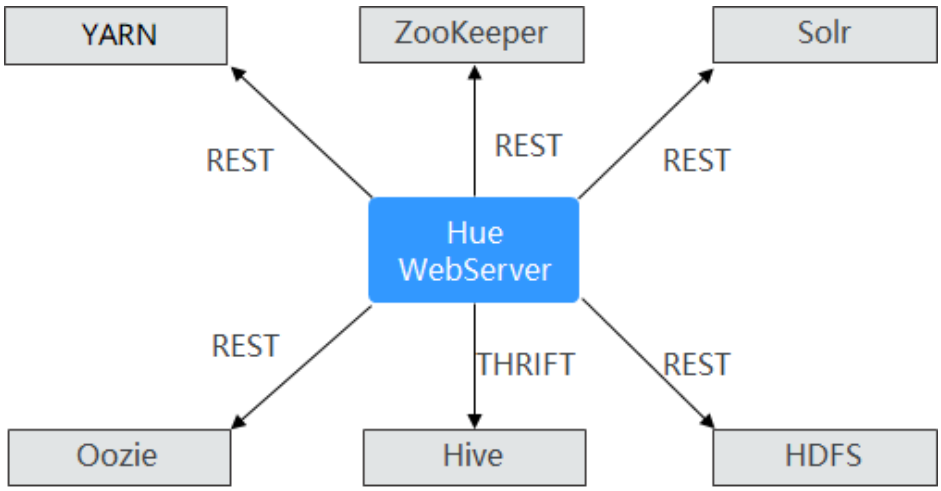


Table 5-19 Relationship Between Hue and Other Components

Connection Name	Description
HDFS	HDFS provides REST APIs to interact with Hue to query and operate HDFS files. Hue packages a user request into interface data, sends the request to HDFS through REST APIs, and displays execution results on the web UI.

Connection Name	Description
Hive	Hive provides Thrift interfaces to interact with Hue, execute Hive SQL statements, and query table metadata. If you edit HQL statements on the Hue web UI, then, Hue submits the HQL statements to the Hive server through the Thrift APIs and displays execution results on the web UI.
YARN/ MapReduce	MapReduce provides REST APIs to interact with Hue and query YARN job information. If you go to the Hue web UI, enter the filter parameters, the UI sends the parameters to the background, and Hue invokes the REST APIs provided by MapReduce (MR1/MR2-YARN) to obtain information such as the status of the task running, the start/end time, the run log, and more.
Oozie	Oozie provides REST APIs to interact with Hue, create workflows, coordinators, and bundles, and manage and monitor tasks. A graphical workflow, coordinator, and bundle editor are provided on the Hue web UI. Hue invokes the REST APIs of Oozie to create, modify, delete, submit, and monitor workflows, coordinators, and bundles.
Solr	Solr provides REST APIs to interact with Hue, define indexes, and search information. In the Hue web UI, screening parameters are set using GUI controls. The parameter settings are sent to the Hue server. The Hue server invokes the REST APIs of Solr and transmits the results returned by Solr in JSON format to the Hue web UI. The Hue web UI then displays the results using icons and controls.
ZooKeeper	ZooKeeper provides REST APIs to interact with Hue and query ZooKeeper node information. ZooKeeper node information is displayed in the Hue web UI. Hue invokes the REST APIs of ZooKeeper to obtain the node information.

5.18.3 Hue Enhanced Open Source Features

Hue Enhanced Open Source Features

- Storage policy: The number of HDFS file copies varies depending on the storage media. This feature allows you to manually set an HDFS directory storage policy or can automatically adjust the file storage policy, modify the number of file copies, move the file directory, and delete files based on the latest access time and modification time of HDFS files to fully utilize storage capacity and improve storage performance.

- MR engine: You can use the MapReduce engine to execute Hive SQL statements.
- Reliability enhancement: Hue is deployed in active/standby mode. When interconnecting with HDFS, Oozie, Hive, Solr, and YARN, Hue can work in failover or load balancing mode.

5.19 IoTDB

5.19.1 IoTDB Basic Principles

Database for Internet of Things (IoTDB) is a software system that collects, stores, manages, and analyzes IoT time series data. Apache IoTDB uses a lightweight architecture and features high performance and rich functions.

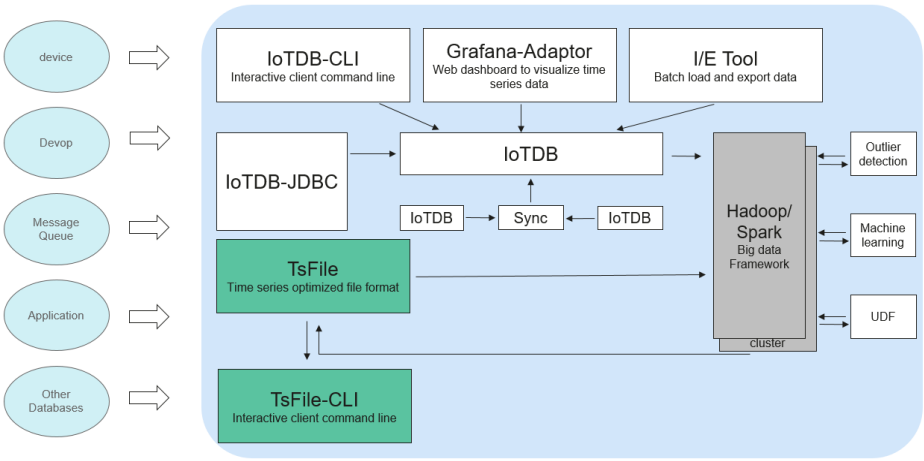
IoTDB sorts time series and stores indexes and chunks, greatly improving the query performance of time series data. IoTDB uses the Raft protocol to ensure data consistency. In time series scenarios, IoTDB pre-computes and stores data to improve analysis performance. Based on the characteristics of time series data, IoTDB provides powerful data encoding and compression capabilities. In addition, its replica mechanism ensures data security. IoTDB is deeply integrated with Apache Hadoop and Flink to meet the requirements of massive data storage, high-speed data reading, and complex data analysis in the industrial IoT field.

IoTDB Architecture

The IoTDB suite consists of multiple components to provide a series of functions such as data collection, data writing, data storage, data query, data visualization, and data analysis.

Figure 5-88 shows the overall application architecture after all components of the IoTDB suite are used. IoTDB refers to the time series database component in the suite.

Figure 5-88 IoTDB architecture



- Users can use Java Database Connectivity (JDBC) or Session to import the time series data and system status data (such as server load, CPU usage and

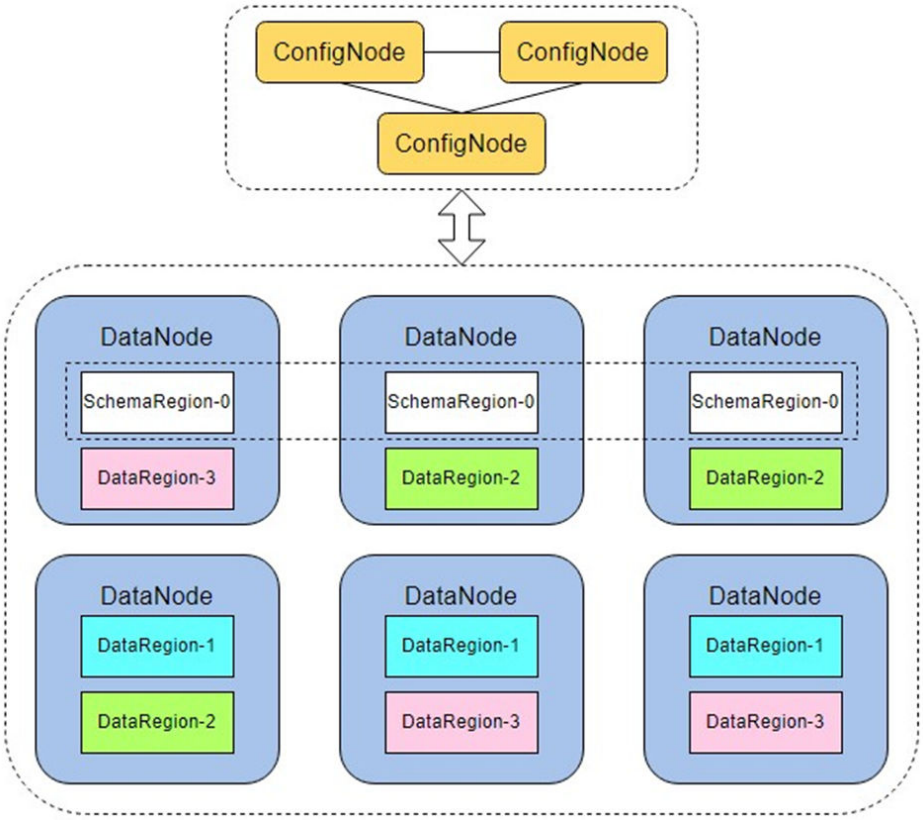
memory usage) collected from device sensors, as well as time series data in message queues, applications, or other databases, to the local or remote IoTDB. Users can also directly write the preceding data into a local TsFile file or a TsFile file in the HDFS.

- Users can write TsFile files to the HDFS to implement data processing tasks such as exception detection and machine learning on the Hadoop or Flink data processing platform.
- The TsFile-Hadoop or TsFile-Flink connector can be used to allow Hadoop or Flink to process the TsFile files written to the HDFS or local host.
- The analysis result can be written back to a TsFile in the same way.
- IoTDB and TsFile also provide client tools to meet users' requirements for viewing and writing data in SQL, script, and graphical formats.

The IoTDB service includes two roles: IoTDBServer (DataNode) and ConfigNode. The role name DataNode of the community edition has the same name as the HDFS role. DataNode is renamed IoTDBServer.

- ConfigNode: management role, which is responsible for DataNode data sharding and load balancing.
- IoTDBServer (DataNode): storage role, which is responsible for storing, querying, and writing data.

Figure 5-89 IoTDB distributed architecture

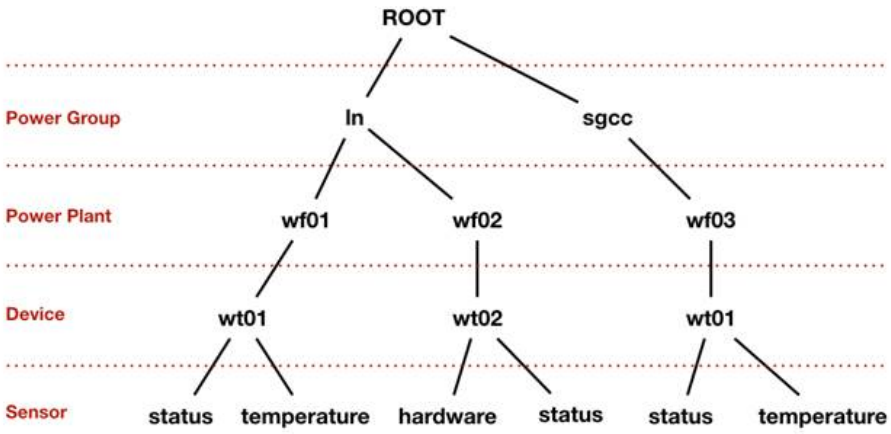


IoTDB Principles

Based on the attribute hierarchy, attribute coverage, and subordinate relationships between data, the IoTDB data model can be represented as the attribute

hierarchy, as shown in **Figure 5-90**. The hierarchy is as follows: power group layer - power plant layer - device layer - sensor layer. **ROOT** is a root node, and each node at the sensor layer is a leaf node. According to the IoTDB syntax, the path from **ROOT** to a leaf node is separated by a dot (.). The complete path is used to name a time series in the IoTDB. For example, the time series name corresponding to the path on the left in the following figure is **ROOT.ln.wf01.wt01.status**.

Figure 5-90 IoTDB data model



5.19.2 Relationship Between IoTDB and Other Components

The IoTDB stores data locally, so it does not depend on any other component for storage. However, in a security cluster environment, IoTDB depends on the KrbServer component for Kerberos authentication.

5.19.3 IoTDB Enhanced Open Source Features

Visualization

- Visualized O&M covers installation, uninstallation, one-click start and stop, configurations, clients, monitoring, alarms, health checks, and logs.
- Visualized permission management does not require background command line operations and supports read and write permission control at the database and table levels.
- Visualized log level configuration dynamically takes effect, supports visualized download and retrieval, and supports log audit.

Security Hardening

User authentication supports Kerberos authentication and SSL encryption, which are compatible with the community authentication mode.

Ecosystem Interconnection

On the basis of native capabilities, the cluster interconnection with MQTT is enhanced.

Enterprise-Level Features

In addition to native capabilities, disk hot swap, backup, and restoration capabilities are enhanced.

Lakehouse

Supports cross-source federation. HetuEngine can be used with HBase and Hive for converged analysis and query, eliminating the need for data transfer.

5.20 JobGateway

5.20.1 JobGateway Basic Principles

JobGateway allows you to submit jobs through REST APIs.

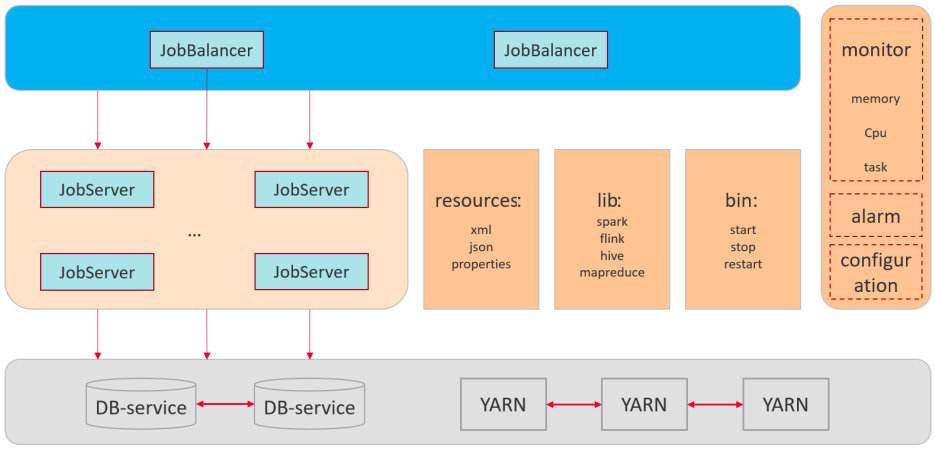
As a gateway component for submitting big data jobs, JobGateway provides fully controllable enterprise-level big data job submission services, such as Spark, HBase, Flink, and Hive.

JobGateway Architecture

JobGateway consists of JobServer and JobBalancer instances.

- JobBalancer provides load balancing.
- JobServer provides REST APIs for submitting jobs.

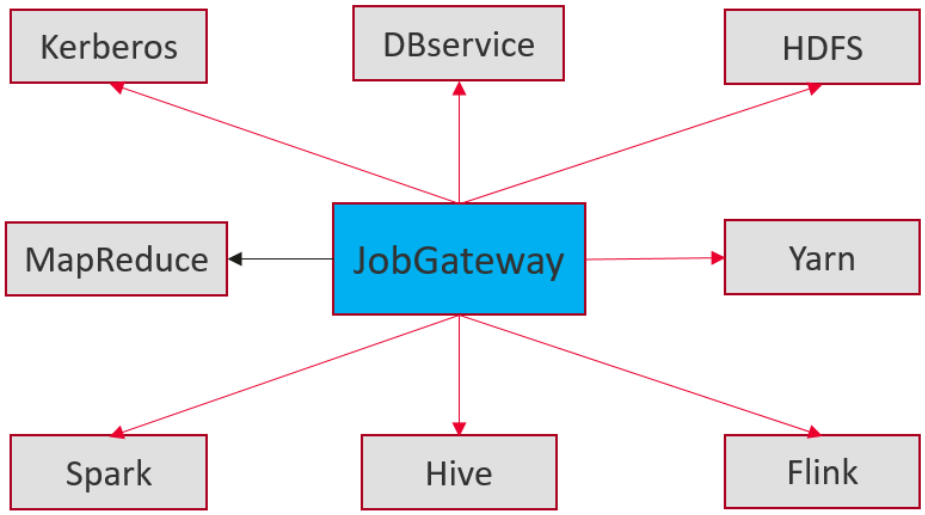
Figure 5-91 JobGateway architecture



5.20.2 Relationships Between JobGateway and Other Components

JobGateway is a service that allows you to submit Spark, Hive, MapReduce, and Flink jobs through REST APIs.

Figure 5-92 Relationships between JobGateway and other components



5.21 Apache Kafka

5.21.1 Kafka Basic Principles

Kafka is an open source, distributed, partitioned, and replicated commit log service. Kafka is publish-subscribe messaging, rethought as a distributed commit log. It provides features similar to Java Message Service (JMS) but another design. It features message endurance, high throughput, distributed methods, multi-client support, and real time. It applies to both online and offline message consumption, such as regular message collection, website activeness tracking, aggregation of statistical system operation data (monitoring data), and log collection. These scenarios engage large amounts of data collection for Internet services.

Kafka Architecture

Producers publish data to topics, and consumers subscribe to the topics and consume messages. A broker is a server in a Kafka cluster. For each topic, the Kafka cluster maintains partitions for scalability, parallelism, and fault tolerance. Each partition is an ordered, immutable sequence of messages that is continually appended to - a commit log. Each message in a partition is assigned a sequential ID, which is called offset.

Figure 5-93 Kafka architecture

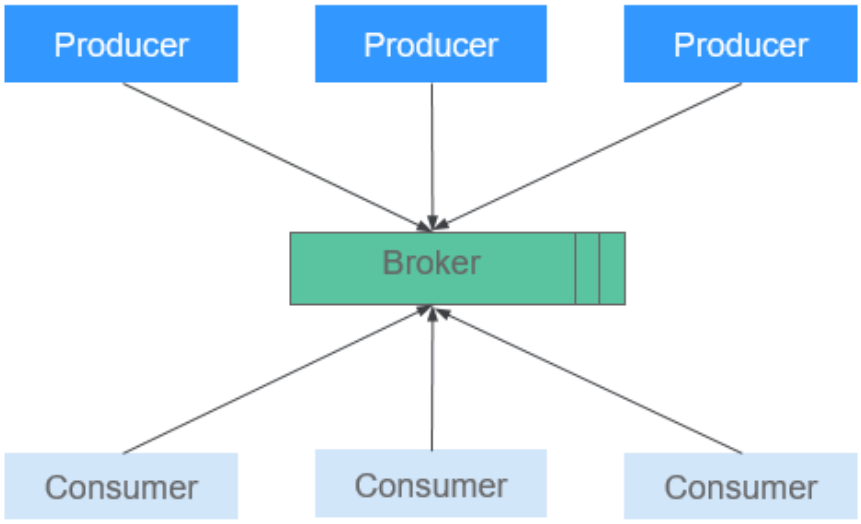
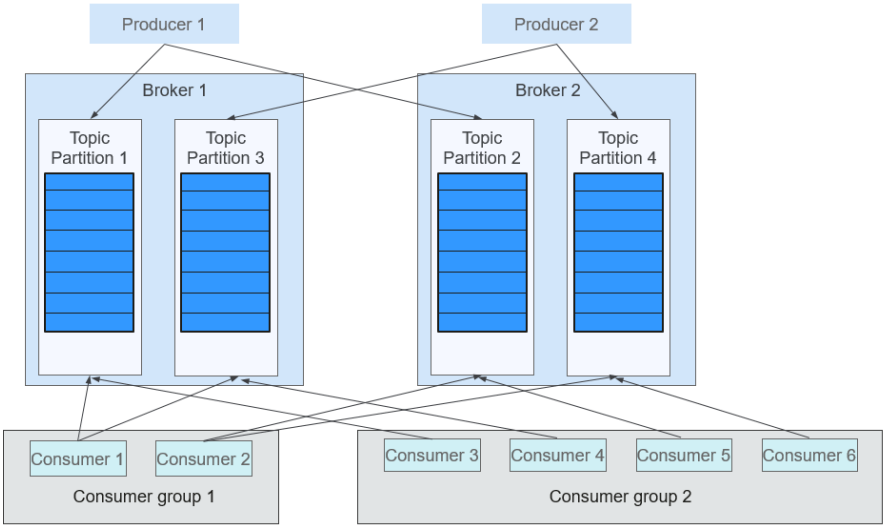


Table 5-20 Kafka architecture description

Name	Description
Broker	A broker is a server in a Kafka cluster.
Topic	A topic is a category or feed name to which messages are published. A topic can be divided into multiple partitions, which can act as a parallel unit.
Partition	A partition is an ordered, immutable sequence of messages that is continually appended to - a commit log. The messages in the partitions are each assigned a sequential ID number called the offset that uniquely identifies each message within the partition.
Producer	Producers publish messages to a Kafka topic.
Consumer	Consumers subscribe to topics and process the feed of published messages.

Figure 5-94 shows the relationships between modules.

Figure 5-94 Relationships between Kafka modules



Consumers label themselves with a consumer group name, and each message published to a topic is delivered to one consumer instance within each subscribing consumer group. If all the consumer instances belong to the same consumer group, loads are evenly distributed among the consumers. As shown in the preceding figure, Consumer1 and Consumer2 work in load-sharing mode; Consumer3, Consumer4, Consumer5, and Consumer6 work in load-sharing mode. If all the consumer instances belong to different consumer groups, messages are broadcast to all consumers. As shown in the preceding figure, the messages in Topic 1 are broadcast to all consumers in Consumer Group1 and Consumer Group2.

For details about Kafka architecture and principles, see **https://kafka.apache.org/24/documentation.html**.

Kafka Principles

- **Message Reliability**

  When a Kafka broker receives a message, it stores the message on a disk persistently. Each partition of a topic has multiple replicas stored on different broker nodes. If one node is faulty, the replicas on other nodes can be used.

- **High Throughput**

  Kafka provides high throughput in the following ways:

  - Messages are written into disks instead of being cached in the memory, fully utilizing the sequential read and write performance of disks.
  - The use of zero-copy eliminates I/O operations.
  - Data is sent in batches, improving network utilization.
  - Each topic is divided in to multiple partitions, which increases concurrent processing. Concurrent read and write operations can be performed between multiple producers and consumers. Producers send messages to specified partitions based on the algorithm used.

- **Message Subscribe-Notify Mechanism**

Consumers subscribe to interested topics and consume data in pull mode. Consumers can choose the consumption mode, such as batch consumption, repeated consumption, and consumption from the end, and control the message pulling speed based on actual situation. Consumers need to maintain the consumption records by themselves.

- **Scalability**

  When broker nodes are added to expand the Kafka cluster capacity, the newly added brokers register with ZooKeeper. After the registration is successful, procedures and consumers can sense the change in a timely manner and make related adjustment.

Open Source Features

- Reliability

  Message processing methods such as **At-Least Once**, **At-Most Once**, and **Exactly Once** are provided. The message processing status is maintained by consumers. Kafka needs to work with the application layer to implement **Exactly Once**.

- High throughput

  High throughput is provided for message publishing and subscription.

- Persistence

  Messages are stored on disks and can be used for batch consumption and real-time application programs. Data persistence and replication prevent data loss.

- Distribution

  A distributed system is easy to be expanded externally. All producers, brokers, and consumers support the deployment of multiple distributed clusters. Systems can be scaled without stopping the running of software or shutting down the machines.

Kafka UI

Kafka UI provides Kafka web services, displays basic information about functional modules such as brokers, topics, partitions, and consumers in a Kafka cluster, and provides operation entries for common Kafka commands. Kafka UI replaces Kafka Manager to provide secure Kafka web services that comply with security specifications.

You can perform the following operations on Kafka UI:

- Check cluster status (topics, consumers, offsets, partitions, replicas, and nodes).
- Redistribute partitions in the cluster.
- Create a topic with optional topic configurations.
- Delete a topic (supported when **delete.topic.enable** is set to **true** for the Kafka service).
- Add partitions to an existing topic.
- Update configurations for an existing topic.
- Optionally enable JMX polling for broker-level and topic-level metrics.

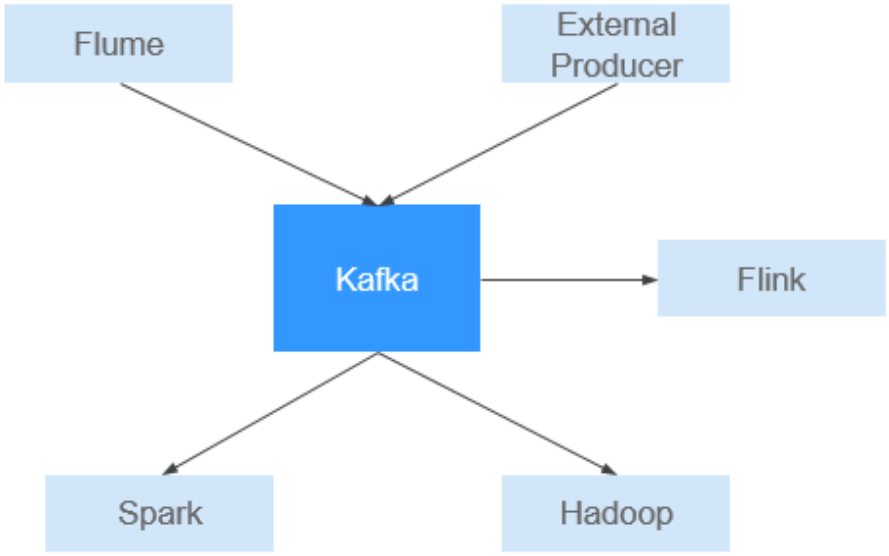
MirrorMaker

MirrorMaker is a tool for implementing data synchronization between active and standby Kafka clusters. It consumes data from the active Kafka cluster and backs up the data to the standby cluster so that a data replica of the active Kafka cluster can be generated.

5.21.2 Relationships Between Kafka and Other Components

As a message publishing and subscription system, Kafka provides high-speed data transmission methods for data transmission between different subsystems of the FusionInsight platform. It can receive external messages in a real-time manner and provides the messages to the online and offline services for processing. The following figure shows the relationship between Kafka and other components.

Figure 5-95 Relationships with other components



5.21.3 Kafka Enhanced Open Source Features

Kafka Enhanced Open Source Features

- Monitors the following topic-level metrics:
  - Topic Input Traffic
  - Topic Output Traffic
  - Topic Rejected Traffic
  - Number of Failed Fetch Requests Per Second
  - Number of Failed Produce Requests Per Second
  - Number of Topic Input Messages Per Second
  - Number of Fetch Requests Per Second
  - Number of Produce Requests Per Second

- Queries the mapping between broker IDs and node IP addresses. On Linux clients, **kafka-broker-info.sh** can be used to query the mapping between broker IDs and node IP addresses.

5.22 KMS

5.22.1 KMS Basic Principles

KMS Basic Principles

Hadoop Key Management Server (KMS) is developed based on KeyProvider API. It provides a client and a server that communicate with each other using REST APIs based on HTTP.

The client is the implementation of KeyProvider and interacts with KMS using KMS HTTP REST API. KMS and its client are configured with built-in security mechanisms that support HTTP SPNEGO Kerberos authentication and HTTPS-based secure transmission.

HDFS supports end-to-end transparent encryption. After the configuration is complete, users do not need to modify any application code when storing data to HDFS. Data encryption and decryption are performed by the client. The HDFS does not store or access unencrypted data or data encryption keys.

NOTE

KMS is supported only by MRS physical machine clusters.

5.22.2 Relationship Between KMS and Other Components

Relationship Between KMS and HDFS

When HDFS is interconnected with KMS, keys are obtained from the KMS during encryption. When an HDFS encrypted area is created, the NameNode obtains the value from the KMS.

Relationship Between ZooKeeper and KMS

Multiple instances of KMS share the token information, and the token information is stored in ZooKeeper.

5.23 KrbServer and LdapServer

5.23.1 KrbServer and LdapServer Principles

Overview

To manage the access control permissions on data and resources in a cluster, it is recommended that the cluster be installed in security mode. In security mode, a

client application must be authenticated and a secure session must be established before the application accesses any resource in the cluster. MRS uses KrbServer to provide Kerberos authentication for all components, implementing a reliable authentication mechanism.

LdapServer supports Lightweight Directory Access Protocol (LDAP) and provides the capability of storing user and user group data for Kerberos authentication.

Architecture

The security authentication function for user login depends on Kerberos and LDAP.

Figure 5-96 Security authentication architecture

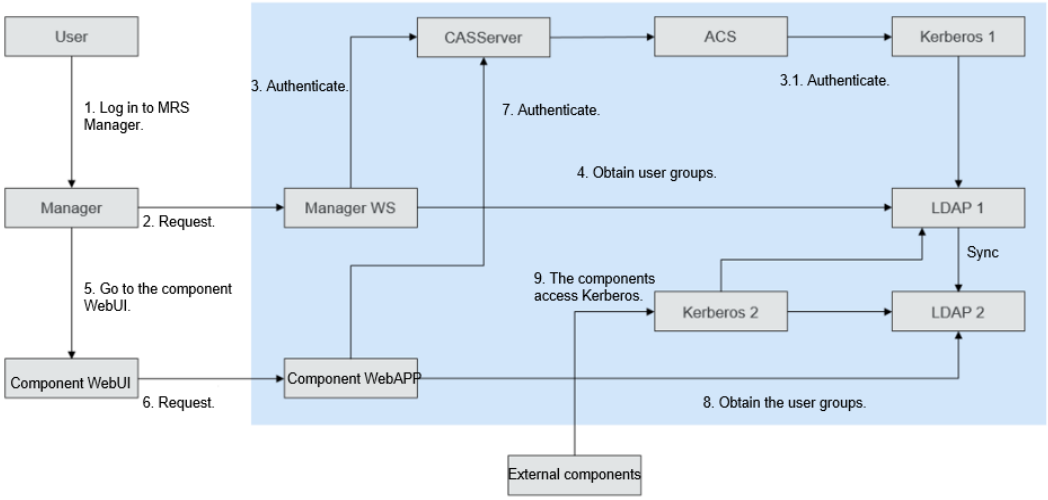


Figure 5-96 includes three scenarios:

- Logging in to the MRS Manager Web UI

  The authentication architecture includes steps 1, 2, 3, and 4.
- Logging in to a component web UI

  The authentication architecture includes steps 5, 6, 7, and 8.
- Accessing between components

  The authentication architecture includes step 9.

Table 5-21 Key modules

Connection Name	Description
Manager	Cluster Manager
Manager WS	WebBrowser
Kerberos1	KrbServer (management plane) service deployed in MRS Manager, that is, OMS Kerberos
Kerberos2	KrbServer (service plane) service deployed in the cluster

Connection Name	Description
LDAP1	LdapServer (management plane) service deployed in MRS Manager, that is, OMS LDAP
LDAP2	LdapServer (service plane) service deployed in the cluster

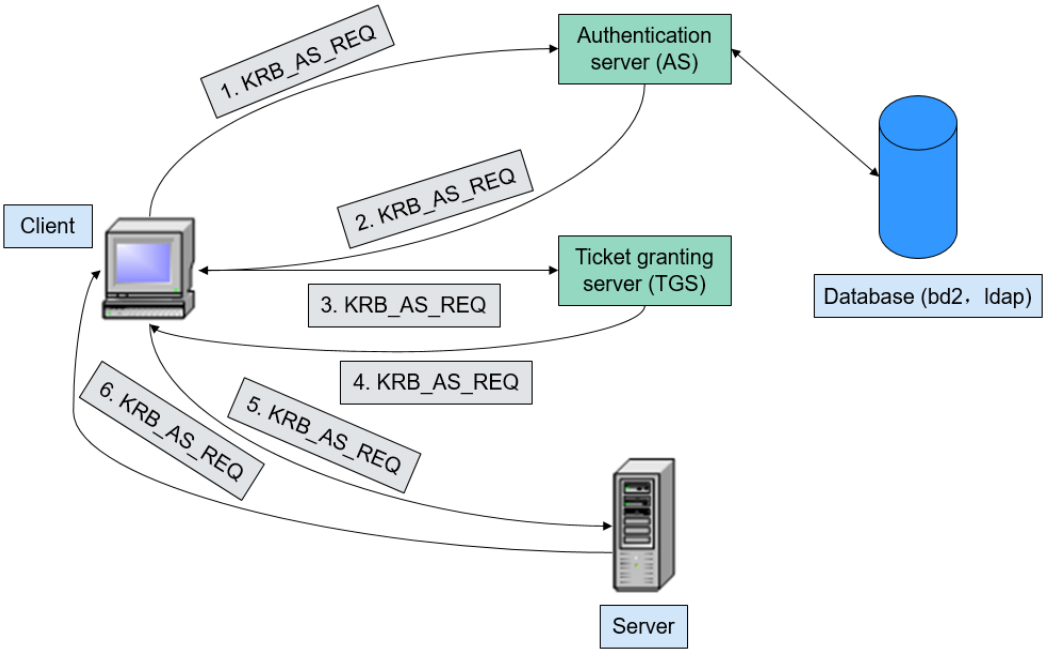
Data operation mode of Kerberos1 in LDAP: The active and standby instances of LDAP1 and the two standby instances of LDAP2 can be accessed in load balancing mode. Data write operations can be performed only in the active LDAP1 instance. Data read operations can be performed in LDAP1 or LDAP2.

Data operation mode of Kerberos2 in LDAP: Data read operations can be performed in LDAP1 and LDAP2. Data write operations can be performed only in the active LDAP1 instance.

Principle

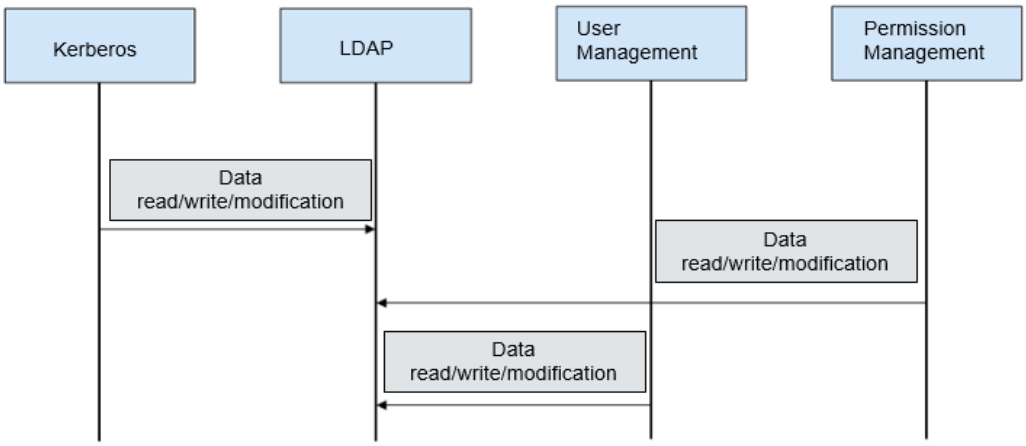
Kerberos authentication

Figure 5-97 Authentication process



LDAP data read and write

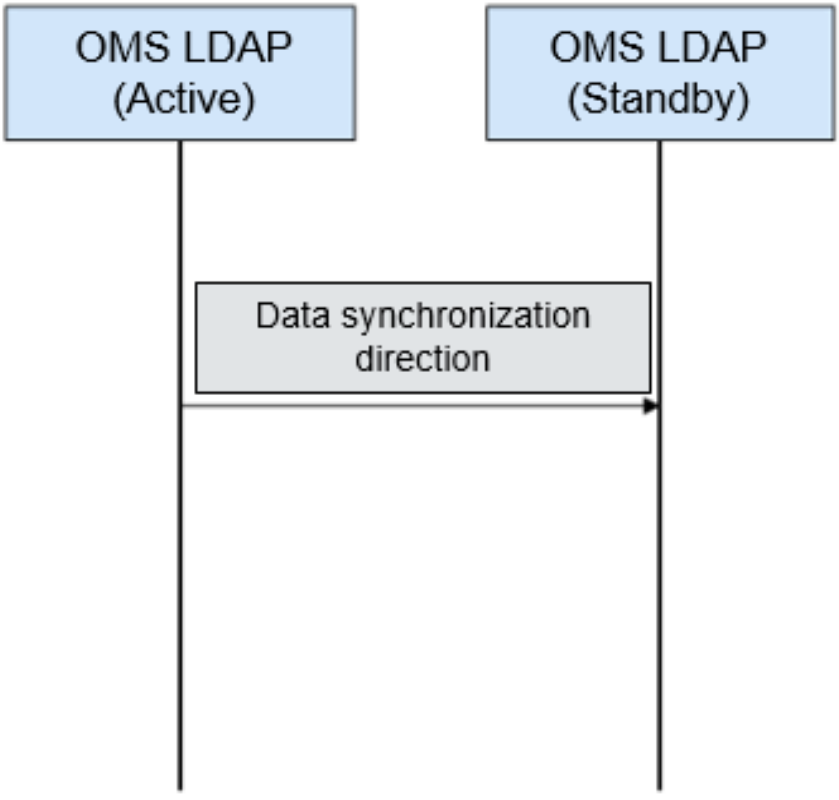
Figure 5-98 Data modification process



LDAP data synchronization

- OMS LDAP data synchronization before cluster installation

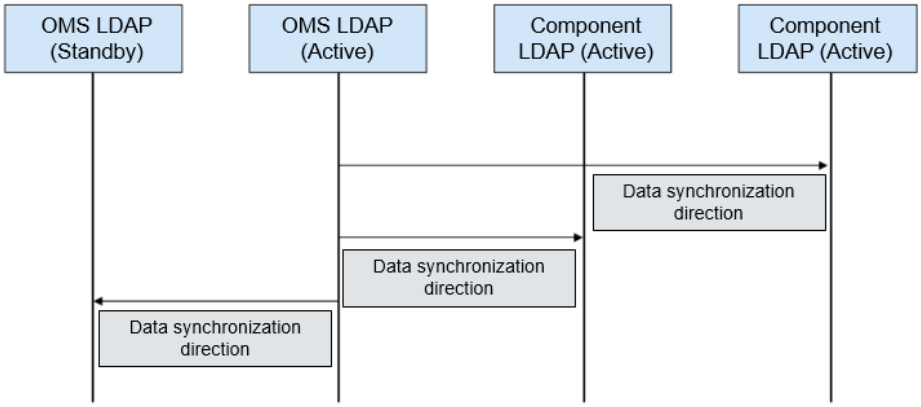
  Figure 5-99 OMS LDAP data synchronization

  Data synchronization direction before cluster installation: Data is synchronized from the active OMS LDAP to the standby OMS LDAP.

- LDAP data synchronization after cluster installation

Figure 5-100 LDAP data synchronization



Data synchronization direction after cluster installation: Data is synchronized from the active OMS LDAP to the standby OMS LDAP, standby component LDAP, and standby component LDAP.

5.23.2 KrbServer and LdapServer Enhanced Open Source Features

Enhanced open-source features of KrbServer and LdapServer: intra-cluster service authentication

In an MRS cluster that uses the security mode, mutual access between services is implemented based on the Kerberos security architecture. When a service (such as HDFS) in the cluster is to be started, the corresponding sessionkey (keytab, used for identity authentication of the application) is obtained from Kerberos. If another service (such as YARN) needs to access HDFS and add, delete, modify, or query data in HDFS, the corresponding TGT and ST must be obtained for secure access.

Enhanced Open-Source Features of KrbServer and LdapServer: Application Development Authentication

MRS components provide application development interfaces for customers or upper-layer service product clusters. During application development, a cluster in security mode provides specified application development authentication interfaces to implement application security authentication and access. For example, the UserGroupInformation class provided by the hadoop-common API provides multiple security authentication APIs.

- **setConfiguration()** is used to obtain related configuration and set parameters such as global variables.
- **loginUserFromKeytab():** is used to obtain TGT interfaces.

Enhanced Open-Source Features of KrbServer and LdapServer: Cross-System Mutual Trust

MRS provides the mutual trust function between two Managers to implement data read and write operations between systems.

5.24 LakeSearch

5.24.1 LakeSearch Basic Principles

Introduction to LakeSearch

LakeSearch is a cutting-edge smart search platform that leverages Pangu search and language models to deliver out-of-the-box functionality. It streamlines the process of constructing a knowledge base by enabling users to upload documents, parse and segment text, vectorize the segmented content, and seamlessly import these vectors into a database with a single click. This platform empowers organizations to effortlessly establish bespoke knowledge search solutions tailored to their specific needs.

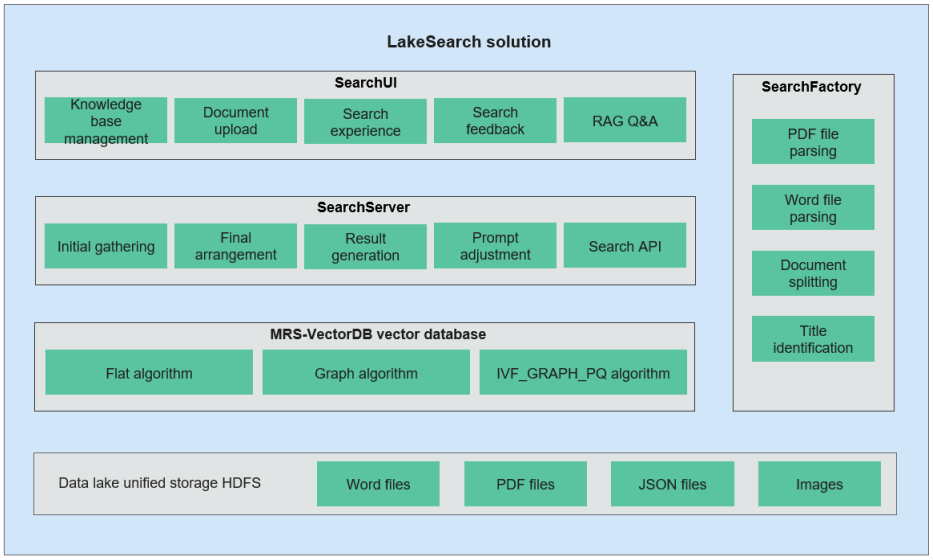
Widely adopted across a diverse range of sectors, LakeSearch is instrumental in enhancing search capabilities for portals, internal corporate knowledge bases, document retrieval, and financial information sharing within the financial sector. It also enhances intelligent customer support search functions, provides guidance for carrier service handling, and improves service knowledge and troubleshooting searches.

LakeSearch Architecture

LakeSearch consists of SearchServer, SearchFactory, and the frontend UI.

- SearchServer provides capabilities such as document upload, vector import, semantic search, and RAG Q&A.
- SearchFactory parses and splits documents.
  - The format, position coordinates, and table structure of PDF, DOC, and DOCX documents will be parsed.
  - A document can be split into multiple parts.

Figure 5-101 LakeSearch architecture



5.24.2 Relationship with Other Components

LakeSearch and Elasticsearch Vector Databases

In LakeSearch, text undergoes transformation into vectors through the application of large-scale models. These vectors are then cataloged within an Elasticsearch vector database, facilitating retrieval.

Relationship Between LakeSearch and HDFS

LakeSearch stores documents in HDFS and supports document upload, download, and deletion.

Relationship Between LakeSearch and DBService

LakeSearch stores metadata in DBService.

5.25 Loader

5.25.1 Loader Basic Principles

Loader is developed based on the open source Sqoop component. It is used to exchange data and files between MRS and relational databases and file systems. Loader can import data from relational databases or file servers to the HDFS and HBase components, or export data from HDFS and HBase to relational databases or file servers.

A Loader model consists of Loader Client and Loader Server, as shown in **Figure 5-102**.

Figure 5-102 Loader model

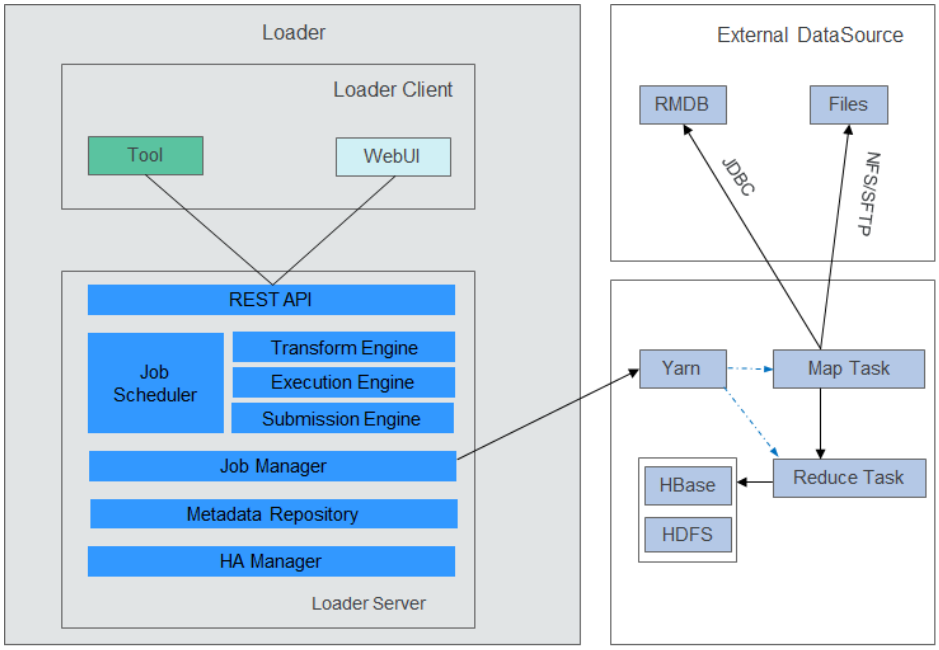


Table 5-22 describes the functions of each module shown in the preceding figure.

Table 5-22 Components of the Loader model

Module	Description
Loader Client	Loader client. It provides two interfaces: web UI and CLI.
Loader Server	Loader server. It processes operation requests sent from the client, manages connectors and metadata, submits MapReduce jobs, and monitors MapReduce job status.
REST API	It provides a Representational State Transfer (RESTful) APIs (HTTP + JSON) to process the operation requests sent from the client.
Job Scheduler	Simple job scheduler. It periodically executes Loader jobs.
Transform Engine	Data transformation engine. It supports field combination, string cutting, and string reverse.
Execution Engine	Loader job execution engine. It executes Loader jobs in MapReduce manner.
Submission Engine	Loader job submission engine. It submits Loader jobs to MapReduce.
Job Manager	It manages Loader jobs, including creating, querying, updating, deleting, activating, deactivating, starting, and stopping jobs.
Metadata Repository	Metadata repository. It stores and manages data about Loader connectors, transformation procedures, and jobs.
HA Manager	It manages the active/standby status of Loader Server processes. The Loader Server has two nodes that are deployed in active/standby mode.

Loader imports or exports jobs in parallel using MapReduce jobs. Some job import or export may involve only the Map operations, while some may involve both Map and Reduce operations.

Loader implements fault tolerance using MapReduce. Jobs can be rescheduled upon a job execution failure.

- **Importing data to HBase**

  When the Map operation is performed for MapReduce jobs, Loader obtains data from an external data source.

  When a Reduce operation is performed for a MapReduce job, Loader enables the same number of Reduce tasks based on the number of Regions. The Reduce tasks receive data from Map tasks, generate HFiles by Region, and store the HFiles in a temporary directory of HDFS.

When a MapReduce job is submitted, Loader migrates HFiles from the temporary directory to the HBase directory.

- **Importing Data to HDFS**

  When a Map operation is performed for a MapReduce job, Loader obtains data from an external data source and exports the data to a temporary directory (named *export directory*-**ldtmp**).

  When a MapReduce job is submitted, Loader migrates data from the temporary directory to the output directory.

- **Exporting data to a relational database**

  When a Map operation is performed for a MapReduce job, Loader obtains data from HDFS or HBase and inserts the data to a temporary table (Staging Table) through the Java DataBase Connectivity (JDBC) API.

  When a MapReduce job is submitted, Loader migrates data from the temporary table to a formal table.

- **Exporting data to a file system**

  When a Map operation is performed for a MapReduce job, Loader obtains data from HDFS or HBase and writes the data to a temporary directory of the file server.

  When a MapReduce job is submitted, Loader migrates data from the temporary directory to a formal directory.

For details about the Loader architecture and principles, see **https://sqoop.apache.org/docs/1.99.3/index.html**.

5.25.2 Relationship Between Loader and Other Components

The components that interact with Loader include HDFS, HBase, MapReduce, and ZooKeeper. Loader works as a client to use certain functions of these components, such as storing data to HDFS and HBase and reading data from HDFS and HBase tables. In addition, Loader functions as an MapReduce client to import or export data.

5.25.3 Loader Enhanced Open Source Features

Loader Enhanced Open-Source Feature: Data Import and Export

Loader is developed based on Sqoop. In addition to the Sqoop functions, Loader has the following enhanced features:

- Provides data conversion functions.
- Supports GUI-based configuration conversion.
- Imports data from an SFTP/FTP server to HDFS/OBS.
- Imports data from an SFTP/FTP server to an HBase table.
- Imports data from an SFTP/FTP server to a Phoenix table.
- Imports data from an SFTP/FTP server to a Hive table.
- Exports data from HDFS/OBS to an SFTP server.
- Exports data from an HBase table to an SFTP server.
- Exports data from a Phoenix table to an SFTP server.

- Imports data from a relational database to an HBase table.
- Imports data from a relational database to a Phoenix table.
- Imports data from a relational database to a Hive table.
- Exports data from an HBase table to a relational database.
- Exports data from a Phoenix table to a relational database.
- Imports data from an Oracle partitioned table to HDFS/OBS.
- Imports data from an Oracle partitioned table to an HBase table.
- Imports data from an Oracle partitioned table to a Phoenix table.
- Imports data from an Oracle partitioned table to a Hive table.
- Exports data from HDFS/OBS to an Oracle partitioned table.
- Exports data from HBase to an Oracle partitioned table.
- Exports data from a Phoenix table to an Oracle partitioned table.
- Imports data from HDFS to an HBase table, a Phoenix table, and a Hive table in the same cluster.
- Exports data from an HBase table and a Phoenix table to HDFS/OBS in the same cluster.
- Imports data to an HBase table and a Phoenix table by using **bulkload** or **put list**.
- Imports all types of files from an SFTP/FTP server to HDFS. The open source component Sqoop can import only text files.
- Exports all types of files from HDFS/OBS to an SFTP server. The open source component Sqoop can export only text files and SequenceFile files.
- Supports file coding format conversion during file import and export. The supported coding formats include all formats supported by Java Development Kit (JDK).
- Retains the original directory structure and file names during file import and export.
- Supports file combination during file import and export. For example, if a large number of files are to be imported, these files can be combined into *n* files (*n* can be configured).
- Supports file filtering during file import and export. The filtering rules support wildcards and regular expressions.
- Supports batch import and export of ETL tasks.
- Supports query by page and key word and group management of ETL tasks.
- Provides floating IP addresses for external components.

5.26 Manager

5.26.1 Manager Basic Principles

Overview

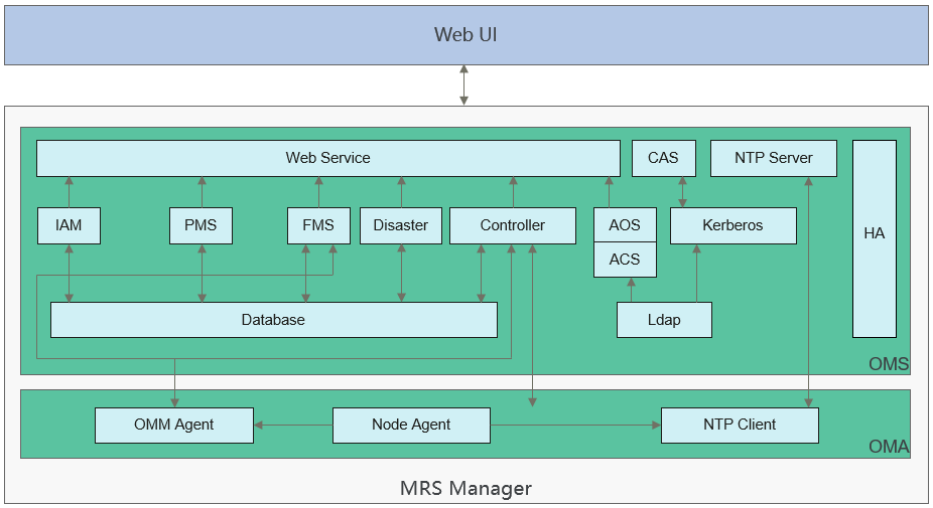
Manager is the O&M management system of MRS and provides unified cluster management capabilities for services deployed in clusters.

Manager provides functions such as installation and deployment, performance monitoring, alarms, user management, permission management, auditing, service management, health check, update, and log collection.

Architecture

Figure 5-103 shows the overall logical architecture of FusionInsight Manager.

Figure 5-103 Manager logical architecture



Manager consists of OMS and OMA.

- OMS: serves as management node in the O&M system. There are two OMS nodes deployed in active/standby mode.
- OMA: managed node in the O&M system. Generally, there are multiple OMA nodes.

Table 5-23 describes the modules shown in **Figure 5-103**.

Table 5-23 Service module description

Module	Description
Web Service	A web service deployed under Tomcat, providing HTTPS API of Manager. It is used to access Manager through the web browser. In addition, it provides the northbound access capability based on the Syslog and SNMP protocols.
OMS	Management node of the O&M system. Generally, there are two OMS nodes that work in active/standby mode.
OMA	Managed node in the O&M system. Generally, there are multiple OMA nodes.

Module	Description
Controller	The control center of Manager. It can converge information from all nodes in the cluster and display it to MRS cluster administrators, as well as receive from MRS cluster administrators, and synchronize information to all nodes in the cluster according to the operation instruction range. Control process of Manager. It implements various management actions: 1. The web service delivers various management actions (such as installation, service startup and stop, and configuration modification) to Controller. 2. Controller decomposes the command and delivers the action to each Node Agent, for example, starting a service involves multiple roles and instances. 3. Controller is responsible for monitoring the implementation of each action.
Node Agent	Node Agent exists on each cluster node and is an enabler of Manager on a single node. ● Node Agent represents all the components deployed on the node to interact with Controller, implementing convergence from multiple nodes of a cluster to a single node. ● Node Agent enables Controller to perform all operations on the components deployed on the node. It allows Controller functions to be implemented. Node Agent sends heartbeat messages to Controller at an interval of 3 seconds. The interval cannot be configured.
IAM	Records audit logs. Each non-query operation on the Manager UI has a related audit log.
PMS	The performance monitoring module. It collects the performance monitoring data on each OMA and provides the query function.
FMS	Alarm module. It collects and queries alarms on each OMA.
Disaster	Module for managing active/standby cluster DR. After DR is configured, data replication between the active and standby clusters is periodically initiated.
OMM Agent	Agent for performance monitoring and alarm reporting on the OMA. It collects performance monitoring data and alarm data on Agent Node.
CAS	Unified authentication center. When a user logs in to the web service, CAS authenticates the login. The browser automatically redirects the user to the CAS through URLs.
AOS	Permission management module. It manages the permissions of users and user groups.

Module	Description
ACS	User and user group management module. It manages users and user groups to which users belong.
Kerberos	LDAP is deployed in OMS and a cluster, respectively. • OMS Kerberos provides the single sign-on (SSO) and authentication between Controller and Node Agent. • Kerberos in the cluster provides the user security authentication function for components. The service name is **KrbServer**, which contains two role instances: – KerberosServer: is an authentication server that provides security authentication for MRS. – KerberosAdmin: manages processes of Kerberos users.
Ldap	LDAP is deployed in OMS and a cluster, respectively. • OMS LDAP provides data storage for user authentication. • The LDAP in the cluster functions as the backup of the OMS LDAP. The service name is **LdapServer** and the role instance is **SlapdServer**.
Database	Manager database used to store logs and alarms.
HA	HA management module that manages the active and standby OMSs.
NTP Server NTP Client	It synchronizes the system clock of each node in the cluster.

5.26.2 Manager Key Features

Key Feature: Unified Alarm Monitoring

Manager provides the visualized and convenient alarm monitoring function. Users can quickly obtain key cluster performance indicators, evaluate cluster health status, customize performance indicator display, and convert indicators to alarms. Manager can monitor the running status of all components and report alarms in real time when faults occur. The online help on the GUI allows you to view performance counters and alarm clearance methods to quickly rectify faults.

Key Feature: Unified User Permission Management

Manager provides permission management of components in a unified manner.

Manager introduces the concept of role and uses role-based access control (RBAC) to manage system permissions. It centrally displays and manages scattered permission functions of each component in the system and organizes the permissions of each component in the form of permission sets (roles) to form a unified system permission concept. By doing so, common users cannot obtain internal permission management details, and permissions become easy for MRS

cluster administrators to manage, greatly facilitating permission management and improving user experience.

Key Feature: SSO

Single sign-on (SSO) is provided between the Manager web UI and component web UI as well as for integration between MRS and third-party systems.

This function centrally manages and authenticates Manager users and component users. The entire system uses LDAP to manage users and uses Kerberos for authentication. A set of Kerberos and LDAP management mechanisms are used between the OMS and components. SSO (including single sign-on and single sign-out) is implemented through CAS. With SSO, users can easily switch tasks between the Manager web UI, component web UIs, and third-party systems, without switching to another user.

NOTE

- To ensure security, the CAS Server can retain a ticket-granting ticket (TGT) used by a user only for 20 minutes.
- If a user does not perform any operation on the page (including on the Manager web UI and component web UIs) within 20 minutes, the page is automatically locked.

Key Feature: Automatic Health Check and Inspection

Manager provides users with automatic inspection on system running environments and helps users check and audit system running health by one click, ensuring correct system running and lowering system operation and maintenance costs. After viewing inspection results, you can export reports for archiving and fault analysis.

Key Feature: Tenant Management

Manager introduces the multi-tenant concept. The CPU, memory, and disk resources of a cluster can be integrated into a set. The set is called a tenant. A mode involving different tenants is called multi-tenant mode.

Manager provides the multi-tenant function, supports a level-based tenant model and allows tenants to be added and deleted dynamically, achieving resource isolation. As a result, it can dynamically manage and configure the computing resources and the storage resources of tenants.

- The computing resources indicate tenants' Yarn task queue resources. The task queue quota can be modified, and the task queue usage status and statistics can be viewed.
- The storage resources can be stored on HDFS. You can add and delete the HDFS storage directories of tenants, and set the quotas of file quantity and the storage space of the directories.

As a unified tenant management platform of MRS, MRS Manager allows users to create and manage tenants in clusters based on service requirements.

- Roles, computing resources, and storage resources are automatically created when tenants are created. By default, all permissions of the new computing resources and storage resources are allocated to a tenant's roles.

- After you have modified the tenant's computing or storage resources, permissions of the tenant's roles are automatically updated.

Manager also provides the multi-instance function so that users can use the HBase, Hive, or Spark alone in the resource control and service isolation scenario. The multi-instance function is disabled by default and can be manually enabled.

Key Feature: Multi-Language Support

Manager supports multiple languages and automatically selects Chinese or English based on the browser language preference. If the browser preferred language is Chinese, Manager displays the portal in Chinese; if the browser preferred language is not Chinese, Manager displays the portal in English. You can also switch between Chinese and English in the lower left corner of the page based on your language preference.

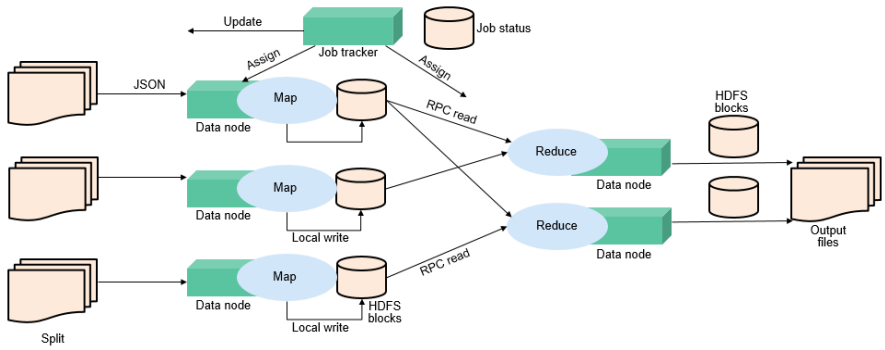
5.27 MapReduce

5.27.1 MapReduce Basic Principles

MapReduce is the core of Hadoop. As a software architecture proposed by Google, MapReduce is used for parallel computing of large-scale datasets (larger than 1 TB). The concepts "Map" and "Reduce" and their main thoughts are borrowed from functional programming language and also borrowed from the features of vector programming language.

Current software implementation is as follows: Specify a Map function to map a series of key-value pairs into a new series of key-value pairs, and specify a Reduce function to ensure that all values in the mapped key-value pairs share the same key.

Figure 5-104 Distributed batch processing engine



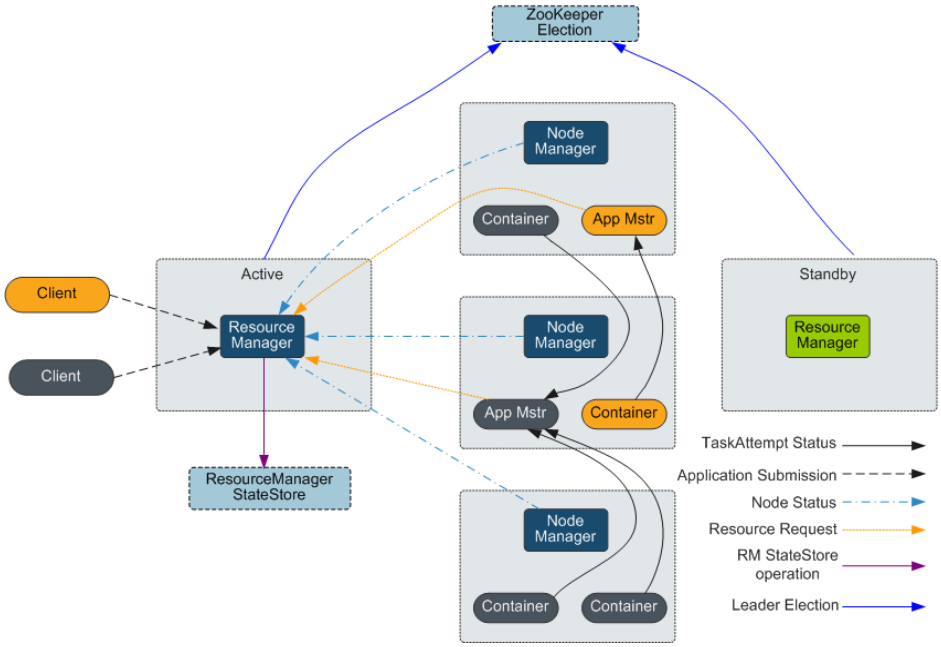
MapReduce is a software framework for processing large datasets in parallel. The root of MapReduce is the Map and Reduce functions in functional programming. The Map function accepts a group of data and transforms it into a key-value pair list. Each element in the input domain corresponds to a key-value pair. The Reduce function accepts the list generated by the Map function, and then shrinks the key-value pair list based on the keys. MapReduce divides a task into multiple parts and allocates them to different devices for processing. In this way, the task can be finished in a distributed environment instead of a single powerful server.

For more information, see **MapReduce Tutorial**.

MapReduce Architecture

As shown in **Figure 5-105**, MapReduce is integrated into YARN through the Client and ApplicationMaster interfaces of YARN, and uses YARN to apply for computing resources.

Figure 5-105 Basic architecture of Apache YARN and MapReduce



5.27.2 Relationship Between MapReduce and Other Components

Relationship Between MapReduce and HDFS

- HDFS features high fault tolerance and high throughput, and can be deployed on low-cost hardware for storing data of applications with massive data sets.
- MapReduce is a programming model used for parallel computation of large data sets (larger than 1 TB). Data computed by MapReduce comes from multiple data sources, such as Local FileSystem, HDFS, and databases. Most data comes from the HDFS. The high throughput of HDFS can be used to read massive data. After being computed, data can be stored in HDFS.

Relationship Between MapReduce and Yarn

MapReduce is a computing framework running on Yarn, which is used for batch processing. MRv1 is implemented based on MapReduce in Hadoop 1.0, which is composed of programming models (new and old programming APIs), running environment (JobTracker and TaskTracker), and data processing engine (MapTask and ReduceTask). This framework is still weak in scalability, fault tolerance (JobTracker SPOF), and compatibility with multiple frameworks. (Currently, only the MapReduce computing framework is supported.) MRv2 is implemented based

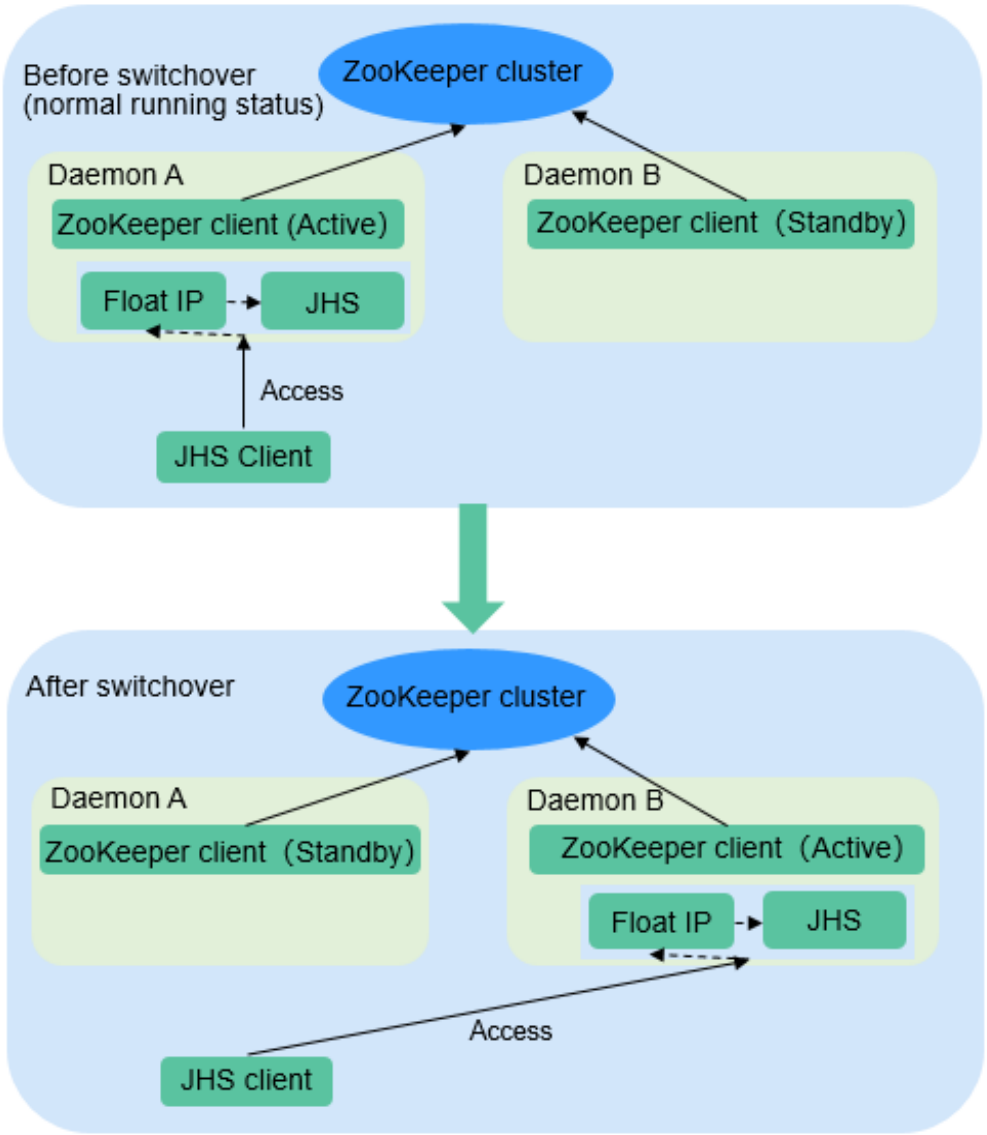
on MapReduce in Hadoop 2.0. The source code reuses MRv1 programming models and data processing engine implementation, and the running environment is composed of ResourceManager and ApplicationMaster. ResourceManager is a brand new resource manager system, and ApplicationMaster is responsible for cutting MapReduce job data, assigning tasks, applying for resources, scheduling tasks, and tolerating faults.

5.27.3 MapReduce Enhanced Open Source Features

MapReduce Enhanced Open-Source Feature: JobHistoryServer HA

JobHistoryServer (JHS) is the server used to view historical MapReduce task information. Currently, the open source JHS supports only single-instance services. JHS HA can solve the problem that an application fails to access the MapReduce API when SPOFs occur on the JHS, which causes the application fails to be executed. This greatly improves the high availability of the MapReduce service.

Figure 5-106 Status transition of the JobHistoryServer HA active/standby switchover



JobHistoryServer High Availability

- ZooKeeper is used to implement active/standby election and switchover.
- JHS uses the floating IP address to provide services externally.
- Both the JHS single-instance and HA deployment modes are supported.
- Only one node starts the JHS process at a time point to prevent multiple JHS operations from processing the same file.
- You can perform scale-out, scale-in, instance migration, upgrade, and health check.

Enhanced Open Source Feature: Improving MapReduce Performance by Optimizing the Merge/Sort Process in Specific Scenarios

The figure below shows the workflow of a MapReduce task.

Figure 5-107 MapReduce job

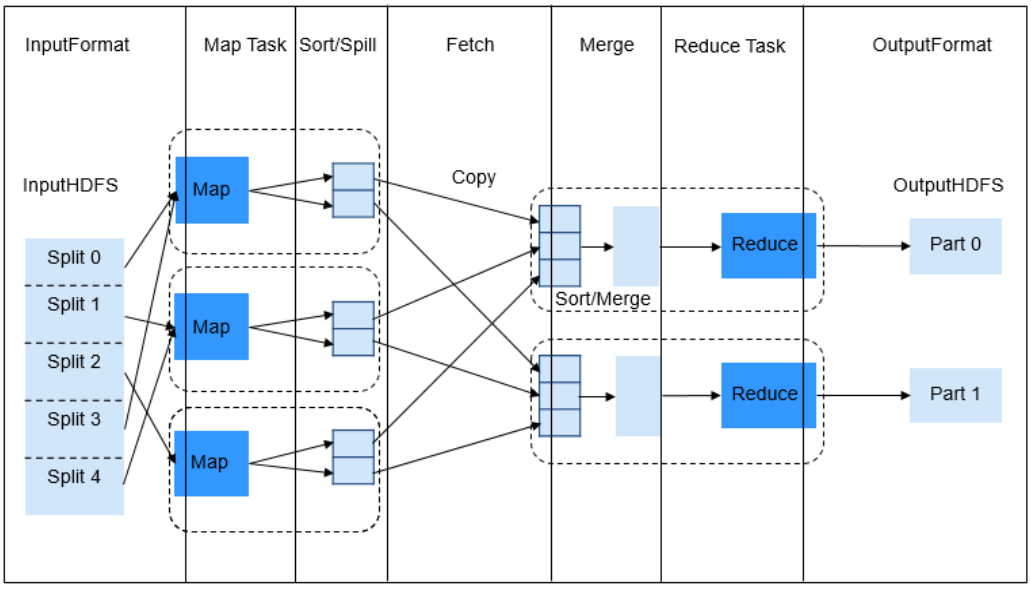
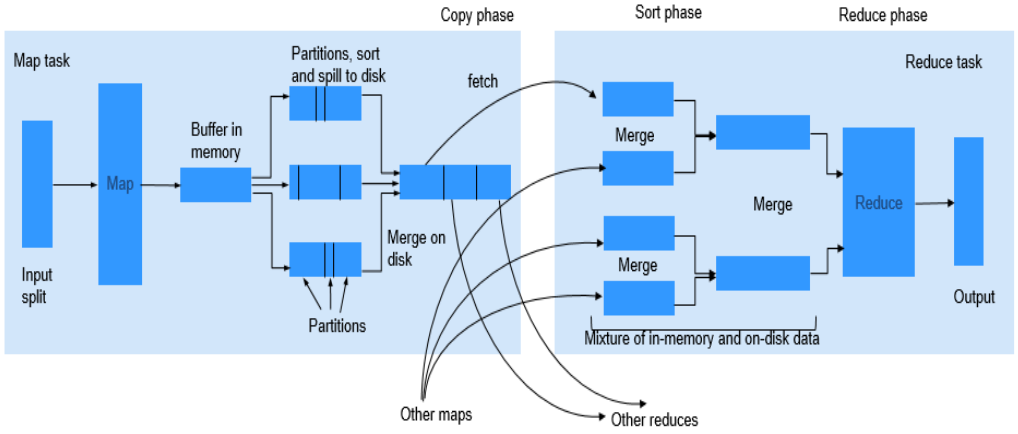


Figure 5-108 MapReduce job execution flow



The Reduce process is divided into three different steps: Copy, Sort (actually supposed to be called Merge), and Reduce. In Copy phase, Reducer tries to fetch the output of Maps from NodeManagers and store it on Reducer either in memory or on disk. Shuffle (Sort and Merge) phase then begins. All the fetched map outputs are being sorted, and segments from different map outputs are merged before being sent to Reducer. When a job has a large number of maps to be processed, the shuffle process is time-consuming. For specific tasks (for example, SQL tasks such as hash join and hash aggregation), sorting is not mandatory during the shuffle process. However, the sorting is required by default in the shuffle process.

This feature is enhanced by using the MapReduce API, which can automatically close the Sort process for such tasks. When the sorting is disabled, the API directly merges the fetched Maps output data and sends the data to Reducer. This greatly saves time, and significantly improves the efficiency of SQL tasks.

Enhanced Open Source Feature: Small Log File Problem Solved After Optimization of MR History Server

After the job running on Yarn is executed, NodeManager uses LogAggregationService to collect and send generated logs to HDFS and deletes them from the local file system. After the logs are stored to HDFS, they are managed by MR HistoryServer. LogAggregationService will merge local logs generated by containers to a log file and upload it to the HDFS, reducing the number of log files to some extent. However, in a large-scale and busy cluster, there will be excessive log files on HDFS after long-term running.

For example, if there are 20 nodes, about 18 million log files are generated within the default clean-up period (15 days), which occupy about 18 GB of the memory of a NameNode and slow down the HDFS system response.

Only the reading and deletion are required for files stored on HDFS. Therefore, Hadoop Archives can be used to periodically archive the directory of collected log files.

Archiving Logs

The AggregatedLogArchiveService module is added to MR HistoryServer to periodically check the number of files in the log directory. When the number of files reaches the threshold, AggregatedLogArchiveService starts an archiving task to archive log files. After archiving, it deletes the original log files to reduce log files on HDFS.

Cleaning Archived Logs

Hadoop Archives does not support deletion in archived files. Therefore, the entire archive log package must be deleted upon log clean-up. The latest log generation time is obtained by modifying the AggregatedLogDeletionService module. If all log files meet the clean-up requirements, the archive log package can be deleted.

Browsing Archived Logs

Hadoop Archives allows URI-based access to file content in the archive log package. Therefore, if MR History Server detects that the original log files do not exist during file browsing, it directly redirects the URI to the archive log package to access the archived log file.

NOTE

- This function invokes Hadoop Archives of HDFS for log archiving. Because the execution of an archiving task by Hadoop Archives is to run an MR application. Therefore, after an archiving task is executed, an MR execution record is added.
- This function of archiving logs is based on the log collection function. Therefore, this function is valid only when the log collection function is enabled.

5.28 MemArtsCC

5.28.1 MemArtsCC Basic Principles

MemArtsCC is a distributed caching service designed for the architecture with decoupled storage and compute. It adopts a lightweight architecture and is

deployed in a compute cluster. It prefetches data from remote object storage to provide high-speed access to these data, accelerating execution of compute tasks.

MemArtsCC shards objects on remote object storage (OBS) and creates indexes, greatly improving the performance of reading cached data. ZooKeeper is used to make service discovery lightweight and provides ultra-high availability. The lifecycle management of sharded data is based on the LRU algorithm.

Main Features

- The decentralized architecture enables all instances to provide same service capabilities.
- With a lightweight design, the resource usage is extremely low.
- MemArtsCC is decoupled from applications and therefore is transparent to them and can be used without adaptation.
- MemArtsCC ensures high availability in case of node failures.

MemArtsCC Structure

There are CCSideCar and CCWorker roles of MemArtsCC instances.

In an architecture with decoupled storage and compute, data of computing and analytics applications such as Spark and Hive is stored in OBS. In a MemArtsCC cluster, a service instance is called a worker. Workers cache some or all of the object data in OBS to local persistent storage (SSD/HDD). When an application reads an object through the MemArtsCC SDK, the application reads sharded data from a specific worker based on the shard index. If the cache is hit, the worker returns the shards. If the cache is not hit, the application directly reads data from OBS. The worker asynchronously loads the shards that are not hit to local storage for subsequent use.

Figure 5-109 MemArtsCC structure

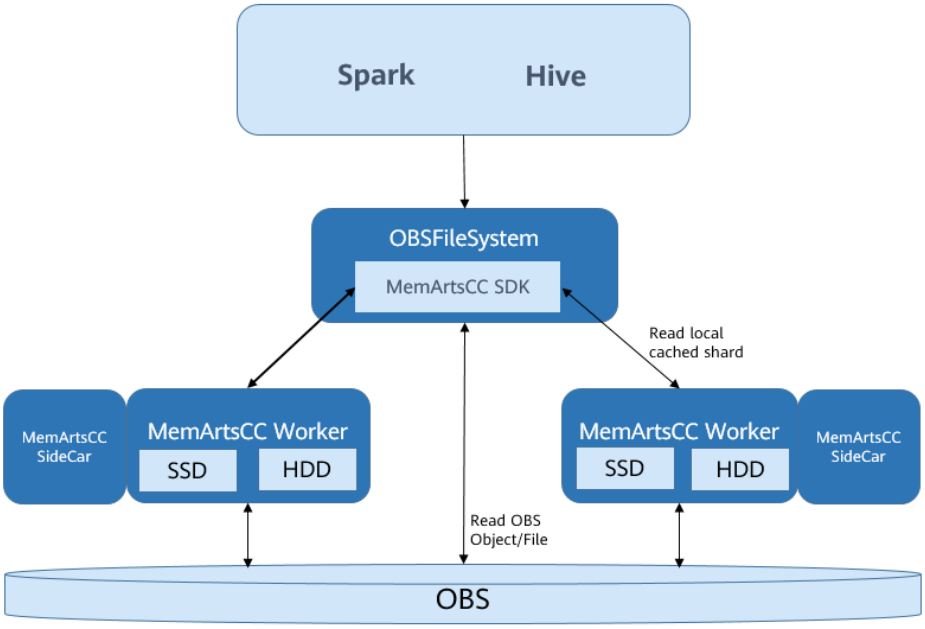


Table 5-24 Structure

Name	Description
MemArtsCC SDK	SDK used by OBSA, a Hadoop client plug-in on the FS client, to access OBS server objects.
CCSideCar	The management plane service monitors MemArtsCC, collects data, delivers configurations, and starts and stops the service.
CCWorker	The data plane service reads/writes, stores, and deletes data cached by MemArtsCC.

5.28.2 Relationships Between MemArtsCC and Other Components

OBS

OBS provides a new InputStream: OBSMemArtsCCInputStream. This InputStream reads data from the MemArtsCC cluster deployed on the compute side to reduce OBS server pressure and improve data read performance.

MemArtsCC persistently stores data to the storage (SSD) on the compute side. OBS interconnects with MemArtsCC to:

1. Improve the data access performance of the architecture where storage and compute are decoupled.

   The local storage of MemArtsCC avoids the cross-network access of hotspot data. This accelerates the data reads of OBS upper-layer applications.

2. Reduce the pressure on the OBS server.

   MemArtsCC stores hotspot data in the compute cluster to reduce the bandwidth pressure of the OBS server.

Spark

Spark reads data from OBS. OBS reads data from MemArtsCC. If data is hit in the local cache, the data is read directly. Otherwise, the data is prefetched.

Hive

Hive reads data from OBS. OBS reads data from MemArtsCC. If data is hit in the local cache, the data is read directly. Otherwise, the data is prefetched.

HetuEngine

HetuEngine reads data from OBS. OBS reads data from MemArtsCC. If data is hit in the local cache, the data is read directly. Otherwise, the data is prefetched.

5.29 MOTService

5.29.1 MOTService Basic Principles

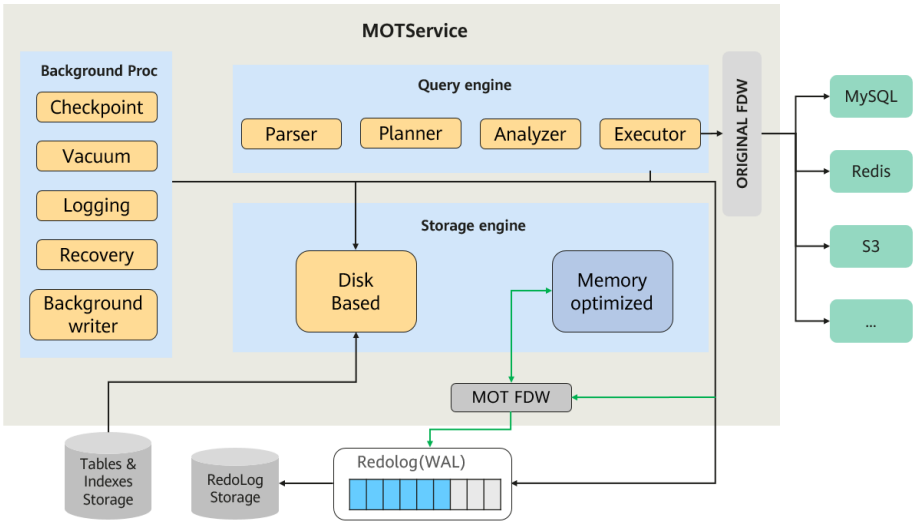
Overview

MOTService is an in-memory table engine developed based on GaussDB(for openGauss). It features high throughput and low latency, and further improves performance based on the high-performance, high-security, and high-reliability enterprise-level relational database capabilities of GaussDB(for openGauss). It supports transactions and complete transaction ACID features. In FusionInsight RTD, MOTService provides data storage, rule calculation, and data query services for RTDService.

Principles

MOTService is an in-memory table engine developed based on GaussDB(for openGauss). It is essentially an OLTP standalone database. It optimizes execution, precompilation of stored procedures, and optimistic locking of MVCC, and achieves millisecond-level latency and thousand-level TPS in RTDService's rule calculation.

Figure 5-110 MOTService structure



- Stored procedure precompilation: Based on LLVM, stored procedures are precompiled to a format that can be directly invoked locally. This skips multi-layer database processing logic and significantly improves performance. The precompilation results of stored procedures are cached in the memory and can be invoked by subsequent sessions like the pointers in C. For the same stored procedure, the precompilation result can be reused even if the request comes from different sessions or parameters.
- Execution optimization: MOTService provides faster data access and more efficient transaction execution through data and indexes completely stored in memory, non-uniform memory access-aware (NUMA-aware) design, algorithms that eliminate locks and lock contention, and query native compilation. In addition, MOTService indexes are based on the state-of-the-art lock-free indexing of Masstree for fast and scalable key-value (KV) storage of multi-core systems, which is implemented through the Trie of a B+ tree. It

achieves excellent performance on multi-core servers and high concurrent workloads.

- MVCC optimistic locking: An optimistic concurrency control (OCC) lock is introduced based on the Silo database. The database does not block in the read/write phase and conflict detection and retry are performed only in the transaction submission phase, greatly reducing the blocking time. Optimistic locking is less expensive and often more efficient, because transaction conflicts are not common in most applications.

Relationship with Other Components

You can define stored procedure rules and real-time query variables using the web UI provided by RTDService. The variables and rules are compiled in real time to generate compilation processes and deployed these processes on the MOTService database. After the event source dimension mapping is brought online, the corresponding BLU execution rule accesses the defined stored procedure of the MOTService in real time.

5.29.2 MOTService Enhanced Features

Memory Optimized Data Structures

MOTService uses a memory-optimized data structure that is more suitable for large-memory and multi-core servers. All data and indexes are stored in the memory, no intermediate page caches are used, and the lock with the shortest duration is used. The data structure and all algorithms are optimized for memory design. Memory-optimized tables are created side by side regular disk-based tables. MOTService's effective design enables almost full SQL coverage and support for a full database feature-set. MOTService is fully ACID compliant and includes strict durability and high availability support.

Lock-Free Transaction Management

While ensuring strict consistency and data integrity, MOT uses optimistic policies to achieve high concurrency and high throughput. During a transaction, the MOT does not lock any version of the data row being updated, greatly reducing contention in some large memory systems. Optimistic concurrency control (OCC) in transactions is implemented without locks. All data modification is performed in the part of memory dedicated to private transactions (also called private transactional memory). This means that during a transaction, related data is updated in the private transactional memory, thereby implementing lock-free read and write. In addition, a lock is locked for a short time only in the commit phase.

Lock-Free Index

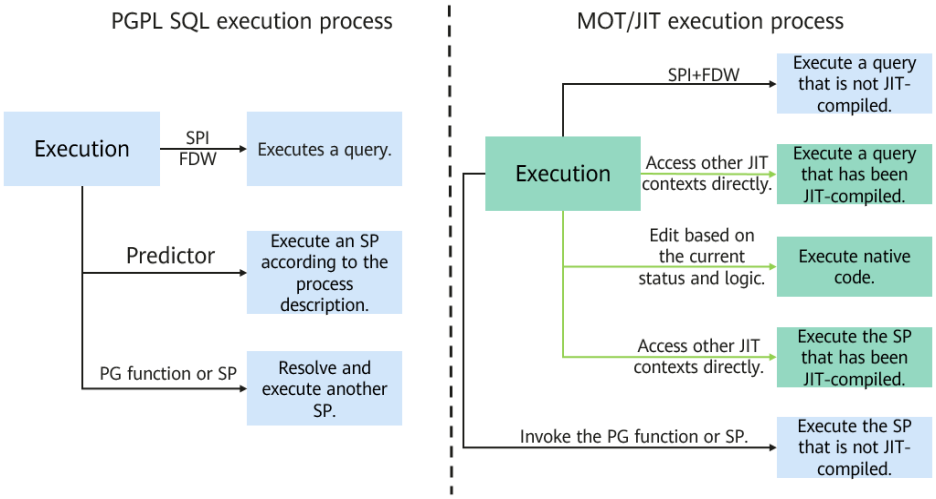
The data and indexes of memory tables are stored in the memory. Therefore, it is important to have an efficient index data structure and algorithm. The MOTService index mechanism is based on the state-of-the-art Masstree, which is a fast and scalable Key Value (KV) storage index for multi-core systems and is implemented using the Trie of the B+ tree. In this way, excellent performance on multi-core servers can be achieved in the case of high-concurrency workloads. Masstree is a combination of tries and a B+ tree that is implemented to carefully

exploit caching, prefetching, optimistic navigation, and fine-grained locking. However, the downside of a Masstree index is its higher memory consumption. MOTService's main innovation was to enhance the original Masstree data structure and algorithm, which did not support non-unique indexes. Another improvement is Arm architecture support.

Native Statements for Query

With the PREPARE client commands, users can execute query and transaction statements interactively. These commands have been pre-compiled into native execution formats, also known as Code-Gen or Just-in-Time (JIT) compilation. In this way, the performance can be improved by 30% on average. If possible, apply compilation and lightweight execution; otherwise, use the standard execution path to process the applicable query. The Cache Plan module has been optimized for OLTP. Different binding settings are used in the entire session and compilation results are reused in different sessions. **Figure 5-111** shows the concepts of JIT queries and stored procedures.

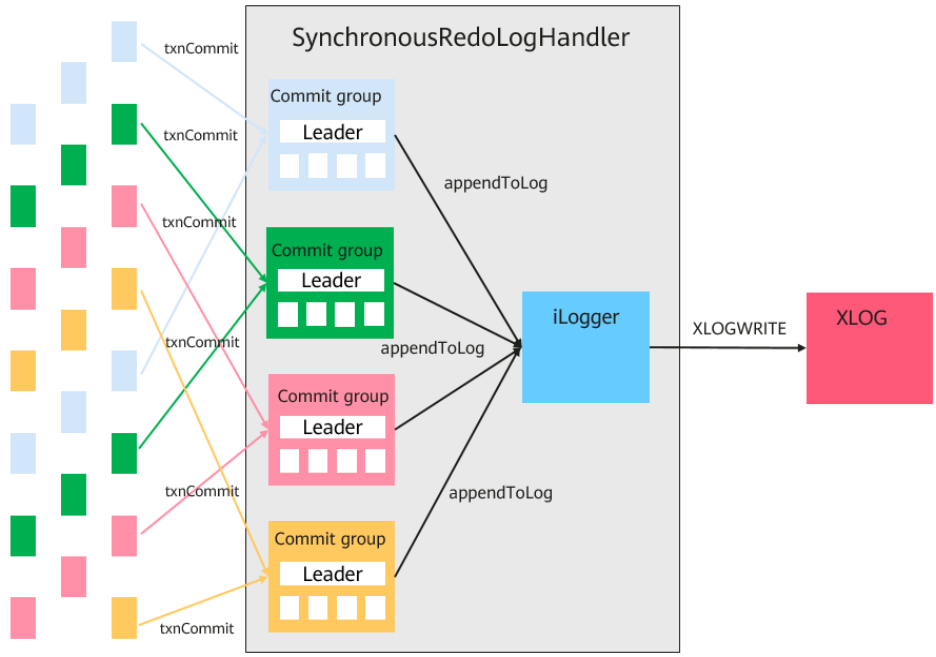
Figure 5-111 JIT queries and stored procedures



NUMA-ware Memory Management

MOTService memory access is designed with Non-Uniform Memory Access (NUMA) awareness. NUMA-aware algorithms enhance the performance of a data layout in memory so that threads access the memory that is physically attached to the core on which the thread is running. This is handled by the memory controller without requiring an extra hop by using an interconnect, such as Intel QPI. MOTService's smart memory control module with pre-allocated memory pools for various memory objects improves performance, reduces locks and ensures stability. Allocation of a transaction's memory objects is always NUMA-local. Deallocated objects are returned to the pool. Minimal usage of OS malloc during transactions circumvents unnecessary locks. The MOTService engine performs synchronous Group Commit logging with NUMA optimization by automatically grouping transactions according to the NUMA socket of the core on which the transaction is running.

Figure 5-112 MOTService memory access



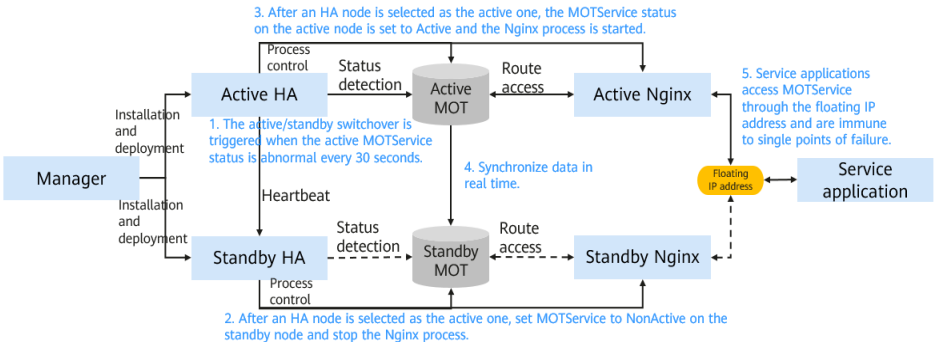
MOTService Active/Standby HA

MOTService uses the HA module of Manager for automatic active/standby switchover. The active HA process checks whether the active MOTService process on the same node is normal every 30 seconds.

If the process is abnormal, MOTService status is set to NonActive, the Nginx process on the same node is stopped, and the original standby instance is promoted to the active instance. Then, MOTService status on the same node is set to Active, and the Nginx process on the same node is started.

Both active and standby Nginx instances are configured to listen to the same floating IP address. Service applications can access MOTService through the Nginx route by connecting to the floating IP address. Therefore, the active/standby switchover of Nginx and MOTService is transparent to the interfaces used by service applications.

Figure 5-113 Active/standby HA deployment



5.30 Oozie

5.30.1 Oozie Basic Principles

Introduction to Oozie

Oozie is an open-source workflow engine that is used to schedule and coordinate Hadoop jobs.

Architecture

The Oozie engine is a web application integrated into Tomcat by default. Oozie uses PostgreSQL databases.

Oozie provides an Ext-based web console, through which users can view and monitor Oozie workflows. Oozie provides an external REST web service API for the Oozie client to control workflows (such as starting and stopping operations), and orchestrate and run Hadoop MapReduce tasks. For details, see **Figure 5-114**.

Figure 5-114 Oozie architecture

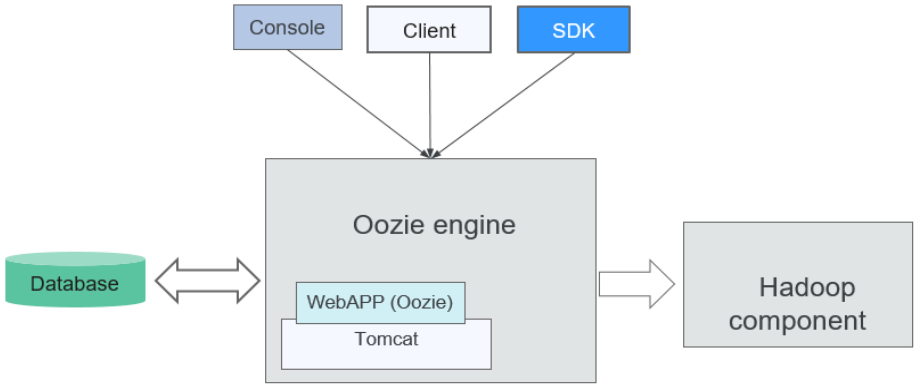


Table 5-25 describes the functions of each module shown in **Figure 5-114**.

Table 5-25 Architecture description

Connection Name	Description
Console	Allows users to view and monitor Oozie workflows.
Client	Controls workflows, including submitting, starting, running, planting, and restoring workflows, through APIs.
SDK	Is short for software development kit. An SDK is a set of development tools used by software engineers to establish applications for particular software packages, software frameworks, hardware platforms, and operating systems.

Connection Name	Description
Database	PostgreSQL database
WebApp (Oozie)	Functions as the Oozie server. It can be deployed on a built-in or an external Tomcat container. Information recorded by WebApp (Oozie) including logs is stored in the PostgreSQL database.
Tomcat	A free open-source web application server
Hadoop components	Underlying components, such as MapReduce and Hive, that execute the workflows orchestrated by Oozie.

Principle

Oozie is a workflow engine server that runs MapReduce workflows. It is also a Java web application running in a Tomcat container.

Oozie workflows are constructed using Hadoop Process Definition Language (HPDL). HPDL is an XML-defined language, similar to JBoss jBPM Process Definition Language (jPDL). An Oozie workflow consists of the Control Node and Action Node.

- Control Node controls workflow orchestration, such as **start**, **end**, **error**, **decision**, **fork**, and **join**.
- An Oozie workflow contains multiple Action Nodes, such as MapReduce and Java.

  All Action Nodes are deployed and run in Direct Acyclic Graph (DAG) mode. Therefore, Action Nodes run in direction. That is, the next Action Node can run only when the running of the previous Action Node ends. When one Action Node ends, the remote server calls back the Oozie interface. Then Oozie executes the next Action Node of workflow in the same manner until all Action Nodes are executed (execution failures are counted).

Oozie workflows provide various types of Action Nodes, such as MapReduce, Hadoop distributed file system (HDFS), Secure Shell (SSH), Java, and Oozie sub-flows, to support a wide range of business requirements.

5.30.2 Oozie Enhanced Open Source Features

Enhanced Open Source Feature: Improved Security

Provides roles of administrator and common users to support Oozie permission management.

Supports single sign-on and sign-out, HTTPS access, and audit logs.

Enhanced Open Source Feature: Improved HA

Uses ZooKeeper's HA feature to prevent single points of failure (SPOFs) when multiple Oozie nodes provide services at the same time.

5.31 Ranger

5.31.1 Ranger Basic Principles

Apache Ranger offers a centralized security management framework and supports unified authorization and auditing. It manages fine grained access control over Hadoop and related components, such as HDFS, Hive, HBase, and Kafka. You can use the front-end web UI console provided by Ranger to configure policies to control users' access to these components.

Figure 5-115 shows the Ranger architecture.

Figure 5-115 Ranger structure

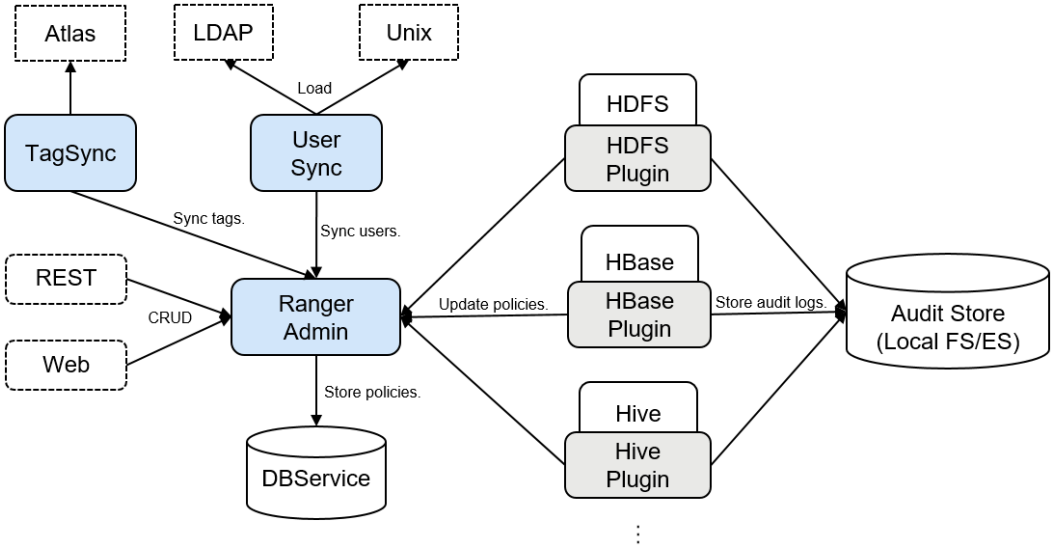


Table 5-26 Architecture description

Connection Name	Description
RangerAdmin	Provides a web UI and RESTful APIs to manage policies, users, and auditing.
UserSync	Periodically synchronizes user and user group information from an external system and writes the information to RangerAdmin.
TagSync	Periodically synchronizes tag information from the external Atlas service and writes the tag information to RangerAdmin.
RangerKMS	Ranger key management service, which can be used for Hadoop transparent encryption.

Ranger Principles

- Ranger Plugins

  Ranger provides policy-based access control (PBAC) plug-ins to replace the original authentication plug-ins of the components. Ranger plug-ins are developed based on the authentication interface of the components. Users set permission policies for specified services on the Ranger web UI. Ranger plug-ins periodically update policies from the RangerAdmin and caches them in the local file of the component. When a client request needs to be authenticated, the Ranger plug-in matches the user carried in the request with the policy and then returns an accept or reject message.

- UserSync User Synchronization

  UserSync periodically synchronizes data from LDAP/Unix to RangerAdmin. In security mode, data is synchronized from LDAP. In non-security mode, data is synchronized from Unix. By default, the incremental synchronization mode is used. In each synchronization period, UserSync updates only new or modified users and user groups. When a user or user group is deleted, UserSync does not synchronize the change to RangerAdmin. That is, the user or user group is not deleted from the RangerAdmin. To improve performance, UserSync does not synchronize user groups to which no user belongs to RangerAdmin.

- Unified auditing

  Ranger plug-ins can record audit logs. Currently, audit logs can be stored in local files or Elasticsearch. By default, audit logs are stored in local files. To enable Elasticsearch storage, enable it by following the instructions provided in the guide and query the audit details of the corresponding components on the Audit tab page of Ranger WebUI.

- High reliability

  Ranger supports two RangerAdmins working in active/active mode. Two RangerAdmins provide services at the same time. If either RangerAdmin is faulty, Ranger continues to work.

- High performance

  Ranger provides the Load-Balance capability. When a user accesses Ranger WebUI using a browser, the Load-Balance automatically selects the RangerAdmin with the lightest load to provide services.

RangerKMS Principles

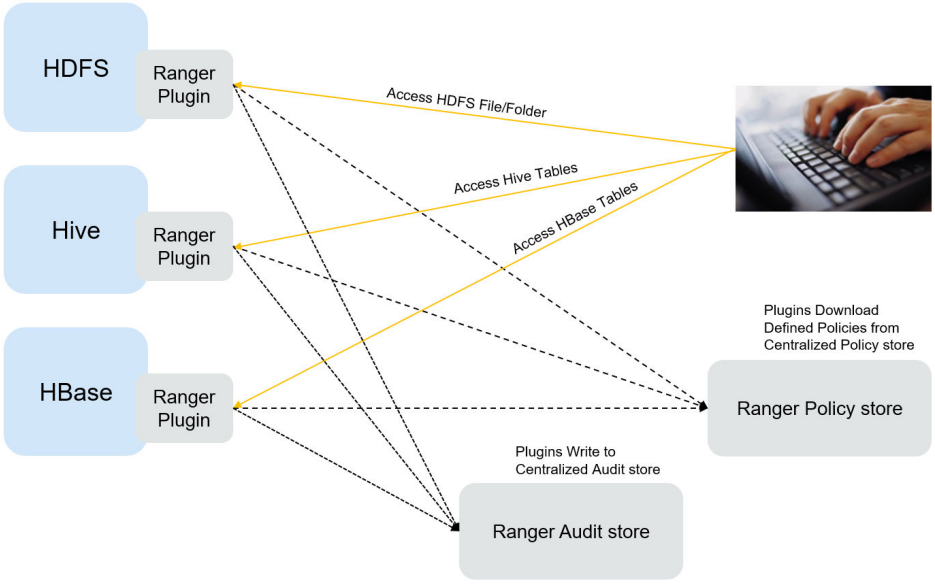
RangerKMS manages authentication keys based on HadoopKMS. Symmetric AES encryption algorithms are used to provide a C/S interaction model that uses REST APIs for HTTP communications. KMS and its clients are secure and support HTTP SPNEGO Kerberos authentication and HTTPS. RangerKMS is a Tomcat web application. RangerKMS outperforms HadoopKMS with the following features:

- Key storage: RangerKMS keys can be stored in databases or HSMs. The keys remain consistent when caching is disabled.
- ACL control: RangerAdmin is used for fine-grained and key permission management.
- Third-party HSMs: RangerKMS can interconnect with Huawei Cloud DEW.

5.31.2 Relationships Between Ranger and Other Components

Ranger provides PABC-based authentication plug-ins for components to run on their servers. Ranger currently supports authentication for the following components like HDFS, YARN, Hive, HBase, Kafka,Elasticsearch, and Spark. More components will be supported in the future.

Figure 5-116 Relationships between Ranger and other components



Relationships Between RangerKMS and HDFS

When HDFS is interconnected with RangerKMS, keys are obtained from RangerKMS during encryption. When an HDFS encrypted area is created, the NameNode obtains the value from RangerKMS.

Relationships Between RangerKMS and ZooKeeper

Multiple instances of RangerKMS share token information, and the token information is stored in ZooKeeper.

5.32 Redis

5.32.1 Redis Basic Principles

Introduction to Redis

Redis is an open-source, network-based, and high-performance key-value database. It makes up for the shortage of memcached key-value storage. In some scenarios, Redis can be used as a supplement to relational databases to meet real-time and high-concurrency requirements.

Redis is similar to Memcached. Besides, it supports data persistence and diverse data types. Redis also supports the calculation of the union, intersection, and complement of sets on the server as well as multiple sorting functions.

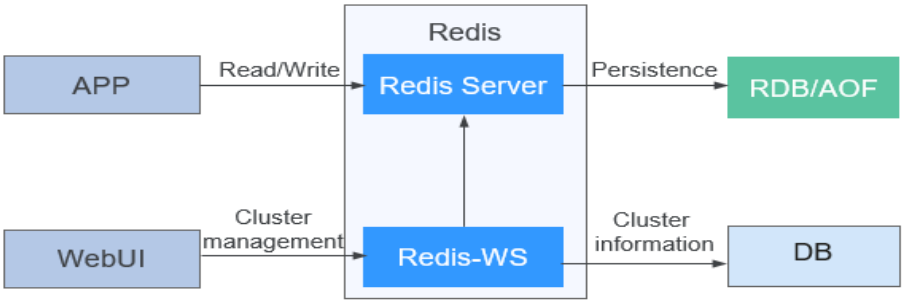
NOTE

The network data transmission between the Redis client and server is not encrypted, which brings security risks. Therefore, It is advised not to use Redis to store sensitive data.

Redis Architecture

Redis consists of Redis Server and Redis-WS, as shown in **Figure 5-117**.

Figure 5-117 Redis logical architecture



- Redis Server: core module of the Redis. It is responsible for data read and write of the Redis protocol, active/standby replication, and maintain the data persistence and cluster functions.
- Redis-WS: Redis WebService management module. It implements operations such as cluster creation/deletion, scaling-out/scaling-in, and cluster querying, and stores cluster management information in the DB.

Redis Principles

Redis persistence

Redis supports the following types of persistence:

- Redis Database File (RDB) persistence

  Point-in-time snapshots are generated for data sets in specified intervals.
- Append Only File (AOF) persistence

  All write operation commands executed by a server are recorded. When the server starts, the recorded commands will be executed to restore data sets. All commands in the AOF file are saved in the Redis protocol format. New commands are added to the end of the file. Redis allows the AOF file to be rewritten in the background, preventing the file size from exceeding the actual size required for storing data set status.

Redis supports AOF and RDB persistence at the same time. When Redis restarts, it preferentially uses AOF to restore data sets because the AOF contains more complete data sets than the RDB. The data persistence function can also be disabled. When it is disabled, data exists only when the server is running.

Redis running mode

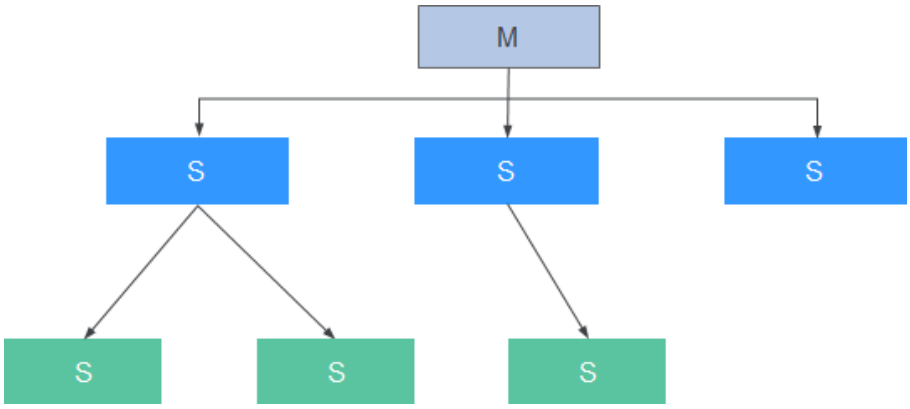
Redis instances can be deployed on one or more nodes, and one or more Redis instances can be deployed on one node. (On the MRS platform, the number of Redis instances on each node is calculated by software based on the node hardware resources.)

The latest Redis supports clusters. That is, multiple Redis instances constitute a Redis cluster to provide a distributed key-value database. Clusters share data through sharding and provide replication and failover functions.

- Single instance mode

  Figure 5-118 shows the logical deployment of the single-instance mode.

  Figure 5-118 Single instance mode

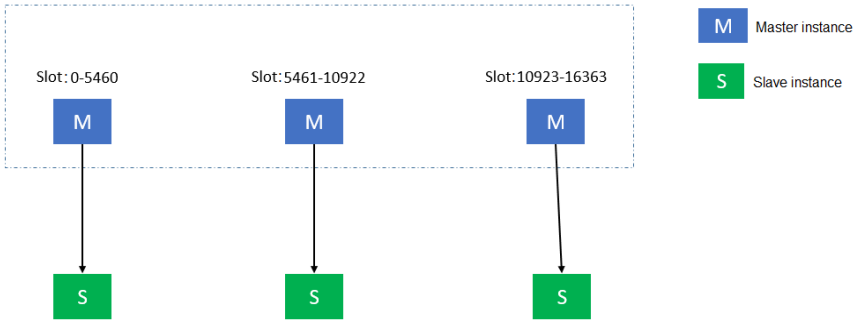
  Note:

  - A master instance has multiple slave instances. A slave instance can have slave instances as well.
  - Command requests sent to the master instance are synchronized to the slave instance in real time.
  - If the master instance is faulty, the slave instance will not be automatically promoted to the master one.
  - By default, the slave instance is read-only. If **slave-read-only** is set to **no**, the slave instance can be written. But if the slave instance is restarted, it will synchronize the data from the master instance, and the data written to the slave instance earlier will be lost.
  - The layered structure of slave instances reduces the number of instances directly connected to the master instance. This structure improves service processing performance of the master instance because the number of slave instances that need to synchronize data from the master instance is reduced.

- Cluster mode

  Figure 5-119 shows the logical deployment mode of the cluster mode.

Figure 5-119 Cluster mode



Note:

- Multiple Redis instances constitute a Redis cluster, in which 16,384 slots are evenly distributed to master instances.
- Every instance in the cluster record the mapping between slots and instances, so do the clients. The client performs hash calculation based on the key and performs modulo operation with 16384 to obtain the slot ID. The message is directly sent to the corresponding instance for processing based on the slot-instance mapping.
- By default, slave instances cannot read or write data. Running the **readonly** command can enable a slave instance to read data only.
- If a master instance is faulty, the remaining master instances in the cluster will select a slave one to serve as a new master instance. The selection can be performed only when more than half of the master instances in the cluster are normal.
- If **cluster-require-full-coverage** is set to **yes**, the cluster status is **FAIL** when a group of master and slave instances is faulty. If this occurs, the cluster cannot process commands. If **cluster-require-full-coverage** is set to **no**, the cluster status is normal as long as more than half of the master instances are normal.
- You can scale out or scale in a Redis cluster (by adding a new instance to the cluster or removing an existing Redis instance from the cluster) and migrate slots.
- At present, each Redis cluster in MRS supports only one-to-one mapping between active and slave instances.

Redis-Data-Sync

Redis-Data-Sync is a tool for implementing data synchronization between the active and standby Redis clusters. It synchronizes data of the logical clusters in the active cluster to the standby cluster in real time and backs up the data to the standby cluster so that a data replica of the active Redis cluster can be generated.

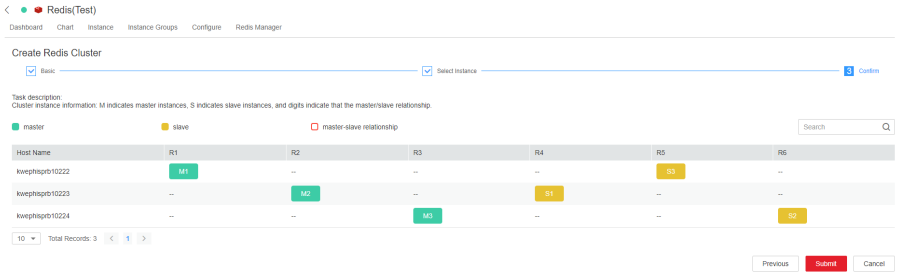
5.32.2 Redis Enhanced Open Source Features

Comprehensive Cluster Management Functions

MRS provides comprehensive Redis cluster management. On Manager, you can create Redis clusters based on the Redis instance groups to improve system processing capabilities and reliability.

- Wizard-based creation of Redis clusters

  Figure 5-120 Creating a Redis cluster

  MRS supports creation of Redis clusters in master/slave mode. The system automatically calculates the number of Redis instances to be installed on nodes and determines the master/slave relationship.

- Cluster scaling-out/scaling-in

  When large-scale data processing is required, you can add one or multiple master/slave instances in the Redis cluster by a few clicks. The system automatically completes data migration and balancing for the scaling-out.

- Balance

  Data in Redis clusters may not be evenly distributed if the scaling-out fails or some instances are offline. MRS Manager provides the balance function to implement automatic balancing of cluster data, ensuring stable operation of clusters.

- Performance monitoring and alarming

  The system provides performance monitoring of Redis clusters and intuitive curves to help users learn Redis cluster status and throughput of instances.

  The system provides diverse alarms, such as alarms for cluster offline, persistency failures, uneven slot distribution, master/slave instance switchover, cluster HA deterioration, and inconsistent memory size between master and slave instances, for Redis clusters. Diverse alarms facilitate the Redis cluster monitoring and management.

Cluster Reliability Guarantee

The cluster management tool **redis-trib.rb** provided by the Redis community enables the master and slave instances to be created in fixed sequence and cannot ensure cluster HA. If the master and slave instances are created on the same host, a failure of one host causes unavailability of the entire cluster.

When creating a Redis cluster, MRS automatically calculates the number of instances based on the selected instance range and deploys the cluster based on

the host-level HA principle. This principle is also ensured during scaling-out and scaling-in. If any host in a cluster is faulty, a master/slave instance switchover is performed, ensuring continuous cluster running.

If the cluster HA cannot be ensured when some nodes or instances are faulty at the same time, alarms will be generated prompting that rectification is required.

Data Import and Export Tool

A Redis cluster has 16,384 slots. The crc16 code of different keys is calculated to determine the slots for storing the keys. This mechanism ensures load balancing of master instances. As a result, different slots store different key values. If two clusters have different topology structures, the keys for different instances are different. This makes data migration or data restoration from backup extremely difficult.

MRS provides a dedicated data import and export tool, which can be used to export data from the Redis cluster and restore data in the original cluster, new cluster, and heterogeneous cluster (cluster with different numbers of nodes).

Comprehensive Security Features

The community Redis provides the simple password authentication mechanism, and the password in the configuration file is not encrypted. This mechanism is insecure for enterprise-class applications. MRS provides comprehensive security features and adds authentication, authorization, and audit mechanisms.

A client can send data to or request data from a server only after the authentication is successful. Authentication is also performed between the servers in a cluster to prevent requests from forged instances. In addition, Redis commands are classified into read, write, and management commands. Users are assigned different permissions to prevent unauthorized operations.

The audit mechanism logs some risky Redis operations, such as changing the cluster topology and clearing Redis data.

Performance Enhancement

Redis is a high-performance distributed database. However, deployment of Redis instances on a command OS causes limited throughout when the number of concurrent requests from clients increases even if the server has sufficient resources. In addition, the Redis cluster performance cannot be linearly improved with the cluster scaling-out. MRS has incorporated OS enhancement, including CPU binding, NIC queue binding, and OS parameter optimization, ensuring high Redis performance, especially linear performance improvement of Redis clusters.

Figure 5-121 Performance comparison of a single instance

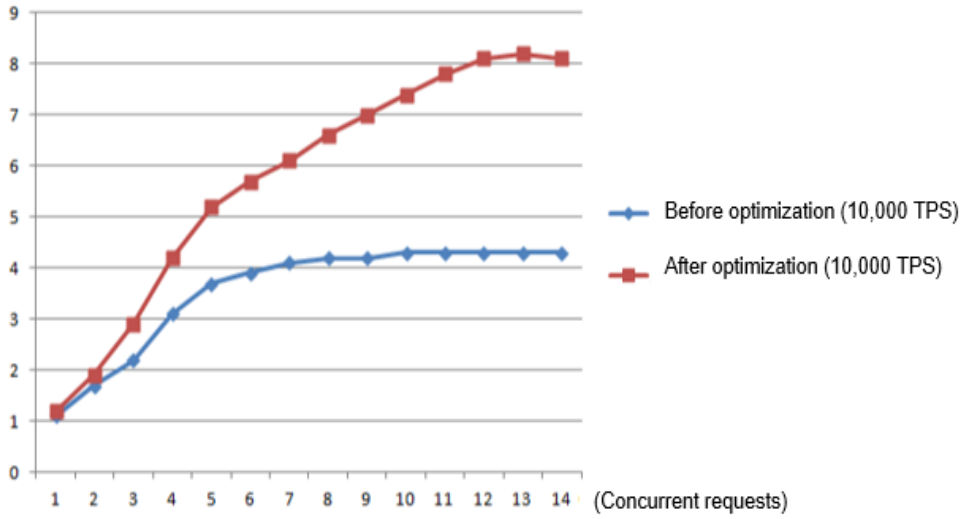
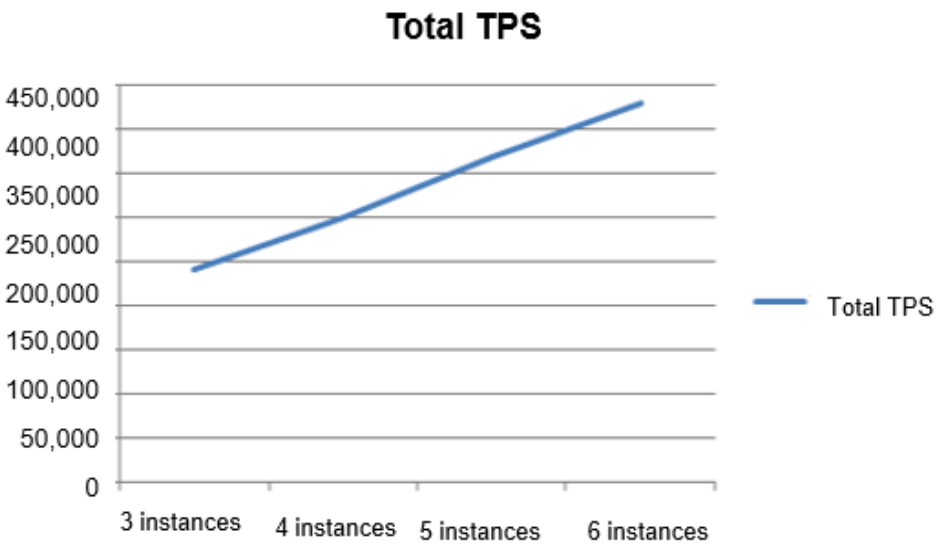


Figure 5-122 Performance comparison of clusters



Enhanced Replacement Algorithm

Redis is a cache system. When the memory usage of Redis reaches the configured maximum value, data replacement occurs. Native Redis supports three replacement policies: Least Recently Used (LRU), Random, and Time to Live (TTL). However, the purpose of replacing cold data and retaining hot data cannot be well achieved in practical use.

MRS Redis enhances the replacement algorithm and introduces the Smart placement policy. This policy is used to replace data based on key hot statistics, ensuring that only the coldest data is eliminated each time as possible. In simulated service tests, the hot data hit rate of the Smart replacement policy is always greater than 99%, and the hot data replacement rate is about 3% to the maximum (the hot data hit rate and replacement rate of the native LRU policy are

85% and 35%, respectively). Due to the improvement of hot data hit rate, the service request throughput is increased.

Cluster Pipeline

The Redis server supports pipeline commands sent from clients. That is, the Redis server can receive and process multiple commands at one time, shortening the network transmission duration and increasing the number of requests processed by the Redis server per second. However, the Jedis community provides only the single-instance pipeline mode. The clients encapsulate Jedis to ensure that the pipeline mode can also be applied in clusters and the use method of such pipeline mode is the same as that of the single-instance pipeline mode.

5.33 RTDService

5.33.1 RTDService Basic Principles

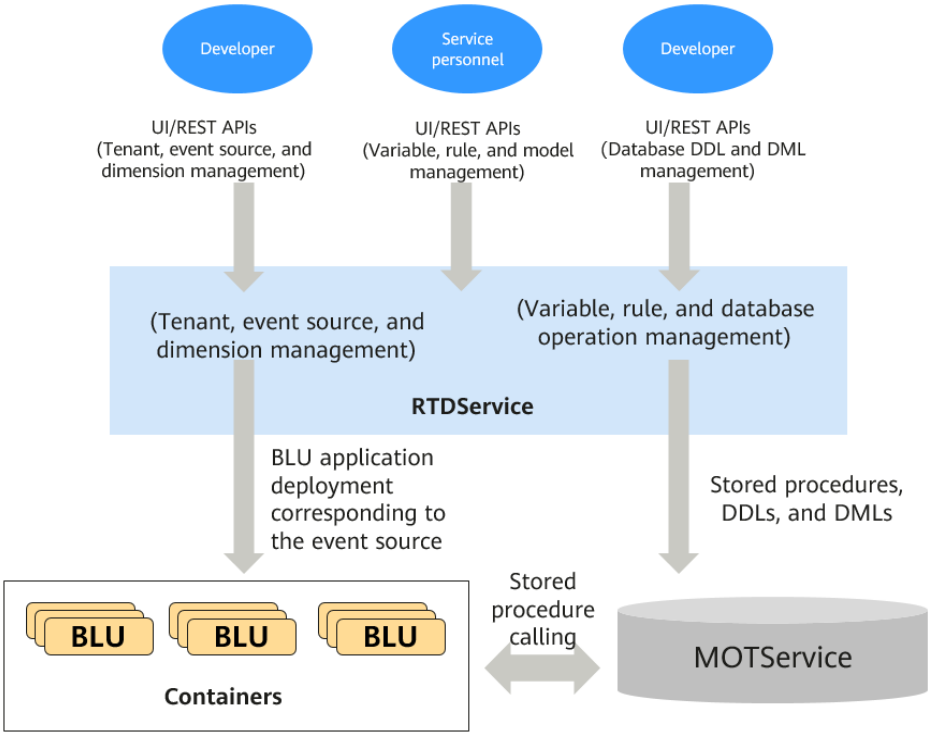
Overview

RTDService provides GUIs for service configuration and RESTful APIs for users to define tenants, event sources, dimensions, variables, rules, and models.

Principles

RTDService consists of the RTDServer role. Metadata such as event sources, dimensions, dimension mapping, variables, models, and rules defined on the web UI is permanently saved to DBService. After the event source dimension mapping is brought online, the RTDServer role automatically generates a BLU application and deploys the application in a group of containers of Containers. After variables or rules defined on the RTDService web UI are brought online, RTDServer automatically generates stored procedures and deploys them in MOTService.

Figure 5-123 Relationship between modules of RTDService and interaction between the modules

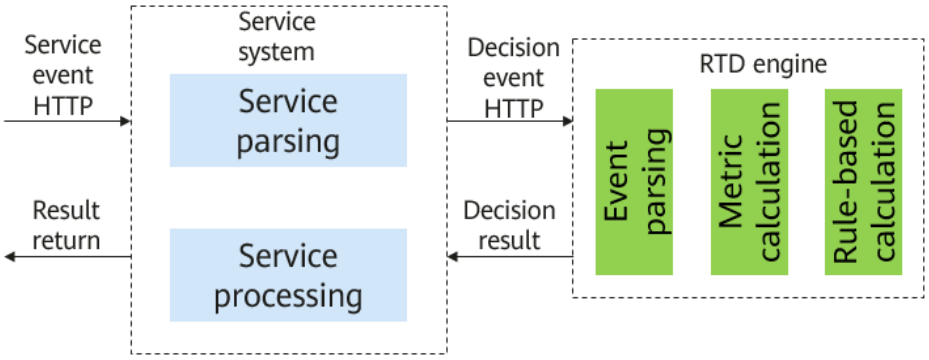


5.33.2 RTDService Enhanced Features

HTTP Event Access

RTDService supports HTTP access. Compared with message queues, there is no latency caused by async/await. Therefore, RTDService can support real-time analysis and decision-making during service events.

Figure 5-124 HTTP event access

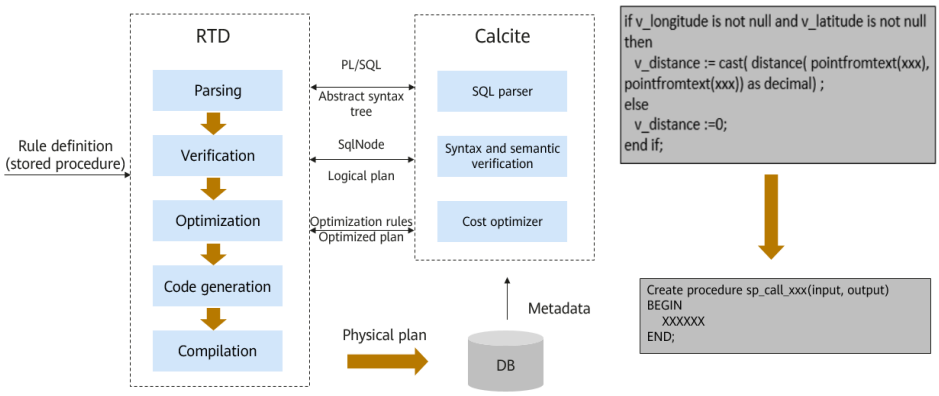


PL/SQL Rules — Dynamic Online and Offline

Rules and variable metrics in RTDService are defined using PL/SQL stored procedures. RTDService uses a widely used language that most developers can master. In addition, rules and variables can be dynamically put into or out of

service in seconds. The rules and variables take effect in real time and do not interrupt services.

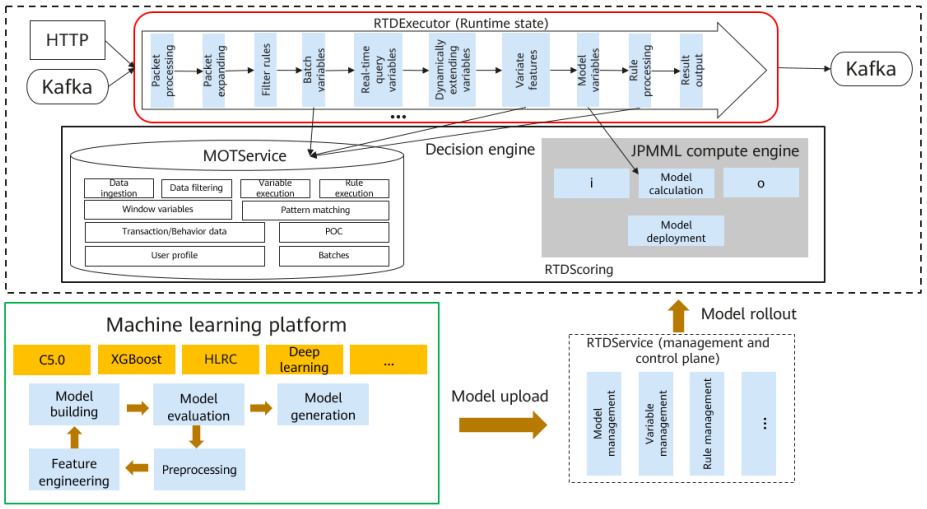
Figure 5-125 Dynamic online and offline



Convergent Decision-Making Based on Models and Rules

RTDService uses the JPMML as the compute engine for modeling and uses the compute results to generate rules to support decision-making based on both models and rules.

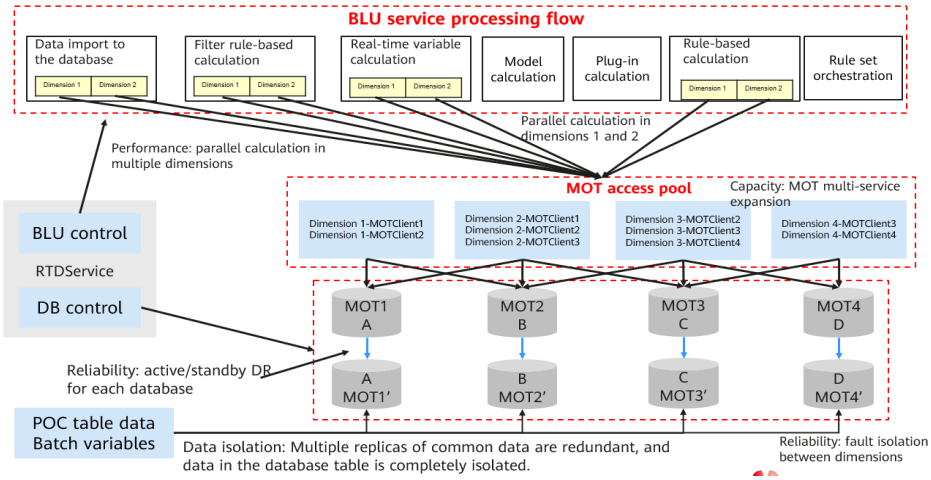
Figure 5-126 Convergent decision-making based on models and rules



Database and Table Sharding

RTDService supports database sharding by dimension. Service data is routed to different databases based on dimension primary keys. Data and resources of each dimension are isolated, leading to improved capacity, performance, and reliability.

Figure 5-127 Database and table sharding



5.34 Apache Solr

5.34.1 Solr Basic Principle

Solr is a high-performance Lucene-based full-text retrieval server. Extended based on Lucene, Solr provides more diversified query languages than Lucene, implements the full-text search function, and supports highlighting display and dynamic clusters, providing high scalability. Solr 4.0 and later versions support the SolrCloud mode. In this mode, centralized configuration, near-real-time search, and automatic fault tolerance functions are supported.

- Uses ZooKeeper as the collaboration service. When ZooKeepers are started, users can specify the related Solr configuration files to be uploaded to the ZooKeepers for multiple machines to share. Configuration in the ZooKeepers will not be cached locally. Solr directly reads the configuration information in the ZooKeepers. Modification of the configuration files will be sensed by all machines.
- Supports automatic fault tolerance. SolrCloud divides a Collection into multiple Shards and creates multiple Replicas for each Shard. After a Replica breaks down, the entire index search service will not be affected. Each Replica can independently provide services to external environments.
- Supports automatic load balancing during indexing and query. The multiple Replicas of a SolrCloud Collection can be distributed on multiple machines to balance the indexing and query pressure. If the indexing and query pressure is huge, users can add machines or Replicas to balance the pressure.
- The Solr index data can be stored in multiple modes. The HDFS can be used as the index file storage system of Solr to provide a high-reliability, high-performance, scalable, and real-time full-text search system. The data can also be stored on local disks for higher data indexing and query speed.

The Solr cluster scheme SolrCloud consists of multiple SolrServer processes, as shown in **Figure 5-128**. **Table 5-27** describes the modules.

Figure 5-128 Solr (SolrCloud) architecture

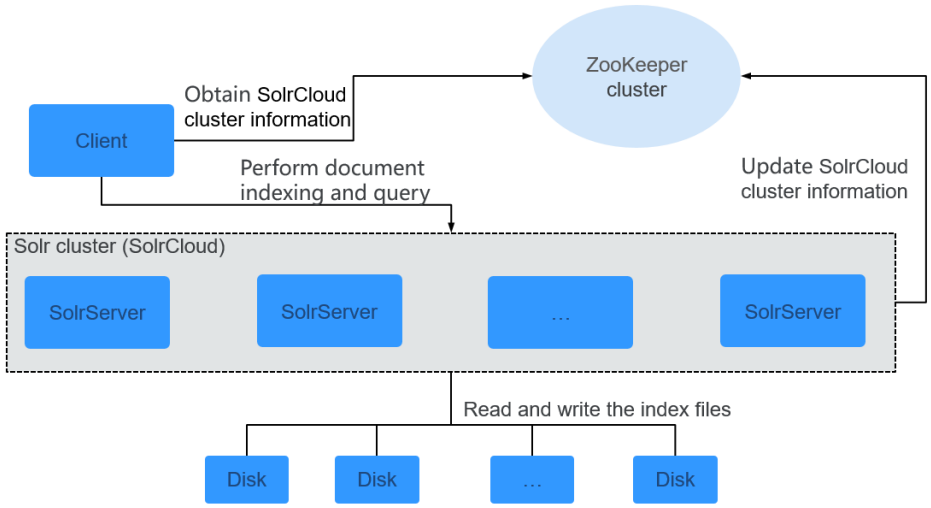


Table 5-27 Solr modules

Name	Description
Client	Client communicates with SolrServer in the Solr cluster (SolrCloud) through the HTTP or HTTPS protocol and performs distributed indexing and distributed search operations.
SolrServer	SolrServer provides various services, such as index creation and full-text retrieval. It is a data computing and processing unit in the Solr cluster.
ZooKeeper cluster	ZooKeeper provides distributed coordination services for various processes in the Solr cluster. Each SolrServer registers its information (collection configuration information and SolrServer health information) with ZooKeeper. Based on the information, Client detects the health status of each SolrServer, thereby determining distribution of indexing and search requests.

Basic Concept

- Collection: a complete logical index in a SolrCloud cluster. A Collection can be divided into multiple Shards that use the same Config Set.
- Config Set: a group of configuration files required by Solr Core to provide services. A Config Set includes **solrconfig.xml** and **managed-schema**.
- Core: refers to Solr Core. A Solr instance includes one or multiple Solr Cores. Each Solr Core independently provides indexing and query functions. Each Solr Core corresponds to an index or a Collection Shard Replica.
- Shard: a logical section of a Collection. Each Shard has multiple Replicas, among which a leader is elected.
- Replica: a copy of a Shard. Each Replica is in a Solr Core.
- Leader: a Shard Replica elected from multiple Replicas. When documents are indexed, SolrCloud transfers them to the leader, and the leader distributes them to Replicas of the Shard.

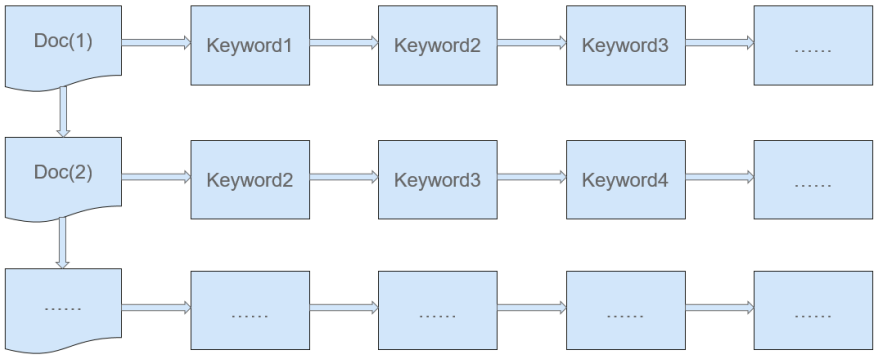
- ZooKeeper: is mandatory in SolrCloud. It provides distributed lock and Leader election functions.

Principle

- **Descending-order Indexing**

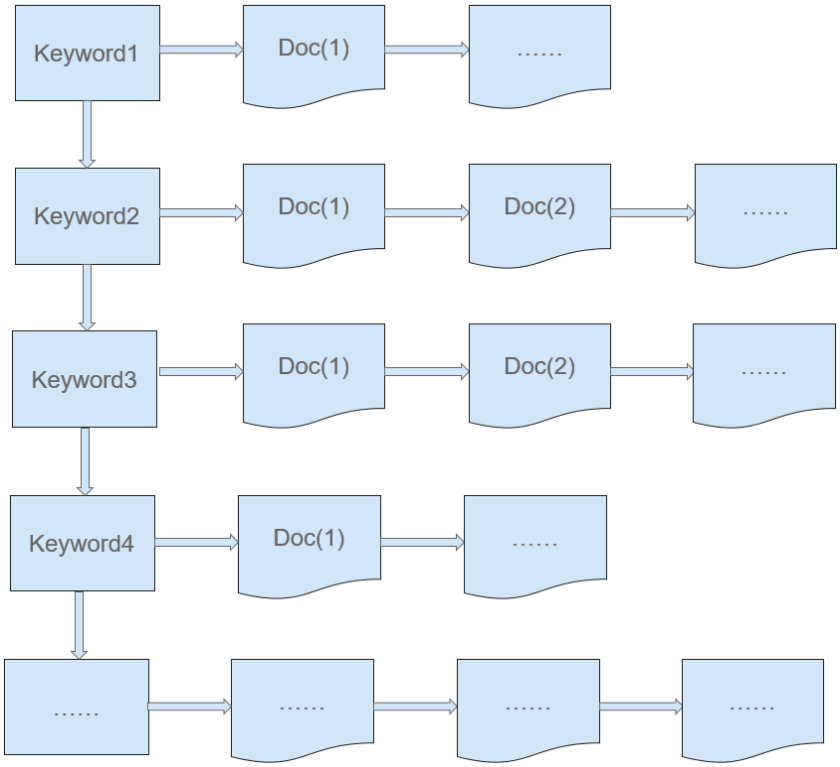
  The traditional search (which uses the ascending-order indexing, as shown in **Figure 5-129**) starts from keypoints and then uses the keypoints to find the specific information that meets the search criteria. In the traditional mode, values are found according to keys. During search based on the ascending-order indexing, keywords are found by document number.

  Figure 5-129 Ascending-order indexing

  The Solr (Lucene) search uses the descending-order indexing mode (as shown in **Figure 5-130**). In this mode, keys are found according to values. Values in the full-text search indicate the keywords that need to be searched. Places where the keywords are stored are called dictionaries. Keys indicate document number lists, with which users can find the documents that contain the search keywords (values), as shown in the following figure. During search based on the descending-order indexing, document numbers are found by keyword and then documents are found by document number.

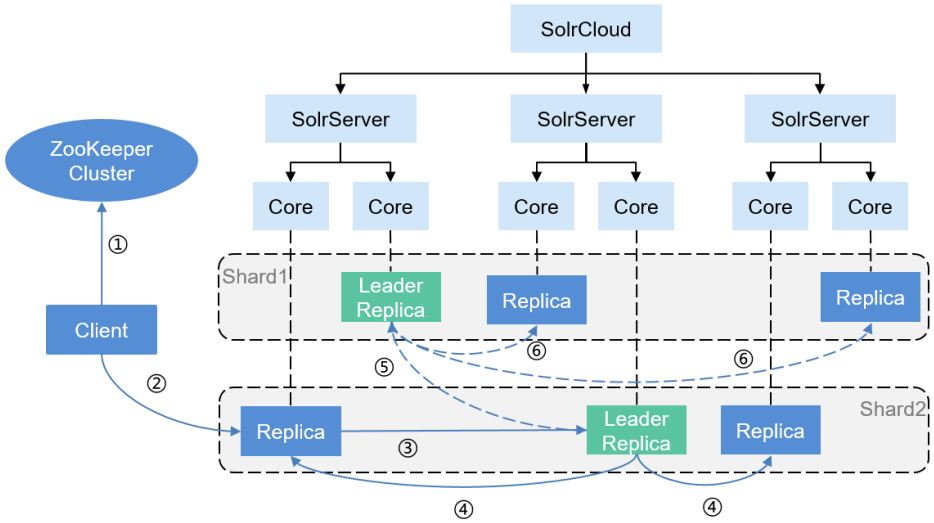
Figure 5-130 Descending-order indexing



- **Distributed Indexing Operation Procedure**

  Figure 5-131 describes the Solr distributed indexing operation procedure.

  Figure 5-131 Distributed indexing operation procedure



The procedure is as follows:

a. When initiating a document indexing request, the Client obtains the SolrServer cluster information of SolrCloud from the ZooKeeper cluster, and then obtains any SolrServer that contains the Collection information according to the Collection information in the request.

b. The Client sends the document indexing request to a Replica of the related Shard in the Collection of the SolrServer.

c. If the Replica is not the Leader Replica, the Replica will forward the document indexing request to the Leader Replica in the same Shard.

d. After indexing documents locally, the Leader Replica routes the document indexing request to other Replicas for processing.

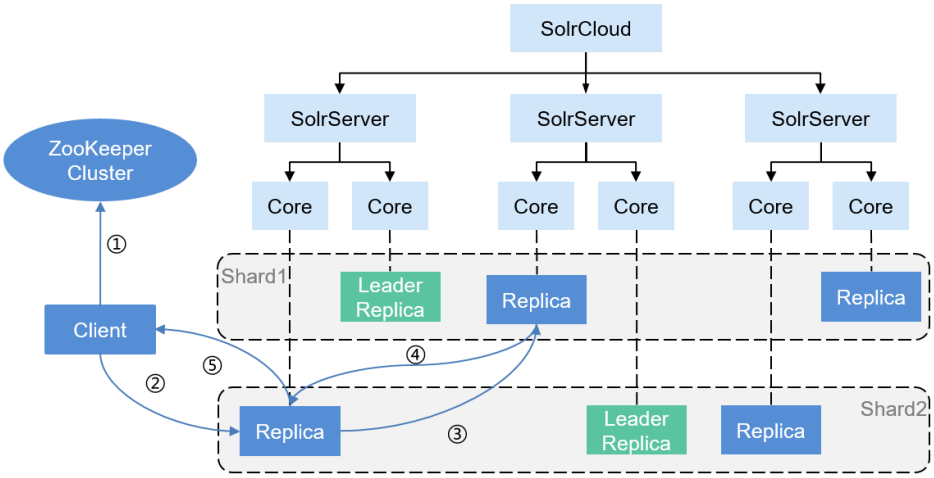
e. If the target Shard of the document indexing is not the Shard of this request, the Leader Replica of the Shard will forward the document indexing request to the Leader Replica of the target Shard.

f. After indexing documents locally, the Leader Replica of the target Shard routes the document indexing request to other Replicas of the Shard of the request for processing.

- **Distributed Search Operation Procedure**

  Figure 5-132 describes the Solr distributed search operation procedure.

  Figure 5-132 Distributed search operation procedure



The procedure is as follows:

a. When initiating a search request, the Client obtains the SolrServer cluster information using ZooKeeper and then randomly selects a SolrServer that contains the Collection.

b. The Client sends the search request to any Replica (which does not need to be the Leader Replica) of the related Shard in the Collection of the SolrServer for processing.

c. The Replica starts a distributed query, converts the query into multiple subqueries based on the number of Shards of the Collection (there are two Shards in Figure 5-132, Shard 1 and Shard 2), and distributes each subquery to any Replica (which does not need to be the Leader Replica) of the related Shard for processing.

d. After each subquery is completed, the query results are returned.

e. After receiving the results of each subquery, the Replicas that receives a query request for the first time combines the query results and then sends the final results to the Client.

5.34.2 Solr Relationship with Other Components

Relationship Between Solr and HDFS

Solr is a project of the Apache Software Foundation and a major component in the ecosystem of the Apache Hadoop project. Solr can use the Hadoop Distributed File System (HDFS) as its index file storage system. Solr is located on the structured storage layer. The HDFS provides highly reliable support for the storage of Solr. All index data files of Solr can be stored in the HDFS.

Relationship Between Solr and HBase

HBase stores massive data. It is a distributed column-oriented storage system built on the HDFS. Indexing for HBase data by Solr is the process of writing HBase data into the HDFS and creating indexes for HBase data. The index ID corresponds to the HBase data according to rowkey. Ensure that each piece of index data is unique and each piece of HBase data is unique, implementing full-text search for HBase data.

5.34.3 Solr Enhanced Open Source Features

Solr Enhanced Open Source Features

- Enhanced Reliability, Availability, and Security
  - The HA and floating IP address mechanisms are implemented, improving the reliability of Solr services.
  - Memory, CPUs, and disk I/Os of Solr instances are monitored, and shard status monitoring and alarms are implemented.
  - Provides the Kerberos authentication to ensure the index data security.
  - The authority control for collection operations, and access control for configuration sets on ZooKeeper are added.
- Multi-instance Deployment

  Five Solr instances can be deployed on each node. In addition, two SolrServerAdmin instances are provided to provide the web UI function.
- Two Replicas Distributed Across Nodes in Multi-instance Deployment Scenarios

  When multiple Solr instances are deployed on each node, during collection creation, two replicas are distributed on different nodes.
- Sensitive Word Filtering

  Sensitive words in query results are filtered.
- HBase Full Text Search
  - HBase Indexer is used to perform synchronous indexing on HBase data and full-text retrieval on HBase data.
  - The mapping between HBase tables and Solr indexes is created to provide a unified API for operating HBase and Solr (Luna). Indexes are stored in Solr and original data is stored in HBase.

5.35 Apache Spark

5.35.1 Spark Basic Principles

Description

Spark is a memory-based distributed computing framework. In iterative computation scenarios, the computing capability of Spark is 10 to 100 times higher than MapReduce, because data is stored in memory when being processed. Spark can use HDFS as the underlying storage system, enabling users to quickly switch to Spark from MapReduce. Spark provides one-stop data analysis capabilities, such as the streaming processing in small batches, offline batch processing, SQL query, and data mining. Users can seamlessly use these functions in a same application. For details about the new open source features of Spark, see **Spark Open Source New Features**.

Features of Spark are as follows:

- Improves the data processing capability through distributed memory computing and directed acyclic graph (DAG) execution engine. The delivered performance is 10 to 100 times higher than that of MapReduce.
- Supports multiple development languages (Scala/Java/Python) and dozens of highly abstract operators to facilitate the construction of distributed data processing applications.
- Builds data processing stacks using **SQL**, **Streaming**, MLlib, and GraphX to provide one-stop data processing capabilities.
- Fits into the Hadoop ecosystem, allowing Spark applications to run on Standalone, Mesos, or Yarn, enabling access of multiple data sources such as HDFS, HBase, and Hive, and supporting smooth migration of the MapReduce application to Spark.

Architecture

Figure 5-133 describes the Spark architecture and **Table 5-28** lists the Spark modules.

Figure 5-133 Spark architecture

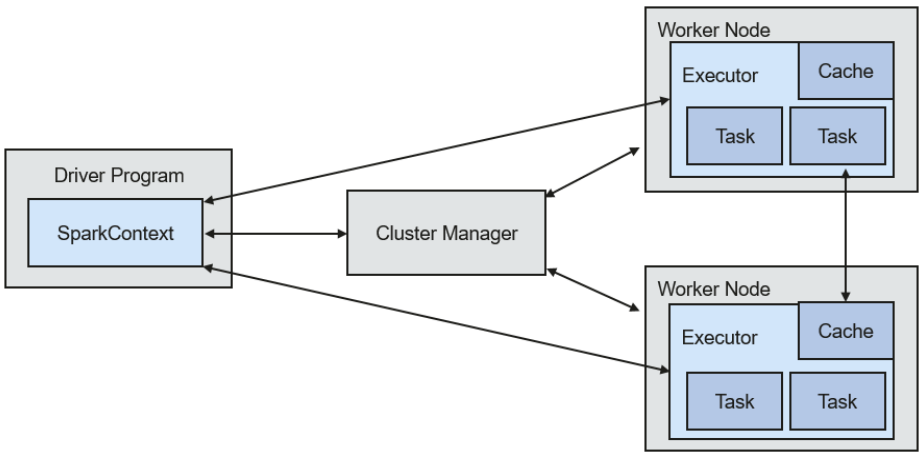


Table 5-28 Basic concepts

Module	Description
Cluster Manager	Cluster manager manages resources in the cluster. Spark supports multiple cluster managers, including Mesos, Yarn, and the Standalone cluster manager that is delivered with Spark. By default, Spark clusters adopt the Yarn cluster manager.
Application	Spark application. It consists of one Driver Program and multiple executors.
Deploy Mode	Deployment in cluster or client mode. In cluster mode, the driver runs on a node inside the cluster. In client mode, the driver runs on the client (outside the cluster).
Driver Program	The main process of the Spark application. It runs the **main()** function of an application and creates SparkContext. It is used for parsing applications, generating stages, and scheduling tasks to executors. Usually, SparkContext represents Driver Program.
Executor	A process started on a Work Node. It is used to execute tasks, and manage and process the data used in applications. A Spark application usually contains multiple executors. Each executor receives commands from the driver and executes one or multiple tasks.
Worker Node	A node that starts and manages executors and resources in a cluster.
Job	A job consists of multiple concurrent tasks. One action operator (for example, a collect operator) maps to one job.
Stage	Each job consists of multiple stages. Each stage is a task set, which is separated by Directed Acyclic Graph (DAG).
Task	A task carries the computation unit of the service logics. It is the minimum working unit that can be executed on the Spark platform. An application can be divided into multiple tasks based on the execution plan and computation amount.

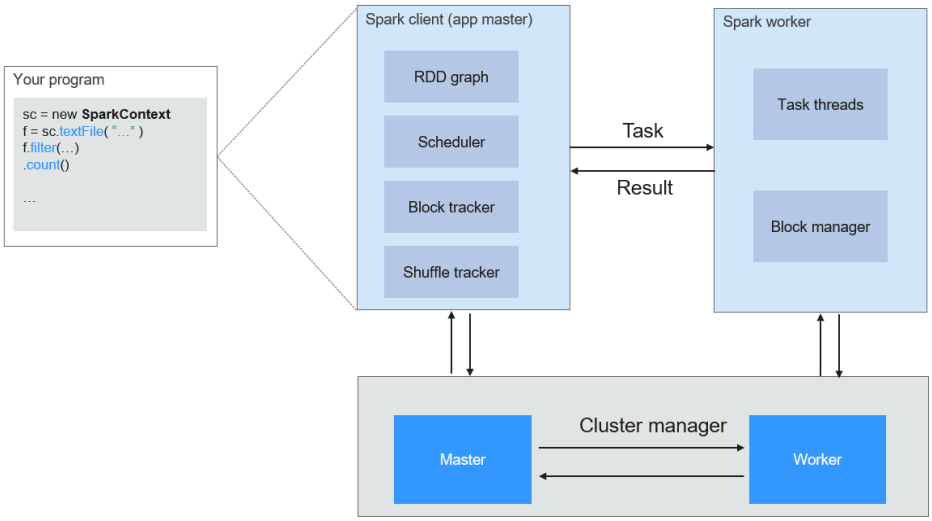
Spark Principles

Figure 5-134 describes the application running architecture of Spark.

1. An application is running in the cluster as a collection of processes. Driver coordinates the running of the application.
2. To run an application, Driver connects to the cluster manager (such as Standalone, Mesos, and Yarn) to apply for the executor resources, and start ExecutorBackend. The cluster manager schedules resources between different applications. Driver schedules DAGs, divides stages, and generates tasks for the application at the same time.

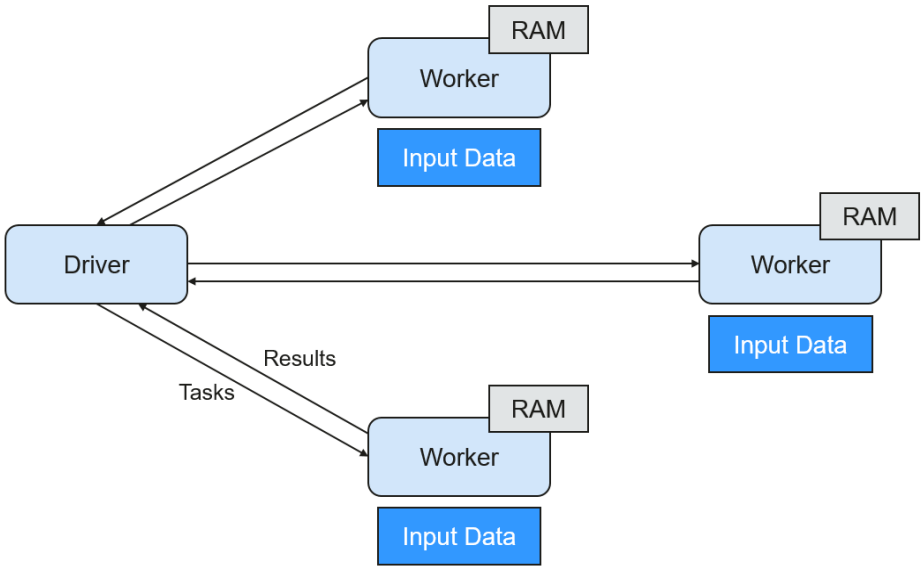
3. Then, Spark sends the codes of the application (the codes transferred to **SparkContext**, which is defined by JAR or Python) to an executor.
4. After all tasks are finished, the running of the user application is stopped.

Figure 5-134 Spark application running architecture



Spark uses Master and Worker modes, as shown in **Figure 5-135**. A user submits an application on the Spark client, and then the scheduler divides a job into multiple tasks and sends the tasks to each Worker for execution. Each Worker reports the computation results to Driver (Master), and then the Driver aggregates and returns the results to the client.

Figure 5-135 Spark Master-Worker mode



Note the following about the architecture:

- Applications are isolated from each other.

  Each application has an independent executor process, and each executor starts multiple threads to execute tasks in parallel. Each driver schedules its

own tasks, and different application tasks run on different JVMs, that is, different executors.

- Different Spark applications do not share data, unless data is stored in the external storage system such as HDFS.
- You are advised to deploy the Driver program in a location that is close to the Worker node because the Driver program schedules tasks in the cluster. For example, deploy the Driver program on the network where the Worker node is located.

Spark on YARN can be deployed in two modes:

- In Yarn-cluster mode, the Spark driver runs inside an ApplicationMaster process which is managed by Yarn in the cluster. After the ApplicationMaster is started, the client can exit without interrupting service running.
- In Yarn-client mode, Driver runs in the client process, and the ApplicationMaster process is used only to apply for requesting resources from Yarn.

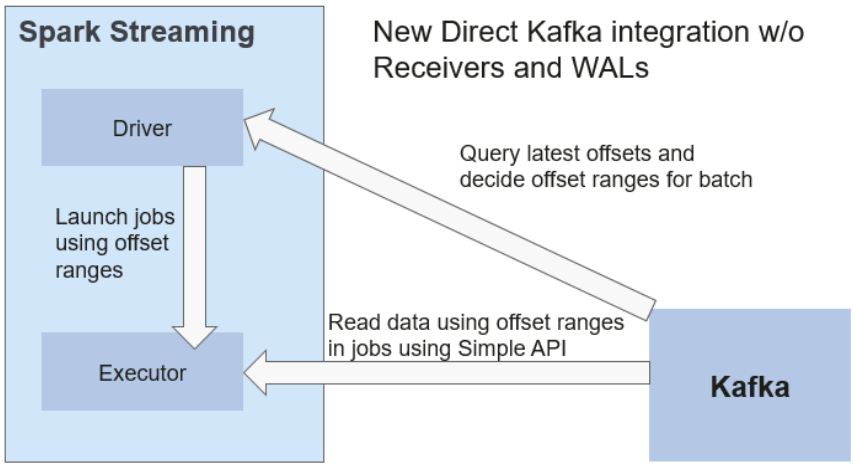
Spark Streaming Principles

Spark Streaming is a real-time computing framework built on the Spark, which expands the capability for processing massive streaming data. Spark supports two data processing approaches: Direct Streaming and Receiver.

Direct Streaming computing process

In Direct Streaming approach, Direct API is used to process data. Take Kafka Direct API as an example. Direct API provides offset location that each batch range will read from, which is much simpler than starting a receiver to continuously receive data from Kafka and written data to write-ahead logs (WALs). Then, each batch job is running and the corresponding offset data is ready in Kafka. These offset information can be securely stored in the checkpoint file and read by applications that failed to start.

Figure 5-136 Data transmission through Direct Kafka API



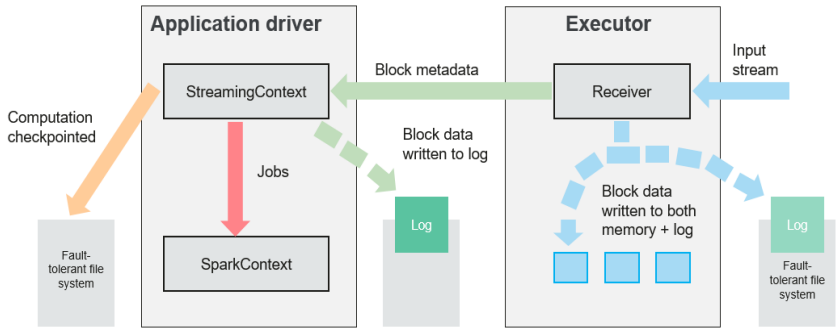
After the failure, Spark Streaming can read data from Kafka again and process the data segment. The processing result is the same no matter Spark Streaming fails or not, because the semantic is processed only once.

Direct API does not need to use the WAL and Receivers, and ensures that each Kafka record is received only once, which is more efficient. In this way, the Spark Streaming and Kafka can be well integrated, making streaming channels be featured with high fault-tolerance, high efficiency, and ease-of-use. Therefore, you are advised to use Direct Streaming to process data.

Receiver computing process

When a Spark Streaming application starts (that is, when the driver starts), the related StreamingContext (the basis of all streaming functions) uses SparkContext to start the receiver to become a long-term running task. These receivers receive and save streaming data to the Spark memory for processing. **Figure 5-137** shows the data transfer lifecycle.

Figure 5-137 Data transfer lifecycle



1. Receive data (blue arrow).

   Receiver divides a data stream into a series of blocks and stores them in the executor memory. In addition, after WAL is enabled, it writes data to the WAL of the fault-tolerant file system.

2. Notify the driver (green arrow).

   The metadata in the received block is sent to StreamingContext in the driver. The metadata includes:

   - Block reference ID used to locate the data position in the Executor memory.
   - Block data offset information in logs (if the WAL function is enabled).

3. Process data (red arrow).

   For each batch of data, StreamingContext uses block information to generate resilient distributed datasets (RDDs) and jobs. StreamingContext executes jobs by running tasks to process blocks in the executor memory.

4. Periodically set checkpoints (orange arrows).
5. For fault tolerance, StreamingContext periodically sets checkpoints and saves them to external file systems.

Fault Tolerance

Spark and its RDD allow seamless processing of failures of any Worker node in the cluster. Spark Streaming is built on top of Spark. Therefore, the Worker node of Spark Streaming also has the same fault tolerance capability. However, Spark Streaming needs to run properly in case of long-time running. Therefore, Spark must be able to recover from faults through the driver process (main process that

coordinates all Workers). This poses challenges to the Spark driver fault-tolerance because the Spark driver may be any user application implemented in any computation mode. However, Spark Streaming has internal computation architecture. That is, it periodically executes the same Spark computation in each batch data. Such architecture allows it to periodically store checkpoints to reliable storage space and recover them upon the restart of Driver.

For source data such as files, the Driver recovery mechanism can ensure zero data loss because all data is stored in a fault-tolerant file system such as HDFS. However, for other data sources such as Kafka and Flume, some received data is cached only in memory and may be lost before being processed. This is caused by the distribution operation mode of Spark applications. When the driver process fails, all executors running in the Cluster Manager, together with all data in the memory, are terminated. To avoid such data loss, the WAL function is added to Spark Streaming.

WAL is often used in databases and file systems to ensure persistence of any data operation. That is, first record an operation to a persistent log and perform this operation on data. If the operation fails, the system is recovered by reading the log and re-applying the preset operation. The following describes how to use WAL to ensure persistence of received data:

Receiver is used to receive data from data sources such as Kafka. As a long-time running task in Executor, Receiver receives data, and also confirms received data if supported by data sources. Received data is stored in the Executor memory, and Driver delivers a task to Executor for processing.

After WAL is enabled, all received data is stored to log files in the fault-tolerant file system. Therefore, the received data does not lose even if Spark Streaming fails. Besides, receiver checks correctness of received data only after the data is pre-written into logs. Data that is cached but not stored can be sent again by data sources after the driver restarts. These two mechanisms ensure zero data loss. That is, all data is recovered from logs or re-sent by data sources.

To enable the WAL function, perform the following operations:

- Set **streamingContext.checkpoint** (path-to-directory) to configure the checkpoint directory, which is an HDFS file path used to store streaming checkpoints and WALs.
- Set **spark.streaming.receiver.writeAheadLog.enable** of SparkConf to **true** (the default value is **false**).

After WAL is enabled, all receivers have the advantage of recovering from reliable received data. You are advised to disable the multi-replica mechanism because the fault-tolerant file system of WAL may also replicate the data.

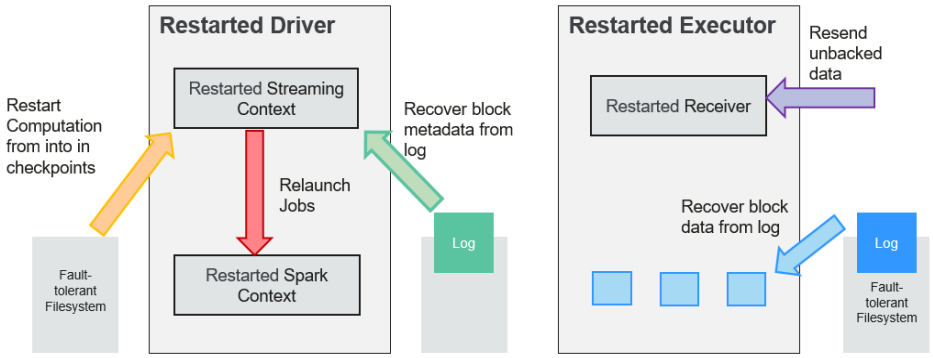
NOTE

The data receiving throughput is lowered after WAL is enabled. All data is written into the fault-tolerant file system. As a result, the write throughput of the file system and the network bandwidth for data replication may become the potential bottleneck. To solve this problem, you are advised to create more receivers to increase the degree of data receiving parallelism or use better hardware to improve the throughput of the fault-tolerant file system.

Recovery Process

When a failed driver is restarted, restart it as follows:

Figure 5-138 Computing recovery process



1. Recover computing. (Orange arrow)

   Use checkpoint information to restart Driver, reconstruct SparkContext and restart Receiver.

2. Recover metadata block. (Green arrow)

   This operation ensures that all necessary metadata blocks are recovered to continue the subsequent computing recovery.

3. Relaunch unfinished jobs. (Red arrow)

   Recovered metadata is used to generate RDDs and corresponding jobs for interrupted batch processing due to failures.

4. Read block data saved in logs. (Blue arrow)

   Block data is directly read from WALs during execution of the preceding jobs, and therefore all essential data reliably stored in logs is recovered.

5. Resend unconfirmed data. (Purple arrow)

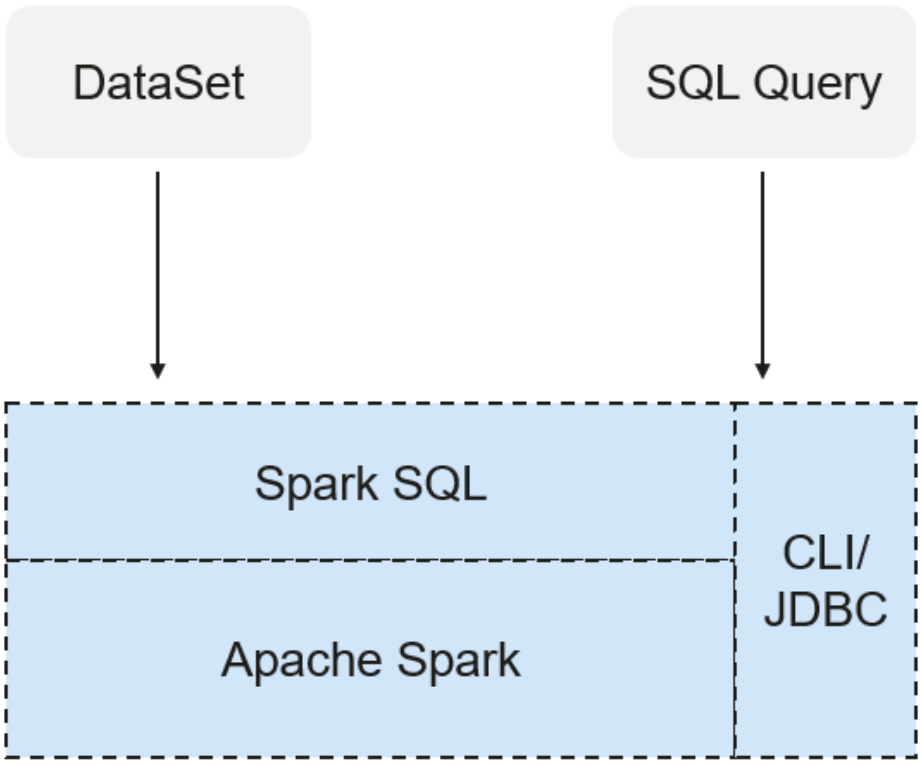
   Data that is cached but not stored to logs upon failures is re-sent by data sources, because the receiver does not confirm the data.

Therefore, by using WALs and reliable Receiver, Spark Streaming can avoid input data loss caused by Driver failures.

SparkSQL and DataSet Principle

SparkSQL

Figure 5-139 SparkSQL and DataSet



Spark SQL is a module for processing structured data. In Spark application, SQL statements or DataSet APIs can be seamlessly used for querying structured data.

Spark SQL and DataSet also provide a universal method for accessing multiple data sources such as Hive, CSV, Parquet, ORC, JSON, and JDBC. These data sources also allow data interaction. Spark SQL reuses the Hive frontend processing logic and metadata processing module. With the Spark SQL, you can directly query existing Hive data.

In addition, Spark SQL also provides API, CLI, and JDBC APIs, allowing diverse accesses to the client.

Spark SQL Native DDL/DML

In Spark 1.5, lots of Data Definition Language (DDL)/Data Manipulation Language (DML) commands are pushed down to and run on the Hive, causing coupling with the Hive and inflexibility such as unexpected error reports and results.

Spark realizes command localization and replaces Hive with Spark SQL Native DDL/DML to run DDL/DML commands. Additionally, the decoupling from the Hive is realized and commands can be customized.

DataSet

A DataSet is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each Dataset also has an untyped view called a DataFrame, which is a Dataset of Row.

The DataFrame is a structured and distributed dataset consisting of multiple columns. The DataFrame is equal to a table in the relationship database or the

DataFrame in the R/Python. The DataFrame is the most basic concept in the Spark SQL, which can be created by using multiple methods, such as the structured dataset, Hive table, external database or RDD.

Operations available on DataSets are divided into transformations and actions.

- A transformation operation can generate a new DataSet, for example, **map**, **filter**, **select**, and **aggregate (groupBy)**.
- An action operation can trigger computation and return results, for example, **count**, **show**, or write data to the file system.

You can use either of the following methods to create a DataSet:

- The most common way is by pointing Spark to some files on storage systems, using the **read** function available on a SparkSession.
  ```
  val people = spark.read.parquet("...").as[Person]  // Scala
  DataSet<Person> people = spark.read().parquet("...").as(Encoders.bean(Person.class));//Java
  ```
- You can also create a DataSet using the transformation operation available on an existing one. For example, apply the map operation on an existing DataSet to create a DataSet:
  ```
  val names = people.map(_.name) // In Scala: names is Dataset.
  Dataset<String> names = people.map((Person p) -> p.name, Encoders.STRING)); // Java
  ```

CLI and JDBCServer

In addition to programming APIs, Spark SQL also provides the CLI/JDBC APIs.

- Both **spark-shell** and **spark-sql** scripts can provide the CLI for debugging.
- JDBCServer provides JDBC APIs. External systems can directly send JDBC requests to calculate and parse structured data.

SparkSession Principle

SparkSession is a unified API in Spark and can be regarded as a unified entry for reading data. SparkSession provides a single entry point to perform many operations that were previously scattered across multiple classes, and also provides accessor methods to these older classes to maximize compatibility.

A SparkSession can be created using a builder pattern. The builder will automatically reuse the existing SparkSession if there is a SparkSession; or create a SparkSession if it does not exist. During I/O transactions, the configuration item settings in the builder are automatically synchronized to Spark and Hadoop.

```
import org.apache.spark.sql.SparkSession
val sparkSession = SparkSession.builder
  .master("local")
  .appName("my-spark-app")
  .config("spark.some.config.option", "config-value")
  .getOrCreate()
```

- SparkSession can be used to execute SQL queries on data and return results as DataFrame.
  ```
  sparkSession.sql("select * from person").show
  ```
- SparkSession can be used to set configuration items during running. These configuration items can be replaced with variables in SQL statements.
  ```
  sparkSession.conf.set("spark.some.config", "abcd")
  sparkSession.conf.get("spark.some.config")
  sparkSession.sql("select ${spark.some.config}")
  ```


- SparkSession also includes a "catalog" method that contains methods to work with Metastore (data catalog). After this method is used, a dataset is returned, which can be run using the same Dataset API.

```
val tables = sparkSession.catalog.listTables()
val columns = sparkSession.catalog.listColumns("myTable")
```

- Underlying SparkContext can be accessed by SparkContext API of SparkSession.

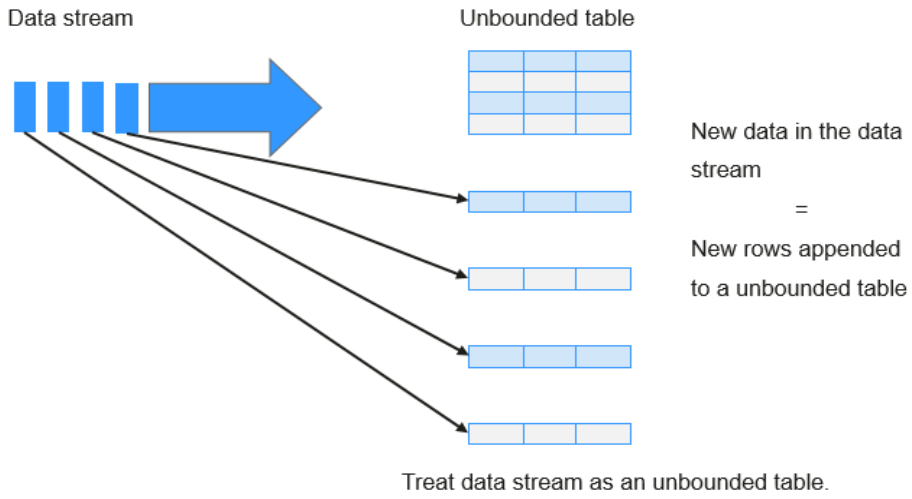
```
val sparkContext = sparkSession.sparkContext
```

Structured Streaming Principles

Structured Streaming is a stream processing engine built on the Spark SQL engine. You can use the Dataset/DataFrame API in Scala, Java, Python, or R to express streaming aggregations, event-time windows, and stream-stream joins. If streaming data is incrementally and continuously produced, Spark SQL will continue to process the data and synchronize the result to the result set. In addition, the system ensures end-to-end exactly-once fault-tolerance guarantees through checkpoints and WALs.

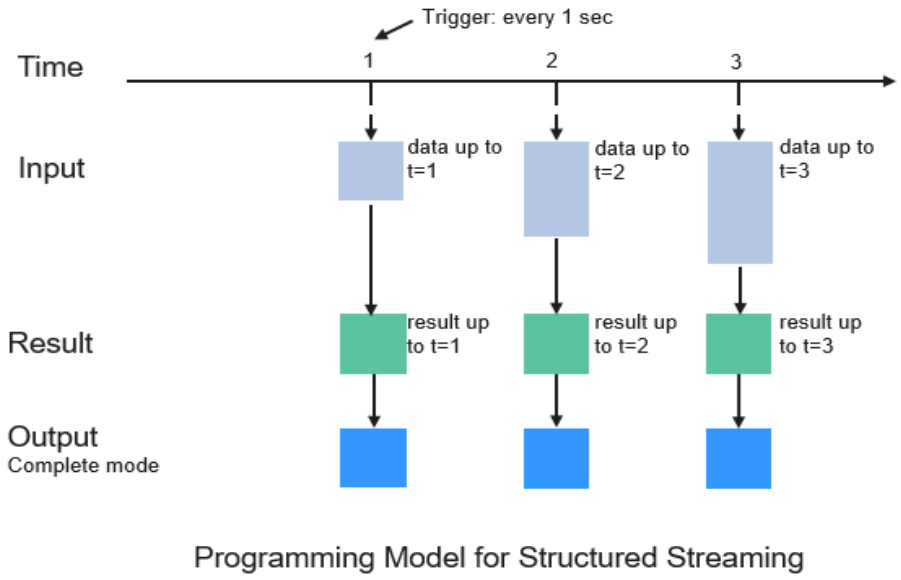
The core of Structured Streaming is to take streaming data as an incremental database table. Similar to the data block processing model, the streaming data processing model applies query operations on a static database table to streaming computing, and Spark uses standard SQL statements for query, to obtain data from the incremental and unbounded table.

Figure 5-140 Unbounded table of Structured Streaming



Each query operation will generate a result table. At each trigger interval, updated data will be synchronized to the result table. Whenever the result table is updated, the updated result will be written into an external storage system.

Figure 5-141 Structured Streaming data processing model



Storage modes of Structured Streaming at the output phase are as follows:

- Complete Mode: The updated result sets are written into the external storage system. The write operation is performed by a connector of the external storage system.
- Append Mode: If an interval is triggered, only added data in the result table will be written into an external system. This is applicable only on the queries where existing rows in the result table are not expected to change.
- Update Mode: If an interval is triggered, only updated data in the result table will be written into an external system, which is the difference between the Complete Mode and Update Mode.

Concepts

- **RDD**

  Resilient Distributed Dataset (RDD) is a core concept of Spark. It indicates a read-only and partitioned distributed dataset. Partial or all data of this dataset can be cached in the memory and reused between computations.

  RDD Creation

  - An RDD can be created from the input of HDFS or other storage systems that are compatible with Hadoop.
  - A new RDD can be converted from a parent RDD.
  - An RDD can be converted from a collection of datasets through encoding.

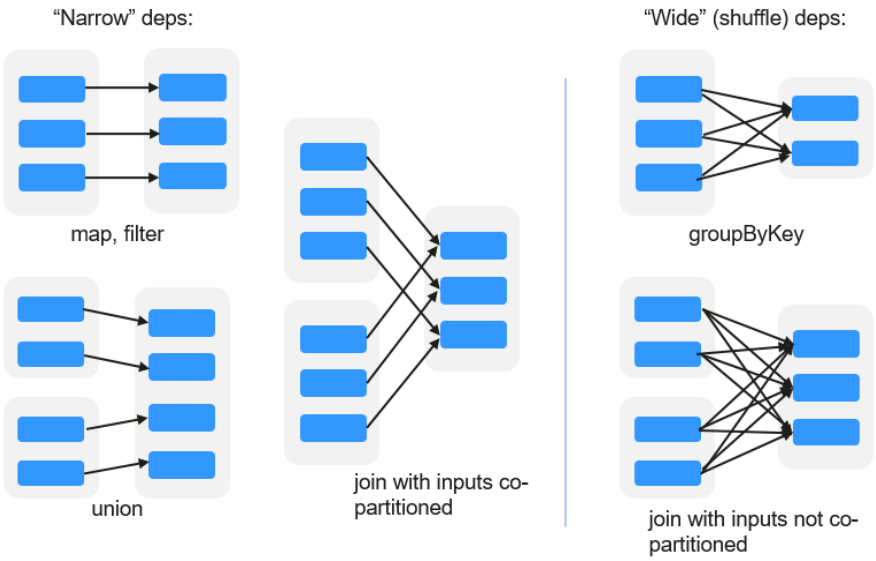
  RDD Storage

  - You can select different storage levels to store an RDD for reuse. (There are 11 storage levels to store an RDD.)
  - By default, the RDD is stored in the memory. When the memory is insufficient, the RDD overflows to the disk.

- **RDD Dependency**

The RDD dependency includes the narrow dependency and wide dependency.

Figure 5-142 RDD dependency



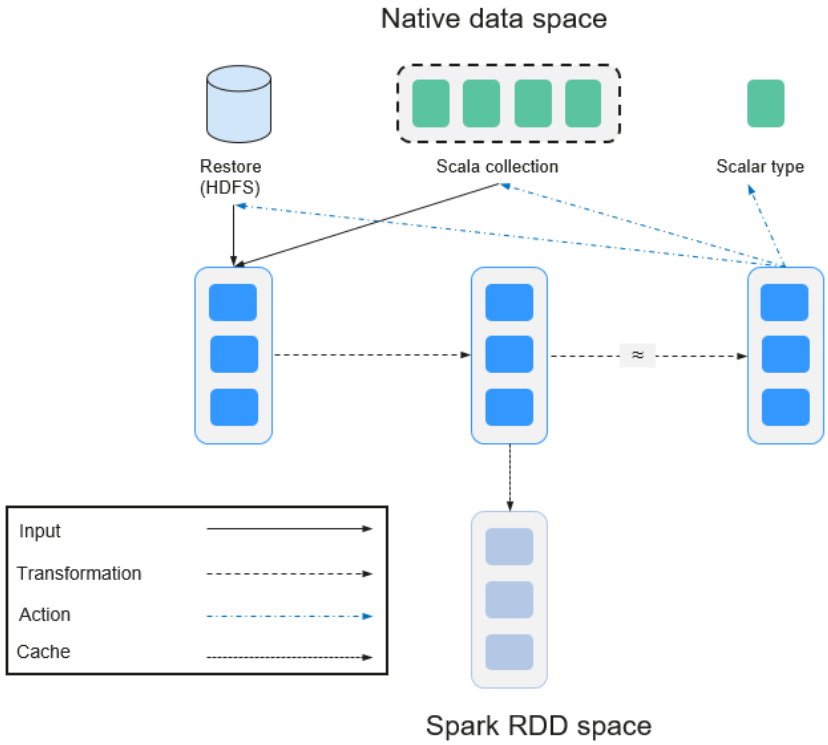
- **Narrow dependency**: Each partition of the parent RDD is used by at most one partition of the child RDD.
- **Wide dependency**: Partitions of the child RDD depend on all partitions of the parent RDD.

The narrow dependency facilitates the optimization. Logically, each RDD operator is a fork/join (the join is not the join operator mentioned above but the barrier used to synchronize multiple concurrent tasks); fork the RDD to each partition, and then perform the computation. After the computation, join the results, and then perform the fork/join operation on the next RDD operator. It is uneconomical to directly translate the RDD into physical implementation. The first is that every RDD (even intermediate result) needs to be physicalized into memory or storage, which is time-consuming and occupies much space. The second is that as a global barrier, the join operation is very expensive and the entire join process will be slowed down by the slowest node. If the partitions of the child RDD narrowly depend on that of the parent RDD, the two fork/join processes can be combined to implement classic fusion optimization. If the relationship in the continuous operator sequence is narrow dependency, multiple fork/join processes can be combined to reduce a large number of global barriers and eliminate the physicalization of many RDD intermediate results, which greatly improves the performance. This is called pipeline optimization in Spark.

- **Transformation and Action (RDD Operations)**

  Operations on RDD include transformation (the return value is an RDD) and action (the return value is not an RDD). **Figure 5-143** shows the RDD operation process. The transformation is lazy, which indicates that the transformation from one RDD to another RDD is not immediately executed. Spark only records the transformation but does not execute it immediately. The real computation is started only when the action is started. The action returns results or writes the RDD data into the storage system. The action is the driving force for Spark to start the computation.

Figure 5-143 RDD operation



The data and operation model of RDD are quite different from those of Scala.

```
val file = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))
errors.cache()
errors.count()
```

a. The textFile operator reads log files from the HDFS and returns files (as an RDD).

b. The filter operator filters rows with **ERROR** and assigns them to errors (a new RDD). The filter operator is a transformation.

c. The cache operator caches errors for future use.

d. The count operator returns the number of rows of errors. The count operator is an action.

Transformation includes the following types:

– The RDD elements are regarded as simple elements.

  The input and output has the one-to-one relationship, and the partition structure of the result RDD remains unchanged, for example, map.

  The input and output has the one-to-many relationship, and the partition structure of the result RDD remains unchanged, for example, flatMap (one element becomes a sequence containing multiple elements after map and then flattens to multiple elements).

  The input and output has the one-to-one relationship, but the partition structure of the result RDD changes, for example, union (two RDDs integrates to one RDD, and the number of partitions becomes the sum of the number of partitions of two RDDs) and coalesce (partitions are reduced).

Operators of some elements are selected from the input, such as filter, distinct (duplicate elements are deleted), subtract (elements only exist in this RDD are retained), and sample (samples are taken).

- The RDD elements are regarded as key-value pairs.

  Perform the one-to-one calculation on the single RDD, such as mapValues (the partition mode of the source RDD is retained, which is different from map).

  Sort the single RDD, such as sort and partitionBy (partitioning with consistency, which is important to the local optimization).

  Restructure and reduce the single RDD based on key, such as groupByKey and reduceByKey.

  Join and restructure two RDDs based on the key, such as join and cogroup.

  NOTE

  The later three operations involving sorting are called shuffle operations.

Action includes the following types:

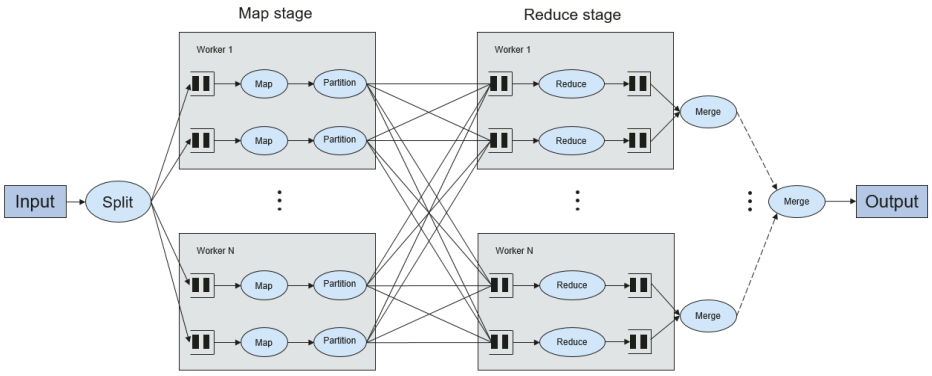
- Generate scalar configuration items, such as **count** (the number of elements in the returned RDD), **reduce**, **fold/aggregate** (the number of scalar configuration items that are returned), and **take** (the number of elements before the return).
- Generate the Scala collection, such as **collect** (import all elements in the RDD to the Scala collection) and **lookup** (look up all values corresponds to the key).
- Write data to the storage, such as **saveAsTextFile** (which corresponds to the preceding **textFile**).
- Check points, such as the **checkpoint** operator. When Lineage is quite long (which occurs frequently in graphics computation), it takes a long period of time to execute the whole sequence again when a fault occurs. In this case, checkpoint is used as the check point to write the current data to stable storage.

- **Shuffle**

  Shuffle is a specific phase in the MapReduce framework, which is located between the Map phase and the Reduce phase. If the output results of Map are to be used by Reduce, the output results must be hashed based on a key and distributed to each Reducer. This process is called Shuffle. Shuffle involves the read and write of the disk and the transmission of the network, so that the performance of Shuffle directly affects the operation efficiency of the entire program.

  The figure below shows the entire process of the MapReduce algorithm.

Figure 5-144 Algorithm process



Shuffle is a bridge connecting data. The following describes the implementation of shuffle in Spark.

Shuffle divides a job of Spark into multiple stages. The former stages contain one or more ShuffleMapTasks, and the last stage contains one or more ResultTasks.

- **Spark Application Structure**

  The Spark application structure includes the initialized SparkContext and the main program.

  - Initialized SparkContext: constructs the operating environment of the Spark Application.

    Constructs the SparkContext object. The following is an example:

    ```
    new SparkContext(master, appName, [SparkHome], [jars])
    ```

    Parameter description:

    master: indicates the link string. The link modes include local, Yarn-cluster, and Yarn-client.

    appName: indicates the application name.

    SparkHome: indicates the directory where Spark is installed in the cluster.

    jars: indicates the code and dependency package of an application.

  - Main program: processes data.

  For details about how to submit an application, visit **https://spark.apache.org/docs/3.3.1/submitting-applications.html**.

- **Spark Shell Commands**

  The basic Spark shell commands support the submission of Spark applications. The Spark shell commands are as follows:

  ```
  ./bin/spark-submit \
    --class <main-class> \
    --master <master-url> \
    ... # other options
    <application-jar> \
    [application-arguments]
  ```

  Parameter description:

  --class: indicates the name of the class of a Spark application.

  --master: indicates the master to which the Spark application links, such as Yarn-client and Yarn-cluster.

  application-jar: indicates the path of the JAR file of the Spark application.

application-arguments: indicates the parameter required to submit the Spark application. This parameter can be left blank.

- **Spark JobHistory Server**

  The Spark web UI is used to monitor the details in each phase of the Spark framework of a running or historical Spark job and provide the log display, which helps users to develop, configure, and optimize the job in more fine-grained units.

5.35.2 Spark HA Solution

5.35.2.1 Spark Multi-Active Instance

Context

Based on existing JDBCServers in the community, multi-active-instance HA is used to achieve the high availability. In this mode, multiple JDBCServers coexist in the cluster and the client can randomly connect any JDBCServer to perform service operations. When one or multiple JDBCServers stop working, a client can connect to another normal JDBCServer.

Compared with active/standby HA, multi-active instance HA eliminates the following restrictions:

- In active/standby HA, when the active/standby switchover occurs, the unavailable period cannot be controlled by JDBCServer, but determined by Yarn service resources.
- In Spark, the Thrift JDBC similar to HiveServer2 provides services and users access services through Beeline and JDBC API. Therefore, the processing capability of the JDBCServer cluster depends on the single-point capability of the primary server, and the scalability is insufficient.

Multi-active instance HA not only prevents service interruption caused by switchover, but also enables cluster scale-out to secure high concurrency.

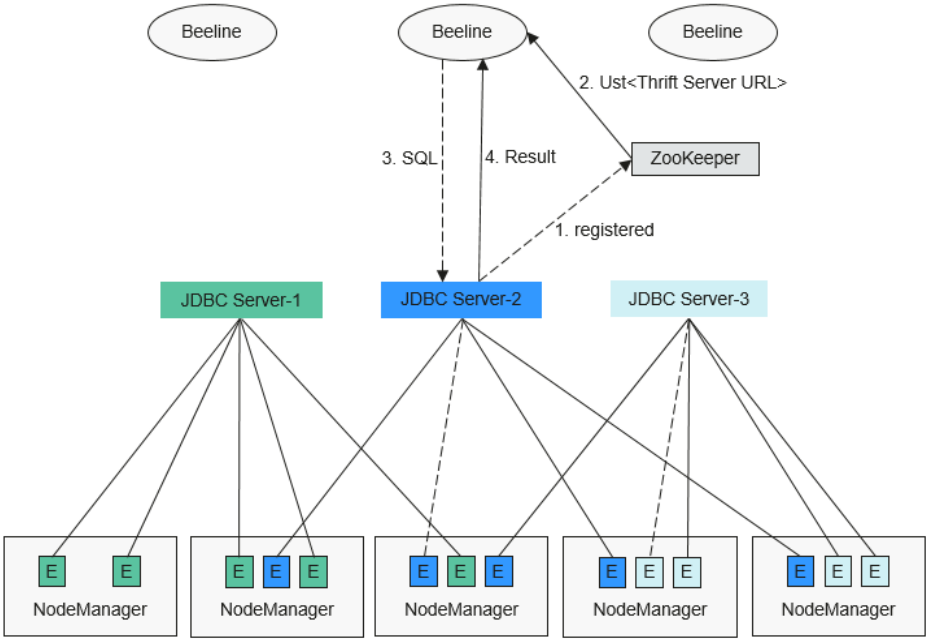
Scenario

When one or more JDBCServer services in a cluster are abnormal, users can automatically connect to other normal JDBCServer services without affecting service running.

Implementation

The following figure shows the basic principle of multi-active instance HA of Spark JDBCServer.

Figure 5-145 Spark JDBCServer HA



1. After JDBCServer is started, it registers with ZooKeeper by writing node information in a specified directory. Node information includes the JDBCServer instance IP, port number, version, and serial number (information of different nodes is separated by commas).

   An example is provided as follows:

   ```
   [serverUri=192.168.169.84:22550
   ;version=8.3.1;sequence=0000001244,serverUri=192.168.195.232:22550 ;version=8.3.1;sequence=00000
   01242,serverUri=192.168.81.37:22550 ;version=8.3.1;sequence=0000001243]
   ```

2. To connect to JDBCServer, the client must specify the namespace, which is the directory of JDBCServer instances in ZooKeeper. During the connection, a JDBCServer instance is randomly selected from the specified namespace. For details about URL, see **URL Connection**.
3. After the connection succeeds, the client sends SQL statements to JDBCServer.
4. JDBCServer executes received SQL statements and sends results back to the client.

In multi-active instance HA mode, all JDBCServer instances are independent and equivalent. When one instance is interrupted during upgrade, other JDBCServer instances can accept the connection request from the client.

Following rules must be followed in the multi-active instance HA of Spark JDBCServer:

- If a JDBCServer instance exits abnormally, no other instance will take over the sessions and services running on this abnormal instance.
- When the JDBCServer process is stopped, corresponding nodes are deleted from ZooKeeper.
- The client randomly selects the server, which may result in uneven session allocation, and finally result in imbalance of instance load.

- After the instance enters the maintenance mode (in which no new connection request from the client is accepted), services still running on the instance may fail when the decommissioning times out.

URL Connection

Multi-active instance mode

In multi-active instance mode, the client reads content from the ZooKeeper node and connects to JDBCServer. The connection strings are as follows:

- Security mode:
  - If Kinit authentication is enabled, the JDBCURL is as follows:
    jdbc:hive2://*<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>*/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=**sparkthriftserver**;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.*<System domain name>*@*<System domain name>*;

    NOTE

    - **<zkNode_IP>:<zkNode_Port>** indicates the ZooKeeper URL. Use commas (,) to separate multiple URLs,

      For example, **192.168.81.37:24002,192.168.195.232:24002,192.168.169.84:24002**.
    - **sparkthriftserver** indicates the directory in ZooKeeper, where a random JDBCServer instance is connected to the client.

    For example, when you use Beeline client for connection in security mode, run the following command:

    sh *CLIENT_HOME*/**spark/bin/beeline -u "jdbc:hive2://***<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>***/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.<System domain name>@<System domain name>;"**
  - If Keytab authentication is enabled, the JDBCURL is as follows:
    jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.*<System domain name>*@*<System domain name>*;user.principal=<principal_name>;user.keytab=<path_to_keytab>

    <principal_name> indicates the principal of Kerberos user, for example, **test@***<System domain name>*. *<path_to_keytab>* indicates the Keytab file path corresponding to *<principal_name>*, for example, **/opt/auth/test/user.keytab**.
- Common mode:
  jdbc:hive2://*<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>*/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=**sparkthriftserver**;

  For example, when you use Beeline client for connection in common mode, run the following command:

  sh *CLIENT_HOME*/**spark/bin/beeline -u "jdbc:hive2://***<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>***/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver;"**

Non-multi-active instance mode

In non-multi-active instance mode, a client connects to a specified JDBCServer node. Compared with multi-active instance mode, the connection string in non-multi-active instance mode does not contain **serviceDiscoveryMode** and **zooKeeperNamespace** parameters about ZooKeeper.

For example, when you use Beeline client to connect JDBCServer in non-multi-active instance mode, run the following command:

sh *CLIENT_HOME*/**spark/bin/beeline -u "jdbc:hive2://** *<server_IP>:<server_Port>*/**;user.principal=spark2x/hadoop.*<System domain name>*@*<System domain name>*;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.*<System domain name>*@*<System domain name>*;"**

NOTE

- **<server_IP>:<server_Port>** indicates the URL of the specified JDBCServer node.
- **CLIENT_HOME** indicates the client path.

Except the connection method, operations of JDBCServer API in multi-active instance mode and non-multi-active instance mode are the same. Spark JDBCServer is another implementation of HiveServer2 in Hive. For details about how to use Spark JDBCServer, visit the official Hive website at **https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients**.

5.35.2.2 Spark Multi-Tenancy

Background

In the JDBCServer multi-active instance mode, JDBCServer implements the Yarn-client mode but only one Yarn resource queue is available. To solve the resource limitation problem, the multi-tenant mode is introduced.

In multi-tenant mode, JDBCServers are bound with tenants. Each tenant corresponds to one or more JDBCServers, and a JDBCServer provides services for only one tenant. Different tenants can be configured with different Yarn queues to implement resource isolation. In addition, JDBCServer can be dynamically started as required to avoid resource waste.

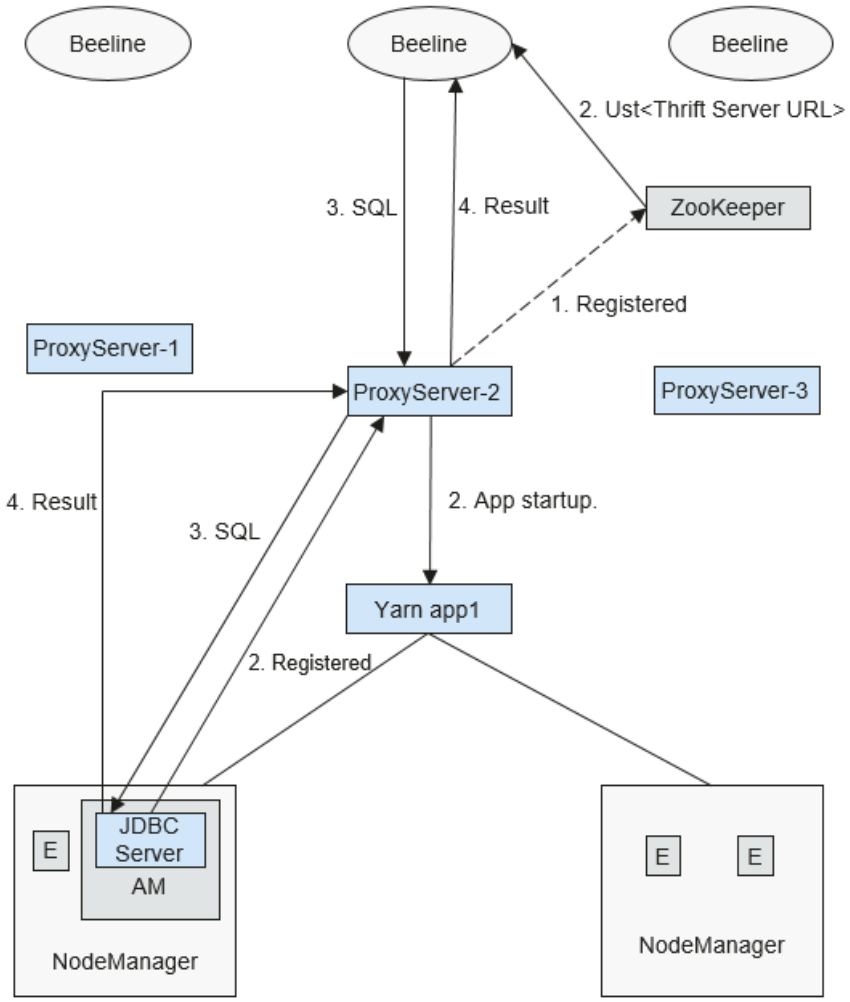
Scenario

When there are multiple tenants in a cluster, JDBCServer is dynamically started as required to ensure resource isolation between tenants, avoiding resource waste.

Implementation

Figure 5-146 shows the HA solution of the multi-tenant mode.

Figure 5-146 Multi-tenant mode of Spark JDBCServer



1. When ProxyServer is started, it registers with ZooKeeper by writing node information in a specified directory. Node information includes the instance IP, port number, version, and serial number (information of different nodes is separated by commas).

   NOTE

   In multi-tenant mode, the JDBCServer instance on MRS page indicates ProxyServer, the JDBCServer agent.

   An example is provided as follows:

   ```
   serverUri=192.168.169.84:22550
   ;version=8.3.1;sequence=0000001244,serverUri=192.168.195.232:22550
   ;version=8.3.1;sequence=0000001242,serverUri=192.168.81.37:22550
   ;version=8.3.1;sequence=0000001243,
   ```

2. To connect to ProxyServer, the client must specify a namespace, which is the directory of the ProxyServer instance that you want to access in ZooKeeper. When the client connects to ProxyServer, an instance under Namespace is randomly selected for connection. For details about the URL, see **URL Connection**.
3. After the client successfully connects to ProxyServer, ProxyServer checks whether the JDBCServer of a tenant exists. If yes, Beeline connects the

JDBCServer. If no, a new JDBCServer is started in Yarn-cluster mode. After the startup of JDBCServer, ProxyServer obtains the IP address of the JDBCServer and establishes the connection between Beeline and JDBCServer.

4. The client sends SQL statements to ProxyServer, which then forwards statements to the connected JDBCServer. JDBCServer returns the results to ProxyServer, which then returns the results to the client.

In multi-tenant HA mode, all ProxyServer instances are independent and equivalent. If one instance is interrupted during upgrade, other instances can accept the connection request from the client.

URL Connection

Multi-tenant mode

In multi-tenant mode, the client reads content from the ZooKeeper node and connects to ProxyServer. The connection strings are as follows:

- Security mode:
  - If Kinit authentication is enabled, the client URL is as follows:
    jdbc:hive2://*<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>*/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=**sparkthriftserver**;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.*<System domain name>*@*<System domain name>*;

    NOTE

    - **<zkNode_IP>:<zkNode_Port>** indicates the ZooKeeper URL. Use commas (,) to separate multiple URLs,

      For example, **192.168.81.37:24002,192.168.195.232:24002,192.168.169.84:24002**.
    - **sparkthriftserver** indicates the ZooKeeper directory, where a random JDBCServer instance is connected to the client.

    For example, when you use Beeline client for connection in security mode, run the following command:

    sh *CLIENT_HOME*/**spark/bin/beeline -u "jdbc:hive2://***<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>***/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.<System domain name>@<System domain name>;"**
  - If Keytab authentication is enabled, the URL is as follows:
    jdbc:hive2://<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.*<System domain name>*@*<System domain name>*;user.principal=<principal_name>;user.keytab=<path_to_keytab>

    <principal_name> indicates the principal of Kerberos user, for example, **test@***<System domain name>*. *<path_to_keytab>* indicates the Keytab file path corresponding to *<principal_name>*, for example, **/opt/auth/test/user.keytab**.
- Common mode:
  jdbc:hive2://*<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>*/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=**sparkthriftserver**;

For example, when you use Beeline client for connection in common mode, run the following command:

sh *CLIENT_HOME*/**spark/bin/beeline -u "jdbc:hive2://***<zkNode1_IP>:<zkNode1_Port>,<zkNode2_IP>:<zkNode2_Port>,<zkNode3_IP>:<zkNode3_Port>*/**;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver;"**

Non-multi-tenant mode

In non-multi-tenant mode, a client connects to a specified JDBCServer node. Compared with multi-active instance mode, the connection string in non-multi-active instance mode does not contain **serviceDiscoveryMode** and **zooKeeperNamespace** parameters about ZooKeeper.

For example, when you use Beeline client to connect JDBCServer in non-multi-tenant instance mode, run the following command:

sh *CLIENT_HOME*/**spark/bin/beeline -u "jdbc:hive2://***<server_IP>:<server_Port>*/**;user.principal=spark2x/hadoop.<System domain name>@<System domain name>;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.<System domain name>@<System domain name>;"**

NOTE

- **<server_IP>:<server_Port>** indicates the URL of the specified JDBCServer node.
- **CLIENT_HOME** indicates the client path.

Except the connection method, other operations of JDBCServer API in multi-tenant mode and non-multi-tenant mode are the same. Spark JDBCServer is another implementation of HiveServer2 in Hive. For details about how to use Spark JDBCServer, visit the official Hive website at **https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients**.

Specifying a Tenant

Generally, the client submitted by a user connects to the default JDBCServer of the tenant to which the user belongs. If you want to connect the client to the JDBCServer of a specified tenant, add the **--hiveconf mapreduce.job.queuename** parameter.

Command for connecting Beeline is as follows (**aaa** indicates the tenant name):

beeline --hiveconf mapreduce.job.queuename=aaa -u 'jdbc:hive2://192.168.39.30:24002,192.168.40.210:24002,192.168.215.97:24002;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.<System domain name>@<System domain name>;'

5.35.3 Relationships Between Spark and Other Components

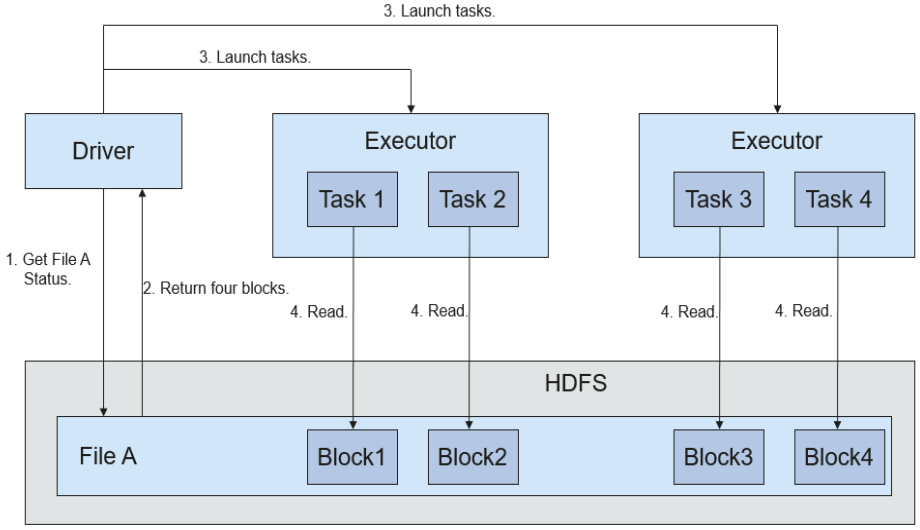
Spark and HDFS

Data computed by Spark comes from multiple data sources, such as local files and HDFS. Most data comes from HDFS which can read data in large scale for parallel computing After being computed, data can be stored in HDFS.

Spark includes Driver and Executor. Driver schedules tasks and Executor runs tasks.

Figure 5-147 describes the file reading process.

Figure 5-147 File reading process

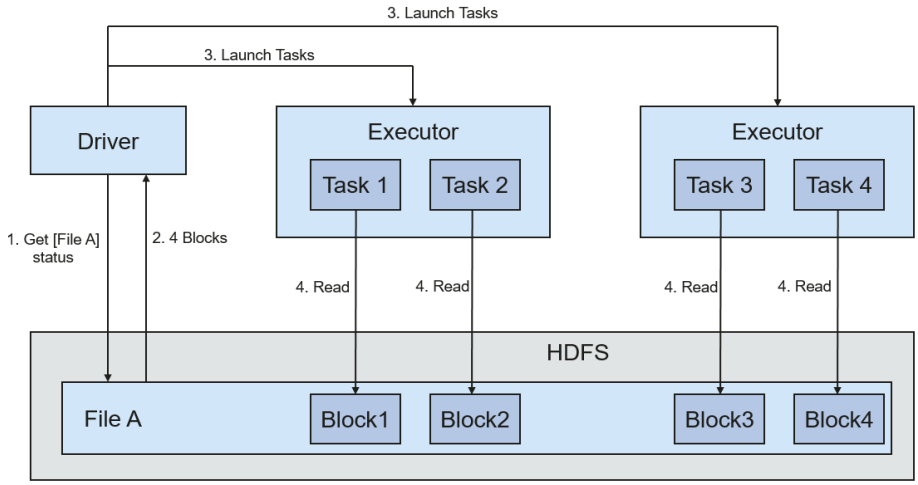


The file reading process is as follows:

1. Driver interconnects with HDFS to obtain the information of File A.
2. The HDFS returns the detailed block information about this file.
3. Driver sets a parallel degree based on the block data amount, and creates multiple tasks to read the blocks of this file.
4. Executor runs the tasks and reads the detailed blocks as part of the Resilient Distributed Dataset (RDD).

Figure 5-148 describes the file writing process.

Figure 5-148 File writing process



The file writing process is as follows:

1. Driver creates a directory where the file is to be written.
2. Based on the RDD distribution status, the number of tasks related to data writing is computed, and these tasks are sent to Executor.
3. Executor runs these tasks, and writes the RDD data to the directory created in 1.

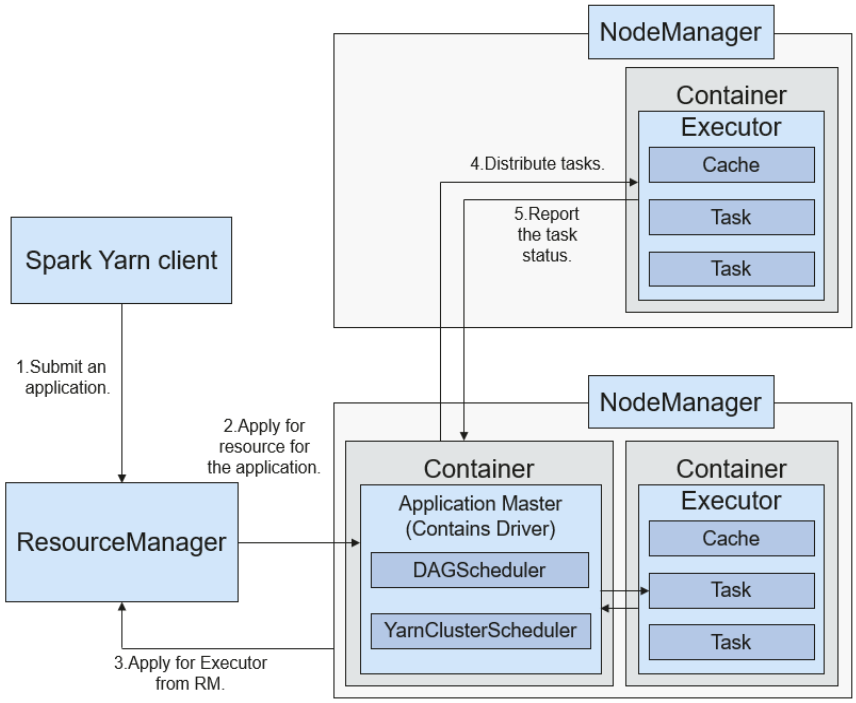
Spark and YARN

The Spark computing and scheduling can be implemented using Yarn mode. Spark enjoys the computing resources provided by Yarn clusters and runs tasks in a distributed way. Spark on Yarn has two modes: Yarn-cluster and Yarn-client.

- Yarn-cluster mode

  Figure 5-149 describes the operation framework.

  Figure 5-149 Spark on Yarn-cluster operation framework



  Spark on Yarn-cluster implementation process:

  a. The client generates the application information, and then sends the information to ResourceManager.
  b. ResourceManager allocates the first container (ApplicationMaster) to SparkApplication and starts the driver on the container.
  c. ApplicationMaster applies for resources from ResourceManager to run the container.

     ResourceManager allocates the containers to ApplicationMaster, which communicates with the related NodeManagers and starts the executor in the obtained container. After the executor is started, it registers with drivers and applies for tasks.

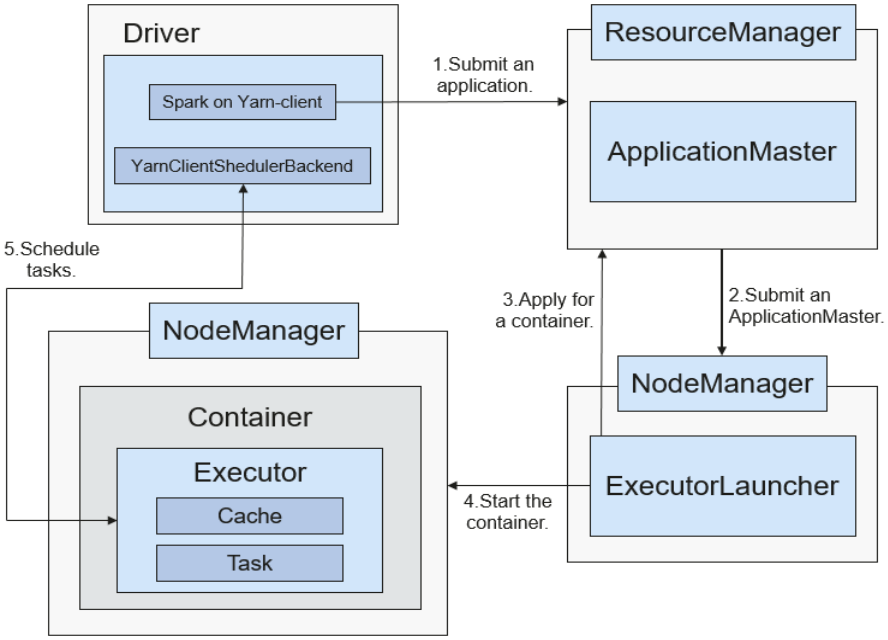
d. Drivers allocate tasks to the executors.

e. Executors run tasks and report the operating status to Drivers.

- Yarn-client mode

  Figure 5-150 describes the operation framework.

  Figure 5-150 Spark on Yarn-client operation framework



  Spark on Yarn-client implementation process:

  NOTE

  In Yarn-client mode, the Driver is deployed and started on the client. In Yarn-client mode, the client of an earlier version is incompatible. The Yarn-cluster mode is recommended.

  a. The client sends the Spark application request to ResourceManager, and packages all information required to start ApplicationMaster and sends the information to ResourceManager. ResourceManager then returns the results to the client. The results include information such as ApplicationId, and the upper limit as well as lower limit of available resources. After receiving the request, ResourceManager finds a proper node for ApplicationMaster and starts it on this node. ApplicationMaster is a role in Yarn, and the process name in Spark is ExecutorLauncher.

  b. Based on the resource requirements of each task, ApplicationMaster can apply for a series of containers to run tasks from ResourceManager.

  c. After receiving the newly allocated container list (from ResourceManager), ApplicationMaster sends information to the related NodeManagers to start the containers.

     ResourceManager allocates the containers to ApplicationMaster, which communicates with the related NodeManagers and starts the executor in the obtained container. After the executor is started, it registers with drivers and applies for tasks.

NOTE

Running Containers will not be suspended to release resources.

d. Drivers allocate tasks to the executors. Executors run tasks and report the operating status to Drivers.

5.35.4 Spark Open Source New Features

Purpose

Spark 3x provides some new open source features compared with Spark 1.5. The specific features or concepts are as follows:

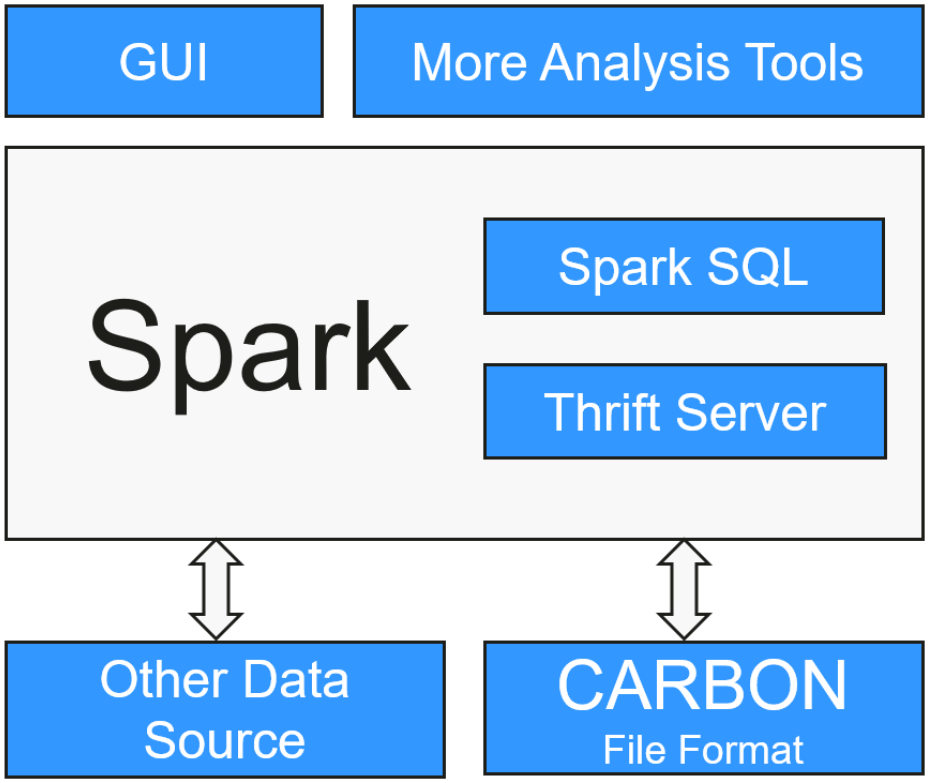
- DataSet: For details, see **SparkSQL and DataSet Principle**.
- Spark SQL Native DDL/DML: For details, see **SparkSQL and DataSet Principle**.
- SparkSession: For details, see **SparkSession Principle**.
- Structured Streaming: For details, see **Structured Streaming Principles**.
- Optimizing Small Files
- Optimizing the Aggregate Algorithm
- Optimizing Datasource Tables
- Merging CBO

5.35.5 Spark Enhanced Open Source Features

5.35.5.1 CarbonData Overview

CarbonData is a new Apache Hadoop native data-store format. CarbonData allows faster interactive queries over PetaBytes of data using advanced columnar storage, index, compression, and encoding techniques to improve computing efficiency. In addition, CarbonData is also a high-performance analysis engine that integrates data sources with Spark.

Figure 5-151 Basic architecture of CarbonData



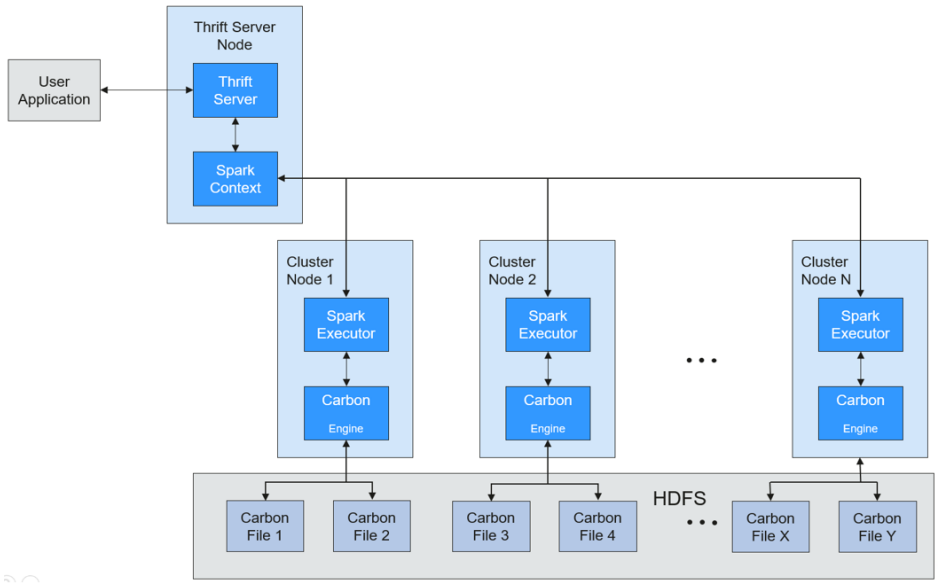
The purpose of using CarbonData is to provide quick response to ad hoc queries of big data. Essentially, CarbonData is an Online Analytical Processing (OLAP) engine, which stores data by using tables similar to those in Relational Database Management System (RDBMS). You can import more than 10 TB data to tables created in CarbonData format, and CarbonData automatically organizes and stores data using the compressed multi-dimensional indexes. After data is loaded to CarbonData, CarbonData responds to ad hoc queries in seconds.

CarbonData integrates data sources into the Spark ecosystem and you can query and analyze the data using Spark SQL. You can also use the third-party tool JDBCServer provided by Spark to connect to SparkSQL.

Topology of CarbonData

CarbonData runs as a data source inside Spark. Therefore, CarbonData does not start any additional processes on nodes in clusters. CarbonData engine runs inside the Spark executor.

Figure 5-152 Topology of CarbonData



Data stored in CarbonData Table is divided into several CarbonData data files. Each time when data is queried, CarbonData Engine reads and filters data sets. CarbonData Engine runs as a part of the Spark Executor process and is responsible for handling a subset of data file blocks.

Table data is stored in HDFS. Nodes in the same Spark cluster can be used as HDFS data nodes.

CarbonData Features

- SQL: CarbonData is compatible with Spark SQL and supports SQL query operations performed on Spark SQL.
- Simple Table dataset definition: CarbonData allows you to define and create datasets by using user-friendly Data Definition Language (DDL) statements. CarbonData DDL is flexible and easy to use, and can define complex tables.
- Easy data management: CarbonData provides various data management functions for data loading and maintenance. CarbonData supports bulk loading of historical data and incremental loading of new data. Loaded data can be deleted based on load time and a specific loading operation can be undone.
- CarbonData file format is a columnar store in HDFS. This format has many new column-based file storage features, such as table splitting and data compression. CarbonData has the following characteristics:
  - Stores data along with index: Significantly accelerates query performance and reduces the I/O scans and CPU resources, when there are filters in the query. CarbonData index consists of multiple levels of indices. A processing framework can leverage this index to reduce the task that needs to be schedules and processed, and it can also perform skip scan in more finer grain unit (called blocklet) in task side scanning instead of scanning the whole file.
  - Operable encoded data: Through supporting efficient compression and global encoding schemes, CarbonData can query on compressed/encoded

data. The data can be converted just before returning the results to the users, which is called late materialized.

- Supports various use cases with one single data format: like interactive OLAP-style query, sequential access (big scan), and random access (narrow scan).

Key Technologies and Advantages of CarbonData

- Quick query response: CarbonData features high-performance query. The query speed of CarbonData is 10 times of that of Spark SQL. It uses dedicated data formats and applies multiple index technologies, global dictionary code, and multiple push-down optimizations, providing quick response to TB-level data queries.
- Efficient data compression: CarbonData compresses data by combining the lightweight and heavyweight compression algorithms. This significantly saves 60% to 80% data storage space and the hardware storage cost.

CarbonData Index Cache Server

To solve the pressure and problems brought by the increasing data volume to the driver, an independent index cache server is introduced to separate the index from the Spark application side of Carbon query. All index content is managed by the index cache server. Spark applications obtain required index data in RPC mode. In this way, a large amount of memory on the service side is released so that services are not affected by the cluster scale and the performance or functions are not affected.

5.35.5.2 Optimizing SQL Query of Data of Multiple Sources

Scenario

Enterprises usually store massive data, such as from various databases and warehouses, for management and information collection. However, diversified data sources, hybrid dataset structures, and scattered data storage lower query efficiency.

The open source Spark only supports simple filter pushdown during querying of multi-source data. The SQL engine performance is deteriorated due of a large amount of unnecessary data transmission. The pushdown function is enhanced, so that **aggregate**, complex **projection**, and complex **predicate** can be pushed to data sources, reducing unnecessary data transmission and improving query performance.

Only the JDBC data source supports pushdown of query operations, such as **aggregate**, **projection**, **predicate**, **aggregate over inner join**, and **aggregate over union all**. All pushdown operations can be enabled based on your requirements.

Table 5-29 Enhanced query of cross-source query

Module	Before Enhancement	After Enhancement
aggregate	The pushdown of **aggregate** is not supported.	• Aggregation functions including **sum**, **avg**, **max**, **min**, and **count** are supported. Example: select count(*) from table • Internal expressions of aggregation functions are supported. Example: select sum(a+b) from table • Calculation of aggregation functions is supported. Example: select avg(a) + max(b) from table • Pushdown of **having** is supported. Example: select sum(a) from table where a>0 group by b having sum(a)>10 • Pushdown of some functions is supported. Pushdown of lines in mathematics, time, and string functions, such as **abs()**, **month()**, and **length()** are supported. In addition to the preceding built-in functions, you can run the **SET** command to add functions supported by data sources. Example: select sum(abs(a)) from table • Pushdown of **limit** and **order by** after **aggregate** is supported. However, the pushdown is not supported in Oracle, because Oracle does not support **limit**. Example: select sum(a) from table where a>0 group by b order by sum(a) limit 5
projection	Only pushdown of simple **projection** is supported. Example: select a, b from table	• Complex expressions can be pushed down. Example: select (a+b)*c from table • Some functions can be pushed down. For details, see the description below the table. Example: select length(a)+abs(b) from table • Pushdown of **limit** and **order by** after **projection** is supported. Example: select a, b+c from table order by a limit 3

Module	Before Enhancement	After Enhancement
predicate	Only simple filtering with the column name on the left of the operator and values on the right is supported. Example: select * from table where a>0 or b in ("aaa", "bbb")	• Complex expression pushdown is supported. Example: select * from table where a +b>c*d or a/c in (1, 2, 3) • Some functions can be pushed down. For details, see the description below the table. Example: select * from table where length(a)>5
aggregate over inner join	Related data from the two tables must be loaded to Spark. The join operation must be performed before the **aggregate** operation.	The following functions are supported: • Aggregation functions including **sum**, **avg**, **max**, **min**, and **count** are supported. • All **aggregate** operations can be performed in a same table. The **group by** operations can be performed on one or two tables and only inner join is supported. The following scenarios are not supported: • **aggregate** cannot be pushed down from both the left- and right-join tables. • **aggregate** contains operations, for example, sum(a+b). • **aggregate** operations, for example, sum(a)+min(b).
aggregate over union all	Related data from the two tables must be loaded to Spark. **union** must be performed before **aggregate**.	Supported scenarios: Aggregation functions including **sum**, **avg**, **max**, **min**, and **count** are supported. Unsupported scenarios: • **aggregate** contains operations, for example, sum(a+b). • **aggregate** operations, for example, sum(a)+min(b).

Precautions

- If external data source is Hive, query operation cannot be performed on foreign tables created by Spark.
- Only MySQL and MPPDB data sources are supported.

5.36 Tez

Tez is Apache's latest open source computing framework that supports Directed Acyclic Graph (DAG) jobs. It can convert multiple dependent jobs into one job, greatly improving the performance of DAG jobs. If projects like Hive use Tez instead of MapReduce as the backbone of data processing, response time will be significantly reduced. Tez is built on YARN and can run MapReduce jobs without any modification.

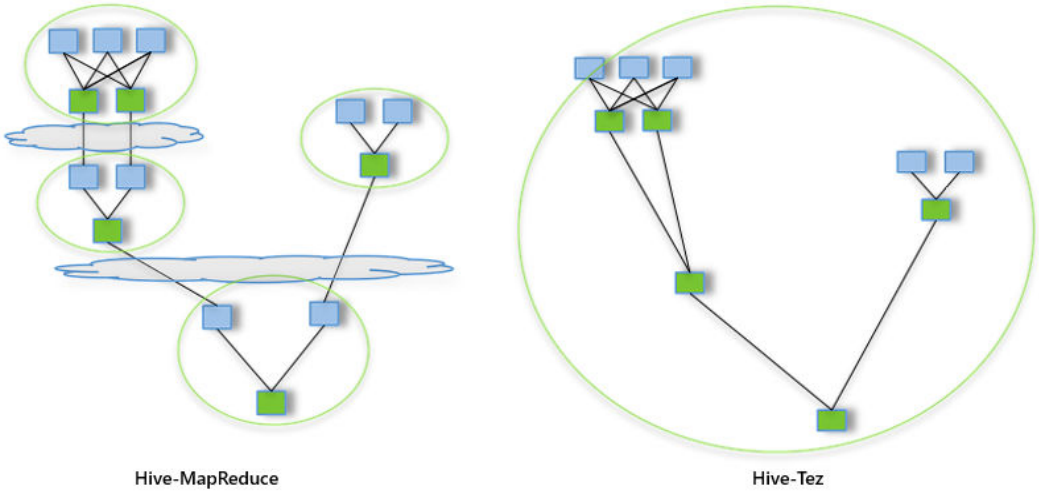
MRS uses Tez as the default execution engine of Hive. Tez remarkably surpasses the original MapReduce computing engine in terms of execution efficiency.

For details about Tez, see **https://tez.apache.org/**.

Relationship Between Tez and MapReduce

Tez uses a DAG to organize MapReduce tasks. In the DAG, a node is an RDD, and an edge indicates an operation on the RDD. The core idea is to further split Map tasks and Reduce tasks. A Map task is split into the Input-Processor-Sort-Merge-Output tasks, and the Reduce task is split into the Input-Shuffle-Sort-Merge-Process-output tasks. Tez flexibly regroups several small tasks to form a large DAG job.

Figure 5-153 Processes for submitting tasks using Hive on MapReduce and Hive on Tez



A Hive on MapReduce task contains multiple MapReduce tasks. Each task stores intermediate results to HDFS. The reducer in the previous step provides data for the mapper in the next step. A Hive on Tez task can complete the same processing process in only one task, and HDFS does not need to be accessed between tasks.

Relationship Between Tez and Yarn

Tez is a computing framework running on Yarn. The runtime environment consists of ResourceManager and ApplicationMaster of Yarn. ResourceManager is a brand new resource manager system, and ApplicationMaster is responsible for cutting MapReduce job data, assigning tasks, applying for resources, scheduling tasks, and tolerating faults. In addition, TezUI depends on TimelineServer provided by Yarn to display the running process of Tez tasks.

5.37 YARN

5.37.1 YARN Basic Principles

The Apache open source community introduces the unified resource management framework YARN to share Hadoop clusters, improve their scalability and reliability, and eliminate a performance bottleneck of JobTracker in the early MapReduce framework.

The fundamental idea of YARN is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate daemons. The idea is to have a global ResourceManager (RM) and per-application ApplicationMaster (AM).

NOTE

An application is either a single job in the classical sense of MapReduce jobs or a Directed Acyclic Graph (DAG) of jobs.

Architecture

ResourceManager is the essence of the layered structure of YARN. This entity controls an entire cluster and manages the allocation of applications to underlying compute resources. The ResourceManager carefully allocates various resources (compute, memory, bandwidth, and so on) to underlying NodeManagers (YARN's per-node agents). The ResourceManager also works with ApplicationMasters to allocate resources, and works with the NodeManagers to start and monitor their underlying applications. In this context, the ApplicationMaster has taken some of the role of the prior TaskTracker, and the ResourceManager has taken the role of the JobTracker.

ApplicationMaster manages each instance of an application running in YARN. The ApplicationMaster negotiates resources from the ResourceManager and works with the NodeManagers to monitor container execution and resource usage (CPU and memory resource allocation).

The NodeManager manages each node in a YARN cluster. The NodeManager provides per-node services in a cluster, from overseeing the management of a container over its lifecycle to monitoring resources and tracking the health of its nodes. MRv1 manages execution of the Map and Reduce tasks through slots, whereas the NodeManager manages abstract containers, which represent per-node resources available for a particular application.

Figure 5-154 Architecture

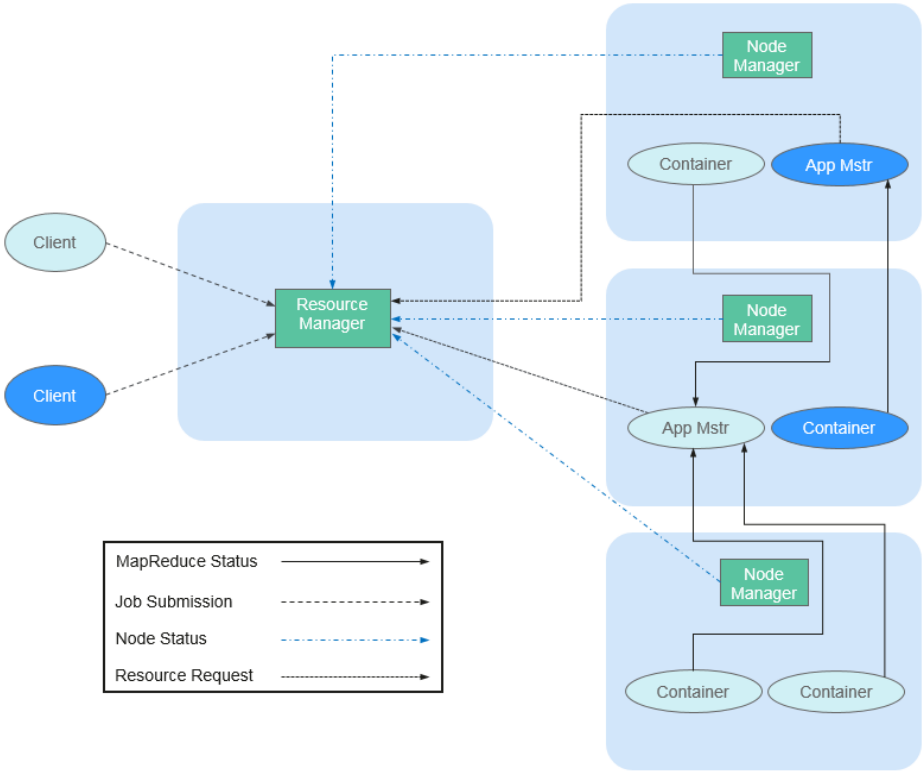


Table 5-30 describes the components shown in Figure 5-154.

Table 5-30 Architecture description

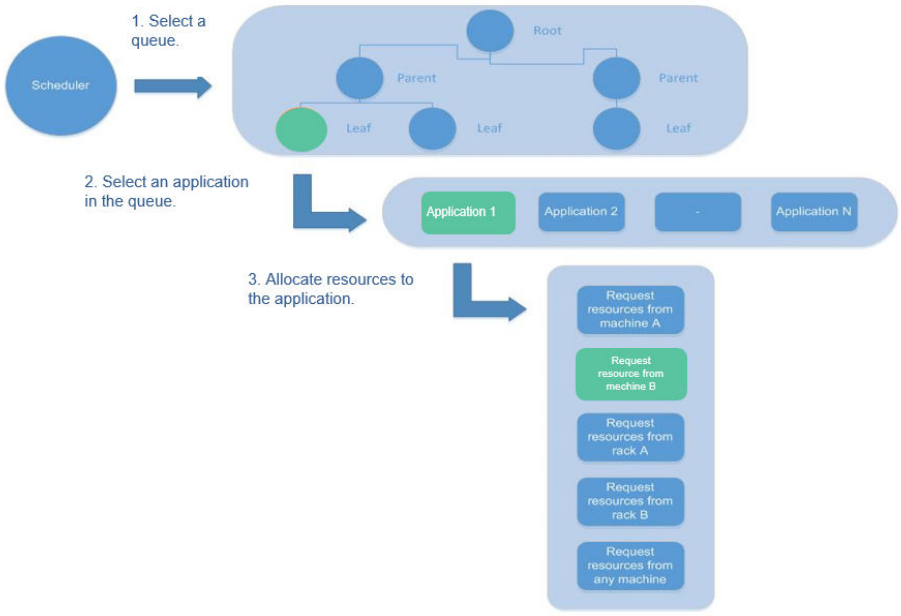
Name	Description
Client	Client of a YARN application. You can submit a task to ResourceManager and query the operating status of an application using the client.
ResourceManager(RM)	RM centrally manages and allocates all resources in the cluster. It receives resource reporting information from each node (NodeManager) and allocates resources to applications on the basis of the collected resources according a specified policy.
NodeManager(NM)	NM is the agent on each node of YARN. It manages the computing node in Hadoop cluster, establishes communication with ResourceManger, monitors the lifecycle of containers, monitors the usage of resources such as memory and CPU of each container, traces node health status, and manages logs and auxiliary services used by different applications.

Name	Description
ApplicationMaster(AM)	AM (App Mstr in the figure above) is responsible for all tasks through the lifcycle of in an application. The tasks include the following: Negotiate with an RM scheduler to obtain a resource; further allocate the obtained resources to internal tasks (secondary allocation of resources); communicate with the NM to start or stop tasks; monitor the running status of all tasks; and apply for resources for tasks again to restart the tasks when the tasks fail to be executed.
Container	A resource abstraction in YARN. It encapsulates multi-dimensional resources (including only memory and CPU) on a certain node. When ApplicationMaster applies for resources from ResourceManager, the ResourceManager returns resources to the ApplicationMaster in a container. YARN allocates one container for each task and the task can only use the resources encapsulated in the container.

In YARN, resource schedulers organize resources through hierarchical queues. This ensures that resources are allocated and shared among queues, thereby improving the usage of cluster resources. The core resource allocation model of Superior Scheduler is the same as that of Capacity Scheduler, as shown in the following figure.

A scheduler maintains queue information. You can submit applications to one or more queues. During each NM heartbeat, the scheduler selects a queue according to a specific scheduling rule, selects an application in the queue, and then allocates resources to the application. If resources fail to be allocated to the application due to the limit of some parameters, the scheduler will select another application. After the selection, the scheduler processes the resource request of this application. The scheduler gives priority to the requests for local resources first, and then for resources on the same rack, and finally for resources from any machine.

Figure 5-155 Resource allocation model



Principle

The new Hadoop MapReduce framework is named MRv2 or YARN. YARN consists of ResourceManager, ApplicationMaster, and NodeManager.

- ResourceManager is a global resource manager that manages and allocates resources in the system. ResourceManager consists of Scheduler and Applications Manager.
  - Scheduler allocates system resources to all running applications based on the restrictions such as capacity and queue (for example, allocates a certain amount of resources for a queue and executes a specific number of jobs). It allocates resources based on the demand of applications, with container being used as the resource allocation unit. Functioning as a dynamic resource allocation unit, Container encapsulates memory, CPU, disk, and network resources, thereby limiting the resource consumed by each task. In addition, the Scheduler is a pluggable component. You can design new schedulers as required. YARN provides multiple directly available schedulers, such as Fair Scheduler and Capacity Scheduler.
  - Applications Manager manages all applications in the system and involves submitting applications, negotiating with schedulers about resources, enabling and monitoring ApplicationMaster, and restarting ApplicationMaster upon the startup failure.
- NodeManager is the resource and task manager of each node. On one hand, NodeManager periodically reports resource usage of the local node and the running status of each Container to ResourceManager. On the other hand, NodeManager receives and processes requests from ApplicationMaster for starting or stopping Containers.
- ApplicationMaster is responsible for all tasks through the lifecycle of an application, these channels include the following:
  - Negotiate with the RM scheduler to obtain resources.

  - Assign resources to internal components (secondary allocation of resources).
  - Communicates with NodeManager to start or stop tasks.
  - Monitor the running status of all tasks, and applies for resources again for tasks when tasks fail to run to restart the tasks.

Capacity Scheduler Principle

Capacity Scheduler is a multi-user scheduler. It allocates resources by queue and sets the minimum/maximum resources that can be used for each queue. In addition, the upper limit of resource usage is set for each user to prevent resource abuse. Remaining resources of a queue can be temporarily shared with other queues.

Capacity Scheduler supports multiple queues. It configures a certain amount of resources for each queue and adopts the first-in-first-out queuing (FIFO) scheduling policy. To prevent one user's applications from exclusively using the resources in a queue, Capacity Scheduler sets a limit on the number of resources used by jobs submitted by one user. During scheduling, Capacity Scheduler first calculates the number of resources required for each queue, and selects the queue that requires the least resources. Then, it allocates resources based on the job priority and time that jobs are submitted as well as the limit on resources and memory. Capacity Scheduler supports the following features:

- Guaranteed capacity: As the MRS cluster administrator, you can set the lower and upper limits of resource usage for each queue. All applications submitted to this queue share the resources.
- High flexibility: Temporarily, the remaining resources of a queue can be shared with other queues. However, such resources must be released in case of new application submission to the queue. Such flexible resource allocation helps notably improve resource usage.
- Multi-tenancy: Multiple users can share a cluster, and multiple applications can run concurrently. To avoid exclusive resource usage by a single application, user, or queue, the MRS cluster administrator can add multiple constraints (for example, limit on concurrent tasks of a single application).
- Assured protection: An ACL list is provided for each queue to strictly limit user access. You can specify the users who can view your application status or control the applications. Additionally, the MRS cluster administrator can specify a queue administrator and a cluster system administrator.
- Dynamic update of configuration files: MRS cluster administrators can dynamically modify configuration parameters to manage clusters online.

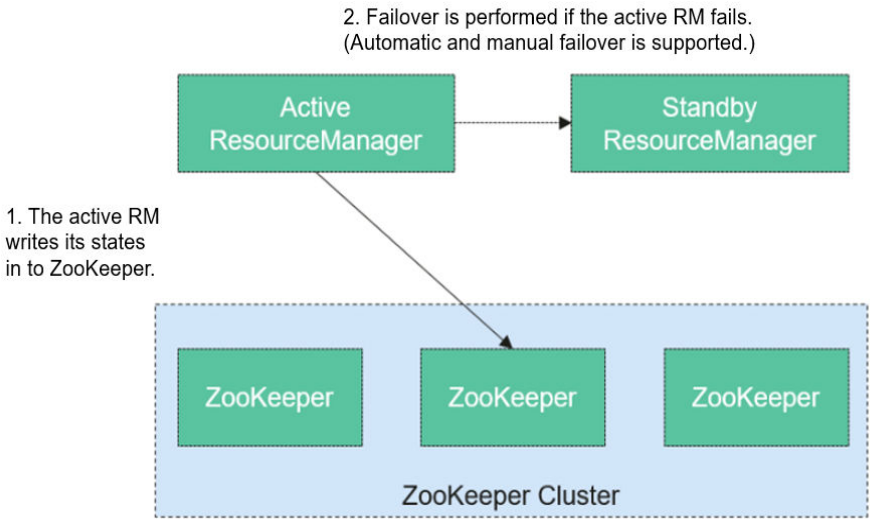
Each queue in Capacity Scheduler can limit the resource usage. However, the resource usage of a queue determines its priority when resources are allocated to queues, indicating that queues with smaller capacity are competitive. If the throughput of a cluster is big, delay scheduling enables an application to give up cross-machine or cross-rack scheduling, and to request local scheduling.

5.37.2 YARN HA Solution

HA Principles and Implementation Solution

ResourceManager in YARN manages resources and schedules tasks in the cluster. In versions earlier than Hadoop 2.4, SPOFs may occur on ResourceManager in the YARN cluster. The YARN HA solution uses redundant ResourceManager nodes to tackle challenges of service reliability and fault tolerance.

Figure 5-156 ResourceManager HA architecture



ResourceManager HA is achieved using active-standby ResourceManager nodes, as shown in **Figure 5-156**. Similar to the HDFS HA solution, the ResourceManager HA allows only one ResourceManager node to be in the active state at any time. When the active ResourceManager fails, the active-standby switchover can be triggered automatically or manually.

When the automatic failover function is not enabled, after the YARN cluster is enabled, MRS cluster administrators need to run the ***yarn rmadmin*** command to manually switch one of the ResourceManager nodes to the active state. Upon a planned maintenance event or a fault, they are expected to first demote the active ResourceManager to the standby state and the standby ResourceManager promote to the active state.

When automatic failover is enabled, a built-in ActiveStandbyElector that is based on ZooKeeper is used to decide which ResourceManager node should be the active one. When the active ResourceManager is faulty, another ResourceManager node is automatically selected to be the active one to take over the faulty node.

When ResourceManager nodes in the cluster are deployed in HA mode, the configuration **yarn-site.xml** used by clients needs to list all the ResourceManager nodes. The client (including ApplicationMaster and NodeManager) searches for the active ResourceManager in polling mode. That is, the client needs to provide the fault tolerance mechanism. If the active ResourceManager cannot be connected with, the client continuously searches for a new one in polling mode.

After the standby ResourceManager node becomes the active one, the upper-layer applications can recover to their status when the fault occurs. For details, see

ResourceManger Restart. When ResourceManager Restart is enabled, the restarted ResourceManager node loads the information of the previous active ResourceManager node, and takes over container status information on all NodeManager nodes to continue service running. In this way, status information can be saved by periodically executing checkpoint operations, avoiding data loss. Ensure that both active and standby ResourceManager nodes can access the status information. Currently, three methods are provided for sharing status information by file system (FileSystemRMStateStore), LevelDB database (LeveldbRMStateStore), and ZooKeeper (ZKRMStateStore). Among them, only ZKRMStateStore supports the Fencing mechanism. By default, Hadoop uses ZKRMStateStore.

For more information about the YARN HA solution, visit the following website:

https://hadoop.apache.org/docs/r3.3.1/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html

5.37.3 Relationships Between YARN and Other Components

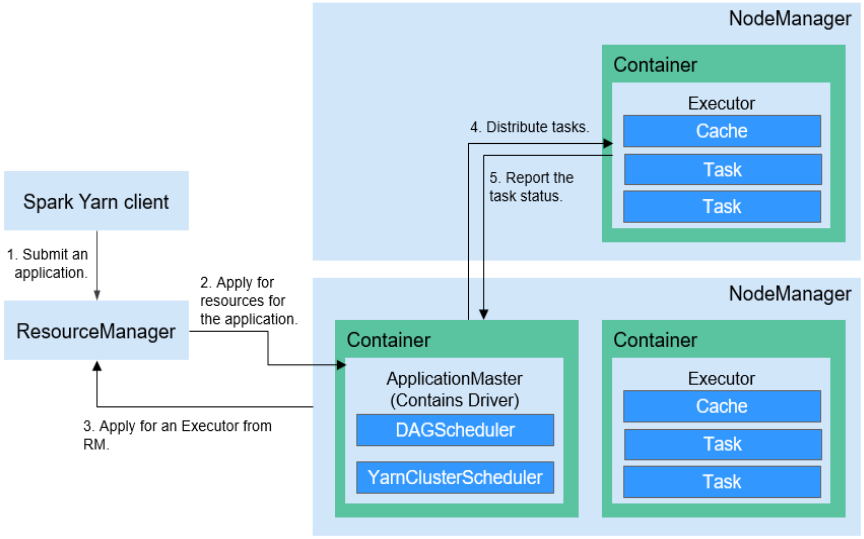
YARN and Spark

The Spark computing and scheduling can be implemented using YARN mode. Spark enjoys the compute resources provided by YARN clusters and runs tasks in a distributed way. Spark on YARN has two modes: YARN-cluster and YARN-client.

- YARN Cluster mode

  Figure 5-157 describes the operation framework.

  Figure 5-157 Spark on YARN-cluster operation framework



  Spark on YARN-cluster implementation process:

  a. The client generates the application information, and then sends the information to ResourceManager.

  b. ResourceManager allocates the first container (ApplicationMaster) to SparkApplication and starts the driver on the container.

c. ApplicationMaster applies for resources from ResourceManager to run the container.

   ResourceManager allocates the containers to ApplicationMaster, which communicates with the related NodeManagers and starts the executor in the obtained container. After the executor is started, it registers with drivers and applies for tasks.

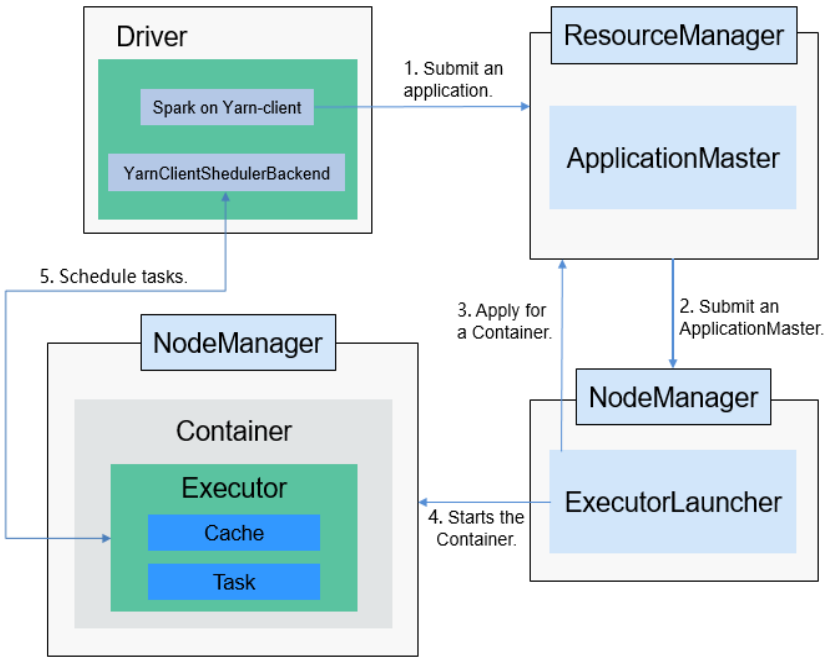
d. Drivers allocate tasks to the executors.

e. Executors run tasks and report the operating status to Drivers.

- YARN Client mode

  Figure 5-158 describes the operation framework.

  Figure 5-158 Spark on YARN-client operation framework



  Spark on YARN-client implementation process:

  NOTE

  In YARN-client mode, the driver is deployed and started on the client. In YARN-client mode, the client of an earlier version is incompatible. You are advised to use the YARN-cluster mode.

  a. The client sends the Spark application request to ResourceManager, then ResourceManager returns the results. The results include information such as Application ID and the maximum and minimum available resources. The client packages all information required to start ApplicationMaster, and sends the information to ResourceManager.

  b. After receiving the request, ResourceManager finds a proper node for ApplicationMaster and starts it on this node. ApplicationMaster is a role in YARN, and the process name in Spark is ExecutorLauncher.

  c. Based on the resource requirements of each task, ApplicationMaster can apply for a series of containers to run tasks from ResourceManager.

d. After receiving the newly allocated container list (from ResourceManager), ApplicationMaster sends information to the related NodeManagers to start the containers.

   ResourceManager allocates the containers to ApplicationMaster, which communicates with the related NodeManagers and starts the executor in the obtained container. After the executor is started, it registers with drivers and applies for tasks.

   NOTE

   Running Containers will not be suspended to release resources.

e. Drivers allocate tasks to the executors. Executors run tasks and report the operating status to Drivers.

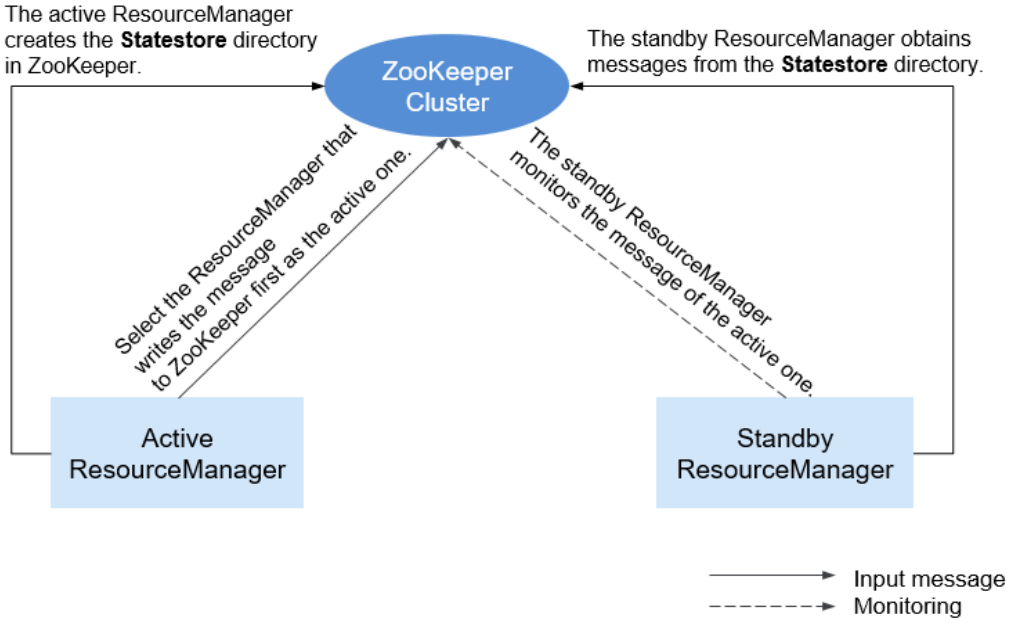
YARN and MapReduce

MapReduce is a computing framework running on YARN, which is used for batch processing. MRv1 is implemented based on MapReduce in Hadoop 1.0, which is composed of programming models (new and old programming APIs), running environment (JobTracker and TaskTracker), and data processing engine (MapTask and ReduceTask). This framework is still weak in scalability, fault tolerance (JobTracker SPOF), and compatibility with multiple frameworks. (Currently, only the MapReduce computing framework is supported.) MRv2 is implemented based on MapReduce in Hadoop 2.0. The source code reuses MRv1 programming models and data processing engine implementation, and the running environment is composed of ResourceManager and ApplicationMaster. ResourceManager is a brand new resource manager system, and ApplicationMaster is responsible for cutting MapReduce job data, assigning tasks, applying for resources, scheduling tasks, and tolerating faults.

YARN and ZooKeeper

Figure 5-159 shows the relationship between ZooKeeper and YARN.

Figure 5-159 Relationship Between ZooKeeper and YARN



1. When the system is started, ResourceManager attempts to write state information to ZooKeeper. ResourceManager that first writes state information to ZooKeeper is selected as the active ResourceManager, and others are standby ResourceManagers. The standby ResourceManagers periodically monitor active ResourceManager election information in ZooKeeper.
2. The active ResourceManager creates the **Statestore** directory in ZooKeeper to store application information. If the active ResourceManager is faulty, the standby ResourceManager obtains application information from the **Statestore** directory and restores the data.

YARN and Tez

The Hive on Tez job information requires the TimeLine Server capability of YARN so that Hive tasks can display the current and historical status of applications, facilitating storage and retrieval.

NOTE

The TimelineServer saves data to the in-memory database LevelDB, which is memory-intensive. It is imperative to allocate a minimum of 30 GB of memory to the node hosting the TimelineServer.

5.37.4 Yarn Enhanced Open Source Features

Priority-based task scheduling

In the native Yarn resource scheduling mechanism, if the whole Hadoop cluster resources are occupied by those MapReduce jobs submitted earlier, jobs submitted later will be kept in pending state until all running jobs are executed and resources are released.

The MRS cluster provides the task priority scheduling mechanism. With this feature, you can define jobs of different priorities. Jobs of high priority can preempt resources released from jobs of low priority though the high-priority jobs are submitted later. The low-priority jobs that are not started will be suspended unless those jobs of high priority are completed and resources are released, then they can properly be started.

This feature enables services to control computing jobs more flexibly, thereby achieving higher cluster resource utilization.

NOTE

Container reuse is in conflict with task priority scheduling. If container reuse is enabled, resources are being occupied, and task priority scheduling does not take effect.

Yarn Permission Control

The permission mechanism of Hadoop Yarn is implemented through ACLs. The following describes how to grant different permission control to different users:

- Admin ACL

  An O&M administrator is specified for the YARN cluster. The Admin ACL is determined by **yarn.admin.acl**. The cluster O&M administrator can access the

ResourceManager web UI and operate NodeManager nodes, queues, and NodeLabel, **but cannot submit tasks**.

- Queue ACL

  To facilitate user management in the cluster, users or user groups are divided into several queues to which each user and user group belongs. Each queue contains permissions to submit and manage applications (for example, terminate any application).

Open source functions:

Currently, Yarn supports the following roles for users:

- Cluster O&M administrator
- Queue administrator
- Common user

However, the APIs (such as the web UI, REST API, and Java API) provided by Yarn do not support role-specific permission control. Therefore, all users have the permission to access the application and cluster information, which does not meet the isolation requirements in the multi-tenant scenario.

This is an enhanced function.

In security mode, permission management is enhanced for the APIs such as web UI, REST API, and Java API provided by Yarn. Permission control can be performed based on user roles.

Role-based permissions are as follows:

- Cluster O&M administrator: performs management operations in the Yarn cluster, such as accessing the ResourceManager web UI, refreshing queues, setting NodeLabel, and performing active/standby switchover.
- Queue administrator: has the permission to modify and view queues managed by the Yarn cluster.
- Common user: has the permission to modify and view self-submitted applications in the Yarn cluster.

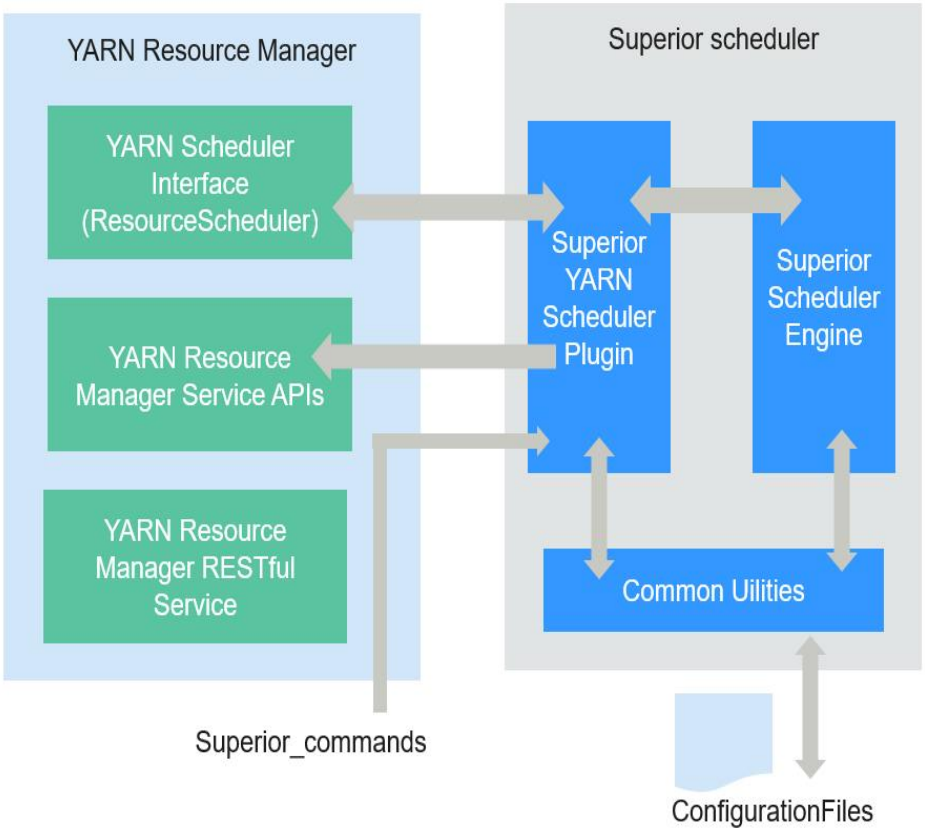
Superior Scheduler Principle (Self-developed)

Superior Scheduler is a scheduling engine designed for the Hadoop Yarn distributed resource management system. It is a high-performance and enterprise-level scheduler designed for converged resource pools and multi-tenant service requirements.

Superior Scheduler achieves all functions of open source schedulers, Fair Scheduler, and Capacity Scheduler. Compared with the open source schedulers, Superior Scheduler is enhanced in the enterprise multi-tenant resource scheduling policy, resource isolation and sharing among users in a tenant, scheduling performance, system resource usage, and cluster scalability. Superior Scheduler is designed to replace open source schedulers.

Similar to open source Fair Scheduler and Capacity Scheduler, Superior Scheduler follows the Yarn scheduler plugin API to interact with Yarn ResourceManager to offer resource scheduling functionalities. **Figure 5-160** shows the overall system diagram.

Figure 5-160 Internal architecture of Superior Scheduler



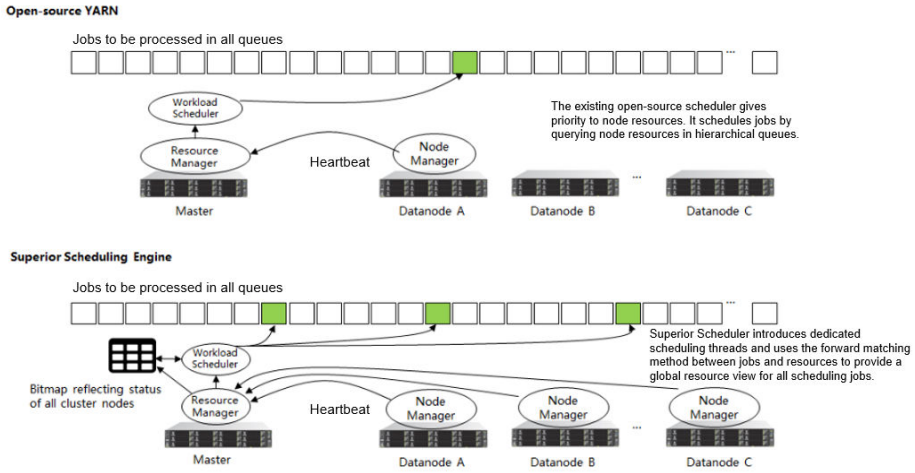
In **Figure 5-160**, Superior Scheduler consists of the following modules:

- Superior Scheduler Engine is a high performance scheduler engine with rich scheduling policies.
- Superior Yarn Scheduler Plugin functions as a bridge between Yarn ResourceManager and Superior Scheduler Engine and interacts with Yarn ResourceManager.

  The scheduling principle of open source schedulers is that resources match jobs based on the heartbeats of computing nodes. Specifically, each computing node periodically sends heartbeat messages to ResourceManager of Yarn to notify the node status and starts the scheduler to assign jobs to the node itself. In this scheduling mechanism, the scheduling period depends on the heartbeat. If the cluster scale increases, bottleneck on system scalability and scheduling performance may occur. In addition, because resources match jobs, the scheduling accuracy of an open source scheduler is limited. For example, data affinity is random and the system does not support load-based scheduling policies. The scheduler may not make the best choice due to lack of the global resource view when selecting jobs.

  Superior Scheduler adopts multiple scheduling mechanisms. There are dedicated scheduling threads in Superior Scheduler, separating heartbeats with scheduling and preventing system heartbeat storms. Additionally, Superior Scheduler matches jobs with resources, providing each scheduled job with a global resource view and increasing the scheduling accuracy. Compared with the open source scheduler, Superior Scheduler excels in system throughput, resource usage, and data affinity.

Figure 5-161 Comparison of Superior Scheduler with open source schedulers



Apart from the enhanced system throughput and utilization, Superior Scheduler provides following major scheduling features:

- Multiple resource pools

  Multiple resource pools help logically divide cluster resources and share them among multiple tenants or queues. The division of resource pools supports heterogeneous resources. Resource pools can be divided exactly according to requirements on the application resource isolation. You can configure further policies for different queues in a pool.

- Multi-tenant scheduling (**reserve**, **min**, **share**, and **max**) in each resource pool

  Superior Scheduler provides flexible hierarchical multi-tenant scheduling policy. Different policies can be configured for different tenants or queues that can access different resource pools. The following figure lists supported policies:

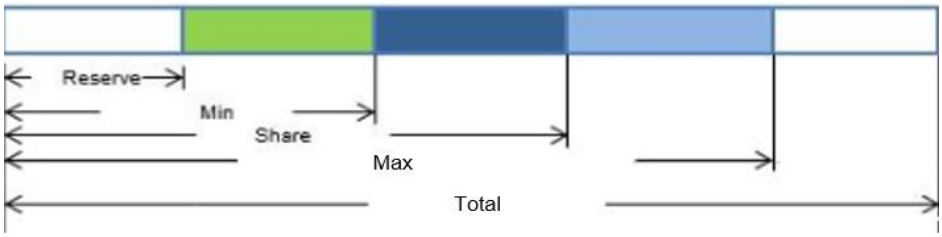
  Table 5-31 Policy description

Name	Description
reserve	This policy is used to reserve resources for a tenant. Even though tenant has no jobs available, other tenant cannot use the reserved resource. The value can be a percentage or an absolute value. If both the percentage and absolute value are configured, the percentage is automatically calculated into an absolute value, and the larger value is used. The default **reserve** value is **0**. Compared with the method of specifying a dedicated resource pool and hosts, the **reserve** policy provides a flexible floating reservation function. In addition, because no specific hosts are specified, the data affinity for calculation is improved and the impact by the faulty hosts is avoided.

Name	Description
min	This policy allows preemption of minimum resources. Other tenants can use these resources, but the current tenant has the priority to use them. The value can be a percentage or an absolute value. If both the percentage and absolute value are configured, the percentage is automatically calculated into an absolute value, and the larger value is used. The default value is **0**.
share	This policy is used for shared resources that cannot be preempted. To use these resources, the current tenant needs to wait for other tenants to complete jobs and release resources. The value can be a percentage or an absolute value.
max	This policy is used for the maximum resources that can be utilized. The tenant cannot obtain more resources than the allowed maximum value. The value can be a percentage or an absolute value. If both the percentage and absolute value are configured, the percentage is automatically calculated into an absolute value, and the larger value is used. By default value, there is no restriction on resources.

Figure 5-162 shows the tenant resource allocation policy.

Figure 5-162 Resource scheduling policies



NOTE

In the above figure, **Total** indicates the total number of resources, not the scheduling policy.

Compared with open source schedulers, Superior Scheduler supports both percentage and absolute value of tenants for allocating resources, flexibly addressing resource scheduling requirements of enterprise-level tenants. For example, resources can be allocated according to the absolute value of level-1 tenants, avoiding impact caused by changes of cluster scale. However, resources can be allocated according to the allocation percentage of sub-tenants, improving resource usages in the level-1 tenant.

- Heterogeneous and multi-dimensional resource scheduling

  Superior Scheduler supports following functions except CPU and memory scheduling:

  - Node labels can be used to identify multi-dimensional attributes of nodes such as **GPU_ENABLED** and **SSD_ENABLED**, and can be scheduled based on these labels.
  - Resource pools can be used to group resources of the same type and allocate them to specific tenants or queues.

- Fair scheduling of multiple users in a tenant

  In a leaf tenant, multiple users can use the same queue to submit jobs. Compared with the open source schedulers, Superior Scheduler supports configuring flexible resource sharing policy among different users in a same tenant. For example, VIP users can be configured with higher resource access weight.

- Data locality aware scheduling

  Superior Scheduler adopts the job-to-node scheduling policy. That is, Superior Scheduler attempts to schedule specified jobs between available nodes so that the selected node is suitable for the specified jobs. By doing so, the scheduler will have an overall view of the cluster and data. Localization is ensured if there is an opportunity to place tasks closer to the data. The open source scheduler uses the node-to-job scheduling policy to match the appropriate jobs to a given node.

- Dynamic resource reservation during container scheduling

  In a heterogeneous and diversified computing environment, some containers need more resources or multiple resources. For example, Spark job may require large memory. When such containers compete with containers requiring fewer resources, containers requiring more resources may not obtain sufficient resources within a reasonable period. Open source schedulers allocate resources to jobs, which may cause unreasonable resource reservation for these jobs. This mechanism leads to the waste of overall system resources. Superior Scheduler differs from open source schedulers in following aspects:

  - Requirement-based matching: Superior Scheduler schedules jobs to nodes and selects appropriate nodes to reserve resources to improve the startup time of containers and avoid waste.
  - Tenant rebalancing: When the reservation logic is enabled, the open source schedulers do not comply with the configured sharing policy. Superior Scheduler uses different methods. In each scheduling period, Superior Scheduler traverses all tenants and attempts to balance resources based on the multi-tenant policy. In addition, Superior Scheduler attempts to meet all policies (**reserve**, **min**, and **share**) to release reserved resources and direct available resources to other containers that should obtain resources under different tenants.

- Dynamic queue status control (**Open**/**Closed**/**Active**/**Inactive**)

  Multiple queue statuses are supported, helping MRS cluster administrators manage and maintain multiple tenants.

  - Open status (**Open/Closed**): If the status is **Open** by default, applications submitted to the queue are accepted. If the status is **Closed**, no application is accepted.
  - Active status (**Active/Inactive**): If the status is **Active** by default, resources can be scheduled and allocated to applications in the tenant. Resources will not be scheduled to queues in **Inactive** status.

- Application pending reason
  If the application is not started, provide the job pending reasons.

Table 5-32 describes the comparison result of Superior Scheduler and Yarn open source schedulers.

Table 5-32 Comparative analysis

Scheduling	Yarn Open Source Scheduler	Superior Scheduler
Multi-tenant scheduling	In homogeneous clusters, either Capacity Scheduler or Fair Scheduler can be selected and the cluster does not support Fair Scheduler. Capacity Scheduler supports the scheduling by percentage and Fair Scheduler supports the scheduling by absolute value.	• Supports heterogeneous clusters and multiple resource pools. • Supports **reservation** to ensure direct access to resources.
Data locality aware scheduling	The node-to-job scheduling policy reduces the success rate of data localization and potentially affects application execution performance.	The **job-to-node scheduling policy** can aware data location more accurately, and the job hit rate of data localization scheduling is higher.
Balanced scheduling based on load of hosts	Not supported	**Balanced scheduling can be achieved when Superior Scheduler considers the host load and resource allocation during scheduling.**
Fair scheduling of multiple users in a tenant	Not supported	Supports keywords **default** and **others**.
Job waiting reason	Not supported	Job waiting reasons illustrate why a job needs to wait.

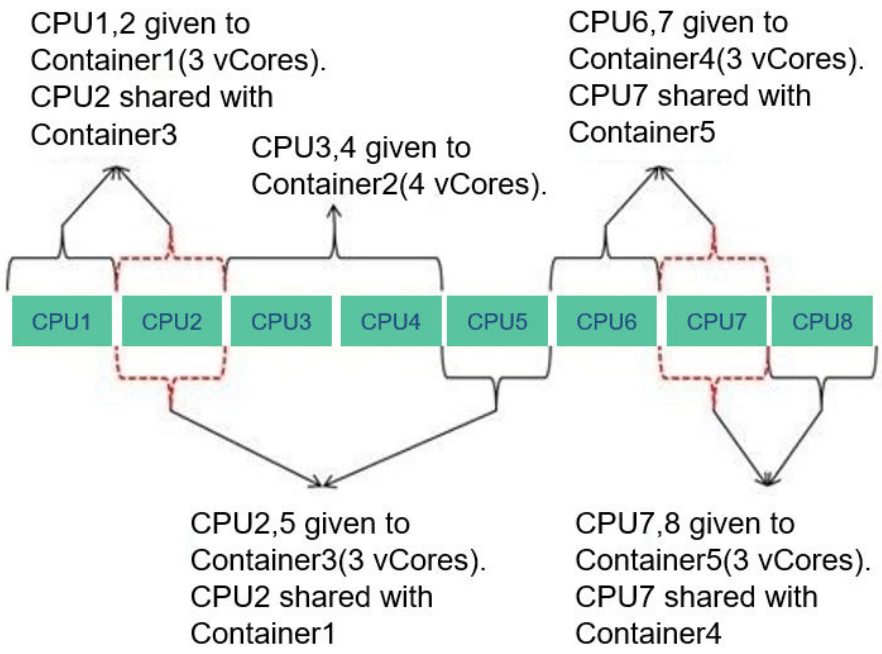
In conclusion, Superior Scheduler is a high-performance scheduler with various scheduling policies and is better than Capacity Scheduler in terms of functionality, performance, resource usage, and scalability.

CPU Hard Isolation

Yarn cannot strictly control the CPU resources used by each container. When the CPU subsystem is used, a container may occupy excessive resources. Therefore, CPUset is used to control resource allocation.

To solve this problem, the CPU resources are allocated to each container based on the ratio of virtual cores (vCores) to physical cores. If a container requires an entire physical core, the container has it. If a container needs only some physical cores, several containers may share the same physical core. The following figure shows an example of the CPU quota. The given ratio of vCores to physical cores is 2:1.

Figure 5-163 CPU quota

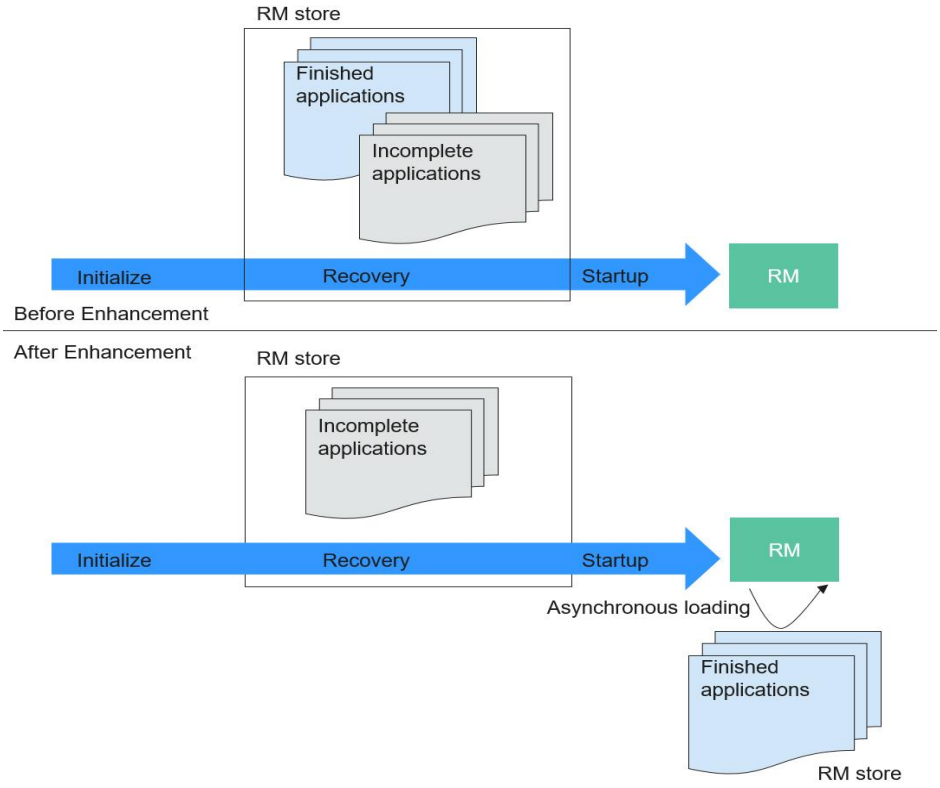


Enhanced Open Source Feature: Optimizing Restart Performance

Generally, the recovered ResourceManager can obtain running and completed applications. However, a large number of completed applications may cause problems such as slow startup and long HA switchover/restart time of ResourceManagers.

To speed up the startup, obtain the list of unfinished applications before starting the ResourceManagers. In this case, the completed application continues to be recovered in the background asynchronous thread. The following figure shows how the ResourceManager recovery starts.

Figure 5-164 Starting the ResourceManager recovery



5.38 Apache ZooKeeper

5.38.1 ZooKeeper Basic Principles

Overview

ZooKeeper is a distributed, highly available coordination service. ZooKeeper is used to provide following functions:

- Prevents the system from SPOFs and provides reliable services for applications.
- Provides distributed coordination services and manages configuration information.

Architecture

Nodes in a ZooKeeper cluster have three roles: Leader, Follower, and Observer, as shown in **Figure 5-165**. Generally, an odd number of (2N+1) ZooKeeper services need to be configured in the cluster, and at least (N+1) vote majority is required to successfully perform the write operation.

Figure 5-165 Architecture

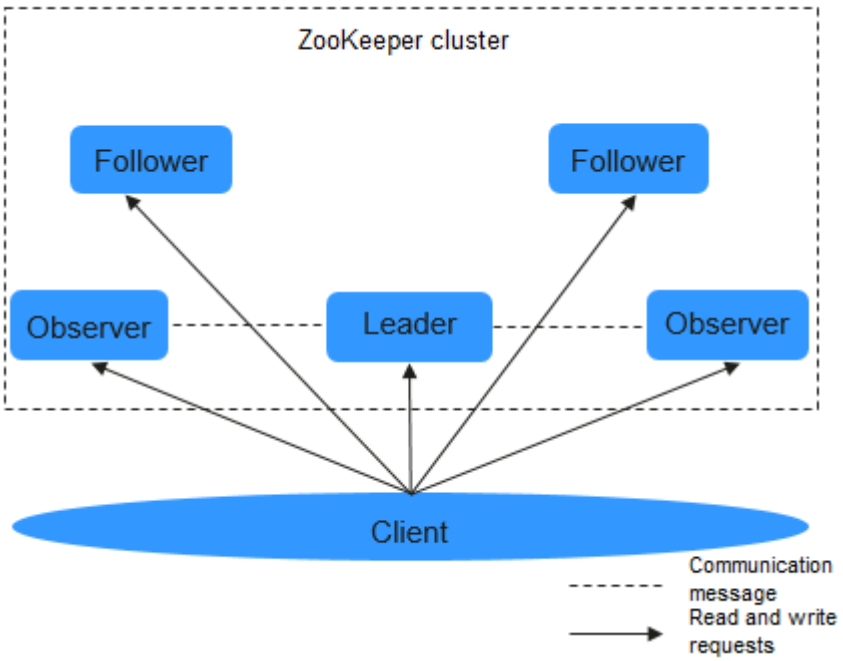


Table 5-33 describes the functions of each module shown in **Figure 5-165**.

Table 5-33 Architecture description

Name	Description
Leader	Only one node serves as the Leader in a ZooKeeper cluster. The Leader, elected by Followers using the ZooKeeper Atomic Broadcast (ZAB) protocol, receives and coordinates all write requests and synchronizes written information to Followers and Observers.
Follower	Follower has two functions: • Prevents SPOFs. A new Leader is elected from Followers when the Leader is faulty. • Processes read requests and interact with the Leader to process write requests.
Observer	The Observer does not take part in voting for election and write requests. It only processes read requests and forwards write requests to the Leader, increasing system processing efficiency.
Client	Reads and writes data from or to the ZooKeeper cluster. For example, HBase can serve as a ZooKeeper client and use the arbitration function of the ZooKeeper cluster to control the active/standby status of HMaster.

If security services are enabled in the cluster, authentication is required during the connection to ZooKeeper. The authentication modes are as follows:

- Keytab mode: You need to obtain a human-machine user from the MRS cluster administrator for MRS console login and authentication, and obtain the Keytab file of the user.
- Ticket mode: Obtain a human-machine user from the MRS cluster administrator for subsequent secure login, enable the renewable and forwardable functions of the Kerberos service, set the ticket update period, and restart Kerberos and related components.

  NOTE

  - By default, the validity period of the user password is 90 days. Therefore, the validity period of the obtained Keytab file is 90 days.
  - The parameters for enabling the renewable and forwardable functions and setting the ticket update interval are on the **System** tab of the Kerberos service configuration page. The ticket update interval can be set to **kdc_renew_lifetime** or **kdc_max_renewable_life** based on the actual situation.

Principles

- **Write Request**
  a. After the Follower or Observer receives a write request, the Follower or Observer sends the request to the Leader.
  b. The Leader coordinates Followers to determine whether to accept the write request by voting.
  c. If more than half of voters return a write success message, the Leader submits the write request and returns a success message. Otherwise, a failure message is returned.
  d. The Follower or Observer returns the processing results.
- **Read-Only Request**

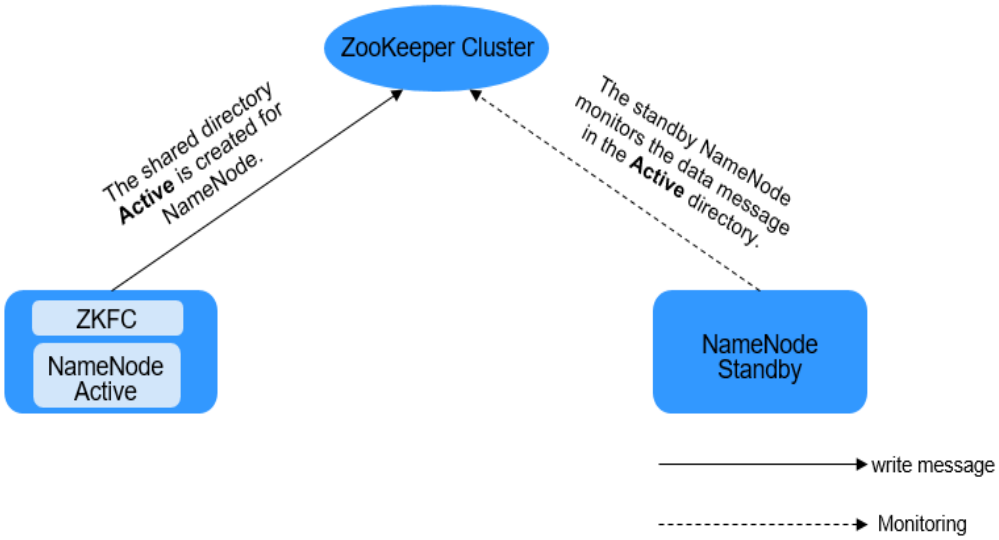
  The client directly reads data from the Leader, Follower, or Observer.

5.38.2 Relationships Between ZooKeeper and Other Components

ZooKeeper and HDFS

Figure 5-166 shows the relationship between ZooKeeper and HDFS.

Figure 5-166 Relationship between ZooKeeper and HDFS



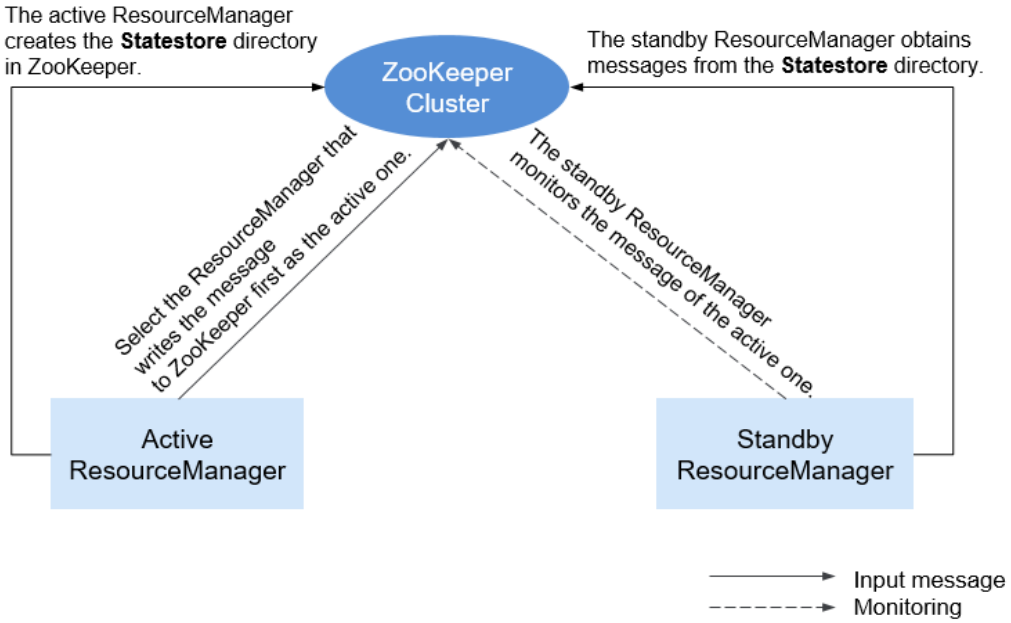
As the client of a ZooKeeper cluster, ZKFailoverController (ZKFC) monitors the status of NameNode. ZKFC is deployed only in the node where NameNode resides, and in both the active and standby HDFS NameNodes.

1. The ZKFC connects to ZooKeeper and saves information such as host names to ZooKeeper under the znode directory **/hadoop-ha**. NameNode that creates the directory first is considered as the active node, and the other is the standby node. NameNodes read the NameNode information periodically through ZooKeeper.
2. When the process of the active node ends abnormally, the standby NameNode detects changes in the **/hadoop-ha** directory through ZooKeeper, and then takes over the service of the active NameNode.

ZooKeeper and YARN

Figure 5-167 shows the relationship between ZooKeeper and YARN.

Figure 5-167 Relationship Between ZooKeeper and YARN

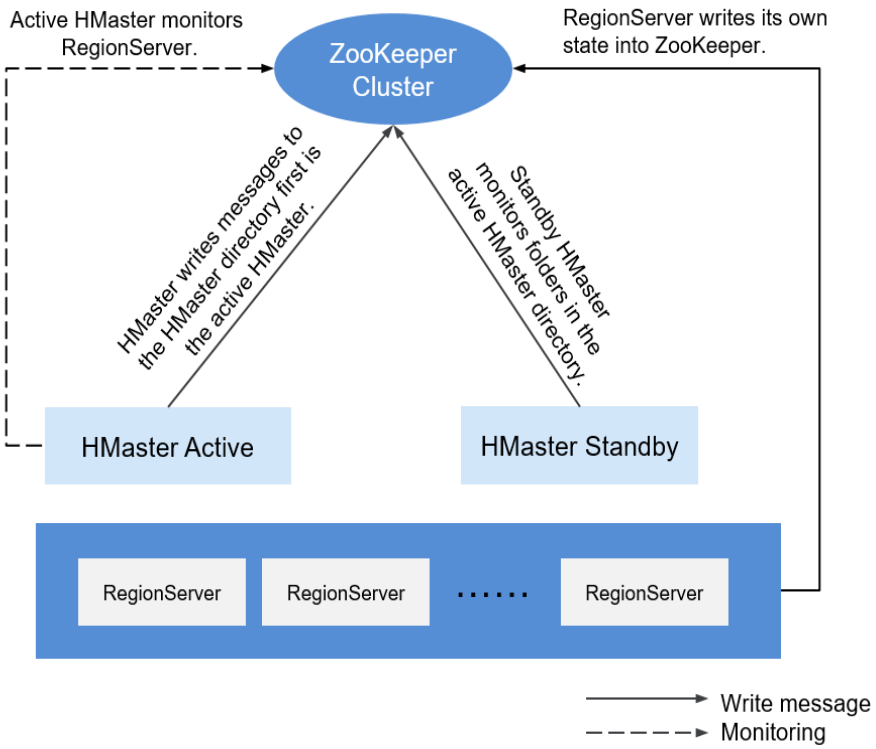


1. When the system is started, ResourceManager attempts to write state information to ZooKeeper. ResourceManager that first writes state information to ZooKeeper is selected as the active ResourceManager, and others are standby ResourceManagers. The standby ResourceManagers periodically monitor active ResourceManager election information in ZooKeeper.
2. The active ResourceManager creates the **Statestore** directory in ZooKeeper to store application information. If the active ResourceManager is faulty, the standby ResourceManager obtains application information from the **Statestore** directory and restores the data.

ZooKeeper and HBase

Figure 5-168 shows the relationship between ZooKeeper and HBase.

Figure 5-168 Relationship between ZooKeeper and HBase

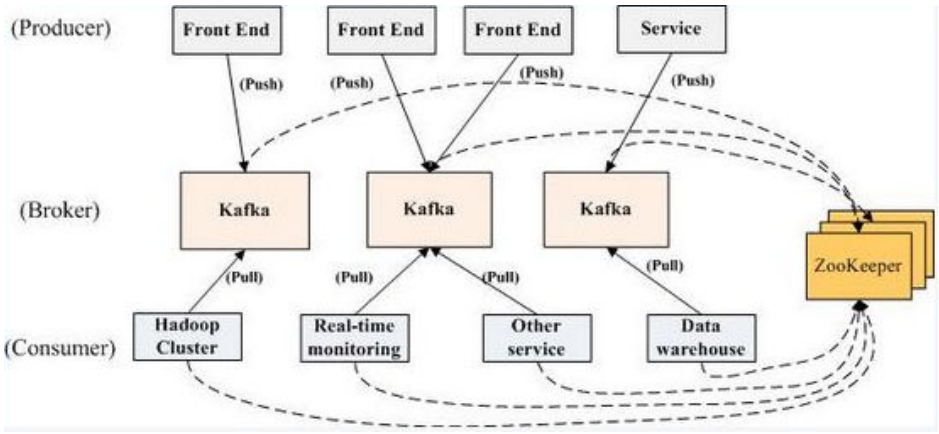


1. RegionServer registers itself to ZooKeeper on Ephemeral node. ZooKeeper stores the HBase information, including the HBase metadata and HMaster addresses.
2. HMaster detects the health status of each RegionServer using ZooKeeper, and monitors them.
3. HBase supports multiple HMaster nodes (like HDFS NameNodes). When the active HMatser is faulty, the standby HMaster obtains the state information about the entire cluster using ZooKeeper. That is, using ZooKeeper can avoid HBase SPOFs.

ZooKeeper and Kafka

Figure 5-169 shows the relationship between ZooKeeper and Kafka.

Figure 5-169 Relationship between ZooKeeper and Kafka



1. Broker uses ZooKeeper to register broker information and elect a partition leader.
2. The consumer uses ZooKeeper to register consumer information, including the partition list of consumer. In addition, ZooKeeper is used to discover the broker list, establish a socket connection with the partition leader, and obtain messages.

5.38.3 ZooKeeper Enhanced Open Source Features

Enhanced Log

In security mode, an ephemeral node is deleted as long as the session that created the node expires. Ephemeral node deletion is recorded in audit logs so that ephemeral node status can be obtained.

Usernames must be added to audit logs for all operations performed on ZooKeeper clients.

On the ZooKeeper client, create a znode, of which the Kerberos principal is **zkcli/hadoop.***<System domain name>***@***<System domain name>*.

For example, open the **<ZOO_LOG_DIR>/zookeeper_audit.log** file. The file content is as follows:

```
2016-12-28 14:17:10,505 | INFO  | CommitProcWorkThread-4 | session=0x12000007553b4903?
user=10.177.223.78,zkcli/hadoop.hadoop.com@HADOOP.COM?ip=10.177.223.78?operation=create znode?
target=ZooKeeperServer?znode=/test1?result=success
2016-12-28 14:17:10,530 | INFO  | CommitProcWorkThread-4 | session=0x12000007553b4903?
user=10.177.223.78,zkcli/hadoop.hadoop.com@HADOOP.COM?ip=10.177.223.78?operation=create znode?
target=ZooKeeperServer?znode=/test2?result=success
2016-12-28 14:17:10,550 | INFO  | CommitProcWorkThread-4 | session=0x12000007553b4903?
user=10.177.223.78,zkcli/hadoop.hadoop.com@HADOOP.COM?ip=10.177.223.78?operation=create znode?
target=ZooKeeperServer?znode=/test3?result=success
2016-12-28 14:17:10,570 | INFO  | CommitProcWorkThread-4 | session=0x12000007553b4903?
user=10.177.223.78,zkcli/hadoop.hadoop.com@HADOOP.COM?ip=10.177.223.78?operation=create znode?
target=ZooKeeperServer?znode=/test4?result=success
2016-12-28 14:17:10,592 | INFO  | CommitProcWorkThread-4 | session=0x12000007553b4903?
user=10.177.223.78,zkcli/hadoop.hadoop.com@HADOOP.COM?ip=10.177.223.78?operation=create znode?
target=ZooKeeperServer?znode=/test5?result=success
2016-12-28 14:17:10,613 | INFO  | CommitProcWorkThread-4 | session=0x12000007553b4903?
user=10.177.223.78,zkcli/hadoop.hadoop.com@HADOOP.COM?ip=10.177.223.78?operation=create znode?
target=ZooKeeperServer?znode=/test6?result=success
2016-12-28 14:17:10,633 | INFO  | CommitProcWorkThread-4 | session=0x12000007553b4903?
```



```
user=10.177.223.78,zkcli/hadoop.hadoop.com@HADOOP.COM?ip=10.177.223.78?operation=create znode?
target=ZooKeeperServer?znode=/test7?result=success
```

The content shows that logs of the ZooKeeper client user **zkcli/hadoop.hadoop.com@HADOOP.COM** are added to the audit log.

User details in ZooKeeper

In ZooKeeper, different authentication schemes use different credentials as users. Based on the authentication provider requirement, any parameter can be considered as users.

Example:

- **SAMLAuthenticationProvider** uses the client principal as a user.
- **X509AuthenticationProvider** uses the user client certificate as a user.
- **IAuthenticationProvider** uses the client IP address as a user.
- A username can be obtained from the custom authentication provider by implementing the **org.apache.zookeeper.server.auth.ExtAuthenticationProvider.getUserName(String)** method. If the method is not implemented, getting the username from the authentication provider instance will be skipped.

Enhanced Open Source Feature: ZooKeeper SSL Communication (Netty Connection)

The ZooKeeper design contains the Nio package and does not support SSL later than version 3.5. To solve this problem, Netty is added to ZooKeeper. Therefore, if you need to use SSL, enable Netty and set the following parameters on the server and client:

The open source server supports only plain text passwords, which may cause security problems. Therefore, such text passwords are no longer used on the server.

- Client
  a. Set **-Dzookeeper.client.secure** in the **zkCli.sh/zkEnv.sh** file to **true** to use secure communication on the client. Then, the client can connect to the secureClientPort on the server.
  b. Set the following parameters in the **zkCli.sh/zkEnv.sh** file to configure the client environment:

Parameter	Description
-Dzookeeper.clientCnxnSocket	Used for Netty communication between clients. Default value: **org.apache.zookeeper.ClientCnxnSocketNetty**
-Dzookeeper.ssl.keyStore.location	Indicates the path for storing the keystore file.
-Dzookeeper.ssl.keyStore.password	Encrypts a password.

Parameter	Description
-Dzookeeper.ssl.trustStore.location	Indicates the path for storing the truststore file.
-Dzookeeper.ssl.trustStore.password	Encrypts a password.
-Dzookeeper.config.crypt.class	Decrypts an encrypted password.
-Dzookeeper.ssl.password.encrypted	Default value: **false** If the keystore and truststore passwords are encrypted, set this parameter to **true**.
-Dzookeeper.ssl.enabled.protocols	Defines the SSL protocols to be enabled for the SSL context.
-Dzookeeper.ssl.exclude.cipher.ext	Defines the list of passwords separated by a comma which should be excluded from the SSL context.

NOTE

The preceding parameters must be set in the **zkCli.sh/zk.Env.sh** file.

- Server
  a. Set **secureClientPort** to **3381** in the **zoo.cfg** file.
  b. Set **zookeeper.serverCnxnFactory** to **org.apache.zookeeper.server.NettyServerCnxnFactory** in the **zoo.cfg** file on the server.
  c. Set the following parameters in the **zoo.cfg** file (in the **zookeeper/conf/zoo.cfg** path) to configure the server environment:

Parameter	Description
ssl.keyStore.location	Path for storing the **keystore.jks** file
ssl.keyStore.password	Encrypts a password.
ssl.trustStore.location	Indicates the path for storing the truststore file.
ssl.trustStore.password	Encrypts a password.
config.crypt.class	Decrypts an encrypted password.
ssl.keyStore.password.encrypted	Default value: **false** If this parameter is set to **true**, the encrypted password can be used.

Parameter	Description
ssl.trustStore.password.encrypted	Default value: **false** If this parameter is set to **true**, the encrypted password can be used.
ssl.enabled.protocols	Defines the SSL protocols to be enabled for the SSL context.
ssl.exclude.cipher.ext	Defines the list of passwords separated by a comma which should be excluded from the SSL context.

d. Start ZKserver and connect the security client to the security port.

- Credential

  The credential used between client and server in ZooKeeper is **X509AuthenticationProvider**. This credential is initialized using the server certificates specified and trusted by the following parameters:

  – zookeeper.ssl.keyStore.location
  – zookeeper.ssl.keyStore.password
  – zookeeper.ssl.trustStore.location
  – zookeeper.ssl.trustStore.password

  NOTE

  If you do not want to use default mechanism of ZooKeeper, then it can be configured with different trust mechanisms as needed.

6 Functions

6.1 Storage-Compute Decoupling

MRS stores data in the parallel file system of OBS 3.0 and uses its clusters for data computing. In this way, mass data analytics can be scaled up or down on demand at a low cost.

Currently, Flink, Hadoop (HDFS/Yarn/MapReduce), HBase, HetuEngine, Hive, Loader, Spark, and Hudi in MRS clusters can connect to OBS 3.0 to help implement storage-compute decoupling. MRS uses the Guardian component to connect to the OBS parallel file system and provide other components with the temporary authentication credentials and fine-grained permission control capabilities for accessing OBS.

NOTE

If the MRS storage and compute decoupling solution is used, note the following:

- In the storage-compute decoupling scenario, use the parallel file system of OBS 3.0 to store data. Do not use OBS buckets.
- Job submission based on the Guardian storage and compute decoupling management plane depends on JobGateWay instead of Executor.
- After an MRS cluster is interconnected with OBS, some function restrictions are as follows:
  - Some refined monitoring metrics collected based on the HDFS file system cannot be properly displayed.
  - The data snapshot, backup, and restoration functions of components are not supported.
  - The components do not support cluster active/standby DR.
  - The tools for migrating data from HBase and HDFS to Elasticsearch of the Elasticsearch component are not supported.
  - IPv6-based MRS clusters cannot connect to OBS using the Guardian service. PM clusters do not support job submission on the management console.

Configuring Storage and Compute Decoupling

1. Install the Guardian service.

   Install basic components, such as Guardian, Ranger, and Hadoop, in the MRS cluster in advance and manage PM clusters on the MRS console in advance.

2. Create an OBS agency.

   Create an agency with OBS access permissions, which is used for interconnecting Guardian with OBS.

3. Enable the interconnection between Guardian and OBS and configure parameters.

   Modify the configuration parameters for the Guardian service and configure the IAM agency authentication information.

4. Configure the policy for clearing component data in the recycle bin directory.

   In the storage-compute decoupling scenario, the prevention against accidental deletion is enabled by default for components connected to OBS. When a user deletes data, the deleted object is moved to the corresponding recycle bin directory. You need to configure a lifecycle rule for the **/user/.Trash** directory in the OBS file system to prevent the storage space from being used up.

5. Interconnect components with OBS.

   Components in the MRS cluster can directly access the corresponding path after the required permissions for accessing OBS buckets are obtained. You can use the component client to directly access resources in the OBS file system in absolute path mode.

Configuring OBS Permissions

If Guardian is deployed with storage and compute decoupled and Ranger authentication is enabled for MRS clusters, Ranger administrators can configure read and write permissions on OBS directories or files for cluster users.

With the Guardian permission model, storage and compute decoupling, and Hive cascading authorization, authorization is not required after the first permission service table authorization on the Ranger page and the system automatically associates the permissions of OBS data storage source in a fine-grained manner. The storage path of the table does not need to be sensed.

NOTE

- On the Ranger page, OBS permission authorization only support Manager custom user groups (built-in user groups are not supported). The user group contains a maximum of 52 characters, including digits 0 to 9, letters A to Z, underscores (_), and number signs (#). Otherwise, the policy fails to be added.
- For clusters in the security mode, Ranger is needed for permission authorization. For normal clusters, OBS permissions are granted by default and no additional configuration is required.

6.2 Multi-tenancy

Definition

Multi-tenancy refers to multiple resource sets (a resource set is a tenant) in the MRS big data cluster and is able to allocate and schedule resources. The resources include computing resources and storage resources.

Context

Modern enterprises' data clusters are becoming more and more centralized and cloud-based. Enterprise-class big data clusters must meet the following requirements:

- Carry data of different types and formats and run jobs and applications of different types (such analysis, query, and stream processing).
- Isolate data of a user from that of another user who has demanding requirements on data security, such as a bank or government institute.

The preceding requirements bring the following challenges to the big data clusters:

- Proper allocation and scheduling of resources to ensure stable operating of applications and jobs.
- Strict access control to ensure data and service security.

Multi-tenancy isolates the resources of a big data cluster into resource sets. Users can lease desired resource sets to run applications and jobs and store data. In a big data cluster, multiple resource sets can be deployed to meet diverse requirements of multiple users.

The MRS big data cluster provides a complete enterprise-class big data multi-tenant solution.

Highlights

- Proper resource configuration and isolation

  The resources of a tenant are isolated from those of another tenant. The resource use of a tenant does not affect other tenants. This mechanism ensures that each tenant can configure resources based on service requirements, improving resource utilization.
- Resource consumption measurement and statistics

  Tenants are system resource applicants and consumers. System resources are planned and allocated based on tenants. Resource consumption by tenants can be measured and collected.
- Assured data security and access security

  In multi-tenant scenarios, the data of each tenant is stored separately to ensure data security. The access to tenants' resources is controlled to ensure access security.

6.3 Multi-Service

Introduction

The multi-service feature means that you do not need to define multiple sets of components. Manager allows you to install multiple sets of the same component in a cluster to better solve resource isolation or performance problems.

The newly added instances have the same functional modules as existing services, such as logs, users, and shell commands. Manager provides unified management

for HBase, Hive, and Spark instances, including monitoring, alarming, and starting or stopping services. When importing and exporting data using Loader, creating roles, backing up and restoring data, or developing applications, the system administrator needs to select specific service instances based on the actual situation.

The multi-service feature can linearly improve the overall service performance. The service instance resources can be customized. Tenants can associate with different service instances to enable services to run in isolated resources, improving customer satisfaction and user experience.

NOTE

- The three sets of HBase components (HBase, HBase-1, and HBase-2) installed in the same cluster are called three service instances.
- If multiple Elasticsearch services are installed in the same cluster, ensure that all Elasticsearch services are in security mode or non-security mode.
- Physical machine clusters support the multi-service feature, whereas the ECS/BMS clusters do not support this feature.

Constraints

1. The multi-service feature does not support co-host deployment. Specifically, multiple services and roles of the same service cannot be deployed on the same host.
2. The multi-service feature does not allow a service to connect to two underlying services at the same time.

   For example, one Hive service cannot be connected to multiple DBServices.

6.4 Cluster Node Hybrid Configuration

Overview

The TaiShan server in the MRS physical machine cluster can be used together with the x86 server to provide users with the cluster deployment capability that is compatible with the TaiShan and x86 platforms. In addition, the dual-platform client installation package can be downloaded.

During installation, an x86 server (or a TaiShan server) with one architecture is used to complete cluster deployment, and an installation package of a client with another operating system or architecture is registered with the cluster, so that the cluster can be supported on the x86 cluster (or the TaiShan cluster). Use the new TaiShan server (or x86 server) client. In addition, the cluster supports capacity expansion using servers of different architectures or operating systems, instead of replacing all x86 servers, protecting existing investments.

Guidelines

For details about the x86 and TaiShan hybrid deployment, see **Table 6-1**.

Table 6-1 Hybrid deployment

Scenario	Description
Hybrid cluster installation	Nodes installed in the same batch must use servers of the same architecture, x86 or TaiShan.
Registering component software packages	The cluster does not support cross-platform component package registration. In the hybrid deployment scenario, only component packages of the same platform type can be registered at the same time.
Client downloading	By default, there is only one type of client installation package on the server. Inject another type of installation package into the cluster and then download the client. For example, by default, the TaiShan cluster server does not have the x86 client installation package. Register the x86 software package with the TaiShan cluster and then download the x86 client from the cluster.
Secondary development	Configure a third-party JAR package if the package is not supported by x86 and TaiShan platforms in the cluster.
Capacity expansion/ reduction or cluster migration	Both the x86 and TaiShan servers are supported for capacity expansion.

6.5 Cross-AZ HA for a single cluster

MRS provides the HA capability for a single physical machine cluster. Nodes in a physical machine cluster are divided into three AZs. Each AZ contains multiple data nodes and control nodes. When an AZ domain is faulty, all or some upper-layer services are not affected.

Currently, the following components support cross-AZ HA: CDL, ClickHouse, DBService, Elasticsearch, Flink, FTP-Server, HBase, HDFS, HetuEngine, Hive, Hue, Kafka, KrbServer, LakeSearch, LdapServer, Loader, MapReduce, Oozie, Redis, Spark, Tez, Yarn, and ZooKeeper.

NOTE

- Different AZs must be in the same network segment, and the cross-AZ network latency must be within 2 ms.
- In the single-cluster cross-AZ solution, Yarn supports only the Superior scheduler.
- It is recommended that the compute nodes, OSs, and basic system configurations (CPU, memory, and disk capacity) of each AZ be the same.
- This function applies only to MRS physical machine clusters.

AZ-level Block Placement Policy Supported by HDFS

HDFS supports AZ-level BPP (that is, cross-AZ replica placement policy). You can flexibly specify the HDFS directory and the number of replicas stored in a target AZ. The first replica is written to the AZ where the HDFS client is located by default.

The system can detect and determine the fault of an AZ. When a replica is written to the faulty AZ, the system ignores the faulty AZ, but other normal AZs continue to write the replica based on the BPP.

AZ-level BPP Mover Supported by HDFS

HDFS supports AZ-level BPP Mover, which verifies the consistency between the AZ of the BPP in the HDFS data directory and the AZ where the replica is actually distributed.

AZ-level Task Scheduling Supported by the Superior Scheduler

The Superior scheduler allows tasks to run in only one AZ or in each AZ in load balancing mode.

The system can detect and determine the fault of an AZ. When a single AZ is faulty, the running tasks are transferred to other normal AZs.

AZ-level HA Supported by Elasticsearch

Elasticsearch shards can be evenly distributed among AZs. When the total number of primary shards and replica shards is greater than or equal to the number of AZs, each AZ can store a complete and independent replica of data. If the total number of primary and replica shards of an index is less than the number of AZs, you can increase the number of replicas by referring to "Running curl Commands in Linux" > "Setting Index Replicas" in *MapReduce Service (MRS) 3.3.1-LTS User Guide (for Huawei Cloud Stack 8.3.1) in the MapReduce Service (MRS) 3.3.1-LTS Usage Guide (for Huawei Cloud Stack 8.3.1)* to ensure that the total number of primary and replica shards is greater than or equal to the number of AZs.

AZ-level HA Supported by Kafka

- Leaders of all partitions of all new topics can be evenly distributed among AZs.
- Replicas of the same partition are distributed in different AZs.
- A reallocation scheme can be generated to balance the partitions of topics created before the HA feature is enabled among AZs.

AZ-Level HA Supported by ClickHouse

Both ClickHouseServer and ClickHouseBalancer support cross-AZ HA.

Plan the ClickHouse role deployment in advance by referring to the cross-AZ instance deployment of a logical cluster. There are no constraints during the installation.

The constraints on cross-AZ instance deployment in a logical ClickHouse cluster are as follows:

- When a logical cluster is deployed, the number of instances is an integer multiple of the number of replicas. A single replica does not support cross-AZ HA.
- For a dual-replica cluster, the total number of ClickHouseServers in two AZs must be at least the number of ClickHouseServers in the other AZ.
- For a three-replica cluster, the number of ClickHouseServers must be evenly distributed in each AZ. That is, the difference between the number of ClickHouseServers in different AZs cannot be greater than 1.
- The ClickHouseBalancer instance must be deployed in two or more AZs.

AZ-Level HA Supported by Redis

Cross-AZ HA is supported only for logical Redis clusters rather than single Redis instances.

When deploying Redis_*N* instances, deploy them in three AZs and ensure that the number of instances in any AZ is less than the total number of rest instances.

Constraints on instance deployment in a Redis logical cluster:

- The number of instances in any AZ is less than the sum of instances in the other two AZs.
- The active and standby instances in a Redis cluster cannot be deployed in the same AZ. When a logical Redis cluster is created, scaled out, or scaled in, the system automatically allocates the active and standby instances to different AZs.
- For Redis logical clusters created before single-cluster cross-AZ HA is enabled, AZ-level HA is not supported. You need to delete the clusters and create them again.

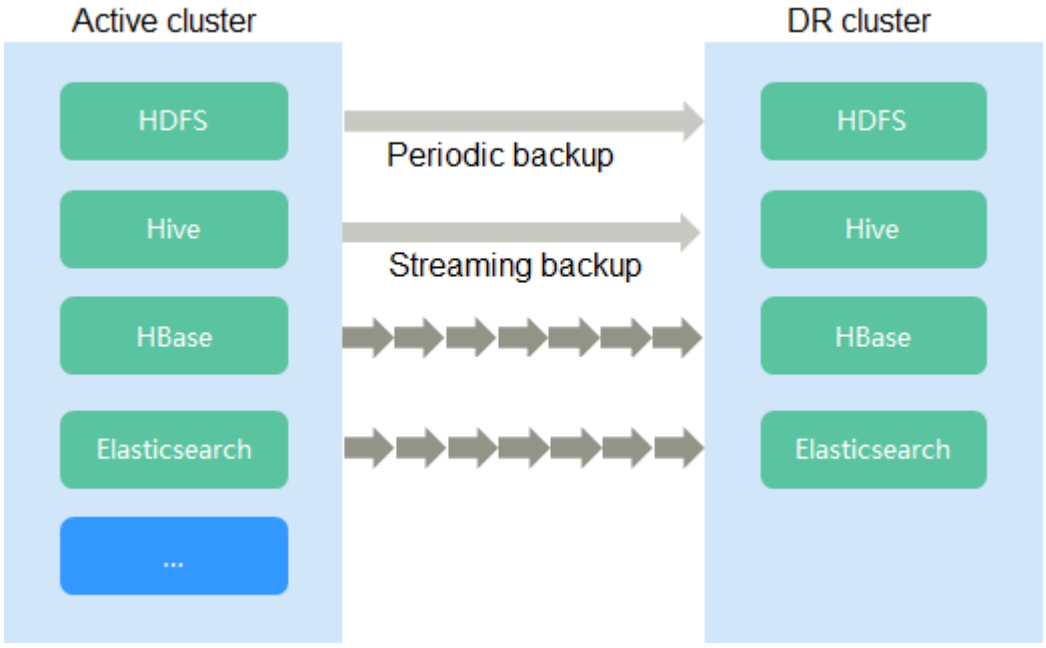
6.6 Active/Standby Cluster DR

MRS provides a remote disaster recovery (DR) solution based on active and DR clusters. The data replication relationship between active and DR clusters ensures data reliability and service continuity in the cluster. If a production center encounters a disaster, protected service data can be restored from the remote DR center.

NOTICE

The active/standby DR feature is restricted. To use this feature, contact Huawei technical support.

Figure 6-1 Active/Standby cluster DR



- "Active" and "DR" indicate a cluster's service status instead of the current running status. The roles of active and DR clusters are fixed and do not change with the running status. In the normal state, an active cluster is used to run services, and a DR cluster is used for backup. In the DR state, a DR cluster is used to run services, and an active cluster is used for backup.
- One active cluster maps one DR cluster. Currently, the following configurations are not supported: One active cluster maps multiple DR clusters (different data is backed up to different clusters), or one DR cluster maps multiple active clusters.
- A DR cluster can be different from the active cluster but must have the service that requires data DR in the active cluster.

Data components in an MRS cluster that can be configured with DR protection include HDFS, Hive, HBase, Elasticsearch, Flink, and Redis. Data backup of protected objects is classified into periodic backup and streaming backup by data type.

- Periodic backup: The system periodically backs up data of protected objects from the active cluster to the DR cluster based on a specified DR protection policy. Components corresponding to periodic backup include HDFS, Hive, and Flink.
- Streaming backup: The system backs up streaming data of the protected objects from the active cluster to the DR cluster based on an expected recovery point objective (RPO). Streaming replication can be implemented on components like HBase, Elasticsearch, and Redis.

After a DR relationship is established between two clusters, you can configure a protection group to specify protected objects. Multiple protection groups can be created. Each protection group can contain one or more components, except streaming components. (A protection group can contain only one streaming component.) Protected objects in different protection groups cannot be the same or have inclusion relationships.

To ensure data access, manually added machine-machine and human-machine users in the active cluster are automatically synchronized to the DR cluster. After human-machine users are synchronized, the original passwords are changed to random ones. If human-machine users are used to run services after active/standby switchover, an administrator needs to reset passwords of these users and these users need to change the passwords upon the first login. Users created in DR clusters will not be synchronized to the active cluster.

If Ranger authentication is enabled for a cluster, the system also synchronizes the authentication information. For a protection group of the periodic backup type, the synchronization of Ranger authentication information is started when each data backup task of the protection group is complete. For streaming backup, Ranger authentication information is synchronized every 10 minutes.

NOTE

- If Ranger DR has been configured for a cluster in security mode, you do not need to add or modify Ranger policies in the standby cluster. The active cluster periodically synchronizes policies and overwrites the Ranger policy on the standby cluster. To modify the Ranger policy, perform operations on the active cluster. User modifications in DR clusters will not be synchronized to the active cluster.
- If Ranger authentication is enabled for a cluster in normal mode, policies cannot be synchronized between active and standby nodes.

6.7 Rolling Restart and Upgrade

Rolling Restart

Rolling restart refers that after the software of a service or role instance is updated or the configuration is modified in a cluster, related objects are restarted without interrupting services.

Conventional common restart (restarting all instances simultaneously) interrupts services. Rolling restart adopts different restart policies for different instance running features to ensure service continuity. However, rolling restart takes a long time and exerts an impact on the throughput and performance of corresponding services.

NOTE

Before performing a rolling restart of instances, ensure that the internal and external interfaces are compatible before and after the rolling restart. If the interfaces are incompatible after a major version update, perform a common restart.

- Rolling restart policy for active and standby instances

  For roles that support high availability (HA), such as the HDFS NameNode, perform a rolling restart on the standby instance first, manually trigger an active/standby switchover, and then restart the original active instance after the switchover.

- Rolling restart policy for the Leader instance

  Each instance of a role is divided into a Leader node and multiple Follower nodes. Therefore, the services are not interrupted when an instance is restarted. In this case, restart all instances one by one. The Leader instance is restarted at last.

- Concurrent rolling restart policy for batch instances

  In a role, m (m ≥ 1) instances are restarted concurrently in rolling mode in each batch to ensure service continuity. This policy applies to roles that do not have functional differences between instances.

  For example, if you restart one HDFS ZKFC once, the service is not interrupted. Therefore, this policy can be used and the concurrent value is 1.

- Rolling restart by instance

  For a role configured with this policy, one instance is restarted in rolling mode each time to ensure that workload of the corresponding service is not interrupted.

  For example, for roles EsNode 1 to EsNode 9 of Elasticsearch, one instance is restarted each time to ensure that at least one shard of an index is available at each moment.

- Dynamic policy

  During RegionServer rolling restart, set the number of concurrences in each batch based on the number of instances deployed in RegionServer.

- Concurrent rack rolling restart policy

  This policy applies to roles that supports the rack awareness function (such as HDFS DataNode) and whose instances belong to two or more racks. Therefore, services are not interrupted when a rack is restarted. When roles meet the preceding conditions, all corresponding instances in each rack are restarted concurrently.

  If each rack contains many instances, divide sub-batches based on the maximum number of concurrent instances configured in the rack policy.

Rolling Upgrade

Rolling upgrade is an online upgrade mode. During the upgrade process, service interruption interval can be minimized.

Components that support rolling upgrade can provide all or part of their services. Component services that do not support rolling upgrade are interrupted during the upgrade process. Compared with the offline upgrade mode, rolling upgrade can ensure that part of services are available during product upgrade.

For rolling upgrade operations and precautions of each service, see corresponding upgrade guide.

Impact of Rolling Restart on the System

Rolling upgrade depends on rolling restart. The impact of rolling restart on the system also applies to rolling upgrade.

The following table describes the impact of rolling restart on each service. The services listed in the table support rolling restart. (KrbServer, LDAPServer, and DBService are internal services in the cluster and are not described in the table.).

Table 6-2 Impact on the system

Service Name	Service Interruption	Affected Service
CDL	The CDL service is normal during rolling restart.	The CDLService UI cannot be accessed.
ClickHouse	During the rolling restart, if the submitted services are executed within the timeout period (30 minutes by default), the services are not affected.	The node that is performing a rolling restart rejects all new requests, which affects single-replica services and the on cluster operation. If a request that is being executed is not complete within the timeout period (30 minutes by default), the request fails.
Elasticsearch	The read and write services of Elasticsearch are normal during the rolling restart only if the following conditions are met: 1. The status of the Elasticsearch cluster is **green**. 2. Each shard of an index has at least one primary shard and one replica shard.	The write performance deteriorates during rolling restart.
Flink	All services are normal.	The FlinkServer UI cannot be accessed.
Flume	To prevent service interruption and data loss, the following conditions must be met: • Persistent cache, for example File Channel, is used. • Flume cascading is configured. • Flume client sink supports fault migration or load balancing.	• Data loss may occur if channels do not have persistent cache. • The performance deteriorates for a short period of time during the fault migration.

Service Name	Service Interruption	Affected Service
GraphBase	Real-time data import and batch data import are normal.	• REST API requests need to be authenticated again. • Gremlin Console needs reconnection. • Gremlin Java APIs need reconnection. • GraphServer processing performance deteriorates.
HBase	HBase read and write services are normal.	• Real-time read and write performance may deteriorate during RegionServer rolling restart. • Real-time read/write services (not including BulkLoad) of HMaster are normal. Other services are affected. Creating a table (**create**) Creating a Namespace (**create_namespace**) Disabling a table (**disable** and **disable_all**) Re-creating the table (**truncate** and **truncate_preserve**) Moving a region (**move**) Getting a region offline (**unassign**) Combining regions (**merge_region**) Splitting a region (**split**) Enabling balance (**balance_switch**) DR operations (**add_peer**, **remove_peer**, **enable_table_replication**, **disable_peer**, **show_peer_tableCFs**, **set_peer_tableCFs**, **enable_peer**, **disable_table_replication**, **set_clusterState_active**, and **set_clusterState_standby**) Querying the cluster status (**status**)

Service Name	Service Interruption	Affected Service
HDFS	• An active/standby switchover is triggered for NameNodes. During switching, there is temporarily no active NameNode. As a result, the HDFS unavailable alarm may be reported, and running read/write tasks may be in errors. However, HDFS services are not interrupted. • If DataNodes are restarted in rolling mode, errors may be reported for some data that is being read or written. Read/write speed is affected when the client retries. • During the rolling restart of ZKFC, an active/standby NameNode switchover occurs.	None

Service Name	Service Interruption	Affected Service
HetuEngine	• At least two HSFabric instances exist, and at least two HSFabric instances are used for interconnection. Cross-domain services are not interrupted during the rolling restart. • HetuEngine services are not interrupted during the rolling restart of HSBroker, HSConsole, and HSConsole. • The rolling restart of HetuEngine compute instances can be performed only when there are at least two Coordinator and Worker nodes respectively. The rolling restart of computing instances does not interrupt services.	• HSFabric nodes in the middle of rolling restart reject all new requests. SQL requests that are being executed will fail if they are not completed within the timeout period (30 minutes by default). • During rolling restart, O&M operations are not available on the HSConsole page. • During the rolling restart of Hetu Engine compute instances, the performance decreases by approximately 20%. If the memory consumed by SQL execution exceeds 80% of **query_max_total_memory**, SQL tasks fail.
Hive	Hive services are normal.	• If the execution time of an existing task exceeds the timeout interval of rolling restart, the task may fail during the restart. You can retry the task if it fails. • Nodes in the middle of rolling start do not receive new requests. The number of requests processed by other instance nodes increases, and these nodes occupy more resources.

Service Name	Service Interruption	Affected Service
IoTDB	IoTDB read and write are normal.	• Some metadata operations are unavailable, including creating and deleting databases, deleting time series, creating, deleting, and exporting device snapshots, and performing permission operations. • Temporary read inconsistency may occur. • The read/write performance deteriorates.
Kafka	During rolling restart, the read and write of Kafka topics with multiple replicas are normal, but operations on Kafka topics with only a single replica are interrupted.	• Topics or partitions cannot be added, deleted, or modified. • If **acks** is set to **1** or **0** in Producer, the next Broker will be forcibly restarted if the data of the copy is not synchronized within 30 minutes during rolling restarting. For a dual-copy Partition whose replicas are in the two Brokers that are started consecutively, if the **unclean.leader.election.enable** parameter is **true** in the server configuration, the data may be lost; if the **unclean.leader.election.enable** is set to **false**, the Partition may have no leader for a period of time until the latter Broker is started.
Ranger	All services are normal.	None
Redis	Redis read and write operations are normal.	Capacity expansion or reduction for Redis clusters cannot be performed.
Solr	Solr read and write are normal only if each index has at least one primary shard (leader) and one replica shard (replica).	Write performance deteriorates during rolling restart.

Service Name	Service Interruption	Affected Service
Spark	Except the listed items, other services are not affected.	• When HBase is restarted, you cannot create or delete Spark on HBase tables in Spark. • When HBase is restarted, an active/standby switchover is triggered for HMaster. During the switching, the Spark on HBase function is unavailable. • If you have used the advanced API of Kafka, interruption may occur when Spark reads/writes data from/to Kafka during the rolling restart, and data may be lost. • During the rolling restart of ZooKeeper, spark-beeline fails to be started. • During the rolling restart of Yarn, Spark tasks fail. • When Spark's JDBCServer is restarted, long-running tasks will be stopped.
Yarn	• An active/standby switchover is triggered for ResourceManager nodes. Running tasks will cause errors, but services are not interrupted. • During the rolling restart of NodeManager, containers submitted to the node may be retried on other nodes.	Excessive task retries during Yarn's rolling restart may lead to service anomalies.
ZooKeeper	ZooKeeper read and write operations are normal.	• The rolling restart affects the ClickHouse service. During instance restart, ClickHouse tables are read-only on each quorumpeer instance for approximately 10 seconds. • The rolling restart causes ZooKeeper disconnection in Loader HA. Loader automatically retries for three times, each for 10 seconds. If ZooKeeper still cannot be connected, the active LoaderServer becomes a standby LoaderServer and then the active LoaderServer will be elected.

Service Name	Service Interruption	Affected Service
MOTService	None	During the rolling restart, an active/standby switchover occurs. During the active/standby switchover, read and write operations are unavailable for approximately 30 seconds.
Containers	All services are normal.	• If BLU is deployed on only one instance, services will be unavailable before the instance status becomes **Good**. • If BLU is deployed on multiple instances, services are not affected.
RTDService	All services are normal.	None
Guardian	All services are normal.	None
JobGateway	There is a possibility that submitted jobs fail.	• Jobs that have been submitted to YARN before rolling restart are not affected. • During the rolling restart, retry the requests fail to be responded by the job interface on the client. • There is a possibility that jobs submitted during rolling restart fail. This impact continues during rolling restart. You can retry the requests on the client.
LakeSearch	There is a possibility that submitted jobs fail.	During the rolling restart, retry the requests fail to be responded by the job interface.
Doris	Doris services are not interrupted during rolling restart only when the following conditions are met: • DBalancer is used to connect to Doris for job submission. • Tables read and written must have multiple replicas (three copies are recommended).	During rolling restart, retry jobs that did not respond to requests.

Rolling Restart Duration Reference

The rolling restart of all roles in the cluster is performed one by one. The following table lists the rolling restart duration of a single instance.

Table 6-3 Rolling restart duration of a single instance of different roles

Service Name	Role Name	Time Required
ClickHouse	ClickHouseServer	30 min
	ClickHouseBalancer	10 min
CDL	CDLConnector	1.5 minutes x Number of instances
	CDLService	2 minutes x Number of instances
Containers	WebContainer_1~5	Rolling upgrade is not involved.
Doris	FE	5 min
	BE	5 min
	DBroker	5 min
Elasticsearch	EsNode1~9	30 minutes x Number of instances
	EsMaster	5 min
Flume	Flume	3 minutes x Number of instances
	MonitorServer	1 minutes x Number of instances
Flink	FlinkServer	3min
	FlinkSource	1min
FTP-Server	FTP-Server	1 min
GraphBase	GraphServer	1 min
	LoadBalancer	2 min
HBase	RegionServer	If a RegionServer has 2,000 regions and the number of requests sent to a RegionServer per second is less than 2,000, the rolling restart takes 15 minutes.
HetuEngine	HSBroker	2 minutes x Number of instances
	HSConsole	2 minutes x Number of instances
	HSFabric	30 minutes x Number of instances
	QAS	1 minute x Number of instances

Service Name	Role Name	Time Required
	Compute instance	Rolling upgrade is not involved.
HDFS	DataNode	2 min
	JournalNode	2 min
	NameNode	4 min + x *x* indicates the NameNode metadata loading duration. It takes about 2 minutes to load 10,000,000 files. For example, *x* is 10 minutes for 50 million files. The startup duration fluctuates with reporting of DataNode data blocks.
	Zkfc	2 min
Hive	HiveServer	1 min
	MetaStore	1 min
Hue	Hue	Rolling upgrade is not involved.
IoTDB	IoTDBServer	3 min
Guardian	TokenServer	Rolling upgrade is not involved.
Kafka	Broker	30 min
	Kafka UI	5 min
KMS	KMSWebServer	2 min
Mapreduce	JobHistoryServer	1 min
MOTService	MOTServer	30 min
Oozie	oozie	Rolling upgrade is not involved.
Ranger	RangerAdmin	5min
	UserSync	3min
	TagSync	2min
Redis	Redis_1, Redis_2, Redis_3...	40 min
RTDService	RTDServer	Rolling upgrade is not involved.
Solr	SolrServerAdmin, SolrServer1-5	30 minutes x Number of instances
Spark	IndexServer	2min

Service Name	Role Name	Time Required
	JDBCServer	6min
	JobHistory	2min
	SparkResource	6min
Yarn	NodeManager	0.5 min
	ResourceManager	2 min
	TimelineServer	0.5 min
Zookeeper	quorumpeer	1min
JobGateway	JobBalance	6min
	JobServer	3 minutes x Number of instances
LakeSearch	SearchFactory	5min
	SearchServer	5min

The following uses HDFS's DataNode role as an example to describe how to calculate the duration of the rolling restart.

- If the rack strategy is disabled and the concurrency is 1, the rolling restart of a single DataNode instance takes about 2 minutes and the duration increases linearly with the number of nodes.
  - 100 nodes: about 3.3 hours.
  - 500 nodes: about 16.7 hours.
  - 1,000 nodes: about 33.3 hours.
- If the rack policy is enabled, restart racks in batches. The restart takes effect only when HDFS or Yarn is restarted. If the number of instances on a single rack is greater than 20, the number of concurrent tasks is fixed to 20. If the value is less than 20, the actual value is used. In this case, changing the value of **Data Nodes to Be Batch Restarted** in the **advanced options** does not take effect. The rolling restart of a single batch takes about 2 minutes and the duration increases linearly with the number of nodes.
  - 100 nodes (calculated based on 20 concurrent users): about 10 minutes.
  - 500 nodes (calculated based on 20 concurrent users): about 50 minutes.
  - 1,000 nodes (calculated based on 20 concurrent users): about 100 minutes.

6.8 Security Enhanced Features

Huawei MRS is a platform for massive data management and analysis and features high security. It ensures user data and service running security from the following aspects:

- Network isolation

  Huawei MRS divides the entire network into two planes: the service plane and management plane. The two planes are physically isolated to ensure security of the service and management networks.

  - MRS interworks with the service network through the service plane to provide service channels, data storage and access, task submission, and computing capabilities for enterprise users.
  - MRS interworks with the operation and maintenance (O&M) network through the management plane to provide the management and maintenance functions, especially cluster management and cluster monitoring, configuration, auditing, and user management services for enterprise users.

- Host security

  Users can deploy third-party antivirus software based on their service requirements. For the operating system (OS) and interfaces, Huawei MRS provides the following security measures:

  - Hardening OS kernel security
  - Installing the latest OS patch
  - Controlling the OS rights
  - Managing OS interfaces
  - Preventing the OS protocols and interfaces from attacks

- Application security

  Huawei MRS provides the following measures to ensure proper running of big data services:

  - Identity authentication
  - Web application security
  - Access control
  - Auditing security
  - Password security

- Data security

  For massive user data, Huawei MRS provides the following measures to ensure data confidentiality, integrity, and availability:

  - Disaster recovery (DR): MRS provides the remote DR function by configuring the active/standby cluster relationship and data tables to be synchronized. When data of the active cluster is damaged due to disasters, such as flood or earthquake, the standby cluster immediately takes over services.
  - Backup: MRS provides backup for metadata on the OMS, ClickHouse, DBService, Elasticsearch, Flink, HBase, IoTDB, Kafka, NameNode, MOTService, RTDService, Containers and Solr. MRS also provides backup for service data on the ClickHouse, Elasticsearch, HBase, HDFS, Hive, IoTDB, Redis, MOTService and Solr.

- Data integrity

  Data verification ensures data integrity during storage and transmission.

  - User data is stored on the HDFS. The HDFS verifies data correctness using CRC32C.

  - The DataNode of the HDFS stores and verifies data. If data sent from the client is abnormal (incomplete), the DataNode sends an error message to the client and requires the client to rewrite the data.
  - When the client reads data from the DataNode, the client also checks the data integrity. If the data is incomplete, the client reads data from other DataNodes.
- Data confidentiality

  The HDFS incorporates encrypted storage for file contents based on the Apache Hadoop version to prevent sensitive data being stored in plain text and improves the data security. Service applications need only to encrypt specified sensitive data. The data encryption and decryption processes are unknown to enterprise users. In addition, Hive implements table-level encryption, and HBase implements column-level encryption. During data creation, specify the encryption algorithm to ensure encrypted storage of sensitive data.

  The data confidentiality is ensured by encrypted data storage and access control.

  - The HBase compresses data before storing the data to the HDFS. In addition, users can configure the AES and SMS4 algorithms to ensure encrypted storage.
  - Each component supports setting of access rights for local data directories. Unauthorized users cannot access the data.
  - Information about users in a cluster is stored in encrypted mode.
- Security authentication
  - The unified user- and role-based authentication system complies with the role-based access control model to manage rights based on the role, ensuring batch user rights authorization.
  - MRS supports the security protocol Kerberos, uses the LDAP server as the account management system, and authenticates account information using Kerberos.
  - MRS provides single sign-on (SSO) to provide unified management and authentication for system users and component users of MRS.
  - MRS provides auditing for users logging in to FusionInsight Manager.
  - MRS provides the unified certificate management function, which allows certificates of the entire cluster to be configured and replaced in a unified manner on the portal. This makes users' certification replacement easier.

6.9 Reliability Enhanced Features

MRS optimizes and improves reliability and performance of main service components based on Apache Hadoop open-source software.

System reliability

- High availability (HA) for management nodes of all components

  Data and compute nodes of the Hadoop open-source version are designed based on the distributed system. Therefore, the whole system is not affected by single point of failures (SPOFs) of data and compute nodes. However,

management nodes operate in centralized mode. SPOFs of management nodes affect the whole system reliability.

Huawei MRS provides the dual-node mechanism for management nodes, such as OMS server, HDFS, NameNode, Hive Server, HBase HMaster, YARN Resources Manager, Kerberos Server, and Ldap Server of all service components. The management nodes work in active/standby or load-sharing mode, preventing impact of SPOFs on system reliability.

- Reliability guarantee in case of exceptions

  By reliability analysis, the following measures for software and hardware exceptions are provided to improve the system reliability:

  - After power supply is restored, services are running properly regardless of a power failure of a single node or the whole cluster, ensuring data reliability in case of unexpected power failures. Key data will not be lost unless the hard disk is damaged.
  - Health status check and fault handling of the hard disk do not affect services.
  - The file system faults can be automatically handled, and affected services can be automatically restored.
  - The process and node faults can be automatically handled, and affected services can be automatically restored.
  - The network faults can be automatically handled, and affected services can be automatically restored.

- Data backup and restoration

  MRS provides full backup, incremental backup, and restoration functions based on service requirements, preventing the impact of data loss and damage on services and ensuring fast system restoration in case of exceptions.

  - Automatic backup

    MRS provides automatic backup for data on Manager. Based on the customized backup policy, data on HBase, OMSServer, LDAP server, and DBService and ESN codes can be automatically backed up.

  - Manual backup

    You can also manually back up data on Manager before capacity expansion, and upgrade to recover the system functions upon faults.

    To improve the system reliability, data on OMS and HBase will be backed up to a third-party server manually.

Node reliability

- OS health status monitoring

  MRS provides the following monitoring measures for the OS:

  - Adjusting OS kernel parameters to restart the OS and restore services when a critical fault, for example, memory exhaust, invalid address accessing, kernel dead lock, or invalid dispatcher occurs in the OS
  - Periodically collecting OS running status data, including the processor status, memory status, hard disk status, and network status

- Process health status monitoring

NodeAgent is deployed on all nodes of MRS to monitor service instance status and health status of service instance processes.

- Automatic processing of hard disk faults

  MRS is enhanced based on the community version. It can monitor the status of hardware and file systems on all nodes. If a partition is faulty, the corresponding partition will be separated from the storage pool. If the whole hard disk is faulty and replaced, the new hard disk will be added to the storage pool. In this case, maintenance operations are simplified. Replacement of faulty hard disks can be complete online. In addition, users can set hot backup disks to reduce the faulty disk restoration time and improve the system reliability.

- RAID group configuration for nodes

  It is recommended that hard disk resources of nodes be planned based on service requirements to improve the MRS is capability against hard disk faults.

  – It is recommended that the OSs of nodes be installed on RAID 1 formed by two hard disks to ensure system disk reliability.
  – If allowed, RAID 1 is recommended for hard disks (HDFS NameNode, database, and ZooKeeper) used for key processes of management nodes to ensure metadata reliability.
  – Configure no RAID groups for data disks (HDFS DataNode, Kafka, Redis, SolrServerAdmin, and SolrServerN). If RAID groups are required (for disk identification), you can configure RAID 0 groups (only one disk in each RAID group).

Data reliability

MRS monitors hardware (especially hard disks), OS, and processes of nodes to discover exceptions in time. In this case, the fault detection and restoration time is reduced, and the data persistence rate of the whole system is improved.

6.10 Transparent Encryption

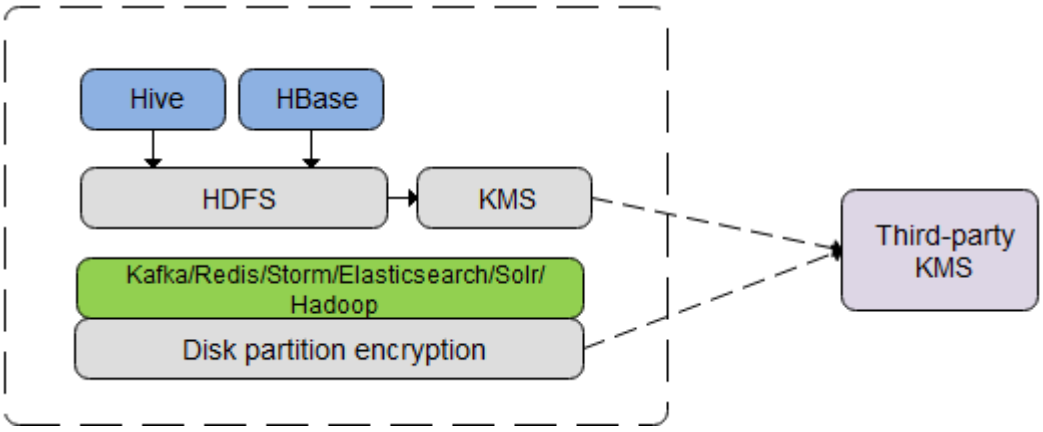
Overview

In traditional big data clusters, user data is stored in plaintext in the HDFS. Cluster maintenance personnel or malicious attackers can bypass the HDFS permission control mechanism or steal disks to directly access user data.

MRS introduces and enhances the Hadoop Key Management Service (KMS). By interconnecting with the third-party KMS or Huawei Cloud Stack KMS, MRS can implement transparent data encryption and ensure user data security.

Interconnecting with a third-party KMS:

Figure 6-2 Storage encryption of data connected to a third-party KMS

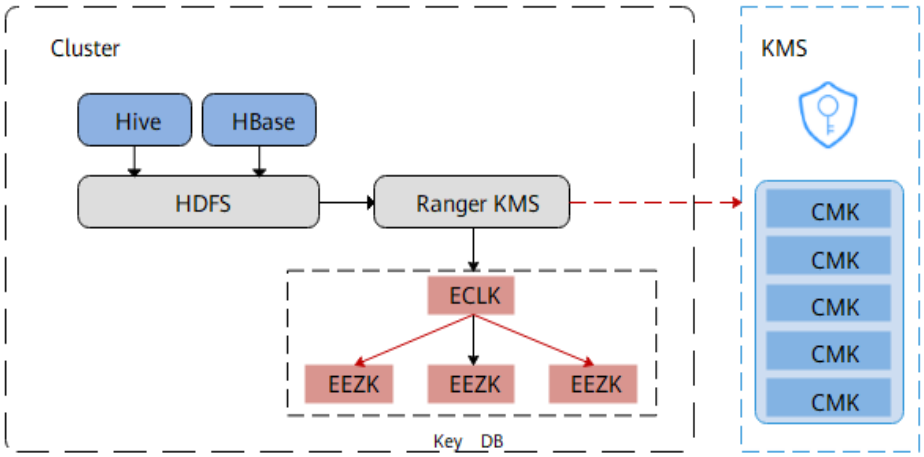


- HDFS supports transparent encryption. Upper-layer components such as Hive and HBase that store data in HDFS are encrypted using HDFS. The encryption key is obtained from the third-party KMS through Hadoop KMS.
- For components such as Kafka and Redis, that store service data on local disks permanently, the LUKS partition encryption mechanism is used to protect user data security.

Interconnecting with Huawei Cloud Stack KMS

HDFS transparent encryption also supports interconnection with Huawei Cloud Stack KMS, as shown in **Figure 6-3**. Encryption keys are managed by Ranger KMS and can be interconnected with Huawei Cloud Stack KMS through Ranger KMS.

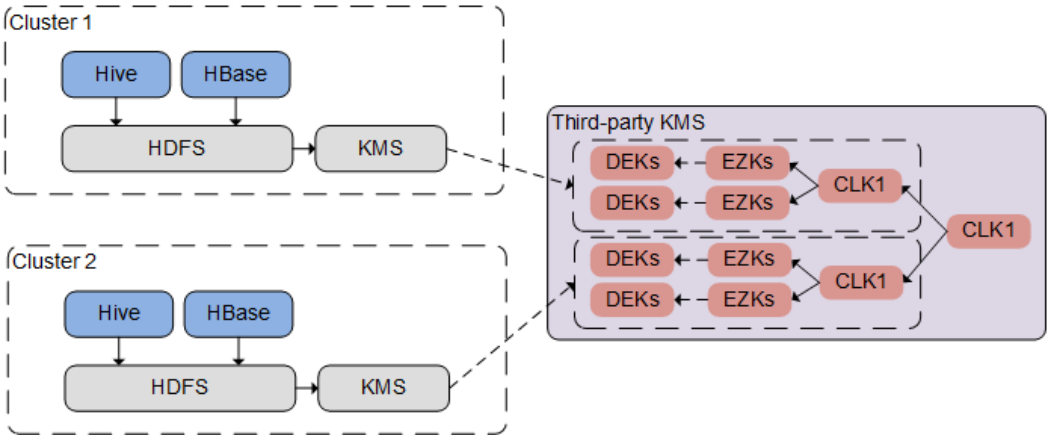
Figure 6-3 HDFS data storage encryption interconnecting with Huawei Cloud Stack KMS



HDFS Transparent Encryption

Figure 6-4 shows the principle of HDFS transparent encryption interconnecting with a third-party KMS.

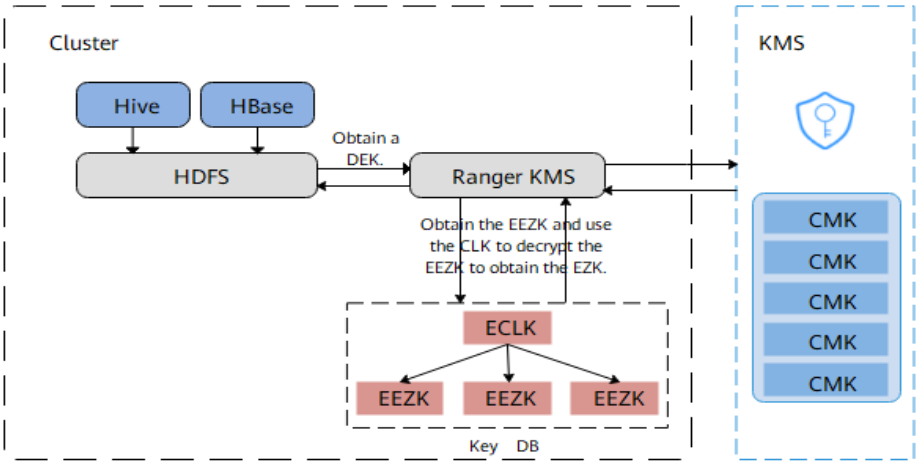
Figure 6-4 Transparent encryption of HDFS connected to a third-party KMS



- HDFS transparent encryption supports AES and SM4/CTR/NOPADDING encryption algorithms. Hive and HBase use HDFS transparent encryption for data encryption protection. The SM4 encryption algorithm is provided by the A-LAB based on OpenSSL.
- The key used for encryption is obtained from the KMS service in the cluster. The KMS service can connect to third-party KMS based on Hadoop KMS REST API.
- One KMS service is deployed in one FusionInsight Manager, and public and private key authentication is used by the KMS service to the third-party KMS. Each KMS service has a CLK in the third-party KMS.
- Multiple EZKs can be applied for in the CLK, which correspond to the encryption area in the HDFS and are used to encrypt the data encryption key. The EZK is stored in the third-party KMS persistently.
- The DEK is generated by the third-party KMS. It is encrypted using EZK and stored in the NameNode permanently. It is also decrypted using EZK.
- The CLK and EZK keys can be rotated. As the root key of each cluster, the CLK is unaware of the cluster. The rotation is controlled and managed by the third-party KMS. The EZK can be managed by the FusionInsight KMS, which can also control and manage FusionInsight KMS. In addition, the third-party KMS administrator has permissions of KMS key management and EZK rotation.

Figure 6-5 shows the principle of HDFS transparent encryption interconnecting Huawei Cloud Stack KMS.

Figure 6-5 HDFS transparent encryption interconnecting with Huawei Cloud Stack KMS



- HDFS transparent encryption supports AES and CTR/NOPADDING encryption algorithms. Hive and HBase use HDFS transparent encryption for data encryption protection.
- The key used for encryption is obtained from Ranger KMS in the cluster. Ranger KMS can connect to the Huawei Cloud Stack KMS service.
- Each cluster has an independent CLK. The CLK is obtained from Huawei Cloud Stack KMS, encrypted using customer master key, and stored in the key database of Ranger KMS. Multiple EZKs can be applied for in the CLK, which correspond to the encryption area in the HDFS and are used to encrypt the data encryption key. The EZK is stored in the key database of Ranger KMS persistently.
- DEKs are generated by Ranger KMS, encrypted using EZK, and then stored in the NameNode permanently. DEKs are decrypted using EZKs when needed.

LUKS Partition Encryption

For components such as Kafka and Redis, that store service data on local disks permanently, FusionInsight clusters support LUKS partition encryption for protecting sensitive information.

FusionInsight script tool uses the LUKS partition encryption solution. This solution generates an access key on each node of a cluster or obtains the access key from the third-party KMS when encrypting partitions. The access key is used to encrypt data keys to improve data key security. After the disk partitions are encrypted in the scenario when the OS is restarted or the disk is changed, the system automatically obtains the key and mounts or creates the encrypted partition.

6.11 SQL Inspector

SQL engines in the big data field are emerging one after another. However, in addition to a wide range of big data solutions, there are problems. For example, the quality of SQL statements is uneven, problems are difficult to locate, and large queries are resource-intensive.

Low-quality SQL statements pose unexpected impacts on data analysis platforms, degrading system performance and platform stability.

Function Description

MRS allows you to configure inspection rules for mainstream SQL engines (Hive, Spark, HetuEngine, and ClickHouse). MRS can identify typical large SQL queries and low-quality SQL statements and intercepts them before execution or blocks them during execution. Users do not need to change the way they submit SQL statements or the SQL syntax they are familiar with. Service modifications are not required and inspection is easy to implement.

- You can configure SQL inspection rules on the UI that also allows you to query and modify the rules.
- During query response and execution, each SQL engine proactively inspects SQL statements based on the rules.
- You can select to display a hint, intercept, or block a SQL statement. The system logs SQL inspection events in real time for audit. O&M engineers can analyze the logs, evaluate SQL statement quality on the live network, detect target statements, and take effective measures.

SQL inspection rules are classified into the following types:

- Static interception: The system displays hints or intercepts SQL statements based on SQL syntax.
- Dynamic interception: The system displays hints or intercepts SQL statements based on the rules regarding data table statistics and metadata information.
- Runtime blocking: The system blocks SQL statements based on system states (such as CPU, memory, and I/O) during the runtime of the SQL statements.

SQL statements that trigger astatic or dynamic interception rules are intercepted. Alternatively, the system gives hints for you to handle the statements properly. SQL statements that trigger runtime blocking rules are blocked.

Rules and Restrictions

- A SQL inspection rule can be associated with multiple SQL engines, and different threshold parameters can be configured for each service.
- A SQL inspection rule can be associated with multiple tenants. A rule takes effect only for associated tenants.

6.12 Unified Metadata Management

LakeFormation service is an enterprise-level one-stop data lake construction service. It provides visualized GUIs and APIs for unified management of data lake metadata and is compatible with Hive metadata models and Ranger permission models, enabling users to easily and efficiently build data lakes and operation services and accelerating the release of service data value.

LakeFormation uses underlying resources to implement cross-AZ deployment, high reliability, auto scaling, unified metadata management, association between metadata and file directories, and interconnection with multiple compute engines.

LakeFormation can interconnect with MapReduce Service (MRS) to enable multiple services and clusters to use unified metadata, maximizing data sharing, avoiding unnecessary data copies, and maximizing the value of service data.

Table 6-4 LakeFormation Function Overview

Operation	Function
Instance	LakeFormation provides different types of instances to meet customers' requirements on performance and costs in different scenarios.
Instance management	LakeFormation provides basic functions such as instance creation, overview, and deletion, helping you easily manage instances and accelerate the planning and deployment of services carried by the data lake.
Metadata management	LakeFormation provides life cycle management functions, such as creating, modifying, deleting, and viewing catalogs, databases, and tables of data lake metadata. It helps you easily initialize and operate a data lake, centrally manage all metadata of LakeFormation instances, and accelerate the planning and deployment of services carried by the data lake.
Data Permissions Management	LakeFormation allows you to authorize, cancel, and view data resources such as catalogs, databases, and tables. Helps you implement convenient and unified data permission management for the data lake.
Access management	LakeFormation provides unified access management capabilities. You can create an access client to establish a network connection stream for a specified client environment. In addition, you can view information such as the access IP address and access domain name in the client details for other cloud services to access LakeFormation instances.

LakeFormation Advantages

- Being compatible with the Hive metadata model, the SDK client supports easy and fast interconnection between compute engines and LakeFormation.
- The API for querying permissions is compatible with the Ranger permission model.

7 List of MRS Component Versions

Software List

Table 7-1 lists the versions of open-source components used by MRS.

Table 7-1 Software list

Component	MRS 3.3.1-LTS	MRS 3.3.0-LTS	MRS 3.2.1-LTS	MRS 3.2.0-LTS	MRS 3.1.3-LTS	MRS 3.1.2-LTS	MRS 3.1.1-LTS	MRS 3.1.0-LTS	MRS 3.0.2-LTS
CarbonData	2.2.0	2.2.0	2.2.0	2.2.0	2.2.0	2.2.0	2.0.1	2.0.1	2.0.1
CDL	1.1.0	1.1.0	1.1.0	1.1.0	1.1.0	1.1.0	1.0.0	-	-
ClickHouse	23.3.2.37	23.3.2.37	22.3.2.2	22.3.2.2	21.8.8.29	21.3.4.25	21.3.4.25	21.3.4.25	-
Containers	2.1.0	2.1.0	2.1.0	-	-	-	-	-	-
DBService	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0
Doris	2.0.5	1.2.3	-	-	-	-	-	-	-
Elasticsearch	7.10.2	7.10.2	7.10.2	7.10.2	7.10.2	7.10.2	7.10.2	7.10.2	7.6.0
Flink	1.17.1	1.15.0	1.15.0	1.15.0	1.12.2	1.12.2	1.12.2	1.12.0	1.10.0
Flume	1.11.0	1.11.0	1.9.0	1.9.0	1.9.0	1.9.0	1.9.0	1.9.0	1.9.0
FTP-Server	1.1.3	1.1.3	1.1.1	1.1.1	1.1.1	1.1.1	1.1.1	1.1.1	1.1.1

Component	MRS 3.3.1-LTS	MRS 3.3.0-LTS	MRS 3.2.1-LTS	MRS 3.2.0-LTS	MRS 3.1.3-LTS	MRS 3.1.2-LTS	MRS 3.1.1-LTS	MRS 3.1.0-LTS	MRS 3.0.2-LTS
Guardian	0.1.0	0.1.0	0.1.0	-	-	-	-	-	-
GraphBase	8.3.1	8.3.0	8.2.1.1	8.2.0	8.1.3	8.1.2	8.1.1	8.1.0.1	8.0.2.1
Hadoop (including HDFS, MapReduce, and Yarn)	3.3.1	3.3.1	3.3.1	3.3.1	3.1.1	3.1.1	3.1.1	3.1.1	3.1.1
HBase	2.4.14	2.4.14	2.4.14	2.2.3	2.2.3	2.2.3	2.2.3	2.2.3	2.2.3
HetuEngine	2.1.0	2.0.0	2.0.0	1.2.0	1.2.0	1.2.0	1.2.0	1.0.0	1.0.0
Hive	3.1.0	3.1.0	3.1.0	3.1.0	3.1.0	3.1.0	3.1.0	3.1.0	3.1.0
Hudi	0.11.0	0.11.0	0.11.0	0.11.0	0.9.0	0.9.0	0.8.0	-	-
Hue	4.7.0	4.7.0	4.7.0	4.7.0	4.7.0	4.7.0	4.7.0	4.7.0	4.7.0
IoTDB	1.1.0	1.1.0	0.14.0	0.14.0	0.12.0	0.12.0	0.12.0	-	-
JobGateway	1.0.0	1.0.0	1.0.0	-	-	-	-	-	-
Kafka	2.12-3.6.1	2.12-2.8.1	2.12-2.8.1	2.11-2.4.0	2.11-2.4.0	2.11-2.4.0	2.11-2.4.0	2.11-2.4.0	2.11-2.4.0
KrbServer	1.20	1.20	1.19	1.18	1.18	1.18	1.17	1.17	1.17
KMS	3.3.1	3.3.1	3.3.1	3.3.1	3.1.1	3.1.1	3.1.1	3.1.1	3.1.1
LakeSearch	1.0.0	-	-	-	-	-	-	-	-
Loader(Sqoop)	1.99.3	1.99.3	1.99.3	1.99.3	1.99.3	1.99.3	1.99.3	1.99.3	1.99.3
LdapServer	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0	2.7.0

Component	MRS 3.3.1-LTS	MRS 3.3.0-LTS	MRS 3.2.1-LTS	MRS 3.2.0-LTS	MRS 3.1.3-LTS	MRS 3.1.2-LTS	MRS 3.1.1-LTS	MRS 3.1.0-LTS	MRS 3.0.2-LTS
MemArtsCC	3.22	-	-	-	-	-	-	-	-
MOTService	2.7.0	2.7.0	2.7.0	-	-	-	-	-	-
Oozie	5.1.0	5.1.0	5.1.0	5.1.0	5.1.0	5.1.0	5.1.0	5.1.0	5.1.0
Phoenix	5.1.2	5.1.2	5.1.2	5.0.0-HBase-2.0	5.0.0-HBase-2.0	5.0.0-HBase-2.0	5.0.0-HBase-2.0	5.0.0-HBase-2.0	5.0.0-HBase-2.0
Ranger	2.3.0	2.3.0	2.3.0	2.0.0	2.0.0	2.0.0	2.0.0	2.0.0	2.0.0
Redis	6.2.7	6.2.7	6.2.7	6.0.12	6.0.12	6.0.12	6.0.12	5.0.4	5.0.4
RTDService	3.2.0	3.2.0	3.2.0	-	-	-	-	-	-
SmallFS	-	-	-	1.0.0	1.0.0	1.0.0	1.0.0	1.0.0	1.0.0
Solr	8.11.2	8.11.2	8.11.2	8.4.0	8.4.0	8.4.0	8.4.0	8.4.0	8.4.0
Spark	3.3.1	3.3.1	3.3.1	-	-	-	-	-	-
Spark2x	-	-	-	3.1.1	3.1.1	3.1.1	3.1.1	2.4.5	2.4.5
Storm	-	-	-	-	-	-	1.2.1	1.2.1	1.2.1
Tez	0.10.2	0.10.2	0.10.2	0.9.2	0.9.2	0.9.2	0.9.2	0.9.2	0.9.2
ZooKeeper	3.8.1	3.8.1	3.6.3	3.6.3	3.6.3	3.6.3	3.6.3	3.5.6	3.5.6

8 External APIs Provided by MRS Components

Table 8-1 describes external APIs provided by MRS components.

Table 8-1 External APIs provided by the components

Name	API Supported in Security Mode	API Supported in Normal Mode
CDL	REST	REST
ClickHouse	CLI, JDBC, and REST	CLI, JDBC, and REST
Containers	Java, REST API, and Socket	Java, REST API, and Socket
Doris	CLI, JDBC, and REST	CLI, JDBC, and REST
Elasticsearch	Java and REST	Java and REST
Flink	CLI, Java, Scala, and REST	CLI, Java, Scala, and REST
Flume	Java	Java
GraphBase	CLI, Java, and REST	CLI, Java, and REST
HBase	CLI, Java, Sqlline, JDBC, and REST	CLI, Java, Sqlline, JDBC, and REST
HDFS	CLI, Java, C, and REST	CLI, Java, C, and REST
HetuEngine	CLI, JDBC, and REST	CLI, JDBC, and REST
Hive	CLI, JDBC, Python, and REST (only for WebHCat)	CLI, JDBC, Python, and REST (only for WebHCat)
IoTDB	CLI, Java, and JDBC	CLI, Java, and JDBC
JobGateway	Java and REST	Java and REST

Name	API Supported in Security Mode	API Supported in Normal Mode
Kafka	CLI and Java	CLI, Java, and Scala
LakeSearch	REST	REST
Loader	CLI and REST	CLI and REST
Manager	CLI, SNMP, Syslog, and REST	CLI, SNMP, Syslog, and REST
MapReduce	Java and REST	Java and REST
MOTService	CLI and JDBC	CLI and JDBC
Oozie	CLI, Java, and REST	CLI, Java, and REST
Ranger	Java and REST	Java and REST
Redis	CLI and Java	CLI and Java
RTDService	HTTP and REST API	HTTP and REST API
Solr	CLI, Java, and REST	CLI, Java, and REST
Spark	CLI, Java, Scala, Python, JDBC, and REST	CLI, Java, Scala, Python, JDBC, and REST
Tez	REST	REST
Yarn	CLI, Java, and REST	CLI, Java, and REST

9 Related Services

This section describes the relationship between an MRS ECS/BMS cluster and other services. After an MRS physical machine cluster is installed, you can configure interconnection between FusionInsight Manager and MRS Console so that cluster information can be reported to MRS Console for unified O&M management.

- Virtual Private Cloud (VPC)

  The MRS ECS/BMS cluster is created in the subnets of a VPC. VPCs provide a secure, isolated, and logical network environment for your MRS clusters.

- Object Storage Service (OBS)

  When MRS is interconnected with OBS 3.0 during MRS installation, the components in the MRS ECS/BMS cluster can store data in OBS to implement storage and compute decoupling.

  Currently, Flink, Hadoop (HDFS/Yarn/MapReduce), HBase, HetuEngine, Hive, Loader, Spark, and Hudi in MRS clusters can connect to OBS 3.0 to help implement storage-compute decoupling. MRS uses the Guardian component to connect to the OBS parallel file system and provide other components with the temporary authentication credentials and fine-grained permission control capabilities for accessing OBS.

- Elastic Cloud Server (ECS)

  Each node in an MRS ECS cluster is an ECS.

- Bare Metal Server (BMS)

  Each node in an MRS BMS cluster is a BMS.

- Simple Message Notification (SMN)

  MRS uses SMN to offer a publish/subscribe model to achieve one-to-multiple alarm message subscriptions and notifications in a variety of message types (SMSs and emails).

10 Permissions Required for Using MRS

List of Permissions

The system provides two types of permissions by default: user management and resource management. User management permissions can manage users, user groups, and user group permissions. Resource management refers to the control of operations that can be performed by users on cloud service resources.

Table 10-1 lists the MRS permissions.

Table 10-1 List of permissions

Permission	Description	How to Assign Permissions
MRS operation permissions	Users that have the full operation permissions on MRS resources	There are two setting methods: • Set the MRS FullAccess, VPC Administrator, EVS Administrator, Server Administrator, and SMN Administrator for the user group where a user belongs. • Assign the MRS Administrator, Server Administrator, Tenant Guest roles to the user group where a user belongs.
Permission to use MRS	Users with this permission can query clusters and configure jobs, files, and alarms.	Set the MRS CommonOperations, VPC Administrator, EVS Administrator, Server Administrator, and SMN Administrator for the user group where a user belongs.

Permission	Description	How to Assign Permissions
Permission to query MRS resources	Users with this permission have the MRS read-only permission, including querying clusters.	Set the MRS ReadOnlyAccess permission policy for the user group where a user belongs. To use the VPC or EVS function, configure the VPC Administrator or EVS Administrator, respectively.

11 MRS Restrictions

Before using MRS, ensure that you have read and understood the following restrictions.

- MRS clusters must be created in VPC subnets.
- When you create an MRS cluster, you can select **Auto Create** from the drop-down list of **Security Group** to create a security group or select an existing security group. After the MRS cluster is created, do not delete or modify the used security group. Otherwise, a cluster exception may occur.
- To prevent illegal access, only assign access permission for security groups used by MRS where necessary.
- Do not perform the following operations because they will cause cluster exceptions:
  - Shutting down, restarting, or deleting MRS cluster nodes displayed in ECS, changing or reinstalling their OS, or modifying their specifications.
  - Deleting the existing processes, applications, or files on cluster nodes.
  - Deleting MRS cluster nodes, which may cause cluster exception and result in your loss.
- Keep the initial password for logging in to the Master node properly because MRS will not save it. Use a complex password to avoid malicious attacks.
- If the cluster is abnormal, contact the technical support for troubleshooting.
- Plan disks of cluster nodes based on service requirements. If you want to store a large volume of service data, add EVS disks or storage space to prevent insufficient storage space from affecting node running.
- The cluster nodes store only users' service data. Non-service data can be stored in the OBS or other ECS nodes.
- The cluster nodes only run MRS cluster programs. Other client applications or user service programs are deployed on separate ECS nodes.

12 Common Specifications

Table 12-1 lists the common system specifications of each service in the MRS cluster.

Table 12-1 System specifications

Type	Indicator	Specifications	Description
Cluster	Maximum number of nodes	30,000	Physical machine cluster, universal x86 servers (not limited to Huawei servers) or Huawei TaiShan servers.
	Number of tenants	5,000	Maximum number of tenants
	Number of peer systems supporting multi-system mutual trust	500	Number of peer systems that support mutual trust configuration between Manager systems and other systems
HDFS	Number of NameServices	10	Physical machine cluster. Maximum number of NameNode pairs supported by the system.
	Maximum number of files for a NameService	150 million	-
	Maximum number of blocks on a DataNode	5 million	-
	Maximum number of blocks on a DataNode disk	500 thousands	-
	Maximum number of file directories in a directory (excluding recursion)	1 million	Configuration parameter: dfs.namenode.fs-limits.max-directory-items

Type	Indicator	Specifications	Description
	Maximum number of blocks in each file	1 million	Configuration parameter: dfs.namenode.fs-limits.max-blocks-per-file
	Maximum length of a file path	8,000	Configuration parameter: dfs.namenode.fs-limits.max-component-length
	Minimum block size	1 MB	Configuration parameter: dfs.namenode.fs-limits.min-block-size
	Minimum number of normal disks allowed by a DataNode	1	Configuration parameter: dfs.datanode.failed.volumes.tolerated
Yarn	Maximum memory allocated to a single NodeManager	Physical memory x 0.8	-
	Maximum virtual cores allocated to a single NodeManager	Logical CPU x 1.5 to 2	-
HBase	Number of HBase RegionServers	1,024	Number of RegionServer instances of a single HBase service
	Number of regions of a RegionServer instance	2,000	Maximum number of regions supported by a RegionServer instance
	Number of active regions supported by a single RegionServer	200	Maximum number of active regions supported by each RegionServer instance
Hive	Number of partitions supported by a single Hive table	1 million	Maximum number of partitions recommended for a single Hive table
	Maximum number of files in a single table	1 million	Maximum number of files that can be stored in HDFS for a single Hive table
	Maximum number of concurrent requests on a HiveServer	500	Maximum number of concurrent requests supported by a HiveServer instance
Kafka	Number of nodes in a Kafka cluster	256	Universal x86 servers (not limited to Huawei servers) or Huawei TaiShan servers

Type	Indicator	Specifications	Description
	Maximum length of topic names	200 bytes	Not greater than 200 bytes
Redis	Number of instances in a single Redis cluster	512	Number of Redis processes
	Number of Redis clusters	512	Maximum number of Redis clusters
Solr	Number of instances in a Solr cluster	500	Number of Solr processes
	Number of cores supported by a single SolrServer	200	-
	Number of records supported by a single core	1 to 400 million	-
	Maximum memory configuration of a single SolrServer	31 GB	-
	Optimal ratio of the memory and disk of a single SolrServer	1:20	-
Elastic search	Number of instances in a single Elasticsearch cluster	512	-
	Maximum number of shards supported by a single Elasticsearch cluster	70,000	A single Elasticsearch cluster supports a maximum of 2 PB data.
	Maximum number of indexes supported by a single Elasticsearch cluster	5,000	-
	Maximum memory configuration of a single Elasticsearch instance	31 GB	-
	Number of records supported by a single shard	1 to 400 million	-
	Amount of data that can be stored on a single shard	30 GB	Recommended storage: 20 GB

Type	Indicator	Specifications	Description
	Maximum number of shards in a single EsNode instance	500	200 to 300 shards are recommended.
	Maximum storage capacity of a single EsNode instance	15 TB	5 TB is recommended.
	Optimal ratio of the memory and disk of a single EsNode instance	1:50	Optimal hot data ratio: 1:50 Optimal cold data ratio: 1:100
ZooKeeper	Number of instances in a ZooKeeper cluster	9	Maximum number of instances in a ZooKeeper cluster • Number of physical machine clusters: 9 • Number of ECS/BMS clusters: 5
	Maximum number of connections supported an IP address for each ZooKeeper instance	2,000	-
	Maximum number of connections supported a ZooKeeper instance	20,000	-
	Maximum number of ZNodes in the case of default parameter configurations	2000000	If there are too many ZNodes, the service will be unstable and the read and write performance of components deteriorates. In regular service scenarios, it is recommended that there be no more than 2 million ZNodes. If you deployed only ClickHouse and its dependent components in the cluster, there can be no more than 6 million ZNodes.
	Size of a single ZNode	4 MB	-
Flume	Maximum number of Flume instances in a cluster	128	Maximum number of Flume instances
GraphBase	Maximum number of connections that can be enabled in a GraphServer instance at the same time	4,096	Configuration parameter: MAX_CONNECTIONS_PERSERVER

Type	Indicator	Specifications	Description
HetuEngine	Number of HetuEngine compute instances in a cluster	1–200	-
	Minimum memory allocated for JVMs of coordinators or workers in a compute instance	1 GB	-
	Number of Coordinators in a compute instance	1–3	-
	Number of workers in a compute instance	1–256	-
	Number of interconnected data sources on the HSConsole page	1–100	-
ClickHouse	Number of ClickHouse instances in a single cluster	256	Maximum number of instances supported by a single ClickHouse cluster
	Maximum number of tables supported by each ClickHouseServer instance	5,000	-
	Maximum number of partitions supported by a table in each ClickHouseServer instance	10,000	-
MOTService	Number of MOTService instances (single service in a single cluster)	2	Common x86/Arm servers (not limited to Huawei servers)
	MOTService database sharding (based on the number of services)	4	A maximum of four shards is recommended for a single tenant.
RTDService	Number of RTDService instances (in a single cluster)	2	Active/Standby deployment
Containers	Number of Containers instances	500	A maximum of five Containers instances can be deployed on a single node.
	Number of BLUs (for a single Containers instance)	15	A maximum of three BLUs can be deployed in a single WebContainer instance.

Type	Indicator	Specifications	Description
Doris	Number of Doris BE instances (in a single cluster)	200	Maximum number of BE instances supported by a single Doris cluster
	Number of Doris FE instances (in a single cluster)	9	Maximum number of FE instances supported by a single Doris cluster
	Maximum number of tables supported by Doris	5000	-
	Maximum number of partitions supported by a single Doris table	10000	-
LakeSearch	Number of SearchFactory instances	2-16	The number of SearchFactory instances in a single cluster must be no less than 2.
	Number of SearchServer instances	2-16	The number of SearchServer instances in a single cluster must be no less than 2.
MemArtsCC	Buffer memory for Worker I/O read	10164 x I/O size	I/O size specified by **spark.hadoop.fs.obs.memartscc.buffer.size**.
	Worker buffer memory for request prefetch	32 x 8MB	-
	Minimum memory of a single Worker instance	4 GB	-
	Number of read file handles hit in Worker cache	10164	-
	Number of file handles prefetched from Worker cache	3000	-

