MapReduce Service

Product Introduction

Issue	01
Date	2024-04-10

HUAWEI

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.

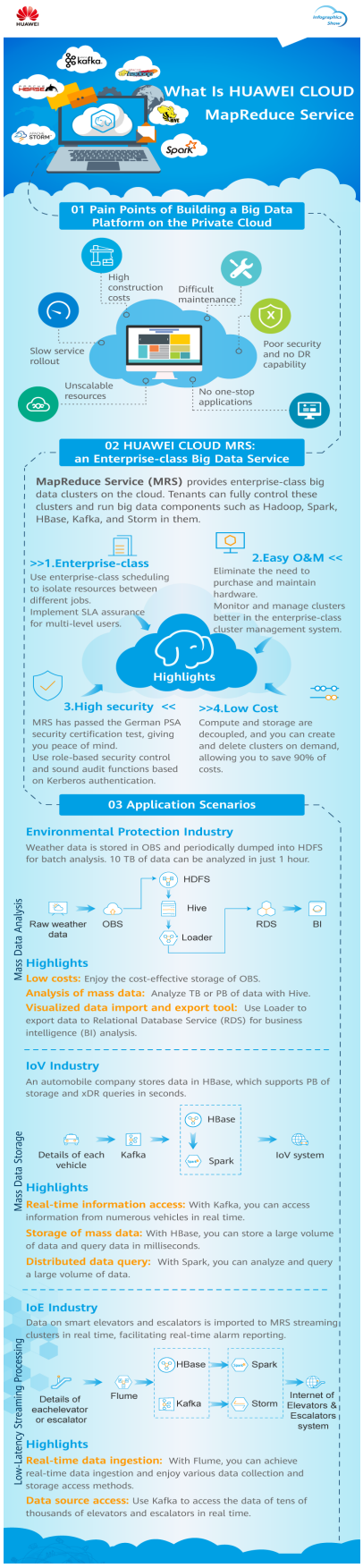
Notice

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.

Contents

1 Infographics



2 What Is MRS?

Big data presents both exciting opportunities and a huge challenge. As the data volume and types increase rapidly, conventional data processing technologies, such as standalone storage systems and relational databases, are struggling to keep up. Rising to this challenge, the Apache Software Foundation (ASF) launched an open source project called Hadoop. Hadoop is an open source distributed computing platform that can fully utilize the computing and storage capabilities of large compute clusters to process massive amounts of data. Hadoop is a powerful framework, but it is not easy to deploy and operationalize — If enterprises try to deploy Hadoop systems all by themselves, they may encounter problems such as high costs, long rollout, difficult maintenance, and inflexible use.

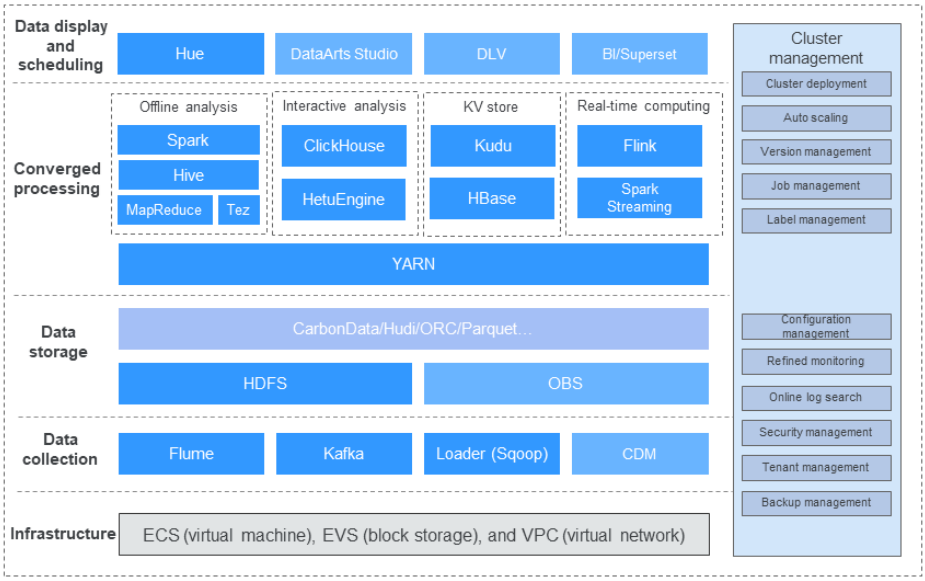
The MapReduce Service (MRS) offers a one-stop service that helps you quickly deploy and manage Hadoop systems on Huawei Cloud with ease. With MRS, you can create an enterprise-class Hadoop cluster with just a few clicks of your mouse. Tenants have total control over their Hadoop clusters and can effortlessly run big data components such as Storm, Hadoop, Spark, HBase, and Kafka. MRS supports a full range of open source APIs, and leveraging Huawei Cloud's deep expertise in compute, storage, and big data, it offers customers a full-stack big data platform featuring high performance, high cost-effectiveness, flexibility, and ease-of-use. Furthermore, the platform can be easily customized to meet new requirements and help enterprises quickly build a massive data processing system and discover new value and business opportunities by analyzing and mining massive amounts of data in real time or in non-real time.

Product Architecture

List of MRS Component Versions lists the MRS component versions.

Figure 2-1 shows the MRS logical architecture.

Figure 2-1 MRS architecture



MRS includes the infrastructure and an end-to-end big data processing pipeline.

- Infrastructure

  MRS big data clusters fully utilize the high scalability, reliability, and security features of the virtualization layer powered by the cloud platform.

  - Virtual Private Cloud (VPC) provides virtual private networks for each tenant on the cloud.
  - Elastic Volume Service (EVS) provides reliable and high-performance storage.
  - Elastic Cloud Server (ECS) provides VMs that are easily scalable. It works with VPCs, security groups, and the EVS multi-replica mechanism to build an efficient, reliable, and secure computing environment.

- Data collection

  The data collection layer provides the ability to efficiently ingest data from various data sources. It consists of Flume (data ingestion), Loader (relational data loading), and Kafka (highly reliable message queue). Alternatively, you can use Huawei Cloud's Cloud Data Migration (CDM) service to ingest external data to MRS clusters.

- Data storage

  MRS clusters can store both structured and unstructured data. They support multiple efficient data formats to meet the requirements of different computing engines, including:

  - HDFS, which is a general-purpose distributed file system for big data platforms.
  - Huawei Cloud OBS is an object storage service that features high availability and low cost.
  - HBase, which supports data storage with indexes, accelerating index-based queries.

- Converged data processing

  - MRS supports multiple mainstream compute engines, including MapReduce (batch processing), Tez (DAG model), Spark (in-memory computing), Spark Streaming (micro-batch stream computing), Storm (stream computing), and Flink (stream computing). They convert data structures and logic into data models that meet the needs of a variety of big data applications.
  - Based on preset data models and easy-to-use SQL data analysis, users can choose Hive (data warehouse), SparkSQL, and Presto (interactive query engine) to run different types of analytical tasks.
- Data display and scheduling

  Data analysis results are displayed intuitively. MRS also integrates with DataArts Studio to provide a one-stop, collaborative big data development platform, helping you easily run a range of different tasks, such as data modeling, data integration, script development, job scheduling, and O&M monitoring, making big data more accessible than ever before.
- Cluster management

  All components of the Hadoop-based big data ecosystem are deployed in distributed mode, and their deployment, management, and O&M are complex.

  MRS provides a unified O&M and management platform for cluster management, supporting one-click cluster deployment, multi-version selection, as well as manual scaling and auto scaling of clusters with zero service interruption. In addition, MRS provides job management, resource tag management, and O&M covering all of the Hadoop components. One-stop O&M capabilities include monitoring, alarm reporting, parameter configuration, and patch upgrade.

Product Advantages

MRS has a strong Hadoop kernel team and is built on top of Huawei's enterprise-class FusionInsight big data platform. MRS can guarantee multi-level Service Level Agreements (SLAs).

MRS has the following advantages:

- High performance

  MRS supports Huawei's own CarbonData storage solution. CarbonData allows a single copy of data to be used for multiple tasks. It supports features such as multi-level indexing, dictionary encoding, pre-aggregation, dynamic partitioning, and quasi-real-time data query. These features improve I/O scanning and computing performance, allowing tens of billions of data records to be analyzed in seconds. In addition, MRS supports the Superior Scheduler also developed by Huawei, which outperforms open-source schedulers in every way and enables efficient scheduling in super large clusters (up to 10,000 nodes).
- Cost-effectiveness

  MRS supports a heterogeneous compute and storage infrastructure with decoupled storage and compute, offering a cost-effective mass storage solution. MRS supports fast auto scaling to accommodate changing demand, maximizing resource utilization for customers. MRS clusters can be quickly created and scaled out as you needed, and can be deleted or scaled when you no longer need them.

- High security

  MRS provides enterprise-class multi-tenant permissions management and security management, with support for table-based and column-based access control and data encryption.

- Easy O&M

  MRS provides an efficient big data cluster management platform that supports one-click rolling patch updates, which ensure the continuity of your services.

- High reliability

  Tested and proven in numerous projects, the long-term reliability and stability of MRS in large-scale deployments can meet enterprise-class standards for production systems. In addition, MRS supports automatic data backup across AZs and regions, as well as automatic anti-affinity, allowing mission-critical VMs to be distributed on different physical machines.

Using MRS for the First Time

If you are a first-time user, you may get started with the following:

- **Basic concepts**

  See **Components** and **Functions** to learn the basic information about MRS, including all its components and their enhancements over their open-source counterparts, as well as the unique features of MRS.

- **Getting started**

  To learn how to use MRS, see **MapReduce Service Getting Started**. "Getting Started" provides detailed operation guides with real-world examples. You can create and use MRS clusters by following these guides.

- **Other functions and operation guides**

  If you are an MRS cluster user or O&M engineer, you can perform operations such as cluster life cycle management, scaling, and job management by referring to **MapReduce Service User Guide**. To learn how to use each component, see **MapReduce Service Component Operation Guide**.

  If you are a developer, you can refer to the operation guides and examples in **MapReduce Service Development Guide** to develop, run, and commission your own applications. For details about how to call the APIs of MRS. For details, see **MapReduce Service API Reference**.

3 Advantages of MRS Compared with Self-Built Hadoop

MRS provides enterprise-level big data clusters on the cloud. Tenants can fully control the clusters and run big data components such as Hadoop, Spark, HBase, Kafka, and Storm with ease. MRS frees you from hardware purchase and maintenance. MRS is built based on enterprise-class big data platform Huawei FusionInsight and has been deployed on tens of thousands of nodes in the industry, providing multi-level SLA assurance with professional Hadoop kernel service support. Compared with self-built Hadoop clusters, MRS has the following advantages:

1. **MRS supports one-click cluster creation, deletion, and scaling. You can use an elastic IP address (EIP) to access MRS Manager, making big data clusters easier to use.**
   - Self-built big data clusters pose problems such as high costs, long periods, difficult and inflexible O&M. To solve these problems, MRS provides one-click cluster creation, deletion, scale-out, and scale-in, allowing you to customize the cluster type, component range, number of nodes of each type, VM specifications, availability zones (AZs), VPC network, and authentication information. MRS can automatically create a cluster that meets the configuration requirements. In addition, you can quickly create multi-application clusters, for example, Hadoop analysis cluster, HBase cluster, and Kafka cluster. MRS supports heterogeneous cluster deployment. That is, VMs of different specifications can be combined in a cluster based on CPU types, disk capacities, disk types, and memory sizes.
   - MRS provides an EIP-based secure channel for you to easily access the web UIs of components. This is more convenient than binding an EIP by yourself, and you can access the web UIs with a few clicks, avoiding the steps for logging in to a VPC, adding security group rules, and obtaining a public IP address.
   - MRS provides custom bootstrap actions to flexibly configure your dedicated clusters. Third-party software that is not supported by MRS can be automatically installed, allowing you to perform custom operations such as modifying the cluster running environment.
   - MRS supports the WrapperFS feature, provides the OBS translation capability (that is, access to OBS through address mapping) and can

smoothly migrate data from HDFS to OBS. After migration, you can access the data stored in OBS from clients without modifying service code logic.

2. **MRS supports auto scaling, which is more cost-effective than the self-built Hadoop cluster.**

   MRS supports auto scaling to address peak and off-peak service loads. It applies for extra resources during peak hours and releases idle resources during off-peak hours, helping you save idle resources on the big data platform during off-peak hours, minimize costs, and focus on core services.

   In big data applications, especially in periodic data analysis and processing, cluster computing resources need to be dynamically adjusted based on service data changes to meet service requirements. The auto scaling function of MRS enables clusters to be elastically scaled out or in based on cluster loads. In addition, if the data volume changes regularly and you want to scale out or in a cluster before the data volume changes, you can use the MRS resource plan feature. MRS supports two types of auto scaling policies: auto scaling rules and resource plans

   - Auto scaling rules: You can increase or decrease Task nodes based on real-time cluster loads. Auto scaling will be triggered when the data volume changes but there may be some delay.
   - Resource plans: If the data volume changes periodically, you can create resource plans to resize the cluster before the data volume changes, thereby avoiding a delay in increasing or decreasing resources.

   Both auto scaling rules and resource plans can trigger auto scaling. You can configure both of them or configure one of them. Configuring both resource plans and auto scaling rules improves the cluster node scalability to cope with occasionally unexpected data volume peaks.

3. **MRS supports storage-compute decoupling, greatly improving the resource utilization of big data clusters.**

   In the traditional big data architecture where storage and compute resources are integrated, scaling-out is difficult and resources are not well-utilized. To solve these problems, MRS adopts a compute-storage separation architecture. Based on OBS, the storage achieves 99.999999999% reliability and unlimited capacity, supporting continuous growth of enterprise data. Computing resources can be elastically scaled in or out from 0 to *N* nodes. Hundreds of nodes can be quickly provisioned. With the new architecture, compute nodes can be elastically scaled. OBS-based cross-AZ data storage ensures higher reliability, frees you from worrying about emergencies such as earthquakes and fiber cuts. Storage and compute resources can be flexibly configured and elastically scaled as required. This makes resource allocation more accurate and reasonable, greatly improving the resource utilization of big data clusters and reducing the comprehensive analysis cost by 50%.

   In addition, the high performance compute-storage separation architecture breaks the limit of parallel computing of the integrated storage-compute architecture. It maximizes the high bandwidth and high concurrency of OBS, and optimizes the data access efficiency and in-depth parallel computing (such as metadata operation and write algorithm optimization) to improve higher performance.

4. **MRS supports self-developed CarbonData and Superior Scheduler, delivering better performance.**

- MRS supports self-developed CarbonData storage technology. CarbonData is a high-performance big data storage solution. It allows one data set to apply to multiple scenarios and supports features, such as multi-level indexing, dictionary encoding, pre-aggregation, dynamic partitioning, and quasi-real-time data query. This improves I/O scanning and computing performance and returns analysis results of tens of billions of data records in seconds.
- In addition, MRS supports self-developed Superior Scheduler, which enhances the scaling capability of a single cluster and is capable of scheduling over 10,000 nodes in a cluster. 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 by multiple users in a tenant, scheduling performance, system resource utilization, and cluster scalability, and is designed to replace open source schedulers.

5. **MRS optimizes software and hardware based on Kunpeng processors to fully release hardware computing power and achieve cost-effectiveness.**

   MRS supports self-developed Kunpeng servers whose multi-core and high-concurrency capabilities are fully utilized to provide full-stack self-optimized chips, and uses self-developed EulerOS, Huawei JDK, and data acceleration layer to ensure hardware performance, delivering high computing power for big data computing. With the similar performance, the cost of the end-to-end big data solution is reduced by 30%.

6. **MRS supports multiple isolation modes and multi-tenant permission management of enterprise-level big data, ensuring higher security.**

   - MRS supports resource deployment and isolation of physical resources in dedicated zones. You can flexibly combine computing and storage resources, such as dedicated computing resources + shared storage resources, shared computing resources + dedicated storage resources, and dedicated computing resources + dedicated storage resources. An MRS cluster supports multiple logical tenants. Permission isolation enables the computing, storage, and table resources of the cluster to be divided based on tenants.
   - With Kerberos authentication, MRS provides role-based access control (RBAC) and sound audit functions.
   - With Cloud Trace Service (CTS) being interconnected with MRS, you are provided with operation records of MRS resource operation requests and request results for querying, auditing, and backtracking. You can use CTS to audit and trace all cluster operations.
   - It is proved that with Host Security Service (HSS) interconnected with MRS, service security is enhanced without deteriorating functions and performance.
   - MRS supports unified user login based on web UI. Manager provides user authentication, which grants you permission to access a cluster.

- MRS supports data storage encryption, encrypted storage of all user accounts and passwords, encrypted transmission of data channels, and bidirectional certificate authentication for cross-trusted-zone data access of service modules.
- MRS big data clusters provide a complete multi-tenant solution for enterprise-level big data. Multi-tenant refers to a collection of multiple resources (each resource set is a tenant) in an MRS big data cluster. It can allocate and schedule resources, including computing and storage resources. Multi-tenant 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.
- MRS supports fine-grained permission management. With the fine-grained authorization capability provided by HUAWEI CLOUD IAM, MRS can specify the operations, resources, and request conditions of specific services. This mechanism allows for more flexible policy-based authorization, meeting requirements for secure access control. For example, you can grant MRS users only the permissions for performing specified operations on MRS clusters, such as creating a cluster and querying a cluster list rather than deleting a cluster. In addition, MRS supports fine-grained permission management of OBS for multiple tenants. Permissions to access OBS buckets and objects in the buckets are differentiated based on user roles, so that MRS users can each control a different directory in OBS buckets.
- MRS supports enterprise project management. The enterprise project is one way of managing cloud resources. Enterprise Management provides comprehensive management services for enterprise customers, such as cloud resources, personnel, permissions, and financial statuses. Common management consoles are oriented to the control and configuration of individual cloud products. The Enterprise Management console, in contrast, is more focused on resource management. It is designed to help enterprises manage cloud-based resources, personnel, permissions, and finances, in a hierarchical management manner, such as management of companies, departments, and projects. MRS allows users who have enabled Enterprise Project Management Service (EPS) to configure enterprise projects for a cluster during cluster creation and use EPS to manage MRS resources by group. This feature is applicable to scenarios where you need to manage multiple resources by group and perform operations such as permission control and project-based fee query on enterprise projects.

7. **MRS implements HA for all management nodes and supports comprehensive reliability mechanism, making the system more reliable.**

   Based on Apache Hadoop open-source software, MRS optimizes and improves the reliability of main service components.

   - HA for all management nodes

     In the Hadoop open-source version, data and compute nodes are managed in a distributed system, in which a single point of failure (SPOF) does not affect the operation of the entire system. However, a SPOF may occur on management nodes running in centralized mode, which becomes the weakness of the overall system reliability.

MRS provides similar double-node mechanisms for all management nodes of the service components, such as Manager, Presto, HDFS NameNodes, Hive Servers, HBase HMasters, YARN Resource Managers, Kerberos Servers, and Ldap Servers. All of them are deployed in active/standby mode or configured with load sharing, effectively preventing SPOFs from affecting system reliability.

- Comprehensive reliability mechanism

  By reliability analysis, the following measures to handle 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 checks 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.

8. **MRS provides a visualized big data cluster management interface in a unified manner, making O&M easier.**
   - On the big data cluster management interface, service startup and stopping, configuration modification, and health check are available. MRS also provides visualized and convenient cluster management, monitoring, and alarm functions. Additionally, you can check and audit the system health status in one click, ensuring normal system running and lowering system O&M costs.
   - After Simple Message Notification (SMN) is configured, MRS can send real-time cluster health status information, including cluster changes and component alarms in real time to you through SMS messages or emails, facilitating O&M, real-time monitoring, and real-time alarm sending.
   - MRS supports rolling patch upgrade and provides visualized patch release information and one-click patch installation without manual intervention, ensuring long-term stability of user clusters.
   - If a problem occurs when you use an MRS cluster, you can initiate O&M authorization on the MRS management console. O&M personnel can help you quickly locate the problem, and you can revoke the authorization at any time. You can also initiate log sharing on the MRS management console to share a specified log scope with O&M personnel, so that O&M personnel can locate faults without accessing the cluster.
   - MRS supports to dump logs about cluster creation failures to OBS for O&M personnel to obtain and analyze the logs.
9. **MRS has an open ecosystem and supports seamless interconnection with peripheral services, allowing you to quickly build a unified big data platform.**

- Based on MRS, a full-stack big data service, enterprises can build a unified big data platform for data ingestion, storage, analysis, and value mining with one click, and interconnect with DataArts Studio and data visualization services to help customers easily migrate data to the cloud, develop and schedule big data jobs, and display data. This frees customers from complex big data platform construction and professional big data calibration and maintenance so customers can stay more focused on industry applications and use one copy of data in multiple service scenarios. DataArts Studio is a one-stop data lifecycle development operations platform that provides a broad range of functions, such as data integration, development, governance, service, and visualization. MRS data can be ingested to DataArts Studio for collaborative one-click visualized development by leveraging DataArts Studio's visualized GUI, abundant data development types (script and job), fully-hosted job scheduling and O&M monitoring, and built-in industry data processing pipelines. This makes big data much easier to use, helps you quickly build big data processing centers, and enables fast monetization.
- MRS is fully compatible with the open source big data ecosystem. With abundant data and application migration tools, MRS helps you quickly migrate data from your own platforms without code modification and service interruption.

4 Application Scenarios

Big data is ubiquitous in our lives. Huawei Cloud MRS is suitable to process big data in the industries such as the Internet of things (IoT), e-commerce, finance, manufacturing, healthcare, energy, and government departments.

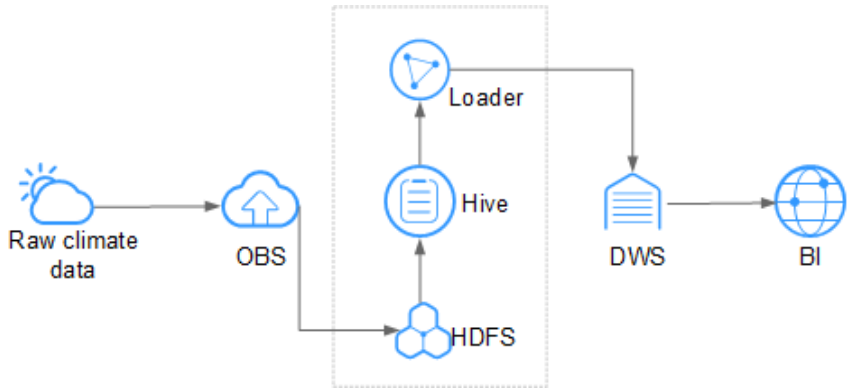
Large-scale data analysis

Large-scale data analysis is a major scenario in modern big data systems. Generally, an enterprise has multiple data sources. After data is accessed,extract, transform, and load (ETL) processing is required to generate modelized data for each service module to analyze and sort out data. This type of service has the following characteristics:

- The requirements for real-time execution are not high, and job execution time ranges from dozens of minutes to hours.
- The data volume is large.
- There are various data sources and diversified formats.
- Data processing usually consists of multiple tasks, and resources need to be planned in detail.

In the environmental protection industry, climate data is stored on OBS and periodically dumped into HDFS for batch analysis. 10 TB of climate data can be analyzed in 1 hour.

Figure 4-1 Large-scale data analysis in the environmental protection industry



MRS has the following advantages in this scenario.

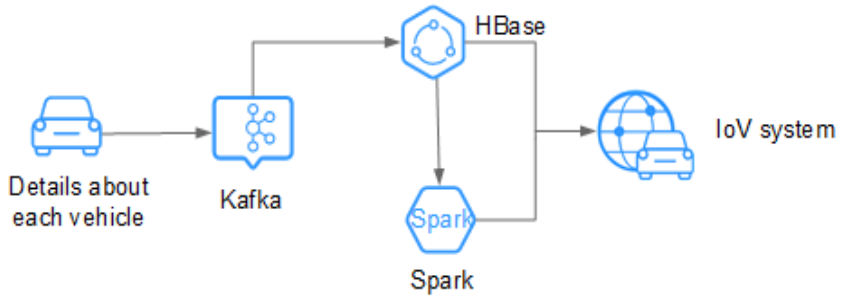
- Low cost: OBS offers cost-effective storage.
- Massive data analysis: TB/PB-level data is analyzed by Hive.
- Visualized data import and export tool: Loader exports data to Data Warehouse Service (DWS) for business intelligence (BI) analysis.

Large-scale data storage

A user who has a large amount of structured data usually requires index-based quasi-real-time query capabilities. For example, in an Internet of Vehicles (IoV) scenario, vehicle maintenance information is queried by vehicle number. Therefore, vehicle information is indexed based on vehicle numbers when it is being stored, to implement second-level response in this scenario. Generally, the data volume is large. The user may store data for one to three years.

For example, in the IoV industry, an automobile company stores data on HBase, which supports PB-level storage and CDR queries in milliseconds.

Figure 4-2 Large-scale data storage in the IoV industry



MRS has the following advantages in this scenario.

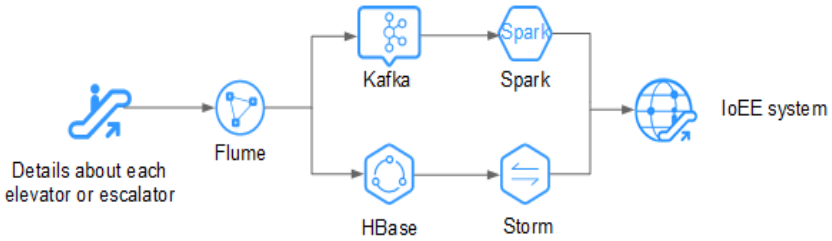
- Real time: Kafka accesses massive amounts of vehicle messages in real time.
- Massive data storage: HBase stores massive volumes of data and supports data queries in milliseconds.
- Distributed data query: Spark analyzes and queries massive volumes of data.

Real-time data processing

Real-time data processing is usually used in scenarios such as anomaly detection, fraud detection, rule-based alarming, and service process monitoring. Data is processed while it is being inputted to the system.

For example, in the Internet of elevators & escalators (IoEE) industry, data of smart elevators and escalators is imported to MRS streaming clusters in real time for real-time alarming.

Figure 4-3 Low-latency streaming processing in the IoEE industry



MRS has the following advantages in this scenario.

- Real-time data ingestion: Flume implements real-time data ingestion and provides various data collection and storage access methods.
- Data source access: Kafka accesses data of tens of thousands of elevators and escalators in real time.

5 How Do I Select an MRS Version?

MRS Cluster Versions

There are two MRS cluster versions: normal and LTS. The components and features of clusters of different versions are slightly different. You can select a version based on service requirements.

- **Normal**
  - **Functions**

    This normal version provides basic cluster operations, such as configuration, management, and O&M. For details, see **MapReduce Service User Guide**.
  - **Components**

    In addition to common components, clusters of the normal version also support components such as Presto, Impala, Kudu, and Sqoop. You can select components based on cluster versions. For details about components of clusters of different versions, see **List of MRS Component Versions** and **MapReduce Service Component Operation Guide**.
- **LTS (long term support)**
  - **Functions**

    In addition to basic cluster operations, clusters of the LTS version support version upgrade. To use this function, contact technical support.
  - **Components**

    In addition to common components, clusters of the LTS version also support HetuEngine, IoTDB, and CDL. You can select components based on cluster versions. For details about components of clusters of different versions, see **List of MRS Component Versions** and **MapReduce Service Component Operation Guide**.

Version Selection Suggestions

- Clusters of the LTS version support version upgrade. To make your clusters upgradable, choose the LTS version.
- Clusters of the LTS version can be deployed in different AZs to implement cross-AZ DR. To make your clusters more secure and have higher DR capabilities, choose the LTS version.

- Clusters of the LTS version support HetuEngine, IoTDB, and CDL. To use HetuEngine, IoTDB, or CDL, choose the LTS version.

  NOTE

  A purchased LTS cluster cannot be switched to the normal edition.

Billing Differences Between Versions

The normal and LTS versions have different functions and therefore they are billed differently. For details, see **Billing**. You can also use the **price calculator** to quickly calculate the reference price of an MRS cluster by selecting the cluster version and node specifications that you need.

6 Components

6.1 List of MRS Component Versions

Components and Versions

Table 6-1 lists the components and their versions required by each MRS cluster version.

NOTE

- Hadoop includes HDFS, Yarn, and MapReduce components.
- DBService, ZooKeeper, KrbServer, and LdapServer are components used in the cluster and are not displayed during cluster creation.
- MRS component versions must be consistent with open-source component versions.
- Components in an MRS cluster cannot be upgraded. Purchase a cluster with components of the required version.

Table 6-1 MRS component versions

Component	MRS 1.9.2 (Applicable to MRS 1.9.*x*)	MRS 3.1.0	MRS 3.1.2-LTS	MRS 3.1.5	MRS 3.2.0-LTS.1
CarbonData	1.6.1	2.0.1	2.2.0	2.2.0	2.2.0
CDL	-	-	-	-	1.0.0
ClickHouse	-	21.3.4.25	21.3.4.25	21.3.4.25	22.3.2.2
DBService	1.0.0	2.7.0	2.7.0	2.7.0	2.7.0
Flink	1.7.0	1.12.0	1.12.2	1.12.2	1.15.0
Flume	1.6.0	1.9.0	1.9.0	1.9.0	1.9.0

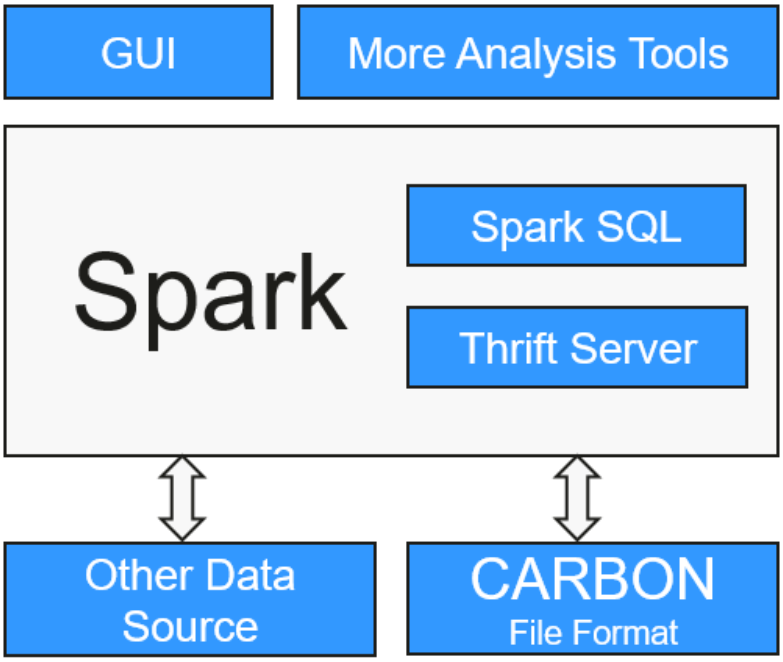
Component	MRS 1.9.2 (Applicable to MRS 1.9.*x*)	MRS 3.1.0	MRS 3.1.2-LTS	MRS 3.1.5	MRS 3.2.0-LTS.1
Guardian	-	-	-	0.1.0	-
HBase	1.3.1	2.2.3	2.2.3	2.2.3	2.2.3
HDFS	2.8.3	3.1.1	3.1.1	3.1.1	3.3.1
HetuEngine	-	-	1.2.0	-	1.2.0
Hive	2.3.3	3.1.0	3.1.0	3.1.0	3.1.0
Hudi	-	0.7.0	0.9.0	0.9.0	0.11.0
Hue	3.11.0	4.7.0	4.7.0	4.7.0	4.7.0
Impala	-	3.4.0	-	3.4.0	-
IoTDB	-	-	-	-	0.14.0
Kafka	1.1.0	2.11-2.4.0	2.11-2.4.0	2.11-2.4.0	2.11-2.4.0
KafkaManager	1.3.3.1	-	-	-	-
KrbServer	1.15.2	1.17	1.18	1.18	1.18
Kudu	-	1.12.1	-	1.12.1	-
LdapServer	1.0.0	2.7.0	2.7.0	2.7.0	2.7.0
Loader	2.0.0	-	1.99.3	-	1.99.3
MapReduce	2.8.3	3.1.1	3.1.1	3.1.1	3.3.1
Oozie	-	5.1.0	5.1.0	5.1.0	5.1.0
OpenTSDB	2.3.0	-	-	-	-
Presto	0.216	333	-	333	-
Phoenix (integrated in HBase)	-	5.0.0	5.0.0	5.0.0	5.0.0
Ranger	1.0.1	2.0.0	2.0.0	2.0.0	2.0.0
Spark	2.2.2	-	-	-	-
Spark2x	-	2.4.5	3.1.1	3.1.1	3.1.1
Sqoop	-	1.4.7	-	1.4.7	-
Storm	1.2.1	-	-	-	-

Component	MRS 1.9.2 (Applicable to MRS 1.9.*x*)	MRS 3.1.0	MRS 3.1.2-LTS	MRS 3.1.5	MRS 3.2.0-LTS.1
Tez	0.9.1	0.9.2	0.9.2	0.9.2	0.9.2
Yarn	2.8.3	3.1.1	3.1.1	3.1.1	3.3.1
ZooKeeper	3.5.1	3.5.6	3.6.3	3.6.3	3.6.3
MRS Manager	1.9.2	-	-	-	-
FusionInsight Manager	-	8.1.0	8.1.2	8.1.2	8.2.0.1

6.2 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 6-1 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).

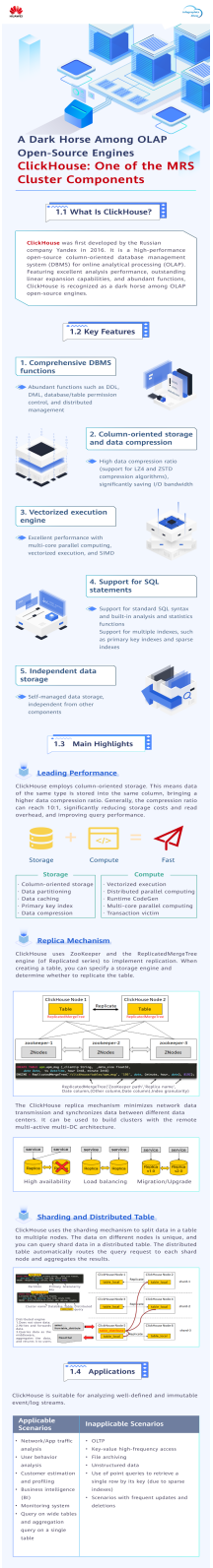
Key technologies and advantages of CarbonData

- 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/**.

6.3 ClickHouse

6.3.1 Infographics for ClickHouse



6.3.2 ClickHouse

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 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 6-2**:

Table 6-2 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 6-3**:

Table 6-3 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/**.

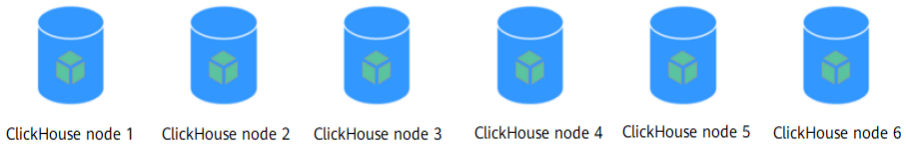
ClickHouse Enhanced Open Source Features

MRS ClickHouse has advantages such as automatic cluster mode, HA deployment, and smooth and elastic scaling.

- Automatic Cluster Mode

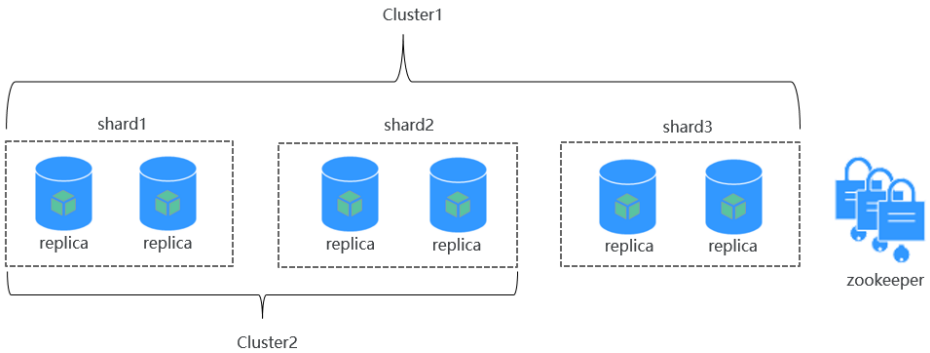
  As shown in **Figure 6-2**, a cluster consists of multiple ClickHouse nodes, which has no central node. It is more of a static resource pool. If the ClickHouse cluster mode is used for services, you need to pre-define the cluster information in the configuration file of each node. Only in this way, services can be correctly accessed.

  Figure 6-2 ClickHouse cluster

  Users are unaware of data partitions and replica storage in common database systems. However, ClickHouse allows you to proactively plan and define detailed configurations such as shards, partitions, and replica locations. The ClickHouse instance of MRS packs the work in a unified manner and adapts it to the automatic mode, implementing unified management, which is flexible and easy to use. A ClickHouse instance consists of three ZooKeeper nodes and multiple ClickHouse nodes. The Dedicated Replica mode is used to ensure high reliability of dual data copies.

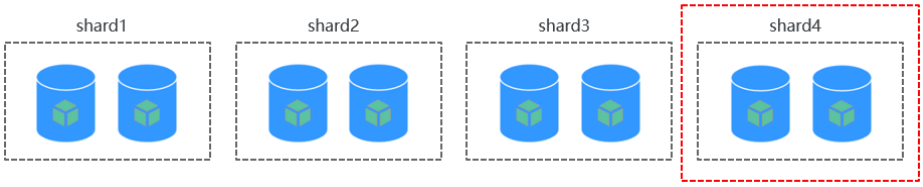
  Figure 6-3 ClickHouse cluster structure

- Smooth and Elastic Scaling

  As business grows rapidly, MRS provides ClickHouse, a data migration tool, for scenarios such as the cluster's storage capacity or CPU compute resources approaching the limit. This tool is used to migrate some partitions of one or multiple MergeTree tables on several ClickHouseServer nodes to the same tables on other ClickHouseServer nodes. In this way, service availability is ensured and smooth capacity expansion is implemented.

  When you add ClickHouse nodes to a cluster, use this tool to migrate some data from the existing nodes to the new ones for data balancing after the expansion.

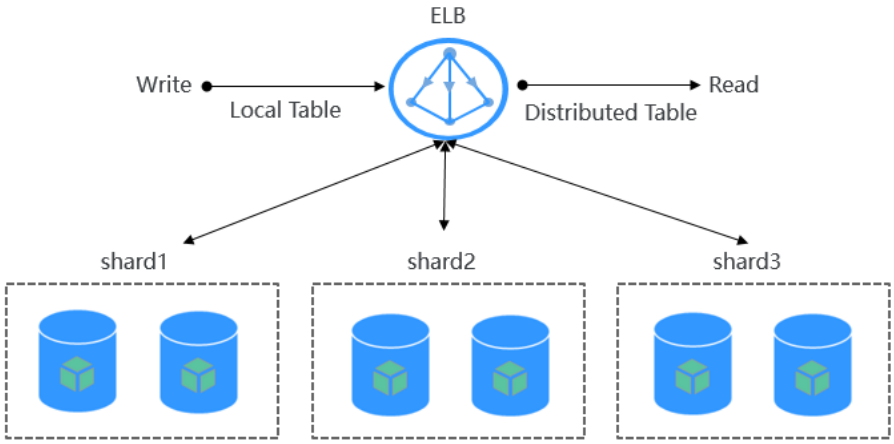


- HA Deployment Architecture

  MRS uses the ELB-based high availability (HA) deployment architecture to automatically distribute user access traffic to multiple backend nodes, expanding service capabilities to external systems and improving fault tolerance. As shown in **Figure 6-4**, when a client application requests a cluster, Elastic Load Balance (ELB) is used to distribute traffic. With the ELB polling mechanism, data is written to local tables and read from distributed tables on different nodes. In this way, data read/write load and high availability of application access are guaranteed.

  After the ClickHouse cluster is provisioned, each ClickHouse instance node in the cluster corresponds to a replica, and two replicas form a logical shard. For example, when creating a ReplicatedMergeTree table, you can specify shards so that data can be automatically synchronized between two replicas in the same shard.

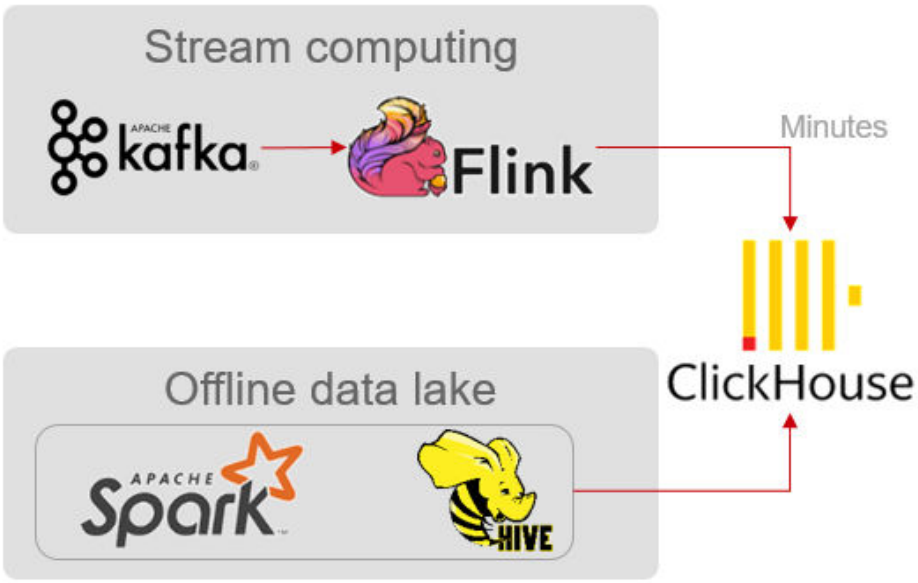
Figure 6-4 HA deployment architecture



Relationships Between ClickHouse and Other Components

ClickHouse depends on ZooKeeper for installation and deployment.

Flink stream computing applications are used to generate common report data (detail flat-wide tables) and write the report data to ClickHouse in quasi-real time. Hive/Spark jobs are used to generate common report data (detail flat-wide tables) and batch import the data to ClickHouse.



NOTE

Currently, ClickHouse does not support interconnection with Kafka in normal mode or HDFS in security mode.

6.4 CDL

6.4.1 CDL Basic Principles

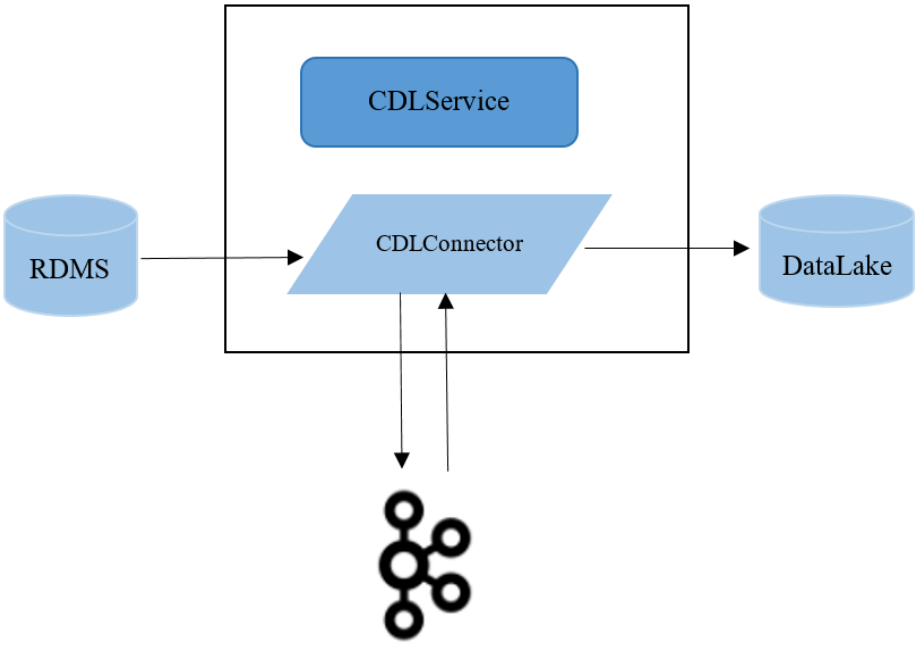
Overview

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 pushes them to Kafka. Then, Sink Connector pushes the events to the big data ecosystem.

Currently, CDL supports MySQL, PostgreSQL, Hudi, Kafka, and ThirdParty-Kafka data sources. Data can be written to Kafka, Hudi, GaussDB(DWS), and ClickHouse.

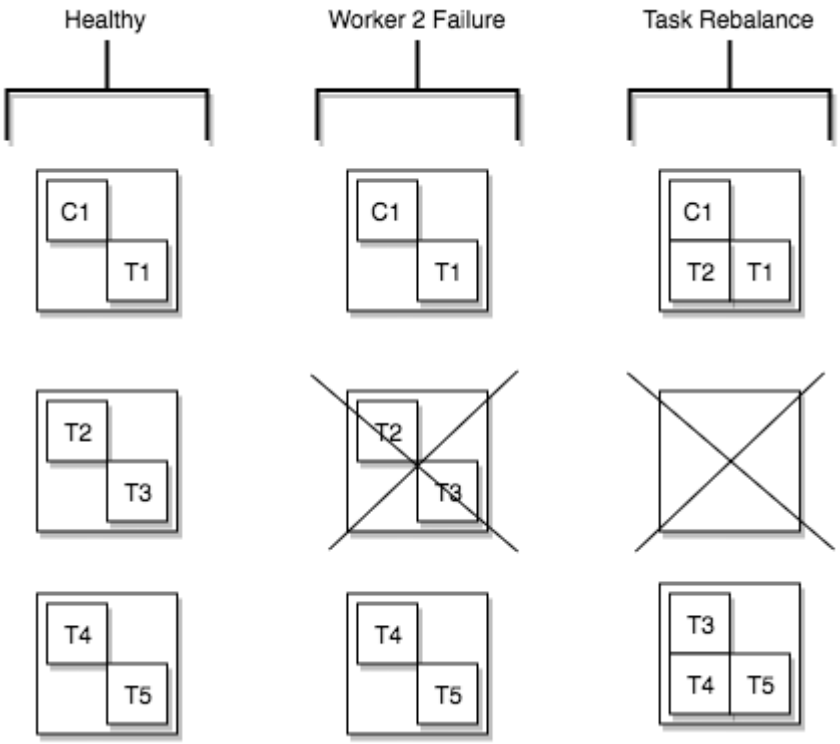
CDL structure

The CDL service has two important roles: CDLConnector and CDLService. CDLConnector, including Source Connector and Sink Connector, executes data capture jobs. CDLService manages and creates jobs.



The CDLService instances of the CDL service work in multi-active mode. Any CDLService instance 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 6-5 Rebalance of a task



6.4.2 Relationship Between CDL and Other Components

The CDL component is based on the Kafka Connect framework. Captured data is forwarded using Kafka topics. Therefore, the CDL component depends on the Kafka component. In addition, the CDL component stores task metadata and monitoring information that are also stored in a database. Therefore, the CDL component also depends on the DBService component.

6.5 DBService

6.5.1 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, CDL, Flink, HuteEngine, Kafka, Metadata, 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 6-6 shows the DBService logical architecture.

Figure 6-6 DBService architecture

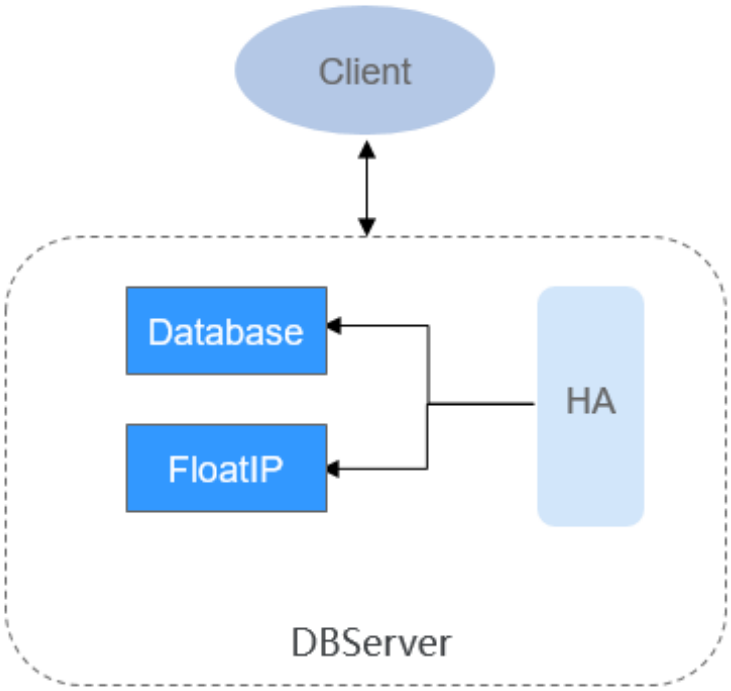


Table 6-4 describes the modules shown in **Figure 6-6**

Table 6-4 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.

6.5.2 Relationship Between DBService and Other Components

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

6.6 Flink

6.6.1 Flink Basic Principles

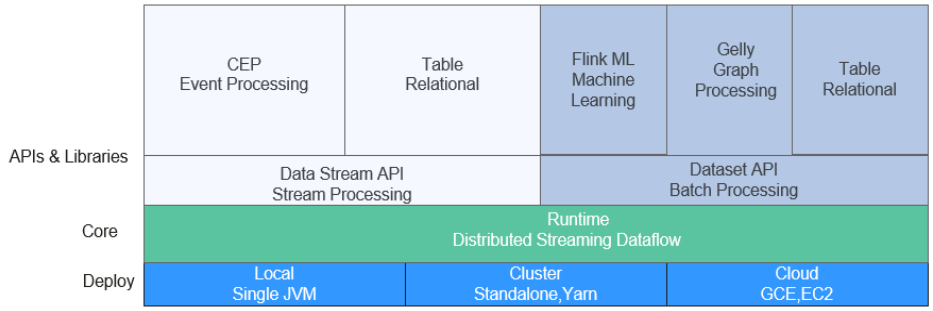
Overview

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 6-7 shows the technology stack of Flink.

Figure 6-7 Technology stack of Flink



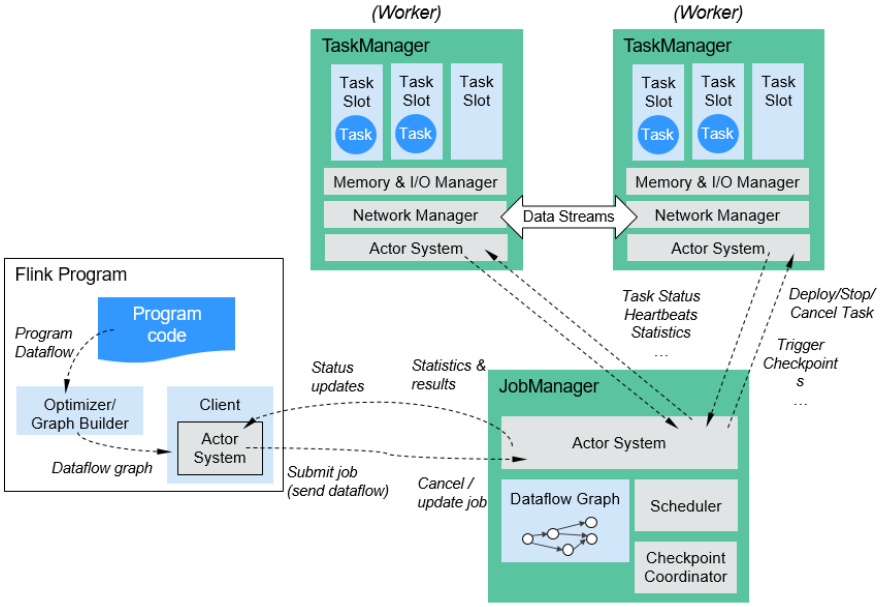
Flink provides the following features in the current version:

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

Flink Architecture

Figure 6-8 shows the Flink architecture.

Figure 6-8 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.

Flink Principles

- **Stream & Transformation & Operator**

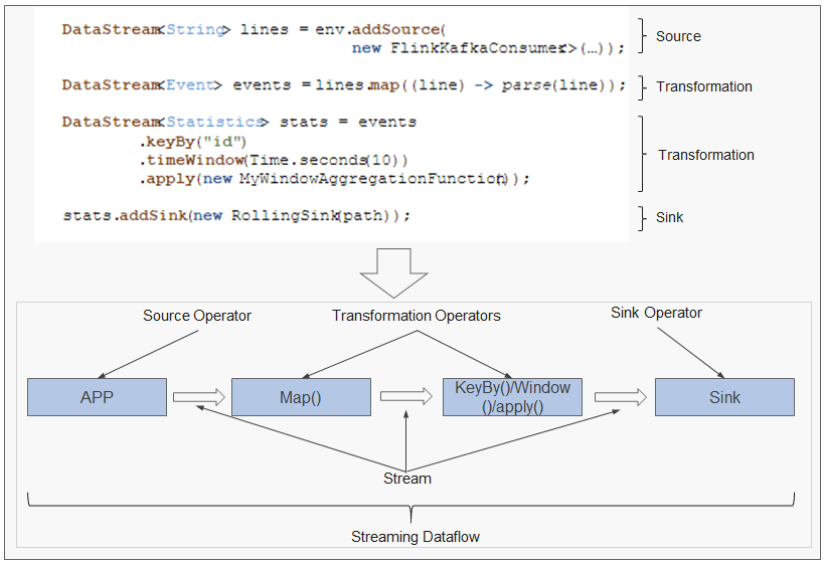
  A Flink program consists of two building blocks: stream and transformation.

  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 6-9 shows the streaming dataflow to which a Flink program is mapped.

Figure 6-9 Example of Flink DataStream



As shown in **Figure 6-9**, **FlinkKafkaConsumer** is a source operator; Map, KeyBy, TimeWindow, and Apply are transformation operators; RollingSink is a sink operator.

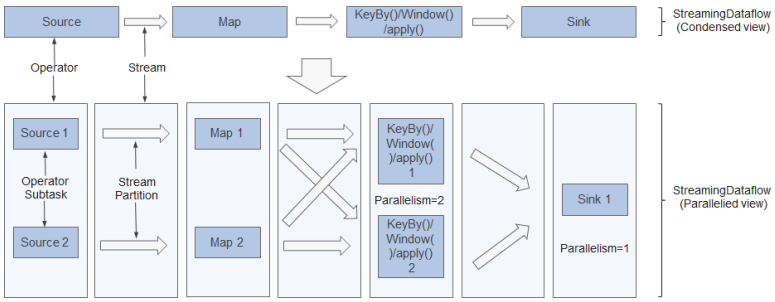
- **Pipeline Dataflow**

  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.

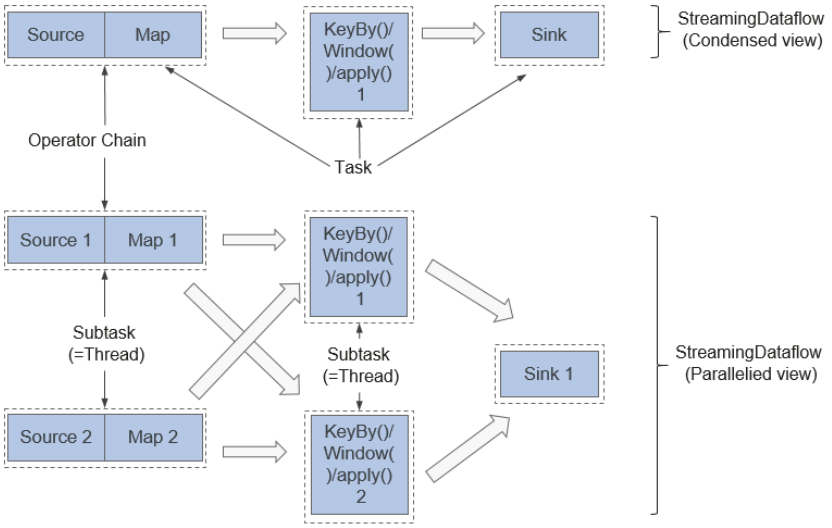
  – Operators with low density cannot be optimized. Each operator subtask is separately executed in different threads. The number of operator subtasks is the parallelism of that particular operator. The parallelism (the total number of partitions) of a stream is that of its producing operator. Different operators of the same program may have different levels of parallelism, as shown in **Figure 6-10**.

  Figure 6-10 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 6-11**.

Figure 6-11 Operator chain



- In the upper part of **Figure 6-11**, 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 6-11**, 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 customized 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 is 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
  ```

- 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
```

6.6.2 Flink HA Solution

Flink HA Solution

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:2181
high-availability.storageDir: hdfs:///flink/recovery
```

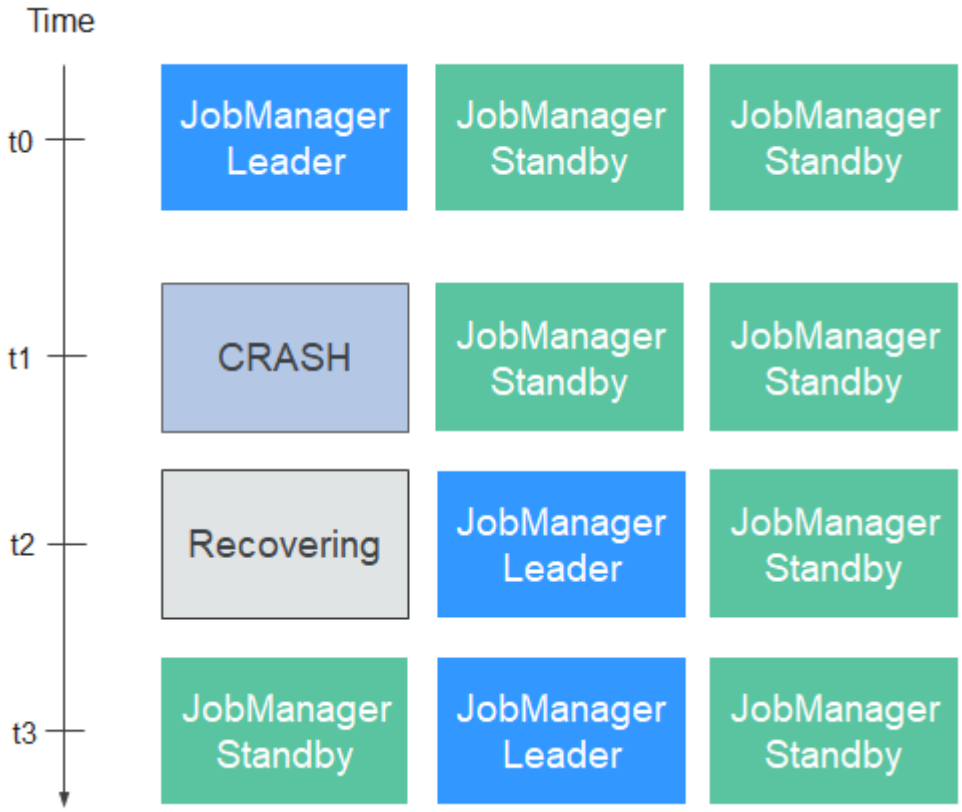
Yarn

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.

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 6-12** shows the process of a leader/standby JobManager switchover.

Figure 6-12 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

If you want to restore jobs, ensure that the startup policy is configured in Flink configuration files. 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.

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.

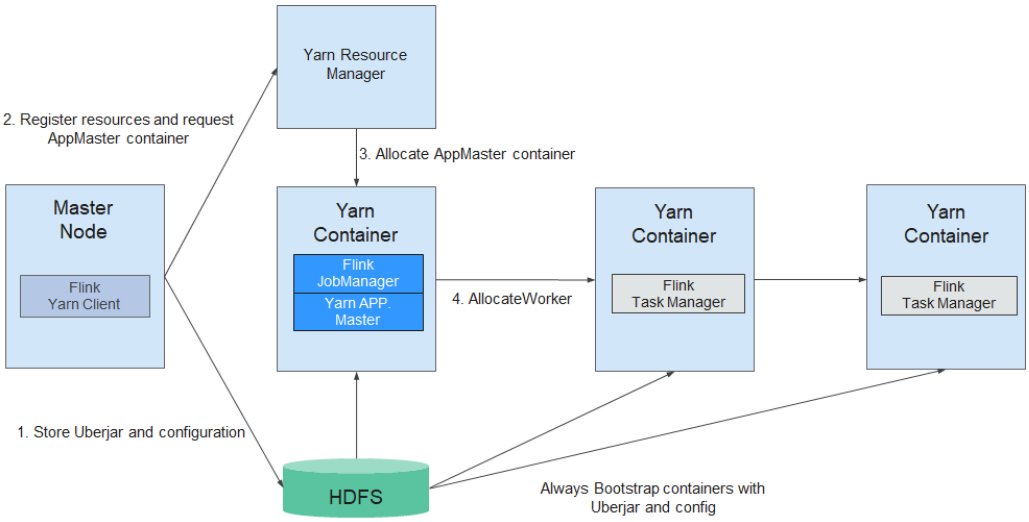
6.6.3 Relationships Between Flink and Other Components

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 6-13 shows the Yarn-based Flink cluster deployment.

Figure 6-13 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.

6.6.4 Flink Enhanced Open Source Features

6.6.4.1 Window

Enhanced Open Source Feature: Window

This section describes the sliding window of Flink and provides the sliding window optimization method.

Introduction to Window

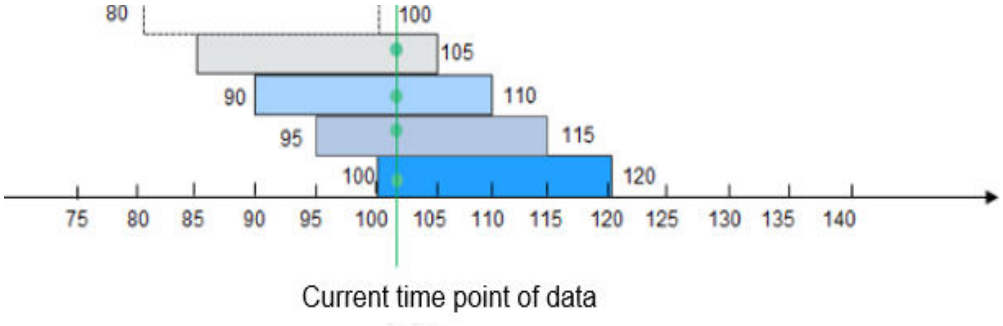
Data in a window is saved as intermediate results or original data. If you perform a sum operation (**window(SlidingEventTimeWindows.of(Time.seconds(20), Time.seconds(5))).sum**) on data in the window, only the intermediate result will be retained. If a custom window (**window(SlidingEventTimeWindows.of(Time.seconds(20), 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 6-14 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).

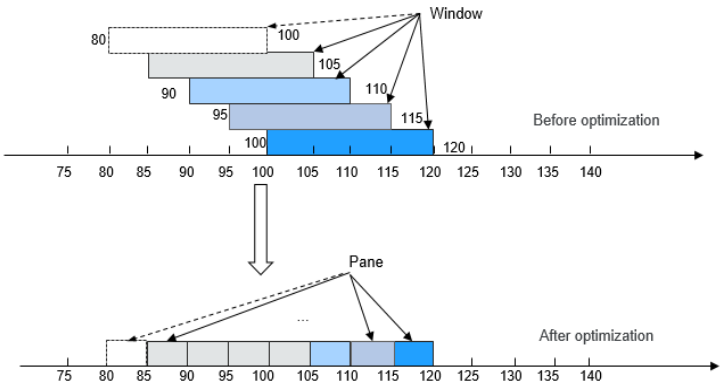
Window Optimization

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).

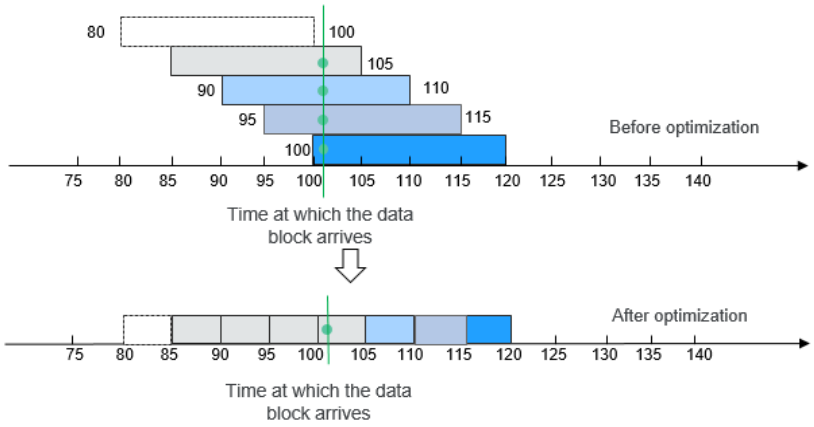
   Figure 6-15 Window optimization



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.

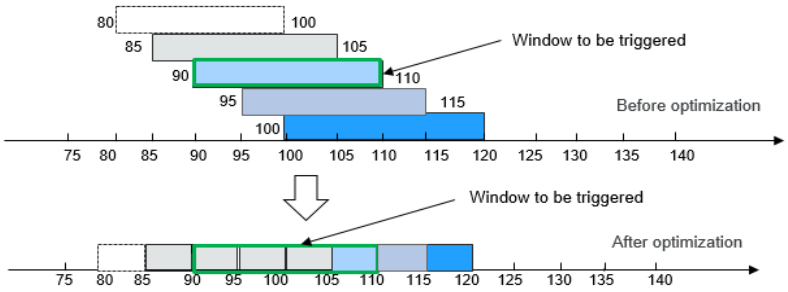
   A data block is saved only in one pane. In this case, only a data copy exists in the memory.

   Figure 6-16 Saving data in a window



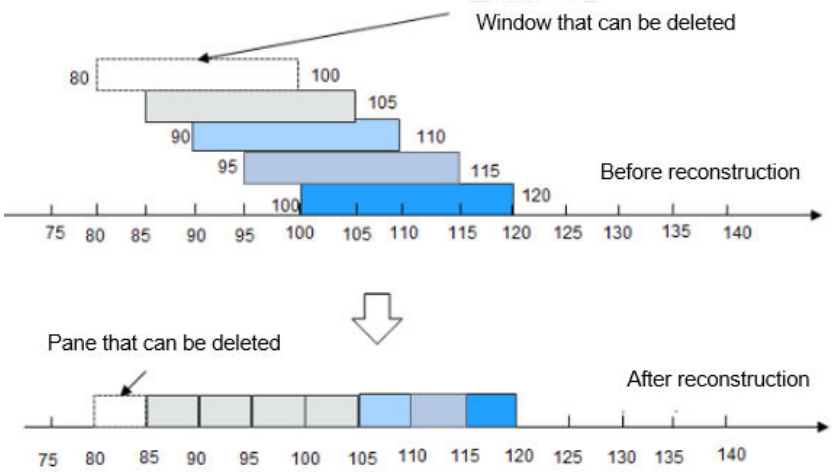
3. To trigger a window, compute all panes contained in the window, and combine all these panes into a complete window.

Figure 6-17 Triggering a window



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

Figure 6-18 Deleting a window



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

6.6.4.2 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:

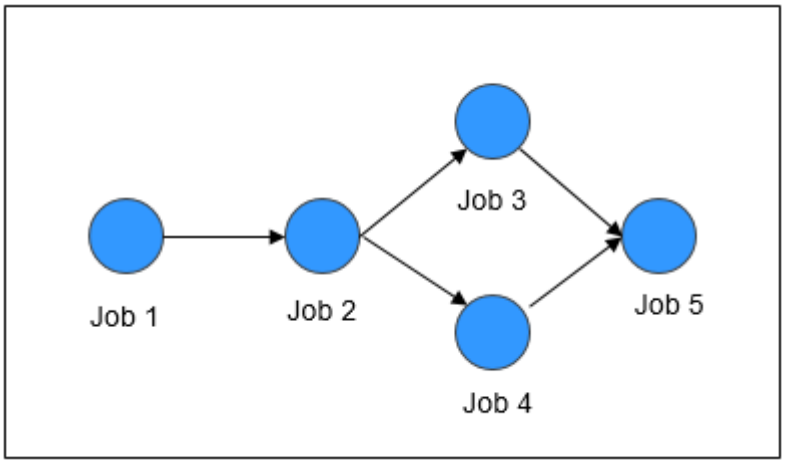
- Data transmission between jobs can be achieved using Kafka. For example, job A transmits data to the topic A in Kafka, and then job B and job C read data from the topic A in Kafka. This solution is simple and easy to implement, but the latency is always longer than 100 ms.

- 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

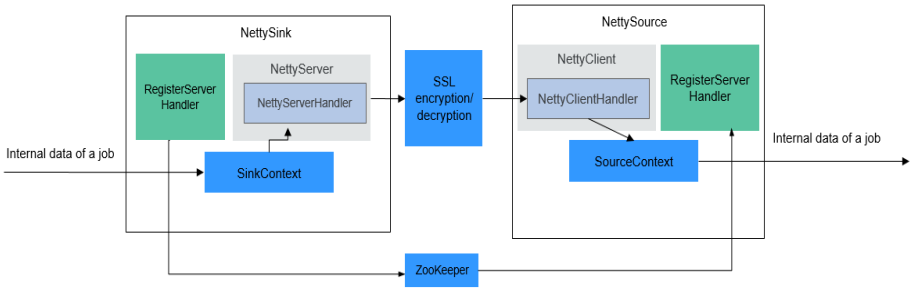
A pipeline consists of multiple Flink jobs connected through TCP. Upstream jobs can send data to downstream jobs. The flow diagram about data transmission is called a job pipeline, as shown in **Figure 6-19**.

Figure 6-19 Job pipeline



Job Pipeline Principles

Figure 6-20 Job Pipeline



- 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

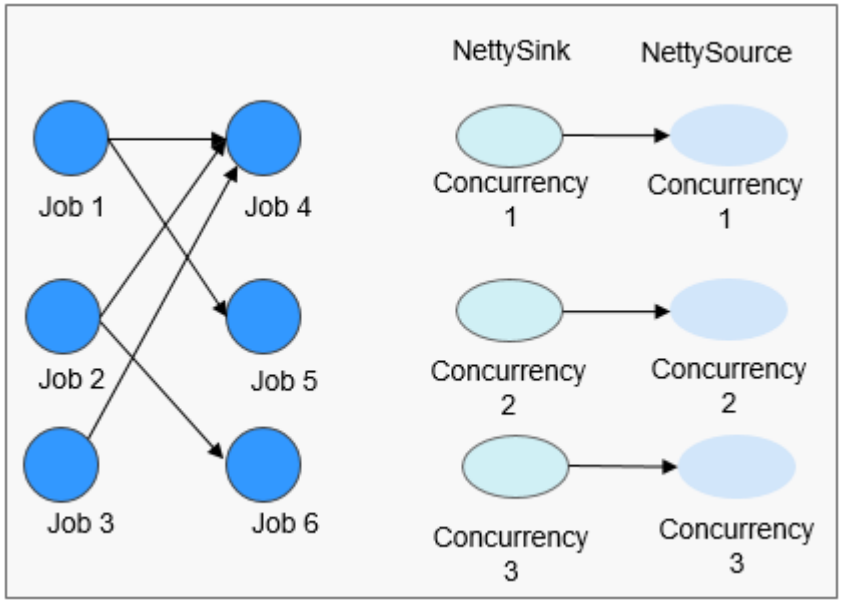
    NettyClient establishes a connection with NettyServer and uses NettyClientHandler to receive data. Each NettySource operator must have a unique name (specified by the user). NettyServer determines whether each client comes from different NettySources based on unique names. When a connection is established 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 6-21**.

Figure 6-21 Relationship diagram



6.6.4.3 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.

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.

6.6.4.4 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.

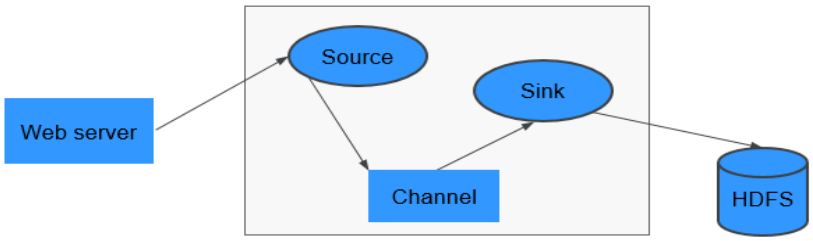
```
SELECT *
   FROM Ticker
   MATCH_RECOGNIZE (
     MEASURES
       AVG(E.price) AS avgPrice
     ONE ROW PER MATCH
     AFTER MATCH SKIP PAST LAST ROW
     PATTERN (A B+ C)
     SUBSET E = (A,C)
     DEFINE
       A AS A.price < 30,
       B AS B.price < 20,
       C AS C.price < 30
   )
```

6.7 Flume

6.7.1 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 6-22 Flume-NG architecture



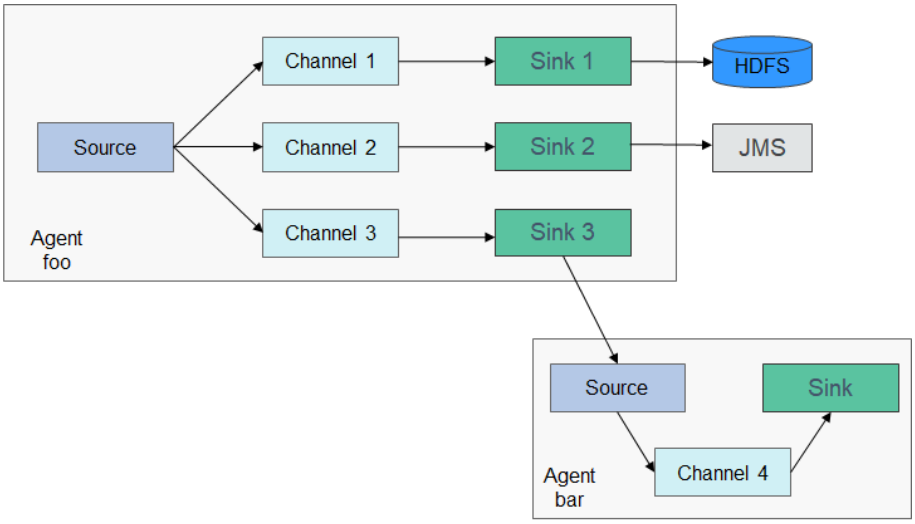
A Flume-NG consists of agents. Each agent consists of three components (source, channel, and sink). A source is used for receiving data. A channel is used for transmitting data. A sink is used for sending data to the next end.

Table 6-5 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 communications between agents, such as Avro A source must be associated with at least one channel.
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.

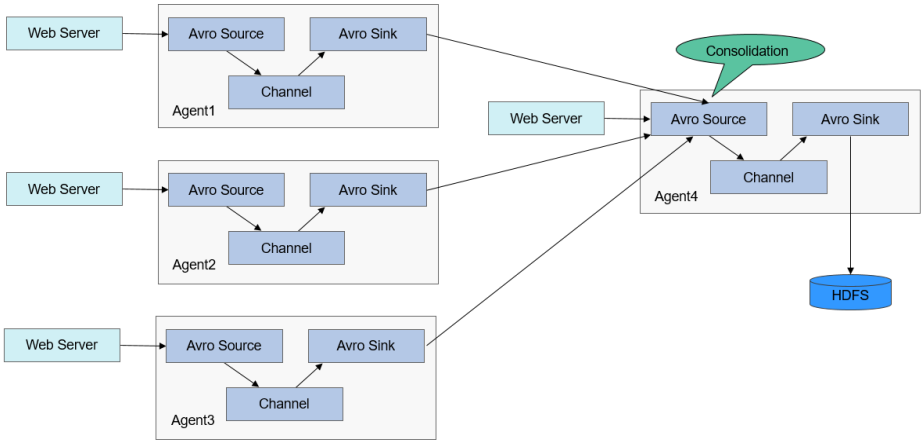
As shown in **Figure 6-23**, a Flume client can have multiple sources, channels, and sinks.

Figure 6-23 Flume structure



The reliability of Flume depends on transaction switchovers between agents. If the next agent is abnormal, 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 6-24 Flume cascading

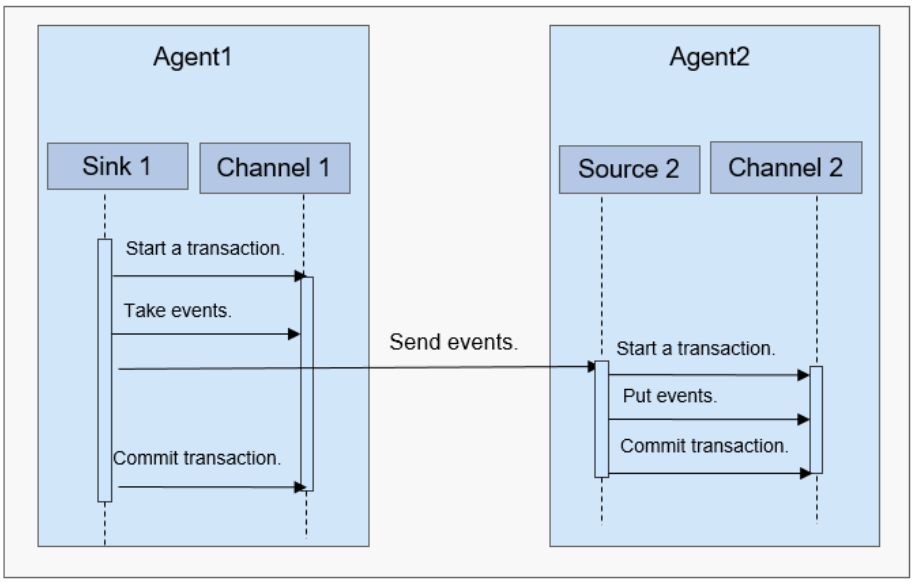


Principle

Reliability Between Agents

Figure 6-25 shows the data exchange between agents.

Figure 6-25 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.

6.7.2 Relationships 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.

6.7.3 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.

6.8 HBase

6.8.1 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 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 6-26 HBase architecture

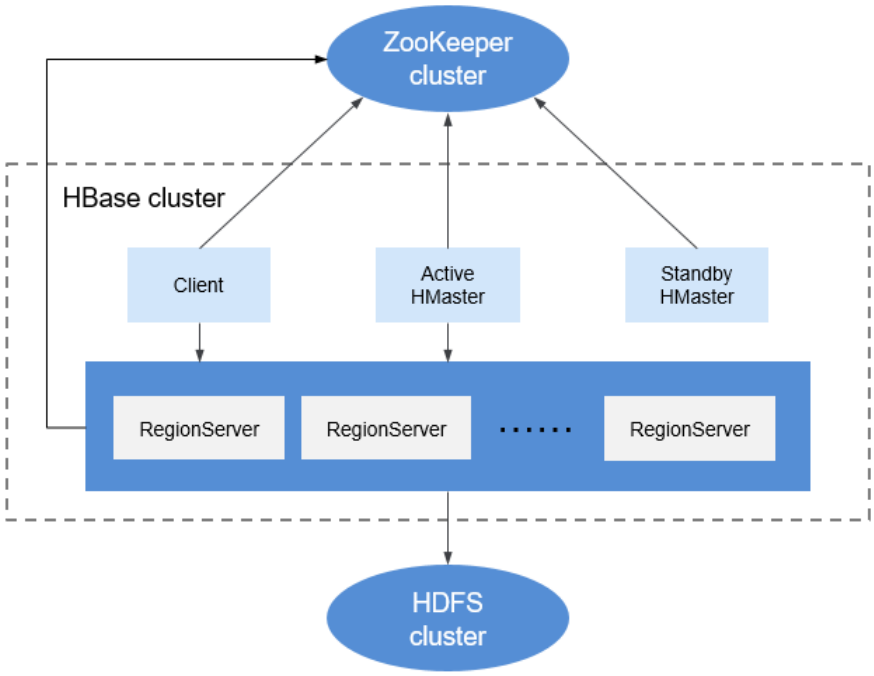


Table 6-6 Module description

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 6-27**. 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 6-27 HBase data model

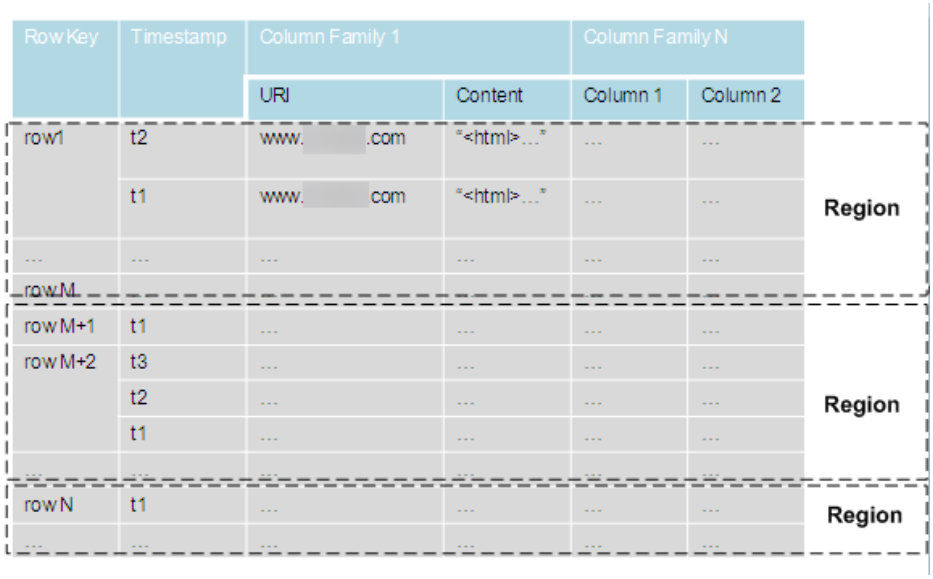


Table 6-7 Concepts

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 6-28** shows the data storage structure of RegionServer.

  Figure 6-28 RegionServer data storage structure

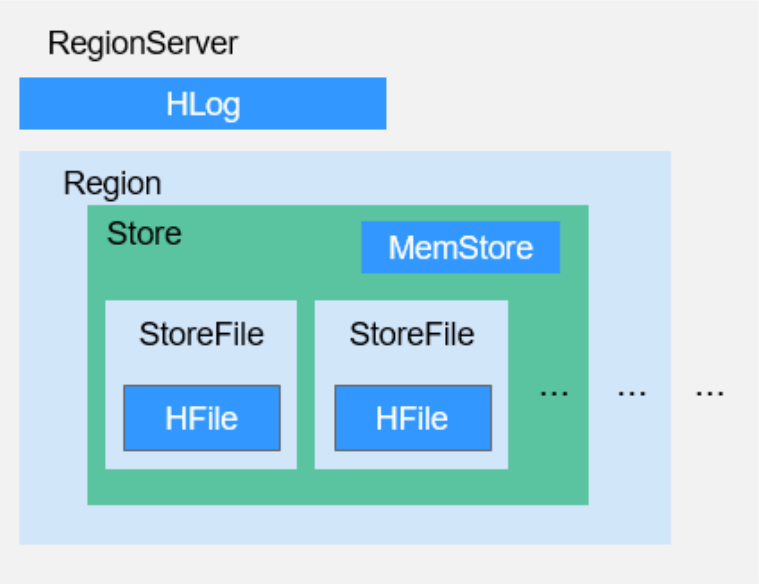


  Table 6-8 lists each component of Region described in **Figure 6-28**.

Table 6-8 Region structure description

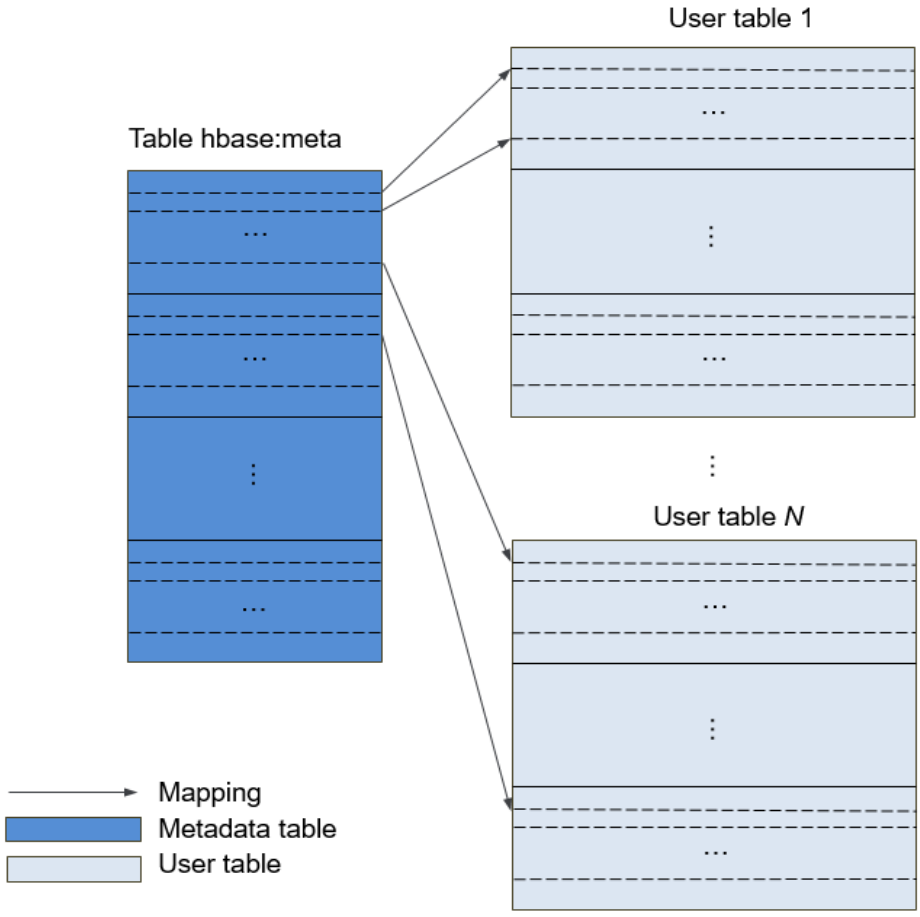
Module	Description
Store	A Region consists of one or multiple Stores. Each Store maps a column family in **Figure 6-27**.
MemStore	A Store contains one MemStore. The MemStore caches data inserted to a Region by the client. When the MemStore capacity reaches the upper limit, RegionServer flushes data 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 6-29 shows the mapping relationship between metadata tables and user tables.

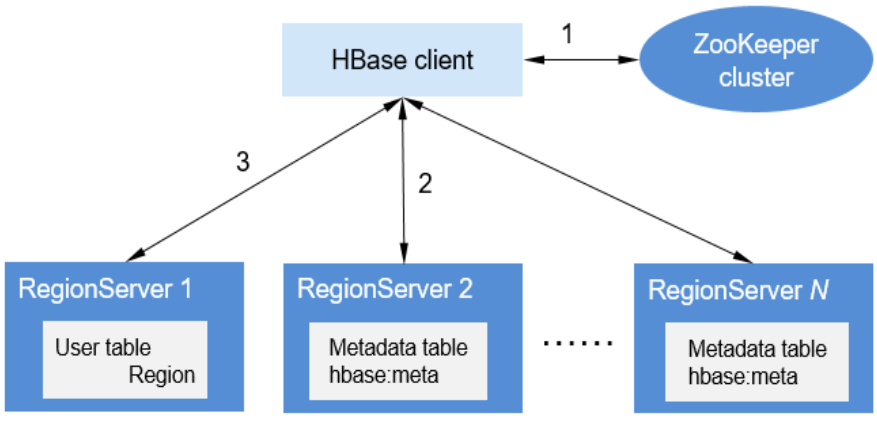
Figure 6-29 Mapping relationships between metadata tables and user tables



- **Data Operation Process**

  Figure 6-30 shows the HBase data operation process.

  Figure 6-30 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.

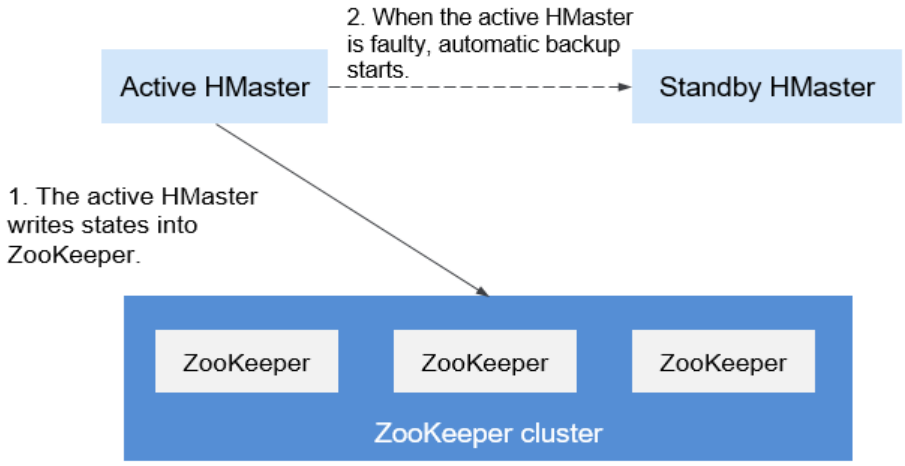
To improve data processing efficiency, the HBase client caches region information of the **hbase:meta** table and user table. When an application initiates a second data operation, the HBase client queries the region information from the memory. If no match is found in the memory, the HBase client performs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

6.8.2 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 6-31 HMaster HA implementation architecture



The HMaster HA architecture is built based on Ephemeral nodes (temporary nodes) created 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.

6.8.3 Relationship with Other Components

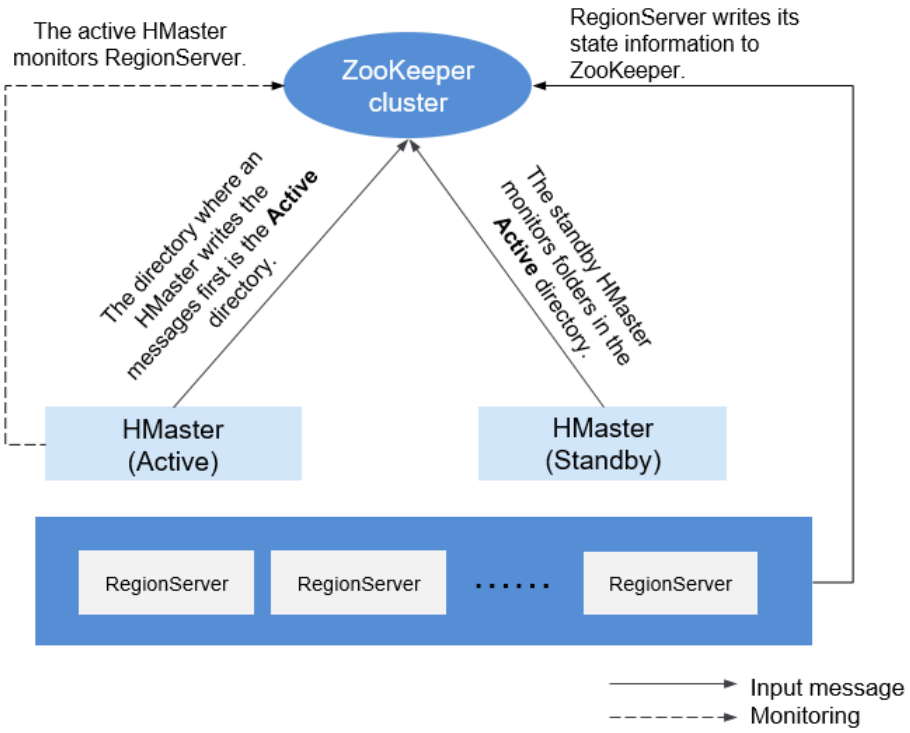
Relationship Between HDFS and HBase

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 6-32 describes the relationship between ZooKeeper and HBase.

Figure 6-32 Relationship between ZooKeeper and HBase



1. HRegionServer 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 HRegionServer 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.

6.8.4 HBase Enhanced Open Source Features

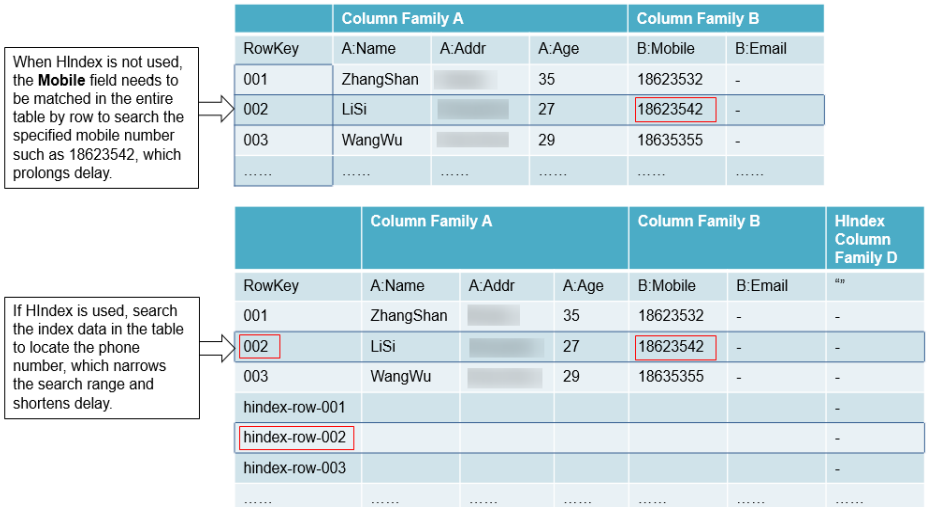
HIndex

HBase is a distributed Key-Value store. 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 6-33 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 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.
- Index tables must use HIndexWALPlayer to replay logs. WALPlayer cannot be used to replay logs.

```
hbase org.apache.hadoop.hbase.hindex.mapreduce.HIndexWALPlayer
Usage: WALPlayer [options] <wal inputdir> <tables> [<tableMappings>]
Read all WAL entries for <tables>.
If no tables ("") are specific, all tables are imported.
(Careful, even -ROOT- and hbase:meta entries will be imported in that case.)
Otherwise <tables> is a comma separated list of tables.

The WAL entries can be mapped to new set of tables via <tableMapping>.
<tableMapping> is a command separated list of targettables.
If specified, each table in <tables> must have a mapping.

By default WALPlayer will load data directly into HBase.
To generate HFiles for a bulk data load instead, pass the option:
  -Dwal.bulk.output=/path/for/output
  (Only one table can be specified, and no mapping is allowed!)
Other options: (specify time range to WAL edit to consider)
  -Dwal.start.time=[date|ms]
  -Dwal.end.time=[date|ms]
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 an index table, delete index data first.

Multi-point Division

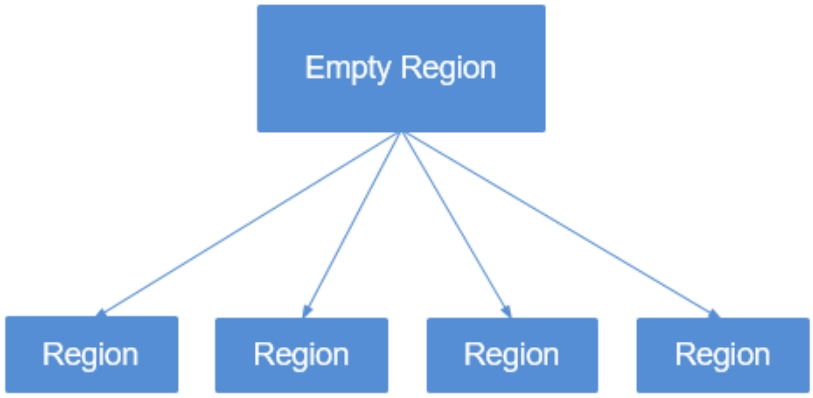
When you create tables that are pre-divided by region in HBase, you may not know the data distribution trend so the division by region may be inappropriate.

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 6-34 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 whitelist function is enabled for the standby cluster. Data can be pushed to clusters only by specifying IP addresses.
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.

HBase MOB

In the actual application scenarios, data in various sizes needs to be stored, for example, image data and documents. Data whose size is smaller than 10 MB can be stored in HBase. HBase can yield optimal read-and-write performance for data whose size is smaller than 100 KB. If the size of data stored in HBase is greater than 100 KB or even reaches 10 MB and the same number of data files are inserted, the total data amount is large, causing frequent compaction and split, high CPU consumption, high disk I/O frequency, and low performance.

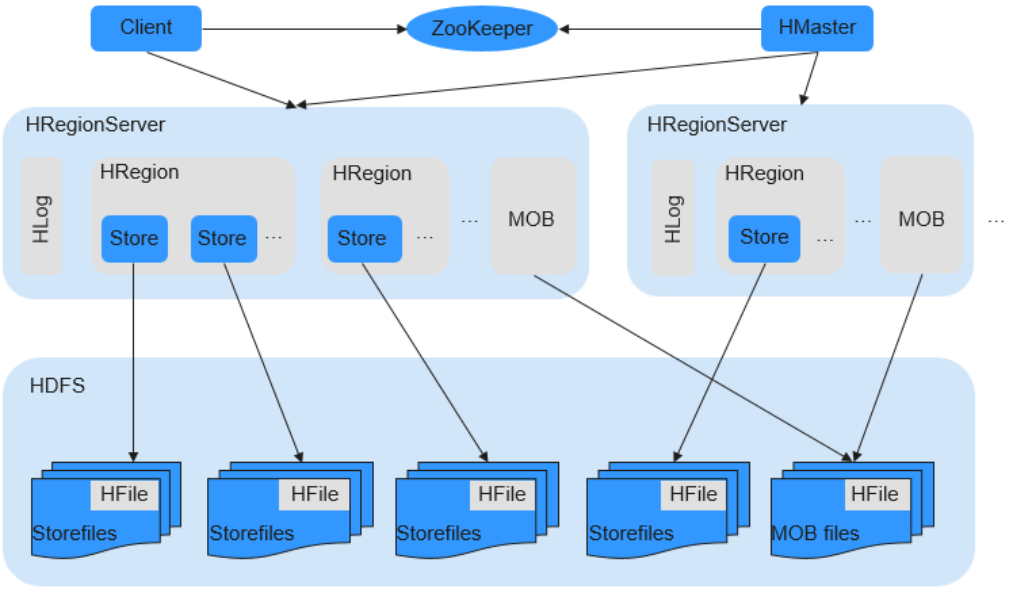
MOB data (whose size ranges from 100 KB to 10 MB) is stored in a file system (for example, HDFS) in HFile format. The expiredMobFileCleaner and Sweeper tools are used to manage HFiles and save the address and size information about the HFiles to the store of HBase as values. This significantly decreases the compaction and split frequency in HBase and improves performance.

As shown in **Figure 6-35**, MOB indicates mobstore stored on HRegion. Mobstore stores keys and values. Wherein, a key is the corresponding key in HBase, and a value is the reference address and data offset stored in the file system. When reading data, mobstore uses its own scanner to read key-value data objects and uses the address and data size information in the value to obtain target data from the file system.

NOTE

This function is not recommended for MRS 3.3.0 or later.

Figure 6-35 MOB data storage principle



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.

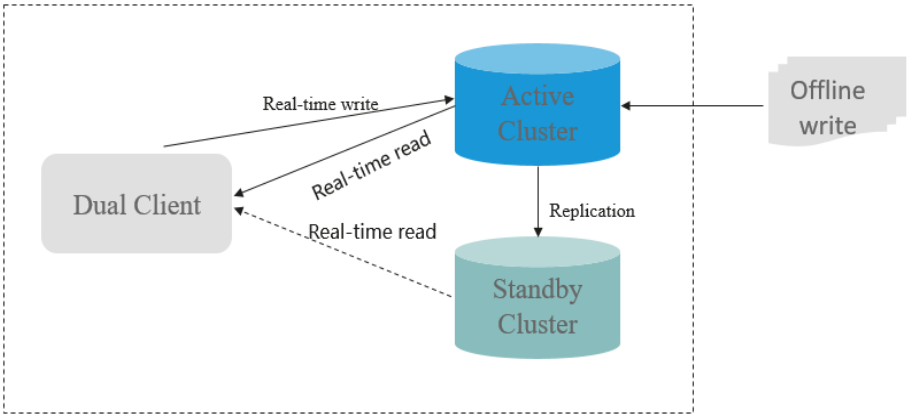
For such a scenario, HFS provides unified operation APIs similar to HBase function APIs.

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

NOTE

This feature is available in MRS 3.2.0 or later.

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.

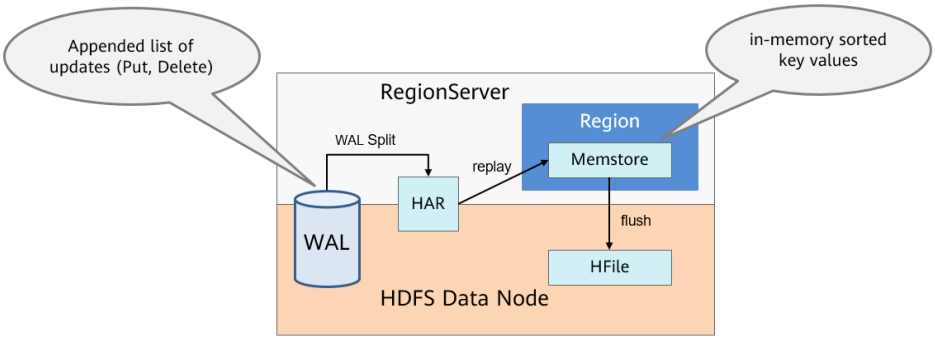
Writing Small Files Generated During WAL File Splitting to the HTTP Archive (HAR) File

NOTE

This feature is available in MRS 3.2.0 or later.

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.



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.

This feature is available in MRS 3.2.0 or later.

Self-Healing from HBase Hotspotting

This function applies to MRS 3.3.0 and later.

HBase is a distributed key-value database. Regions are the smallest units for HBase data management. Poorly designed row keys make requests directed at a few fixed regions. As a result, the service pressure is high on a node, causing performance deterioration or even request failures.

MetricController instances are added to HBase. After hotspotting detection is enabled, the request traffic of each RegionServer node is monitored. Through aggregation analysis, the nodes and regions with excessive requests can be identified, accelerating hotspotting detection. In addition, this self-healing function transfers workload or performs region splitting. For hotspotting caused by a single rowkey and sequential writes where the self-healing function does not work, traffic limiting is provided instead to minimize the impact on other services on a node.

6.9 HDFS

6.9.1 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.

NOTE

To use HDFS, ensure that the Hadoop service has been installed in the MRS cluster.

Architecture

HDFS consists of active and standby NameNodes and multiple DataNodes, as shown in **Figure 6-36**.

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 6-36 HA HDFS architecture

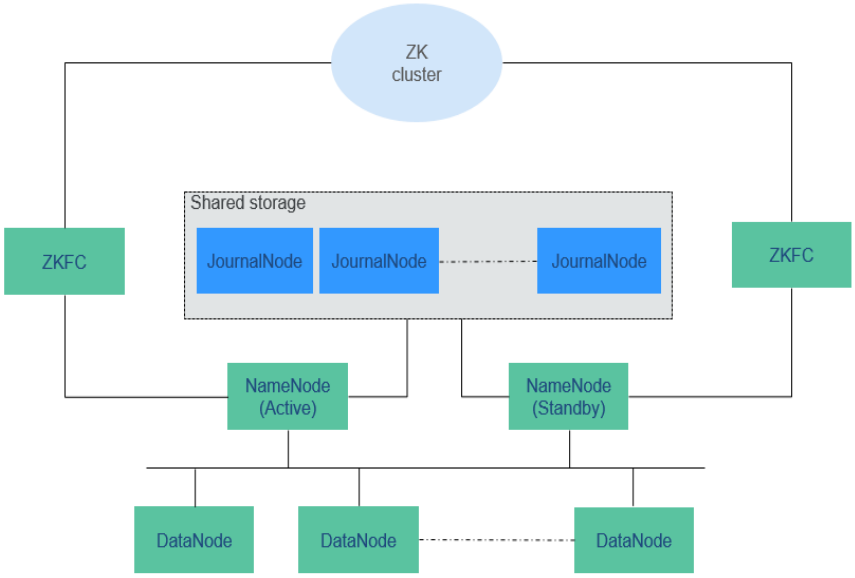


Table 6-9 describes the functions of each module shown in **Figure 6-36**.

Table 6-9 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.

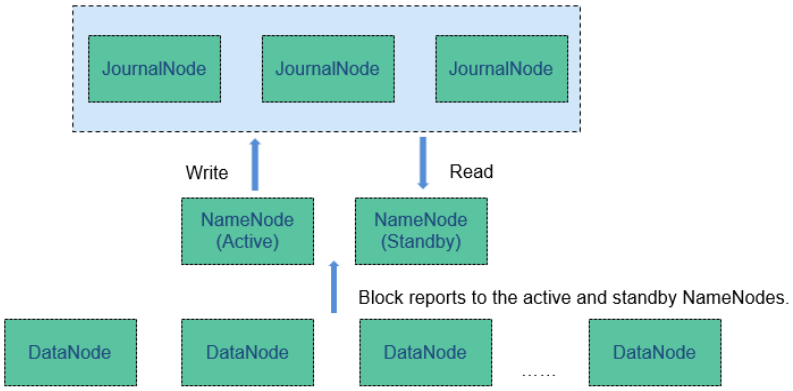
  A shared storage system is required to support metadata synchronization of the active and standby NameNodes. This version provides Quorum Journal Manager (QJM) HA solution, as shown in **Figure 6-37**. A group of JournalNodes are used to synchronize metadata between the active and standby NameNodes.

  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 6-37 QJM-based HDFS architecture

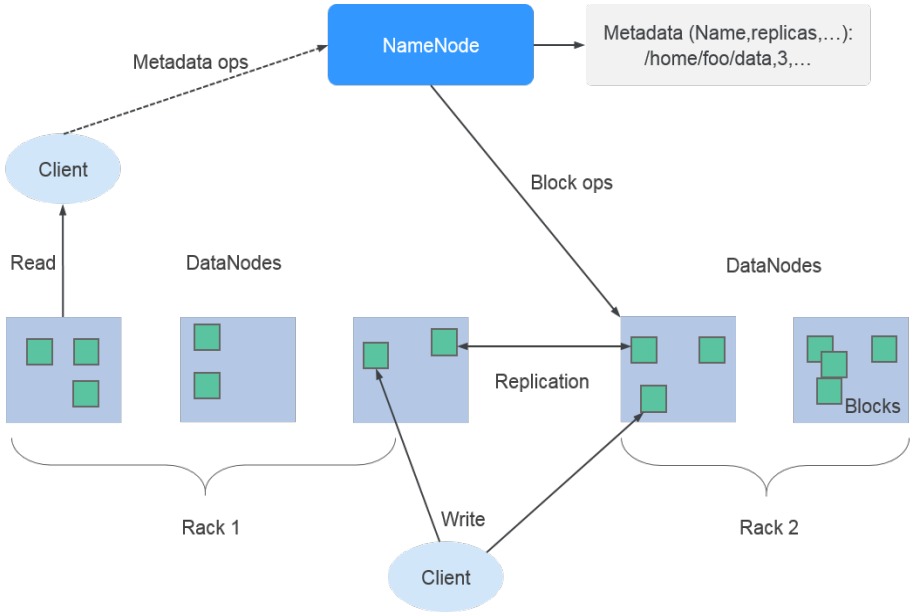


HDFS Reliability

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 6-38 HDFS architecture



Supports scheduling for node balance on HDFS and disk balance on a single node, improving HDFS storage performance after node or disk scale-out.

6.9.2 HDFS HA Solution

HDFS HA Background

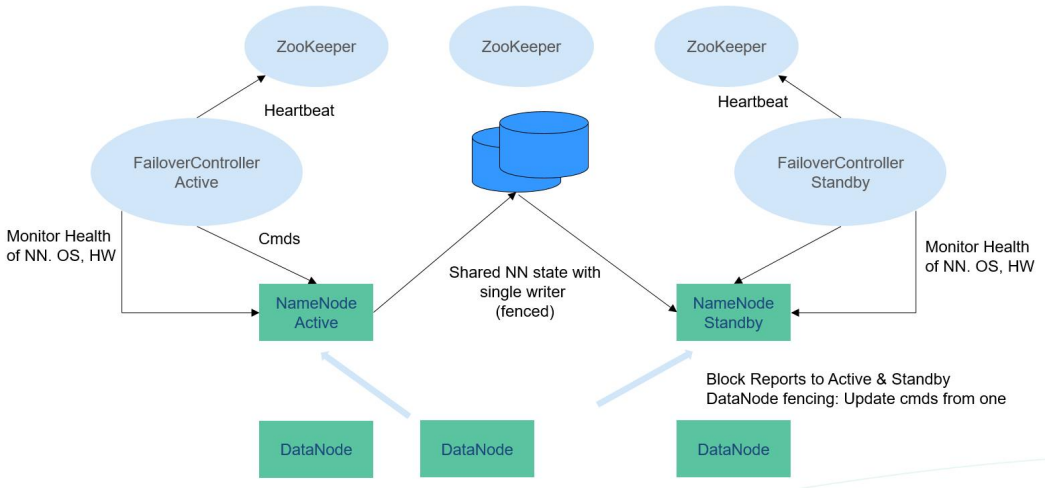
In versions earlier than Hadoop 2.0.0, SPOF occurs in the HDFS cluster. Each cluster has only one NameNode. If the host where the NameNode is located is faulty, the HDFS cluster cannot be used unless the NameNode is restarted or started on another host. This affects the overall availability of HDFS in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

HDFS HA Implementation

Figure 6-39 Typical HA deployment



In a typical HA cluster (as shown in **Figure 6-39**), 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.

6.9.3 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.

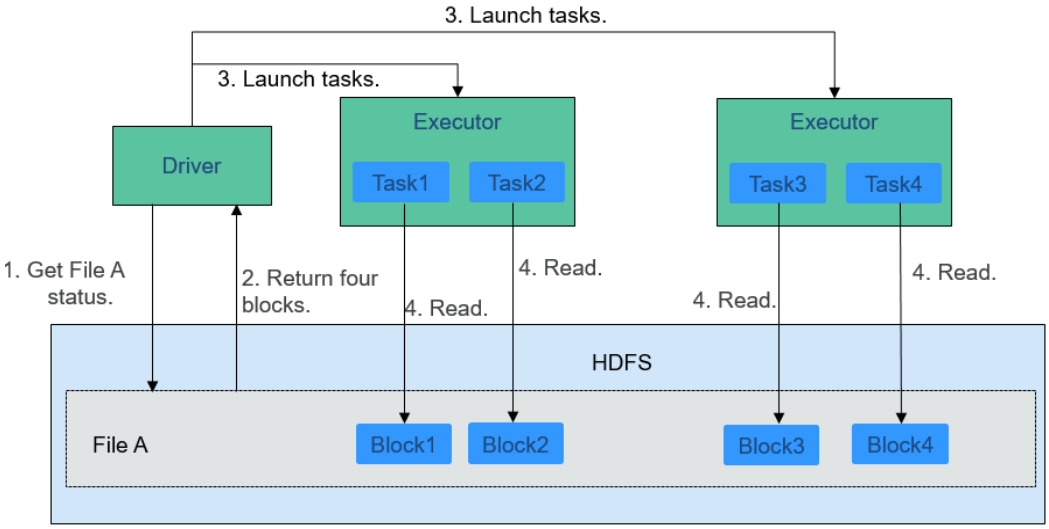
Relationship Between HDFS and Spark

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 6-40 shows how data is read from a file.

Figure 6-40 File reading process



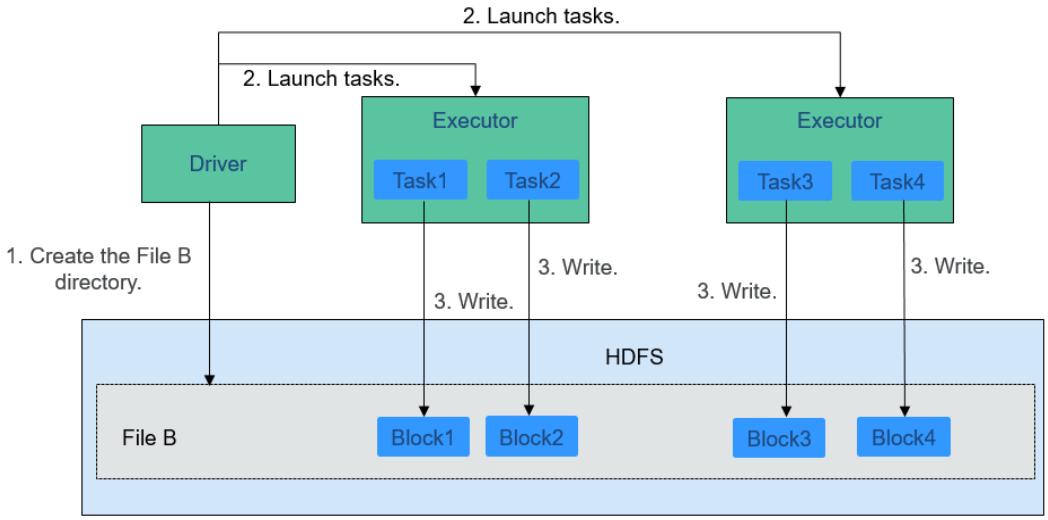
The file reading process is as follows:

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 6-41 shows how data is written to a file.

Figure 6-41 File writing process



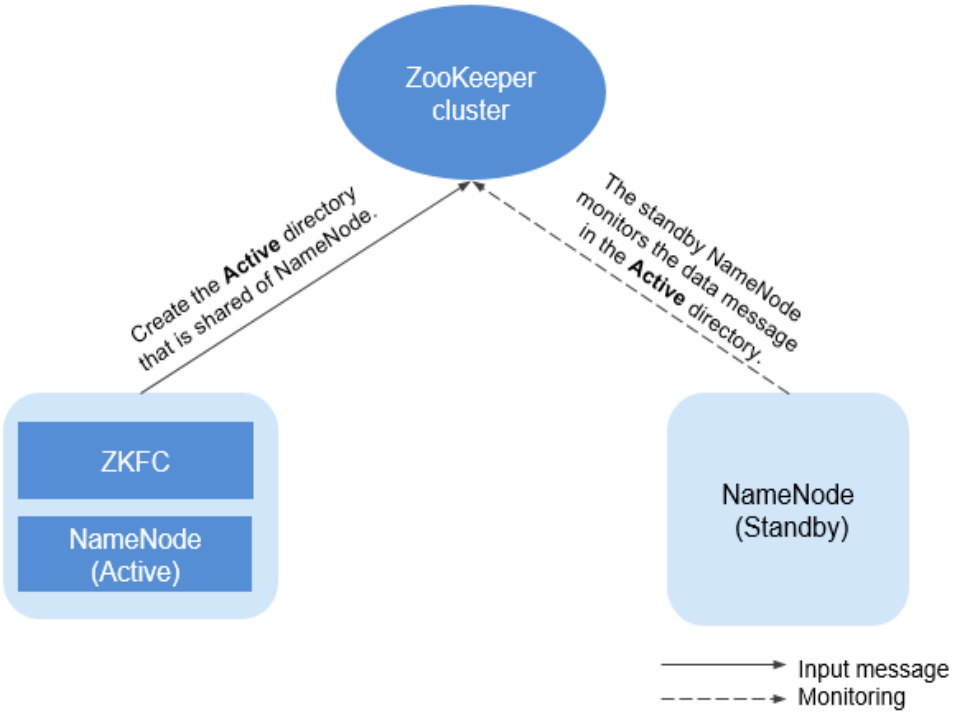
The file writing process is as follows:

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 6-42 shows the relationship between ZooKeeper and HDFS.

Figure 6-42 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.

6.9.4 HDFS Enhanced Open Source Features

Enhanced Open Source Feature: File Block Colocation

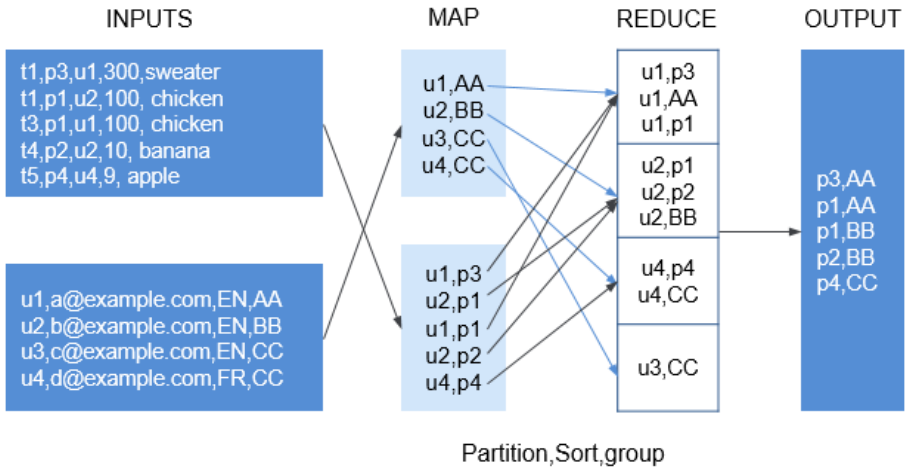
In the offline data summary and statistics scenario, Join is a frequently used computing function, and is implemented in MapReduce as follows:

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 6-43**.

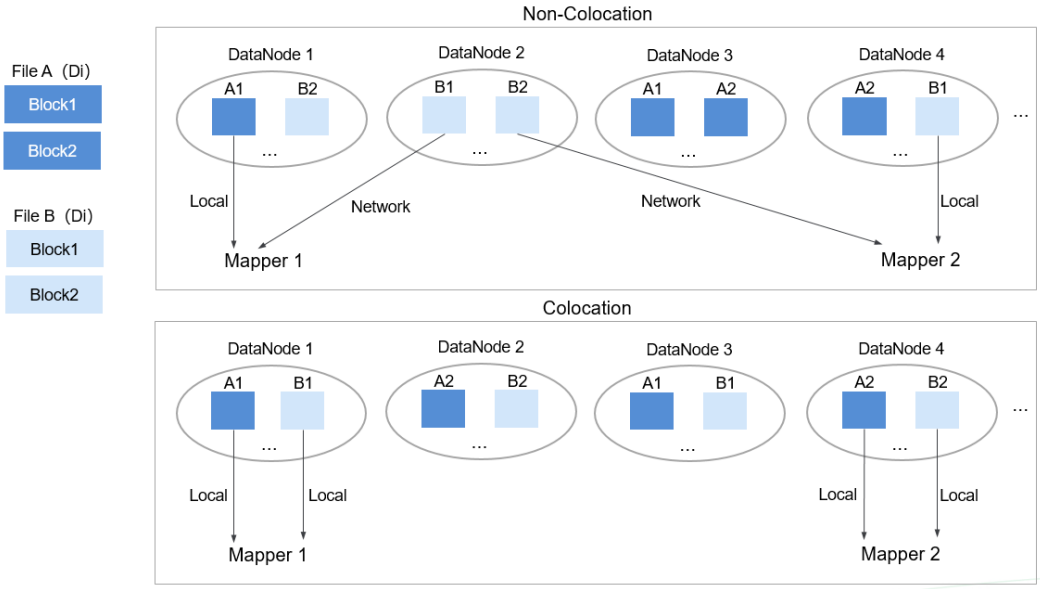
Figure 6-43 Data transmission in the non-colocation scenario



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 6-44 Data block distribution in colocation and non-colocation scenarios

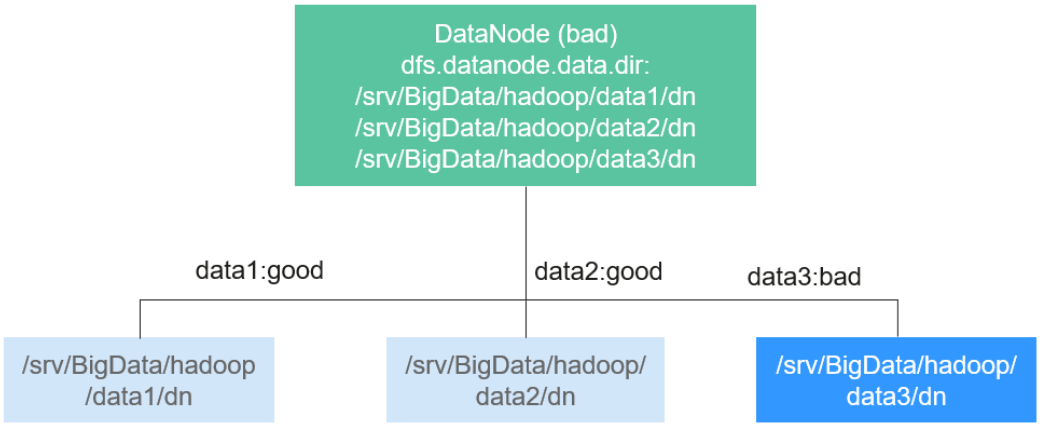


Enhanced Open Source Feature: Damaged Hard Disk Volume Configuration

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.

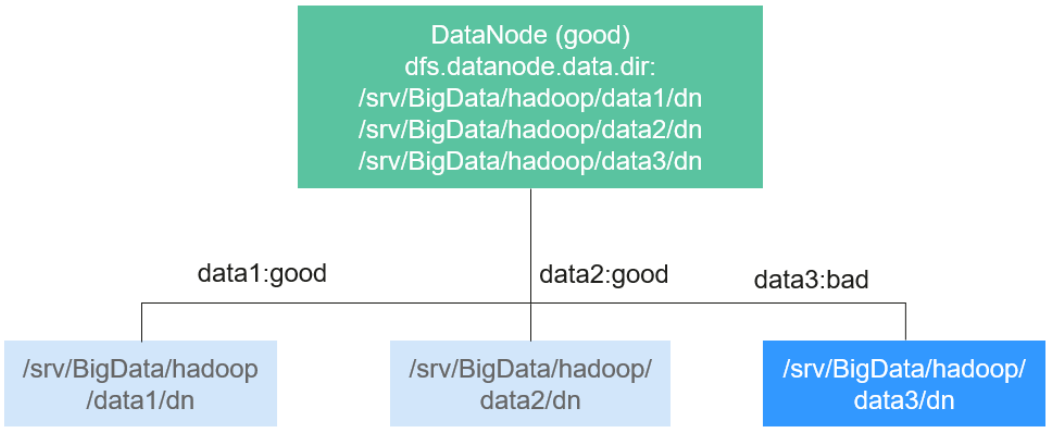
The value of **dfs.datanode.failed.volumes.tolerated** ranges from -1 to the number of disk volumes configured on the DataNode. The default value is **-1**, as shown in **Figure 6-45**.

Figure 6-45 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 6-46**.

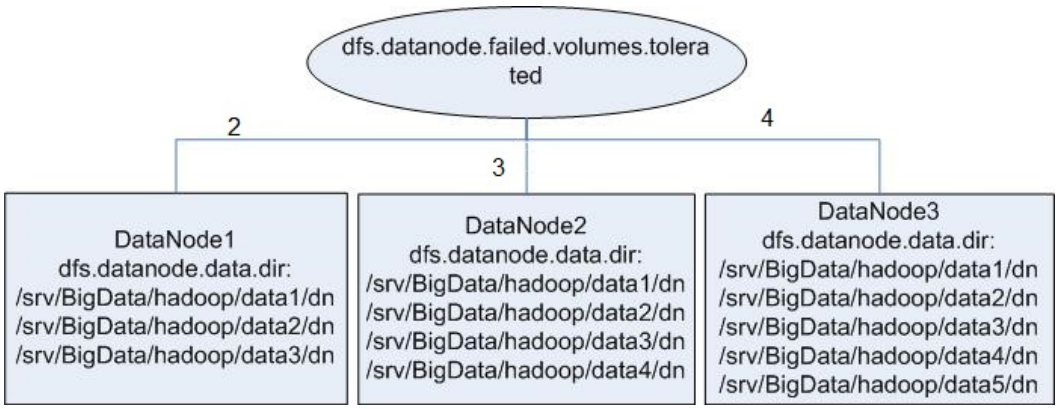
Figure 6-46 Item being set to 1



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 6-47**.

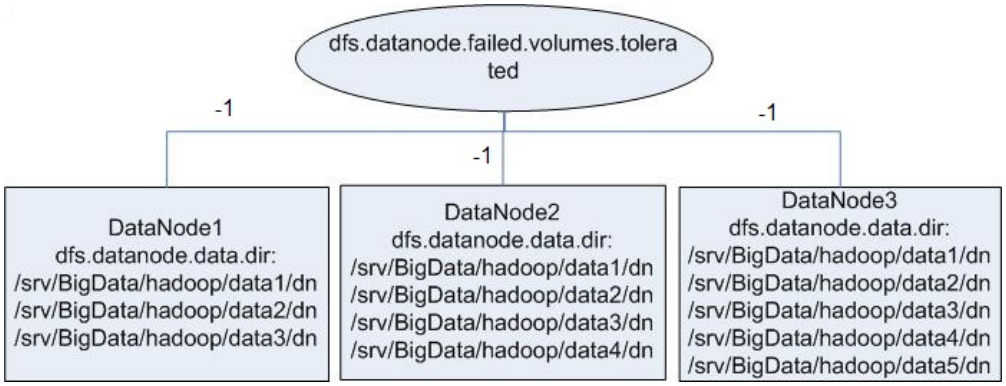
Figure 6-47 Attribute configuration before being enhanced



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 6-48**.

Figure 6-48 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 6-49 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 6-49 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.

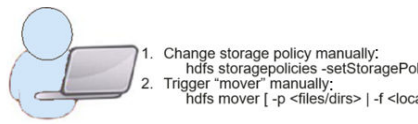
However, as Hadoop is increasingly used for upper-layer applications that demand frequent random I/O access such as Hive and HBase, low latency disks such as solid state disk (SSD) are favored in delay-sensitive scenarios. To cater to the trend, HDFS supports a variety of storage types. Users can choose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

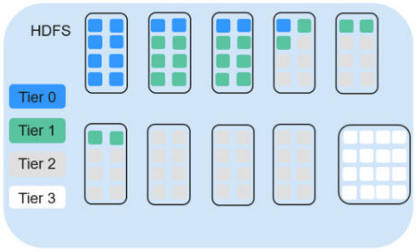
However, low latency disks are far more expensive than spinning disks. Data typically sees heavy initial usage with decline in usage over a period of time. Therefore, it can be useful if data that is no longer used is moved out from expensive disks to cheaper ones storage media.

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 identifying data is frequently or rarely used based on the age rules, setting a data storage policy, triggering the HDFS Auto Data Movement tool to migrate data, and invoking the HDFS cold and hot data migration tool to migrate data across layers.
  - **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.

6.10 HetuEngine

6.10.1 HetuEngine Product Overview

HetuEngine Description

HetuEngine is an in-house 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 6-50** shows the architecture.

Figure 6-50 HetuEngine architecture

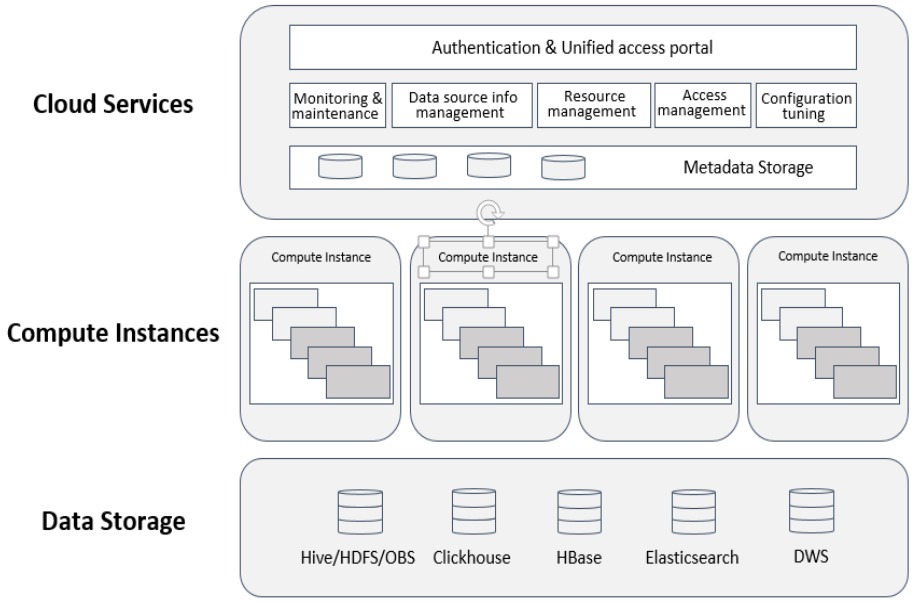


Table 6-10 Module description

Module	Concept	Description
Cloud service layer	HetuEngine CLI/ JDBC	HetuEngine client, through which the query request is submitted and the results is 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 high-performance and secure data transfer across domains (data centers).
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), and ClickHouse) and cross-domain (multiple regions or data

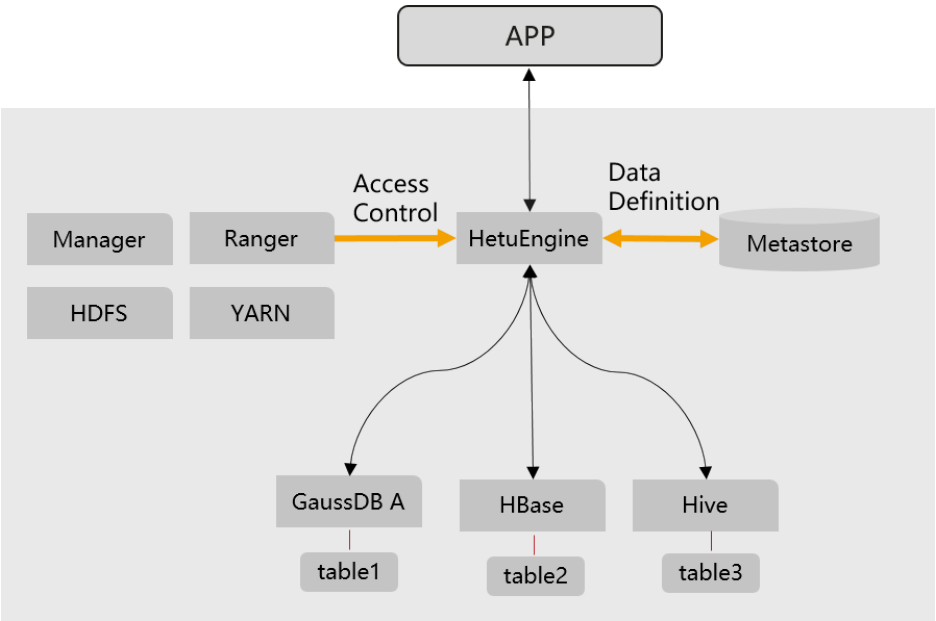
centers) quick joint query, especially for interactive quick query of Hive and Hudi data in the Hadoop cluster (MRS).

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.

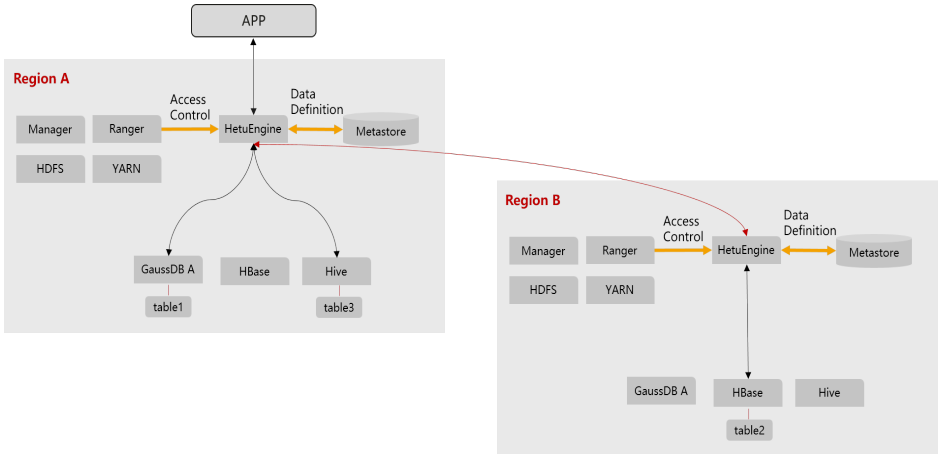
Figure 6-51 HetuEngine cross-source function



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 6-52 HetuEngine cross-region functions



6.10.2 Relationships Between HetuEngine and Other Components

The HetuEngine installation depends on the MRS cluster. **Table 6-11** lists the components on which the HetuServer installation depends.

Table 6-11 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.

6.11 Hive

6.11.1 Hive Basic Principles

Hive is a data warehouse infrastructure built on top of Hadoop. It provides a series of tools that can be used to extract, transform, and load (ETL) data. Hive is a mechanism that can store, query, and analyze mass data stored on Hadoop. Hive defines a simple SQL-like query language, which is known as Hive SQL langage (HQL). It allows a user familiar with SQL to query data. Hive data computing depends on MapReduce, Spark, and Tez.

The new execution engine Tez is used to replace the original MapReduce, significantly improving performance. Tez can convert multiple dependent jobs into one job, so only once HDFS write is required and fewer transit nodes are needed, greatly improving the performance of DAG jobs.

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 system structure:

- User interface: CLI, Client, and web UI CLI is the most frequently-used user interface. A Hive transcript is started when CLI is started. Client refers to a Hive client, and a client user connects to the Hive Server. When entering the client mode, you need to specify the node where the Hive Server resides and start the Hive Server on this node. The web UI is used to access Hive through a browser. MRS can access Hive only in client mode.
- Metadata storage: Hive stores metadata into databases, for example, MySQL and Derby. Metadata in Hive includes a table name, table columns and partitions and their properties, table properties (indicating whether a table is an external table), and the directory where table data is stored.

Hive Framework

Hive is a single-instance service process that provides services by translating HQL into related MapReduce jobs or HDFS operations. **Figure 6-53** shows how Hive is connected to other components.

Figure 6-53 Hive framework

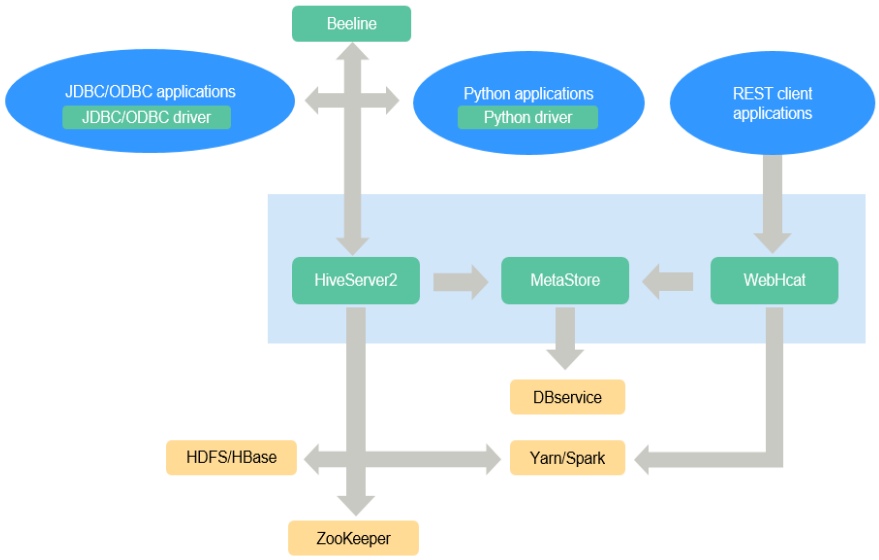


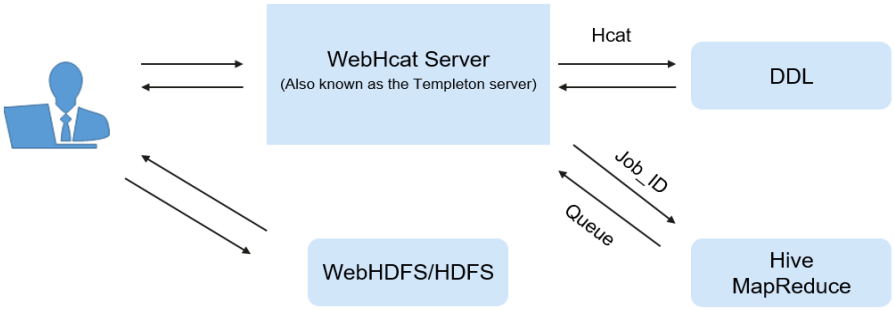
Table 6-12 Module description

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.

Module	Description
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 6-54**.

Figure 6-54 WebHCat logical architecture



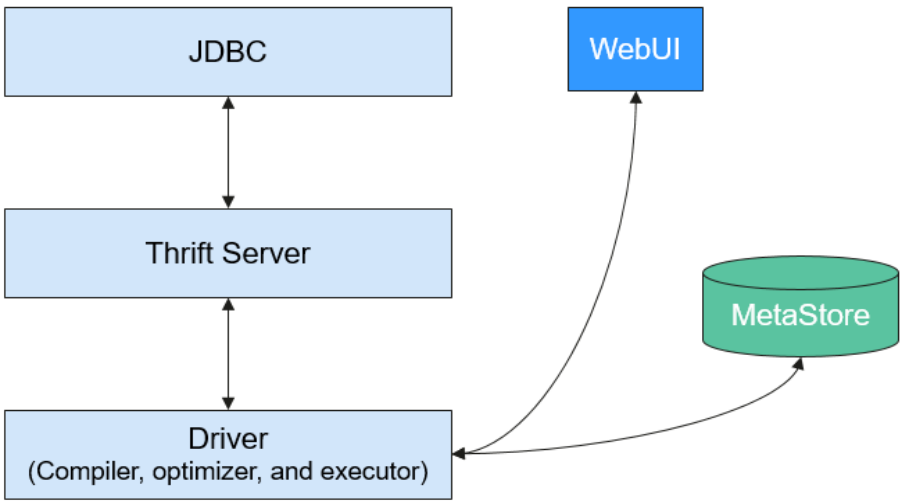
Principles

Hive functions as a data warehouse based on HDFS and MapReduce architecture and translates HQL statements into MapReduce jobs or HDFS operations.

Figure 6-55 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 HQL 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 web UI and JDBC APIs and provides APIs for user access.

Figure 6-55 Hive framework



6.11.2 Hive CBO Principles

Hive CBO Principles

CBO is short for Cost-Based Optimization.

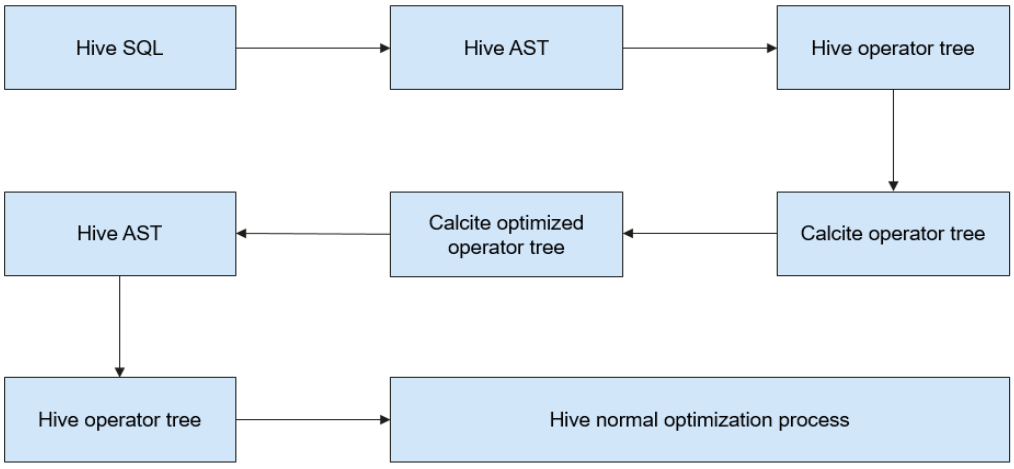
It will optimize the following:

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 6-56** shows the working flow.

Figure 6-56 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';
```

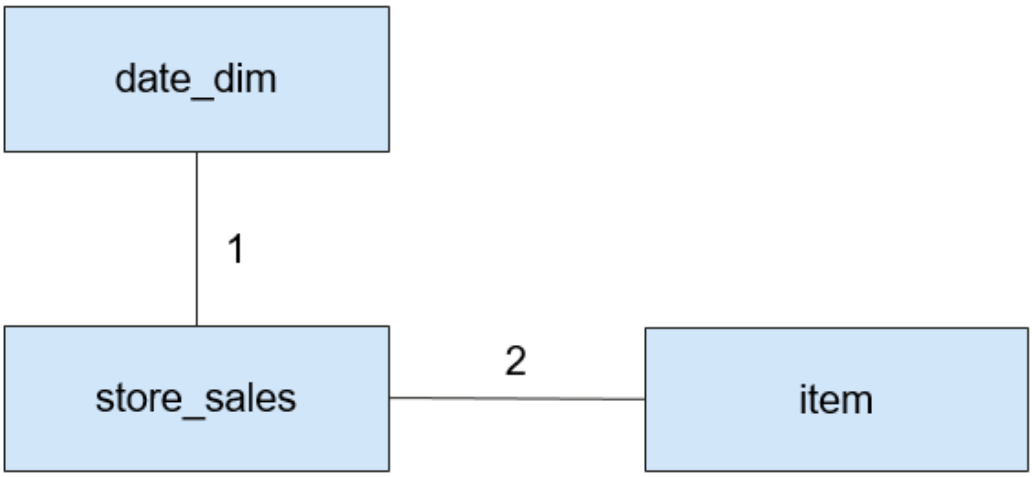
The estimated number of queried data entries is: 1,000,000 x 1/50 = 20,000. The selection rate is 2%.

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 6-57** shows the join relationship.

Figure 6-57 Join relationship



The CBO must first select the tables that bring better 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 6-13**.

Table 6-13 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 6-58 shows the join process when the CBO is disabled.

Figure 6-58 Join process when the CBO is disabled

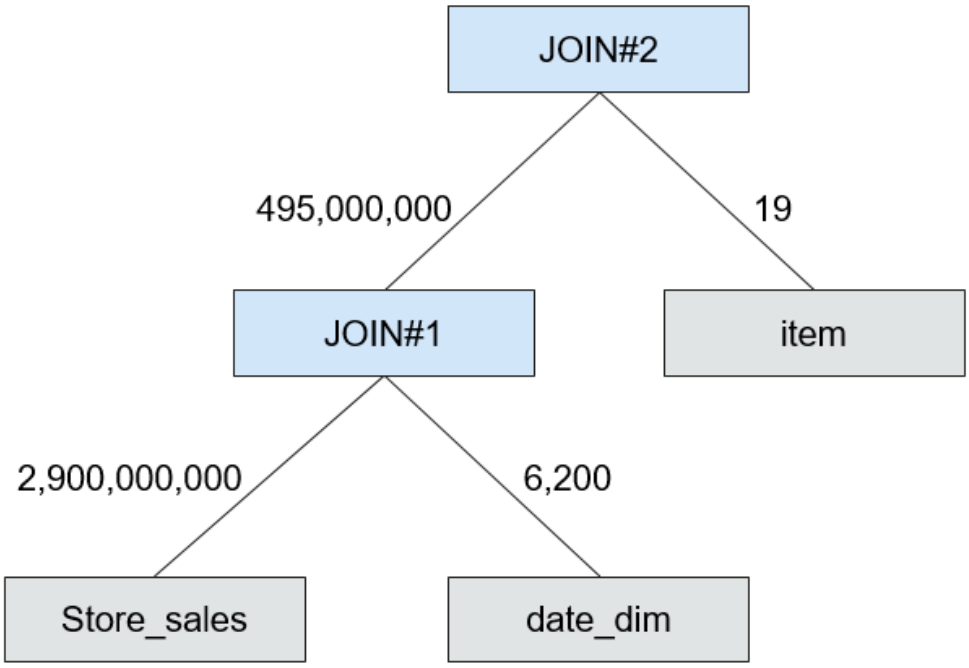
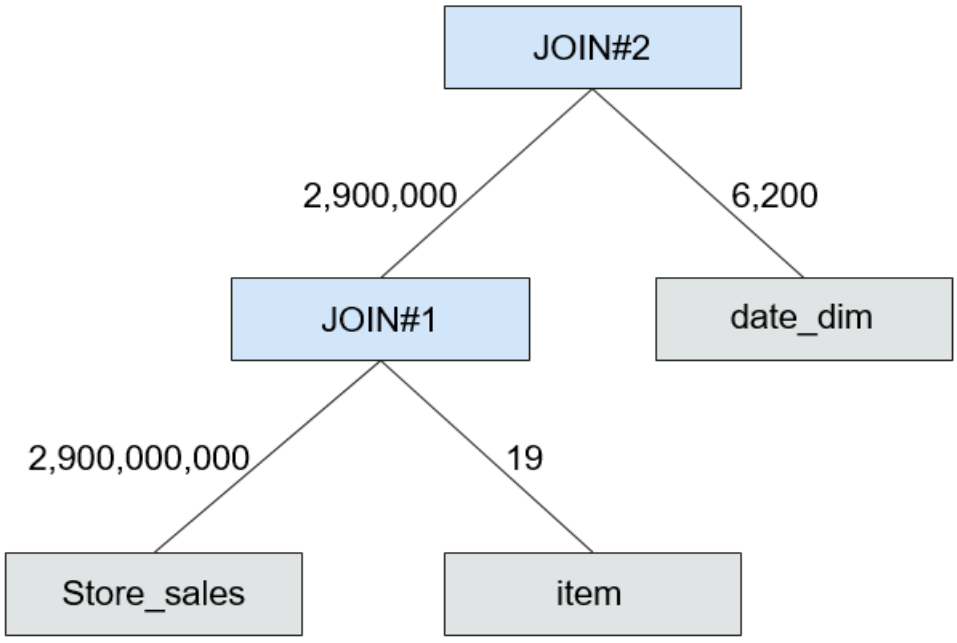


Figure 6-59 shows the join process when the CBO is enabled.

Figure 6-59 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.

6.11.3 Relationships Between Hive and Other Components

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.

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.

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.

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, and the metadata service is provided by the Metadata component.

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.

6.11.4 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**.

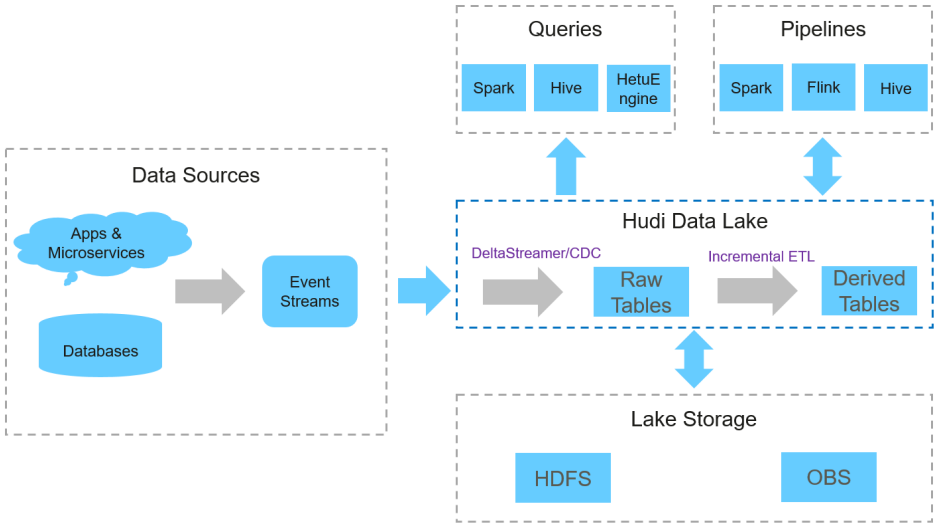
6.12 Hudi

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 Spark2x service has been installed in the MRS cluster.

Figure 6-60 Basic architecture of Hudi



Feature

- 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.

Two Types of Tables Supported by Hudi

- Copy On Write

  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

  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.

Hudi Supporting Three Types Of Views for Read Capabilities 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.

6.13 Hue

6.13.1 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 HSQL statements, the execution fails, because the browser has functional problems. You are advised to use a compatible browser, for example, Google Chrome.
- Impala:
  - 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.
- ZooKeeper: Hue provides the ZooKeeper browser function for you to use ZooKeeper in GUI mode.

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 6-61**.

Figure 6-61 Hue architecture

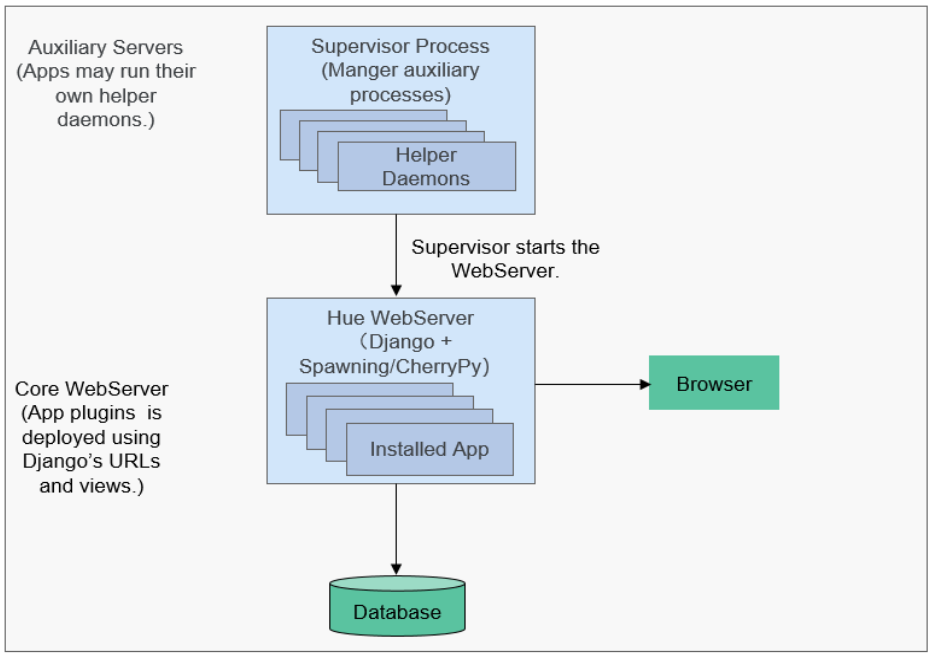


Table 6-14 describes the components shown in **Figure 6-61**.

Table 6-14 Architecture description

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

6.13.2 Relationships Between Hue and Other Components

Relationship Between Hue and Hadoop Clusters

Figure 6-62 shows how Hue interacts with Hadoop clusters.

Figure 6-62 Hue and Hadoop clusters

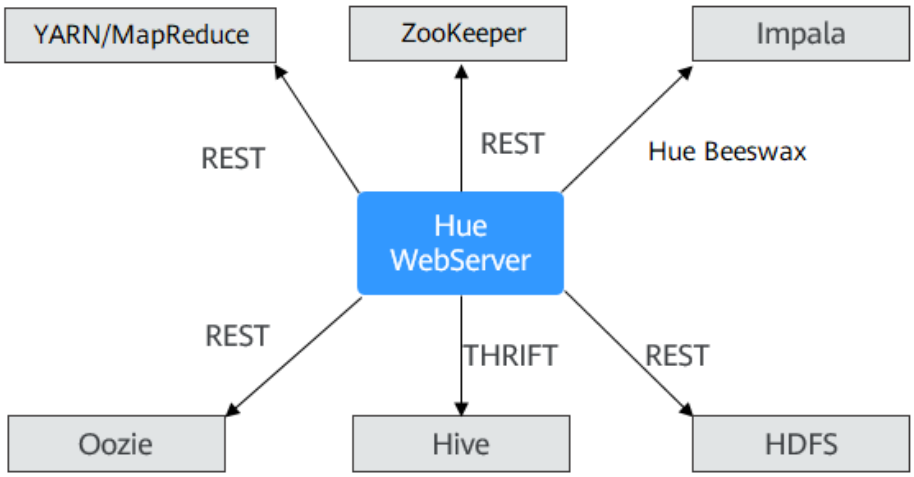


Table 6-15 Relationships between Hue and other components

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.
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.

Connection Name	Description
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.
Impala	Impala provides Hue Beeswax APIs 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 Hue Beeswax APIs and displays execution results on the web UI.

6.13.3 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, and YARN, Hue can work in failover or load balancing mode.

6.14 Impala

Impala

Impala provides fast, interactive SQL queries directly on your Apache Hadoop data stored in HDFS, HBase, or the Object Storage Service (OBS). In addition to using the same unified storage platform, Impala also uses the same metadata, SQL syntax (Hive SQL), ODBC driver, and user interface (Impala query UI in Hue) as Apache Hive. This provides a familiar and unified platform for real-time or batch-oriented queries. Impala is an addition to tools available for querying big data. Impala does not replace the batch processing frameworks built on MapReduce such as Hive. Hive and other frameworks built on MapReduce are best suited for long running batch jobs.

Impala provides the following features:

- Most common SQL-92 features of Hive Query Language (HQL) including SELECT, JOIN, and aggregate functions

- HDFS, HBase, and OBS storage, including:
  - HDFS file formats: delimited text files, Parquet, Avro, SequenceFile, and RCFile
  - Compression codecs: Snappy, GZIP, Deflate, BZIP
- Common data access interfaces including:
  - JDBC driver
  - ODBC driver
  - Hue Beeswax and the Impala query UI
- **Impala-shell** command line interface
- Kerberos authentication

Impala applies to offline analysis (such as log and cluster status analysis) of real-time data queries, large-scale data mining (such as user behavior analysis, interest region analysis, and region display), and other scenarios.

Impala consists of three roles: Impala Daemon (Impalad), Impala StateStore, and Impala Catalog Service.

Impala Daemon

The core Impala component is the Impala daemon, physically represented by the **impalad** process.

A few of the key functions that an Impala daemon performs are:

- Runs on all data nodes.
- Reads and writes to data files.
- Accepts queries transmitted from the **Impala-shell** command, Hue, JDBC, or ODBC.
- Parallelizes the queries and transmits intermediate query results back to the central coordinator.
- Invokes a node to return the query results to the client.

The Impala daemons are in constant communication with StateStore, to confirm which daemons are healthy and can accept new work.

Impala StateStore

The Impala component known as the StateStore checks on the health of all Impala daemons in a cluster, and continuously relays its findings to each of those daemons. It is physically represented by a daemon process named **statestored**. You only need such a process on one host in a cluster. If an Impala daemon goes offline due to hardware failure, network error, software issue, or other reason, the StateStore informs all the other Impala daemons so that future queries can avoid making requests to the unreachable Impala daemon.

Impala Catalog Service

The Impala component known as the Catalog Service relays the metadata changes from Impala SQL statements to all the Impala daemons in a cluster. It is physically represented by a daemon process named **catalogd**. When you create a table, load

data, and so on through Hive, you do need to issue REFRESH or INVALIDATE METADATA on an Impala daemon before executing a query there. The catalog service avoids the need to issue REFRESH and INVALIDATE METADATA statements when the metadata changes are performed by statements issued through Impala.

Relationships with other components

- Hadoop Distributed File System (HDFS)

  Impala uses the HDFS to store files. It parses and processes structured data with highly reliable underlying storage supported by HDFS. Impala does not move data in HDFS and provides faster access.

- Hive

  Impala uses Hive metadata, Open Database Connectivity (ODBC) driver, and SQL syntax. Unlike Hive, which is over MapReduce, Impala implements a distributed architecture based on daemon and handles all query executions on the same node. Therefore, Impala is faster than Hive by reducing the latency caused by MapReduce.

- Kudu

  Kudu is closely integrated with Impala to replace the combination of Impala, HDFS, and Parquet, allowing you to insert, query, update, and delete data from Kudu tablets using Impala's SQL syntax. In addition, you can use JDBC or ODBC. Impala functions as a proxy to connect to Kudu for data operations.

- HBase

  By default, Impala tables use data files stored in HDFS to facilitate batch loading and query in full table scans. However, HBase can provide convenient and efficient query of OLTP-style organization data.

6.15 IoTDB

6.15.1 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.

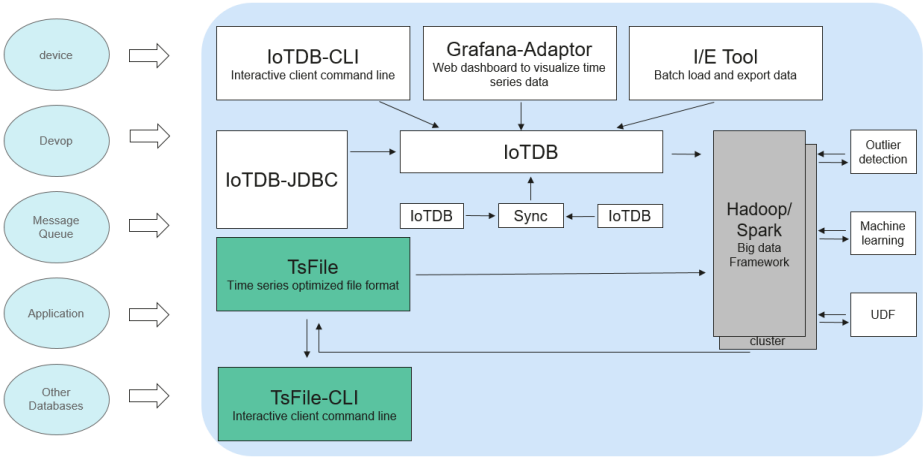
IoTDB sorts time series and stores indexes and chunks, greatly improving the query performance of time series data. IoTDB uses the Raft protocol to ensure data consistency. In time series scenarios, IoTDB pre-computes and stores data to improve analysis performance. Based on the characteristics of time 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 6-63 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 6-63 IoTDB architecture

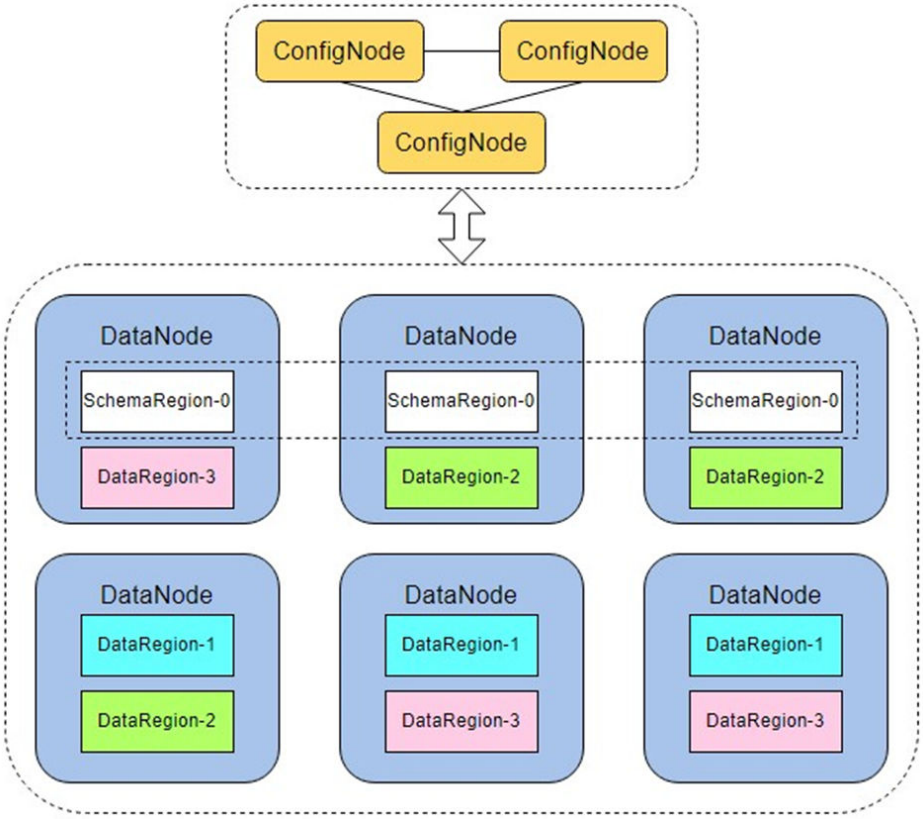


- Users can use Java Database Connectivity (JDBC) or Session to import the time series data and system status data (such as server load, CPU usage and memory usage) collected from device sensors, as well as time series data in message queues, applications, or other databases, to the local or remote IoTDB. Users can also directly write the preceding data into a local TsFile file or a TsFile file in the HDFS.
- You can write TsFile files to HDFS to meet the access requirements of data processing tasks such as Hadoop and Flink.
- 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 writing and viewing data in SQL, script, and graphical forms.

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.

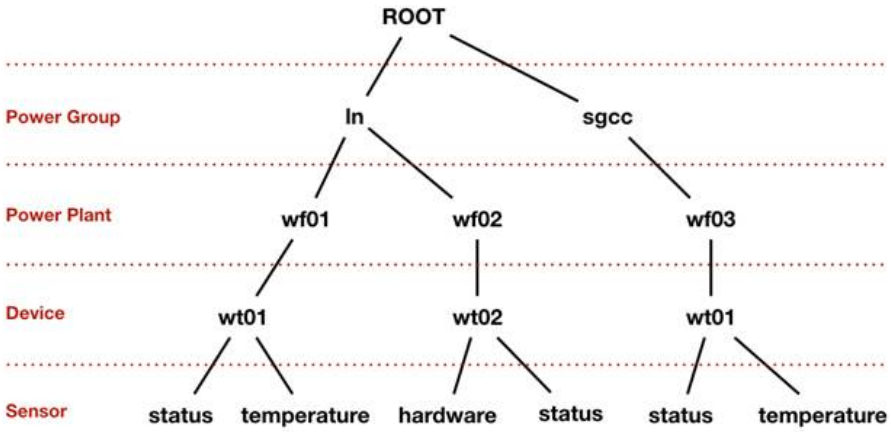
Figure 6-64 IoTDB distributed architecture



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 6-65**. 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 6-65 IoTDB data model



6.15.2 Relationships 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.

6.15.3 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.

6.16 Kafka

6.16.1 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 Structure

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 6-66 Kafka architecture

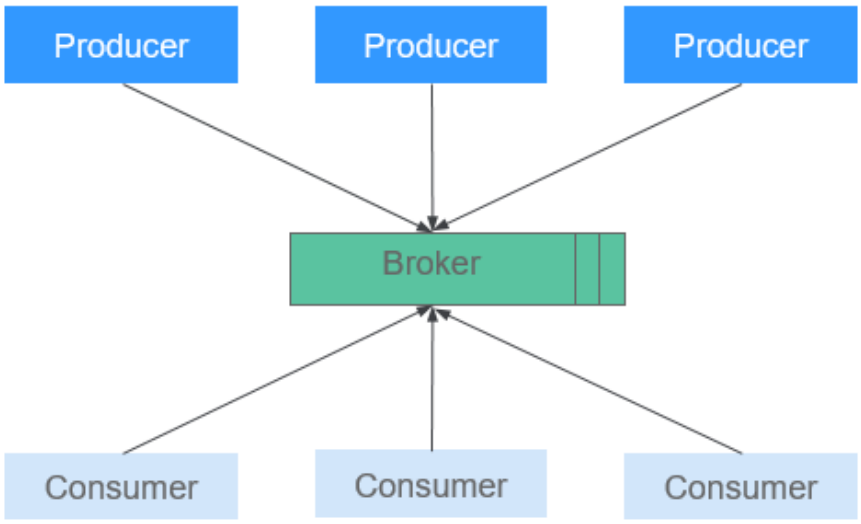


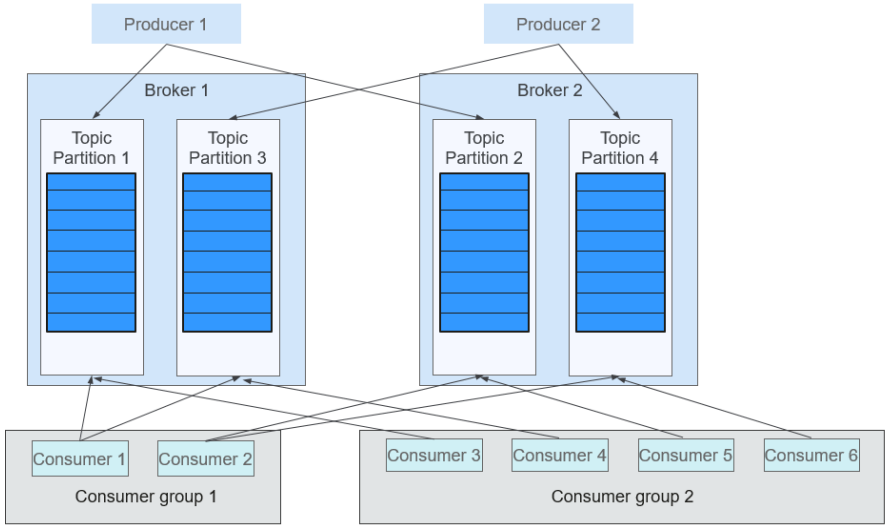
Table 6-16 Kafka architecture description

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.

Name	Description
Consumer	Consumers subscribe to topics and process the feed of published messages.

Figure 6-67 shows the relationships between modules.

Figure 6-67 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.

Principle

- **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. Multiple Producers, Brokers, and Consumers can be deployed in each cluster to form a distributed cluster. The system can be expanded without shutting down the system.

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:

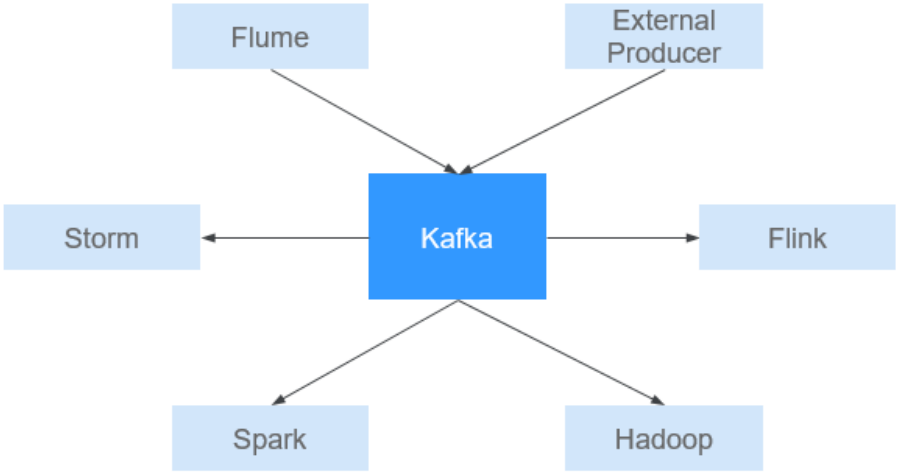
- Check cluster status (topics, consumers, offsets, partitions, replicas, and nodes).
- Redistribute partitions in the cluster.
- Create a topic with optional topic configurations.

- 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.

6.16.2 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 6-68 Relationships with other components



6.16.3 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.

6.17 KafkaManager

KafkaManager is a tool for managing Apache Kafka and provides GUI-based metric monitoring and management of Kafka clusters.

KafkaManager supports the following operations:

- Manage multiple Kafka clusters.
- Easy inspection of cluster states (topics, consumers, offsets, partitions, replicas, and nodes)
- Run preferred replica election.
- Generate partition assignments with option to select brokers to use.
- Run reassignment of partition (based on generated assignments).
- Create a topic with optional topic configurations (Multiple Kafka cluster versions are supported).
- Delete topic on the UI (supported only by clusters of version 0.8.2 with **delete.topic.enable** set to **true**)
- Batch generate partition assignments for multiple topics with option to select brokers to use.
- Batch run reassignment of partitions for multiple topics.
- Add partitions to an existing topic.
- Update configurations for an existing topic.
- Optionally enable JMX polling for broker-level and topic-level metrics.
- Optionally filter out consumers that do not have ids/ owner / & offsets/ directories in ZooKeeper.

6.18 KrbServer and LdapServer

6.18.1 KrbServer and LdapServer Principles

Overview

To manage the access control permissions on data and resources in a cluster, it is recommended that the cluster in security mode be installed. 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 6-69 Security authentication architecture

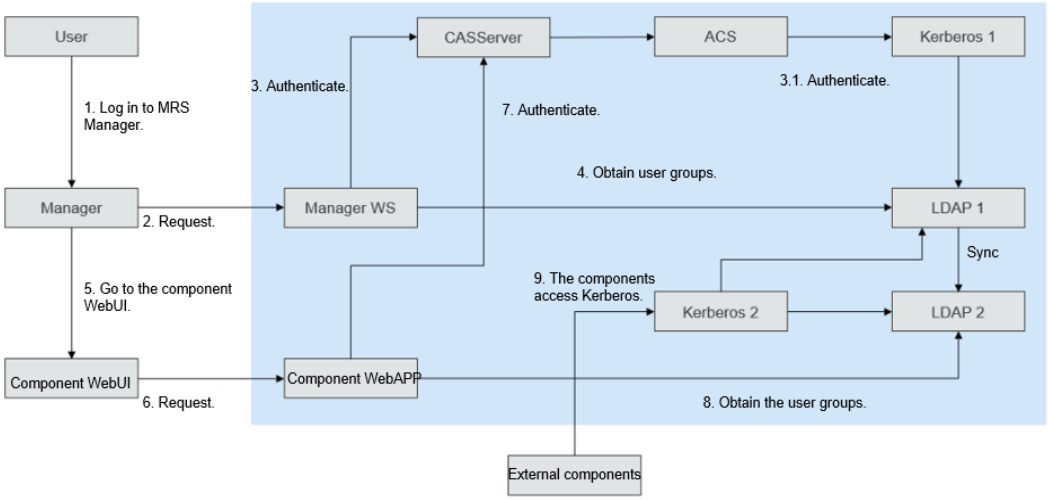


Figure 6-69 includes three scenarios:

- 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 6-17 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
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

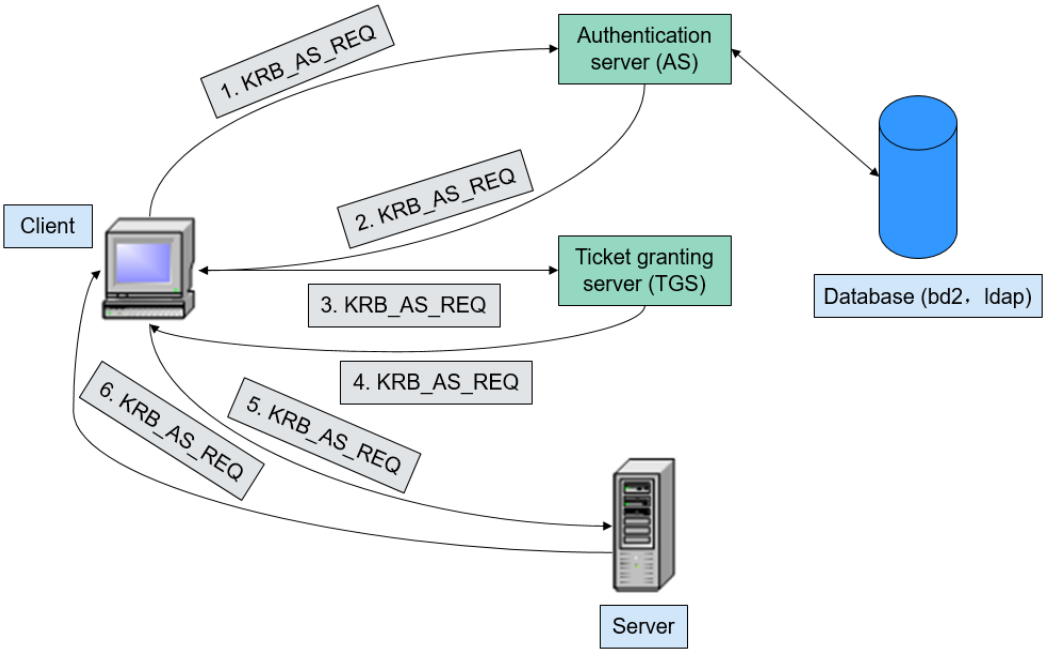
mode. Data write operations can be performed only in the active LDAP1 instance. Data read operations can be performed in LDAP1 or LDAP2.

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

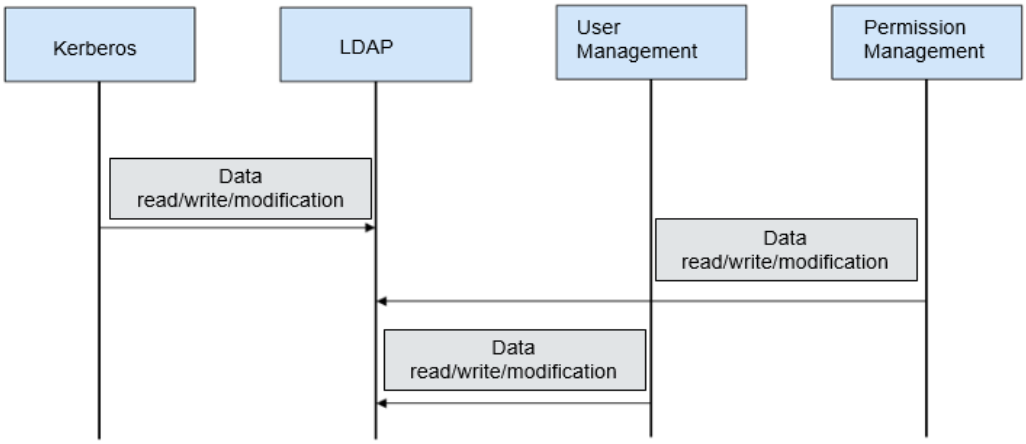
Kerberos authentication

Figure 6-70 Authentication process



LDAP data read and write

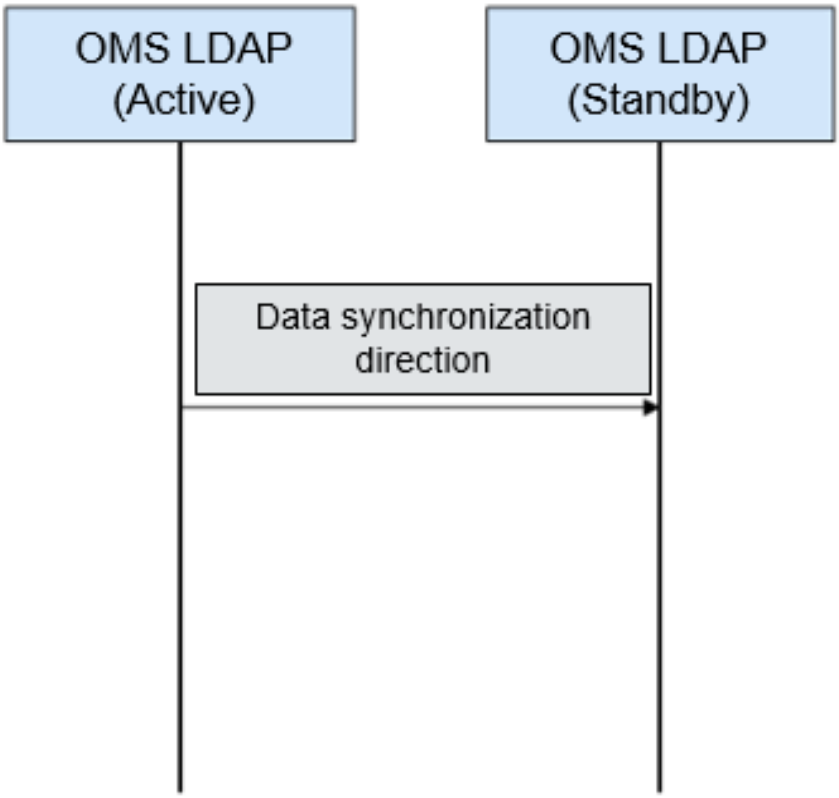
Figure 6-71 Data modification process



LDAP data synchronization

- OMS LDAP data synchronization before cluster installation

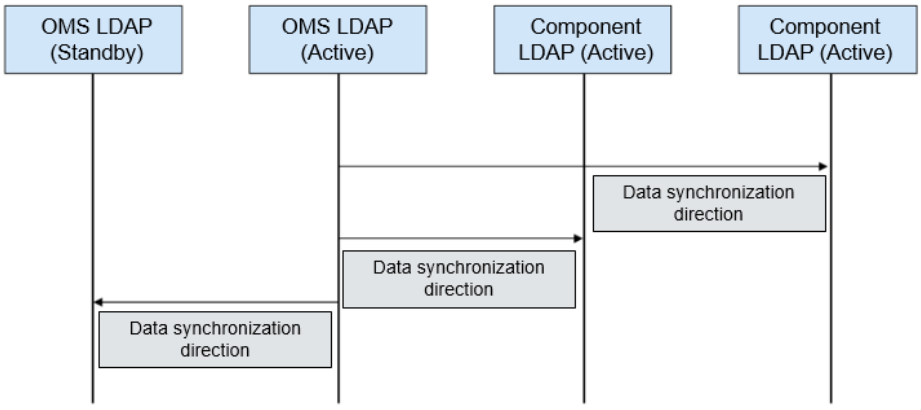
  Figure 6-72 OMS LDAP data synchronization

  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 6-73 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.

6.18.2 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 in **hadoop-common api** includes 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.

6.19 Kudu

Kudu is a column-store manager developed for the Apache Hadoop platform. Kudu shares the common technical properties of Hadoop ecosystem applications, that is, it runs on commodity hardware, which is horizontally scalable, delivering high availability.

Kudu's design has the following benefits:

- Fast processing of OLAP workloads
- Integration with MapReduce, Spark and other Hadoop ecosystem components
- Tight integration with Apache Impala, making it a good, mutable alternative to using HDFS with Apache Parquet
- Strong but flexible consistency model, allowing you to choose consistency requirements on a per-request basis, including the option for strict-serializable consistency

- Strong performance for running sequential and random workloads simultaneously
- Easy to manage
- High availability Tablet Servers and Masters use the Raft Consensus Algorithm, which ensures that as long as more than half the total number of replicas is available, the tablet is available for reads and writes. For example, if 2 out of 3 replicas or 3 out of 5 replicas are available, the tablet is available. Reads can be serviced by read-only follower tablets, even in the event of a leader tablet failure.
- Structured data model

By combining all of these properties, Kudu targets support for families of applications that are difficult or impossible to implement on current generation Hadoop storage technologies.

A few examples of applications for which Kudu is a great solution are:

- Reporting applications where newly-arrived data needs to be immediately available for end users
- Time-series applications that must simultaneously support queries across large amounts of historic data and granular queries about an individual entity that must return very quickly
- Applications that use predictive models to make real-time decisions with periodic refreshes of the predictive model based on all historic data

Relationships with other components

- HBase

  Kudu is designed based on the HBase structure and can implement fast random read/write and update functions that HBase features.

  The differences are as follows:

  – HBase uses ZooKeeper to ensure data consistency, whereas Kudu uses the Raft consensus algorithm to ensure consistency.

  – HBase uses HDFS for resilient data storage, whereas Kudu uses TServer to ensure strong data consistency and reliability.

6.20 Loader

6.20.1 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 6-74**.

Figure 6-74 Loader model

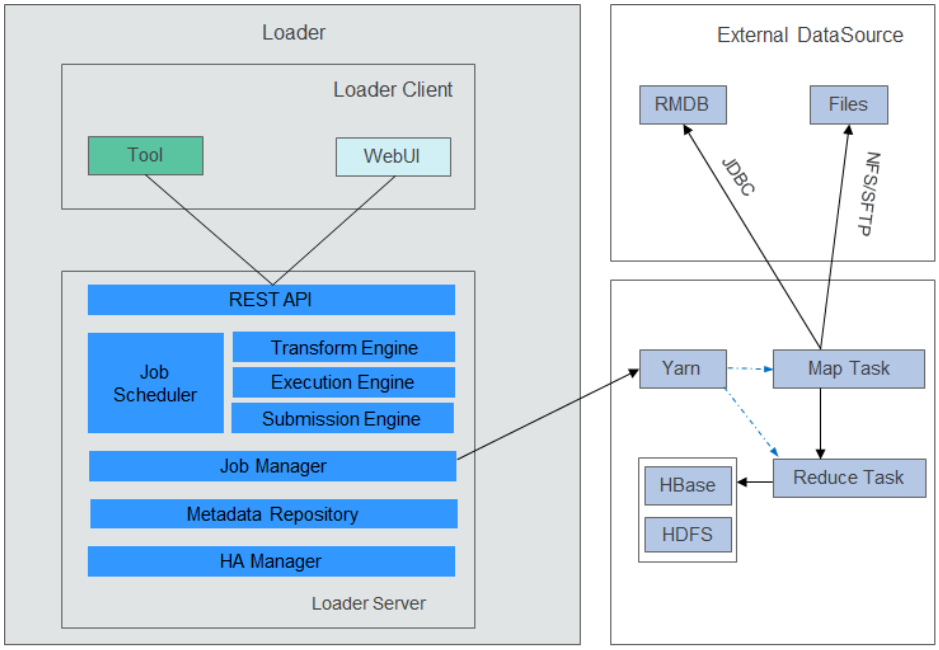


Table 6-18 describes the functions of each module shown in the preceding figure.

Table 6-18 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.

Module	Description
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.

6.20.2 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.

6.20.3 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.

6.21 Manager

6.21.1 Manager Basic Principles

Overview

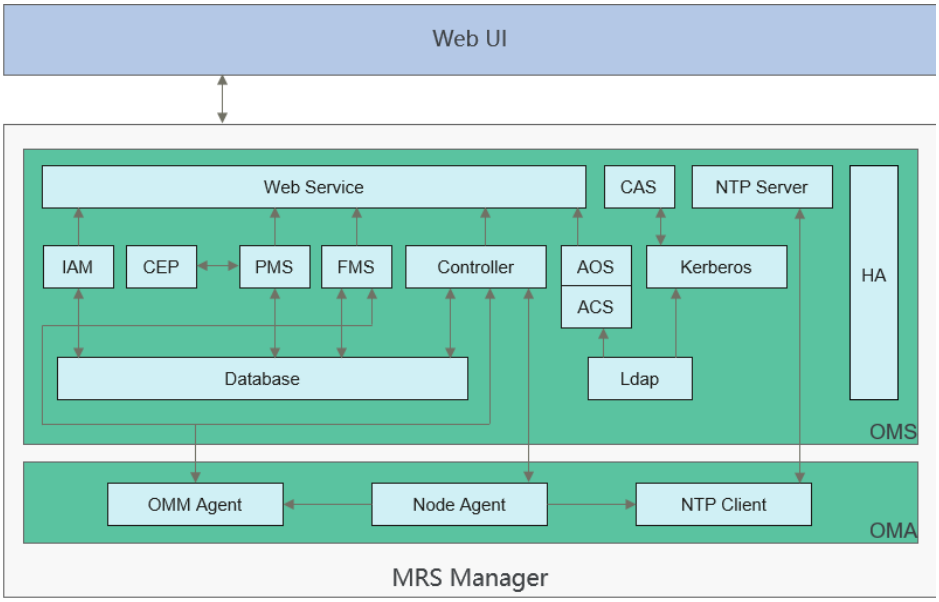
Manager is the O&M management system of MRS and provides unified cluster management capabilities for services deployed in clusters.

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

Architecture

Figure 6-75 shows the overall logical architecture of FusionInsight Manager.

Figure 6-75 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 6-19 describes the modules shown in **Figure 6-75**.

Table 6-19 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.
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.

Module	Description
PMS	The performance monitoring module. It collects the performance monitoring data on each OMA and provides the query function.
CEP	Convergence function module. For example, the used disk space of all OMAs is collected as a performance indicator.
FMS	Alarm module. It collects and queries alarms on each OMA.
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.
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.

6.21.2 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. (Only MRS 3.x and later versions support one-click switching between Chinese and English.)

6.22 MapReduce

6.22.1 MapReduce Basic Principles

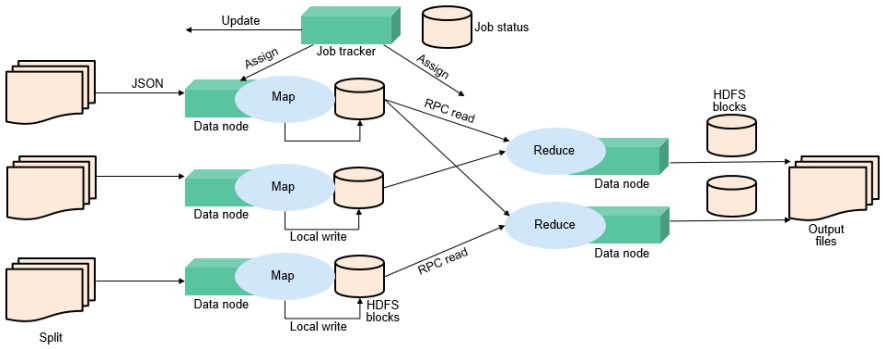
NOTE

To use MapReduce, ensure that the Hadoop service has been installed in the MRS cluster.

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 ideas are derived from functional programming languages and vector programming languages.

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 6-76 Distributed batch processing engine



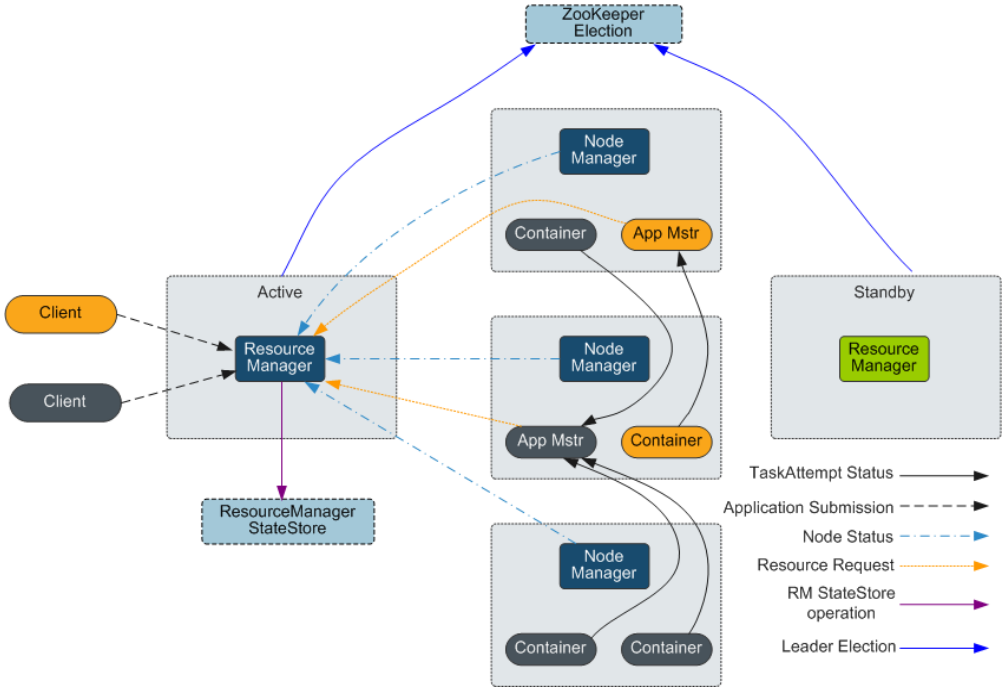
MapReduce is a software framework for processing large datasets in parallel. The root of MapReduce is the Map and Reduce functions in functional programming. The Map function accepts a group of data and transforms it into a key-value pair list. Each element in the input domain corresponds to a key-value 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 structure

As shown in **Figure 6-77**, MapReduce is integrated into YARN through the Client and ApplicationMaster interfaces of YARN, and uses YARN to apply for computing resources.

Figure 6-77 Basic architecture of Apache YARN and MapReduce



6.22.2 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. HDFS is most commonly used. MapReduce uses the high throughput performance of HDFS to read a large amount of data for computing and the data is stored in HDFS.

Relationship Between MapReduce and Yarn

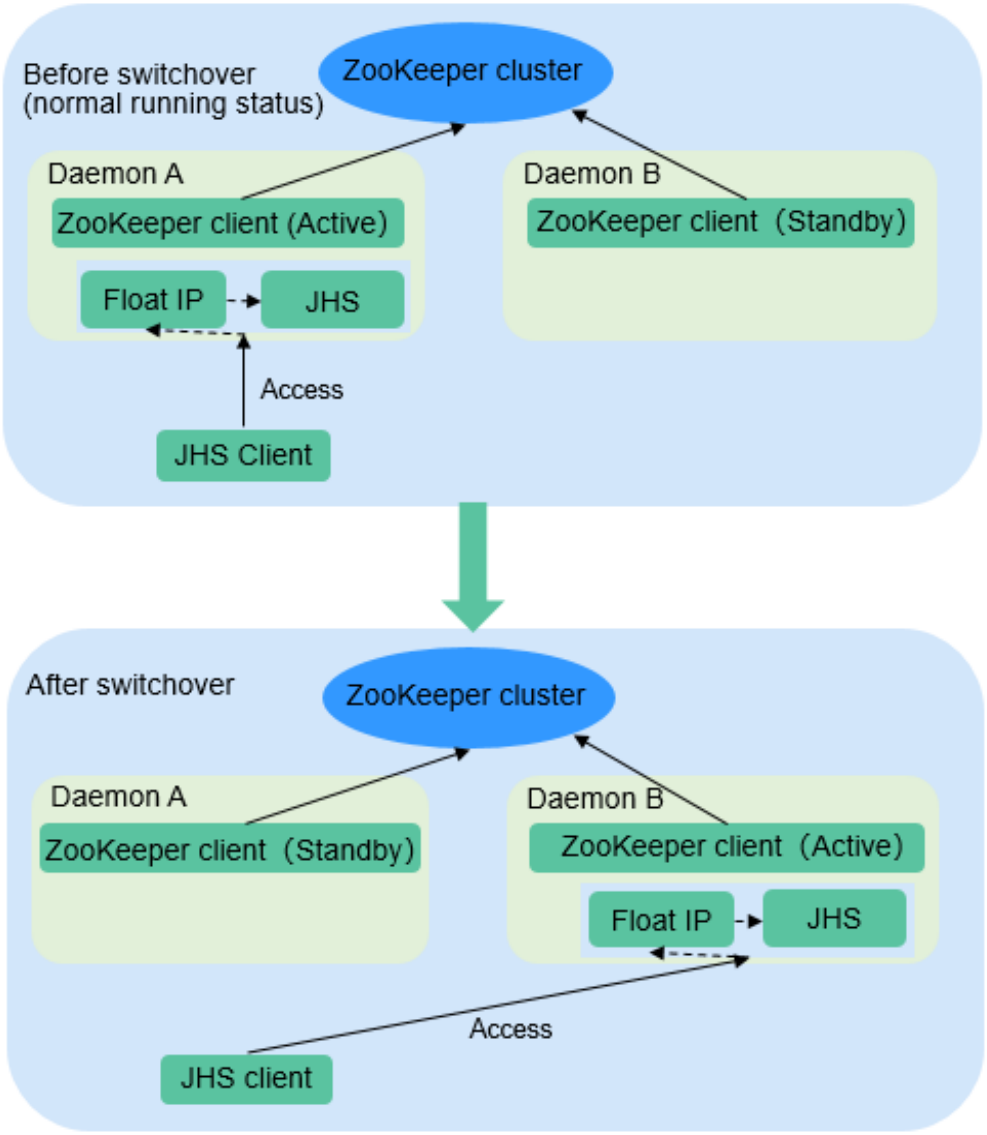
MapReduce is a computing framework running on Yarn, which is used for batch processing. MRv1 is implemented based on MapReduce in Hadoop 1.0, which is composed of programming models (new and old programming APIs), running environment (JobTracker and TaskTracker), and data processing engine (MapTask and 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.

6.22.3 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 6-78 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 6-79 MapReduce job

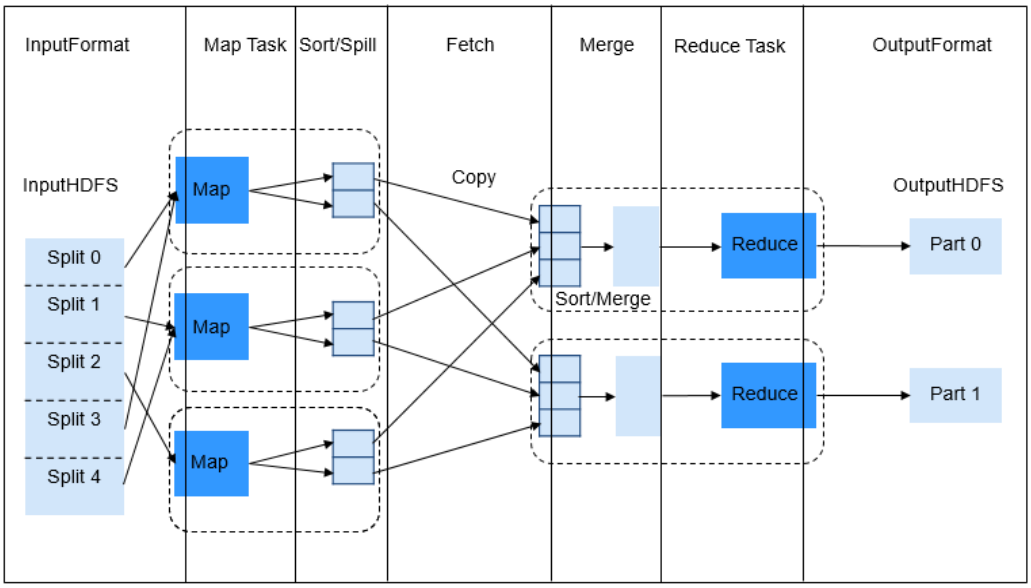
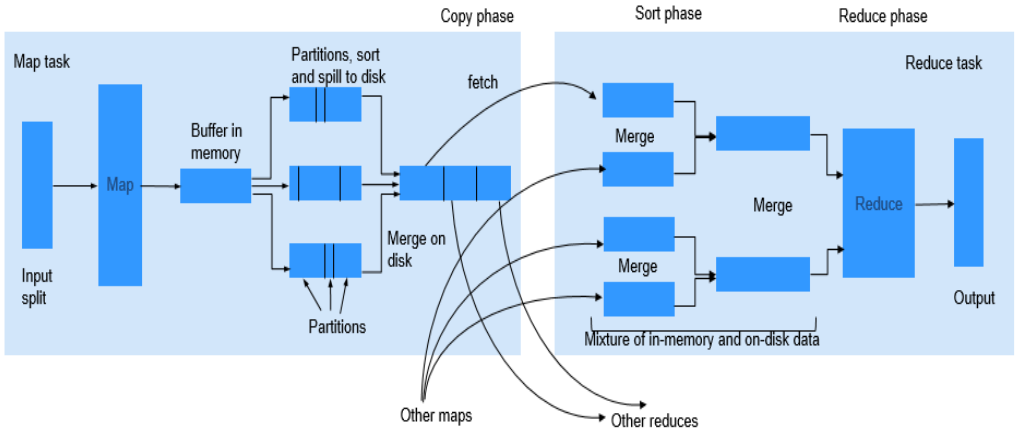


Figure 6-80 Job execution process



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 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 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 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 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.

6.23 Oozie

6.23.1 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 6-81**.

Figure 6-81 Oozie architecture

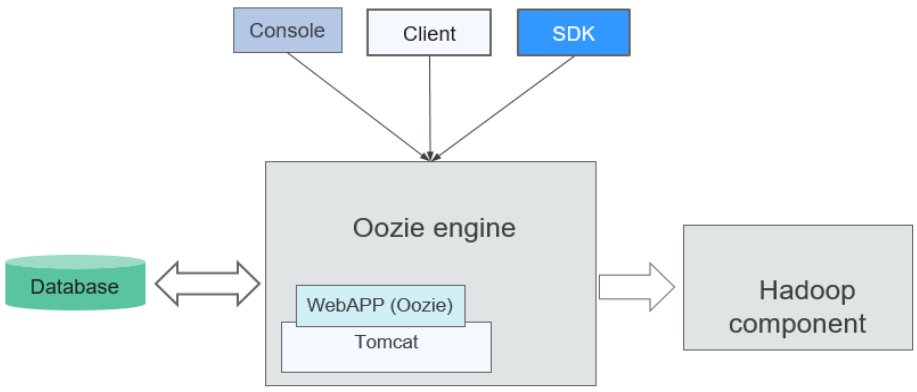


Table 6-20 describes the functions of each module shown in **Figure 6-81**.

Table 6-20 Architecture description

Connection Name	Description
Console	Allows users to view and monitor Oozie workflows.

Connection Name	Description
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.
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.

6.23.2 Oozie Enhanced Open Source Features

Enhanced Open Source Feature: Improved Security

Provides roles of administrator and common users to support Oozie permission management.

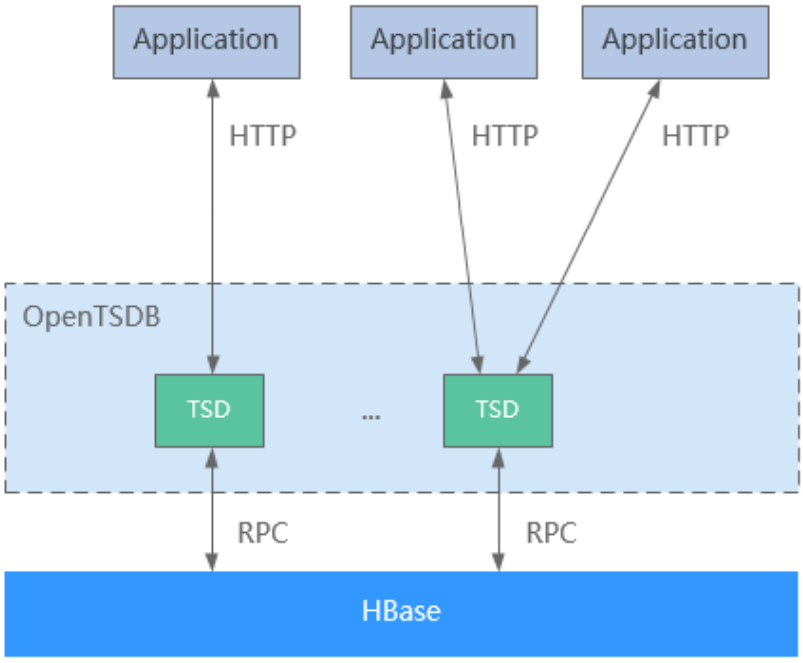
Supports single sign-on and sign-out, HTTPS access, and audit logs.

6.24 OpenTSDB

OpenTSDB is a distributed, scalable time series database based on HBase. OpenTSDB is designed to collect monitoring information of a large-scale cluster and implement second-level data query, eliminating the limitations of querying and storing massive amounts of monitoring data in common databases.

OpenTSDB consists of a Time Series Daemon (TSD) as well as a set of command line utilities. Interaction with OpenTSDB is primarily implemented by running one or more TSDs. Each TSD is independent. There is no master server and no shared state, so you can run as many TSDs as required to handle any load you throw at it. Each TSD uses HBase in a CloudTable cluster to store and retrieve time series data. The data schema is highly optimized for fast aggregations of similar time series to minimize storage space. TSD users never need to directly access the underlying storage. You can communicate with the TSD through an HTTP API. All communications happen on the same port (the TSD figures out the protocol of the client by looking at the first few bytes it receives).

Figure 6-82 OpenTSDB architecture



Application scenarios of OpenTSDB have the following features:

- The collected metrics have a unique value at a time point and do not have a complex structure or relationship.
- Monitoring metrics change with time.
- Like HBase, OpenTSDB features high throughput and good scalability.

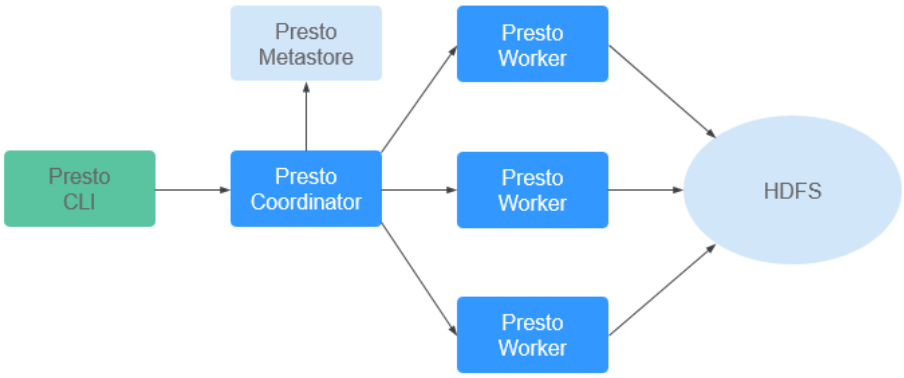
OpenTSDB provides an HTTP based application programming interface to enable integration with external systems. Almost all OpenTSDB features are accessible via the API such as querying time series data, managing metadata, and storing data points.

6.25 Presto

Presto is an open source SQL query engine for running interactive analytic queries against data sources of all sizes. It applies to massive structured/semi-structured data analysis, massive multi-dimensional data aggregation/report, ETL, ad-hoc queries, and more scenarios.

Presto allows querying data where it lives, including HDFS, Hive, HBase, Cassandra, relational databases or even proprietary data stores. A Presto query can combine different data sources to perform data analysis across the data sources.

Figure 6-83 Presto architecture



Presto runs in a cluster in distributed mode and contains one coordinator and multiple worker processes. Query requests are submitted from clients (for example, CLI) to the coordinator. The coordinator parses SQL statements, generates execution plans, and distributes the plans to multiple worker processes for execution.

Multiple Presto Instances

MRS supports the installation of multiple Presto instances for a large-scale cluster by default. That is, multiple Worker instances, such as Worker1, Worker2, and Worker3, are installed on a Core/Task node. Multiple Worker instances interact with the Coordinator to execute computing tasks, greatly improving node resource utilization and computing efficiency.

Presto multi-instance applies only to the Arm architecture. Currently, a single node supports a maximum of four instances.

6.26 Ranger

6.26.1 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 Storm, 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 6-84 shows the Ranger architecture.

Figure 6-84 Ranger structure

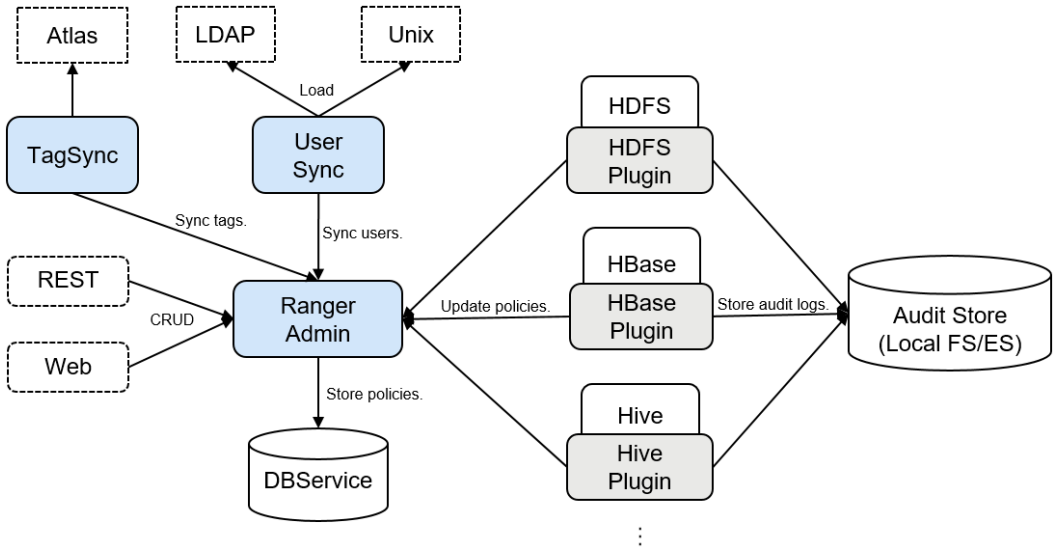


Table 6-21 Architecture description

Connection Name	Description
RangerAdmin	Provides a WebUI and RESTful API 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.

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.

- 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.

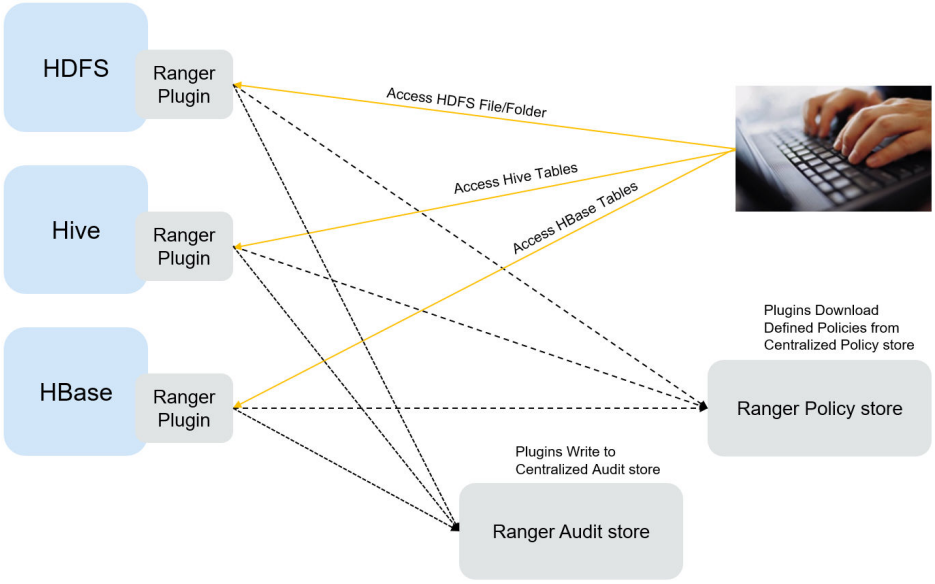
- High performance

  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.

6.26.2 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, Storm, and Spark2x. More components will be supported in the future.

Figure 6-85 Relationships between Ranger and other components



6.27 Spark

6.27.1 Spark Basic Principles

NOTE

The Spark component applies to versions earlier than MRS 3.x.

Description

Spark is an open source parallel data processing framework. It helps you easily develop unified big data applications and perform offline processing, stream processing, and interactive analysis on data.

Spark provides a framework featuring fast computing, write, and interactive query. Spark has obvious advantages over Hadoop in terms of performance. Spark uses the in-memory computing mode to avoid I/O bottlenecks in scenarios where multiple tasks in a MapReduce workflow process the same dataset. Spark is implemented by using Scala programming language. Scala enables distributed datasets to be processed in a method that is the same as that of processing local data. In addition to interactive data analysis, Spark supports interactive data mining. Spark adopts in-memory computing, which facilitates iterative computing. By coincidence, iterative computing of the same data is a general problem facing data mining. In addition, Spark can run in Yarn clusters where Hadoop 2.0 is installed. The reason why Spark cannot only retain various features like MapReduce fault tolerance, data localization, and scalability but also ensure high performance and avoid busy disk I/Os is that a memory abstraction structure called Resilient Distributed Dataset (RDD) is created for Spark.

Original distributed memory abstraction, for example, key-value store and databases, supports small-granularity update of variable status. This requires backup of data or log updates to ensure fault tolerance. Consequently, a large

amount of I/O consumption is brought about to data-intensive workflows. For the RDD, it has only one set of restricted APIs and only supports large-granularity update, for example, map and join. In this way, Spark only needs to record the transformation operation logs generated during data establishment to ensure fault tolerance without recording a complete dataset. This data transformation link record is a source for tracing a data set. Generally, parallel applications apply the same computing process for a large dataset. Therefore, the limit to the mentioned large-granularity update is not large. As described in Spark theses, the RDD can function as multiple different computing frameworks, for example, programming models of MapReduce and Pregel. In addition, Spark allows you to explicitly make a data transformation process be persistent on hard disks. Data localization is implemented by allowing you to control data partitions based on the key value of each record. (An obvious advantage of this method is that two copies of data to be associated will be hashed in the same mode.) If memory usage exceeds the physical limit, Spark writes relatively large partitions into hard disks, thereby ensuring scalability.

Spark has the following features:

- Fast: The data processing speed of Spark is 10 to 100 times higher than that of MapReduce.
- Easy-to-use: Java, Scala, and Python can be used to simply and quickly compile parallel applications for processing massive amounts of data. Spark provides over 80 operators to help you compile parallel applications.
- Universal: Spark provides many tools, for example, **Spark SQL** and **Spark Streaming**. These tools can be combined flexibly in an application.
- Integration with Hadoop: Spark can directly run in a Hadoop cluster and read existing Hadoop data.

The Spark component of MRS has the following advantages:

- The Spark Streaming component of MRS supports real-time data processing rather than triggering as scheduled.
- The Spark component of MRS provides Structured Streaming and allows you to build streaming applications using the Dataset API. Spark supports exactly-once semantics and inner and outer joins for streams.
- The Spark component of MRS uses **pandas_udf** to replace the original user-defined functions (UDFs) in PySpark to process data, which reduces the processing duration by 60% to 90% (affected by specific operations).
- The Spark component of MRS also supports graph data processing and allows modeling using graphs during graph computing.
- Spark SQL of MRS is compatible with some Hive syntax (based on the 64 SQL statements of the Hive-Test-benchmark test set) and standard SQL syntax (based on the 99 SQL statements of the TPC-DS test set).

Architecture

Figure 6-86 describes the Spark architecture and **Table 6-22** lists the Spark modules.

Figure 6-86 Spark architecture

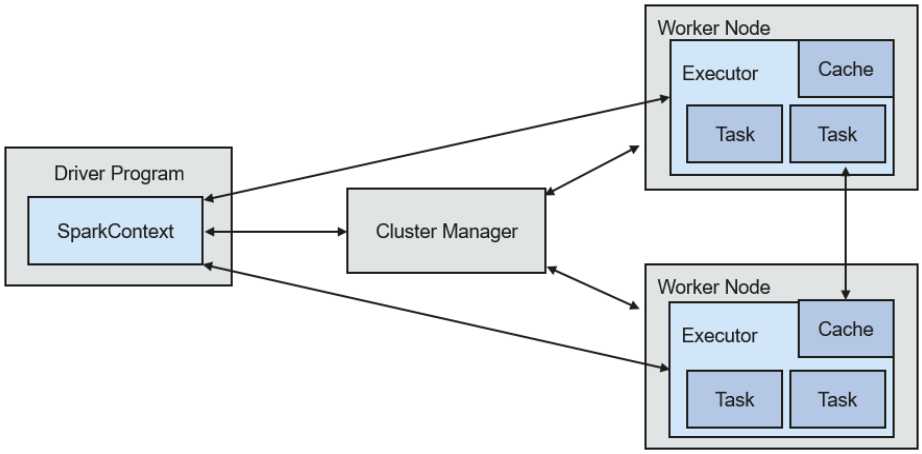


Table 6-22 Basic concepts

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.
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).

Module	Description
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 Application Running Principle

Figure 6-87 shows the Spark application running architecture. The running process is as follows:

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 6-87 Spark application running architecture

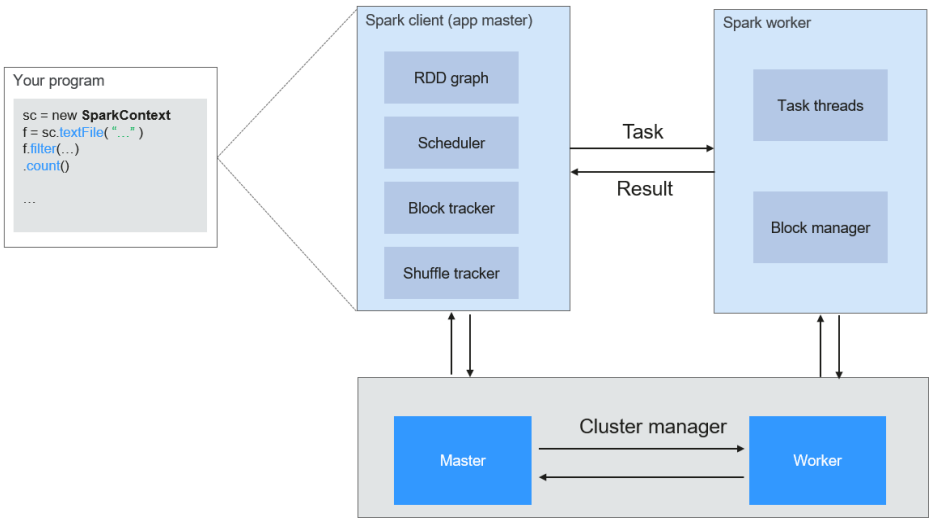
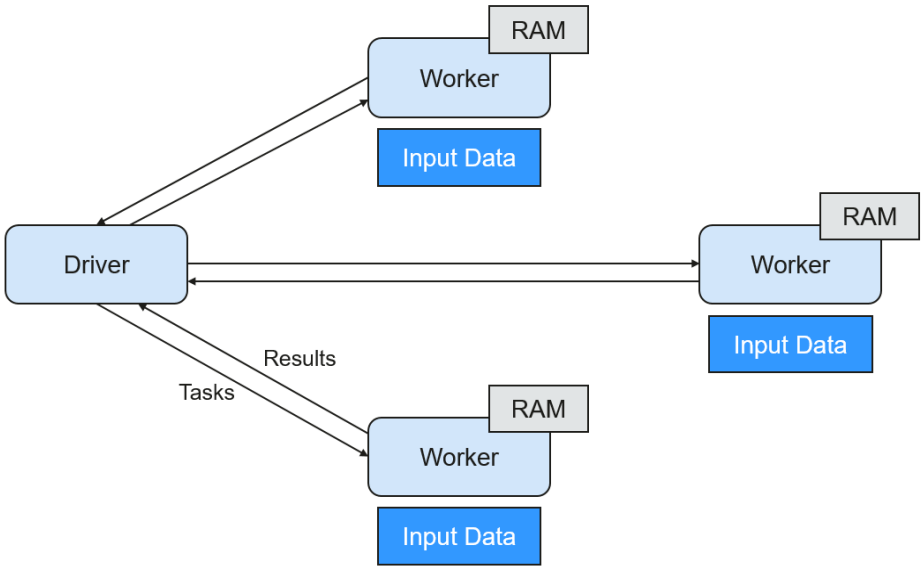


Figure 6-88 shows the Master and Worker modes adopted by Spark. A user submits an application on the Spark client, and then the scheduler divides a job into multiple tasks and sends the tasks to each Worker for execution. Each Worker reports the computation results to Driver (Master), and then the Driver aggregates and returns the results to the client.

Figure 6-88 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. Whether in terms of scheduling or task running on executors. Each driver independently schedules its own tasks. 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, the driver is started in the client process, and the ApplicationMaster process is used only to apply for resources from the Yarn cluster.

Spark Streaming Principle

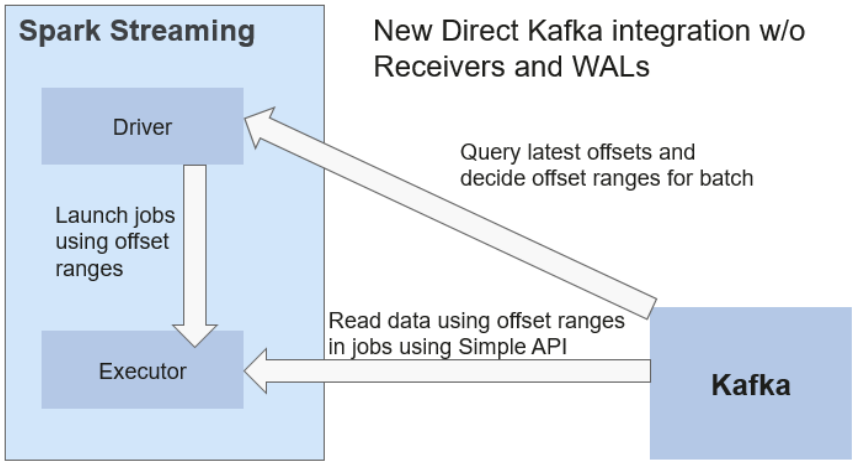
Spark Streaming is a real-time computing framework built on the Spark, which expands the capability for processing massive streaming data. Currently, Spark supports the following data processing methods:

- Direct Streaming

  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 6-89 Data transmission through Direct Kafka API



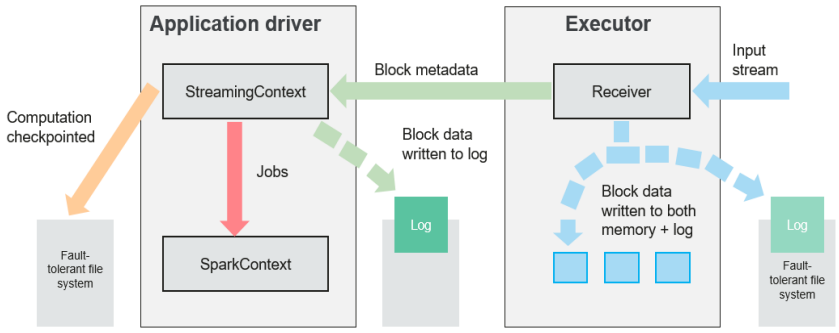
After the failure, Spark Streaming can read data from Kafka again and process the data segment. The processing result is the same no matter Spark Streaming fails or not, because the semantic is processed only once.

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

  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 6-90** shows the data transfer lifecycle.

Figure 6-90 Data transfer lifecycle



a. 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.

b. 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).

c. 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.

d. Periodically set checkpoints (orange arrows).

   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** 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.

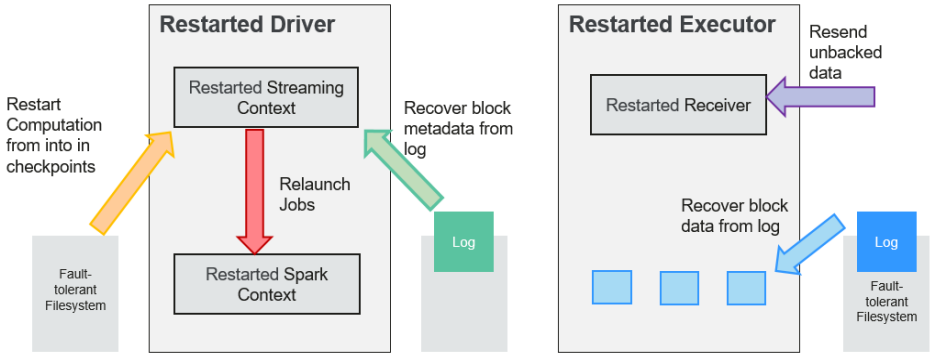
NOTE

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 6-91 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)

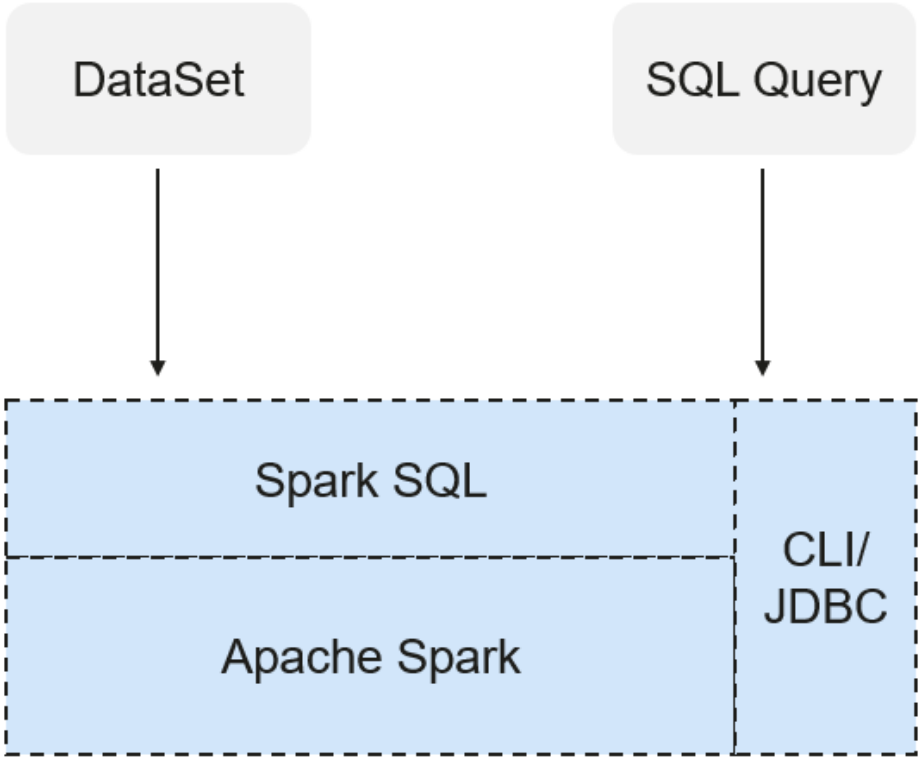
   Data that is cached but not stored to logs upon failures is re-sent by data sources, because the receiver does not confirm the data.

Therefore, by using WALs and reliable Receiver, Spark Streaming can avoid input data loss caused by Driver failures.

SparkSQL and DataSet Principle

SparkSQL

Figure 6-92 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 3.1.1 realizes command localization and replaces the 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 a 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 for Spark programming 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()
```

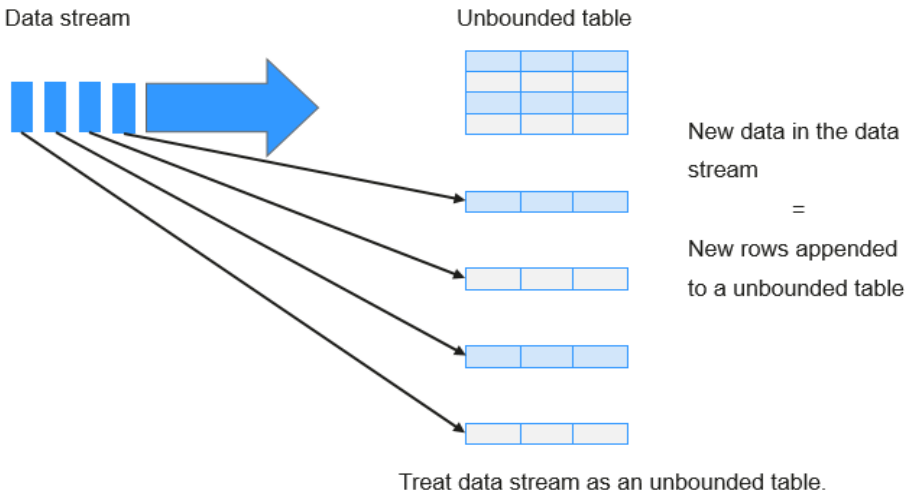
- SparkSession can be used to execute SQL queries on data and return results as DataFrame.
  ```
  sparkSession.sql("select * from person").show
  ```
- SparkSession can be used to set configuration items during running. These configuration items can be replaced with variables in SQL statements.
  ```
  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 Principle

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.

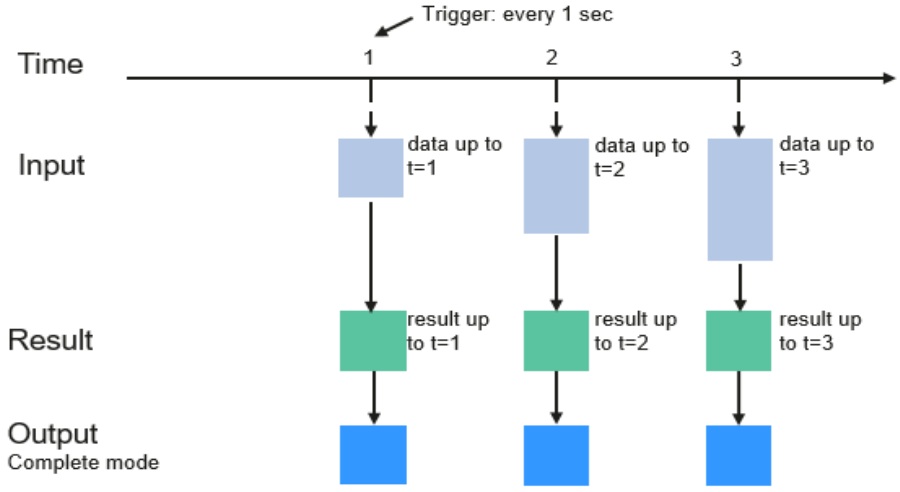
The core of Structured Streaming is to take streaming data as an incremental database table. Similar to the data block processing model, the streaming data processing model applies query operations on a static database table to streaming computing, and Spark uses standard SQL statements for query, to obtain data from the incremental and unbounded table.

Figure 6-93 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 6-94 Structured Streaming data processing model



Programming Model for Structured Streaming

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.

Basic 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.

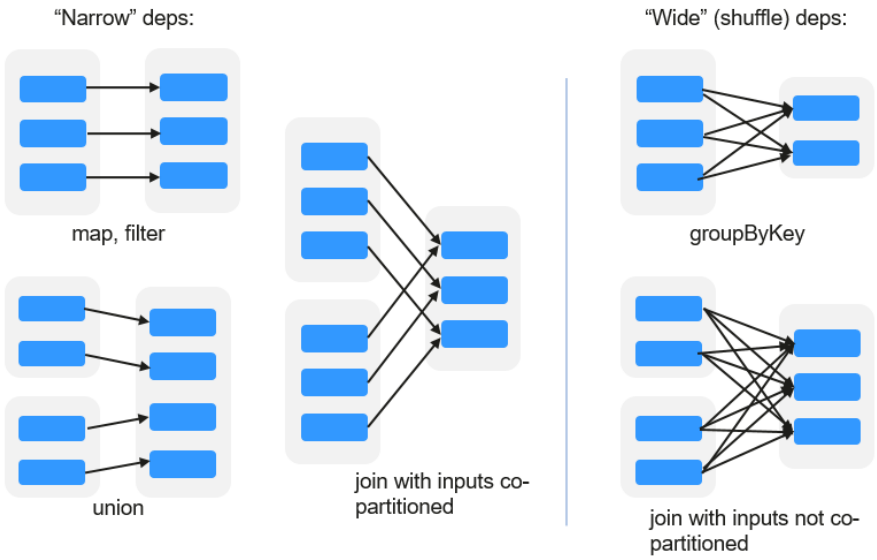
  RDD Storage

  – 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 6-95 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.

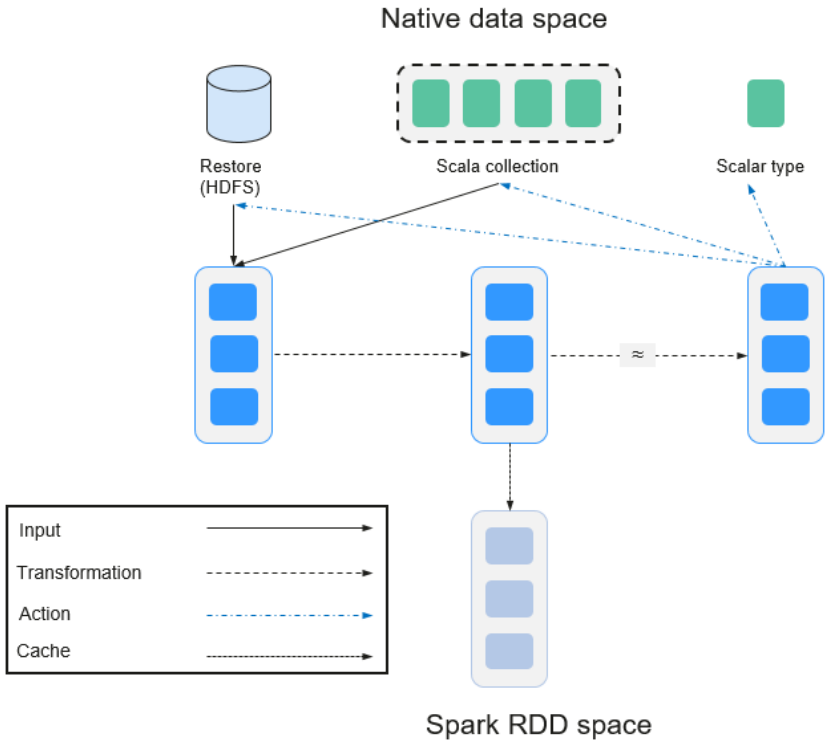
  The narrow dependency facilitates the optimization. Logically, each RDD operator is a fork/join (the join is not the join operator mentioned above but the barrier used to synchronize multiple concurrent tasks); fork the RDD to

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 6-96** 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 6-96 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**).

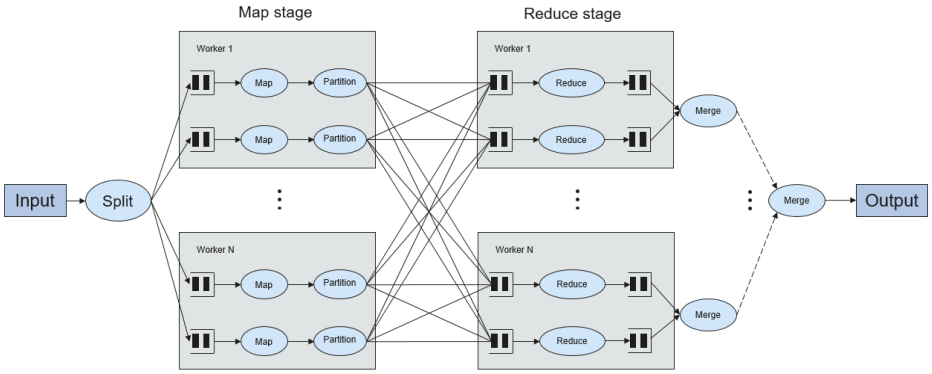
  - Check points, such as the **checkpoint** operator. When Lineage is quite long (which occurs frequently in graphics computation), it takes a long period of time to execute the whole sequence again when a fault occurs. In this case, checkpoint is used as the check point to write the current data to stable 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 6-97 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/3.1.1/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.

6.27.2 Spark HA Solution

Spark Multi-Active Instance HA Principles and Implementation Solution

Based on existing JDBCServer in the community, multi-active-instance mode is used to achieve HA. 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 mode, multi-active instance mode has following advantages:

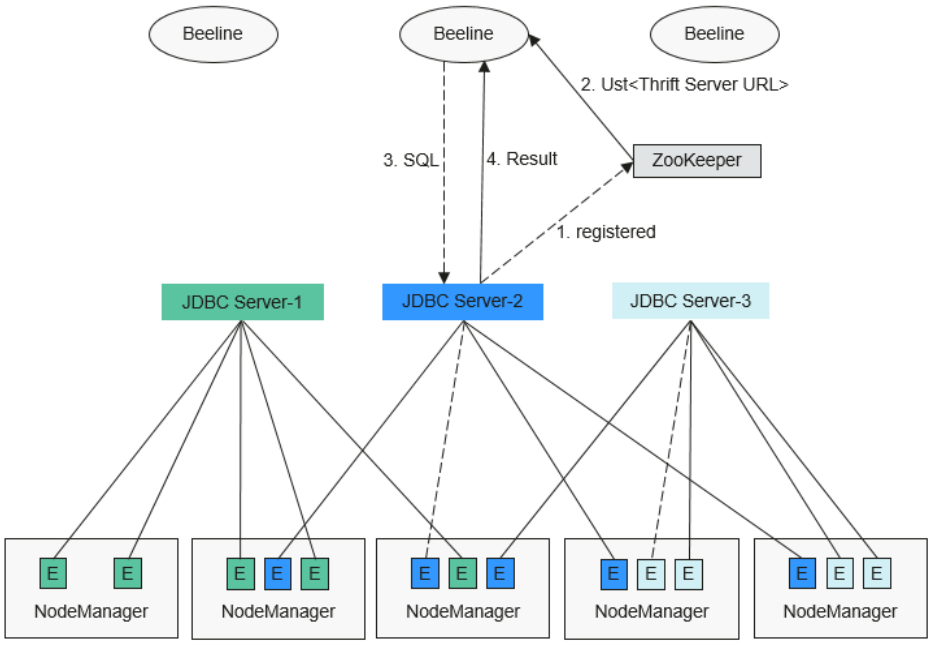
- In active/standby HA, when the active/standby switchover occurs, the unavailable period cannot be controlled by JDBCServer, but it depends on Yarn service resources.
- In Spark, the Thrift JDBC similar to HiveServer2 provides services and users access services through Beeline and JDBC API. Therefore, the processing capability of the JDBCServer cluster depends on the single-point capability of the primary server, and the scalability is insufficient.

The multi-active instance HA mode not only can prevent service interruption caused by switchover, but also enables cluster scale-out to improve high concurrency.

- **Implementation**

  The following figure shows the basic principle of multi-active instance HA of Spark JDBCServer.

Figure 6-98 Spark JDBCServer HA



1. When a JDBCServer is started, it registers with ZooKeeper by writing node information in a specified directory. Node information includes the instance IP address, port number, version, and serial number.
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.
3. After the connection succeeds, the client sends SQL statements to JDBCServer.
4. JDBCServer executes received SQL statements and returns results to the client.

If multi-active instance HA of Spark JDBCServer is enabled, all JDBCServer instances are independent and equivalent. When one JDBCServer instance is interrupted during upgrade, other JDBCServer instances can accept the connection request from the client.

The rules below 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 the 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 caused by random distribution of policy results, and finally result in load imbalance of instances.
- After the instance enters the maintenance mode (in which no new connection requests from clients are accepted), services 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 list below.

- 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=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark/hadoop.**<*System domain name*>**@**<*System domain name*>**;**

  NOTE

  - In the above JDBCURL, **<zkNode_IP>:<zkNode_Port>** indicates the ZooKeeper URL. Use commas (,) to separate multiple URLs,

    Example: 192.168.81.37:2181,192.168.195.232:2181,192.168.169.84:2181.
  - **sparkthriftserver2x** indicates the ZooKeeper directory, where a random JDBCServer instance is connected to the client.

  For example, when you use Beeline client to connect JDBCServer, 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=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark/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=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark/hadoop.**<*System domain name*>@<*System domain name*>**;user.principal=**<*principal_name*>**;user.keytab=**<*path_to_keytab*>

  In the above URL, <*principal_name*> indicates the principal of the 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=sparkthriftserver2x;**

  For example, when you use Beeline client, in normal mode, for connection, 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=sparkthriftserver2x;"**

– Non-multi-active instance mode

  In this mode, a client connects to a specified JDBCServer node. Compared with multi-active instance mode, the connection string in this mode does not contain **serviceDiscoveryMode** and **zooKeeperNamespace** parameters about ZooKeeper.

For example, when you use Beeline client, in security mode, 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=spark/hadoop.**<*System domain name*>@<*System domain name*>**;saslQop=auth-conf;auth=KERBEROS;principal=spark/hadoop.**<*System domain name*>@<*System domain name*>**;"**

NOTE

- In the above command, **<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 the two modes are the same. Spark JDBCServer is another implementation of HiveServer2 in Hive. For details about how to use Spark JDBCServer, see **https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients**.

Spark Multi-Tenant HA

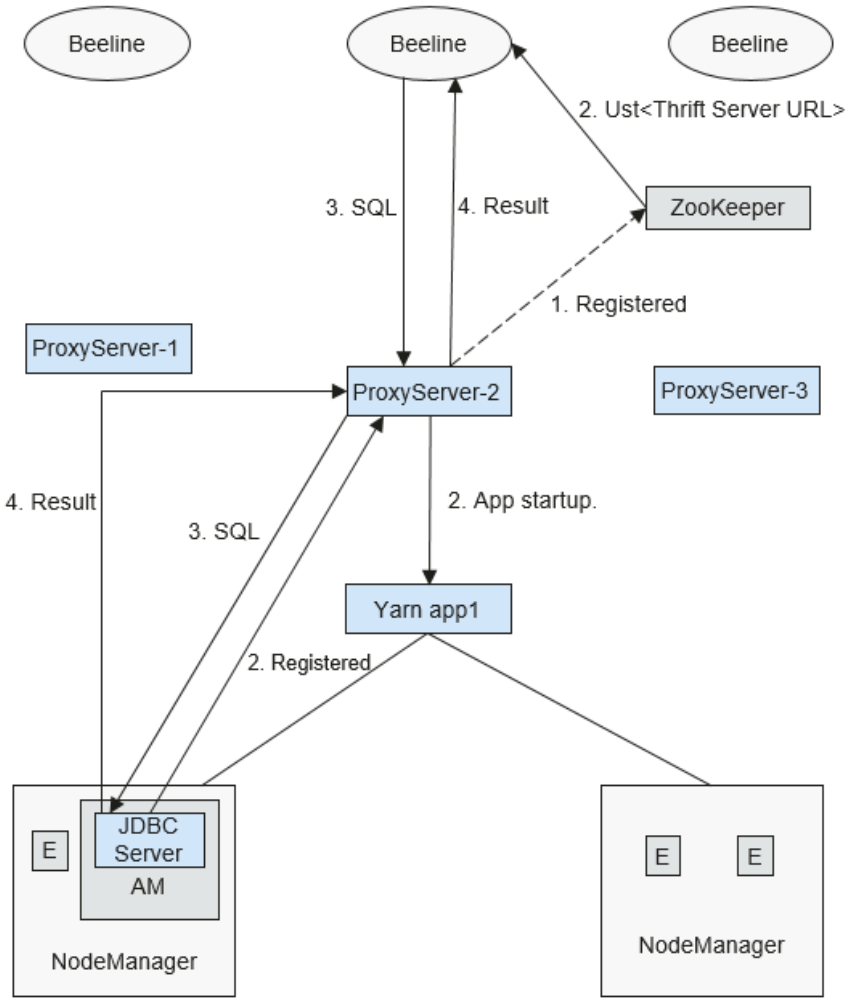
In the JDBCServer multi-active instance solution, JDBCServer uses the Yarn-client mode, but there is only one Yarn resource queue available. To solve this resource limitation problem, the multi-tenant mode is introduced.

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.

- **Implementation**

  Figure 6-99 shows the HA solution of the multi-tenant mode.

Figure 6-99 Multi-tenant mode of Spark JDBCServer



a. When ProxyServer is started, it registers with ZooKeeper by writing node information in a specified directory. Node information includes the instance IP address, port number, version, and serial number.

   NOTE

   In multi-tenant mode, the JDBCServer instance refers to the ProxyServer (JDBCServer proxy).

b. To connect to ProxyServer, the client must specify a namespace, which is the directory of the ProxyServer instance where you want to access ZooKeeper. When the client connects to the ProxyServer, a random instance under the namespace is selected for connection. For details about the URL, see **URL Connection Overview**.

c. After the client successfully connects to the ProxyServer, which first 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.

d. The client sends SQL statements to ProxyServer, which forwards statements to the connected JDBCServer. JDBCServer returns the results to ProxyServer, which then returns the results to the client.

In the multi-active instance HA mode, all instances are independent and equivalent. If one instance is interrupted during upgrade, other instances can accept the connection request from the client.

- **URL Connection Overview**
  - Multi-tenant mode

    In multi-tenant mode, the client reads content from the ZooKeeper node and connects to ProxyServer. The connection strings are list below.

    - 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=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark/hadoop.**<*System domain name*>@<*System domain name*>;

      NOTE

      - In the above URL, **<zkNode_IP>:<zkNode_Port>** indicates the ZooKeeper URL. Use commas (,) to separate multiple URLs,

        Example:
        192.168.81.37:2181,192.168.195.232:2181,192.168.169.84:2181.
      - **sparkthriftserver2x** indicates the ZooKeeper directory, where a random JDBCServer instance is connected to the client.

      For example, when you use Beeline client for connection, 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=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark/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=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark/hadoop.**<*System domain name*>@<*System domain name*>**;user.principal=**<*principal_name*>**;user.keytab=**<*path_to_keytab*>

      In the above URL, <*principal_name*> indicates the principal of the 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=**sparkthriftserver2x**;

      For example, run the following command when you use Beeline client for connection in normal mode:

      sh *CLIENT_HOME*/**spark/bin/beeline -u "jdbc:hive2://**<*zkNode1_IP*>:<*zkNode1_Port*>,<*zkNode2_IP*>:<*zkNode2_Port*>,<*zkN*

ode3_IP>:<zkNode3_Port>/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;"

- Non-multi-tenant mode

  In non-multi-tenant mode, a client connects to a specified JDBCServer node. Compared with multi-tenant instance mode, the connection string in this 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=spark/hadoop.*<System domain name>@<System domain name>*;saslQop=auth-conf;auth=KERBEROS;principal=spark/hadoop.*<System domain name>@<System domain name>*;"

  NOTE

  - In the above command, **<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, go to 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.

  If you use Beeline client for connection, run the following command (**aaa** is the tenant name):

  beeline --hiveconf mapreduce.job.queuename=aaa -u 'jdbc:hive2://192.168.39.30:2181,192.168.40.210:2181,192.168.215.97:2181;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark/hadoop.*<System domain name>@<System domain name>*;'

6.27.3 Relationship Among Spark, HDFS, and Yarn

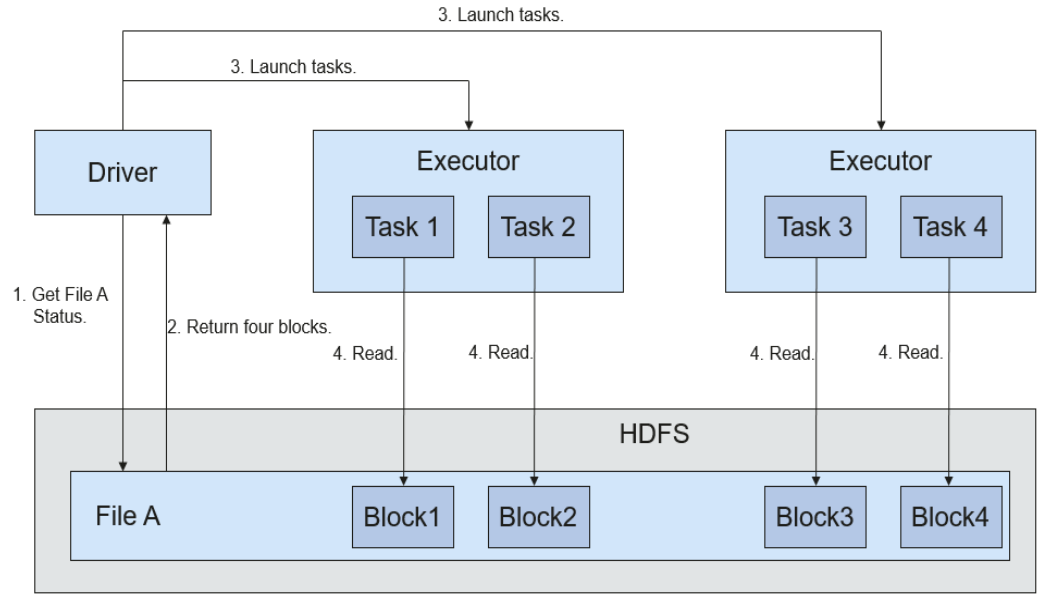
Relationship Between Spark and HDFS

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

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

Figure 6-100 shows the process of reading a file.

Figure 6-100 File reading process

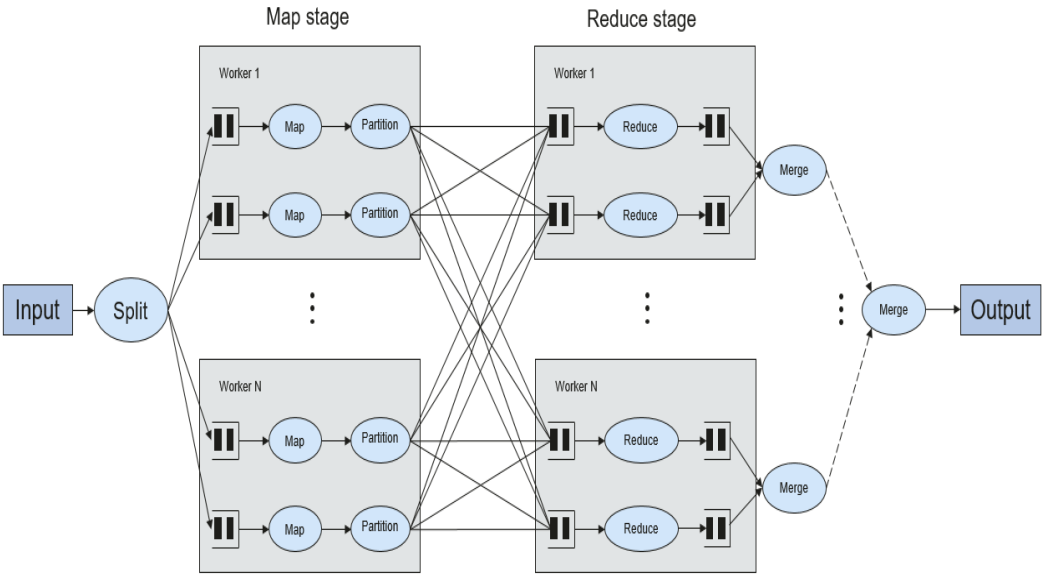


The file reading process is as follows:

1. Driver interconnects with the 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 6-101 shows the process of writing data to a file.

Figure 6-101 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.

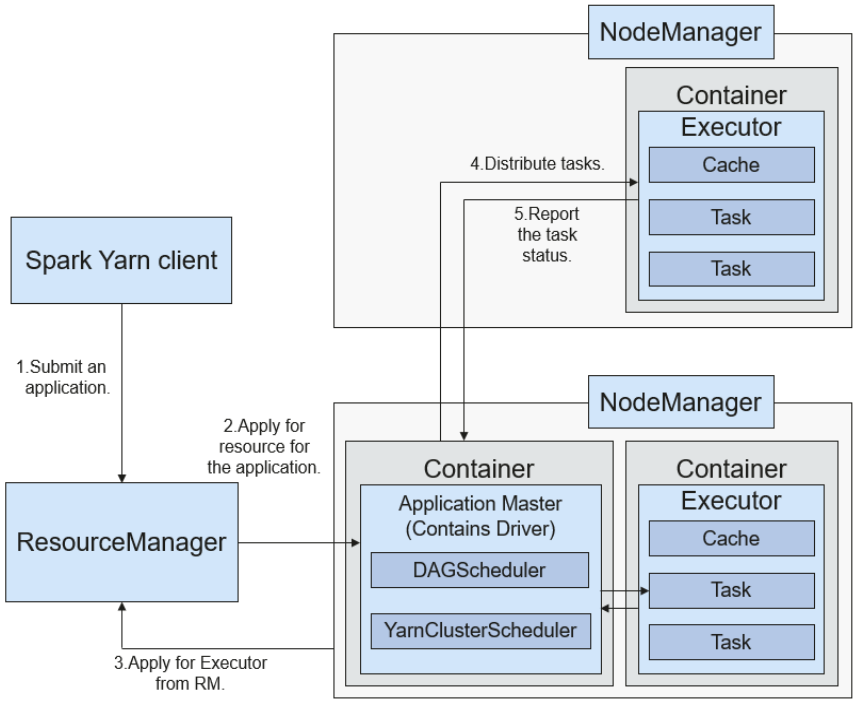
Relationship Between Spark and Yarn

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 6-102 shows the running framework of Spark on Yarn-cluster.

  Figure 6-102 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 driver on the container.
  c. ApplicationMaster applies for resources from ResourceManager to run the container.

     ResourceManager allocates the container to ApplicationMaster, which communicates with NodeManager, and starts the executor in the obtained container. After the executor is started, it registers with the driver and applies for tasks.

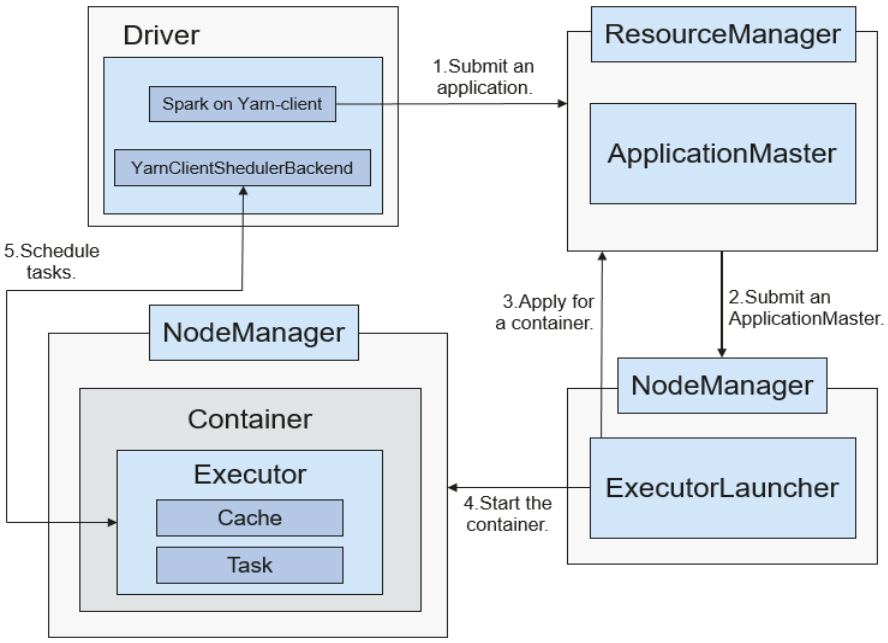
d. The driver allocates tasks to the executor.

e. The executor runs tasks and reports the operating status to the driver.

- Yarn-client mode

  Figure 6-103 shows the running framework of Spark on Yarn-cluster.

  Figure 6-103 Spark on Yarn-client operation framework



Spark on Yarn-client implementation process:

NOTE

In Yarn-client mode, Driver is deployed on the client and started on the client. In Yarn-client mode, the client of the 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 executors in the obtained containers. After the executors are started, it registers with drivers and applies for tasks.

NOTE

Running containers are not suspended and resources are not released.

e. The drivers allocate tasks to the executors. The executor executes tasks and reports the operating status to the driver.

6.27.4 Spark Enhanced Open Source Feature: Optimized SQL Query of Cross-Source Data

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 6-23 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.

6.28 Spark2x

6.28.1 Spark2x Basic Principles

NOTE

The Spark2x component applies to MRS 3.x and later versions.

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 Spark2x, see **Spark2x 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 6-104 describes the Spark architecture and **Table 6-24** lists the Spark modules.

Figure 6-104 Spark architecture

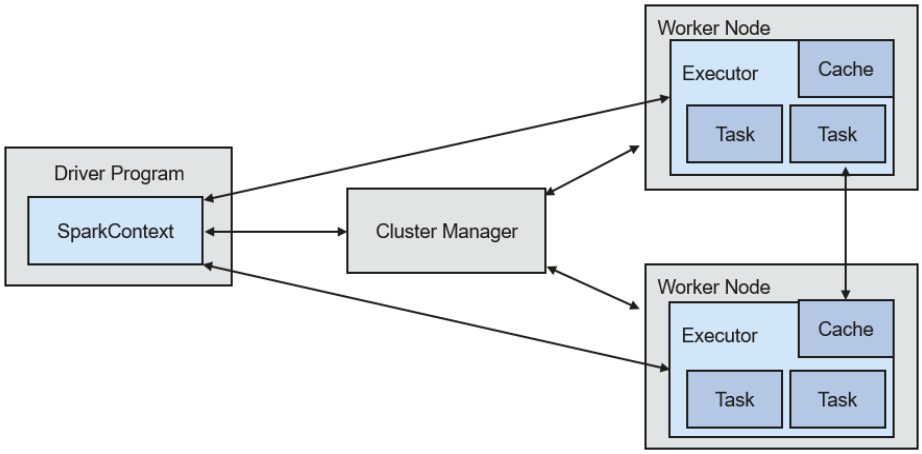


Table 6-24 Basic concepts

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).

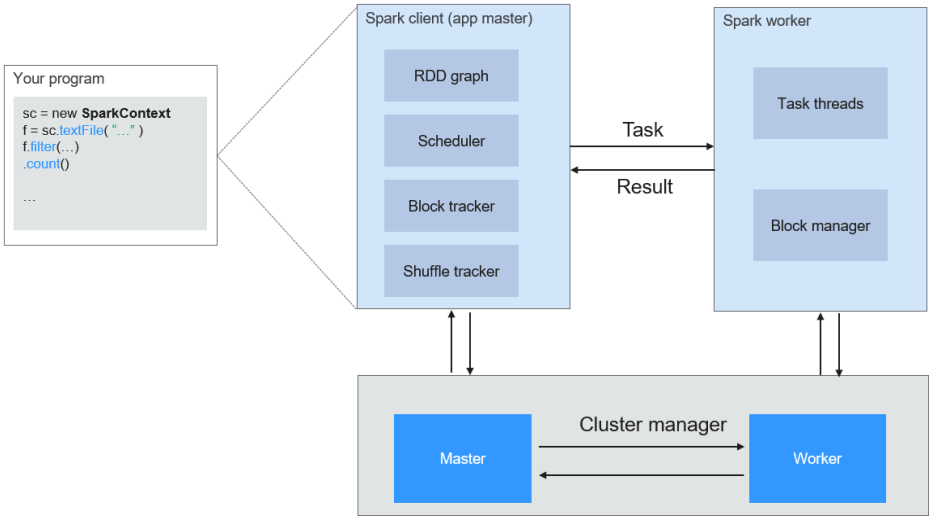
Module	Description
Task	A task carries the computation unit of the service logics. It is the minimum working unit that can be executed on the Spark platform. An application can be divided into multiple tasks based on the execution plan and computation amount.

Spark Principle

Figure 6-105 describes the application running architecture of Spark.

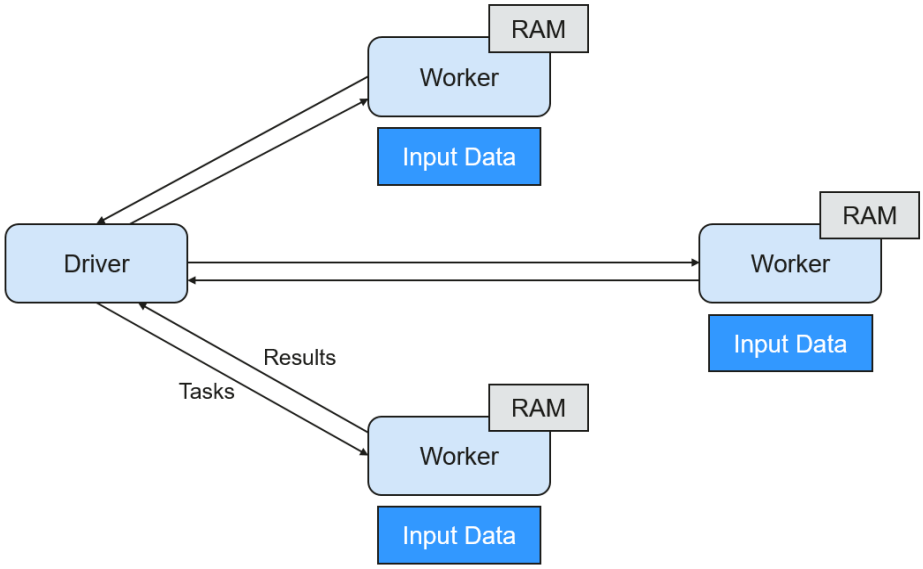
1. An application is running in the cluster as a collection of processes. Driver coordinates the running of the application.
2. To run an application, Driver connects to the cluster manager (such as Standalone, Mesos, and Yarn) to apply for the executor resources, and start ExecutorBackend. The cluster 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 6-105 Spark application running architecture



Spark uses Master and Worker modes, as shown in **Figure 6-106**. 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 6-106 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 Principle

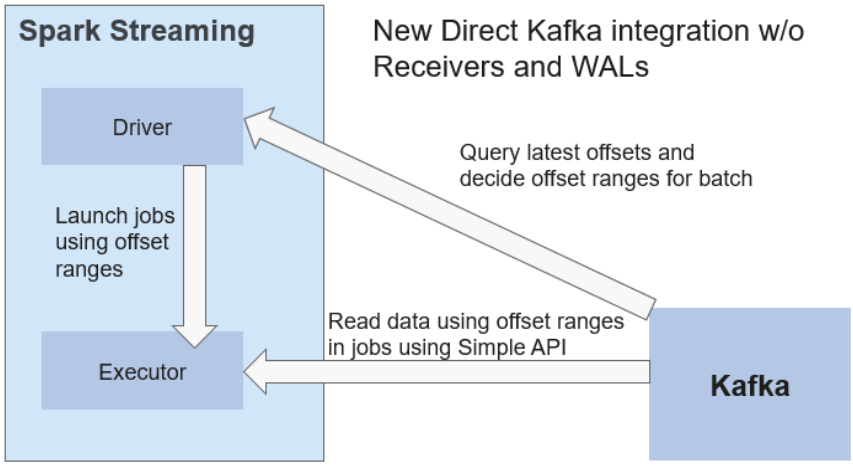
Spark Streaming is a real-time computing framework built on the Spark, which expands the capability for processing massive streaming data. Spark supports two data processing approaches: Direct Streaming and Receiver.

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 6-107 Data transmission through Direct Kafka API



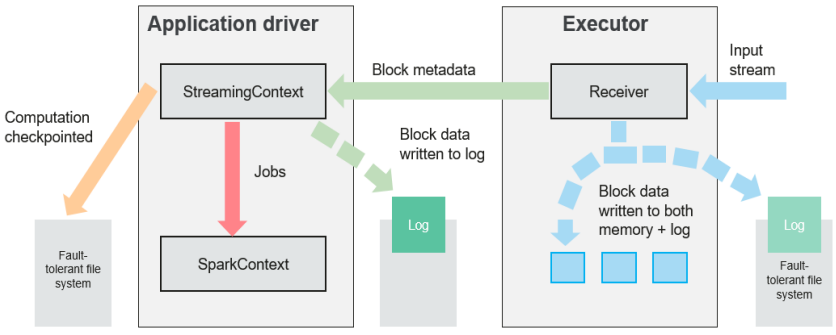
After the failure, Spark Streaming can read data from Kafka again and process the data segment. The processing result is the same no matter Spark Streaming fails or not, because the semantic is processed only once.

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 6-108** shows the data transfer lifecycle.

Figure 6-108 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.

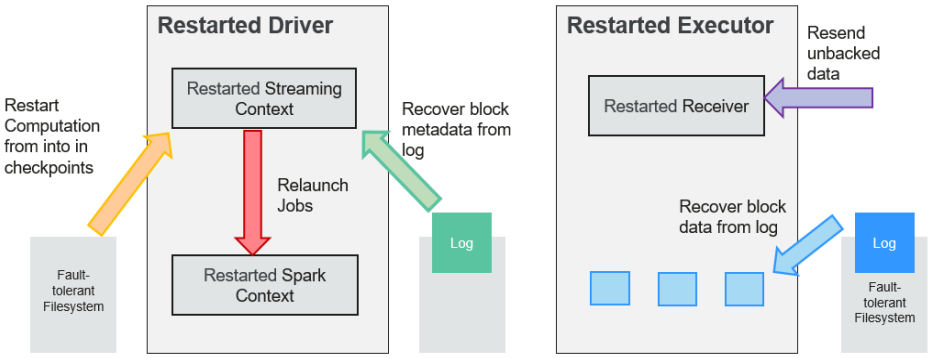
NOTE

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 6-109 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)

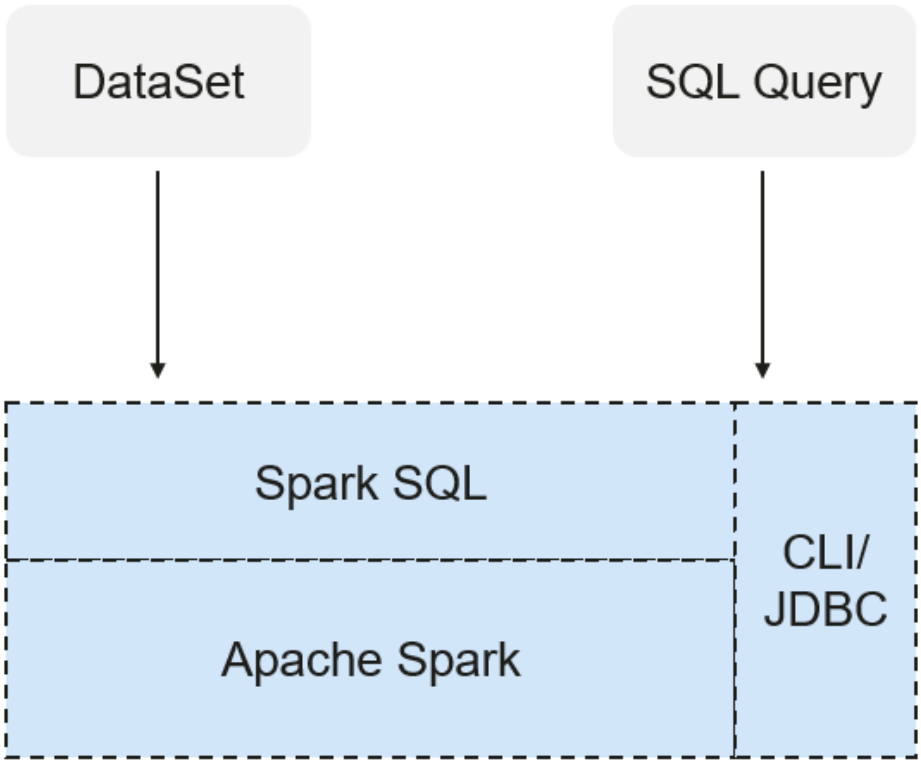
   Data that is cached but not stored to logs upon failures is re-sent by data sources, because the receiver does not confirm the data.

Therefore, by using WALs and reliable Receiver, Spark Streaming can avoid input data loss caused by Driver failures.

SparkSQL and DataSet Principle

SparkSQL

Figure 6-110 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.

Spark2x realizes command localization and replaces the 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 Spark2x 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()
```

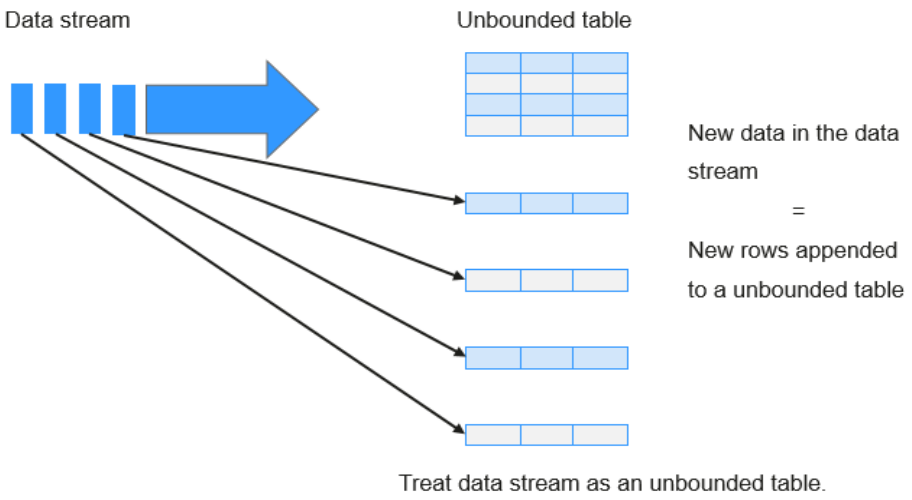
- SparkSession can be used to execute SQL queries on data and return results as DataFrame.
  ```
  sparkSession.sql("select * from person").show
  ```
- SparkSession can be used to set configuration items during running. These configuration items can be replaced with variables in SQL statements.
  ```
  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 Principle

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.

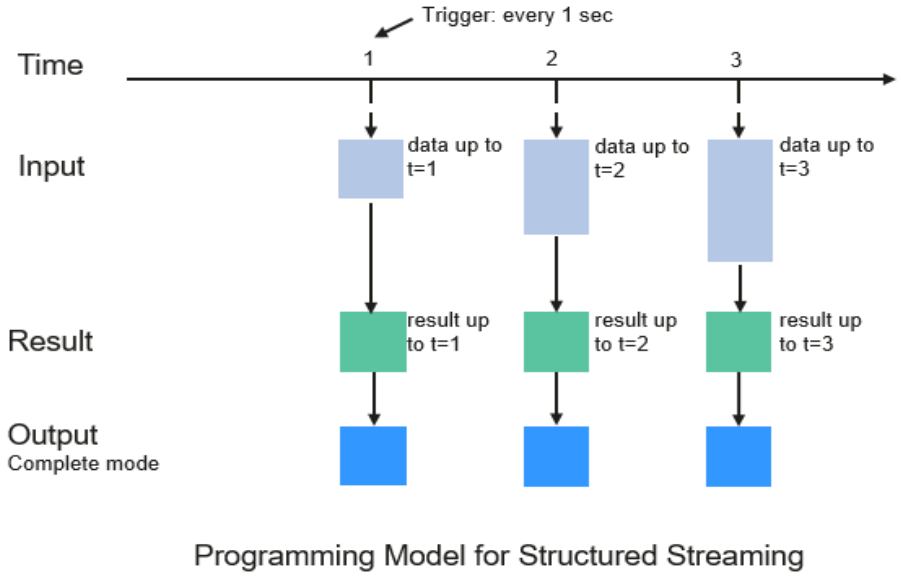
The core of Structured Streaming is to take streaming data as an incremental database table. Similar to the data block processing model, the streaming data processing model applies query operations on a static database table to streaming computing, and Spark uses standard SQL statements for query, to obtain data from the incremental and unbounded table.

Figure 6-111 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 6-112 Structured Streaming data processing model



Programming Model for Structured Streaming

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.

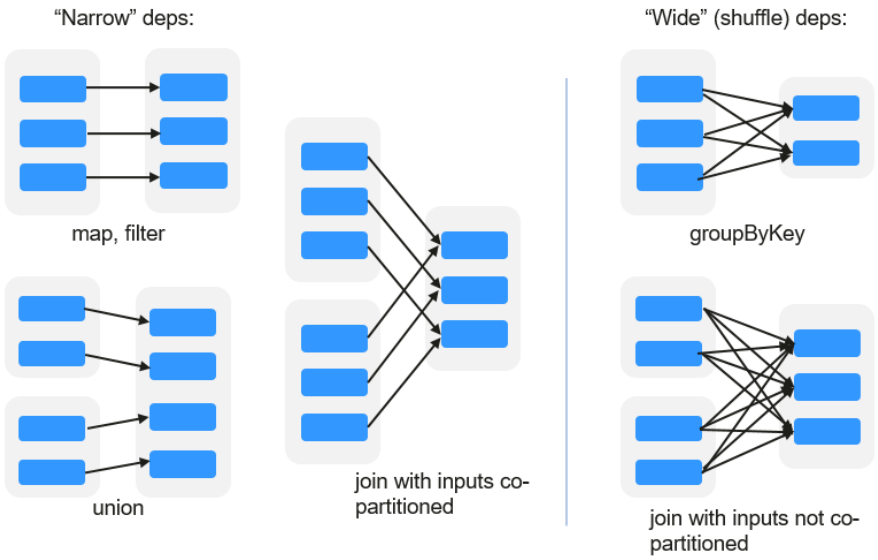
  RDD Storage

  - 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 6-113 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.

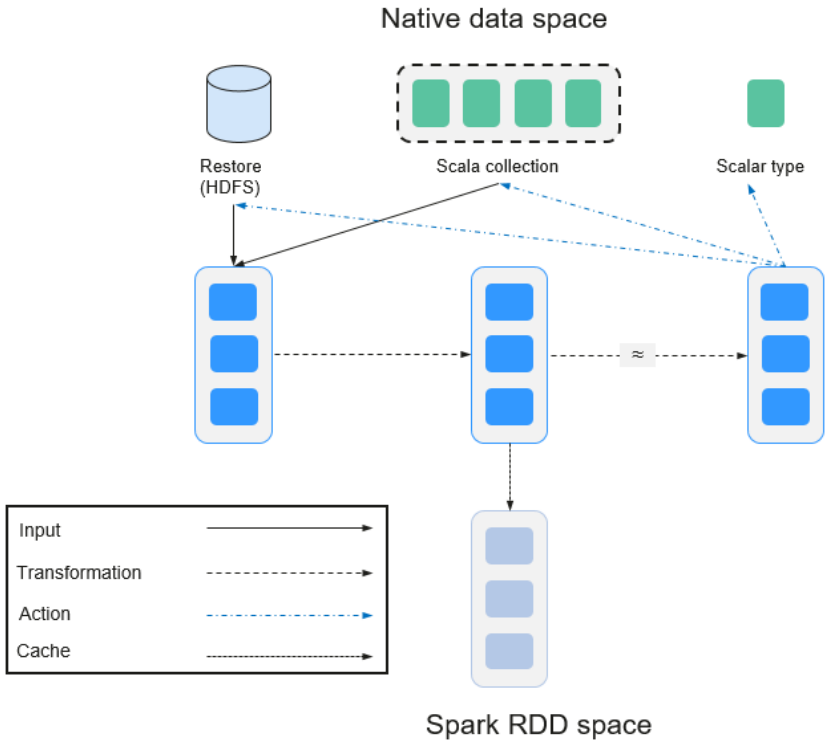
  The narrow dependency facilitates the optimization. Logically, each RDD operator is a fork/join (the join is not the join operator mentioned above but the barrier used to synchronize multiple concurrent tasks); fork the RDD to

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 6-114** 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 6-114 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**).

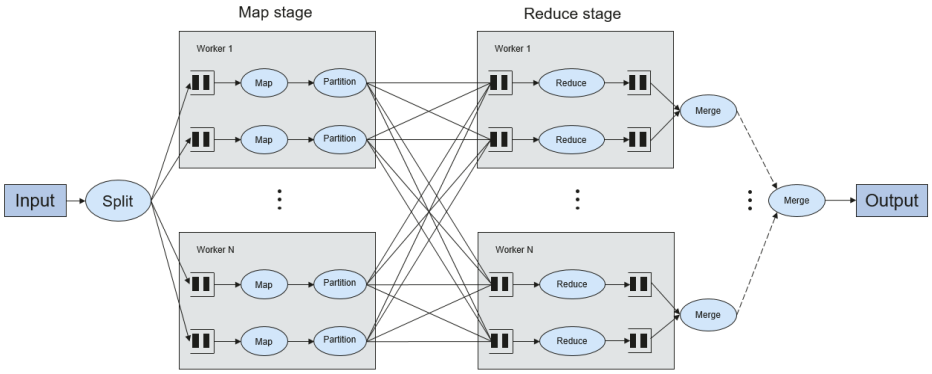
  - Check points, such as the **checkpoint** operator. When Lineage is quite long (which occurs frequently in graphics computation), it takes a long period of time to execute the whole sequence again when a fault occurs. In this case, checkpoint is used as the check point to write the current data to stable 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 6-115 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/3.1.1/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.

6.28.2 Spark2x HA Solution

6.28.2.1 Spark2x Multi-active Instance

Background

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:

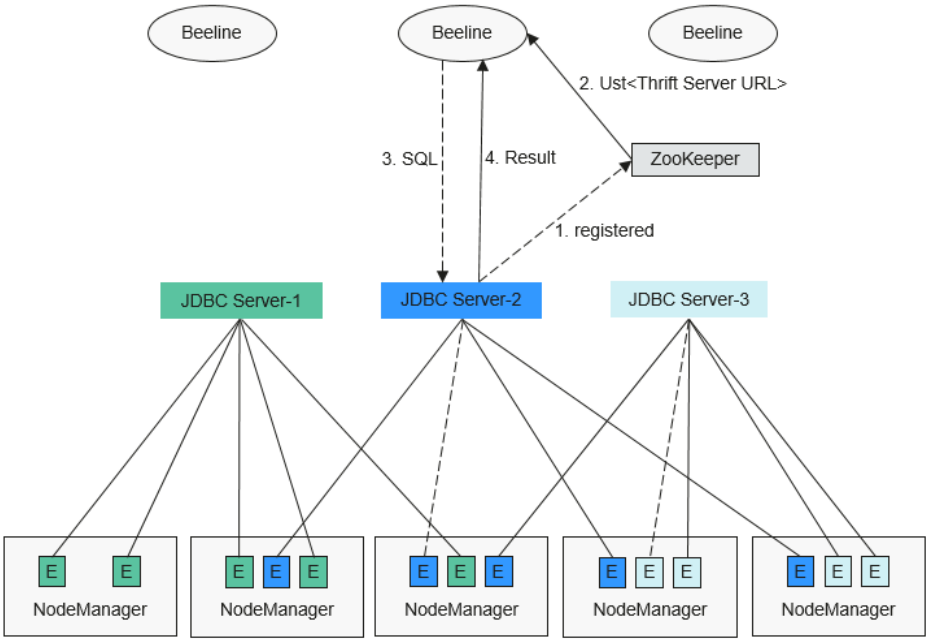
- In active/standby HA, when the active/standby switchover occurs, the unavailable period cannot be controlled by JDBCServer, but determined by Yarn service resources.
- In Spark, the Thrift JDBC similar to HiveServer2 provides services and users access services through Beeline and JDBC API. Therefore, 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.

Implementation

The following figure shows the basic principle of multi-active instance HA of Spark JDBCServer.

Figure 6-116 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=8.1.0.1;sequence=0000001244,serverUri=192.168.195.232:22550 ;version=8.1.0.1;sequence=000
   0001242,serverUri=192.168.81.37:22550;version=8.1.0.1;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=sparkthriftserver2x;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,
      192.168.81.37:2181,192.168.195.232:2181,192.168.169.84:2181.
    - **sparkthriftserver2x** 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=sparkthriftserver2x;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=sparkthriftserver2x;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=**sparkthriftserver2x**;

  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=sparkthriftserver2x;"

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, go to the official Hive website at **https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients**.

6.28.2.2 Spark2x Multi-tenant

Background

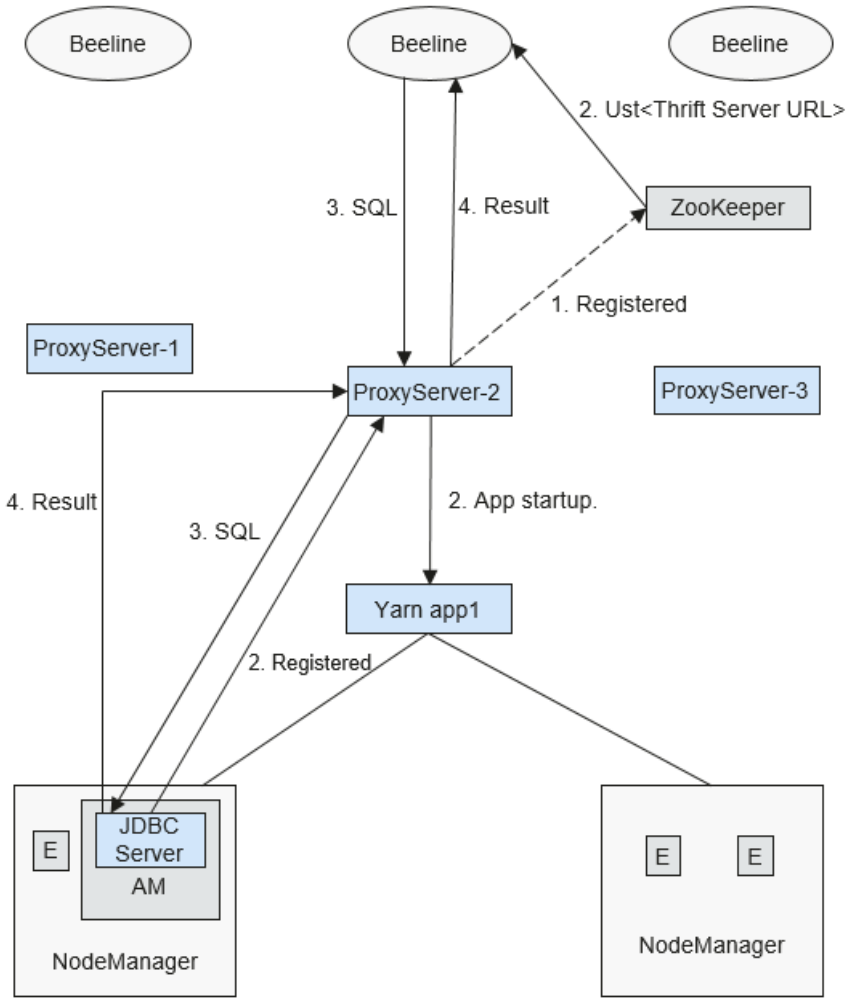
In the JDBCServer multi-active instance mode, JDBCServer implements the Yarn-client mode but only one Yarn resource queue is available. To solve the resource limitation problem, the multi-tenant mode is introduced.

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.

Implementation

Figure 6-117 shows the HA solution of the multi-tenant mode.

Figure 6-117 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=8.1.0.1;sequence=0000001244,serverUri=192.168.195.232:22550
   ;version=8.1.0.1;sequence=0000001242,serverUri=192.168.81.37:22550
   ;version=8.1.0.1;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. During the connection, the system performs the modulo operation on the hash value of the tenant name and the number of namespace instances in ZooKeeper to obtain the connected instance. 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=sparkthriftserver2x;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, **192.168.81.37:2181,192.168.195.232:2181,192.168.169.84:2181**.
    - **sparkthriftserver2x** 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=sparkthriftserver2x;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=sparkthriftserver2x;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>/;service
DiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;
```

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=sparkthriftserver2x;"**

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=spark/hadoop.**<*System domain name*>**@**<*System domain name*>**;saslQop=auth-conf;auth=KERBEROS;principal=spark/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, go to 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://192.168.39.30:2181,192.168.40.210:2181,192.168.215.97:2181;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=sparkthriftserver2x;saslQop=auth-conf;auth=KERBEROS;principal=spark2x/hadoop.*<System domain name>*@*<System domain name>*;'

6.28.3 Relationship Between Spark2x and Other Components

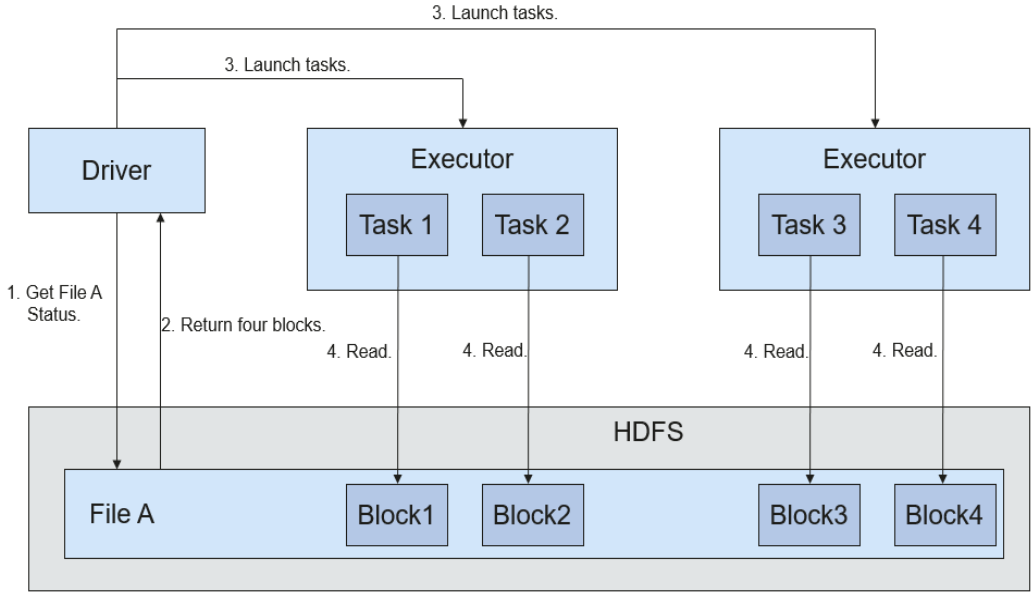
Relationship Between Spark and HDFS

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

Figure 6-118 describes the file reading process.

Figure 6-118 File reading process

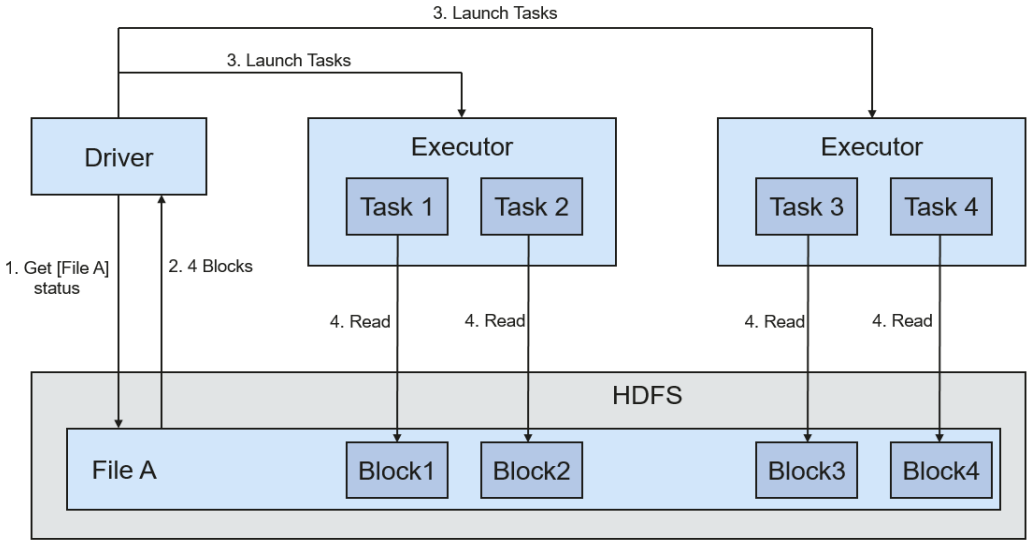


The file reading process is as follows:

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 6-119 describes the file writing process.

Figure 6-119 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.

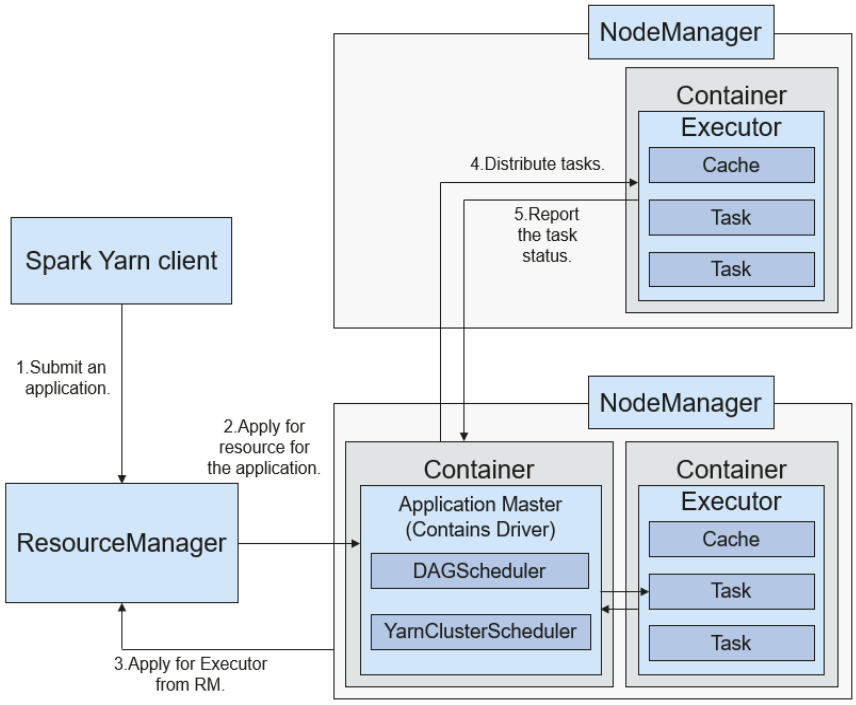
Relationship with Yarn

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 6-120 describes the operation framework.

Figure 6-120 Spark on Yarn-cluster operation framework



Spark on Yarn-cluster implementation process:

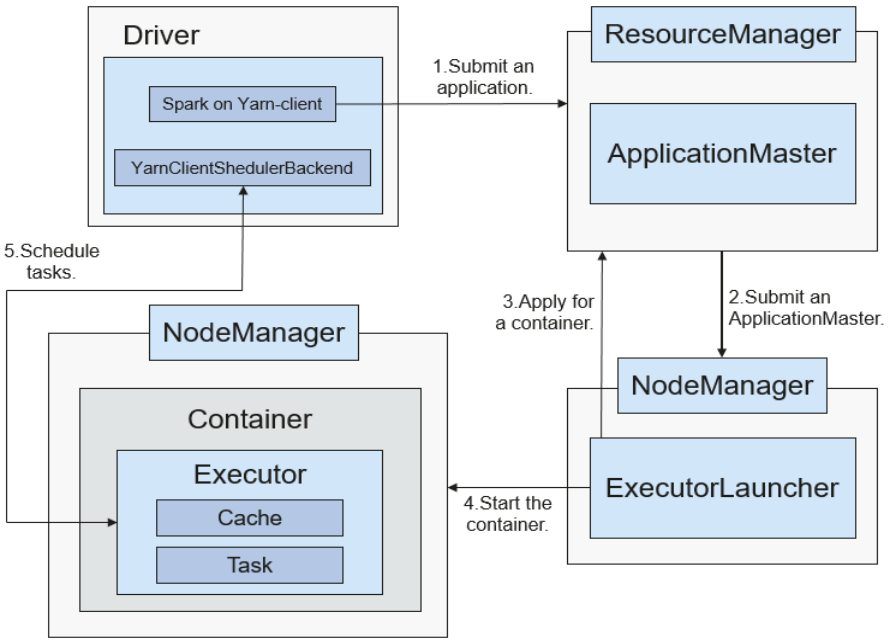
a. The client generates the application information, and then sends the information to ResourceManager.
b. ResourceManager allocates the first container (ApplicationMaster) to SparkApplication and starts the driver on the container.
c. ApplicationMaster applies for resources from ResourceManager to run 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 6-121 describes the operation framework.

Figure 6-121 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.

6.28.4 Spark2x Open Source New Features

Purpose

Compared with Spark 1.5, Spark2*x* has some new open-source features. The specific features or concepts are as follows:

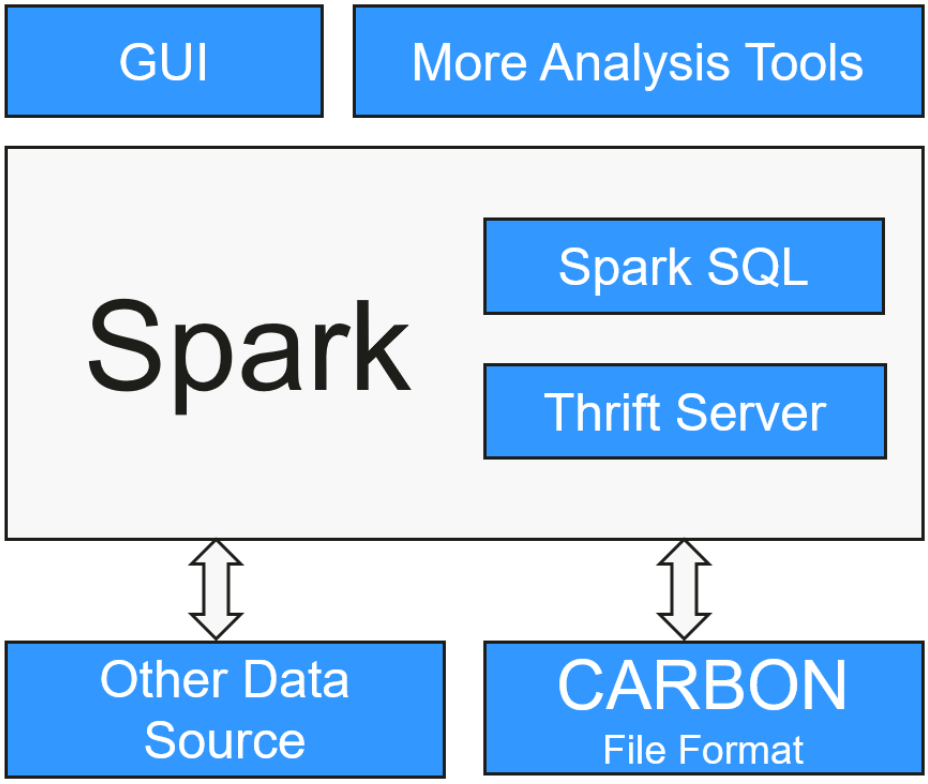
- DataSet: For details, see **SparkSQL and DataSet Principle**.
- Spark SQL Native DDL/DML: For details, see **SparkSQL and DataSet Principle**.
- SparkSession: For details, see **SparkSession Principle**.
- Structured Streaming: For details, see **Structured Streaming Principle**.
- Optimizing Small Files
- Optimizing the Aggregate Algorithm
- Optimizing Datasource Tables
- Merging CBO

6.28.5 Spark2x Enhanced Open Source Features

6.28.5.1 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 6-122 Basic architecture of CarbonData



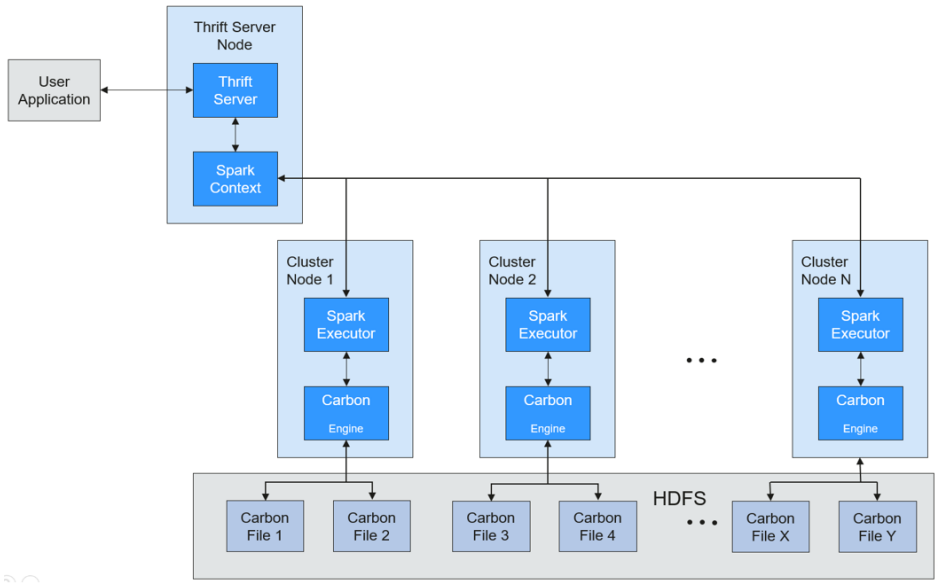
The purpose of using CarbonData is to provide quick response to ad hoc queries of big data. Essentially, CarbonData is an Online Analytical Processing (OLAP) engine, which stores data by using tables similar to those in Relational Database Management System (RDBMS). You can import more than 10 TB data 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 6-123 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.

6.28.5.2 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 6-25 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

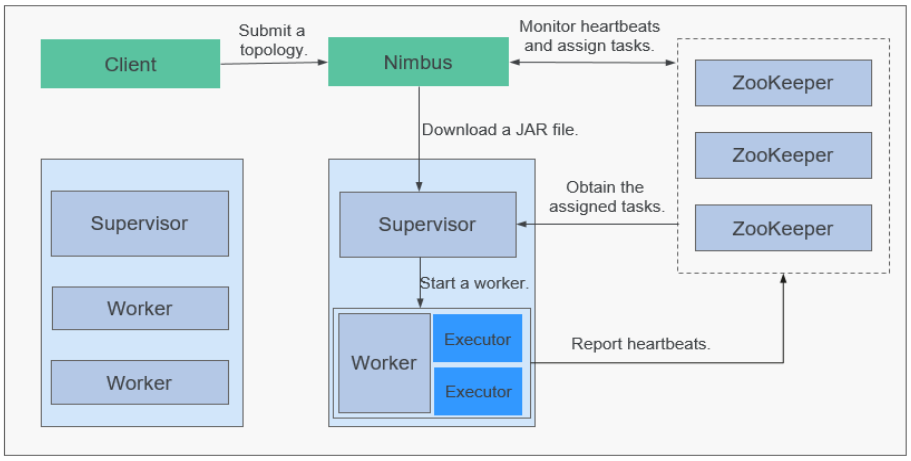
- If external data source is Hive, query operation cannot be performed on foreign tables created by Spark.
- Only MySQL and MPPDB data sources are supported.

6.29 Storm

6.29.1 Storm Basic Principles

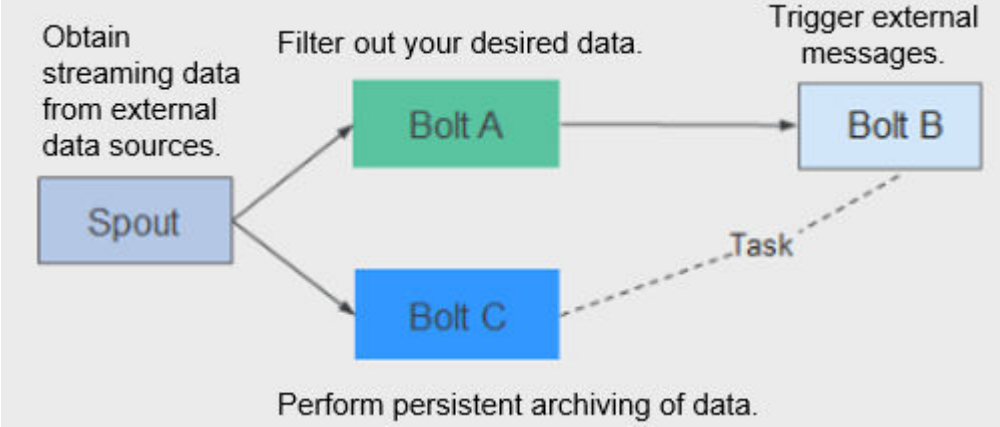
Apache Storm is a distributed, reliable, and fault-tolerant real-time stream data processing system. In Storm, a graph-shaped data structure called topology needs to be designed first for real-time computing. The topology will be submitted to a cluster. Then a master node in the cluster distributes codes and assigns tasks to worker nodes. A topology contains two roles: spout and bolt. A spout sends messages and sends data streams in tuples. A bolt converts the data streams and performs computing and filtering operations. The bolt can randomly send data to other bolts. Tuples sent by a spout are unchangeable arrays and map to fixed key-value pairs.

Figure 6-124 Basic architecture of Storm



Service processing logic is encapsulated in the topology of Storm. A topology is a set of spout (data sources) and bolt (logical processing) components that are connected using Stream Groupings in DAG mode. All components (spout and bolt) in a topology are working in parallel. In a topology, you can specify the parallelism for each node. Then, Storm allocates tasks in the cluster for computing to improve system processing capabilities.

Figure 6-125 Topology



Storm is applicable to real-time analysis, continuous computing, and distributed extract, transform, and load (ETL). It has the following advantages:

- Wide applications
- High scalability
- Zero data loss
- High fault tolerance
- Easy to construct and control
- Multi-language support

Storm is a computing platform and provides Continuous Query Language (CQL) in the service layer to facilitate service implementation. CQL has the following features:

- Easy to use: The CQL syntax is similar to the SQL syntax. Users who have basic knowledge of SQL can easily learn CQL and use it to develop services.
- Rich functions: In addition to basic expressions provided by SQL, CQL provides functions, such as windows, filtering, and concurrency setting, for stream processing.
- Easy to scale: CQL provides an extension API to support increasingly complex service scenarios. Users can customize the input, output, serialization, and deserialization to meet specific service requirements.
- Easy to debug: CQL provides detailed explanation of error codes, facilitating users to rectify faults.

For details about Storm architecture and principles, see **https://storm.apache.org/**.

Principle

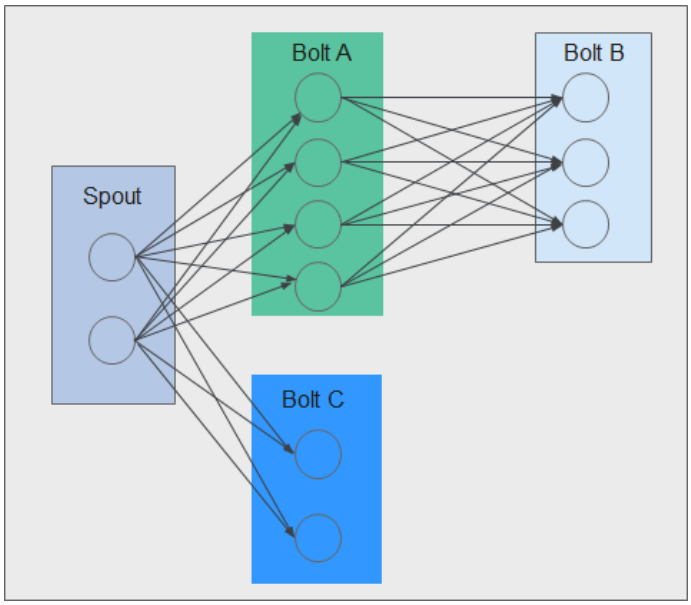
- **Basic Concepts**

Table 6-26 Concepts

Concept	Description
Tuple	A tuple is an invariable key-value pair used to transfer data. Tuples are created and processed in distributed manner.
Stream	A stream is an unbounded sequence of tuples.
Topology	A topology is a real-time application running on the Storm platform. It is a Directed Acyclic Graph (DAG) composed of components. A topology can concurrently run on multiple machines. Each machine runs a part of the DAG. A topology is similar to a MapReduce job. The difference is that the topology is a resident program. Once started, the topology cannot stop unless it is manually terminated.
Spout	A spout is the source of tuples. For example, a spout may read data from a message queue, database, file system, or TCP connection and converts them as tuples, which are processed by the next component.
Bolt	In a Topology, a bolt is a component that receives data and executes specific logic, such as filtering or converting tuples, joining or aggregating streams, and performing statistics and result persistence.
Worker	A Worker is a physical processing in running state in a Topology. Each Worker is a JVM process. Each Topology may be executed by multiple Workers. Each Worker executes a logic subset of the Topology.
Task	A task is a spout or bolt thread of a Worker.
Stream groupings	A stream grouping specifies the tuple dispatching policies. It instructs the subsequent bolt how to receive tuples. The supported policies include Shuffle Grouping, Fields Grouping, All Grouping, Global Grouping, Non Grouping, and Directed Grouping.

Figure 6-126 shows a Topology (DAG) consisting of a Spout and Bolt. In the figure, a rectangle indicates a Spout or Bolt, the nodes in each rectangle indicate tasks, and the lines between tasks indicate streams.

Figure 6-126 Topology



- **Reliability**

  Storm provides three levels of data reliability:

  - At Most Once: The processed data may be lost, but it cannot be processed repeatedly. This reliability level offers the highest throughput.
  - At Least Once: Data may be processed repeatedly to ensure reliable data transmission. If a response is not received within the specified time, the Spout resends the data to Bolts for processing. This reliability level may slightly affect system performance.
  - Exactly Once: Data is successfully transmitted without loss or redundancy processing. This reliability level delivers the poorest performance.

  Select the reliability level based on service requirements. For example, for the services requiring high data reliability, use Exactly Once to ensure that data is processed only once. For the services insensitive to data loss, use other levels to improve system performance.

- **Fault Tolerance**

  Storm is a fault-tolerant system that offers high availability. **Table 6-27** describes the fault tolerance of the Storm components.

Table 6-27 Fault tolerance

Scenario	Description
Nimbus failed	Nimbus is fail-fast and stateless. If the active Nimbus is faulty, the standby Nimbus takes over services immediately, and provide external services.
Supervisor failed	Supervisor is a background daemon of Workers. It is fail-fast and stateless. If a Supervisor is faulty, the Workers running on the node are not affected but cannot receive new tasks. The OMS can detect the fault of the Supervisor and restart the processes.

Scenario	Description
Worker failed	If a Worker is faulty, the Supervisor on the Worker will restart it again. If the restart fails for multiple times, Nimbus reassigns tasks to other nodes.
Node failed	If a node is faulty, all the tasks being processed by the node time out and Nimbus will assign the tasks to another node for processing.

Open Source Features

- Distributed real-time computing

  In a Storm cluster, each machine supports the running of multiple work processes and each work process can create multiple threads. Each thread can execute multiple tasks. A task indicates concurrent data processing.

- High fault tolerance

  During message processing, if a node or a process is faulty, the message processing unit can be redeployed.

- Reliable messages

  Data processing methods including At-Least Once, At-Most Once, and Exactly Once are supported.

- Security mechanism

  Storm provides Kerberos-based authentication and pluggable authorization mechanisms, supports SSL Storm UI and Log Viewer UI, and supports security integration with other big data platform components (such as ZooKeeper and HDFS).

- Flexible topology defining and deployment

  The Flux framework is used to define and deploy service topologies. If the service DAG is changed, users only need to modify YAML domain specific language (DSL), but do not need to recompile or package service code.

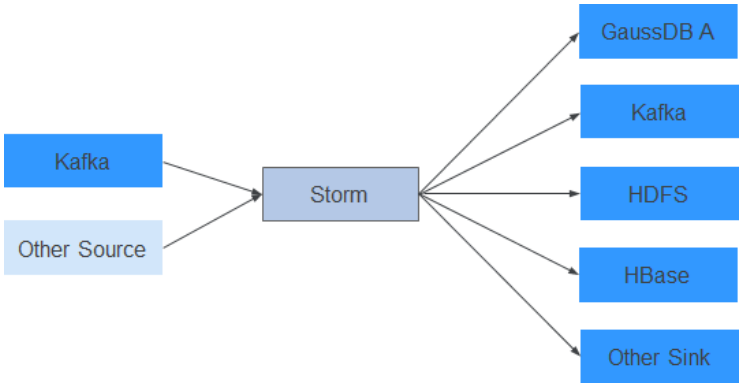
- Integration with external components

  Storm supports integration with multiple external components such as Kafka, HDFS, HBase, Redis, and JDBC/RDBMS, implementing services that involve multiple data sources.

6.29.2 Relationships Between Storm and Other Components

Storm provides a real-time distributed computing framework. It can obtain real-time messages from data sources (such as Kafka and TCP connection), perform high-throughput and low-latency real-time computing on a real-time platform, and export results to message queues or implement data persistence. **Figure 6-127** shows the relationship between Storm and other components.

Figure 6-127 Relationship with other components



Relationship between Storm and Streaming

Both Storm and Streaming use the open source Apache Storm kernel. However, the kernel version used by Storm is 1.2.1 whereas that used by Streaming is 0.10.0. Streaming is used to inherit transition services in upgrade scenarios. For example, if Streaming has been deployed in an earlier version and services are running, Streaming can still be used after the upgrade. Storm is recommended in a new cluster.

Storm 1.2.1 has the following new features:

- **Distributed cache**: Provides external resources (configurations) required for sharing and updating the topology using CLI tools. You do not need to re-package and re-deploy the topology.
- **Native Streaming Window API**: Provides window-based APIs.
- **Resource scheduler**: Added the resource scheduler plug-in. When defining a topology, you can specify the maximum resources available and assign resource quotas to users, thus to manage topology resources of the users.
- **State management**: Provides the Bolt API with the checkpoint mechanism. When an event fails, Storm automatically manages the Bolt status and restore the event.
- **Message sampling and debugging**: On the Storm UI, you can enable or disable topology- or component-level debugging to output stream messages to specified logs based on the sampling ratio.
- **Worker dynamic analysis**: On the Storm UI, you can collect jstack and heap logs of the Worker process and restart the Worker process.
- **Dynamic adjustment of topology logs**: You can dynamically change the running topology logs on the CLI or Storm UI.
- **Improved performance**: Compared with earlier versions, the performance of Storm is greatly improved. Although the topology performance is closely related to the use case scenario and dependency on external services, the performance is three times higher in most scenarios.

6.29.3 Storm Enhanced Open Source Features

- CQL

  Continuous Query Language (CQL) is an SQL-like language used for real-time stream processing. Compared with SQL, CQL has introduced the concept of

(time-sequencing) window, which allows data to be stored and processed in the memory. The CQL output is the computing results of data streams at specific time. The use of CQL accelerates service development, enables tasks to be easily submitted to the Storm platform for real-time processing, facilitates output of results, and allows tasks to be terminated at the appropriate time.

- High Availability

  Nimbus HA ensures continuous service processing such as adding topologies and management even if one Nimbus is faulty, improving cluster availability.

6.30 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.

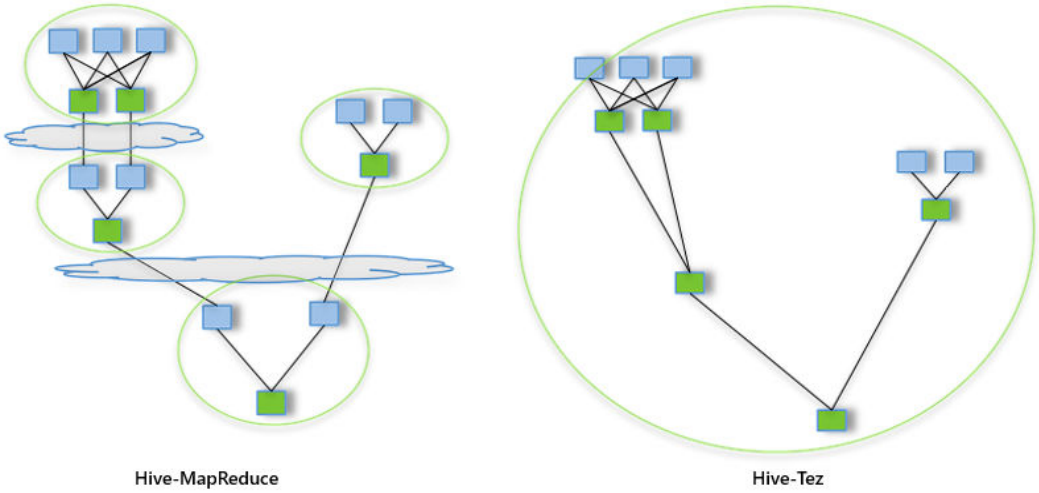
MRS uses Tez as the default execution engine of Hive. Tez remarkably surpasses the original MapReduce computing engine in terms of execution efficiency.

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 6-128 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.

6.31 YARN

6.31.1 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

- To use Yarn, ensure that the Hadoop service has been installed in the MRS cluster.
- An application is either a single job in the classical sense of MapReduce jobs or a Directed Acyclic Graph (DAG) of jobs.

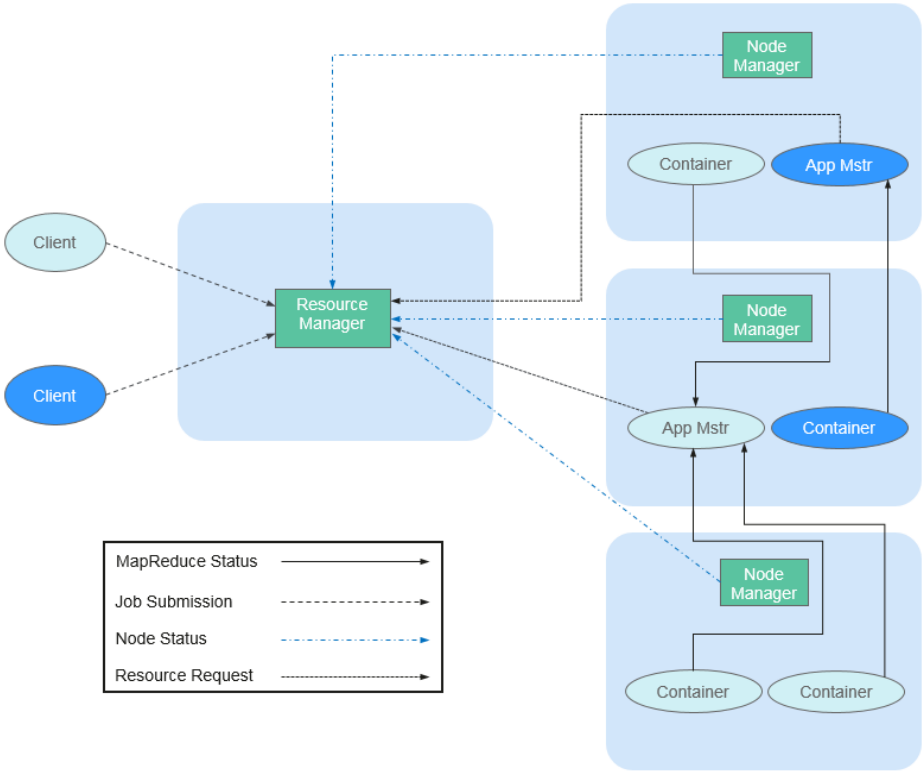
Architecture

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 6-129 Architecture



[Table 6-28](#) describes the components shown in [Figure 6-129](#).

Table 6-28 Architecture description

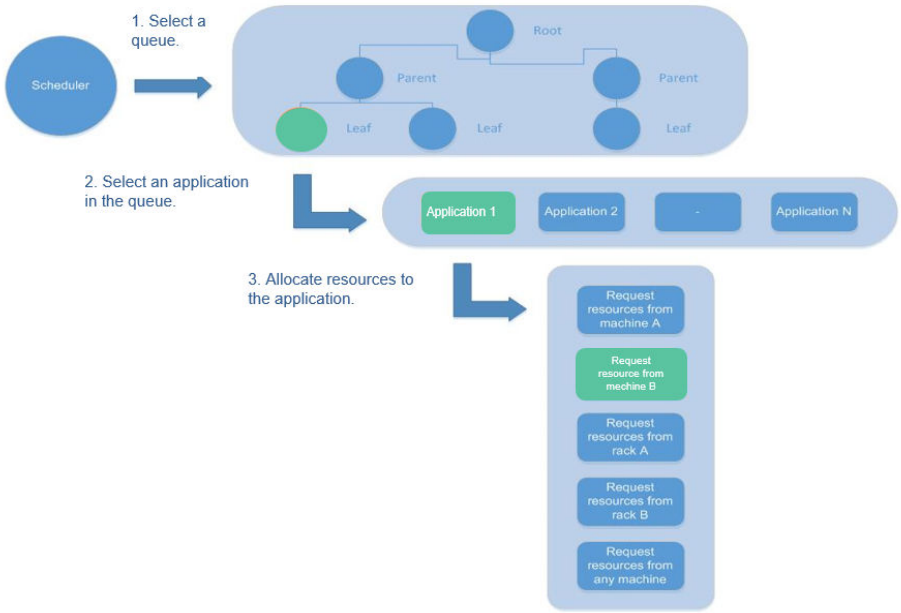
Name	Description
Client	Client of a YARN application. You can submit a task to ResourceManager and query the operating status of an application using the client.
ResourceManager(RM)	RM centrally manages and allocates all resources in the cluster. It receives resource reporting information from each node (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 lifecycle 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 6-130 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.

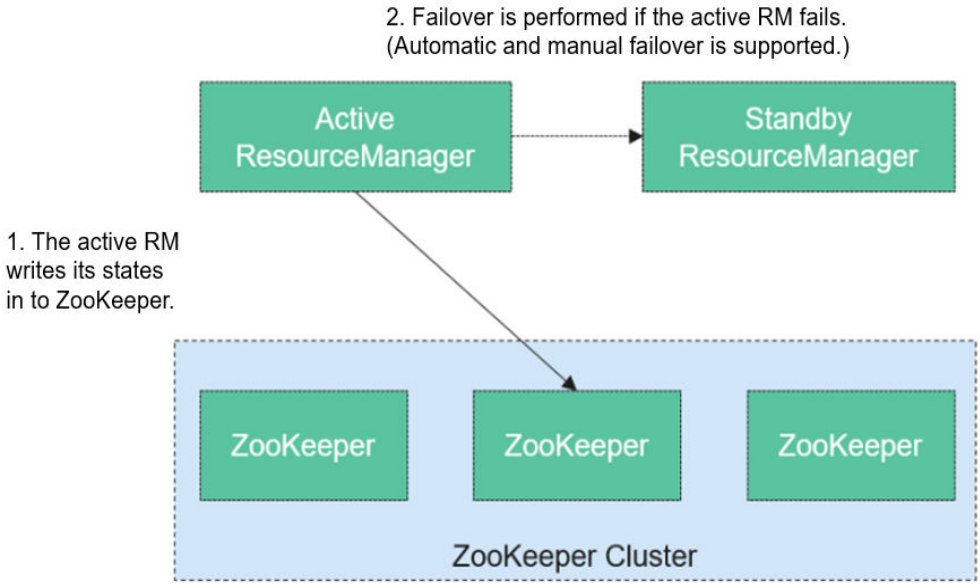
Each queue in Capacity Scheduler can limit the resource usage. However, the resource usage of a queue determines its priority when resources are allocated to queues, indicating that queues with smaller capacity are competitive. If the throughput of a cluster is big, delay scheduling enables an application to give up cross-machine or cross-rack scheduling, and to request local scheduling.

6.31.2 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 6-131 ResourceManager HA architecture



ResourceManager HA is achieved using active-standby ResourceManager nodes, as shown in **Figure 6-131**. 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 **ResourceManager 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.

6.31.3 Relationships Between YARN and Other Components

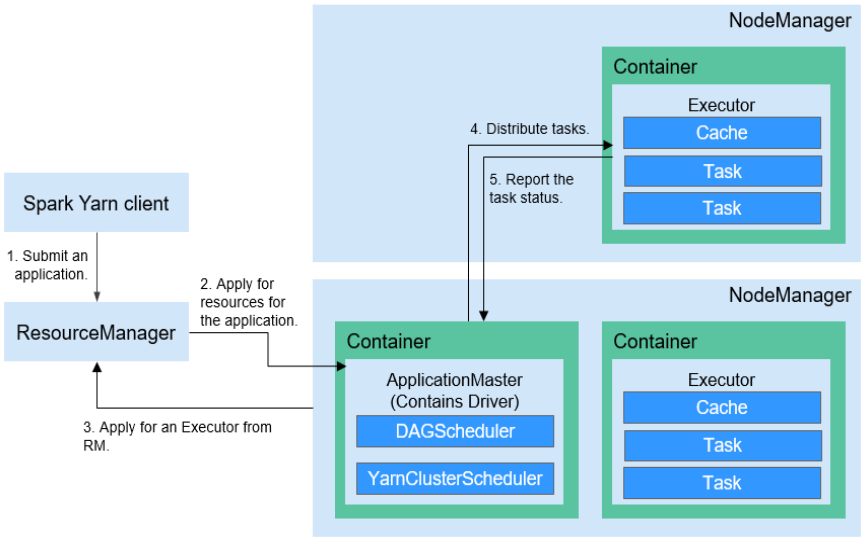
Relationship Between YARN and Spark

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 6-132 describes the operation framework.

  Figure 6-132 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.

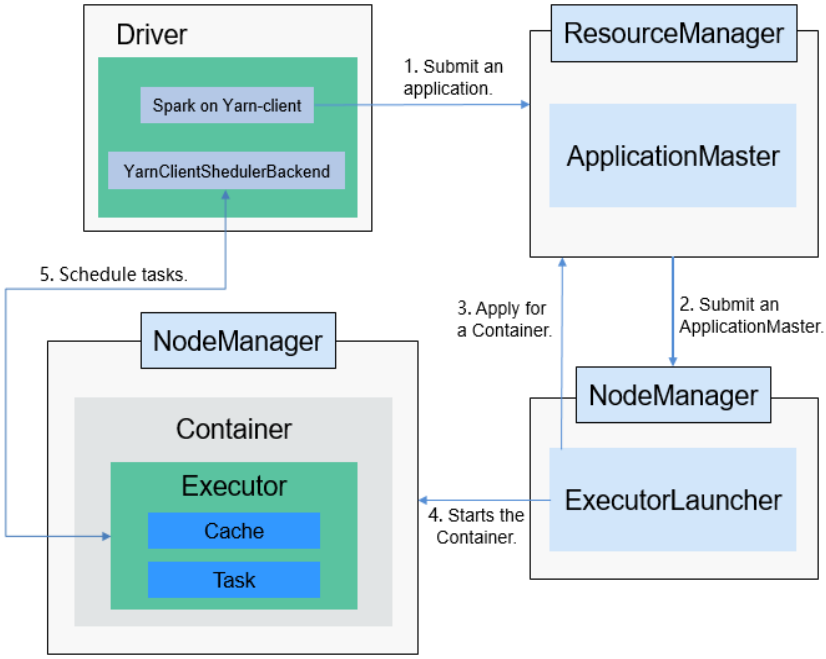
d. Drivers allocate tasks to the executors.

e. Executors run tasks and report the operating status to Drivers.

- YARN Client mode

  Figure 6-133 describes the operation framework.

  Figure 6-133 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 are not suspended and resources are not released.

e. Drivers allocate tasks to the executors. Executors run tasks and report the operating status to Drivers.

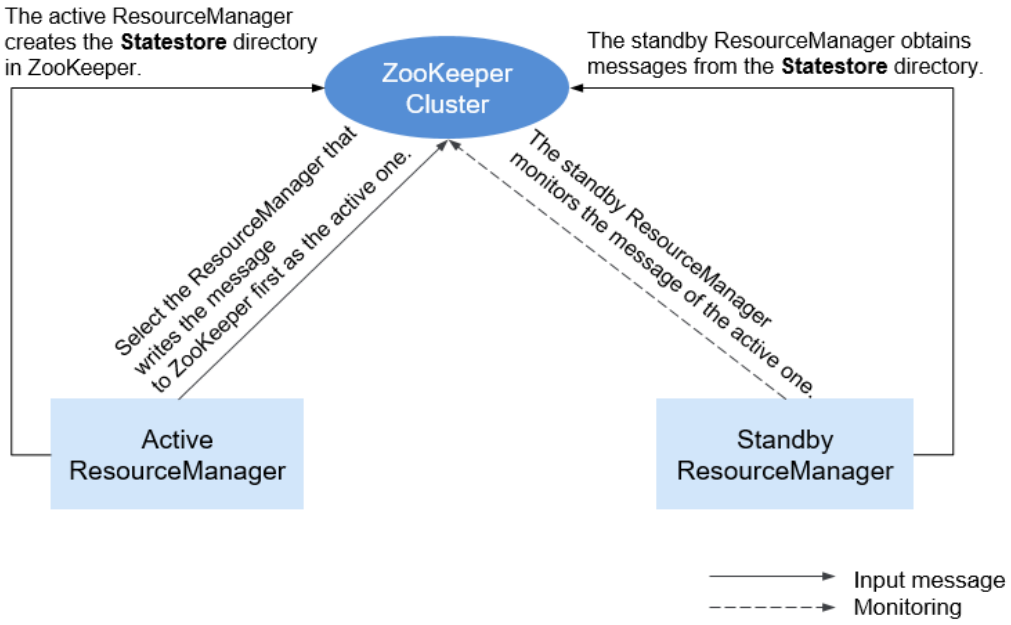
Relationship Between YARN and MapReduce

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.

Relationship Between YARN and ZooKeeper

Figure 6-134 shows the relationship between ZooKeeper and YARN.

Figure 6-134 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.

Relationship Between 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.

6.31.4 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.

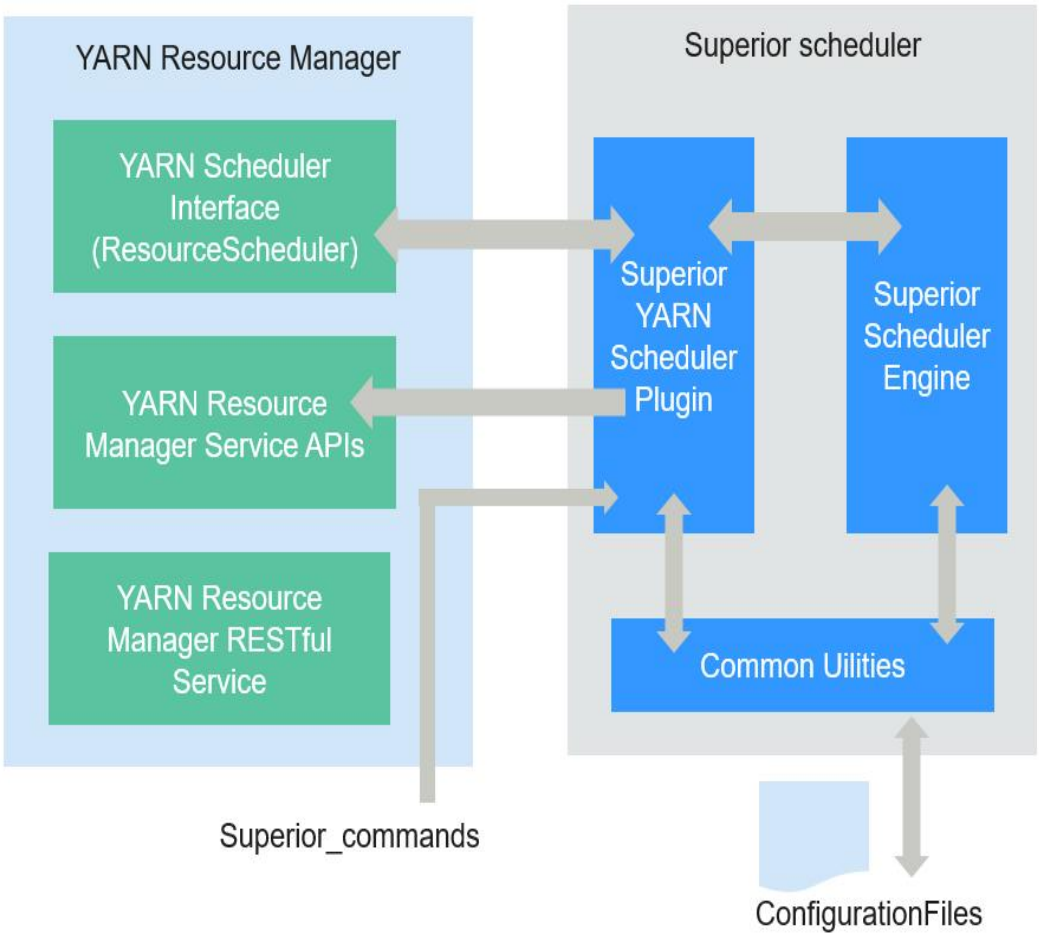
Superior Scheduler Principle (Self-developed)

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 6-135** shows the overall system diagram.

Figure 6-135 Internal architecture of Superior Scheduler



In **Figure 6-135**, Superior Scheduler consists of the following modules:

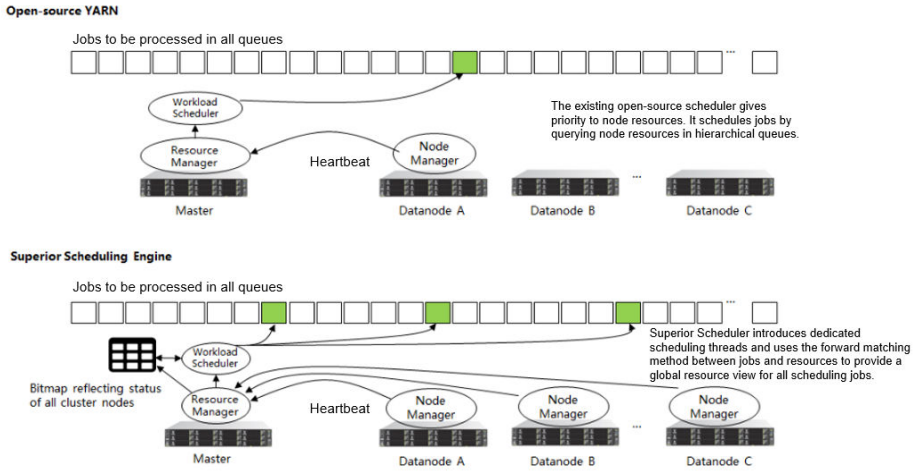
- Superior Scheduler Engine is a high performance scheduler engine with rich scheduling policies.
- Superior Yarn Scheduler Plugin functions as a bridge between Yarn ResourceManager and Superior Scheduler Engine and interacts with Yarn ResourceManager.

  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 an optimal 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 6-136 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

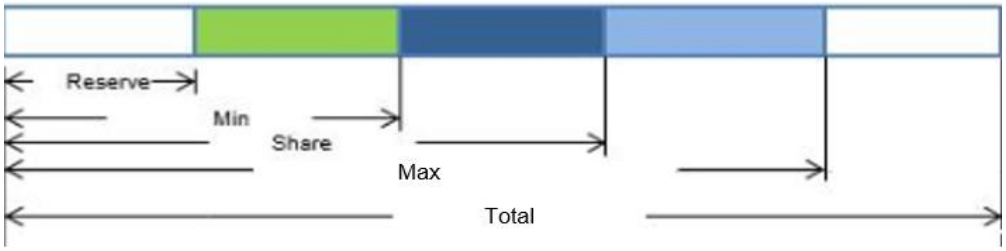
  Superior Scheduler provides flexible hierarchical multi-tenant scheduling policy. Different policies can be configured for different tenants or queues that can access different resource pools. The following figure lists supported policies:

Table 6-29 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.
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 6-137 shows the tenant resource allocation policy.

Figure 6-137 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 6-30 describes the comparison result of Superior Scheduler and Yarn open source schedulers.

Table 6-30 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**.

Scheduling	Yarn Open Source Scheduler	Superior Scheduler
Job waiting reason	Not supported	Job waiting reasons illustrate why a job needs to wait.

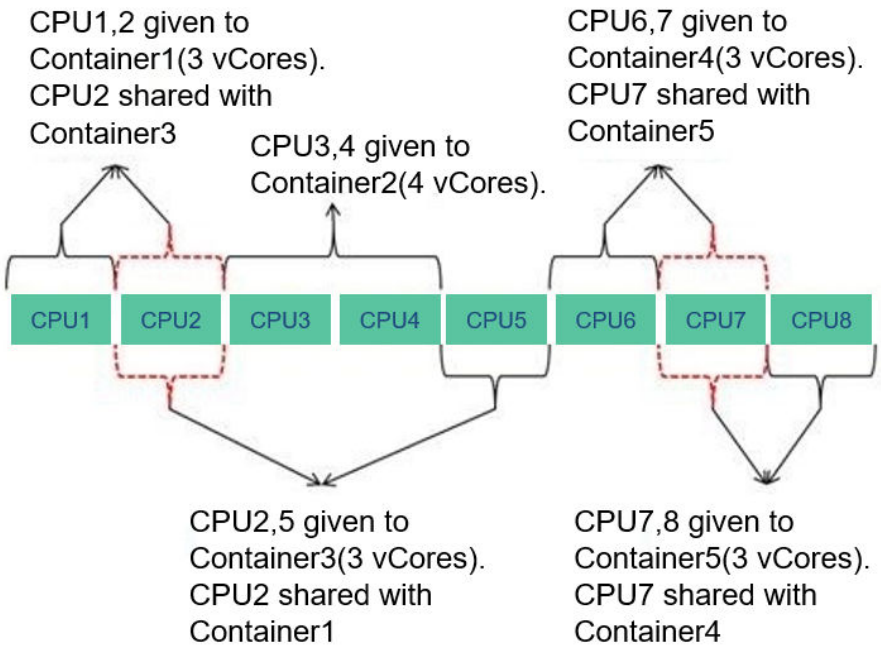
In conclusion, Superior Scheduler is a high-performance scheduler with various scheduling policies and is better than Capacity Scheduler in terms of functionality, performance, resource usage, and scalability.

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. In this case, 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 6-138 CPU quota



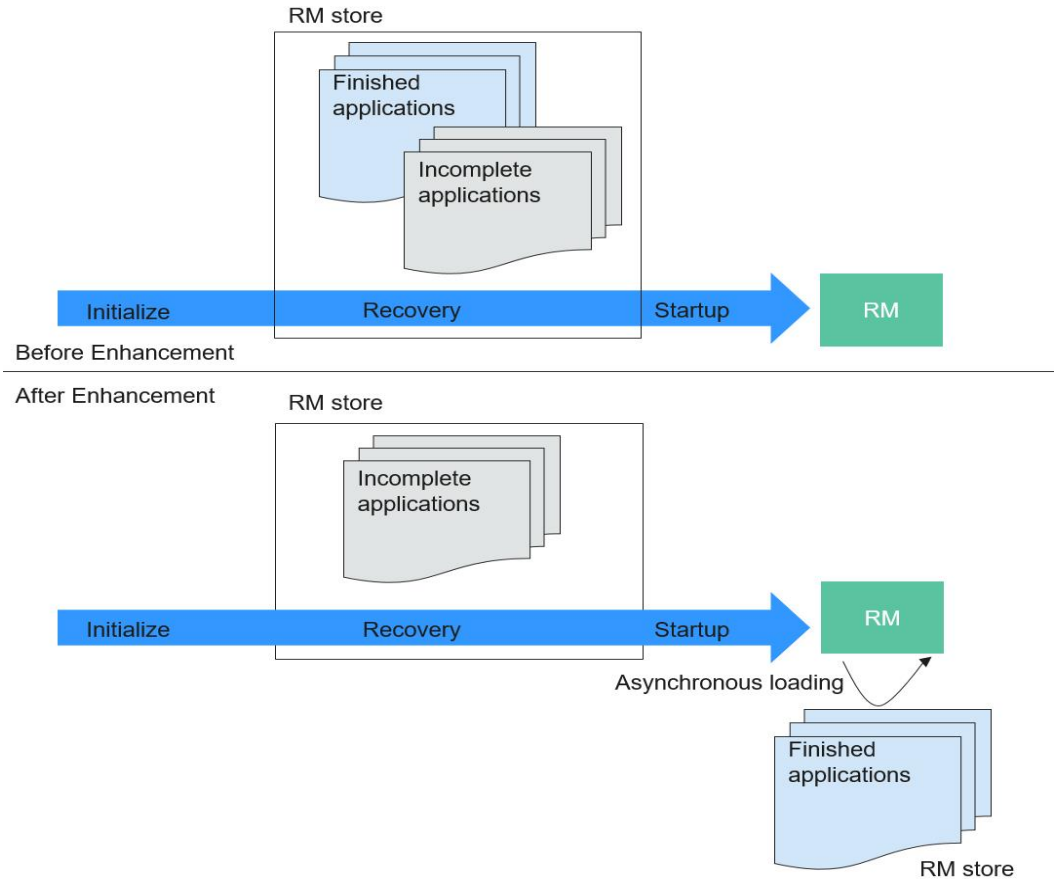
Enhanced Open Source Feature: Optimizing Restart Performance

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 6-139 Starting the ResourceManager recovery



6.32 ZooKeeper

6.32.1 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 6-140**. 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 6-140 Architecture

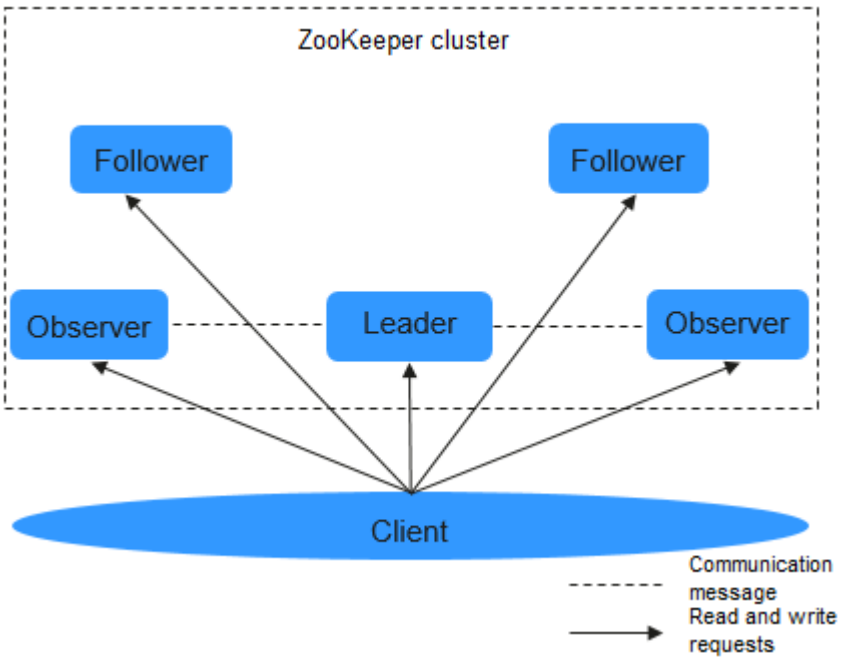


Table 6-31 describes the functions of each module shown in **Figure 6-140**.

Table 6-31 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.

Name	Description
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.

Typical Specifications

Typical Specifications lists the typical specifications of the ZooKeeper service.

Table 6-32 Typical specifications

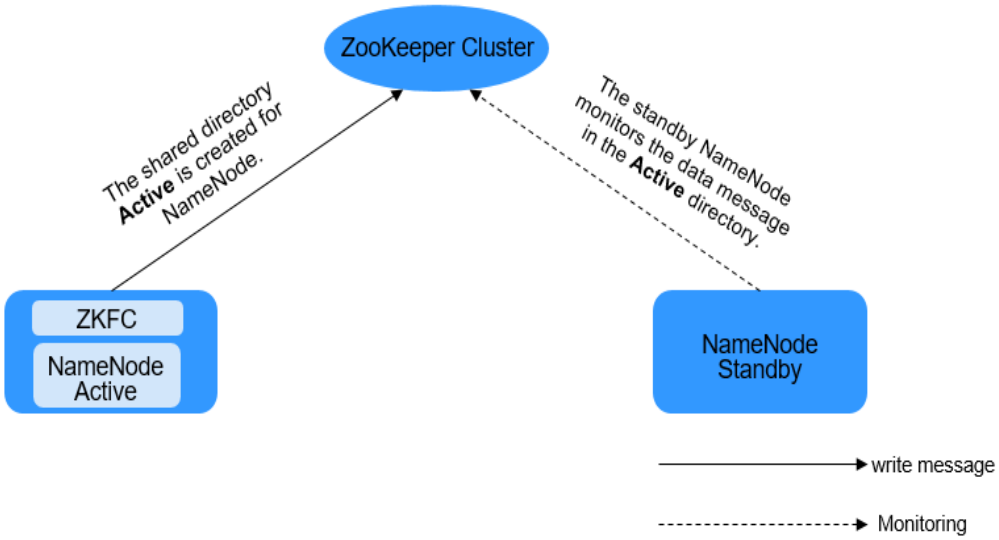
Specification	Value	Description
Maximum number of ZooKeeper instances in a cluster	9	
Maximum number of connections per IP address for each ZooKeeper instance	2000	-
Maximum number of connections for a ZooKeeper instance	20000	-
Maximum number of ZNodes in the case of default configurations	2000000	If there are too many ZNodes, the service will be unstable and the read and write performance of the component 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	-

6.32.2 Relationships Between ZooKeeper and Other Components

Hadoop Distributed File System (HDFS)

Figure 6-141 shows the relationship between ZooKeeper and HDFS.

Figure 6-141 Relationship between ZooKeeper and HDFS



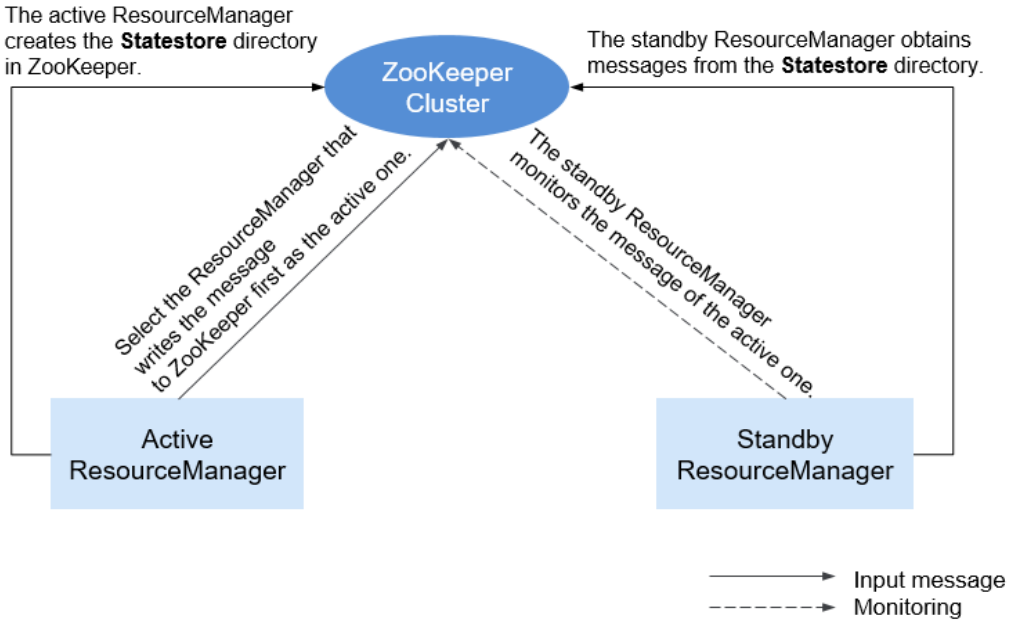
As the client of a ZooKeeper cluster, ZKFailoverController (ZKFC) monitors the status of NameNode. ZKFC is deployed only in the node where NameNode is deployed, and in both the active and standby HDFS NameNodes.

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.

YARN

Figure 6-142 shows the relationship between ZooKeeper and YARN.

Figure 6-142 Relationship Between ZooKeeper and YARN

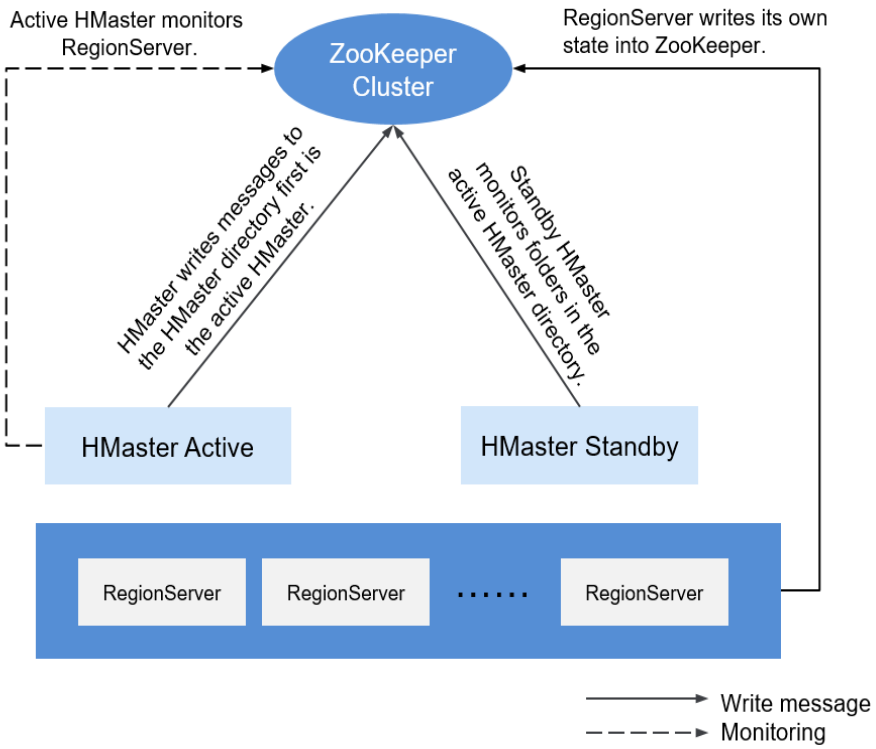


1. When the system is started, ResourceManager attempts to write state information to ZooKeeper. ResourceManager that first writes state information to ZooKeeper is selected as the active ResourceManager, and others are standby ResourceManagers. The standby ResourceManagers periodically monitor active 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.

HBase

Figure 6-143 shows the relationship between ZooKeeper and HBase.

Figure 6-143 Relationship between ZooKeeper and HBase

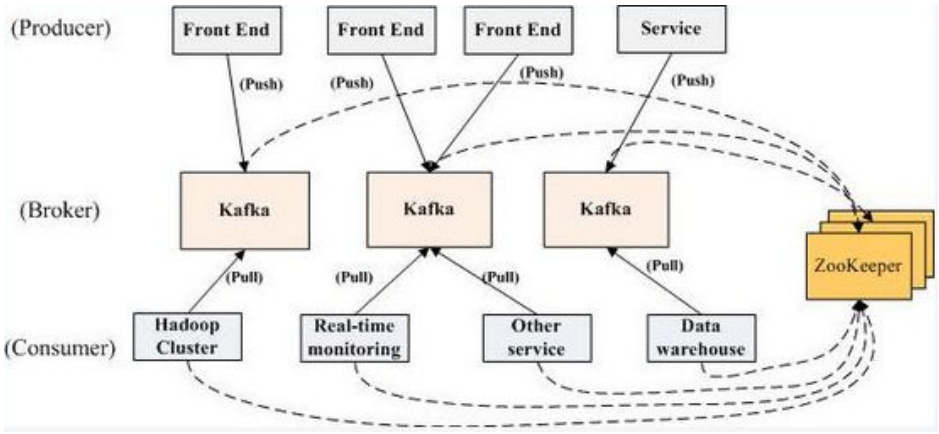


1. RegionServer registers itself to ZooKeeper on Ephemeral node. ZooKeeper stores the HBase information, including the HBase metadata and HMaster addresses.
2. HMaster detects the health status of each RegionServer using ZooKeeper, and monitors them.
3. HBase supports multiple HMaster nodes (like HDFS 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.

Kafka

Figure 6-144 shows the relationship between ZooKeeper and Kafka.

Figure 6-144 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.

6.32.3 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.

7 Functions

7.1 Multi-tenant

Feature Introduction

Modern enterprises' data clusters are developing towards centralization and cloudification. 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 (analysis, query, and stream processing).
- Isolate data of a user from that of another user who has high requirements on data security, such as a bank or government institute.

The preceding requirements bring the following challenges to the big data cluster:

- Proper allocation and scheduling of resources to ensure stable operating of applications and jobs
- Strict access control to ensure data and service security

Multi-tenant 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. Multi-tenant is a collection of multiple resources (each resource set is a tenant) in an MRS big data cluster. It can allocate and schedule resources, including computing and storage resources.

Advantages

- 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 recorded.

- Ensured 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.

Enhanced Schedulers

Schedulers are divided into the open source Capacity scheduler and Huawei proprietary Superior scheduler.

To meet enterprise requirements and tackle challenges facing the Yarn community in scheduling, Huawei develops the Superior scheduler. In addition to inheriting the advantages of the Capacity scheduler and Fair scheduler, this scheduler is enhanced in the following aspects:

- Enhanced resource sharing policy

  The Superior scheduler supports queue hierarchy. It integrates the functions of open source schedulers and shares resources based on configurable policies. In terms of instances, MRS cluster administrators can use the Superior scheduler to configure an absolute value or percentage policy for queue resources. The resource sharing policy of the Superior scheduler enhances the label scheduling policy of Yarn as a resource pool feature. The nodes in the Yarn cluster can be grouped based on the capacity or service type to ensure that queues can more efficiently utilize resources.

- Tenant-based resource reservation policy

  Resources required by tenants must be ensured for running critical tasks. The Superior scheduler builds a mechanism to support the resource reservation policy. By doing so, reserved resources can be allocated to the tasks run by the tenant queues in a timely manner to ensure proper task execution.

- Fair sharing among tenants and resource pool users

  The Superior scheduler allows shared resources to be configured for users in a queue. Each tenant may have users with different weights. Heavily weighted users may require more shared resources.

- Ensured scheduling performance in a big cluster

  The Superior scheduler receives heartbeats from each NodeManager and saves resource information in memory, which enables the scheduler to control cluster resource usage globally. The Superior scheduler uses the push scheduling model, which makes the scheduling more precise and efficient and remarkably improves cluster resource utilization. Additionally, the Superior scheduler delivers excellent performance when the interval between NodeManager heartbeats is long and prevents heartbeat storms in big clusters.

- Priority policy

  If the minimum resource requirement of a service cannot be met after the service obtains all available resources, a preemption occurs. The preemption function is disabled by default.

7.2 Security Hardening

MRS is a platform for massive data management and analysis and has high security. MRS protects user data and service running from the following aspects:

- Network isolation

  The entire system is deployed in a VPC on the public cloud to provide an isolated network environment and ensure service and management security of the cluster. By combining the subnet division, route control, and security group functions of VPC, MRS provides a secure and reliable isolated network environment.

- Resource isolation

  MRS supports resource deployment and isolation of physical resources in dedicated zones. You can flexibly combine computing and storage resources, such as dedicated computing resources + shared storage resources, shared computing resources + dedicated storage resources, and dedicated computing resources + dedicated storage resources.

- Host security

  MRS can be integrated with public cloud security services, including Vulnerability Scan Service (VSS), Host Security Service (HSS), Web Application Firewall (WAF), Cloud Bastion Host (CBH), and Web Tamper Protection (WTP). The following measures are provided by Huawei Cloud to improve security of the OS and ports:

  - Security hardening of OS kernels
  - OS permission control
  - OS port management

- Application security

  The following measures are used to ensure normal running of big data services:

  - Identification and authentication
  - Web application security
  - Access control
  - Audit security
  - Password security

- Data security

  The following measures are provided to ensure the confidentiality, integrity, and availability of massive amounts of user data:

  - Disaster recovery: MRS supports data backup to OBS and cross-region high reliability.
  - Backup: MRS supports backup of DBService, NameNode, and LDAP metadata and backup of HDFS and HBase service data.

- Data integrity

  Data is verified to ensure its integrity during storage and transmission.

  - CRC32C is used by default to verify the correctness of user data stored in HDFS.

  - DataNodes of HDFS store the verified data. If the data transmitted from a client is abnormal (incomplete), DataNodes report the abnormality to the client, and the client rewrites the data.
  - The client checks data integrity when reading data from a DataNode. If the data is incomplete, the client will read data from another DataNode.
- Data confidentiality

  Based on Apache Hadoop, the distributed file system of MRS supports encrypted storage of files to prevent sensitive data from being stored in plaintext, improving data security. Applications need only to encrypt specified sensitive data. Services are not affected during the encryption process. Based on file system data encryption, Hive provides table-level encryption and HBase provides column family-level encryption. Sensitive data can be encrypted and stored after you specify an encryption algorithm during table creation.

  Encrypted storage and access control of data are used to ensure user data security.

  - HBase stores service data to the HDFS after compression. Users can configure the AES and SMS4 encryption algorithm to encrypt data.
  - All the components allow access permissions to be set for local data directories. Unauthorized users are not allowed to access data.
  - All cluster user information is stored in ciphertext.
- Security authentication
  - Uses a unified user- and role-based authentication system as well as an account- and role-based access control (RBAC) model to centrally control user permissions and batch manage user authorization.
  - Employs Lightweight Directory Access Protocol (LDAP) as an account management system and performs the Kerberos authentication on accounts.
  - Provides the single sign-on (SSO) function that centrally manages and authenticates MRS system and component users.
  - Audits users who have logged in to Manager.

7.3 Easy Access to Web UIs of Components

Big data components have their own web UIs to manage their own systems. However, you cannot easily access the web UIs due to network isolation. For example, to access the HDFS web UI, you need to create an ECS to remotely log in to the web UI. This makes the UI access complex and unfriendly.

MRS provides an EIP-based secure channel for you to easily access the web UIs of components. This is more convenient than binding an EIP by yourself, and you can access the web UIs with a few clicks, avoiding the steps for logging in to a VPC, adding security group rules, and obtaining a public IP address. For the Hadoop, Spark, HBase, and Hue components in analysis clusters and the Storm component in streaming clusters, you can quickly access their web UIs from the entries on Manager.

7.4 Reliability Enhancement

Based on Apache Hadoop open source software, MRS optimizes and improves the reliability and performance of main service components.

System Reliability

- HA for all management nodes

  In the Hadoop open source version, data and compute nodes are managed in a distributed system, in which a single point of failure (SPOF) does not affect the operation of the entire system. However, a SPOF may occur on management nodes running in centralized mode, which becomes the weakness of the overall system reliability.

  MRS provides similar double-node mechanisms for all management nodes of the service components, such as Manager, HDFS NameNodes, HiveServers, HBase HMasters, Yarn ResourceManagers, KerberosServers, and LdapServers. All of them are deployed in active/standby mode or configured with load sharing, effectively preventing SPOFs from affecting system reliability.

- Reliability guarantee in case of exceptions

  By reliability analysis, the following measures to handle 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 checks 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 damages 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 clusters, including LdapServer and DBService data, can be automatically backed up.

  - Manual backup

    You can also manually back up data of the cluster management system before the capacity expansion and patch installation to recover the cluster management system functions upon faults.

To improve the system reliability, data on Manager and HBase is backed up to a third-party server manually.

Node Reliability

- OS health status monitoring

  MRS periodically collects OS hardware resource usage data, including usage of CPUs, memory, hard disks, and network resources.

- Process health status monitoring

  MRS checks the status of service instances and health indicators of service instance processes, enabling you to know the health status of processes in a timely manner.

- Automatic disk troubleshooting

  MRS is enhanced based on the open source version. It can monitor the status of hardware and file systems on all nodes. If an exception occurs, the corresponding partitions will be removed from the storage pool. If a disk is faulty and replaced, a new hard disk will be added for running services. In this case, maintenance operations are simplified. Replacement of faulty disks can be completed online. In addition, users can set hot backup disks to reduce the faulty disk restoration time and improve the system reliability.

- LVM configuration for node disks

  MRS allows you to configure Logic Volume Management (LVM) to plan multiple disks as a logical volume group. Configuring LVM can avoid uneven usage of disks. It is especially important to ensure even usage of disks on components that can use multiple disk capabilities, such as HDFS and Kafka. In addition, LVM supports disk capacity expansion without re-attaching, preventing service interruption.

Data Reliability

MRS can use the anti-affinity node groups provided by ECS and the rack awareness capability of Hadoop to redundantly distribute data to multiple physical host machines, preventing data loss caused by physical hardware failures.

7.5 Job Management

The job management function provides an entry for you to submit jobs in a cluster, including MapReduce, Spark, HQL, and SparkSQL jobs. MRS works with Huawei Cloud DataArts Studio to provide a one-stop big data collaboration development environment and fully-managed big data scheduling capabilities, helping you effortlessly build big data processing centers.

DataArts Studio allows you to develop and debug MRS HQL/SparkSQL scripts online and develop MRS jobs by performing drag-and-drop operations to migrate and integrate data between MRS and over 20 heterogeneous data sources. Powerful job scheduling and flexible monitoring and alarming help you easily manage data and job O&M.

7.6 Bootstrap Actions

Feature Introduction

MRS provides standard elastic big data clusters on the cloud. Big data components, such as Hadoop and Spark, can be installed and deployed. Currently, standard cloud big data clusters cannot meet all user requirements, for example, in the following scenarios:

- Common operating system configurations cannot meet data processing requirements, for example, increasing the maximum number of system connections.
- Software tools or running environments need to be installed, for example, Gradle and dependency R language package.
- Big data component packages need to be modified based on service requirements, for example, modifying the Hadoop or Spark installation package.
- Other big data components that are not supported by MRS need to be installed.

To meet the preceding customization requirements, you can manually perform operations on the existing and newly added nodes. The overall process is complex and error-prone. In addition, manual operations cannot be traced, and data cannot be processed immediately after creating a cluster based on your demand.

Therefore, MRS supports custom bootstrap actions that enable you to run scripts on a specified node before or after a cluster component is started. You can run bootstrap actions to install third-party software that is not supported by MRS, modify the cluster running environment, and perform other customizations. If you choose to run bootstrap actions when expanding a cluster, the bootstrap actions will be run on the newly added nodes in the same way. MRS runs the script you specify as user **root**. You can run the **su - *xxx*** command in the script to switch the user.

Customer Benefits

You can use the custom bootstrap actions to flexibly and easily configure your dedicated clusters and customize software installation.

7.7 Enterprise Project Management

An enterprise project is a cloud resource management mode. Enterprise Management provides users with comprehensive management of cloud-based resources, personnel, permissions, and finances. Common management consoles are oriented to the control and configuration of individual cloud products. The Enterprise Management console, in contrast, is more focused on resource management. It is designed to help enterprises manage cloud-based resources, personnel, permissions, and finances, in a hierarchical management manner, such as management of companies, departments, and projects.

MRS allows users who have enabled Enterprise Project Management Service (EPS) to configure enterprise projects for a cluster during cluster creation and use EPS to manage MRS resources by group.

- The users can manage multiple resources by group.
- The users can view resource information and expenditure details of enterprise projects.
- The users can control access permissions at the enterprise project level.
- The users can view detailed financial information by enterprise project, including orders, expenditure summary, and expenditure details.

NOTE

If the MRS cluster and VPC are not in the same enterprise project, you need to add the VPC view permission in the IAM view to view VPC and cluster information.

7.8 Metadata

MRS provides multiple metadata storage methods. When deploying Hive and Ranger during MRS cluster creation, select one of the following storage modes as required:

- **Local**: Metadata is stored in the local GaussDB of a cluster. When the cluster is deleted, the metadata is also deleted. To retain the metadata, manually back up the metadata in the database in advance.
- **External data connection**: After the cluster is created, you can select **RDS PostgreSQL database** or **RDS MySQL database** that is associated with the same VPC and subnet as the current cluster. Metadata is stored in the associated database and is not deleted when the current cluster is deleted. Multiple MRS clusters can share the same metadata.

NOTE

Hive in MRS 1.9.*x* or later allows you to specify a metadata storage method.

7.9 Cluster Management

7.9.1 Cluster Lifecycle Management

MRS supports cluster lifecycle management, including creating and terminating clusters.

- Creating a cluster: After you specify a cluster type, components, number of nodes of each type, VM specifications, AZ, VPC, and authentication information, MRS automatically creates a cluster that meets the configuration requirements. You can run customized scripts in the cluster. In addition, you can create clusters of different types for multiple application scenarios, such as Hadoop analysis clusters, HBase clusters, and Kafka clusters. The big data platform supports heterogeneous cluster deployment. That is, VMs of different specifications can be combined in a cluster based on CPU types, disk capacities, disk types, and memory sizes. Various VM specifications can be mixed in a cluster.

- Terminating a cluster: You can terminate a pay-per-use cluster that is no longer needed (including data and configurations in the cluster). MRS will delete all resources related to the cluster.
- Renewal: MRS provides two billing modes: pay-per-use and yearly/monthly. In pay-per-use mode, fees are deducted every hour and insufficient balance can lead to overdue payments. In yearly/monthly mode, clusters need to be renewed before they expire. If your subscription for the pay-per-use or yearly/monthly cluster is not renewed, your services will keep running, but enter into a retention period, during which the MRS clusters will stop running but data is retained.
- Unsubscription: If you have purchased a yearly/monthly cluster and do not need the cluster resources before the cluster resources expire, you can unsubscribe from the cluster resources on MRS.

Buying a Cluster

On the MRS management console, you can buy an MRS cluster on a pay-per-use or yearly/monthly basis and select a region and cloud resource specifications that are suitable for your company with a few clicks. MRS automatically installs and deploys the Huawei Cloud enterprise-level big data platform and optimizes parameters based on the selected cluster type, version, and node specifications.

MRS provides you with fully managed big data clusters. When creating a cluster, you can set a VM login mode (password or key pair). You can use all resources of the created MRS cluster. In addition, MRS allows you to deploy a big data cluster on two ECSs with 4 vCPUs and 8 GB memory, providing more flexible choices for testing and development.

MRS clusters are classified into analysis, streaming, and hybrid clusters.

- Analysis cluster: is used for offline data analysis and provides Hadoop components.
- Streaming cluster: is used for streaming tasks and provides stream processing components.
- Hybrid cluster: is used for not only offline data analysis but also streaming processing, and provides Hadoop components and stream processing components.
- Custom: You can flexibly combine required components (MRS 3.x and later versions) based on service requirements.

MRS cluster nodes are classified into Master, Core, and Task nodes.

- Master nodes are management nodes in a cluster. The Master process, Manager, and databases of the distributed system are deployed on these nodes. Master nodes cannot be expanded. The processing capability of Master nodes determines the upper limit of the management capability of the entire cluster. MRS supports scale-up of Master node specifications to provide support for management of a larger cluster.
- Core node: used for both storage and computing and can be scaled in or out. Since Core nodes bear data storage, there are many restrictions on scale-in to prevent data loss and auto scaling cannot be performed.
- Task node: used only for computing only and can be scaled in or out. Task nodes bear only computing tasks. Therefore, auto scaling can be performed.

You can buy a cluster in two modes: custom configuration and quick configuration.

- **Custom Config**: On the **Custom Config** page, you can flexibly configure cluster parameters based on application scenarios, such as the billing mode and ECS specifications to better suit your service requirements.
- **Quick Config**: On the **Quick Config** page, you can quickly buy a cluster based on application scenarios, improving cluster configuration efficiency. Currently, Hadoop analysis clusters, HBase clusters, Kafka clusters, ClickHouse clusters, and real-time analysis clusters are supported.
  - Hadoop analysis cluster: uses components in the open-source Hadoop ecosystem to analyze and query vast amounts of data. For example, use Yarn to manage cluster resources, Hive and Spark to provide offline storage and computing of large-scale distributed data, Spark Streaming and Flink to offer streaming data computing, and Presto to enable interactive queries, and Tez to provide a distributed computing framework of directed acyclic graphs (DAGs).
  - HBase cluster: uses Hadoop and HBase components to provide a column-oriented distributed cloud storage system featuring enhanced reliability, great performance, and elastic scalability. It applies to the storage and distributed computing of massive amounts of data. You can use HBase to build a storage system capable of storing TB- or even PB-level data. With HBase, you can filter and analyze data with ease and get responses in milliseconds, rapidly mining data value.
  - Kafka cluster: uses Kafka and Storm to provide an open source message system with high throughput and scalability. It is widely used in scenarios such as log collection and monitoring data aggregation to implement efficient streaming data collection and real-time data processing and storage.
  - ClickHouse cluster: ClickHouse is a columnar database management system used for online analysis. It features the optimal compression rate and fast query performance. It is widely used in Internet advertisement, app and web traffic analysis, telecom, finance, and IoT fields.
  - Real-time analysis clusters: uses Hadoop, Kafka, Flink, and ClickHouse components to provide a system for collection, real-time analysis, and query of data at scale.

Terminating a Cluster

MRS allows you to terminate a cluster when it is no longer needed. After the cluster is terminated, all cloud resources used by the cluster will be released. Before terminating a cluster, you are advised to migrate or back up data. Terminate the cluster only when no service is running in the cluster or the cluster is abnormal and cannot provide services based on O&M analysis. If data is stored on EVS disks or pass-through disks in a big data cluster, the data will be deleted after the cluster is terminated. Therefore, exercise caution when terminating a cluster.

7.9.2 Cluster Scaling

The processing capability of a big data cluster can be horizontally expanded by adding nodes. If the cluster scale does not meet service requirements, you can

manually scale out or scale in the cluster. MRS intelligently selects the node with the least load or the minimum amount of data to be migrated for scale-in. The node to be scaled in will not receive new tasks, and continues to execute the existing tasks. At the same time, MRS copies its data to other nodes and the node is decommissioned. If the tasks on the node cannot be completed after a long time, MRS migrates the tasks to other nodes, minimizing the impact on cluster services.

Scaling Out a Cluster

Currently, you can add Core or Task nodes to scale out a cluster to handle peak service loads. Adding MRS cluster nodes does not affect the services of the existing cluster. For details about how to rectify data skew caused by capacity expansion, see **Configuring HDFS DataNode Data Balancing**.

Scaling Out a Cluster Charged in Yearly/Monthly Mode

If your service growth rate exceeds the expected value after you subscribe to an MRS cluster charged in **Yearly/Monthly** mode, cluster scale-out beyond your subscription is required. MRS allows you to scale out clusters charged in **Yearly/Monthly** mode while enjoying the subscription discounts.

You only need to go to the MRS console page and expand the number of nodes you need. The cluster scale-out process does not require manual intervention and takes only a few minutes, which helps ease pressure on growing service data processing needs.

Scaling In a Cluster

You can reduce the number of Core or Task nodes to scale in a cluster so that MRS delivers better storage and computing capabilities at lower O&M costs based on service requirements. After you scale in an MRS cluster, MRS automatically selects nodes that can be scaled in based on the type of services installed on the nodes.

During the scale-in of Core nodes, data on the original nodes is migrated. If the data location is cached, the client automatically updates the location information, which may affect the latency. Node scale-in may affect the response duration of the first access to some HBase on HDFS data. You can restart HBase or disable or enable related tables to avoid this problem.

Task nodes do not store cluster data. They are compute nodes and do not involve migration of data on the nodes.

7.9.3 Auto Scaling

Feature Introduction

More and more enterprises use technologies such as Spark and Hive to analyze data. Processing a large amount of data consumes huge resources and costs much. Typically, enterprises regularly analyze data in a fixed period of time every day rather than all day long. To meet enterprises' requirements, MRS provides the auto scaling function to apply for extra resources during peak hours and release resources during off-peak hours. This enables users to use resources on demand and focus on core business at lower costs.

In big data applications, especially in periodic data analysis and processing scenarios, cluster computing resources need to be dynamically adjusted based on service data changes to meet service requirements. The auto scaling function of MRS enables clusters to be elastically scaled out or in based on cluster loads. In addition, if the data volume changes regularly and you want to scale out or in a cluster before the data volume changes, you can use the MRS resource plan feature.

MRS supports two types of auto scaling policies: auto scaling rules and resource plans

- Auto scaling rules: You can increase or decrease Task nodes based on real-time cluster loads. Auto scaling will be triggered when the data volume changes but there may be some delay.
- Resource plans: If the data volume changes periodically, you can create resource plans to resize the cluster before the data volume changes, thereby avoiding a delay in increasing or decreasing resources.

Both auto scaling rules and resource plans can trigger auto scaling. You can configure both of them or configure one of them. Configuring both resource plans and auto scaling rules improves the cluster node scalability to cope with occasionally unexpected data volume peaks.

In some service scenarios, resources need to be reallocated or service logic needs to be modified after cluster scale-out or scale-in. If you manually scale a cluster, you can log in to cluster nodes to reallocate resources or modify service logic. If you use auto scaling, MRS enables you to customize automation scripts for resource reallocation and service logic modification. Automation scripts can be executed before and after auto scaling and automatically adapt to service load changes, all of which eliminates manual operations. In addition, automation scripts can be fully customized and executed at various moments, which can meet your personalized requirements and improve auto scaling flexibility.

Customer Benefits

MRS auto scaling provides the following benefits:

- Reducing costs

  Enterprises may not analyze data in batches all the time but perform a batch analysis task in a specified period of time, for example, 03:00 a.m. The task may take only two hours.

  The auto scaling function allows you to add nodes for batch analysis and automatically releases the nodes after completion of the analysis, minimizing costs.
- Meeting instant query requirements

  Enterprises usually encounter instant analysis tasks, for example, data reports for supporting enterprise decision-making. As a result, resource consumption increases sharply in a short period of time. With the auto scaling function, compute resources can be added for emergent big data analysis, avoiding service breakdown due to insufficient compute nodes. After the service spike ends and you do not need additional resources, MRS schedules and performs a scale-in.
- Focusing on core business

It is difficult for developers to determine resource consumption on the big data secondary development platform because of complex query analysis conditions (such as global sorting, filtering, and merging) and data complexity, for example, uncertainty of incremental data. As a result, estimating the computing volume is difficult. MRS's auto scaling function enable developers to focus on service development without the need for resource estimation.

7.9.4 Task Node Creation

Feature Introduction

Task nodes can be created and used for computing only. They do not store persistent data and are the basis for implementing auto scaling.

Customer Benefits

When MRS is used only as a computing resource, Task nodes can be used to reduce costs and facilitate cluster node scaling, flexibly meeting users' requirements for increasing or decreasing cluster computing capabilities.

Application Scenarios

If the cluster data volume is stable but the processing demand increase greatly, you can add Task nodes to the cluster to handle more service requests.

- The number of temporary services is increased, for example, report processing at the end of the year.
- A large number of tasks, for example, emergent analysis tasks, need to be processed in a short period.

7.9.5 Isolating a Host

When detecting that a host is abnormal or faulty and cannot provide services or affects cluster performance, you can exclude the host from the available nodes in the cluster temporarily so that the client can access other available nodes. In scenarios where patches are to be installed in a cluster, you can also exclude a specified node from patch installation. Only non-management nodes can be isolated.

After a host is isolated, all role instances on the host will be stopped, and you cannot start, stop, or configure the host and all instances on the host. In addition, after a host is isolated, statistics about the monitoring status and metric data of hardware and instances on the host cannot be collected or displayed.

7.9.6 Managing Tags

Tags are cluster/node identifiers. Adding tags to clusters/nodes can help you identify and manage your cluster/node resources. By associating with Tag Management Service (TMS), MRS allows users with a large number of cloud resources to tag cloud resources, quickly search for cloud resources with the same tag attribute, and perform unified management operations such as review, modification, and deletion, facilitating unified management of big data clusters and other cloud resources.

You can add a maximum of 10 tags to a cluster when creating the cluster or add them on the details page of the created cluster.

You can add tags to a node only after the node is created. You can add a maximum of 10 tags to a node.

7.10 Cluster O&M

Alarm Management

MRS can monitor big data clusters in real time and identify system health status based on alarms and events. In addition, MRS allows you to customize monitoring and alarm thresholds to focus on the health status of each metric. When monitoring data reaches the alarm threshold, the system triggers an alarm.

MRS can also interconnect with the message service system of the Huawei Cloud Simple Message Notification (SMN) service to push alarm information to users by SMS message or email. For details, see **Message Notification**.

Patch Management

MRS supports cluster patching operations and will release patches for open source big data components in a timely manner. On the MRS cluster management page, you can view patch release information related to running clusters, including the detailed description of the resolved issues and impacts. You can determine whether to install a patch based on the service running status. One-click patch installation involves no manual intervention, and will not cause service interruption through rolling installation, ensuring long-term availability of the clusters.

MRS can display the detailed patch installation process, and supports patch uninstallation and rollback.

O&M Support

Cluster resources provided by MRS belong to users. Generally, when O&M personnel's support is required for troubleshooting of a cluster, O&M personnel cannot directly access the cluster. To better serve customers, MRS provides the following two methods to improve communication efficiency during fault locating:

- Log sharing: You can initiate log sharing on the MRS management console to share a specified log scope with O&M personnel, so that O&M personnel can locate faults without accessing the cluster.
- O&M authorization: If a problem occurs when you use an MRS cluster, you can initiate O&M authorization on the MRS management console. O&M personnel can help you quickly locate the problem, and you can revoke the authorization at any time.

Health Check

MRS provides automatic inspection on system running environments for you to check and audit system running health status in one click, ensuring proper system

running and lowering system operation and maintenance costs. After viewing inspection results, you can export reports for archiving and fault analysis.

7.11 Message Notification

Feature Introduction

The following operations are often performed during the running of a big data cluster:

- Big data clusters often change, for example, cluster scale-out and scale-in.
- When a service data volume changes abruptly, auto scaling will be triggered.
- After related services are stopped, a big data cluster needs to be stopped.

To immediately notify you of successful operations, cluster unavailability, and node faults, MRS uses Simple Message Notification (SMN) to send notifications to you through SMS and emails, facilitating maintenance.

Customer Benefits

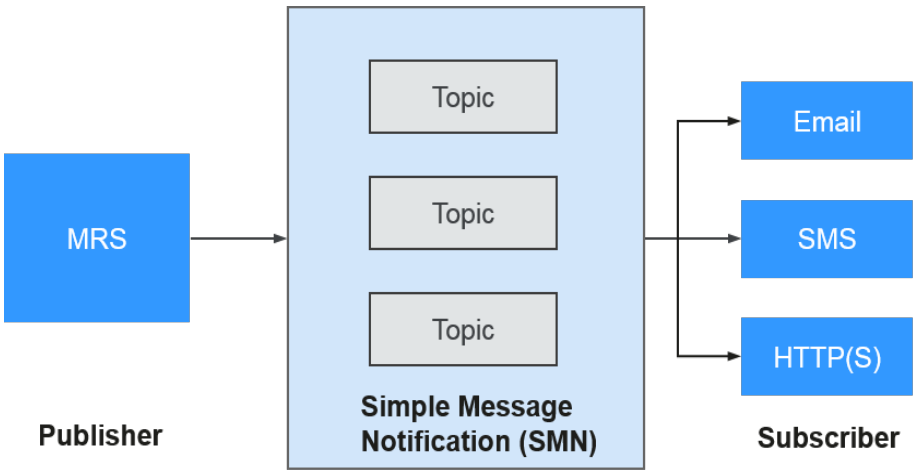
After configuring SMN, you can receive MRS cluster health status, updates, and component alarms through SMS or emails in real time. MRS sends real-time monitoring and alarm notification to help you easily perform O&M and efficiently deploy big data services.

Feature Description

MRS uses SMN to provide one-to-multiple message subscription and notification over a variety of protocols.

You can create a topic and configure topic policies to control publisher and subscriber permissions on the topic. MRS sends cluster messages to the topic to which you have permission to publish messages. Then, all subscribers who subscribe to the topic can receive cluster updates and component alarms through SMS and emails.

Figure 7-1 Implementation process



8 Constraints

Before using MRS, ensure that you have read and understand the following restrictions.

- MRS clusters must be created in VPC subnets.
- You are advised to use any of the following browsers to access MRS:
  - Google Chrome: 36.0 or later
  - Internet Explorer: 9.0 or later

    If you use Internet Explorer 9.0, you may fail to log in to the MRS management console because user **Administrator** is disabled by default in some systems. The browser automatically selects a system user for installation. As a result, the browser cannot access the management console. Reinstall the browser or a later version (recommended) or run the browser as user **Administrator**.
  - Microsoft Edge is updated with the Windows operating system.
- When you create an MRS cluster, you can select **Auto create** from **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 on the ECS console, changing or reinstalling their OS, or modifying their specifications.
  - Deleting the existing processes, applications or files on cluster nodes.
  - Deleting MRS cluster nodes. Clusters with deleted nodes are still be charged.
- If a cluster exception occurs when no incorrect operations have been performed, contact technical support engineers. They will ask you for your password and then perform troubleshooting.
- 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.

- MRS clusters are still charged during exceptions. Contact technical support engineers to handle cluster exceptions.
- 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.
- The capacity (including storage and computing capabilities) of an MRS cluster can be expanded by adding core or task nodes.
- If a Master node in an MRS cluster is shut down and the cluster is still used to execute jobs or modify component configurations, you must start the stopped Master node before stopping other nodes. Otherwise, data may be lost due to an active/standby switchover.
- If all nodes in an MRS cluster have been stopped, start them in the reverse order of node shutdown.
- The Capacity and Superior scheduler switchover is complete when the MRS cluster is used, while configuration synchronization is not complete. Configure synchronization again based on the new scheduler if necessary.

9 Technical Support

MRS is a one-stop big data platform that provides enterprise-class clusters on the cloud. Tenants can fully control clusters and easily run big data components such as Hadoop, Hive, Spark, HBase, Kafka, and Flink. In addition, MRS helps enterprises quickly build a system to process massive amounts of data and discover new value and business opportunities in real time or in non-real time.

Maintenance Policy Statement

MRS provides tenants with fully controllable clusters and semi-hosting cloud services. By default, cloud services do not have permissions to perform operations on the clusters. Tenants are responsible for routine cluster O&M and management. They can contact the technical support team for help if any technical issues occur, excluding those not related to MRS, for example, how to build an application system based on the big data platform.

Technical Support Scope

- Supported services
  - The MRS console provides the following functions:
    - Creating, deleting, scaling out, and scaling in clusters
    - Managing cluster jobs
    - Managing cluster alarms
    - Managing cluster patches
    - Managing IAM users
    - Managing external APIs
  - MRS provides open-source big data components.
  - MRS supports vulnerability analysis of open-source components, such as impact analysis and fixing suggestions. It enables tenants to evaluate the impact of vulnerabilities on services and fix the vulnerabilities.
- Services not supported
  - Huawei is not responsible for providing O&M operations for MRS clusters and open-source big data components, including configuration

modification, restart, capacity planning, component performance optimization, and any O&M operations on clusters.

- Huawei is not responsible for answering and handling questions about application development on MRS clusters, such as service design, coding, job performance optimization, and workload migration.
- If the MRS cluster component service has no exception or quality defect, Huawei is not responsible for troubleshooting and analyzing running exceptions of your big data jobs.
- Huawei is not responsible for analyzing or resolving unexpected problems caused by any non-standard operations on MRS clusters. Common high-risk operations include reinstalling the OS, deleting data by mistake, deleting service directories and files, modifying OS system configurations and file permissions, deleting **/etc/hosts** configurations, uninstalling disks in the background, changing node IP addresses, and deleting default security group rules. For more information about these operations, see **High-Risk Operations**.
- Huawei is not responsible for troubleshooting and resolving problems of third-party components that are not provided by MRS and installed by users of the MRS cluster environment.

10 Billing

MRS billing is simple and predictable. MapReduce Service (MRS) billing is simple and predictable and supports a pay-per-use basis. You can also select a yearly or monthly package depending on what is more economical. The total price of an MRS cluster will be automatically calculated so that you can purchase a cluster in few clicks.

Billing Items

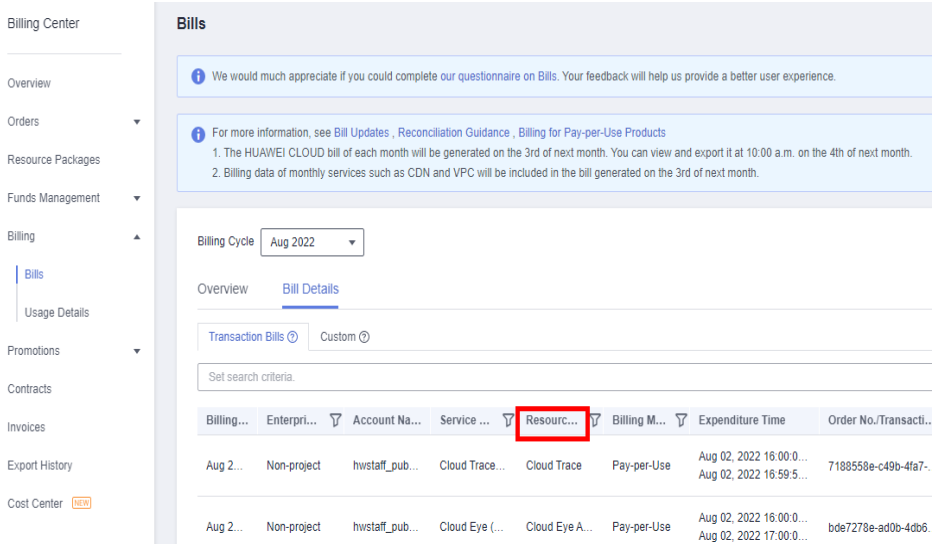
The price of an MRS cluster consists of two parts:

- MRS management fee

NOTE

You can view detailed MRS management fee by logging in to the Billing Center, choosing **Billing** > **Bills**, and filtering management fees.

Figure 10-1 Viewing MRS management fee



  - If **Cluster Type** is **LTS**, filter management fees by **MRS-LTS Service Fee**.
  - If **Cluster Type** is **Normal**, filter management fees by:
    - **MapReduce Service VM** for clusters purchased in June 2022 or earlier
    - **MRS-BASIC Service Fee** for clusters purchased after June 2022
- Fees of IaaS infrastructure resources, including Elastic Cloud Server (ECS), Elastic Volume Service (EVS), elastic IP (EIP), and bandwidth

For details about the MRS management fee, see **Product Pricing Details**.

You can use the **price calculator** of MRS to quickly obtain an estimate price of a cluster with the specifications you select.

The terminated or unsubscribed MRS cluster is no longer billed.

Billing Modes

Before using MRS, you must purchase an MRS cluster.

- Yearly/Monthly: You can pay for clusters by year or month. The minimum duration is 1 month and the maximum duration is 1 year.
- Pay-per-use: Nodes are billed by actual duration of use, with a billing cycle of one hour.

Changing Billing Mode

Before subscribing to MRS, choose Master and Core node instances that best fit your needs. MRS provides the following methods for you to change cluster configuration after a cluster is started.

- **Configure Task Node**: Add Task nodes. For details, see **Related Operations** in **Manually Scaling Out a Cluster**.

- **Scale Out**: Manually add Core or Task nodes. For details, see **Manually Scaling Out a Cluster**.
- **Auto Scaling**: The number of nodes in a cluster can be automatically adjusted based on the service data volume to increase or decrease resources. For details, see **Configuring Auto Scaling Rules**.

If the configuration change methods provided by MRS do not meet your requirements, you can create a cluster again and migrate data to the cluster to realize cluster configuration change.

Renewal

To renew the subscription, go to the **Renewals** page.

Overdue Payment

Overdue payment does not apply to yearly or monthly subscribed clusters.

In pay-per-use mode, cluster fees are deducted every hour. If your account balance is insufficient to pay for the expense occurred in the last hour, your account will be in arrears, and MRS clusters have a **retention period**. If the clusters are renewed within the retention period, they will be available and charged from the original expiration date.

You are advised to renew the cluster as soon as possible if your cluster is in arrears. Otherwise, the following operations are restricted:

- Creating Clusters
- Scaling out a cluster
- Scaling in a cluster
- Adding a Task node
- Scaling up Master node specifications

Expiration

- Expiration does not apply to pay-per-use clusters.
- If your yearly or monthly subscription expires, the cluster will enter into a **retention period**. During the grace period and retention period, you cannot perform operations on the cluster on the MRS management console, related APIs cannot be called, and O&M operations such as automatic monitoring and alarm reporting will be stopped. If your subscription is not renewed at the end of the retention period, services in the cluster will be terminated and data in the system will be deleted permanently.

Security Deposits

When you purchase a pay-per-use cluster, Huawei Cloud may freeze a prepaid balance based on the user level and historical usage. The deposit is automatically unfrozen when resources are released.

11 Permissions Management

If you need to assign different permissions to employees in your enterprise to access your MRS resources on Huawei Cloud, IAM is a good choice for fine-grained permissions management. IAM provides identity authentication, permissions management, and access control, helping you secure access to your Huawei Cloud resources.

With IAM, you can create IAM users under your Huawei Cloud account, and assign permissions to these users to control their access to specific resources. For example, some software developers in your enterprise need to use MRS resources but must not delete MRS clusters or perform any high-risk operations. To achieve this goal, you can create IAM users for the software developers and grant them only the permissions required for using MRS cluster resources.

If your Huawei Cloud account does not require individual IAM users for permissions management, skip this section.

IAM is free of charge. You pay only for the resources you use. For more information about IAM, see **IAM Service Overview**.

MRS Permission Description

By default, new IAM users do not have any permissions. To assign permissions to a user, add the user to one or more groups and assign permissions policies or roles to these groups. The user then inherits permissions from the groups it is a member of and can perform specified operations on cloud services based on the permissions.

MRS is a project-level service deployed and accessed in specific physical regions. To assign permissions to a user group, specify **Scope** as **Region-specific projects** and select projects in the corresponding region for the permissions to take effect. If **All projects** is selected, the permissions will take effect for the user group in all region-specific projects. When accessing MRS, the users need to switch to a region where they have been authorized to use the MRS service.

You can grant permissions by using roles and policies.

- Roles: A type of coarse-grained authorization mechanism that defines permissions related to user responsibilities. This mechanism provides only a limited number of service-level roles for authorization. When using roles to grant permissions, you need to also assign other roles on which the

permissions depend to take effect. However, roles are not an ideal choice for fine-grained authorization and secure access control.

- Policies: A type of fine-grained authorization mechanism that defines permissions required to perform operations on specific cloud resources under certain conditions. This mechanism allows for more flexible policy-based authorization, meeting requirements for secure access control. For example, you can grant MRS users only the permissions for performing specified operations on MRS clusters, such as creating a cluster and querying a cluster list rather than deleting a cluster. Most policies define permissions based on APIs. For the API actions supported by MRS, see **Permissions Policies and Supported Actions**.

Table 11-1 lists all the default system policies supported by MRS.

Table 11-1 MRS system policies

Policy	Description	Type
MRS FullAccess	Administrator permissions for MRS. Users granted these permissions can operate and use all MRS resources.	Fine-grained policy
MRS CommonOperations	Common user permissions for MRS. Users granted these permissions can use MRS but cannot add or delete resources.	Fine-grained policy
MRS ReadOnlyAccess	Read-only permission for MRS. Users granted these permissions can only view MRS resources.	Fine-grained policy
MRS Administrator	Permissions: • All operations on MRS • Users with permissions of this policy must also be granted permissions of the **Tenant Guest** and **Server Administrator** policies.	RBAC policy

Table 11-2 lists the common operations supported by each system-defined policy or role of MRS. Select the policies or roles as required.

Table 11-2 Common operations supported by each system-defined policy

Operation	MRS FullAccess	MRS CommonOperations	MRS ReadOnlyAccess	MRS Administrator
Creating a cluster	√	x	x	√

Operation	MRS FullAccess	MRS CommonOperations	MRS ReadOnlyAccess	MRS Administrator
Resizing a cluster	√	x	x	√
Upgrading node specifications	√	x	x	√
Deleting a cluster	√	x	x	√
Querying cluster details	√	√	√	√
Querying a cluster list	√	√	√	√
Configuring an auto scaling rule	√	x	x	√
Querying a host list	√	√	√	√
Querying operation logs	√	√	√	√
Creating and executing a job	√	√	x	√
Stopping a job	√	√	x	√
Deleting a single job	√	√	x	√
Deleting jobs in batches	√	√	x	√
Querying job details	√	√	√	√
Querying a job list	√	√	√	√
Creating a folder	√	√	x	√

Operation	MRS FullAccess	MRS CommonOperations	MRS ReadOnlyAccess	MRS Administrator
Deleting a file	√	√	x	√
Querying a file list	√	√	√	√
Operating cluster tags in batches	√	√	x	√
Creating a single cluster tag	√	√	x	√
Deleting a single cluster tag	√	√	x	√
Querying a resource list by tag	√	√	√	√
Querying cluster tags	√	√	√	√
Accessing Manager	√	√	x	√
Querying a patch list	√	√	√	√
Installing a patch	√	√	x	√
Uninstalling a patch	√	√	x	√
Authorizing O&M channels	√	√	x	√
Sharing O&M channel logs	√	√	x	√
Querying an alarm list	√	√	√	√

Operation	MRS FullAccess	MRS CommonOperations	MRS ReadOnlyAccess	MRS Administrator
Subscribing to alarm notification	√	√	x	√
Submitting an SQL statement	√	√	x	√
Querying SQL results	√	√	x	√
Canceling an SQL execution task	√	√	x	√

MRS FullAccess Policy

```
{
   "Version": "1.1",
   "Statement": [
      {
         "Action": [
            "mrs:*:*",
            "ecs:*:*",
            "bms:*:*",
            "evs:*:*",
            "vpc:*:*",
            "kms:*:*",
            "rds:*:*",
            "bss:*:*"
         ],
         "Effect": "Allow"
      }
   ]
}
```

MRS CommonOperations Policy

```
{
   "Version": "1.1",
   "Statement": [
      {
         "Action": [
            "mrs:*:get*",
            "mrs:*:list*",
            "ecs:*:get*",
            "ecs:*:list*",
            "bms:*:get*",
            "bms:*:list*",
            "evs:*:get*",
            "evs:*:list*",
```



```
            "vpc:*:get*",
            "vpc:*:list*",
            "mrs:job:submit",
            "mrs:job:stop",
            "mrs:job:delete",
            "mrs:job:checkSql",
            "mrs:job:batchDelete",
            "mrs:file:create",
            "mrs:file:delete",
            "mrs:tag:batchOperate",
            "mrs:tag:create",
            "mrs:tag:delete",
            "mrs:manager:access",
            "mrs:patch:install",
            "mrs:patch:uninstall",
            "mrs:ops:grant",
            "mrs:ops:shareLog",
            "mrs:alarm:subscribe",
            "mrs:alarm:delete",
            "kms:*:get*",
            "kms:*:list*",
            "rds:*:get*",
            "rds:*:list*",
            "mrs:bootstrap:*",
            "bss:*:view*"
         ],
         "Effect": "Allow"
      },
      {
         "Action": [
            "mrs:cluster:create",
            "mrs:cluster:resize",
            "mrs:cluster:scaleUp",
            "mrs:cluster:delete",
            "mrs:cluster:policy"
         ],
         "Effect": "Deny"
      }
   ]
}
```

MRS ReadOnlyAccess Policy

```
{
   "Version": "1.1",
   "Statement": [
      {
         "Action": [
            "mrs:*:get*",
            "mrs:*:list*",
            "mrs:tag:count",
            "ecs:*:get*",
            "ecs:*:list*",
            "bms:*:get*",
            "bms:*:list*",
            "evs:*:get*",
            "evs:*:list*",
            "vpc:*:get*",
            "vpc:*:list*",
            "kms:*:get*",
            "kms:*:list*",
            "rds:*:get*",
```

```
            "rds:*:list*",
            "bss:*:view*"
        ],
        "Effect": "Allow"
    },
    {
        "Action": [
            "mrs:cluster:create",
            "mrs:cluster:resize",
            "mrs:cluster:scaleUp",
            "mrs:cluster:delete",
            "mrs:cluster:policy",
            "mrs:job:submit",
            "mrs:job:stop",
            "mrs:job:delete",
            "mrs:job:batchDelete",
            "mrs:file:create",
            "mrs:file:delete",
            "mrs:tag:batchOperate",
            "mrs:tag:create",
            "mrs:tag:delete",
            "mrs:manager:access",
            "mrs:patch:install",
            "mrs:patch:uninstall",
            "mrs:ops:grant",
            "mrs:ops:shareLog",
            "mrs:alarm:subscribe"
        ],
        "Effect": "Deny"
    }
  ]
}
```

MRS Administrator Policy

```
{
  "Depends": [
    {
      "catalog": "BASE",
      "display_name": "Server Administrator"
    },
    {
      "catalog": "BASE",
      "display_name": "Tenant Guest"
    }
  ],
  "Version": "1.0",
  "Statement": [
    {
      "Action": [
        "MRS:MRS:*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

Helpful Links

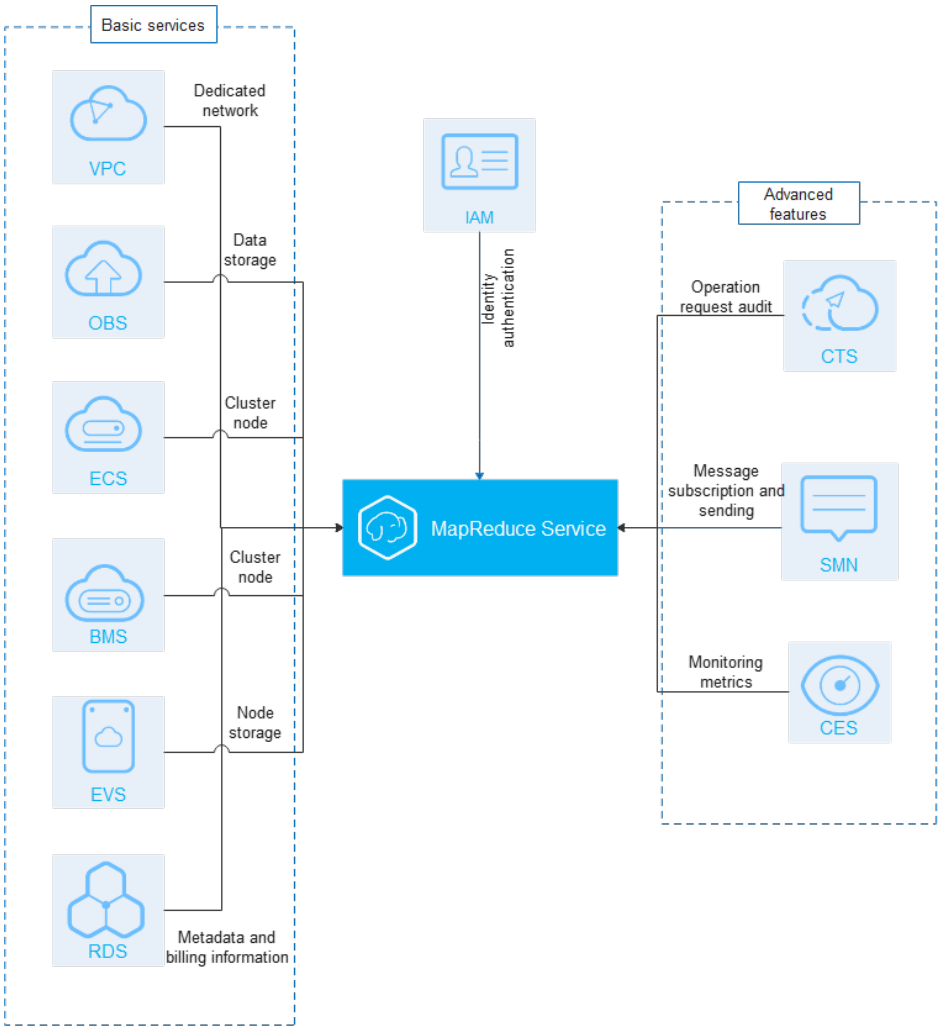
- **IAM Service Overview**
- **Creating User Groups and Users and Granting MRS Permissions**

- **Permissions Policies and Supported Actions**

12 Related Services

Figure 12-1 shows the relationship between MRS and other services.

Figure 12-1 Relationships with other services



Relationships with Other Services

Table 12-1 Relationships with other services

Service	Relationships	Reference
Virtual Private Cloud (VPC)	MRS clusters are created in the subnets of a VPC. VPCs provide a secure, isolated, and logical network environment for your MRS clusters.	**Creating a VPC and Subnet**
Object Storage Service (OBS)	OBS stores the following user data: • MRS job input data, such as user programs and data files • MRS job output data, such as result files and log files of jobs In MRS clusters, HDFS, Hive, MapReduce, YARN, Spark, Flume, and Loader can import or export data from OBS. MRS uses the parallel file system of OBS to provide services.	**Configuring a Storage-Compute Decoupled Cluster (Agency)** **Configuring a Storage-Compute Decoupled Cluster (AK/SK)**
Elastic Cloud Server (ECS)	MRS uses ECSs as cluster nodes.	**Preparing an Operating Environment** **Creating a Cluster**
Relational Database Service (RDS)	RDS stores MRS system running data, including MRS cluster metadata.	**Configuring Data Connections**
Identity and Access Management (IAM)	IAM provides authentication for MRS.	**Creating a User and Granting Permissions** **Creating MRS Custom Policies** **Synchronizing IAM Users to MRS**
Simple Message Notification (SMN)	MRS uses SMN to provide one-to-multiple message subscription and notification over a variety of protocols.	**Configuring Job Notification Rules**
Cloud Trace Service (CTS)	CTS provides you with operation records of MRS resource operation requests and request results for querying, auditing, and backtracking.	**Table 12-2**
Elastic Volume Service (EVS)	EVS provides scalable block storage that features high reliability and high performance to meet various service requirements.	-

Service	Relationships	Reference
Cloud Eye	Cloud Eye is a monitoring platform. You can use Cloud Eye to monitor the utilization of cloud resources, track the running status of cloud services, and configure alarm rules and notifications so that you can quickly respond to resource changes.	-
Bare Metal Server (BMS)	BMS provides dedicated physical servers that feature excellent computing performance and data security on the cloud for MRS.	**BMS Specifications Used by MRS**

Table 12-2 MRS operations recorded by CTS

Operation	Resource Type	Trace Name
Creating a cluster	cluster	createCluster
Deleting a cluster	cluster	deleteCluster
Expanding a cluster	cluster	scaleOutCluster
Shrinking a cluster	cluster	scaleInCluster

After you enable CTS, the system starts recording operations on cloud resources. You can view operation records of the last 7 days on the CTS management console. For details, see the *Cloud Trace Service (CTS) User Guide*.

13 Quota Description

Available resource quotas are configured for each user account in an environment to prevent resource abuse.

The following lists the resources used by MRS. Quotas are managed by each basic service. If you need to increase quotas, contact technical support of the corresponding service.

- ECS
- BMS
- VPC
- EVS
- Image Management Service (IMS)
- OBS
- EIP
- SMN
- IAM

For details about how to view and modify quotas, see **Quotas**.

14 Common Concepts

HBase Table

An HBase table is a three-dimensional map comprised of one or more columns or rows of data.

Column

Column is a dimension of an HBase table. The column name is in the format of *<family>*:*<label>*, where *<family>* and *<label>* can be any combination of characters. An HBase table consists of a set of column families. Each column in the HBase table belongs to a column family.

Column Family

A column family is a collection of columns stored in the HBase schema. To create columns, you must create a column family first. A column family organizes data with the same property in HBase. Each row of data in the same column family is stored on the same server. Each column family can be one attribute, such as compressed packages, timestamps, and data block cache.

MemStore

MemStore is a core of HBase storage. When the amount of data stored in WAL reaches the upper limit, the data is loaded to MemStore for sorting and storage.

RegionServer

RegionServer is a service running on each DataNode in the HBase cluster. It is responsible for serving and managing regions, uploading the load information of regions, and managing distributed master nodes.

Timestamp

A timestamp is a 64-bit integer used to index different versions of the same data. A timestamp can be automatically assigned by HBase when data is written or assigned by users.

Store

Store is a core of HBase storage. A Store hosts one MemStore and multiple StoreFiles. A Store corresponds to a column family of a table in a region.

Index

An index is a data structure that improves the efficiency of data retrieval in a database table. One or more columns in a database table can be used for fast random retrieval of data and efficient access to ordered records.

Coprocessor

A coprocessor is an interface provided by HBase for implementing calculation logic on RegionServer. Coprocessors are classified into system coprocessors and table coprocessors. The former can import all data tables on RegionServer, and the latter can process a specified table.

Block Pool

A block pool is a collection of blocks that belong to a single namespace. DataNodes store blocks from all block pools in a cluster. Each block pool is managed independently, which allows a namespace to generate an ID for a new block without relying on other namespaces. If one NameNode is invalid, the DataNode can still provide services for other NameNodes in the cluster.

DataNode

A DataNode is a worker node in the HDFS cluster. Scheduled by the client or NameNode, DataNodes store and retrieve data and periodically report file blocks to NameNodes.

File Block

A file block is the minimum logical unit stored in the HDFS. Each HDFS file is stored in one or more file blocks. All file blocks are stored in DataNodes.

Block Replica

A replica is a block copy stored in HDFS. A file block stores multiple replicas for system availability and fault tolerance.

NodeManager

NodeManager executes applications, monitors the usage of resources (including CPUs, memory, disks, and network resources) of applications, and reports the resource usage to the ResourceManager.

ResourceManager

ResourceManager schedules resources required by applications. It provides a scheduling plug-in for allocating cluster resources to multiple queues and

applications. The scheduling plug-in schedules resources based on existing capabilities or using the fair scheduling model.

Kafka Partitions

Each topic can be divided into multiple partitions. Each partition corresponds to an appendant log file whose sequence is fixed.

Follower

A follower processes read requests and works with a leader to process write requests. It can also be used as a leader backup. When the leader is faulty, a follower is elected to take over the leader's workload to prevent a single point of failure.

Observer

Observers do not take part in voting for election and write requests. They only process read requests and forward write requests to the leader, improving processing efficiency.

DStream

DStream is an abstract concept provided by Spark Streaming. It is a continuous data stream which is obtained from the data source or the transformed input stream. In essence, a DStream is a series of continuous resilient distributed datasets (RDDs).

Heap Memory

A heap indicates the data area where the Java Virtual Machine (JVM) is running and from which memory for all class instances and arrays is committed. The initial heap memory is controlled by the JVM startup parameter **-Xms**.

- Maximum heap memory: Heap memory that can be committed to a program at most by the system, which is specified by the **-Xmx** parameter.
- Committed heap memory: total heap memory committed by the system for running a program. It ranges from the initial heap memory and the maximum heap memory.
- Used heap memory: heap memory that has been used by a program. It is smaller than the committed heap memory.
- Non-heap memory: memory excluded from the JVM heaps and the memory area for running the JVM. Non-heap memory has the following three memory pools:
  - Code Cache: stores JIT compiled code. Its value is set through the JVM startup parameter **-XX:InitialCodeCacheSize -XX:ReservedCodeCacheSize**. The default value is 240 MB.
  - Compressed Class Space: stores metadata of a pointer. Its value is set through the JVM startup parameter **-XX:CompressedClassSpaceSize**. The default value is 1024 MB.
  - Metaspace: stores metadata. Its value is set through the JVM startup parameter **-XX:MetaspaceSize -XX:MaxMetaspaceSize**.

- Maximum non heap memory: non-heap memory committed to a program at most by the system. The value is the sum of the code buffer, space of compressed class pointers, and maximum metaspace.
- Committed non-heap memory: total non-heap memory committed by the system for running a program. It ranges from the initial non-heap memory and the maximum non-heap memory.
- Used non heap memory: non heap memory that has been used by a program. It is smaller than the committed non heap memory.

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 amounts of 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 can process petabytes of data. Hadoop consists of HDFS, MapReduce, HBase, and Hive.

Role

A role is an element of a service. A service contains one or multiple roles. Services are installed on servers through roles so that they can run properly.

Cluster

A cluster is computer technology that enables multiple servers to work as one server. Clusters improve the stability, reliability, and data processing or service capability of the system. For example, clusters can prevent single point of failures (SPOFs), share storage resources, reduce system load, and improve system performance.

Instance

An instance is formed when a service role is installed on the host. A service has one or more role instances.

Metadata

Metadata is data that provides information about other data and is also called media data or relay data. It is used to define data properties, specify data storage locations and historical data, retrieve resources, and record files.

