HCIA-Big Data V3.0 & V3.5 Training Material

Table of Contents

1. Big Data Development Trend and Kunpeng Big Data Solution

This chapter introduces the concept of big data, its characteristics, application fields, and the opportunities and challenges it presents, along with Huawei's solutions.

Big Data Era

Big data is defined as data sets whose size is beyond the ability of commonly used software tools to capture, curate, manage, and process within a tolerable elapsed time.

It is characterized by **4 Vs**:

- **Volume**: Large amounts of data (in GB, TB, or PB)
- **Velocity**: Data needs to be analyzed quickly
- **Variety**: Different types of data, including structured, semi-structured, and non-structured data
- **Value**: Data has low value density but can drive experience, decision-making, and processes

The world is transitioning to an intelligent era driven by cloud computing, big data, IoT, and AI.

Big Data Application Fields

Big data has penetrated various industries and business domains, including telecom carriers, financial institutions, governments, education, public security, transportation planning, and clean energy.

Top enterprise-level application scenarios include marketing analysis, customer analysis, and internal operational management.

Big Data Computing Tasks

I/O-intensive Tasks

Involve network, disk, and memory I/O. They have low CPU usage, with most latency caused by I/O wait, making network transmission and read/write efficiency top priorities.

CPU-intensive Tasks

Involve a large number of computing tasks (e.g., Pi calculation, HD video decoding) that consume CPU resources. The number of parallel tasks should ideally equal the number of CPU cores for best efficiency.

Data-intensive Tasks

Involve a large number of independent data analysis and processing tasks running on loosely coupled computer clusters, requiring high I/O throughput for massive data volumes and often having a data-flow-driven process.

Major Computing Modes

- Batch processing (MapReduce, Spark)
- Stream computing (Spark, Storm, Flink, Flume, DStream)
- Graph computing (GraphX, Gelly, Giraph, PowerGraph)
- Query & analytics computing (Hive, Impala, Dremel, Cassandra)

Challenges and Opportunities for Enterprises in the Big Data Era

Challenges

- High cost of mass data storage
- Insufficient batch/stream processing performance
- Limited scalability
- Data silos
- Poor data quality
- Unsatisfactory management technologies
- Data security risks
- Lack of talent
- Trade-off between data openness and privacy

Opportunities

- Big data mining as the core of business analysis
- Bolstering information technology applications
- Acting as a new engine for the information industry's growth

Huawei Kunpeng Big Data Solution

Kunpeng Ecosystem

Huawei's strategy to improve computing and data governance capabilities, based on Kunpeng processors and TaiShan servers. It aims to build an open, secure, reliable, and efficient computing capability for data centers.

Huawei Big Data Solution

Provides one-stop high-performance big data computing and data security. It uses a public cloud architecture with **storage-compute decoupling**, elastic Kunpeng computing power, and Object Storage Service (OBS) for cost-effectiveness and scalability.

HUAWEI CLOUD Big Data Services

MRS (MapReduce Service)

A HUAWEI CLOUD service for deploying and managing Hadoop systems, offering enterprise-level big data clusters with high performance, low cost, flexibility, and ease-of-use. It supports components like Hadoop, Spark, HBase, Kafka, Storm, and is fully compatible with open-source APIs.

Advantages:

- High performance (CarbonData, Superior scheduler, Kunpeng optimization)
- Easy O&M (visualized platform, rolling patch upgrade, HA)
- High security (Kerberos authentication, RBAC, physical isolation)
- Cost-effectiveness (compute/storage separation, flexible scaling, temporary clusters, auto scaling)

Application Scenarios:

- Offline analysis of massive data
- Large-scale data storage (real-time with Kafka/HBase)
- Low-latency real-time data analysis (Flume, Kafka, HBase, Storm, Spark)

DWS (Data Warehouse Service)

Cloud-based data warehouse based on GaussDB, fully compatible with SQL standards. It's an enterprise-level, distributed, multi-module data warehouse supporting OLAP analysis and time series, with storage-compute decoupling and independent scaling. Optimized for Kunpeng chips, it shows 50% performance improvement.

CSS (Cloud Search Service)

End-to-end data retrieval and analysis solution, based on Elasticsearch, offering high performance, low cost, and high availability for log search, recommendation, and database acceleration. Features include read/write separation, latency-based routing, dynamic retry, storage-compute decoupling, and automatic roll-up.

GES (Graph Engine Service)

A hyper-scale integrated graph analysis and query engine for social apps, enterprise relationship analysis, logistics distribution, knowledge graphs, and risk control.

DAYU (Intelligent Data Lake Operation Platform)

A one-stop platform for data lifecycle management and intelligent data management. It provides functions for data integration, design, development, quality control, asset management, and data services, aiming to eliminate data silos and unify standards.

2. HDFS and ZooKeeper

This chapter focuses on HDFS as a distributed storage system and ZooKeeper for distributed coordination.

HDFS (Hadoop Distributed File System)

Overview

A distributed file system designed to run on commodity hardware, offering high fault tolerance and high-throughput access for large data sets. It was originally built for Apache Nutch.

Concepts

> **Block**
>
> Default size is 128 MB. Files are divided into blocks as storage units, simplifying design and supporting large files and data backup.

> **NameNode**
>
> Manages the file system namespace, storing metadata (FsImage for file system tree, EditLog for operations). It maintains mapping between files, blocks, and DataNodes.

> **DataNode**
>
> Worker nodes that store and retrieve data, periodically reporting stored blocks to NameNodes. Data is stored in the local Linux file system.

> **Client**
>
> A library providing HDFS file system interfaces, supporting common operations and various access modes (Java APIs, HTTP, Shell).

Architecture

Follows a master-slave architecture with one NameNode and multiple DataNodes. Communication is based on TCP/IP protocols (Client Protocol, DataNode Protocol, RPC).

Key Features

- **High Availability (HA)**: Uses redundant NameNodes (active/standby) with ZooKeeper, ZKFC, and JournalNodes to prevent SPOFs.
- **Metadata Persistence**: NameNode metadata (FsImage and EditLog) is persisted, with a Secondary NameNode assisting in merging for disaster recovery.
- **HDFS Federation**: Supports multiple independent NameNodes (namespaces) sharing a common block storage pool, improving scalability and isolation.
- **Data Replica Mechanism**: Stores multiple copies of data blocks (default 3) across different racks to ensure fault tolerance and data availability.
- **Data Integrity Assurance**: Includes rebuilding replica data of failed disks, cluster data balancing, metadata reliability (log mechanism, snapshots), and a security mode.
- **Space Reclamation**: Supports recycle bin and dynamic setting of copies.

Data Read/Write Process

Detailed steps for how clients interact with NameNode and DataNodes to write and read data, including data packet writing, acknowledgment, and data block retrieval.

ZooKeeper Distributed Coordination Service

Overview

A distributed service framework solving data management problems in distributed applications, providing distributed and highly available coordination services. It is a bottom-layer component depended on by Kafka, HDFS, HBase, and Storm. It can use Kerberos and LdapServer for security authentication in security mode.

Architecture

Consists of a group of server nodes (one **Leader**, others **Followers**). The leader is elected during startup and ensures data consistency using a custom atomic message protocol.

Disaster Recovery (DR) Capability

ZooKeeper can provide services if it can complete an election, requiring an instance to obtain more than half of the votes to become the leader. DR capability depends on the number of instances.

Key Features

- **Eventual consistency**
- **Real-time capability**
- **Reliability**
- **Wait-free**
- **Atomicity** (data update succeeds or fails)
- **Sequence consistency**

Read/Write Functions

Read operations can be performed with any node due to eventual consistency. Write operations go through the leader, which then propagates proposals to followers.

3. Hive - Distributed Data Warehouse

This chapter introduces Hive, a distributed data warehouse built on Hadoop, emphasizing its architecture, functions, and SQL operations.

Introduction to Hive

A data warehouse tool running on Hadoop, supporting PB-level distributed data query and management. It offers flexible ETL, supports multiple computing engines (Tez, Spark), direct access to HDFS/HBase, and is easy to use.

Application Scenarios

- Non-real-time data analysis (log/text analysis)
- Data summarization (user behavior, traffic statistics)
- Data mining

Comparison with Traditional Data Warehouses

Aspect	Hive	Traditional Data Warehouses
Data Scale	Handles large data (GB, TB, PB) using HDFS (infinite expansion)	Limited to MBs
Execution Engine	Uses Tez (default) or Spark	Fixed engines
Flexibility	Decouples metadata and data	Less flexible
Analysis Speed	Speed depends on cluster scale and expands easily, often faster for large data	Limited scalability

Index	Lower index efficiency	High index efficiency
Reliability	Leverages HDFS for high data reliability and fault tolerance	Traditional reliability measures
Price	Open-source and free	Expensive commercial licenses

Advantages of Hive

- **High reliability and fault tolerance** (HiveServer cluster deployment, double MetaStores, timeout retry)
- **SQL-like syntax** (built-in functions)
- **Scalability** (user-defined storage format/function)
- **Multiple APIs** (Beeline, JDBC, Thrift, ODBC)

Architecture and Running Process

Components include Driver (Compiler, Optimizer, Executor), MetaStore, Thrift Server, Web Interface. It runs on Tez/MapReduce/Spark and uses YARN for resource allocation. Clients submit HQL, which is executed, and results are returned via JDBC.

Data Storage Model

Data is organized into Databases, Tables, Partitions (directories), and Buckets (files). **Partitions** allow data to be grouped by field values, enabling hierarchical organization. **Buckets** further subdivide partitions using hashing for better management and sorting.

Managed Table and External Table

> ### Managed Tables (default)
>
> Hive manages both metadata and data. When dropped, both are deleted.

> ### External Tables
>
> Hive only manages metadata; data resides outside the warehouse directory and is not deleted when the table is dropped. Recommended for shared datasets.

Functions Supported

Built-in functions (mathematical, date, string) and User-Defined Functions (UDF).

Basic Hive Operations (SQL)

DDL (Data Definition Language)

- `CREATE TABLE`
- `SHOW TABLES`
- `DESCRIBE`
- `ALTER TABLE`

DML (Data Management Language)

- `LOAD DATA LOCAL INPATH`
- `EXPORT TABLE`

DQL (Data Query Language)

- `SELECT`
- `WHERE`
- `GROUP BY`
- `INSERT OVERWRITE DIRECTORY/TABLE`
- `MULTITABLE INSERT`
- `JOIN`
- `TRANSFORM` (streaming)

4. HBase Technical Principles

This chapter details HBase, a non-relational distributed database suitable for large-scale, real-time data processing.

Introduction to HBase

A **column-based** distributed storage system with high reliability, performance, and scalability. It's suitable for **BigTable data** (billions of rows, millions of columns) and provides **real-time data access**. It uses **HDFS** for file storage and **ZooKeeper** for collaboration.

Comparison Between HBase and RDB

Aspect	HBase	Traditional RDB
Data Indexing	Only one index (the **row key**)	Multiple complex indexes
Data Maintenance	Generates a new version when data is updated, retaining the original	Overwrites data
Scalability	Flexible **horizontal expansion** by adding/reducing hardware	Difficult to scale horizontally

HBase Application Scenarios

- Data storage
- Time series data
- Meteorological data
- Cube analysis
- NewSQL
- Feeds
- Message/order storage
- User profile

Data Model

Data is stored in **tables** consisting of **rows** and **columns**. Columns belong to **column families**. The intersection of a row and column is a **cell**, which is versioned by **timestamps**. **Row keys** are byte arrays and tables are sorted by key.

> ### Conceptual View
>
> Tables appear as sparse rows.

> ### Physical View
>
> Data is differentiated by column family storage, allowing new columns without declaration.

> ### Row-based vs. Column-based Storage
>
> HBase uses column-based storage which is efficient for reading/calculating by column, but may require multiple I/O for row reads.

HBase Architecture

HMaster

Manages and maintains partition information, HRegionServer list, allocates regions, balances loads, and manages table operations. It uses **ZooKeeper for HA**.

HRegionServer

Core module that stores and maintains allocated regions, processing read/write requests from clients.

Client

Accesses HBase through an interface, queries hbase:meta table via ZooKeeper for region location, and then directly accesses the HRegionServer (not HMaster).

Table and Region

An HBase table starts with one region and splits into multiple regions as data grows. Regions are sorted lexicographically by row key.

Region Positioning

Uses Meta Regions (hbase:meta table) to record routing information for User Regions, saved in memory for speed.

HBase Key Processes

Data Read and Write Process

Data is written to HLog, then MemStore, and finally to StoreFile on disk. Reads first check MemStore cache, then StoreFile.

Cache Refreshing

MemStore content is periodically written to StoreFile, cache cleared, and a tag written to HLog. Each write generates a new StoreFile.

Merging StoreFiles (Compaction)

Multiple StoreFiles are combined into one to improve search speed. Triggered when the number of StoreFiles reaches a threshold.

HLog (Write-Ahead Log)

Ensures system recovery by recording all updates before writing to MemStore/disk. HRegionServers have their own HLogs, split and processed by HMaster during recovery.

HBase Performance Tuning and Highlights

- **Compaction**: Minor (small-scale merging of continuous time ranges) and Major (all HFiles in a column family) to reduce HFiles and improve read performance.
- **OpenScanner**: Creates StoreFileScanner for HFile and MemStoreScanner for MemStore to read data.
- **BloomFilter**: Optimizes random reads (Get scenarios) by quickly determining if data exists in a large dataset, with reliable "does not exist" judgments.
- **Row Key Design**: Critical for performance as data is stored alphabetically. Timestamps (e.g., `Long.MAX_VALUE - timestamp`) can ensure new data is easily accessed.
- **Secondary Index**: HBase primarily uses row keys. Huawei's Hindex provides secondary indexing for multiple tables/columns/values.

Common HBase Shell Commands

- `create`
- `list`
- `put`
- `scan`
- `get`
- `enable/disable`
- `drop`

5. MapReduce and YARN Technical Principles

This chapter covers MapReduce for batch/offline processing and YARN for resource management in Hadoop.

MapReduce Overview

A parallel computing framework based on the "divide-and-conquer" algorithm for large-scale datasets (>1 TB). It features **highly abstract programming ideas**, **outstanding scalability**, and **high fault tolerance** (computing/data migration).

MapReduce Process

Divided into two phases:

Map Phase

Input files are split (default 1 block per split). Data is processed, and intermediate key-value pairs are generated and stored in a ring memory buffer. When the buffer is ~80% full, a "spill" occurs, writing data to local disks, followed by optional combine and merge operations.

Reduce Phase

Map output files (MOF) are copied, sorted, and merged. If data is small, it's merged in cache; otherwise, it's spilled to disk before merging. The final merged result is output to the user-defined Reduce function.

Shuffle Process

The intermediate data transfer between Map and Reduce phases, involving fetching, sorting, and merging MOF files by Reduce tasks.

WordCount Example

Illustrates how MapReduce counts word frequencies in a text file through Map (splitting, counting) and Reduce (aggregating counts) phases.

YARN (Yet Another Resource Negotiator) Overview

A resource management system in Hadoop 2.0 that provides **unified resource management and scheduling** for upper-layer applications, improving cluster resource utilization and data sharing. It overcomes limitations of Hadoop 1.0 (SPOFs, job/resource scheduling mix, lack of isolation/security).

YARN Architecture

Consists of **ResourceManager** (RM, for cluster resource management and scheduling), **NodeManager** (NM, per-node agent for containers), and **ApplicationMaster** (AM, per-application for negotiating resources and monitoring progress). Applications run within **Containers**.

Job Scheduling Process on YARN

Client submits job -> RM allocates AM container -> AM requests containers from RM for Map/Reduce tasks -> NM launches containers -> tasks execute -> results returned.

YARN HA Solution

Uses redundant ResourceManager nodes (active/standby) with ZooKeeper to prevent SPOFs. The active RM writes states to ZooKeeper, and failover is automatic.

YARN ApplicationMaster Fault Tolerance Mechanism

Allows AMs to restart upon failure, with recovery of containers.

Resource Management

Configurable parameters for memory (`yarn.nodemanager.resource.memory-mb`, `vmem-pmem-ratio`) and CPU (`resource.cpu-vcore`) resources per NodeManager. Hadoop 3.x supports user-defined countable resource types beyond just CPU/ memory.

YARN Resource Schedulers

FIFO Scheduler

Applications in a queue are processed first-in, first-out.

Capacity Scheduler

Allows multiple organizations to share a cluster via dedicated queues, each with resource limits. Supports resource division within queues.

Fairs Scheduler

Aims for equal resource share among all applications.

Enhanced Features (Huawei MRS clusters)

- **Dynamic Memory Management**: Allows container running while calculating memory usage and stopping excessive usage containers if thresholds are exceeded.
- **Label-based YARN Scheduling**: Enables intelligent task distribution to nodes with specific capabilities (e.g., standard performance, large memory, high I/O).

6. Spark - An In-Memory Distributed Computing Engine

This chapter introduces Spark, an in-memory distributed computing engine, its data structures, and various modules like Spark SQL and Spark Streaming.

Introduction to Spark

Developed at UC Berkeley AMPLab, Spark is a fast, versatile, and scalable **memory-based big data computing engine**. It's a one-stop solution integrating batch processing, real-time streaming, interactive query, graph programming, and machine learning.

Application Scenarios

- ETL (batch processing)
- Machine learning (sentiment analysis)
- Interactive analysis (Hive warehouse queries)
- Streaming processing (page-click analysis, recommendation systems, public opinion analysis)

Highlights of Spark

- **Lightweight** (30,000 lines of core code)
- **Fast** (sub-second latency for small datasets)
- **Flexible** (different levels of flexibility)
- **Smart** (utilizes existing big data components)

Spark and MapReduce Comparison

Spark is significantly faster than MapReduce, demonstrating higher processing rates per minute and per node across large datasets.

Spark Data Structure

RDD (Resilient Distributed Datasets)

The core concept of Spark. RDDs are **elastic, read-only, and partitioned distributed datasets**. They are stored in memory by default (spilling to disk if insufficient) and have a **lineage mechanism** for rapid data recovery.

RDD Dependencies

- **Narrow Dependencies**: Each parent RDD partition is used by at most one child RDD partition (e.g., `map`, `filter`, `union`). Fault tolerance is localized; data transmission can be completed on one node.
- **Wide Dependencies**: Multiple child RDD partitions depend on the same parent RDD partition (e.g., `groupByKey`, `reduceByKey`, `sortByKey`). Recalculation might be extensive on fault; usually involves **shuffle operations** and data transmission between nodes.

Stage Division

RDD operations are divided into stages based on wide dependencies.

RDD Operation Types

- **Creation**: From memory collections or external storage (HDFS, Hadoop-supported systems).
- **Transformation (Lazy Operations)**: Transforms an RDD into a new RDD but doesn't execute immediately (e.g., `map`, `filter`, `reduceByKey`, `join`).
- **Control (Persistence)**: Stores RDDs in memory or disk (e.g., `persist`, `cache`) to improve iterative computing and data sharing.
- **Action**: Triggers Spark execution, outputs results, or saves RDDs to external systems (e.g., `reduce`, `collect`, `count`, `first`, `take`, `saveAsTextFile`).

DataFrame

Similar to RDDs (invariable, elastic, distributed), but also records **schema** (data structure information), similar to a two-dimensional table. Optimized by Spark Catalyst Optimizer for efficient execution.

DataSet

A **strongly typed dataset**. DataFrame is a special case (`DataFrame = Dataset[Row]`). `Row` represents table structure information.

Differences between RDD, DataFrame, and DataSet

Illustrated through examples of how the same data appears in each structure, highlighting the addition of schema and type safety.

Spark Principles and Architecture (Spark Core)

Spark Core is the foundational general execution engine, managing memory, fault recovery, scheduling, and I/O. It can run on Standalone, YARN, or Mesos cluster managers.

Spark SQL

Module for structured data processing, allowing seamless use of SQL statements or DataFrame APIs. It leverages a **Catalyst Optimizer** for query plan optimization.

Spark SQL vs. Hive

Spark SQL uses **Spark Core** as its engine (10-100x faster than Hive's default MapReduce engine). Spark SQL depends on Hive metadata and is compatible with most Hive syntax and functions, but does not support buckets.

Spark Structured Streaming

A stream processing engine built on Spark SQL. It allows compiling streaming processes like static RDD data, incrementally processing continuous data streams, and updating results to a result set.

Spark Streaming

Splits real-time input data streams by **time slice** (in seconds) and processes each slice using the Spark engine, similar to batch processing. It's **quasi-real-time** (seconds latency) with high throughput and supports transaction mechanisms.

7. Flink, Stream and Batch Processing in a Single Engine

This chapter focuses on Apache Flink, an open-source stream processing framework, highlighting its architecture, time/window mechanisms, and fault tolerance.

Overview and Key Concepts

Flink is a stream processing framework supporting distributed, high-performance applications with **high throughput and exactly-once semantics**, also providing batch data processing. Flink fundamentally models **batch processing as a special case of stream processing** (opposite of Spark). Key concepts include **continuous processing of streaming data, event time, stateful stream processing, and state snapshots**.

Core Ideas (State Management)

Flink's biggest differentiator is its **built-in state management**, reducing dependency on external systems, simplifying O&M, and significantly improving performance.

Overall Architecture and Job Running Process

Components

- **Source** (reads data from Kafka, HDFS, etc.)
- **Transformation** (data manipulation)
- **Sink** (outputs data to HDFS, Kafka, etc.)

Execution

A user submits a Flink program to **JobClient**. JobClient parses/optimizes and submits to **JobManager**. **TaskManager** then runs the task. JobClient acts as a bridge, handling program submission and result return.

Operators

Flink has Source, Transformation (e.g., `map`, `flatMap`, `reduce`), and Sink operators.

Data Processing

Flink supports both batch and stream processing using a single architecture.

> ### Bounded Stream
>
> Has a defined start and end; data can be processed after all data arrives (batch processing).

> ### Unbounded Stream
>
> Only a defined start; data sources generate endlessly, requiring continuous processing as data arrives (stream processing).

Batch processing is a specific case of stream processing using a global window.

Stream and Batch Processing Mechanisms

Flink provides **DataStream API** (for stream processing) and **DataSet API** (for batch processing). It also supports **Table API and SQL** for unified batch and stream processing in analytical use cases.

Flink Time & Window

Time Classification

- **Event Time**: Time when an event actually occurs.
- **Ingestion Time**: Time when an event arrives at the stream processing system.
- **Processing Time**: Time when an event is processed by the system. Processing time is simple but yields uncertain results due to network delays, while event time is complex but reproducible.

Window Overview

Splits infinite data sets into finite "buckets" for computation, essential for processing infinite streams.

Window Types

- **Tumbling Window**: Fixed window length, time-aligned, non-overlapping (e.g., hourly aggregation).
- **Sliding Window**: Fixed window length, sliding interval, can be overlapping (e.g., 5-minute failure rate every 30 seconds).
- **Session Window**: Defined by a series of events and a timeout interval, useful for grouping user activities.

Flink Watermark

Addresses the **out-of-order problem** in event time processing caused by network delays. A watermark is a monotonically increasing timestamp `t` indicating that all data with a timestamp less than or equal to `t` has arrived, allowing windows to be safely triggered and destroyed.

Delayed Data Processing

Flink can handle delayed events (those whose out-of-order degree exceeds watermark prediction) through:

- **Side Output Mechanism**: Places delayed events into an independent data stream for special processing.
- **Allowed Lateness Mechanism**: Keeps window state for a specified period after closing to allow delayed events to trigger recalculations, though this consumes extra memory.

Fault Tolerance of Flink (Checkpointing)

Checkpoint

Flink ensures **exactly-once semantics** by periodically creating **distributed snapshots** of task/operator states, allowing the system to reset to a correct state in case of failure. Checkpoint barriers are injected into the data stream to divide data for snapshots.

Checkpoint Configuration

Disabled by default. Can be configured for interval, semantic mode (Exactly-once or At-least-once, with Exactly-once being default), timeout, minimum time between checkpoints, maximum concurrent checkpoints, and externalized checkpoints (persisted externally for job failure recovery).

Savepoint

A special, manually triggered checkpoint used to save system state data for planned operations like upgrades or maintenance, ensuring end-to-once semantics.

Savepoint vs. Checkpoint

Checkpoints are automatic, lightweight, and for quick recovery from failures; Savepoints are manual, persistent, and for planned backups/restorations.

State Storage Methods (State Backends)

Determine how state is represented and persisted:

- **MemoryStateBackend**: State and checkpoints in JobManager/TaskManager memory. Limited capacity. Good for local testing/small states (ETL).
- **FsStateBackend**: State in TaskManager memory, checkpoints to external file system (local or HDFS). Larger capacity. Recommended for jobs with large state and HA setups in production.
- **RocksDBStateBackend**: State in KV database on TaskManager (memory + disk), checkpoints to external file system. Suitable for very large states (day-window aggregations) and production, but may not have high read/write performance.

8. Flume - Massive Log Aggregation

This chapter introduces Flume, an open-source system for massive log aggregation, its architecture, and key features.

What is Flume?

A **streaming log collection tool** that roughly processes data and writes it to data receivers. It collects data from various sources like local files, real-time logs, REST messages, Thrift, Avro, Syslog, and Kafka.

What Can Flume Do?

Collects log information from fixed directories or in real-time to destinations (HDFS, HBase, Kafka). Supports **cascading** (connecting multiple Flumes) and **data conflation**, as well as custom data collection tasks.

Flume Agent Architecture

> ### Infrastructure
>
> Single agent for data collection within a cluster.

> ### Multi-agent architecture
>
> Connects multiple agents to collect raw data and store it in the final system, used for importing data from outside the cluster.

Multi-Agent Consolidation

Multiple level-1 agents can point to the source of a level-2 agent, which consolidates and sends events to a single channel.

Agent Principles

Events flow from a **Source**, through a **Channel**, to a **Sink**. Interceptors and Channel Selectors can process events within the flow.

Basic Concepts

Source

Receives or generates events and places them into channels. Can be **driver-based** (external systems send data) or **polling** (Flume periodically obtains data). Must be associated with at least one channel. Common types: `exec`, `avro`, `thrift`, `http`, `syslog`, `spooling directory`, `jms`, `kafka`.

Channel

Located between a source and a sink, functioning as a queue for caching temporary events. Events are removed when successfully sent. Supports transactions and weak sequence assurance.

Persistence

Varies by type: **Memory channel** (not persistent, high throughput, data loss possible), **File channel** (persistent via WAL, complex config), **JDBC channel** (persistent via embedded DB, high reliability).

Sink

Sends events to the next hop or final destination and removes them from the channel. Must work with a specific channel. Common types: `hdfs`, `avro`, `thrift`, `file roll`, `hbase`, `kafka`, `MorphlineSolr`.

Key Features

- **Log File Collection**: Collects logs to HDFS, HBase, or Kafka for analysis.
- **Multi-level Cascading and Multi-channel Replication**: Supports multiple Flume agents in a cascade and data replication within nodes.
- **Cascading Message Compression and Encryption**: Improves transmission efficiency and security between agents.
- **Data Monitoring**: Provides metrics for received, cached, and sent data volume.
- **Transmission Reliability**: Uses transaction management to ensure data completeness and reliability. File channel ensures data is not lost on restart.
- **Failover**: Automatically switches to another channel if the next-hop agent fails or data receiving is abnormal.
- **Data Filter During Data Transmission**: Roughly filters/cleans data; supports third-party filter plugins for complex filtering.

Flume Operation Examples

Demonstrates configurations and steps for ingesting logs to HDFS and clickstream logs to Kafka in real-time.

9. Loader Data Conversion

This chapter introduces Loader, a data loading tool based on Apache Sqoop, used for efficient data import/export between big data platforms and relational databases.

Loader Definition

A data loading tool for exchanging data and files between big data platforms (Hadoop, HBase, Hive) and relational databases/file systems (SFTP). It provides a visualized wizard-based interface for job configuration and management, and a job scheduler for periodic execution.

Loader Features

- **High Performance**: Uses MapReduce for parallel data processing.
- **High Reliability**: Active-standby deployment, MapReduce job retry mechanism, and no data remanence after job failures.
- **Graphical**: Provides GUI for configuration and monitoring, easy to use.
- **Secure**: Supports Kerberos authentication and job permission management.

Loader Modular Architecture

Includes Loader Client (WebUI, CLI), Loader Server (processes requests, manages connectors/metadata, submits MapReduce jobs, monitors), REST API, Job Scheduler, Transform Engine, Execution Engine, Submission Engine, Job Manager, Metadata Repository, and HA Manager.

Loader Job Management

> ### Job Conversion Rules
>
> Loader provides various conversion operators for data transformation, such as long integer to time, null value conversion, constant field adding, random value conversion, concatenation, delimiter conversion, modulo conversion, and character string cutting.

> ### Job Creation
>
> Involves configuring basic information, HBase/HDFS details, and task settings.

Monitoring Job Execution Statuses

Users can view the execution statuses of all jobs and their historical records (start time, running time, status, failure cause, rows read/written/skipped, MapReduce log link).

Client Scripts

Loader provides shell scripts (`lt-ctl`, `lt-ucj`, `lt-ucc`) for command-line management of data sources and jobs (add, delete, query, modify, start, stop, status check).

10. Kafka Distributed Publish-Subscribe Messaging System

This chapter describes Kafka's basic concepts, architecture, and functions, with a focus on its reliability for data storage and transmission.

Introduction to Messaging Systems

Kafka

A distributed, partitioned, replicated, and ZooKeeper-based messaging system, originally developed by LinkedIn. Main applications are log collection and message systems.

Point-to-Point Messaging

Messages are persisted in a queue. Each message is consumed by a maximum of one consumer and disappears from the queue after being read. Ensures data processing sequence.

Publish-Subscribe (Pub-Sub) Messaging

Messages are persisted in a topic. Consumers subscribe to topics and can consume all messages. Messages are not deleted immediately and can be consumed by multiple consumers. Kafka implements this pattern.

Kafka Features

- **O(1) time complexity for message persistence** (maintaining performance with terabytes of data)
- **High throughput** (100,000 messages/sec per node)
- **Message partitioning** and **distributed consumption** (sequence per partition)
- Supports **offline and real-time processing**
- **Scale-out**

Architecture and Functions

Topology

Involves **Producers** (pushing messages), **Brokers** (Kafka cluster instances), **Consumers** (pulling messages), and **ZooKeeper** (for coordination).

Basic Concepts

> **Broker**
>
> A Kafka cluster server instance.

> **Topic**
>
> A category for messages, similar to a message queue.

> **Partition**
>
> Topics are divided into one or more ordered, immutable sequences of messages for throughput improvement. Each partition is a directory.

Producer

Sends messages to Kafka Brokers.

Consumer

Reads messages from Kafka Brokers.

Consumer Group

Consumers belong to a group, allowing shared data between groups but exclusive consumption within a group.

Offset

The unique long integer position of each message in a log file, used by consumers to track records.

Offset Storage Mechanism

Consumers commit offsets to Kafka to resume reading from the last committed position after failure.

Replica

A copy of a partition, guaranteeing high availability.

Leader

The active replica that producers/consumers interact with.

Follower

Replicates data from the leader.

Controller

A server in the Kafka cluster for leader election and failovers.

Data Management

Data Storage Reliability

Partition Replica

Multiple replicas of partitions distributed across brokers for availability.

Master-Slave Message Synchronization

Followers pull messages from the leader.

Kafka HA

Ensures data consumption and production continuity even if a broker fails, by electing a new leader from in-sync replicas (ISR). If all replicas fail, options include waiting for ISR replica to return (no data loss, long time) or choosing first available replica (data loss possible, short time).

Data Transmission Reliability

Message Delivery Semantics

- **At Most Once**: Messages lost, never redelivered
- **At Least Once**: Messages never lost, may be redelivered/reprocessed
- **Exactly Once**: Messages never lost, processed only once

Idempotency

Ensures an operation has the same effect if performed multiple times. Kafka uses sequence numbers in message batches for deduplication, persistent to replica logs.

Acks Mechanism

Producer's acknowledgment configuration for reliability: `acks=0` (no wait, no guarantee), `acks=1` (leader writes, no follower ack, data loss possible on leader fail), `acks=all` (leader waits for all in-sync replicas, strongest guarantee).

Old Data Processing Methods

Kafka retains all messages. Partitions are subdivided into segments for periodical clearing. Log cleanup policies include `delete` (by retention time/size) and `compact`.

11. LDAP and Kerberos

This chapter describes identity and access management (IAM) using LDAP for directory services and Kerberos for single sign-on (SSO) and authentication in big data platforms.

IAM (Identity and Access Management)

Purpose

Manages access control permissions for data and resources, providing unified authentication and session management for users across various open-source components.

Features

User management, authentication, SSO, hierarchical management, permission management, session management, and good OS compatibility.

Structure

IAM management module, IAM authentication server, and identity storage server. Huawei solutions use OpenLDAP for identity storage and Kerberos for unified authentication.

Directory Services and Basic Principles of LDAP

Directory Service

Optimized for querying, browsing, and searching data, stored in a tree-like structure.

LDAP (Lightweight Directory Access Protocol)

A protocol for centralized account management, running over TCP/IP. It is an IETF standard.

LDAP Server (Directory Service System)

Functions as a directory service system for centralized account management. Based on OpenLDAP, using Berkeley DB as default backend.

Organizational Model (Directory Tree)

Information is organized in a tree structure. Each node (entry) has a unique distinguished name (dn). Root defines domain components (`dc`), below which are organization units (`ou`), containing objects (e.g., users identified by `cn` and `uid`). This tree structure is suitable for write-once, query-multiple-times scenarios.

Functional Model

Supports query (search, compare), update (add, delete, modify entries/dn), authentication (bind, unbind), and other operations (abandon, extend).

Integration Design (e.g., with Hive)

Involves identity authentication architecture, function, and process design. LDAP manages users via groups and roles (user's dn added to group's member attribute, `nsrole` for roles).

Advantages

Centralized user account management (easier creation, reclamation, permissions, auditing), secure access across systems/databases (centralized policies), and improved big data platform access authentication security by separating account management from access control.

SSO (Single Sign-On) and Basic Principles of Kerberos

SSO

Part of big data platform identity management. Users log in once and can freely switch between components. Offers convenience, improved O&M efficiency, and simplified application development.

Mainstream SSO Approaches

Cookies-based, Broker-based (Huawei uses Kerberos), Agent-based, Agent & Broker-based, Gateway-based, Token-based.

Kerberos

An authentication concept (named after Hades' three-headed dog) using a client-server model and cryptographic algorithms (DES, AES). Provides mutual authentication. Huawei uses KrbServer for all components, protecting against eavesdropping, replay attacks, and ensuring data integrity.

Three Core Elements of KrbServer

Kerberos Client, **Kerberos KDC Server** (consisting of **AS** - Authentication Server, and **TGS** - Ticket Granting Server), and **Key Distribution Center (KDC)**.

KrbServer Authentication Process

Involves client requesting TGT from AS, then ST from TGS using TGT, and finally using ST to access the Kerberos Server.

Strengths

Prevents brute-force/replay attacks (short-term session keys, timestamps), supports mutual authentication, provides reliable performance (reduces server/KDC pressure).

Weaknesses

Master key transmitted multiple times, KDC needs to store many keys (high maintenance cost), strict time synchronization needed for timestamps.

Scenario Architecture of Huawei Big Data Security Authentication

Interaction between Kerberos and LDAP

Kerberos is the authentication server center, providing unified authentication. LDAP is the user data storage center, storing user info (passwords, groups, roles). Kerberos obtains user info from LDAP for each authentication.

User Storage

LDAP stores default and created user information, including Kerberos (username, password for authentication query) and LDAP (user type, group, role, email for permission identification) information.

Service Authentication in the Cluster

In security mode, all service components (e.g., HDFS) depend on Kerberos. They must pass Kerberos authentication to provide external services. Services obtain session keys (keytab) from Kerberos in advance.

Common Role Deployment Mode

Kerberos Server and Kerberos Admin roles deployed in load sharing mode on control nodes. LDAP Server (SLAPD server) deployed in active/standby mode on control nodes. Recommended to deploy LDAP and KrbServer on the same node for optimal performance.

Enhanced Open-Source LDAP+Kerberos Features

Include service authentication in the cluster, application development authentication, and cross-Manager mutual trust.

12. Elasticsearch - Distributed Search Engine

This chapter provides an overview of Elasticsearch, its architecture, and key features as a distributed search engine and data analysis tool.

Elasticsearch Overview

A high-performance, Lucene-based **full-text search service**. It's a distributed **RESTful search and data analysis engine**, also usable as a NoSQL database. Offers Lucene extension, seamless switchover between environments, horizontal scaling, and supports structured/unstructured data.

Elasticsearch Features

- **High performance**: Immediate search results, inverted index for full-text search.
- **Scalability**: Horizontal scaling, runs on hundreds/thousands of servers.
- **Relevance**: Search results sorted by various elements (word frequency, proximity, popularity).
- **Reliability**: Automatic fault detection (hardware, network segmentation) ensures cluster security and availability.

Elasticsearch Application Scenarios

Log search/analysis, spatiotemporal search, time series search, and intelligent search. Handles complex data types (structured, semi-structured, unstructured) via cleansing, word segmentation, and inverted index creation. Supports diversified full-text search criteria and real-time search on written data.

Elasticsearch Ecosystem

ELK/ELKB (Elasticsearch, Logstash, Kibana, Beats) provides a complete open-source solution for diverse requirements.

Elasticsearch System Architecture

A **Cluster** contains **EsNodes**, one of which is the **EsMaster**. Data is stored in **Shards** and their **Replicas** across nodes. Clients interact with the cluster to index and search, with ZooKeeper potentially involved for cluster information.

Internal Architecture

Provides RESTful/Java APIs, uses a cluster discovery mechanism, supports script languages, and is based on Lucene. Indexes are stored in local files, shared files, or HDFS.

Basic Concepts

> ### Index
>
> A logical namespace.

> ### Type
>
> (Deleted in Elasticsearch 7) Used to store different document types.

> ### Document
>
> Basic unit that can be indexed.

Mapping

Restricts field types.

Cluster

Contains multiple nodes, with one elected as master.

EsNode

An Elasticsearch instance.

EsMaster

Manages cluster-level changes (index creation/deletion, node adding/removal).

Shard

A split of a complete index, distributed across nodes.

Replica

Copies of shards, improving fault tolerance and search efficiency by load balancing.

Recovery

Data recovery/redistribution when nodes are added/deleted or fail.

Gateway

Stores Elasticsearch index snapshots; allows recovery from disabled clusters.

Transport

Interaction mode between internal nodes or cluster and client (TCP, HTTP, Thrift, etc.).

Key Features

Inverted Index

Core of full-text search. Searches for keywords (values) to locate corresponding documents (keys), opposite of forward index.

Elasticsearch Access APIs

RESTful requests (GET, POST, PUT, DELETE, HEAD) or language-specific APIs (Java, cURL) for data operations.

Elasticsearch Routing Algorithm

Determines which shard a document belongs to. Default route uses `hash(routing) % number_of_primary_shards`. Custom routes allow specified shard writing/searching.

Elasticsearch Balancing Algorithm

Provides automatic balancing across nodes for capacity expansion/reduction and data import, based on index and node weights.

Elasticsearch Capacity Expansion/Reduction

Scenarios include high resource consumption, excessive index data volume. Methods involve adding EsNode instances/nodes, followed by automatic balancing. Reduction requires ensuring replica existence and data migration.

Elasticsearch Indexing HBase Data

HBase data is written to HDFS, and Elasticsearch creates corresponding indexes mapped to HBase rowkeys, enabling full-text search. Supports batch indexing via MR tasks.

Elasticsearch Multi-instance Deployment on a Node

Increases single-node CPU, memory, and disk usage, improving indexing/search capability. Instances are differentiated by IP/port.

Elasticsearch Cross-node Replica Allocation Policy

Parameter `cluster.routing.allocation.same_shard.host` ensures replicas are allocated across different physical nodes to prevent SPOFs, even with multiple instances on one node.

New Features

HBase full-text indexing (via HBase2ES tool) and encryption/authentication for secure cluster access.

13. Redis In-Memory Database

This chapter provides an overview of Redis, its application scenarios, data types, optimization methods, and service development.

Redis Overview and Features

A network-based, high-performance **key-value in-memory database**. Similar to Memcached but supports **data persistence** and **diverse data types**. Features high performance, low latency, and access to varied data structures.

Redis Application Scenarios

- Obtaining latest N data
- Top N applications (ranking)
- Applications requiring precise expiration time (user sessions, SMS verification codes)
- Counter applications (website access)
- Constructing queue systems
- Caching frequently accessed table data
- Publish/Subscription (pub/sub)

Redis Service Process

> ### Architecture
>
> Without a central control node. Node status exchanged via **Gossip protocol**. Each node maintains key-to-server mapping. Client sends requests to any node, which **redirects** (not forwards) to the correct node.

> ### Data Reading and Writing Processes
>
> Client accesses any node, gets cluster topology (node list, slot-to-node mapping), caches it. Client calculates key's slot (`hash(KEY)%16384`) and accesses the correct server directly. Redirection occurs if the key is on a different node.

Redis Features - Multiple Databases

Supports multiple databases (default 16), numbered 0 onwards. `SELECT` command switches. Databases are not completely isolated; `flushall` clears all DBs on an instance, `flushdb` clears current DB.

Basic Commands

- `keys`
- `exists`
- `del`
- `type`

Commands are case insensitive.

Redis Data Types

> ### String
>
> Most basic, stores any content (binary, image) up to 1 GB. Commands: `set` / `get`, `mset` / `mget`, `incr` / `decr`, `append`, `strlen`.

Hash

Stores mapping between fields and string values. Suitable for objects. Commands: `hset` / `hget`, `hmset` / `hmget`, `hexists`, `hincrby`, `hdel`, `hkeys` / `hvals`, `hlen`.

List

Ordered string list, implemented as a bidirectional link. Can be used as a queue. Commands: `lpush` / `rpush`, `lpop` / `rpop`, `llen`, `lrange`, `lrem`, `lindex`, `lset`, `ltrim`, `linsert`, `rpoplpush`.

Set

Unordered collection of unique elements. Commands: `sadd` / `smembers`, `srem` / `sismember`, `sdiff` / `sinter` / `sunion`, `scard`, `spop`, `srandmember`.

Sorted Set

Elements with associated scores, ordered by score. Commands: `zadd` / `zscore`, `zrange` / `zrevrange`, `zrangebyscore`, `zincrby`, `zcard`, `zcount`, `zrem`, `zremrangebyrank` / `zremrangebyscore`.

Setting TTL of a Key (Expire Command)

Sets a Time To Live for keys (`expire` in seconds, `pexpire` in milliseconds). Keys are automatically deleted after TTL. `ttl` / `pttl` checks remaining time, `persist` cancels TTL. Used for time-limited info, cache, frequency limits.

Redis Pipeline

Not a command-line feature but supported by clients (e.g., Jedis). Improves performance by sending multiple commands in one round trip.

Data Sorting (sort Command)

Sorts list, set, and ordered set types. Supports `desc`, `limit`, `by reference key`, `get` parameters. Performance optimization: reduce elements, use `limit`, `store` results.

Redis Task Queues

`lpush` and `rpop` implement common task queues. `brpop` supports priority queues with timeout.

Redis Persistence

Two modes, used separately or together:

> ### RDB mode (Default)
>
> Snapshot-based. Redis takes snapshots of all data in memory and stores it to disk (e.g., `dump.rdb`) when conditions are met (`save 900 1`). `save` (blocks) or `bgsave` (subprocess) commands trigger snapshots.

> ### AOF mode (Append Only File)
>
> Log-based. Disabled by default, enabled by `appendonly yes`. Commands are appended to log file. Synchronization policies: `always`, `every second` (recommended), `no` (OS sync). AOF is preferred for data restoration if both are enabled.

Redis Memory Usage

32-bit vs. 64-bit systems. 64-bit uses more memory per pointer but supports larger memory. Recommended for large-scale services.

Redis Optimization

- **Simplify key names and values**.
- **Disable persistence** if not needed.
- **Optimize internal coding** (Redis automatically adjusts).
- **SLOWLOG**: Records slow commands; configurable thresholds.
- **Modify Linux kernel memory allocation policy** (`vm.overcommit_memory = 1`).
- **Disable Transparent Huge Pages (THP)** to prevent memory locks affecting performance.
- **Modify maximum TCP connections** (`net.core.somaxconn`).
- **Limit Redis memory size** (`maxmemory`, `maxmemory-policy`) to prevent swap usage or OOM errors.
- **Use pipelines or multi-data commands** (e.g., `mset`, `hmset`) due to single-thread model to improve efficiency.

Redis Application Cases

Using Redis as a cache for online recommendation services, storing user info (hash) and recommendation results (set), with daily data import from HBase and real-time reads from Redis (falling back to HBase/real-time compute if cache fails).

14. Huawei Big Data Solution

This chapter outlines Huawei's big data strategy, its cloud services, and the intelligent data lake platform.

Development Trend of ICT Industry

Data as a new asset, intelligence as new productivity, shifting from siloed applications to integrated cloud-based data mid-ends.

HUAWEI CLOUD Big Data Services

Overview

Provides stable, reliable, secure cloud services, a powerful computing platform, and an easy-to-use development platform for Huawei's full-stack AI strategy. Aims to build an open, cooperative ecosystem.

HUAWEI CLOUD Stack

Ideal hybrid cloud for governments and enterprises, offering diverse services, seamless evolution, and multi-architecture computing power.

Industrial Cloud Solution

1+N architecture for digital transformation, integrating cloud computing, big data, AI. Features include one cloud for multi-branch/multi-site, four-dimensional collaboration, continuous innovation, and an open ecosystem.

Huawei Data Mid-End Solution

One-stop AI + data services, full-stack business continuity with Kunpeng/Ascend chips, and integration with MRS, DWS, DLI, CSS, GES, and ModelArts.

Huawei Cloud Intelligent Data Lake Operation Platform (DAYU)

Features

Unified metadata, data map, data asset catalog, data quality monitoring, E2E data lineage, unified data standards, data security management, periodic task processing/scheduling.

Operation Platform

Provides one-stop data asset management, development, exploration, and sharing. Connects seamlessly with MRS, DWS, DLI. Simplifies data integration, governance, and development.

Advantages

Systematic/process-based data standards, full-link data requirements, automated/codeless development, metadata center, GUI-based operations, hierarchical open APIs, intelligent auxiliary governance, rich data foundations, one-stop governance.

MRS: Huawei's Big Data Platform

Trends

Evolution from single-node/midrange computer to distributed/general-purpose server, then cluster/appliance, and now cloud-native data lake with distributed/cloud/AI integration.

Huawei Cloud MRS

Enterprise-level big data clusters with full control, high performance, low cost, flexibility, and ease-of-use.

MRS Highlights

Decoupled storage/compute (unified data lake, cost-effective), first-class performance (full-stack acceleration, millisecond response), cutting-edge open-source technologies (Spark/Hive/Flink reconstruction, CarbonData, Superior Scheduler for 20,000+ nodes), high security/availability (cross-AZ HA, rolling upgrade, task reconnection, network/account/data security).

MRS Architecture

Manager for cluster management, Ranger for data security, YarnScheduler, HDFS/OBS storage, Superior scheduler, Hudi storage engine, HetuEngine for interactive query, Kafka/Flink for real-time analytics, ClickHouse for real-time retrieval, HBase/Redis/Elasticsearch for real-time/offline, Hive/Tez/Spark for offline analysis. Deployed on PM/VM/BMS.

MRS Application Scenarios

Offline data lake, real-time data lake, logical data lake, and specialized data marts (`three data lakes + one data mart` strategy). Supports high-performance interactive query, real-time incremental import, cross-lake analysis, and integration with various specialized databases.

MRS Components

Hudi

Apache open-source project (Huawei participated in dev). A data lake table format providing **update and delete capabilities** for HDFS datasets, supporting multiple compute engines, IUD interfaces, and streaming primitives (upsert, incremental pull).

Features

Fast updates (custom indexes), snapshot isolation for read/write, file size/layout management, timeline, data rollback, savepoints, asynchronous merging, clustering for storage optimization.

Architecture

Supports **Copy On Write (COW)** for high read and **Merge On Read (MOR)** for high write performance. Uses open-source Parquet/HFile. Supports HDFS/OBS. Provides read-optimized, incremental, and real-time views.

HetuEngine

Huawei-developed high-performance engine for **distributed SQL query and data virtualization**. Compatible with big data ecosystem for mass data query within seconds. Supports heterogeneous data sources for one-stop SQL analysis.

Features

Cross-domain (GB/s data transmission, zero metadata sync), cross-source (data virtualization, materialized views, UDFs, HQL compatibility, BI tool interconnection), and cloud-native (auto scaling, multi-instance/tenant, rolling restart, backup/DR).

Comparison with Open-Source Presto

HetuEngine offers significant performance and scalability improvements, integrated security, higher availability/reliability, and enhanced SQL syntax compatibility, along with dynamic cloud-native deployment.

Ranger

Apache Ranger provides a **centralized security management framework** for **unified authorization and auditing** across Hadoop components (HDFS, Hive, HBase, Kafka, Storm, Spark). Users configure policies via a web UI.

LDAP+Kerberos Security Authentication

LDAP

Lightweight Directory Access Protocol for centralized account management and directory services.

Kerberos

Authentication protocol for SSO, providing mutual authentication and protecting against attacks. Huawei's MRS uses KrbServers for all components to ensure reliable authentication.

Architecture

User login authentication flows through Manager WebUI, CAS Server, Kerberos, and LDAP for user group and authentication data. Components access Kerberos for service authentication.

MRS Cloud-Native Data Lake Baseline Solution

Implements the "three lakes + mart" service scenario:

Offline Data Lake

Stores vast data in native format for multi-dimensional analysis. Data typically ingested with >15 minutes delay (offline).

Real-Time Data Lake

Data ingested within one minute (real-time) or 1-15 minutes (quasi-real-time) after generation.

Logical Data Lake

A virtual data lake combining multiple physically dispersed data platforms, reducing migration and improving analysis efficiency.

Benefits

Rolling upgrades for service continuity, decoupled storage-compute for resource optimization (30%+ compute, 100%+ storage usage, 60% TCO reduction), and collaborative analysis across lakes/warehouses (80% ETL reduction, 10x+ analysis efficiency).

15. ClickHouse — Online Analytical Processing Database Management System

This chapter introduces ClickHouse as an OLAP DBMS, covering its overview, use cases, architecture, and features.

ClickHouse Overview

An **OLAP (Online Analytical Processing) column-oriented database management system** developed by Yandex. It is independent of the Hadoop big data system and features ultimate compression rate and fast query performance, especially for aggregation analysis and large, wide tables, being one order of magnitude faster than other analytical databases.

ClickHouse Advantages

- **Relational model and SQL query**
- **Data sharding and distributed query**
- **Excellent performance**
- **Comprehensive DBMS functions**
- **Column-oriented storage and data compression**
- **Vectorized execution engine**

Use Cases

Integrates into **logical data lakes**, **offline data lakes**, and **real-time data lakes** for interactive and real-time OLAP. Applicable for network/app traffic analysis, user behavior analysis, BI, monitoring systems, and queries on wide tables with aggregation. **Inapplicable for OLTP, high-frequency key-value access, document/unstructured data storage, point queries (sparse indexes), or frequent updates/deletions**.

ClickHouse Architecture and Basic Features

Architecture

Clients (CLI, JDBC Driver) interact with ClickHouse Servers, which coordinate via a ZooKeeper cluster for ZNodes.

SQL Capability

Supports declarative SQL queries (`GROUP BY`, `ORDER BY`, `FROM`, `JOIN`, `IN`, uncorrelated subqueries). Compatible with standard syntax for basic operations (`CREATE DATABASE/TABLE`, `INSERT INTO`, `SELECT`, `ALTER TABLE`, `DESC`, `DROP`, `SHOW`). Does not support standard `UPDATE`/`DELETE` (implemented via `ALTER TABLE`), correlated subqueries, or window functions.

Table Engine

Critical for data storage, query modes, concurrency, indexing, multi-thread requests, and replication. Common engines include `TinyLog`, `Memory`, `MergeTree`, `ReplacingMergeTree`.

Replicas

Prevent data loss and increase storage redundancy; data between replicas is identical. Table-level configurable, requiring ZooKeeper for synchronization. Supports multi-active architecture where `INSERT` / `ALTER` queries on any replica have the same effect.

Distributed Query

Achieves linear expansion through sharding and a **distributed table mechanism**. A distributed table doesn't store data but acts as a proxy, routing queries to shard nodes.

Enhanced Features of ClickHouse (Huawei)

Visualized O&M

GUI for installation, config, startup/stop, client. Over 70 monitoring/alarm metrics, health checks. Dynamic log level changes, visualized log download/retrieval, and audit logs for fault locating.

Load Balancing

Uses `ClickHouseBalancer` for TCP/HTTPS connections.

Online Scale-out and Data Migration

Supports visualized migration tools (`clickhouse-copier`, Loader, Flink, HetuEngine).

16. Huawei DataArts Studio

Huawei DataArts Studio is a one-stop data governance and operations platform designed to help enterprises achieve digital transformation by providing intelligent data lifecycle management.

Core Purpose

DataArts Studio aims to help enterprises maximize the value of their data by treating it as an asset, through the orchestration of people, processes, technologies, and policies. It supports the intelligent construction of industrial knowledge libraries and incorporates fundamental big data storage, computing, and analytical engines. It enables the elimination of data silos, the unification of data standards, and the acceleration of data monetization, thereby promoting digital transformation.

Key Functions and Modules

DataArts Studio offers a comprehensive set of functions:

DataArts Migration

Facilitates efficient ingestion of multiple heterogeneous data sources. It supports batch data migration between over 20 homogeneous and heterogeneous data sources, including file systems, relational databases, data warehouses, NoSQL databases, big data cloud services, and object storage. It uses a distributed compute framework and concurrent processing for rapid, downtime-free batch data migration. It also includes DIS (Data Ingestion Service) for transmitting real-time streaming data to the cloud, continuously capturing and storing large volumes of data from various sources like logs and clickstreams.

DataArts Architecture

Provides visualized, automated, and intelligent data modeling. It incorporates data governance methods for visualizing operations, connecting data across layers, formulating data standards, and generating data assets. It supports ER modeling and dimensional modeling, and helps build standard metric systems to eliminate data ambiguity and facilitate inter-departmental communication.

DataArts Factory

A one-stop agile big data development platform with a visualized graphical development interface. It supports script development and job development, offers fully-hosted job scheduling and O&M monitoring capabilities, includes a built-in industry data processing pipeline, and supports one-click development and online collaborative development by multiple users.

DataArts Quality

Enables verifiable and controllable data quality by monitoring metrics and data quality to identify unqualified data promptly. Users can create metrics, rules, or scenarios and schedule them in real-time or recursively. It checks data accuracy based on requirements like integrity, validity, timeliness, consistency, accuracy, and uniqueness, standardizing data and monitoring it periodically.

DataArts Catalog

Provides end-to-end data asset visualization with enterprise-class metadata management. It clarifies information assets, supports data drilling and source tracing, and displays data lineage and a panorama of data assets for intelligent data search, operations, and monitoring.

DataArts DataService

Improves access, query, and search efficiency by centrally managing enterprise APIs. It controls access to subjects, profiles, and metrics, aiming to enhance the experience for data consumers and the efficiency of data asset monetization. It uses a serverless architecture, allowing users to focus on API query logic without managing the underlying infrastructure, and supports elastic scaling of compute resources to reduce O&M costs.

DataArts Security

Offers all-round protection including cyber security (tenant isolation, access permissions control, security hardening), user permissions control (role-based access control, fine-grained policies), and data security (review mechanisms for key processes, data classification, privacy protection, data masking, watermarking, encryption, and audit logs).

Advantages

- **High Efficiency**: Offers three times higher development efficiency with simple drag-and-drop development, multi-dimensional real-time search, and zero-code or low-code API development.
- **Integration and Ecosystem**: Provides 100+ open APIs to accelerate integration and development for industry ISVs. It also includes data standards, models, metrics, and APIs from over 10 partners. It seamlessly connects to data foundations like MRS, DWS, and DLI on HUAWEI CLOUD.
- **Intelligent and Automated**: Minimizes human intervention in data governance by applying AI and machine learning technologies. Provides intelligent auxiliary governance features.
- **Visualized Operations**: Features a GUI-based and simplified operation drawing from Huawei's internal digital operation experience and methodology.
- **Comprehensive Management**: Offers one-stop management of metadata, data standards, data quality, and data security.
- **Supports Heterogeneous Data**: Enables data integration for government data with complex data sources and various data types without switching multiple tools.

In essence, DataArts Studio provides a holistic platform for managing, developing, governing, and monetizing data assets within an enterprise's big data landscape.