MapReduce Service

Component Development Specifications

Issue 01

Date 2024-12-11

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Trademarks and Permissions


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

Notice

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

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

Huawei Cloud Computing Technologies Co., Ltd.

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

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

Contents

1 ClickHouse

1.1 ClickHouse Application Development Rules

Ensure That the Time on the Client Is the Same as That on the Server If the Cluster Is Installed in the Security Mode

If the cluster is of the security edition and Kerberos authentication is required, the time on the server must be the same as that on the client. Pay attention to the time difference conversion between time zones. If the time is inconsistent, the client authentication fails and subsequent service processes cannot be executed.

ClickHouse Uses Its Own ZooKeeper Service

ClickHouse relies heavily on ZooKeeper and does many read and write operations on it. To avoid affecting other services, each ClickHouse service should use its own ZooKeeper service.

Use partition fields and index fields of data tables properly

The MergeTree engine organizes and stores data in partition directories. During data query, partitions can be used to effectively skip useless data files and reduce data reading.

The MergeTree engine sorts data based on the index field and generates sparse indexes based on the **index_granularity** configuration. Data can be quickly filtered based on index fields, reducing data reading and improving query performance.

Insert a large volume of data at a low frequency

Each time data is inserted in ClickHouse, one or more part files are generated. If there are too many data parts, the pressure on merging increases and an exception may occur, affecting data insertion. You are advised to insert 100,000 rows at a time and ensure the frequency is no more than once per second.

Do not use the character type to store data of the time, date, or numeric type

Especially when the time, date, or numeric field needs to be calculated or compared.

The number of records in a single table (distributed table) cannot exceed trillions, and the number of records in a single table (local table) cannot exceed ten billions

The performance of querying trillions of tables is poor, and the cluster maintenance is difficult.

Data lifecycle management must be considered during table design

The disk space is limited, and data lifecycle management needs to be considered. The MergeTree engine supports column fields and table-level TTL when creating tables. When the values in a column field expire, ClickHouse replaces them with the default values of the data type. If all values of a column in a partition have expired, ClickHouse deletes the column files in the partition directory from the file system. When the data in a table expires, the ClickHouse deletes all the corresponding rows.

The external component ensures the idempotence of imported data

ClickHouse does not support transactions for data write. Use the external import module to control data idempotence. For example, if data of a batch fails to be imported, drop the corresponding partition data. After the fault is rectified, import the partition data again.

When a local ClickHouse table is created, the partition by keyword must be carried. Otherwise, the table cannot be migrated on the ClickHouse data migration page of Manager

The ClickHouse data migration page depends on the partition field of the table during table data migration. If partition by is not used to create partitions when the table is created, the table cannot be migrated on the ClickHouse data migration page of Manager.

Place a small table on the right for join query

When two tables are joined, the data in the right table is loaded to the memory, and then the data in the left table is traversed based on the data in the right table for matching. Placing the small table on the right reduces the number of match queries. According to the usage, the performance of joining a large table to a small table is improved by several orders of magnitude compared with that of joining a small table to a large table.

1.2 ClickHouse Application Development Suggestions

Properly configure the maximum number of concurrent operations

ClickHouse has a high processing speed because it uses the parallel processing mechanism. Even if a query is performed, half of the CPU of the server is used by default. Therefore, the ClickHouse does not support high-concurrency query scenarios. The default maximum number of concurrent requests is 100. You can adjust this number as needed, but it should be no more than 200.

Deploy the load balancing component. The query is performed based on the load balancing component to prevent the performance from being affected due to heavy single-point query pressure

ClickHouse can connect to any node in the cluster for query. If the query is performed on one node, the node may be overloaded and the reliability is low. You are advised to use ClickHouseBalancer or other load balancing services to balance the query load and improve reliability.

Properly set the partition key, ensure that the number of partitions is less than 1000, and use the integer type for the partition field

1. You are advised to use toYYYYMMDD (table field pt_d) as the partition key. The table field pt_d is of the date type.
2. If hourly partitioning is required in the service scenario, use *toYYYYMMDD* (table field **pt_d**) and *toYYYYMMDD* (table **field pt_h**) as the joint partitioning key. *toYYYYMMDD* (table **field pt_h**) is an integer number of hours.
3. If data needs to be stored for many years, you are advised to create partitions by month, for example, *toYYYYMM* (table field **pt_d**).
4. Properly control the number of parts based on factors such as the data partition granularity, volume of data submitted in each batch, and data storage period.

During query, the most frequently used and most filtered fields are used as the primary keys. The fields are sorted in descending order of access frequency and dimension cardinality

Data is sorted and stored based on primary keys. When querying data, you can quickly filter data based on primary keys. Setting primary keys properly during table creation can greatly reduce the amount of data to be read and improve query performance. For example, if the service ID needs to be specified for all analysis, the service ID field can be used as the first field of the primary key.

Properly set the sparse index granularity based on service scenarios

The primary key index of ClickHouse is stored by using a sparse index. The default sampling granularity of the sparse index is 8192 rows, that is, one record is selected from every 8192 rows in the index file.

Suggestions:

1. The smaller the index granularity is, the more effective the query in a small range is. This avoids the waste of query resources.
2. The larger the index granularity is, the smaller the index file is, and the faster the index file is processed.
3. If the table index granularity exceeds 1 billion, set this parameter to **16384**. Otherwise, set this parameter to **8192** or a smaller value.

Local Table Creation Reference

Reference:

```
CREATE TABLE mybase_local.mytable
(
    `did` Int32,
    `app_id` Int32,
    `region` Int32,
    `pt_d` Date
)
ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/mybase_local/mytable', '{replica}')
PARTITION BY toYYYYMMDD(pt_d)
ORDER BY (app_id, region)
SETTINGS index_granularity = 8192, use_minimalistic_part_header_in_zookeeper = 1;
```

Instructions:

1. Select a table engine:

   ReplicatedMergeTree: MergeTree engine that supports the replica feature. It is the most commonly used engine.

2. Table information registration path on ZooKeeper, which is used to distinguish different configurations in the cluster:

   /clickhouse/tables/{shard}/{databaseName}/{tableName}: **{shard}** indicates the shard name, **{databaseName}** indicates the database name, and **{tableName}** indicates the replicated table name.

3. **order by** primary key field:

   The most frequently used and most filterable field is used as the primary key. The dimensions are sorted in ascending order of access frequency and dimension cardinality. It is recommended that the number of sorting fields be less than or equal to 4. Otherwise, the merge pressure is high. The sorting field cannot be null. If the sorting field is null, data conversion is required.

4. **partition by** field

   The partition key cannot be null. If the field contains a null value, data conversion is required.

5. Table-level parameter configuration:

   index_granularity: sparse index granularity. The default value is **8192**.

   use_minimalistic_part_header_in_zookeeper: whether to enable the optimized storage mode of the new version for data storage in the ZooKeeper.

6. For details about how to create a table, visit **https://clickhouse.tech/docs/en/engines/table-engines/mergetree-family/mergetree/**.

Distributed Table Creation Reference

Reference:

```
CREATE TABLE mybase.mytable AS mybase_local.mytable
ENGINE = Distributed(cluster_3shards_2replicas, mybase_local, mytable, rand());
```

Instructions:

1. Name of the distributed table: **mybase.mytable**.
2. Name of the local table: **mybase_local.mytable**.
3. Use **AS** to associate the distributed table with the local table to ensure that the field definitions of the distributed table are the same as those of the local table.
4. Parameter description of the distributed table engine:

   cluster_3shards_2replicas: name of a logical cluster.

   mybase_local: name of the database where the local table is located.

   mytable: local table name.

   rand(): (optional) sharding key, which can be the raw data (such as did) of a column in the table or the result of a function call, such as rand(). Note that data must be evenly distributed in this key. Another common operation is to use the hash value of a column with a large difference, for example, **intHash64(user_id)**.

Select the minimum type that meets the requirements based on the fields in the service scenario table

Numeral type, such as UInt8/UInt16/UInt32/UInt64, Int8/Int16/Int32/Int64, Float32/Float64. The performance varies according to the length.

Perform data analysis based on large and wide tables. Do not join large tables. Convert distributed join queries into join queries of local tables to improve performance

The performance of ClickHouse distributed join is poor. You are advised to aggregate data into a wide table on the model side and then import the table to ClickHouse. Queries in distributed join mode are converted to join queries on local tables. This eliminates the transmission of a large volume of data between nodes and reduces the volume of data involved in the calculation of local tables. The service layer summarizes data based on the local join results of all shards. The performance is improved remarkably.

Properly set the part size

The **min_bytes_to_rebalance_partition_over_jbod** parameter indicates the minimum size of the part involved in automatic balancing and distribution among disks in a JBOD array. The value must be appropriately set.

If the value is smaller than **max_bytes_to_merge_at_max_space_in_pool/1024**, the ClickHouse server process fails to be started and unnecessary parts move between disks.

If the value of **min_bytes_to_rebalance_partition_over_jbod** is greater than that of **max_data_part_size_bytes** (maximum size of parts that can be stored on disks in one array), no part can meet the condition for automatic balancing.

2 Doris

2.1 Table Creation Rules

This topic describes the rules and suggestions for creating Doris tables.

Doris Table Creation Rules

- When creating a Doris table and specifying buckets, make sure that each bucket contains data ranging from 100 MB to 3 GB. Additionally, ensure that the maximum number of buckets in a single partition does not exceed 5,000.
- You must set a bucketing policy for tables that have over 500 million data records.
- Do not set too many bucketing columns in a table. Generally, one or two columns are enough. In addition, you need to ensure even data distribution and balanced query throughput.
  - Data should be evenly distributed to prevent data skew in some buckets, which affects data balancing and query efficiency.
  - The query throughput is reduced with bucketing and tailoring of query SQL statements to avoid full bucket scanning and improve query performance.
  - Preferentially use columns with evenly distributed data and those that are commonly used as query conditions as bucketing columns.

    You can use the following methods to analyze whether data skew occurs:

    SELECT a, b, COUNT(*) FROM tab GROUP BY a,b;

    Once the command is executed, verify if the variation in the number of data records among the groups is minimal. If the difference surpasses 2/3 or 1/2, select another bucket field.
- Do not use dynamic partitioning for less than 20 million data records. Dynamic partitioning generates partitions automatically, but users may overlook small tables. Consequently, numerous useless buckets are created in partitions.
- When creating a table, make sure to have three to five sort keys. Having too many sort keys can impede data writing and importing.

- If Auto Bucket is not used, bucketing should be determined by the data volume to enhance the performance of data import and query. Auto Bucket causes superfluous tablets and a large number of small files.
- Set at least 2 replicas when you create a table. The default replication factor is 3. Do not use a single backup.
- Do not create a table that does not have an aggregate function column as an AGGREGATE table.
- When creating a primary key table, ensure that the primary key column is unique. Do not set all columns as primary key columns. Set a value column for the primary key table. Do not use primary key tables in data deduplication scenarios.

Doris Table Creation Suggestions

- Use no more than six materialized views in a single table. Do not nest materialized views or them in ETL tasks such as heavy aggregations and joins during data writing.
- If there are many historical partitions for a little historical data and the data is unbalanced or the data query probability is low, you can create historical partitions on a yearly/monthly basis and store all historical data in the corresponding partitions.

  To create history partition, use **FROM ("***2000-01-01***") TO ("**i**") INTERVAL 1 YEAR**.
- If the data volume is less than 10 million to 200 million, you do not need to set partitions (the Doris has a default partition). Instead, you can directly use the bucket policy.
- If more than 30% data skew occurs in the bucketing fields, do not use the hash bucketing policy. Instead, use the random bucketing policy. The related commands are as follows:

  Create table ... **DISTRIBUTED BY RANDOM BUCKETS** *10* ...
- The first field must be the most frequently queried one in the table you created. By default, you can quickly query data with prefix indexes. Select the column that is most frequently queried and has a high cardinality as the prefix index. The first 36 bytes of a row are used as the prefix index of the row by default (for varchar columns, the first 20 bytes are matched as the prefix index, and excessive bytes are truncated).
- To fuzzy match or use equivalent/in conditions in a query of more than 100 million data records, use inverted indexes (supported since Doris 2.x) or Bloomfilter. For orthogonal queries with low cardinality columns, use bitmap indexes. (The recommended cardinality of bitmap indexes ranges from 10000 to 100000.)
- Plan the number of fields to be used when creating a table. You can reserve dozens of integer or character fields. If fields are insufficient, you need to add fields temporarily at a high cost.

2.2 Data Change

This topic describes the rules and suggestions for changing Doris data.

Doris Data Change Rules

- Do not directly use the **delete** or **update** statement to change data. Instead, use the **upsert** of the CDC.
- Avoid frequently adding or deleting fields in tables during peak hours. Reserve fields for future use when you create tables. If fields must be added or deleted, or field types and comments must be modified, stop writing and modifying tasks on the target table during off-peak hours and then re-create a table.
  a. Create a table. The structure of the table is the same as that of the table you want to modify. Add new fields to the new table, delete unnecessary fields, or change field types.
  b. Specify fields and insert them to the newly created table.

     INSERT INTO *Newly created table* **SELECT** *Specified fields* **FROM** *Existing table whose columns need to be modified*;

     NOTE

     To prevent high CPU or memory usage and minimize the impact on query service, you can import data to a new table in batches based on time if the table has a significant amount of data. The command is as follows:

     insert into *tab1* **select** *col* **from** *tab* **where** *date <= xx*;
  c. Exchange the names of the two tables. For more information, see **Exchange Tables**.

     ALTER TABLE *[db.]tbl1* **REPLACE WITH TABLE** *tbl2* **[PROPERTIES('swap' = 'true')];**
- Some queries may take a long time and consume a lot of memory and CPU resources. You can set the query timeout parameter **query_timeout** that works on SQL statements or for a user.

Doris Data Change Suggestions

When you run large SQL statements, set session variables with **hint** by using a method similar to **SELECT /*+ SET_VAR(query_timeout = xxx*/ from table**. Do not set global system variables.

2.3 Naming Conventions

This topic describes the rules and suggestions for naming databases and tables.

Doris Naming Rules

The database character set must be UTF-8 and only UTF-8 is supported.

Doris Naming Suggestions

- Set database names in lowercase and use underscores (_) to link words. A name must be less than 62 bytes.
- Table names of Doris are case sensitive. Set a name in lowercase and use underscores (_) to link words. A name must be less than 64 bytes.

2.4 Data Query

This topic describes the rules and suggestions for querying Doris data.

Doris Data Query Rules

- When you are using the data query code, retry the query and issue the query again if the query fails.
- If the enumerated value of the constant **in** exceeds 1000, you must use a subquery.
- Do not use the Statement Execution Action REST APIs to execute a large number of SQL queries. These interfaces are used only for cluster maintenance.
- When dealing with query results exceeding 50,000 records, consider using either JDBC Catalog or the OUTFILE method to export the data. Otherwise, allowing a large amount of data to accumulate on the front end (FE) can impact cluster stability.
  - When performing interactive queries, export data using pagination with an offset limit. You can achieve this by using the ORDER BY command.
  - If data is exported for a third party, use **outfile** or **export**.
- Utilize Colocation Join for joining more than two tables with over 300 million records.
- Avoid using **select *** for querying large tables with hundreds of millions of records and specify the required fields instead.
  - Do not use **select *** when you use SQL Block.
  - For high-concurrency point queries, enable row-based storage (supported by Doris 2.x) and use PreparedStatement.
- Bucketing conditions must be set for queries of tables of hundreds of millions records.
- Do not perform full-partition scan on partitioned tables.

Doris Data Query Suggestions

- When an **INSERT INTO SELECT** statement inserts over 100 million data records, divide them into smaller batches for execution.
- Do not use OR as a JOIN condition.
- Do not frequently delete and modify data. Instead, delete data in batches occasionally with conditions to improve system stability and deletion efficiency.
- To return som data after sorting a large amount of data (more than 500 million records), reduce the data range for sorting. Sorting a large amount of data affects query performance. The following is an example:

  Instead of using **from table order by datatime desc limit** *10*, use **from table where datatime='***2023-10-20***' order by datatime desc limit** *10*.
- Pay attention to the following points when using **parallel_fragment_exec_instance_num** to optimize query task performance:

This parameter determines the maximum number of fragments that can run simultaneously at the session level. Too many concurrent fragments will use up a significant amount of CPU resources. You can leave this parameter blank. If you need to set this parameter to accelerate query speed, comply with the following rules:

- Do not set this parameter to take effect globally, that is, do not use the **set global** command to set this parameter.
- Set this parameter to an even number (2 or 4). The maximum value cannot exceed half of the number of CPU cores on a single node.
- Check the CPU usage before you set the parameter. You can set this parameter only when the CPU usage is less than 50%.
- If you use **insert into select** to insert a large amount of data, do not set this parameter.

2.5 Data Import

This topic describes the technical suggestions for importing Doris data.

Doris Data Import Suggestions

- Do not frequently perform the **update**, **delete**, or **truncate** operation. Perform an operation every several minutes. To use the **delete** operation, you must set the partitioning condition or primary key column.
- Avoid using **INSERT INTO** *tbl1* **VALUES("1"),("a");** to frequently import small amounts of data. Instead, opt for StreamLoad, BrokerLoad, SparkLoad, or Flink Connector.
- When Flink writes data to Doris in real time, set the checkpoint based on the data volume of each batch. If the data volume of each batch is too small, a large number of small files will be generated. The recommended value is 60s.
- Do not use **insert values** as the main data write mode. StreamLoad, BrokerLoad, or SparkLoad is recommended for batch data import.
- If there are downstream dependencies or queries when you use **INSERT INTO WITH LABEL** *XXX* **SELECT** to import data, check whether the imported data is visible.

  Run the **show load where label=** *'xxx'* SQL command to check whether the current INSERT task is **VISIBLE**. The imported data is visible only when the status is **VISIBLE**.
- Streamload is suitable for importing data of less than 10 GB, and Brokerload is suitable for data of less than 100 GB. For large-scale data, use SparkLoad.
- Do not use Routine Load of Doris to import data. Instead, use Flink to query Kafka data and then write the data to Doris. This limits the amount of data to be imported in a single batch and avoids a large number of small files. If Routine Load has already been used to import data, set **max_tolerable_backend_down_num** to **1** on the FE before you change the import method to improve reliability.
- Import data in batches at a low frequency. The average interval for importing a single table must be greater than 30s. Import 1000 to 100000 rows of data each time at a recommended interval of 60s.

2.6 UDF Development

This topic describes the rules and suggestions for developing Doris UDF programs.

Doris UDF Development Rules

- The UDF invocation must be thread-safe.
- Do not load external large files to the memory when implementing a UDF. Otherwise, the memory could be used up.
- Avoid a large number of recursive calls. Otherwise, stack overflow or OOM may occur.
- Do not create objects or arrays continuously. Otherwise, the memory could be used up.
- Use a Java UDF to capture and process possible exceptions. Do not send exceptions to services. Use the try-catch block to handle exceptions and record exception information if necessary.
- In the UDF, do not define static collection classes for storing temporary data or query large objects in external data. Otherwise, the memory usage is high.
- Ensure that the imported packages in the class do not conflict with the packages on the server. You can run the **grep -lr** *"Fully-qualified class name"* command to check JAR package conflicts. Use fully-qualified class names to avoid such conflicts.

Doris UDF Development Suggestions

- To prevent stack memory overflow, do not copy a large amount of data.
- Do not concatenate a large number of strings. Otherwise, the memory usage is high.
- Java UDFs should use meaningful names so that other developers can easily understand their purpose. Use camel-case names and end a name with a UDF, for example, *MyFunctionUDF*.
- A Java UDF should specify the data type of the return value and must have a return value. Do not set the return value to **NULL** when it should be the default value or when an exception occurs. Use basic data types or Java classes as return value types.

2.7 Connection and Running

This topic describes the specifications you need comply with when you connect to Doris and run Doris tasks.

- Use an ELB to connect to Doris to prevent services interruption when the connected FE is faulty.
- If a hardware fault or a single Doris instance failure occurs, Doris can continue executing running tasks, but newly submitted tasks cannot run. You should enable retry upon failures so that submitted jobs can still run in case of unknow exceptions.

3 Flink

3.1 Flink Specification Overview

Scope

This document describes the design and development rules of using the MRS Flink component for data lake and data house integration and unified stream and batch data processing. The following specifications are provided:

- Data table design
- Resource configurations
- Performance tuning
- Fault handling
- Typical parameter settings

Terms

This section uses the following terms for description:

- **Rule**: a principle that must be observed during programming.
- **Suggestion**: a principle that must be considered during programming.
- **Description**: an explanation of the rule or suggestion in question.
- **Example**: an example for a rule or suggestion.

Application Scope

- Design, develop, test, and maintain data storage and data processing jobs based on MRS-Flink data storage.
- Use this specification for MRS 3.2.0 and later versions.
- Parameter optimization guide is available for MRS 3.2.0 and later versions.
- For any inconsistency between this document and the open source community, use the specifications in this document.

Reference

Flink open-source community: https://nightlies.apache.org/flink/flink-docs-stable/.

3.2 FlinkSQL Connector Development

3.2.1 Development Rules

Create a ClickHouse Table in Advance

If a Flink job cannot find a specified table in ClickHouse, an error will be reported. You need to ensure that the table has been created in ClickHouse.

Deletion Is Not Supported When Writing ClickHouse Data with Flink

Flink cannot roll back the write of ClickHouse data because the deletion operation is not supported. The withdrawal streams generated when Flink processes updated data cannot be executed in ClickHouse. As a result, the data processing result is incorrect.

This also affects data deletion when Flink CDC is used to connect to the upstream database to write ClickHouse data. When physical operations are performed on the upstream database, data in ClickHouse cannot be deleted synchronously.

3.2.2 Development Suggestions

Configure Multiple IP Addresses for the ClickHouseBalancer Instance

Configuring multiple IP addresses can prevent single point of failure (SPOF) for ClickHouseBalancer. The configuration (with properties) is as follows:

```
'url' = 'jdbc:clickhouse://IP address 1 of the ClickHouseBalancer instance:ClickHouseBalancer port,IP address 2 of the ClickHouseBalancer instance:ClickHouseBalancer port/default',
```

Configure Proper Batch Parameters for Sink Tables

Parameters for batch write:

Flink stores data in the memory and flushes the data to the database table when the trigger condition is met.

Configurations:

- **sink.buffer-flush.max-rows**: number of rows written to ClickHouse. The default value is **100**
- **sink.buffer-flush.interval**: interval for batch write. The default value is **1s**.

If either of the two conditions is met, a sink operation is triggered. That is, data will be flushed to the database table.

- Example 1: sink every 60 seconds

```
'sink.buffer-flush.max-rows' = '0',
'sink.buffer-flush.interval' = '60s'
```


- Example 2: sink every 100 records
  ```
  'sink.buffer-flush.max-rows' = '100',
  'sink.buffer-flush.interval' = '0s'
  ```
- Example 3: no sink
  ```
  'sink.buffer-flush.max-rows' = '0',
  'sink.buffer-flush.interval' = '0s'
  ```

Create the ReplacingMergeTree Table in the ClickHouse for Data Deduplication

When Flink writes data to ClickHouseBalancer, data with the same key cannot be written to the same ClickHouseServer. The merge of data with the same key depends on the ReplacingMergeTree engine of ClickHouse.

3.2.3 Development Rules

Create a Doris Table in Advance

If a Flink job cannot find a specified table in Doris, an error will be reported. You need to ensure that the table has been created in Doris.

Enable Checkpoint When Doris Is the Sink

Flink jobs write data to the Doris table only when a checkpoint is triggered.

3.2.4 Development Rules

Topic Must Be Specified When Kafka Is the Sink

[Example] Insert a message to the **test_sink** topic in Kafka.

```
CREATE TABLE KafkaSink(
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `age` INT
) WITH (
  'connector' = 'kafka',
  'topic' = 'test_sink',
  'properties.bootstrap.servers' ='Service IP address of the Kafka Broker instance:Kafka port',
  'scan.startup.mode' = 'latest-offset',
  'value.format' = 'csv',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.kerberos.domain.name' = 'hadoop.System domain name'
);
INSERT INTO KafkaSink (`user_id`, `user_name`, `age`)VALUES ('1', 'John Smith', 35);
```

properties.group.id Must Be Specified When Kafka Is the Source

[Example] Use **testGroup** as the user group to read Kafka messages whose topic is **test_sink**.

```
CREATE TABLE KafkaSource(
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `age` INT
) WITH (
  'connector' = 'kafka',
  'topic' = 'test_sink',
```



```
  'properties.bootstrap.servers' ='Service IP address of the Kafka Broker instance:Kafka port',
  'scan.startup.mode' = 'latest-offset',
  'properties.group.id' = 'testGroup',
  'value.format' = 'csv',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.kerberos.domain.name' = 'hadoop.System domain name'
);
SELECT * FROM KafkaSource;
```

Do Not Set Both topic-pattern and topic

topic-pattern: topic pattern, which is used for the source table. The topic name supports the regular expressions.

[Example] Subscribe to all topic messages that start with **test-topic-** and end with a single digit for the source table:

```
CREATE TABLE payments (
    payment_id INT,
    customer_id INT,
    payment_date TIMESTAMP(3),
    payment_amount DECIMAL(10, 2)
) WITH (
    'connector' = 'kafka',
    'topic-pattern' = 'test-topic-[0-9]',
    'properties.bootstrap.servers' = 'localhost:9092',
    'format' = 'json'
);
SELECT * FROM payments WHERE payment_amount < 500;
```

3.2.5 Development Suggestions

Traffic Limiting Must Be Set When Kafka Is the Source

This rule is available for MRS 3.3.0 or later.

To prevent job exceptions caused by heavy traffic, set a traffic limit, which should be the peak value of the pressure test for service rollout.

[Example]

```
# The following parameter takes effect at any parallelism:
'scan.records-per-second.limit' = '1000'
# The actual traffic limit is as follows:
min( parallelism * scan.records-per-second.limit, partitions num * scan.records-per-second.limit)
```

Write Data with the Same Key to the Same Kafka Partition for Data Accuracy

Flink uses the **fixed** policy to write data to Kafka and performs hash calculation based on the key before writing data.

[Example]

```
CREATE TABLE kafka (
 f_sequence INT,
 f_sequence1 INT,
 f_sequence2 INT,
 f_sequence3 INT
 ) WITH (
 'connector' = 'kafka',
 'topic' = 'yxtest123',
```

```
'properties.bootstrap.servers' = '192.168.0.104:9092',
'properties.group.id' = 'testGroup1',
'scan.startup.mode' = 'latest-offset',
'format' = 'json',
'sink.partitioner'='fixed'
);

insert into kafka select /*+ DISTRIBUTEBY('f_sequence','f_sequence1') */ * from datagen;
```

Set Kafka Source Parallelism Same as the Number of Topic Partitions for Faster Kafka Consumption

When the parallelism of Kafka Source is greater than the number of topic partitions, no more data is consumed.

3.2.6 Development Rules

Create an HBase Table in Advance

If a Flink job cannot find a specified table in HBase, an error will be reported. You need to ensure that the table has been created in HBase.

Only Flink and HBase in Normal Clusters Can Be Interconnected

When HBase and Flink are in the same cluster or clusters with mutual trust, FlinkServer can be interconnected with HBase.

If HBase and Flink are in different clusters without mutual trust, Flink in a normal cluster can be interconnected with HBase in a normal cluster.

Configure HBASE_CONF_DIR When FlinkServer Interconnects With HBase

Step 1 Log in to the node where the client is installed as the client installation user and copy all configuration files in the **/opt/client/HBase/hbase/conf/** directory of HBase to an empty directory of all nodes where FlinkServer is deployed, for example, **/tmp/client/HBase/hbase/conf/**.

Change the owner of the configuration file directory and its upper-layer directory on the FlinkServer node to **omm**.

chown omm: /tmp/client/HBase/ -R

NOTE

- FlinkServer nodes:

  Log in to FusionInsight Manager, choose **Cluster** > **Services** > **Flink** > **Instances**, and check the **Service IP Address** of FlinkServer.
- If the node where a FlinkServer instance is deployed is the node where the HBase client is installed, skip this step on this node.

Step 2 Log in to Manager and choose **Cluster** > **Services** > **Flink**. Click **Configurations** then **All Configurations**, search for the **HBASE_CONF_DIR** parameter, and enter the FlinkServer directory (for example, **/tmp/client/HBase/hbase/conf/**) to which the HBase configuration files are copied in **Step 1** in **Value**.

NOTE

If the node where a FlinkServer instance resides is the node where the HBase client is installed, enter the **/opt/client/HBase/hbase/conf/** directory of HBase in **Value** of the **HBASE_CONF_DIR** parameter.

Step 3 After the parameters are configured, click **Save**. After confirming the modification, click **OK**.

Step 4 Click **Instances**, select all FlinkServer instances, choose **More** > **Restart Instance**, and operate as prompted.

----End

3.2.7 Development Suggestions

Add HBase Configuration Using the With Properties When Submitting a Job on the Client

Submit a job on the Flink client. For example, on a SQL client, add the following configuration to the table creation statement:

Table 3-1 Flink job with properties

Configuration	Description
'properties.hbase.rpc.protection' = 'authentication'	This parameter must be consistent with that on the HBase server.
'properties.zookeeper.znode.parent' = '/hbase'	If there are multiple services, hbase1 and hbase2 coexist. You must clarify the cluster to be accessed.
'properties.hbase.security.authorization' = 'true'	This parameter is used to enable authentication.
'properties.hbase.security.authentication' = 'kerberos'	This parameter is used to enable Kerberos authentication.

[Example]

```
CREATE TABLE hsink1 (
    rowkey STRING,
    f1 ROW < q1 STRING >,
    PRIMARY KEY (rowkey) NOT ENFORCED
   ) WITH (
     'connector' = 'hbase-2.2',
     'table-name' = 'cc',
     'zookeeper.quorum' = 'x.x.x.x:clientPort',
     'properties.hbase.rpc.protection' = 'authentication',
     'properties.zookeeper.znode.parent' = '/hbase',
     'properties.hbase.security.authorization' = 'true',
     'properties.hbase.security.authentication' = 'kerberos'
  );
```

Enable Asynchronous Lookup Join for Faster Dimension Table Join

Add the following **with** property for the HBase dimension table:

```
'lookup.async'='true'
```

Increase the Parallelism of the Lookup Join Operator for Faster Dimension Table Join

Add the following **with** property for the HBase dimension table:

```
'lookup.parallelism'='xx'
```

Increase the Parallelism of the Sink HBase Operator for Higher Write Performance

Add the following **with** property for the HBase sink table:

```
'sink.parallelism'='xx'
```

3.3 Flink on Hudi

3.3.1 Development Rules

The following table describes the parameter specifications you need to comply with to read Hudi tables on Flink streams.

Table 3-2 Parameter specifications

Parameter	Mandatory	Description	Example
Connector	Yes	Type of the table to be read	hudi
Path	Yes	Path for storing the table	Set this parameter based on site requirements.
table.type	Yes	Hudi table type. The default value is **COPY_ON_WRITE**.	MERGE_ON_READ
hoodie.datasource.write.recordkey.field	Yes	Primary key of the table	Set this parameter as needed.
write.precombine.field	Yes	Data combination field	Set this parameter as needed.
read.tasks	No	Hudi table read parallelism. The default value is **4**.	4

Parameter	Mandatory	Description	Example
read.streaming.enabled	Yes	• **true**: Data is read on streams incrementally. • **false**: Data is read in batches.	Set this parameter based on the site requirements. For streaming read, set this parameter to **true**.
read.streaming.start-commit	No	Start commit (closed interval) in the yyyyMMddHHmmss format. By default, the latest commit is used.	-
hoodie.datasource.write.keygenerator.type	No	Primary key generation type of the upstream table	COMPLEX
read.streaming.check-interval	No	Check interval for finding new source commits. The default value is 1 minute.	5 (The default value is recommended for heavy traffic.)
read.end-commit	No	• Incremental stream consumption. Use **read.streaming.start-commit** to specify the start position. • Batch incremental consumption. Use **read.streaming.start-commit** to specify the start position, and the **read.end-commit** to specify the end position (closed interval). The start and end positions are included. By default, the latest commit is the end position.	-
changelog.enabled	No	Whether to write changelog messages. The default value is **false**. Set this parameter to **true** for CDC.	false

3.3.2 Suggestions

Set Proper Consumption Parameters to Avoid "File Not Found"

When the downstream consumes Hudi files too slowly, the upstream archives the Hudi files. As a result, the "File Not Found" error occurs.

- Increase the value of **read.tasks**.
- If there is a triffic limit, increase the upper limit.
- Increase the upstream **compaction**, **archive**, and **clean** parameters.

3.3.3 Development Rules

Parameter Specifications

The following table describes the parameter specifications you need to comply with to write Hudi tables on Flink streams.

Table 3-3 Parameter specifications

Parameter	Mandatory	Description	Recommended Value
Connector	Yes	Type of the table to be read	hudi
Path	Yes	Path for storing the table	Set this parameter as required based on service requirements.
hoodie.datasource.write.recordkey.field	Yes	Primary key of the table	Set this parameter as required based on service requirements.
write.precombine.field	Yes	Data combination field	Set this parameter as required based on service requirements.
write.tasks	No	Hudi table write parallelism. The default value is **4**.	4

Parameter	Mandatory	Description	Recommended Value
index.bootstrap.enabled	No	Flink uses the memory index, which caches the primary key of data to the memory to ensure unique data in the target table. This parameter must be set. Otherwise, data may be duplicate. The default value is **true**. The default value is **FALSE**. Do not set this parameter when bucketing indexes are used.	TRUE
write.index_bootstrap.tasks	No	This parameter is valid only after **index.bootstrap.enabled** is enabled. Increase the number of tasks to accelerate startup.	4
index.state.ttl	No	Duration for storing index data. The default value is **0**, indicating that the index data is permanently valid. You can change the value based on the service requirements.	0
compaction.delta_commits	No	Condition for triggering the compaction plan of the MOR table	200
compaction.async.enabled	Yes	Whether to enable online compaction. The compaction operation is transferred to SparkSQL to improve the write performance.	FALSE
hive_sync.enable	No	Whether to synchronize table information to Hive.	True
hive_sync.metastore.uris	No	Hivemeta URI	Set this parameter as required based on service requirements.
hive_sync.jdbc_url	No	Hive JDBC link	Set this parameter as required based on service requirements.

Parameter	Mandatory	Description	Recommended Value
hive_sync.table	No	Hive table name	Set this parameter as required based on service requirements.
hive_sync.db	No	Name of the Hive database. The default value is **default**.	Set this parameter as required based on service requirements.
hive_sync.support_timestamp	No	Whether to support timestamps	True
changelog.enabled	No	Whether to write changelog messages. The default value is **false**. Set this parameter to **true** for CDC.	false

Table Name Must Meet Hive Format Requirements

- Must start with a letter or underscore (_) and cannot start with a digit.
- Can contain only letters, digits, and underscores (_).
- Can contain a maximum of 128 characters.
- Cannot contain spaces or special characters, such as colons (:), semicolons (;), and slashes (/).
- Is case insensitive. **Lowercase letters are recommended.**
- Cannot be Hive reserved keywords, such as **select**, **from**, and **where**.

[Example]

```
my_table, customer_info, sales_data
```

3.3.4 Development Suggestions

- Use spark SQL to centrally create tables.
- Use Spark asynchronous tasks to compact Hudi tables.

3.3.5 Configuration Rules

Flink Job Parameter Configuration Specifications

The following table describes the rules for configuring Flink job parameters.

Table 3-4 Parameter configuration specifications

Parameter	Mandatory	Description	Recommended Value
-c	Yes	Main class name	Set this parameter as you need.
-ynm	Yes	Flink YARN job name	Set this parameter as you need.
execution.checkpointing.interval	Yes	Interval for triggering a checkpoint, which can be added using -**yD**. The unit is ms.	60000
execution.checkpointing.timeout	Yes	Checkpoint timeout interval. You can run the **-yD** command to add a checkpoint timeout interval. The default value is 30 minutes.	30min
parallelism.default	No	Job parallelism. For example, to add the job parallelism for the join operator, use **-yD**. The default value is **1**.	Set this parameter based on the site requirements.
table.exec.state.ttl	Yes	TTL (join ttl) of Flink state, which can be added using **-yD**. The default value is **0**.	Set this parameter based on the site requirements.

Checkpoint Interval Should Be Longer Than the Checkpoint Execution Duration

The checkpoint execution duration depends on checkpoint data volume. The larger the data volume, the longer the execution duration.

Checkpoint Timeout Duration Should Be Longer Than the Checkpoint Interval

The checkpoint interval indicates the interval for triggering a checkpoint. If the execution duration is longer than the checkpoint timeout interval, the job fails.

If CDC is used, changelog needs to be enabled for Hudi table read and write.

To ensure Flink calculation accuracy when CDC is used, retain +I, +U, -U, and -D in Hudi tables. Changelog must be enabled when data is written to or read from the same Hudi table.

3.3.6 Configuration Suggestions

Set Traffic Limit When a Hudi Table Is the Source

To prevent job exceptions caused by heavy traffic, set a traffic limit, which should be the peak value of the pressure test for service rollout.

Add the following parameter:

```
'read.rate.limit' = '1000'
```

Set execution.checkpointing.tolerable-failed-checkpoints

For Flink On Hudi jobs, set checkpoint tolerance times to a larger value, for example, 100.

3.4 Flink Jobs

3.4.1 Development Rules

Ensure Data Accuracy When Aggregating Updated Data

When aggregating updated data, you need to select a proper solution. Otherwise, the aggregation result can be incorrect.

The following statement is an example:

```
Create table t1(
    id  int,
    partid int,
    value int
);
select
   partid,sum(value)
 from t1
 group by partid;
```

- First batch of data: [1,1,10],[2,1,11],[3,2,8]

  Aggregation result: [1,21], [2,8]

- Second batch of data: [2,1,12] //Update the record whose ID is 2.

  Error result: [1,33], [2,8] //Updated data (ID=2) cannot be identified.

  Aggregation result: [1,22], [2,8] //The result is correct because update is identified.

There are three ways to identify whether the data is updated:

- Using the state backend

  The state backend stores all raw data. The new arriving data is determined as the updated data based on the status. Then, the Flink aggregation withdrawal function is used to update the aggregation result data.

  Advantage: The aggregation accuracy is ensured. This solution has no requirement on data which is easy to use.

  Disadvantage: When there is a large amount of data, large state backend storage is required.

- Using data in CDC format

  The update operation record of CDC data contains both original data and updated data. The previous aggregation result is removed based on original data, and the latest calculation result is updated based on updated data.

  Advantage: Large state backend storage is not required. The overall compute resource pressure is lower than that of the state backend solution.

  Disadvantage: This solution depends on the CDC format. Typically, data is captured by a CDC collection tool and sent to Kafka, and then Flink reads Kafka data for calculation.

- Using changelog data

  The changelog format is similar to the CDC format. The only difference is that the CDC format records original and updated data in one row, and the changelog format stores updated data in two rows. One row is used to delete the original data, and the other row stores insertion operation records of updated data. Flink deletes the aggregation result based on the updated data and inserts the calculation result based on the updated data. Changelog can be implemented based on Hudi tables. Data in CDC format can be converted into changelog data and stored in the log files of Hudi MOR tables. Changelog data of Hudi can also be generated based on state backends.

  Advantage: Aggregation consistency of updated data can be ensured based on data lake storage.

  Disadvantage:

  – Only the log file of a Hudi MOR table contains changelog data. If upstream data is stacked when Flink job calculation is delayed and the log file is cleared, changelog data will be lost. You need to retain more versions and properly configure resources for Flink jobs to make the data stacking period shorter than the clearance period.
  – The generation of changelogs based on the state backend also depends on the state backend. Generally, the TTL is configured for the state backend and is not retained permanently. In this scenario, the update operation is arbitrary and there is no update period limit. For example, to update data of the last month, set TTL to a value greater than one month. To update all data, set TTL to permanent (do not do so for large tables).
  – Currently, the MOR table of changelogs can only be compacted by the Flink engine. Spark engine is not available for compaction.

3.4.2 Development Suggestions

Consider Increasing the Number of Checkpoints for High Availability

By default, only the latest checkpoint status file is saved. If this file is unavailable (for example, all copies of the HDFS file are damaged), state restoration fails. If we keep two state file checkpoints, Flink rolls back to the state file of the previous one even if the latest checkpoint is unavailable. You can increase the number of reserved checkpoints as needed.

[Example] Set the number of reserved checkpoint files to 2.

```
state.checkpoints.num-retained: 2
```


Use Incremental RocksDB as the State Backend in the Production Environment

Flink provides three state backends: MemoryStateBackend, FsStateBackend, and RocksDBStateBackend.

- MemoryStateBackend stores states on the Java heap memory of JobManager. Each state cannot be bigger than an akka frame, and the total size cannot exceed the heap memory size of JobManager. This state backend is suitable for local development and debugging or small-size states.
- FsStateBackend is the file system state backend. Generally, states are stored in the TaskManager heap memory. In checkpointing, states are stored in the file system. The JobManager memory stores only a small amount of metadata (which is stored in ZooKeeper in HA scenarios). Since there is sufficient storage space in the file system, this backend is suitable for stateful processing tasks with large states, long window, or large key value states, and is also suitable for the HA solution.
- RocksDBStateBackend is an embedded database backend. Generally, states are stored in the RocksDB database, and the database data is stored on the local disk. In checkpointing, states are stored in the file system, and the JobManager memory stores a small amount of metadata (which is stored in ZooKeeper in HA scenarios). This state backend is the only one that supports incremental checkpointing. In addition to same scenarios of the FsStateBackend, it is also suitable for processing ultra-large states.

Table 3-5 Flink state backends

Type	MemoryStateBack-end	FsStateBackend	RocksDBStateBack-end
Method	Checkpoint data is directly returned to the master node and is not flushed to disks.	Data is written to a file whose path is then sent to the master node.	Data is written to a file whose path is then sent to the master node.
Storage	Heap memory	Heap memory	RocksDB (local disk)
Performance	Best performance among the three (generally not used)	High performance	Poor performance
Disadvantage	Small data volume only and easy data loss	OOM	Time-consuming read/write, serialization, and I/O
Incremental	Not supported	Not supported	Supported

[Example] Configure a RockDBStateBackend (**flink-conf.yaml**):


```
state.backend: rocksdb
state.checkpoints.dir: hdfs://namenode:40010/flink/checkpoints
```

Use EXACTLY ONCE Stream Processing Semantics to Ensure End-to-End Consistency

There are three types of stream processing semantics: EXACTLY ONCE, AT LEAST ONCE, and AT MOST ONCE.

- AT MOST ONCE: The integrity of data cannot be ensured, but the performance is the best.
- AT LEAST ONCE: The integrity of data can be ensured, but the accuracy cannot be ensured. The performance is moderate.
- EXACTLY ONCE: Data processing accuracy can be ensured, but the performance is the worst.

Check whether EXACTLY_ONCE can be ensured. This semantics requires data replay in the source (for example, Kafka message replay) and transactional in the sink (for example, MySQL atomic data writing). If these requirements cannot be met, you can degrade to AT LEAST ONCE or AT MOST ONCE.

- If the source does not support replay, only AT MOST ONCE can be ensured.
- If the sink does not support atomic write, only AT LEAST ONCE can be ensured.

[Example] Use EXACTLY ONCE semantics with API calls:

```
env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)
```

[Example] Set Exactly once semantics in the resource file.

```
# Semantics of checkpoint
execution.checkpointing.mode: EXACTLY_ONCE
```

Locate Back Pressure Point by Monitoring Information

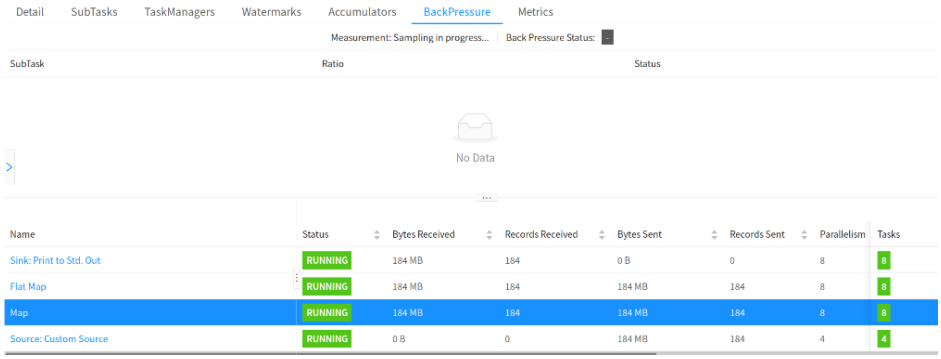
Flink provides many monitoring metrics for you to analyze the performance states and bottlenecks of a job.

[Example] Configure the number of samples and sampling interval.

```
# Sampling interval when the valid backpressure result is discarded and backpressed, in ms
web.backpressure.refresh-interval: 60000
# Number of backpressure samples
web.backpressure.num-samples: 100
# Interval for backpressure sampling, in ms
web.backpressure.delay-between-samples: 50
```

You can view **BackPressure** in the **Overview** tab of the job. The following figure shows that sampling is in progress. By default, sampling takes about 5 seconds.

Figure 3-1 Sampling in progress



As shown in the following figure, **OK** indicates that there is no back pressure, and **HIGH** indicates that a subtask is backpressed.

Figure 3-2 No back pressure

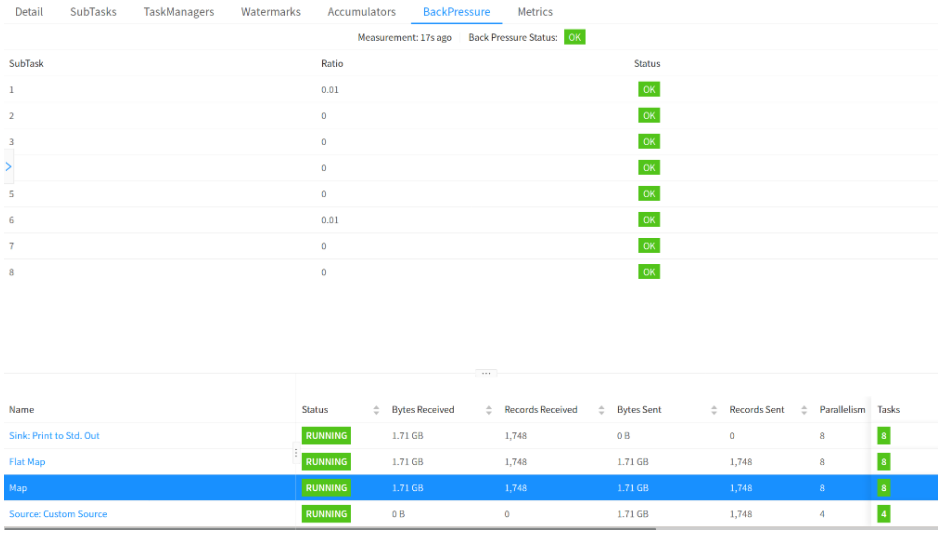
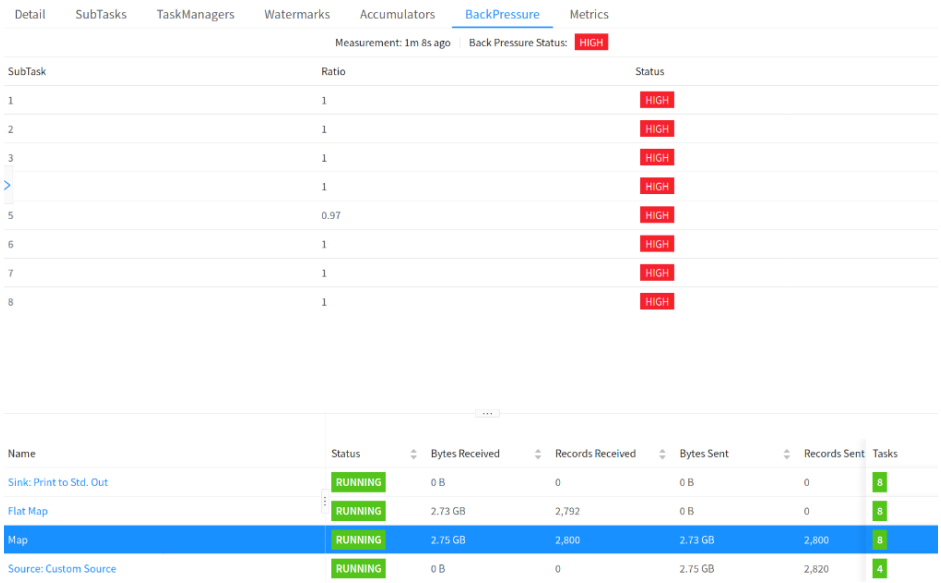


Figure 3-3 Backpressed subtask



Switch to the Hive Dialect When Hive SQL Is Used and the Flink Syntax Is Incompatible

Currently, Flink parses SQL syntax with the default or Hive engine. The former supports Flink native SQL, and the latter supports Hive SQL. DDL and DML of some Hive syntax cannot be run using Flink SQL. You can switch to Hive dialect. Pay attention to the following things when using Hive dialect:

- Hive dialect can only be used to operate Hive tables only. Hive dialects should be used with HiveCatalog.
- Although all Hive versions support the same syntax, whether there are specific functions still depends on the Hive version in use. For example, database location update is supported only in Hive-2.7.0 or later.
- Hive and Calcite have different reserved keywords. For example, **default** is a reserved keyword in Calcite and a non-reserved keyword in Hive. When using Hive dialect, you must use backquotes (`) to reference such keywords so that they can be used as identifiers.
- Views created in Flink cannot be queried in Hive.

[Example] Use Hive syntax to parse SQL statement (**sql-submit-defaults.yaml**):

```
configuration:  table.sql-dialect: hive
```

Use Memory Dimension Tables (such as Hudi) for Small- and Medium-scale Data

- In a memory dimension table, dimension data is loaded to the memory. Each TM loads full data and point query joins are performed in the memory. If the data volume is too large, you need to allocate large memory space to the TM. Otherwise, job exceptions may occur.
- In an external dimension table, dimension data is stored in a high-speed K-V database. Point query joins are implemented through remote K-V query. Typical open-source K-V databases include HBase
- State dimension table data is read to streaming jobs in real time as a stream table. Data stream withdrawal is used to ensure data consistency for dimension update and unsynchronized data. Dimension tables are stored for a long time. Currently, Flink on Hudi allows you to set the TTL for a Hudi dimension table.

Table 3-6 Comparison of dimension table implementations

Dimension	Memory Dimension Table (Hive/ Hudi)	External Dimension Table (HBase)	State Dimension Table
Performance	Very high (within milliseconds)	Medium (millisecond-level)	High (within and in milliseconds)
Data volume	Small, less than 1 GB for a single TM	Large, in TB level	Medium, in GB level

Dimension	Memory Dimension Table (Hive/ Hudi)	External Dimension Table (HBase)	State Dimension Table
Storage	High memory consumption, full storage of a single TM	No storage consumption (external storage is used)	Distributed storage for each TM: memory and disks
Timeliness	Periodic data loading, low timeliness	Relatively high	High
Join result	Low	Medium	-

Use HBase for Large Dimension Tables

If the data volume is large and data consistency requirement is not high, use HBase KV databases to support point query joins of dimension tables.

Data in the K-V database needs to be written by another job, which has a time difference with the Flink job. The current Flink job may not query the latest data in the K-V database, and the lookup query does not support cancellation. The association result is inconsistent.

Use Stream Tables as Dimension Tables for High Data Consistency

When you are using a Hudi dimension source table, the TTL of the table can be set separately. Data will not age based on the overall TTL of the job. Dimension data can be stored in the state backend for a long time. In addition, stream tables can be used as dimension tables to ensure data consistency with the Flink withdrawal.

3.5 Flink SQL Logic

3.5.1 Development Rules

Do Not Add Over Five Dimension Tables In Lookup Join

Hudi dimension tables are stored in the TM heap memory. When there are too many dimension tables, heap memory stores too much dimension table data, and the TM will consistently trigger GC. As a result, the job performance deteriorates.

[Example] Set the number of dimension tables in a lookup join to 5:

```
CREATE TABLE table1(id  int, param1 string) with(...);
CREATE TABLE table2(id  int, param2 string) with(...);
CREATE TABLE table3(id  int, param3 string) with(...);
CREATE TABLE table4(id  int, param4 string) with(...);
CREATE TABLE table5(id  int, param5 string) with(...);
CREATE TABLE orders (
```



```
    order_id    STRING,
    price       DECIMAL(32,2),
    currency    STRING,
    order_time  TIMESTAMP(3),
    WATERMARK FOR order_time AS order_time
) WITH (/* ... */);

select
   o.*, t1.param1, t2.param2, t3.param3, t4.param4, t5.param5
from
   orders AS o
   JOIN table1 FOR SYSTEM_TIME AS OF o.proc_time AS t1 ON o.order_id = t1.id
   JOIN table2 FOR SYSTEM_TIME AS OF o.proc_time AS t2 ON o.order_id = t2.id
   JOIN table3 FOR SYSTEM_TIME AS OF o.proc_time AS t3 ON o.order_id = t3.id
   JOIN table4 FOR SYSTEM_TIME AS OF o.proc_time AS t4 ON o.order_id = t4.id
   JOIN table5 FOR SYSTEM_TIME AS OF o.proc_time AS t5 ON o.order_id = t5.id;
```

In a multi-stream join, the number of fact stream tables cannot exceed three.

When there are too many tables, the back-end pressure is too high, increasing the latency.

[Example] Join three dimension tables in real time.

```
CREATE TABLE table1(id  int, param1 string) with(...);
CREATE TABLE table2(id  int, param2 string) with(...);
CREATE TABLE table3(id  int, param3 string) with(...);
CREATE TABLE orders (
    order_id    STRING,
    price       DECIMAL(32,2),
    currency    STRING,
    order_time  TIMESTAMP(3),
    WATERMARK FOR order_time AS order_time
) WITH (/* ... */);

select
   o.*, t1.param1, t2.param2, t3.param3
from
   orders AS o
   JOIN table1 AS t1 ON o.order_id = t1.id
   JOIN table2 AS t2 ON o.order_id = t2.id
   JOIN table3 AS t3 ON o.order_id = t3.id;
```

The number of nested joins cannot exceed three.

A larger number of nesting levels indicates a larger amount of data to be withdrawn.

[Example] Nest three joins.

```
SELECT *
   FROM table1 WHERE column1 IN
(
  SELECT column1
  FROM table2 WHERE column2 IN (
    SELECT column2
    FROM table3 WHERE column3 = 'value'
  )
)
```


A Single Lookup Join Table Should Not Be Larger than 1 GB in Hudi

Hudi dimension tables are stored in the TM heap memory. When a dimension table is too large, heap memory stores too much dimension table data, and the TM will consistently trigger GC. As a result, the job performance deteriorates.

Do Not Add Batch Source Operator to Stream Joins

Changed the Source operator to a dimension table operator based on service requirements.

3.5.2 Development Suggestions

Filter Data Before Aggregate and Join to Reduce Data to Be Calculated

Filtering data before the shuffle phase to reduce network I/Os and improve query efficiency.

For example, filtering data before joining a table is more effective than filtering data when ON or WHERE is executed. The execution sequence is changed to filtering before shuffling.

[Example] Move predicate condition A.userid>10 before the subquery statement to reduce the shuffle data volume.

- SQL statements before optimization:

```
select... from A
join B
on A.key = B.key
where A.userid > 10
    and B.userid < 10
    and A.dt='20120417'
    and B.dt='20120417';
```

- SQL statement after optimization:

```
select ... from (
    select ... from A where dt='201200417' and userid > 10
)a
join (
    select ... from B where dt='201200417' and userid < 10
)b
on a.key = b.key;
```

Exercise Caution When Using the Regular Expression Function REGEXP

Regular expressions are time-consuming and require x100 performance overhead of addition, subtraction, multiplication, and division. In addition, regular expressions may enter an infinite loop in some extreme cases, causing job blocking. You are advised to use LIKE to replace regular expressions. Typical regular expression functions you may use include:

- REGEXP
- REGEXP_EXTRACT
- REGEXP_REPLACE

[Example]

- The following statement uses a regular expression:


```
SELECT
 *
FROM
 table
WHERE username NOT REGEXP ''test|ceshi|tester'
```

- The following statement uses Like for fuzzy query:

```
SELECT
 *
FROM
 table
WHERE username NOT LIKE '%test%'
 AND username NOT LIKE '%ceshi%'
 AND username NOT LIKE '%tester%'
```

Do Not Use Long Expressions When You Nest UDFs

If the expression used to nest UDFs is too long, the code generated after Flink optimization exceeds 64 KB and a compilation error occurs. It is recommended that a maximum of six UDFs be nested.

[Example] Nest UDFs.

```
SELECT
   SUM(get_order_total(order_id))
FROM orders WHERE customer_id = (
   SELECT customer_id FROM customers WHERE customer_name = get_customer_name('John Doe')
)
```

Replace CASE WHEN in Aggregate Functions with FILTER

In aggregate functions, FILTER is a SQL standard clause for data filtering that improves performance. FILTER is a modifier used in aggregate functions to limit the values used in aggregation.

[Example] In the following example, CASE WHEN is used to collect UV statistics from different dimensions, for example, Android UV, iPhone UV, web UV, and total UV.

- Before modification

```
SELECT
day,
COUNT(DISTINCT user_id) AS total_uv,
COUNT(DISTINCT CASE WHEN flag IN (android', ''iphone'') THEN user_id ELSE NULL END) AS app_uv,
COUNT(DISTINCT CASE WHEN flag IN(wap', 'other') THEN user_id ELSE NULL END) AS web_uv
FROM T
GROUP BY day
```

- After modification

```
SELECT
day,
COUNT(DISTINCT user_id) AS total_uv,
COUNT(DISTINCT user_id) FILTER (WHERE flag IN ('android', 'iphone')) AS app_uv,
COUNT(DISTINCT user_id) FILTER(WHERE flag IN ('wap', 'other'))AS web_uv
FROM T
GROUP BY day
```

The Flink SQL optimizer can identify different filter parameters on the same distinct key. In the example, three COUNT DISTINCT are used on the **user_id** column. Flink can use only one shared state instance instead of three to reduce state access and state size. For some workloads, Flink can have significant improvement in performance.

Split Distinct Aggregation to Eliminate Data Skew

Two-phase aggregation can eliminate typical data skew, but the performance of processing distinct aggregation is poor. Even if two-phase aggregation is enabled, distinct keys cannot be combined to eliminate duplicate values, and the accumulator still contains all original records.

Different aggregations (**COUNT(DISTINCT col)**) can be divided into two levels:

For the first aggregation, use group key and additional bucket key to shuffle. The bucket key is calculated using HASH_CODE(distinct_key) % BUCKET_NUM. The default value of **BUCKET_NUM** is **1024**, which can be configured using the **table.optimizer.distinct-agg.split.bucket-num** option.

For the second aggregation, use the original group key to shuffle and SUM to aggregate the **Count DISTINCT** values from different buckets. The same distinct key is calculated in the same bucket only. The conversion is equivalent. The bucket key functions as an additional group key to share the load of hotspots in the group key. The bucket key makes jobs to be scalable to avoid data skew and hotspotting in aggregations.

[Example]

- Add the following configurations in the resource file:
  ```
  table.optimizer.distinct-agg.split.enabled: true
  table.optimizer.distinct-agg.split.bucket-num: 1024
  ```
- Query the number of unique users who have logged in today:
  ```
  SELECT day, COUNT(DISTINCT user_id)
  FROM T
  GROUP BY day
  ```
- The query is rewritten as follows:
  ```
  SELECT day, SUM(cnt)
  FROM(
      SELECT day, COUNT(DISTINCT user_id) as cnt
  FROM T
  GROUP BY day, MOD(HASH_CODE(user_id), 1024)
      )
  GROUP BY day
  ```

Set the Join Field as the Primary Key When Joining Streams

When the join field is not the primary key, Flink uses a hash-based shuffle, which means that the original sequence of data is not preserved. This leads to multiple replicas of the same join key field stored in the state backend, causing a Cartesian product to be generated during the join.

For example, the field in table A is **id, field1**, and the field in table B is **id, field2**. Join tables A and B based on **id**. Table A has historical data (1, a1), and table B has historical data (1, b1). When table A changes from (1, a1) to (1, a2) and table B changes (1, b1) to (1, b2), the join result is as follows and the join result cannot preserve the data sequence:

```
1, a1, b1
1, a2, b1
1, a1, b2
1, a2, b2
```

- SQL statement before optimization:
  ```
  create table t1 (
   id int,
  ```



```
  field1 string
) with(
  ......
);
create table t1 (
  id int,
  field2 string
) with(
  ......
);
select t1.id, t1.field1, t2.field2
from t1
left join t2 on t1.id = t2.id;
```

- SQL statement after optimization:

```
create table t1 (
  id int,
  field1 string,
  primary key (id) not enforced
) with(
  ......
);
create table t1 (
  id int,
  field2 string,
  primary key (id) not enforced
) with(
  ......
);
select t1.id, t1.field1, t2.field2
from t1
left join t2 on t1.id = t2.id;
```

Specify All Fields That Have the Composite Primary Key in the Select Clause When the Join Key Is a Composite Primary Key

If all fields that have the composite primary key are not used in Select, the join operator discards some primary keys. As a result, the join spec is NoUniqueKey.

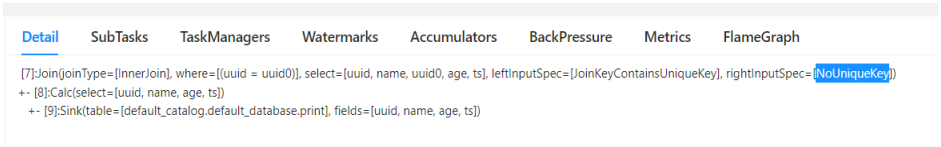
- SQL statements before optimization:

```
create table table1(
  uuid varchar(20),
  name varchar(10),
  age int,
  ts timestamp,
  primary key (uuid) not enforced
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
create table table2(
  uuid varchar(20),
  name varchar(10),
  age int,
  ts timestamp,
  primary key (uuid, name) not enforced
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
create table print(
  uuid varchar(20),
  name varchar(10),
  age int,
  ts timestamp
) with ('connector' = 'print');
insert into
  print
```



```
select
  t1.uuid,
  t1.name,
  t2.age,
  t2.ts
from
  table1 t1
  join table2 t2 on t1.uuid = t2.uuid;
```

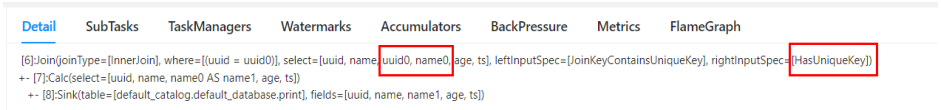
Figure 3-4 NoUniqueKey join spec



- SQL statements after optimization:

```
create table table1(
  uuid varchar(20),
  name varchar(10),
  age int,
  ts timestamp,
  primary key (uuid) not enforced
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
create table table2(
  uuid varchar(20),
  name varchar(10),
  age int,
  ts timestamp,
  primary key (uuid, name) not enforced
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
create table print(
  uuid varchar(20),
  name varchar(10),
  name1 varchar(10),
  age int,
  ts timestamp
) with ('connector' = 'print');
insert into
  print
select
  t1.uuid,
  t1.name,
  t2.name as name1,
  t2.age,
  t2.ts
from
  table1 t1
  join table2 t2 on t1.uuid = t2.uuid;
```

Figure 3-5 Optimized SQL



Use the Snowflake Schema When the Join Key Changes in a Left Join of Tables

Data disorder occurs when the left join key changes. You are advised to associate the right table with a view and then associate the view with the left table.

The change of the join key **group_id** causes the disorder of "-D" and "+I." Although the parallelism degree are the same during the **user_id**-based hashing, the "+I" message arrives first, and the "-D" message arrives later. As a result, the records are overwritten in the wide table.

- SQL statement before optimization:

```
select...
from t1
left join t2 on t2.user_id = t1.user_id
left join t10 on t10.user_id = t1.user_id
left join t11 on t11.group_id = t10.group_id
left join t12 on t12.user_id = t1.user_id
```

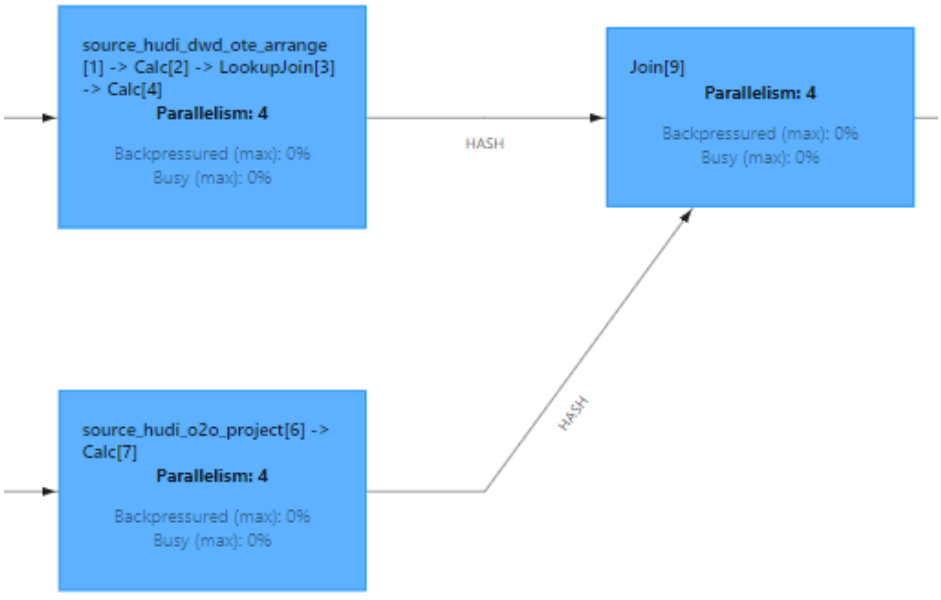
- SQL statement after optimization:

```
create view tmp_view as(
select
..
from t10
left join t11 on t11.group_id = t10.group_id
);
select...
from t1
left join t2 on t2.user_id = t1.user_id
left join tmp_view on tmp_view.user_id = t1.user_id
left join t12 on t12.user_id = t1.user_id
```

Use Lookup Join After All Dual-stream Joins for Table Left Join

Data disorder occurs when you run left join LATERAL TABLE in the downstream if a lookup join is performed before dual-stream joins.

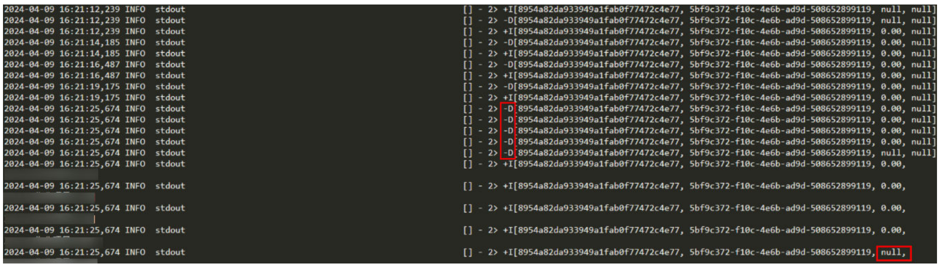
Figure 3-6 Left join of tables



The specified primary key of the left stream cannot be inferred after a lookup join, so all left stream historical data is stored in the state. When right stream data arrives, each left stream record is matched with the latest state, withdrawn, and then associated with the corresponding source data in the LATERAL TABLE. This causes the data to become out of order.

To avoid incorrect data, perform a lookup join after a dual-stream join, as multiple consecutive "-D" messages can cause the last record to be incorrect.

Figure 3-7 Consecutive "-D" messages



- SQL statement before optimization:

```
select...
from t1
left join t2 FOR SYSTEM_TIME AS OF t1.proctime AS t21 on t21.id = t1.id
left join t3 on t3.id = t1.id
left join LATERAL TABLE(udtf()) AS  t4(res1,res2.res3,res4) on true
```

- SQL statement after optimization:

```
select...
from t1
left join t3 on t3.id = t1.id
left join t2 FOR SYSTEM_TIME AS OF t1.proctime AS t21 on t21.id = t1.id
left join LATERAL TABLE(udtf()) AS  t4(res1,res2.res3,res4) on true
```

Specify the Precision When Using the Char Type or Use the String Data Type

cast(id as char) truncates only the first digit during data type conversion, causing incorrect data. If the converted field is the primary key field, a large amount of data will be lost.

You are not advised to use **table.exec.legacy-cast-behaviour=ENABLED** to handle the conversion error.

In versions earlier than Flink 1.15, you can set **table.exec.legacy-cast-behaviour** to **enabled** to enable type conversion. However, in Flink 1.15 and later versions, this flag is disabled by default. In particular, this will:

- Disable trimming/padding for casting to CHAR/VARCHAR/BINARY/VARBINARY
- CAST never fails but returns NULL, behaving as TRY_CAST but without inferring the correct type
- Formatting of some casting to CHAR/VARCHAR/STRING produces slightly different results.

We discourage the use of this flag and we strongly suggest for new projects to keep this flag disabled and use the new casting behavior. This flag will be removed in the next Flink versions.

- SQL statement before optimization:

```
select
cast(id as char) as id,
...
from t1
```

- SQL statement after optimization:

```
select
cast(id as string) as id,
...
from t1
```

Filter Out the Data To Be Withdrawn When Multiple Flink Jobs or INSERT INTO Statements Are Written into the Same Gauss for MySQL Database

When multiple Flink jobs write data to the same MySQL table, one job sends the withdrawal data (-D and -U) to delete the entire row and then inserts the updated data, causing other jobs to lose their changes.

- SQL statement before optimization:

```
create table source-A(
id,
user_id
)with(
'connector' = 'kafka'
);
create table source-B(
id,
org_id
)with(
'connector' = 'kafka'
);
create table sink-A(
id,
user_id
)with(
'connector' = 'jdbc'
'url' = 'jdbc:mysql://****',
'table-name' = 'sink-table'
);
create table sink-B(
id,
org_id
)with(
'connector' = 'jdbc'
'url' = 'jdbc:mysql://****',
'table-name' = 'sink-table'
);
insert into sink-A select id,user_id from source-A;
insert into sink-B select id,org_id  from source-B;
```

- SQL statement after optimization:

```
create table source-A(
id,
user_id
)with(
'connector' = 'kafka'
);
create table source-B(
id,
org_id
)with(
'connector' = 'kafka'
);
create table sink-A(
id,
user_id
)with(
'connector' = 'jdbc'
```



```
'url' = 'jdbc:mysql://****',
'table-name' = 'sink-table',
'filter.record.enabled' = 'true'
);
create table sink-B(
id,
org_id
)with(
'connector' = 'jdbc'
'url' = 'jdbc:mysql://****',
'table-name' = 'sink-table',
'filter.record.enabled' = 'true'
);
insert into sink-A select id,user_id from source-A;
insert into sink-B select id,org_id  from source-B;
```

3.6 Flink Performance Tuning

3.6.1 Performance Tuning Rules

Run Compaction on Hudi Tables to Prevent Long Checkpointing of the Hudi Source Operator

If the checkpointing of the Hudi Source operator takes a long time, check whether the compaction of the Hudi table is normal. If there was no compaction for a long time, the list performance deteriorates.

Set Table TTL to Reduce the Backend Data Volume When Joining a Fact Table and a Dimension Table

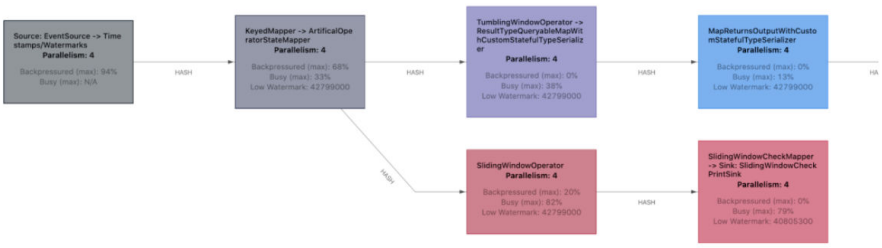
For details, see **Optimize State Backends Through Table-Level TTL**.

Set Proper Degree of Parallelism

The processing speed of tasks is related to parallelism. Generally, increasing parallelism can effectively improve read speed. However, if parallelism is too high, some node resources may be wasted, and if parallelism is too low, some nodes may run tasks slowly. A SQL statement cannot set parallelism for a specific task. You can set one for all.

Set source parallelism based on the upstream component. For a streaming system, the parallelism is recommended to be the same as the number of upstream partitions (for example, the number of Kafka topic partitions). For a batch system, the parallelism is recommended to be the same as the number of upstream slices (for example, the number of HDFS blocks).

The parallelism of Flink jobs using Source, Sink, and intermediate computing operators should be adjusted. If intermediate computing is busy according to the job flow diagram, you need to adjust the parallelism of the job to change the parallelism of the operators, for example, the join operator.



3.6.2 Performance Tuning Suggestions

Enable the Log Indexes for Hudi MOR Stream Tables for Faster Flink Streaming Reads on the MOR Table

To enable log indexes for better read and write performance of Hudi MOR tables, add **'hoodie.log.index.enabled'='true'** for the Sink and Source tables.

Adjust Operator Parallelism to Improve Performance

- You can set the parallelism parameters of read and write operators to improve Hudi read and write performance.

  The **read.tasks** parameter is for the parallelism of the read operator.

  The **write.tasks** parameter is for the parallelism of the write operator.
- When state indexes are used and the job is restarted (not checkpoint restart), the target table needs to be read to rebuild the indexes. You can increase the parallelism of the operator to improve the performance.

  The **write.index_bootstrap.tasks** parameter controls the parallelism for loading indexes.
- When state indexes are used to write data, check the uniqueness of the primary key and allocate a specific file to be written to improve operator parallelism for better performance.

  The **write.bucket_assign.tasks** parameter controls the task parallelism for bucket assign. The default value is the parallelism of the execution environment.

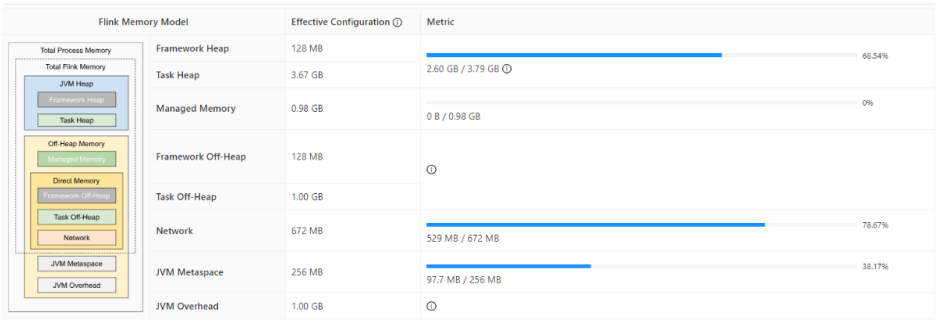
Optimize Resources to Improve the Performance of Stateless Computing

Flink computing operations are classified into the following types:

- Stateless computing: These operators (such as filter, union all, and lookup join) do not need to save computing states.
- Stateful computing: These operators (such as join, union, window, group by, and aggregation operators) compute based on data state changes.

For non-stateful computing, you can adjust Heap Size and NetWork of TaskManager to optimize performance.

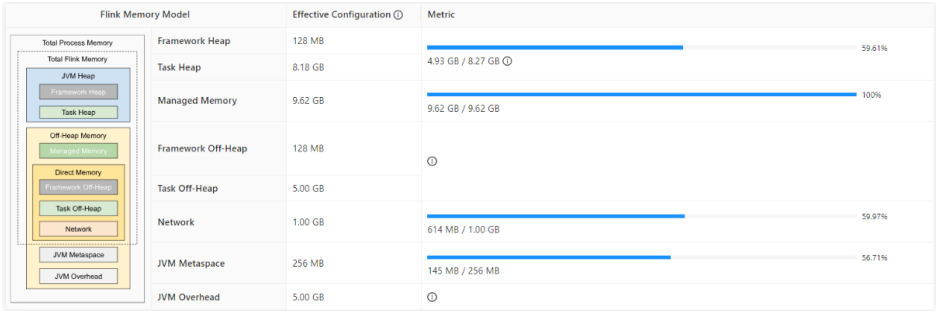
For example, if a job only reads and writes data, TaskManager does not need extra vCores. The default values of **off-Heap** and **Overhead** are 1 GB, and the memory is mainly allocated to heap and network.



Optimize Resources to Improve the Performance of Stateful Computing

The SQL logic contains many operations such as join and convolution calculation. Tune state backend performance, vCore, and Manage Memory.

For example, if a job joins over three tables and the performance requirement is high, add six extra vCores to a single TaskManager, increase the off-Heap and overhead to 5 GB, and set the Manage Memory used for Flink status management to 9.6 GB.



Optimize State Backends Through Table-Level TTL

This suggestion is available for MRS 3.3.0 or later.

When you join two Flink streams, there is a possibility that data in one table changes rapidly (short TTL) and data in the other table changes slowly (long TTL). Currently, Flink supports only table-level TTL. To ensure join accuracy, you need to set the table-level TTL to a long time. In this case, a large amount of expired data is stored in state backends, causing great workload pressure. To reduce the pressure, you can use Hints to set different expiration time for left and right tables. The WHERE clause is not supported.

For example, set the TTL of the left table (**state.ttl.left**) to 60 seconds and that of the right table (**state.ttl.right**) to 120 seconds.

- Use Hints in the following format:

```
table_path /*+ OPTIONS(key=val [, key=val]*) */

key:
    stringLiteral
val:
    stringLiteral
```

- The following is a configuration example with a SQL statement:

```
CREATE TABLE user_info (`user_id` VARCHAR, `user_name` VARCHAR) WITH (
  'connector' = 'kafka',
  'topic' = 'user_info_001',
```



```
  'properties.bootstrap.servers' = '192.168.64.138:21005',
  'properties.group.id' = 'testGroup',
  'scan.startup.mode' = 'latest-offset',
  'value.format' = 'csv'
);
CREATE table print(
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `score` INT
) WITH ('connector' = 'print');
CREATE TABLE user_score (user_id VARCHAR, score INT) WITH (
  'connector' = 'kafka',
  'topic' = 'user_score_001',
  'properties.bootstrap.servers' = '192.168.64.138:21005',
  'properties.group.id' = 'testGroup',
  'scan.startup.mode' = 'latest-offset',
  'value.format' = 'csv'
);
INSERT INTO
  print
SELECT
  t.user_id,
  t.user_name,
  d.score
FROM
  user_info as t
  LEFT JOIN
  -- Set different TTLs for left and right tables.
  /*+ OPTIONS('state.ttl.left'='60S', 'state.ttl.right'='120S') */
  user_score as d ON t.user_id = d.user_id;
```

Optimize the State Backend Through Table-level JTL

This suggestion is available for MRS 3.3.0 or later.

If backend data deletion upon one join is allowed in a Flink dual-stream inner join, this feature can be used.

This feature is available for inner joins of streams only.

You can use hints to set different join times for left and right tables.

- Use Hints in the following format:

```
table_path /*+ OPTIONS(key=val [, key=val]*) */

key:
    stringLiteral
val:
    stringLiteral
```

- The following is a configuration example with a SQL statement:

```
CREATE TABLE user_info (`user_id` VARCHAR, `user_name` VARCHAR) WITH (
  'connector' = 'kafka',
  'topic' = 'user_info_001',
  'properties.bootstrap.servers' = '192.168.64.138:21005',
  'properties.group.id' = 'testGroup',
  'scan.startup.mode' = 'latest-offset',
  'value.format' = 'csv'
);
CREATE table print(
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `score` INT
) WITH ('connector' = 'print');
CREATE TABLE user_score (user_id VARCHAR, score INT) WITH (
  'connector' = 'kafka',
  'topic' = 'user_score_001',
  'properties.bootstrap.servers' = '192.168.64.138:21005',
```



```
  'properties.group.id' = 'testGroup',
  'scan.startup.mode' = 'latest-offset',
  'value.format' = 'csv'
);
INSERT INTO
  print
SELECT
  t.user_id,
  t.user_name,
  d.score
FROM
  user_info as t
  JOIN
  -- Set different JTL join times for left and right tables.
  /*+ OPTIONS('eliminate-state.left.threshold'='1','eliminate-state.right.threshold'='1') */
  user_score as d ON t.user_id = d.user_id;
```

Number of TM Slots Should be a Multiple of the Number of TM CPUs

In Flink, each task is divided into subtasks. A subtask is an execution thread unit that runs on the TM. If Slot Sharing Group is disabled, a subtask is deployed in a slot. Even if Slot Sharing Group is enabled, the subtasks in a slot are load balanced in most cases. The number of slots on the TM indicates the number of running task threads.

The number of slots must be the same as the number of CPU cores. When hyper-threading is used, each slot occupies two or more hardware threads.

[Example] Set the number of TM slots to 2 to 4 times the number of CPU cores.

```
taskmanager.numberOfTaskSlots: 4
taskmanager.cpu.cores: 2
```

Adjust Network Memory When Shuffle Is Enabled, Data Volume Is Large, and Concurrency Is High

When there are a large number of concurrent requests and a large amount of data, there are massive amounts of network I/Os after shuffle. Increasing the network cache memory can increase the amount of data read at a time, thereby improving the I/O speed.

[Example]

```
#Ratio of the network memory usage to the process memory usage
taskmanager.memory.network.fraction: 0.6
# Minimum size of the network cache memory
taskmanager.memory.network.min: 1g
#Maximum size of the network cache memory. (In MRS 3.3.1 and later versions, you do not need to change the value. The default value is Long#MAX_VALUE.)
taskmanager.memory.network.max: 20g
```

Use Simple Data Types Such as POJO and Avro Based on Serialization Performance

When using APIs to code Flink programs, you should consider the serialization of Java objects. In most cases, Flink can efficiently process serialization. SQL data is ROW data. SQL uses the built-in efficient serializer of Flink.

Table 3-7 Serialization

Serializer	Opts/s
PojoSeriallizer	813
Kryo	294
Avro(Reflect API)	114
Avro(SpecificRecord API)	632

Network Communication Optimization

Flink communication mainly depends on the Netty network. Netty settings are especially important for Flink application execution. The network determines the data exchange speed and task execution efficiency.

[Example]

```
# Number of threads on the netty server. The value -1 indicates the default parameter numOfSlot.
taskmanager.network.netty.server.numThreads -1(numOfSlot)
# Number of netty client threads (The value -1 indicates the default parameter numofSlot).
taskmanager.network.netty.client.numThreads : -1
# Timeout interval for connecting to the netty client.
taskmanager.network.netty.client.connectTimeoutSec: 120s
# Size of the sending and receiving buffers of netty (0 indicates the default parameter of netty, 4 MB)
taskmanager.network.netty.sendReceiveBufferSize: 0
# Netty transmission mode. The default option selects the mode based on the platform.
taskmanager.network.netty.transport: auto
```

Overall Memory Optimization

Flink has the heap memory and off-heap memory. The Java heap memory is specified when the Java program is created, which is also part of the memory where the JVM automatically triggers GC. Off-heap memory can be classified into managed memory and memory cannot be managed by the JVM. Managed Memory and Direct Memory that can be managed by the JVM are the focus of optimization. JVM Metaspace and JVM Overhead that cannot be managed by the JVM are native memory.

Figure 3-8 Memory

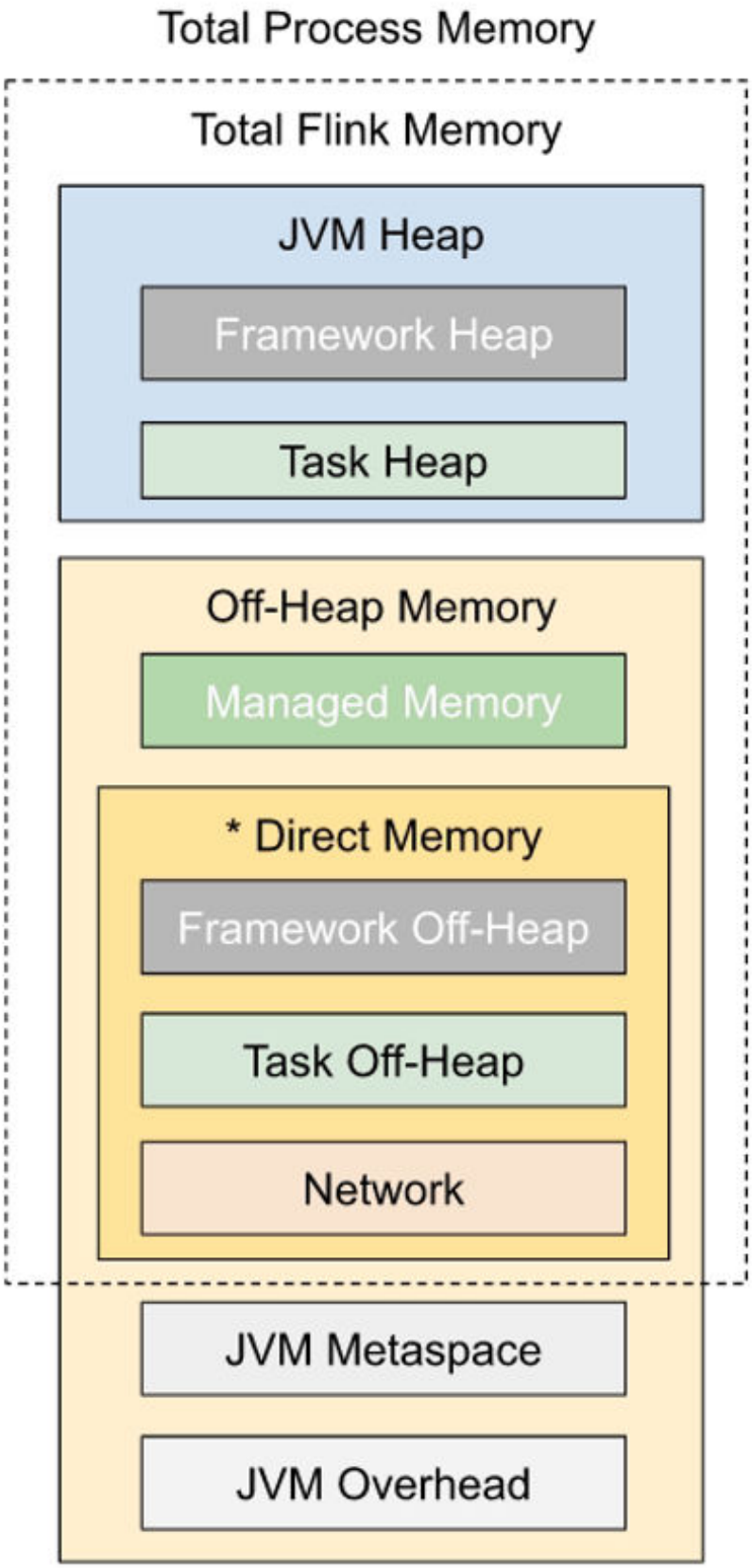


Table 3-8 Related parameters

Parameter	Configuration	Description	Remarks
Total Memory	taskmanager.memory.flink.size: none	Total memory size managed by Flink. There is no default value. Metaspace and Overhead are not included. Set this parameter in standalone mode.	Overall memory
	taskmanager.memory.process.size: none	Memory size used by the entire Flink process. Set this parameter when containers are used.	
FrameWork	taskmanager.memory.framework.heap.size: 128mb	Size of the heap memory occupied by runtime. Generally, you do not need to change the value. The occupied space is relatively fixed.	Memory occupied by RUNTIME. Generally, you do not need to change the value greatly.
	taskmanager.memory.framework.off-heap.size: 128mb	Size of the off-heap memory occupied by runtime. Generally, you do not need to change the value. The occupied space is relatively fixed.	
Task	taskmanager.memory.task.heap.size: none	There is no default value. The value is obtained by subtracting the memory for framework, hosting, and network from **flink.size**.	Operator logic in regular objects of user code (such as UDFs), which occupies memory
	taskmanager.memory.task.off-heap.size: 0	The default value is **0**, indicating the off-heap memory used by **task**	

Parameter	Configuration	Description	Remarks
Managed Memory	taskmanager.memory.managed.fraction: 0.4	Ratio of managed memory to **taskmanager.memory.flink.size**. The default value is **0.4**.	The managed memory used for intermediate result caching, sorting, hashing (batch calculation) and by RocksDB state backends (stream computing). For batch processing, a fixed size of memory is applied for at the beginning. For stream processing, the memory is applied on demand.
	taskmanager.memory.managed.size: 0	Size of managed memory. Generally, this parameter is not specified. The default value is **0**. The size is calculated based on **taskmanager.memory.managed.fraction**. If this parameter is specified, the memory ratio will be overwritten.	
Network	taskmanager.memory.network.min: 64mb	Minimum network memory.	Network memory for shuffle and broadcast between TaskManagers, and for network buffer.
	taskmanager.memory.network.max: 1gb	Maximum size of the network cache. (For MRS 3.3.1 and later versions, you do not need to change the value. The default value is **Long#MAX_VALUE**.)	
	taskmanager.memory.network.fraction: 0.1	Fraction of **taskmanager.memory.flink.size** used as the network memory. The default value is **0.1**, which is limited to the value between **network.min** and **network.max**.	Network memory for shuffle and broadcast between TaskManagers, and for network buffer.
Others	taskmanager.memory.jvm-metaspace.size: 256M	Maximum size of the metaspace. The default value is 256 MB.	Memory managed by users
	taskmanager.memory.jvm-overhead.min: 192M	Minimum extra overhead of the JVM. The default value is 192 MB.	

Parameter	Configuration	Description	Remarks
	taskmanager.memory.jvm-overhead.max: 1G	Maximum extra overhead of JVM. The default value is 1 GB.	
	taskmanager.memory.jvm-overhead.fraction: 0.1	Ratio of the extra JVM overhead to **taskmanager.memory.process.size**. The default value is **0.1**. The calculated extra JVM overhead is limited between **jvm-overhead.min** and **jvm-overhead.max**.	

NOTE

In MRS 3.3.1 and later versions, you do not need to change the value of **taskmanager.memory.network.max**.

Reduce Shuffled Data As Much As Possible If Broadcast Join Cannot Be Used

If broadcast join is not supported, shuffling will occur. You can use various methods, such as predicate pushdown and runtime filter, to reduce the amount of shuffled data.

[Example]

```
# Configure runtime filter
table.exec.runtime-filter.enabled: true
# Pushdown
table.optimizer.source.predicate-pushdown-enabled: true
```

Use a Local-Global Optimization Policy When Data Skew Occurs

[Example]

```
# Enable mini-batch optimization.
table.exec.mini-batch.enabled: true
#Maximum waiting time
table.exec.mini-batch.allow-latency: 20ms
#Maximum number of cached records
table.exec.mini-batch.size: 8000
# Enable two-phase aggregation.
table.optimizer.agg-phase-strategy: TWO_PHASE
```

Use MiniBatch Aggregation to Increase Throughput

The core idea of MiniBatch aggregation is caching a group of input data in the buffer of the aggregation operator. When the input data is triggered for processing, each key can access states with only one operation, which greatly reduces state overhead and achieves better throughput. However, latency may

increase because it buffers some records instead of processing them immediately, which is a trade-off between throughput and latency. This function is disabled by default.

- Configure with APIs:

```
// Instantiate table environmentTableEnvironment tEnv = ...
// Access flink configuration.
Configuration configuration = tEnv.getConfig().getConfiguration();
// set low-level key-value options
configuration.setString("table.exec.mini-batch.enabled", "true"); // enable mini-batch optimizationconfiguration.setString("table.exec.mini-batch.allow-latency", "5 s"); // use 5 seconds to buffer input recordsconfiguration.setString("table.exec.mini-batch.size", "5000"); // the maximum number of records can be buffered by each aggregate operator task
```

- Configure in the resource file (**flink-conf.yaml**):

```
table.exec.mini-batch.enabled: true
table.exec.mini-batch.allow-latency : 5 s
table.exec.mini-batch.size: 5000
```

Use Local-Global Two-Phase Aggregation to Reduce Data Skew

Local-Global aggregation is proposed to solve the data skew problem. A group of aggregations is divided into two phases: local aggregation in the upstream and global aggregation in the downstream, which is similar to the Combine + Reduce in MapReduce.

Records in a data stream may skew. Instances of some aggregation operators must process more records than others, which can cause hotspotting. Local aggregation can accumulate a certain amount of input data with the same key to a single accumulator. Global aggregation receives only the reduced accumulator instead of a large amount of original input data, which greatly reduces network shuffle and state access. The amount of input data accumulated in each local aggregation is based on the mini-batch interval, which means that local-global aggregation depends on mini-batch optimization.

- Configure with APIs:

```
// Instantiate table environmentTableEnvironment tEnv = ...
// access flink configuration
Configuration configuration = tEnv.getConfig().getConfiguration();// set low-level key-value options
configuration.setString("table.exec.mini-batch.enabled", "true"); // local-global aggregation depends on mini-batch is enabled
configuration.setString("table.exec.mini-batch.allow-latency", "5 s");
configuration.setString("table.exec.mini-batch.size", "5000");
configuration.setString("table.optimizer.agg-phase-strategy", "TWO_PHASE"); // enable two-phase, i.e. local-global aggregation
```

- Configure in the resource file:

```
table.exec.mini-batch.enabled: true
table.exec.mini-batch.allow-latency : 5 s
table.exec.mini-batch.size: 5000
table.optimizer.agg-phase-strategy: TWO_PHASE
```

Use Multiple Disks to Improve I/O Performance When RocksDB Is the State Backend

RocksDB uses memory and disks to store data. When state is large, disk space usage is high. If there are frequent read requests to RocksDB, the disk I/O will limit speed of Flink tasks. When a TaskManager contains three slots, disks of a single server are frequently read and written. Concurrent operations contend for the I/O of the same disk, and the throughput of the three slots decreases. You can specify multiple disks to reduce I/O competition.

[Example] Configure checkpoint directories of RockDB on different disks (**flink-conf.yaml**).

```
state.backend.rocksdb.localdir:/data1/flink/rocksdb,/data2/flink/rocksdb
```

Replace the ValueState Storage Containers with MapState or ListState when RocksDB is the Status Backend

RocksDB is an embedded KV database. Data is stored in key-value pairs. For map data, if ValueState is used, the data is stored as a record in RocksDB, and the value is the entire map. If MapState is used, the data is stored in multiple records in RocksDB. This allows only a small part of data be serialized during query or modification. When the map is stored as a whole, adding, deleting, or modifying the map causes a large number of serialization operations. For List data, ListState can be used to dynamically add elements without serialization.

In addition, the state in Flink supports TTL. TTL encapsulates the timestamp and **userValue**. The TTL of ValueState is based on the entire key. The TTL of MapState<UK, UV> is based on the UK. It has a smaller granularity and supports more TTL semantics.

Configure Compaction to Reduce the Checkpoint Size

In I/O-intensive applications, you can enable checkpoint compaction to improve I/O performance at the cost of a little CPU performance.

[Example] Enable compaction in checkpoint configuration (**flink-conf.yaml**).

```
execution.checkpointing.snapshot-compression: true
```

Recover Large-State Checkpoint from Local States

To quickly recover, each task writes checkpoint data to the local disk and distributed remote storage at the same time. Each data record has two replications. When an application needs to recover, the system checks if the local checkpoint data is okay. If it is, the system uses it first. This makes it faster to get the state data without having to get it from a remote location.

[Example] Configure checkpoints to be preferentially restored from the local host (**flink-conf.yaml**):

```
state.backend.local-recovery: true
```

3.7 Development Examples

Flink can interconnect with multiple services, such as ClickHouse, HBase, and HDFS. The following table lists the supported versions and examples.

- **Interconnecting FlinkServer to ClickHouse**
- **Interconnecting FlinkServer with HBase**
- **Interconnecting FlinkServer with HDFS**
- **Interconnecting FlinkServer with Hive**
- **Interconnecting FlinkServer with Hudi**

- **Interconnecting FlinkServer with Kafka**

4 HBase

4.1 HBase Application Development Rules

Create a Configuration instance

Call the create() method of HBaseConfiguration to instantiate this class. Otherwise, the HBase configurations cannot be successfully loaded.

Correct:

```
//This part is declared in the class member variable declaration.
private Configuration hbaseConfig = null;
//Instantiate this class using its constructor function or initialization method.
hbaseConfig = HBaseConfiguration.create();
```

Incorrect:

```
hbaseConfig = new Configuration();
```

Share the Configuration instance

The HBase client codes obtain rights to interact with an HBase cluster by creating an HConnection with Zookeeper. Each HConnection has a Configuration instance. The created HConnection instances are cached. That is, if the HBase client needs to communicate with an HBase cluster, a Configuration instance is transferred to the cache. Then, the HBase client checks for an HConnection instance for the Configuration instance in the cache. If a match is found, the HConnection instance is returned. If no match is found, an HConnection instance will be created.

If the Configuration instance is frequently created, a lot of unnecessary HConnection instances will be created, causing the number of connections to Zookeeper to reach the upper limit.

Therefore, it is recommended that the client codes share the same **Configuration** instance.

Create an Table instance

```
public abstract class TableOperationImpl {
  private static Configuration conf = null;
```



```
  private static Connection connection = null;
  private static Table table = null;
  private static TableName tableName = TableName.valueOf("sample_table");

  public TableOperationImpl() {
    init();
  }
  public void init() {
    conf = ConfigurationSample.getConfiguration();
    try {
      connection = ConnectionFactory.createConnection(conf);
      table = conn.getTable(tableName);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  public void close() {
    if (table != null) {
      try {
        table.close();
      } catch (IOException e) {
        System.out.println("Can not close table.");
      } finally {
        table = null;
      }
    }
    if (connection != null) {
      try {
        connection.close();
      } catch (IOException e) {
        System.out.println("Can not close connection.");
      } finally {
        connection = null;
      }
    }
  }
  public void operate() {
    init();
    process();
    close();
  }
}
```

An Table instance cannot be used by multiple threads at the same time

Table is not thread safe for reads or write. If an Table instance is used by multiple threads at the same time, exceptions will occur.

Cache a frequently used Table instance

Cache the Table instance that will be frequently used by a thread for a long period of time. A cached instance, however, will not be necessarily used by a thread permanently. In special circumstances, you need to rebuild an Table instance. See the next rule for details.

Correct:

NOTE

In this example, the Table instance is cached by Map. This method applies when multiple threads and Table instances are required. If an Table instance is used by only one thread and the thread has only one Table instance, Map need not be used.

```
//In this Map, TableName is the Key value. Cache all Table instances.
private Map<String, Table> demoTables = new HashMap<String, Table>();
//All Table instances share this Configuration instance.
```



```
private Configuration demoConf = null;
/**
* <Initialize an HTable class>
* <Detailed function description>
* @param tableName
* @return
* @throws IOException
* @see [class, class#method, class#member]
*/
private Table initNewTable(String tableName) throws IOException
{
try (Connection conn = ConnectionFactory.createConnection(demoConf)){
      return conn.getTable(tableName);
    }
}
/**
* <Obtain Table instances>
* <Detailed function description>
* @see [class, class#method, class#member]
*/
private Table getTable(String tableName)
{
if (demoTables.containsKey(tableName))
{
return demoTables.get(tableName);
} else {
Table table = null;
try
{
table = initNewTable(tableName);
demoTables.put(tableName, table);
}
catch (IOException e)
{
// TODO Auto-generated catch block
e.printStackTrace();
}
return table;
}
}
/**
* <Write data>
* <Multi-thread multi-Table instance design optimization is not involved. The synchronization method is
used
* because the Table is not thread safe. It is recommended that an Table instance be used by only one data
write thread at the same
*time.>
* @param dataList
* @param tableName
* @see [class, class#method, class#member]
*/
public void putData(List<Put> dataList, String tableName)
{Table table = getTable(tableName);
//Synchronization is not required if the Table instance is not shared by multiple threads.
//Note that Table is not thread safe.
synchronized (table)
{
try
{
table.put(dataList);
table.notifyAll();
}
catch (IOException e)
{
                // When IOE is detected, the cached instance needs to be re-created.
try {
    // Close the Connection.
      table.close();
                // Re-create the instance.
```

```
                table = initNewTable(tableName);
} catch (IOException e1) {
// TODO
}
}
}
}
```

Incorrect:

```
public void putDataIncorrect(List<Put> dataList, String tableName)
{Table table = null;
try
{
//Create an HTable instance each time when data is written.
table = initNewTable(tableName);
table.put(dataList);
}
catch (IOException e1)
{
// TODO Auto-generated catch block
e1.printStackTrace();
}
finally
{
table.close();
}
}
```

Rebuild an Table instance

Rebuilt a cached Table when IOException is detected. See the example of the previous rule.

Do not call the following methods unless necessary:

- **Configuration#clear**

  Do not call this method if a Configuration is used by an object or a thread. The Configuration#clear method clears all attributes loaded. If this method is called for a Configuration used by Table, all the parameters of this Configuration will be deleted from Table. As a result, an exception occurs when Table uses the Configuration the next time.

  Therefore, avoid calling this method each time you rebuild an Table instance. Call this method when all the threads need to quit.

- **HConnectionManager#deleteAllConnections**

  This method deletes all connections from the **Connection set**. As the Table stores the links to the connections, the connections being used cannot be stopped after the HConnectionManager#deleteAllConnections method is called, which eventually causes information leakage.

Handle the data failed to write

Some data write operations may fail due to instant exceptions or process failures. Therefore, the data must be recorded so that it can be written to the HBase when the cluster is restored.

The failed data returned by the HBase client will not be automatically rewritten. The interface caller is only informed of the data failed to be written. To prevent data loss, measures must be taken to temporarily save the data in a file or in memory.

Correct:

```
private List<Row> errorList = new ArrayList<Row>();
/**
* <Insert data in PutList mode. >
* <Synchronization is not required if the method is not called by multiple threads.>
* @param put a data record
* @throws IOException
* @see [class, class#method, class#member]
*/
public synchronized void putData(Put put)
{
// Temporarily cache data in this List.
dataList.add(put);
// Perform a Put operation when the dataList size reaches PUT_LIST_SIZE.
if (dataList.size() >= PUT_LIST_SIZE)
{
try
{
demoTable.put(dataList);
}
catch (IOException e)
{
// If RetriesExhaustedWithDetailsException occurs,
// certain data failed to be written, which
// is caused by process errors in the HBase cluster or migration of a large number of
// Regions.
if (e instanceof RetriesExhaustedWithDetailsException)
{
RetriesExhaustedWithDetailsException ree =
  (RetriesExhaustedWithDetailsException)e;
int failures = ree.getNumExceptions();
for (int i = 0; i < failures; i++)
{
errorList.add(ree.getRow(i));
}
}
}
dataList.clear();
}
}
```

Release resources

Call the Close method to release resources when the ResultScanner and Table instances are not required. To enable the Close method to be called, add the Close method to the **finally** block.

Correct:

```
ResultScanner scanner = null;
try
{
scanner = demoTable.getScanner(s);
//Do Something here.
}
finally
{
scanner.close();
}
```

Incorrect:

1. The code does not call the scanner.close() method to release resources.
2. The scanner.close() method is not placed in the **finally** block.

   ```
   ResultScanner scanner = null;
   scanner = demoTable.getScanner(s);
   ```



```
//Do Something here.
scanner.close();
```

Add fault-tolerance mechanism for Scan

Exceptions, such as lease expiration, may occur when Scan is performed. Retry operations need to be performed when exceptions occur.

Retry operations can be applied in HBase-related interface methods to improve fault tolerance capabilities.

Stop Admin as soon as it is not required

Stop Admin as soon as possible. Do not cache the same Admin instance for an extended period of time.

4.2 HBase Application Development Suggestions

Do not call the closeRegion method of Admin to close a Region

Admin interface provides an API to close a Region:

public void closeRegion(final String regionname, final String serverName)

When this method is used to close a Region, the HBase Client sends an RPC request to the RegionServer of the Region to be closed. The Master is unaware of the whole process. That is, the Master does not know even if the Region is closed. If the closeRegion method is called when the Master determines to migrate the Region based on the execution result of Balance, the Region cannot be closed or migrated. (In the current HBase version, this issue has not been resolved).

Therefore, do not call the closeRegion method of Admin to close a Region.

Write data in PutList mode

Table provides two data write interfaces:

- public void put(final Put put) throws IOException
- public void put(final List<Put> puts) throws IOException

The second one is recommended because it provides better performance than the first one.

Specify StartKey and EndKey for a Scan

A Scan with a specific range offers higher performance than a Scan without specific range.

Example:

```
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("familyname"),Bytes.toBytes("columnname"));
scan.setStartRow( Bytes.toBytes("rowA")); // StartKey is rowA.
scan.setStopRow( Bytes.toBytes("rowB")); // EndKey is rowB.
for(Result result : demoTable.getScanner(scan)) {
// process Result instance
}
```


Do not disable WAL

Write-Ahead-Log (WAL) allows data to be written in a log file before being stored in the database.

WAL is enabled by default. The Put class provides an interface to disable WAL:

public void setWriteToWAL(boolean write)

If WAL is disabled (writeToWAL is set to False), data of the last 1s (The time can be specified by the **hbase.regionserver.optionallogflushinterval** parameter on the RegionServer. It is 1s by default) will be lost. WAL can be disabled only when high data write speed is required and data loss of the last 1s is allowed.

Set blockcache to true when creating a table or when Scan is performed

Set blockcache to true when a table is created or when Scan is performed on the HBase client. If there are a large number of repeated records, setting this parameter to true can improve efficiency.

By default, blockcache is true. Avoid setting this parameter to false forcibly, for example:

```
HColumnDescriptor fieldADesc = new HColumnDescriptor("value".getBytes());
fieldADesc.setBlockCacheEnabled(false);
```

The HBase does not support query by Orderby or with the search criteria specified. It is based on the lexicographic order and can only be read by Rowkey.

HBase should not be used in scenarios of random query and sequencing.

Suggestions on Services List Design

1. Pre-allocate regions in a balanced manner in order to improve concurrency capabilities.
2. Avoid excessive hotspot regions. Import the time factor to Rowkey if necessary.
3. It is preferred that concurrently accessed data be stored continuously. Concurrently read data should be stored nearby, on the same row and in the same cell.
4. Put frequently queried attributes property before Rowkey. Rowkey should be designed to match the main query criteria in terms of criterion sequencing.
5. Attributes with high dispersions should be contained in RowKey. Design the services list based on data dispersion and query scenarios.
6. Store redundant information to enhance indexing performance. Use secondary index to adapt to more query scenarios.
7. Enable automatic deletion of expired data by setting the expiration time and version quantity.

NOTE

In the HBase, Regions busy writing data are called hotspot Region.

5 HDFS

5.1 HDFS Application Development Rules

Set the HDFS NameNode metadata storage path

NameNode metadata is stored in **${BIGDATA_DATA_HOME}/namenode/data** by default. This parameter sets the storage path of HDFS metadata.

Enable NameNode image backup for the HDFS

fs.namenode.image.backup.enable specifies whether to enable the NameNode image backup function. You need to set this parameter to **true**. Then the system can periodically back up the NameNode data.

Set the HDFS DataNode data storage path

DataNode data is stored in **${BIGDATA_DATA_HOME}/hadoop/dataN/dn/datadir** by default. *N* indicates the number of directories is greater than or equal to1.

For example, **${BIGDATA_DATA_HOME}/hadoop/data1/dn/datadir, ${BIGDATA_DATA_HOME}/hadoop/data2/dn/datadir**.

After the storage path is set, data is stored in the corresponding directory of each mounted disk on a node.

Improve HDFS read/write performance

The data write process is as follows:

After receiving service data and obtaining the data block number and location from the NameNode, the HDFS client contacts DataNodes and establishes a pipeline with the DataNodes to be written. Then, the HDFS client writes data to DataNode1 using a proprietary protocol, and DataNode1 writes data to DataNode2 and DataNode3 (three duplicates). After data is written, a message is returned to the HDFS client.

1. Set a proper block size. For example, set **dfs.blocksize** to **268435456** (256 MB).
2. It is not necessary to cache the big data that is not reused. In this case, set the following parameters to **false**:

   dfs.datanode.drop.cache.behind.reads and **dfs.datanode.drop.cache.behind.writes**

Set the MapReduce intermediate file storage path

Only one default path is provided for storing MapReduce intermediate files, that is, **${hadoop.tmp.dir}/mapred/local**.It is recommended that intermediate files be stored on each disk.

For example, **/hadoop/hdfs/data1/mapred/local, /hadoop/hdfs/data2/mapred/local, /hadoop/hdfs/data3/mapred/local**. Directories that do not exist are automatically ignored.

Release applied resources in finally during Java development.

Applied HDFS resources are released in try/finally and cannot be released outside the try statement only. Otherwise, resource leakage occurs.

HDFS file operation APIs

Almost all Hadoop file operation classes are in the **org.apache.hadoop.fs** package. These APIs support operations such as opening, reading, writing, and deleting a file. FileSystem is the interface class provided for users in the Hadoop class library. FileSystem is an abstract class. Concrete classes can be obtained only using the get method. The get method has multiple overload versions, and the following get method is often used.

```
static FileSystem get(Configuration conf);
```

This class encapsulates almost all file operations, such as mkdir and delete. The program library framework for file operations is as follows:

```
operator()
{
    Obtain the Configuration object.
    Obtain the FileSystem object.
    Perform file operations.
 }
```

HDFS initialization method

HDFS initialization is a prerequisite for using APIs provided by HDFS.

To initialize HDFS, load the HDFS service configuration file, implement Kerberos security authentication, and instantiate FileSystem. Obtain keytab files for Kerberos security authentication in advance.

```
Example:
private  void init() throws IOException {
 Configuration conf = new Configuration();
 // Read a configuration file.
  conf.addResource("user-hdfs.xml");
 // Implement security authentication in security mode.
  if ("kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))) {
```



```
String PRINCIPAL = "username.client.kerberos.principal";
String KEYTAB = "username.client.keytab.file";
// Set the keytab key file.
 conf.set(KEYTAB, System.getProperty("user.dir") + File.separator + "conf" + File.separator +
conf.get(KEYTAB));
 // Set the Kerberos configuration file path. */
String krbfilepath = System.getProperty("user.dir") + File.separator + "conf" + File.separator + "krb5.conf";
System.setProperty("java.security.krb5.conf", krbfilepath);
// Implement login authentication. */
SecurityUtil.login(conf, KEYTAB, PRINCIPAL);
}
 // Instantiate FileSystem.
fSystem = FileSystem.get(conf);
}
```

Upload local files to the HDFS

FileSystem.copyFromLocalFile (Path src, Patch dst) is used to upload local files to a specified directory in the HDFS. *src* and *dst* indicate complete file paths.

Example:

```
public class CopyFile {
      public static void main(String[] args) throws Exception {
      Configuration conf=new Configuration();
      FileSystem hdfs=FileSystem.get(conf);
           //Local file
              Path src =new Path("D:\\HebutWinOS");
               //To the HDFS
      Path dst =new Path("/");
            hdfs.copyFromLocalFile(src, dst);
      System.out.println("Upload to"+conf.get("fs.default.name"));
           FileStatus files[]=hdfs.listStatus(dst);
           for(FileStatus file:files){
              System.out.println(file.getPath());
           }
      }
}
```

Create files on the HDFS

FileSystem.mkdirs (Path f) is used to create folders on HDFS. *f* indicates a complete folder path.

Example:

```
public class CreateDir {
   public static void main(String[] args) throws Exception{
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
             Path dfs=new Path("/TestDir");
             hdfs.mkdirs(dfs);
    }
 }
```

Query the modification time of an HDFS file

FileSystem.getModificationTime() is used to query the modification time of a specified HDFS file.

Example:

```
   public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
```



```
        Path fpath =new Path("/user/hadoop/test/file1.txt");
        FileStatus fileStatus=hdfs.getFileStatus(fpath);
        long modiTime=fileStatus.getModificationTime();
        System.out.println("file1.txt modification time is"+modiTime);
    }
```

Read all files in an HDFS directory

FileStatus.getPath() is used to query all files in an HDFS directory.

Example:

```
public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path listf =new Path("/user/hadoop/test");

        FileStatus stats[]=hdfs.listStatus(listf);
        for(int i = 0; i < stats.length; ++i) {
          System.out.println(stats[i].getPath().toString());
        }
        hdfs.close();
 }
```

Query the location of a specified file in an HDFS cluster

FileSystem.getFileBlockLocation (FileStatus file, long start, long len) is used to query the location of a specified file in an HDFS cluster. *file* indicates a complete file path, and *start* and *len* specify the file path.

Example:

```
public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        Path fpath=new Path("/user/hadoop/cygwin");

        FileStatus filestatus = hdfs.getFileStatus(fpath);
        BlockLocation[] blkLocations = hdfs.getFileBlockLocations(filestatus, 0, filestatus.getLen());

        int blockLen = blkLocations.length;
        for(int i=0;i < blockLen;i++){
            String[] hosts = blkLocations[i].getHosts();
            System.out.println("block_"+i+"_location:"+hosts[0]);
        }
    }
```

Obtain all node names in an HDFS cluster

DatanodeInfo.getHostName() is used to obtain all node names in an HDFS cluster.

Example:

```
public static void main(String[] args) throws Exception {
        Configuration conf=new Configuration();
        FileSystem fs=FileSystem.get(conf);

        DistributedFileSystem hdfs = (DistributedFileSystem)fs;

        DatanodeInfo[] dataNodeStats = hdfs.getDataNodeStats();

        for(int i=0;i < dataNodeStats.length;i++){
            System.out.println("DataNode_"+i+"_Name:"+dataNodeStats[i].getHostName());
        }
    }
```


Multithread security login mode

If multiple threads are performing login operations, the relogin mode must be used for the subsequent logins of all threads after the first successful login of an application.

login example code:

```
private Boolean login(Configuration conf){
  boolean flag = false;
  UserGroupInformation.setConfiguration(conf);
  try {
       UserGroupInformation.loginUserFromKeytab(conf.get(PRINCIPAL), conf.get(KEYTAB));
       System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
       flag = true;
      } catch (IOException e) {
        e.printStackTrace();
      }
   return flag;
}
```

relogin example code:

```
public Boolean relogin(){
      boolean flag = false;
      try {
           UserGroupInformation.getLoginUser().reloginFromKeytab();
           System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
           flag = true;
          } catch (IOException e) {
            e.printStackTrace();
          }
      return flag;
  }
```

NOTICE

Repetitive logins will cause a newly created session to overwrite the previous session. As a result, the previous session cannot be maintained or monitored, and some functions are unavailable after the previous session expires.

5.2 HDFS Application Development Suggestions

Notes for reading and writing HDFS files

The HDFS does not support random read/write.

Data can be appended only to the end of an HDFS file.

Only data stored in the HDFS supports append. **edit.log** and metadata files do not support append. When using the append function, set **dfs.support.append** in *hdfs-site.xml* to **true**.

NOTE

- **dfs.support.append** is disabled by default in open-source versions but enabled by default in FusionInsight versions.
- This parameter is a server parameter. You are advised to enable this parameter to use the append function.
- Store data in other modes, such as HBase, if the HDFS is not applicable.

The HDFS is not suitable for storing a large number of small files

The HDFS is not suitable for storing a large number of small files because the metadata of small files will consume excessive memory resources of the NameNode.

Back up HDFS data in three duplicates

Three duplicates are enough for DataNode data backup. System data security is improved when more duplicates are generated but system efficiency is reduced. When a node is faulty, data on the node is balanced to other nodes.

Periodical HDFS Image Back-up

The system can back up the data on NameNode periodically after the image back-up parameter **fs.namenode.image.backup.enable** is set to **true**.

Provide operations to ensure data reliability

When you invoke the write function to write data, HDFS client does not write the data to HDFS but caches it in the client memory. If the client is abnormal, power-off, the data will be lost. For high-reliability demanding data, invoke hflush to refresh the data to HDFS after writing finishes.

6 Hive

6.1 Hive Application Development Rules

Load the Hive JDBC driver

The client software connects to HiveServer using Java database connectivity (JDBC). Therefore, you must load the JDBC driver class org.apache.hive.jdbc.HiveDriver for Hive.

Use the current class loader to load the driver class.

If there is no jar package in classpath, the client software throws "Class Not Found" and exits.

Example:

```
Class.forName("org.apache.hive.jdbc.HiveDriver").newInstance();
```

Set up a database connection

The driver management class java.sql.DriverManager of JDK is used to obtain a connection to the Hive database.

The Hive database URL is url="jdbc:hive2://xxx10.64xxx.22xxx.231xxx:2181,10xxx.64xxx.22xxx.232xxx:2181,10xxx.64xxx.22xxx.233xxx:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2;sasl.qop=auth-conf;auth=KERBEROS;principal=hive/hadoop.hadoop.com@HADOOP.COM;user.principal=hive/hadoop.hadoop.com;user.keytab=conf/hive.keytab";

In this example, ZooKeeper is deployed on three nodes and the default port is 2181. **xxx.xxx.xxx.xxx** indicates each of the IP addresses of the three nodes.The user name and password are null or empty because authentication has been performed successfully.

Example:

```
// Set up a connection.
 connection = DriverManager.getConnection(url, "", "");
```


Execute HQL

Note that the HQL statement cannot end with a semicolon (;).

Correct:

```
String sql = "SELECT COUNT(*) FROM employees_info";
 Connection connection = DriverManager.getConnection(url, "", "");
 PreparedStatement statement = connection.prepareStatement(sql);
 resultSet = statement.executeQuery();
```

Incorrect:

```
String sql = "SELECT COUNT(*) FROM employees_info;";
 Connection connection = DriverManager.getConnection(url, "", "");
 PreparedStatement statement = connection.prepareStatement(sql);
 resultSet = statement.executeQuery();
```

Close a database connection

After the client executes the HQL, close the database connection to prevent memory leakage.

Close the statement and connection objects of the JDK.

Example:

```
finally {
            if (null != statement) {
                statement.close();
            }

            // Close the JDBC connection.
            if (null != connection) {
                connection.close();
            }
        }
```

HQL syntax used to check for null values

Use **is null** to check whether a field is empty, that is, the field has no value. Use **is not null** to check whether a field is not mull, that is, the field has a value.

If you use **is null** for a character whose type is String and length is 0, False is returned. Use **col = ''** to check for null values, and use **col != ''** to check for non-null values.

Correct:

```
select * from default.tbl_src where id is null;
 select * from default.tbl_src where id is not null;
 select * from default.tbl_src where name = '';
 select * from default.tbl_src where name != '';
```

Incorrect:

```
select * from default.tbl_src where id = null;
 select * from default.tbl_src where id != null;
 select * from default.tbl_src where name is null;
 select * from default.tbl_src where name is not null;
```

Note that the type of the id field in the tbl_src table is Int, and the type of the name field is String.

The client configuration parameters must be consistent with the server configuration parameters

If the configuration parameters of the Hive, YARN, and HDFS servers of the cluster are modified, the related parameter in a client program will be modified. You need to check whether the configuration parameters submitted to the HiveServer before the configuration parameters are modified are consistent with those on the servers. If the configuration parameters are inconsistent, modify them on the client and submit them to the HiverServer. In the following example, if the parameter of YARN in the cluster is modified, the parameter submitted to the HiverServer from the Hive client and sample program before the modification must be reviewed and modified.

Initial state:

The parameter configuration of YARN in the cluster is as follows:

```
mapreduce.reduce.java.opts=-Xmx2048M
```

The parameter configuration on the client is as follows:

```
mapreduce.reduce.java.opts=-Xmx2048M
```

The parameter configuration of YARN in the cluster after the modification is as follows:

```
mapreduce.reduce.java.opts=-Xmx1024M
```

If the parameter in the client program is not changed, the parameter is still valid. This will result in insufficient memory for reducer and lead to MR running failure.

Multithread security login mode

If multiple threads are performing login operations, the relogin mode must be used for the subsequent logins of all threads after the first successful login of an application.

login example code:

```
  private Boolean login(Configuration conf){
    boolean flag = false;
    UserGroupInformation.setConfiguration(conf);

    try {
      UserGroupInformation.loginUserFromKeytab(conf.get(PRINCIPAL), conf.get(KEYTAB));
      System.out.println("UserGroupInformation.isLoginKeytabBased(): "
+UserGroupInformation.isLoginKeytabBased());
      flag = true;
    } catch (IOException e) {
      e.printStackTrace();
    }
    return flag;

  }
```

relogin example code:

```
public Boolean relogin(){
        boolean flag = false;
        try {

          UserGroupInformation.getLoginUser().reloginFromKeytab();
          System.out.println("UserGroupInformation.isLoginKeytabBased(): "
```



```
+UserGroupInformation.isLoginKeytabBased());
          flag = true;
        } catch (IOException e) {
            e.printStackTrace();
        }
        return flag;
    }
```

Prerequisites for using the REST interface of WebHCat to submit an MR task in Streaming mode

The REST interface depends on the streaming packages of Hadoop. Before submitting an MR task to WebHCat in Streaming mode, upload **hadoop-streaming-2.7.0.jar** to the specified path of the HDFS: **hdfs:///apps/templeton/hadoop-streaming-2.7.0.jar**. Log in to the node where the client and Hive service are installed. Assume that the client installation path is **/opt/client**.

source /opt/client/bigdata_env

Run the **kinit** command to log in to the node as the human-machine or machine-machine user.

hdfs dfs -put ${BIGDATA_HOME}/FusionInsight_HD_8.1.0.1/FusionInsight-Hadoop-*/hadoop/share/hadoop/tools/lib/hadoop-streaming-*.jar /apps/templeton/

/apps/templeton/ need to be modified based on different instances. The default instance uses **/apps/templeton/** and the Hive1 instance uses **/apps1/templeton/**. The others follow the same rule

Read and write operations cannot be performed on the same table at the same time

Currently, Hive does not support concurrent operations. Read and write operations cannot be performed on the same table at the same time. Otherwise, query results may be inaccurate and tasks may fail.

A bucket table does not support insert into

A bucket table does not support insert into, and only supports insert overwrite; otherwise, the number of files and the number of buckets will be inconsistent.

Prerequisites for using some REST interfaces of WebHCat

Some REST interfaces of WebHCat depend on the JobHistoryServer instance of MapReduce. The interfaces are as follows:

- mapreduce/jar(POST)
- mapreduce/streaming(POST)
- hive(POST)
- jobs(GET)
- jobs/:jobid(GET)
- jobs/:jobid(DELETE)

Hive Authorization Description

It is recommended that Hive authorization (databases, tables, or views) be performed on the Manager authorization page. Authorization in command-line interface is not recommended except in the **alter databases databases_name set owner='user_name'** scenario.

Hive on HBase partition tables cannot be created

Data of Hive on HBase tables is stored on HBase. Because HBase tables are divided into multiple partitions that are scattered on RegionServer, Hive on HBase partition tables cannot be created on Hive.

A Hive on HBase table does not support insert overwrite

HBase uses a RowKey to uniquely identify a record. If data to be inserted has the same RowKey as the existing data, HBase will use the new data to overwrite the existing data. If insert overwrite is performed for a Hive on HBase table on Hive, only data with the same RowKey will be overwritten.

6.2 Hive Application Development Suggestions

HQL - implicit type conversion

If the query statements use the field value for filtering, do not use the implicit type conversion of Hive to compile HQL. The reason is that the implicit type conversion is not conducive to code reading and migration.

Correct:

```
select * from default.tbl_src where id = 10001;
 select * from default.tbl_src where name = 'TestName';
```

Incorrect:

```
select * from default.tbl_src where id = '10001';
 select * from default.tbl_src where name = TestName;
```

NOTE

Note that the type of the id field in the tbl_src table is Int, and the type of the name field is String.

HQL - object name length

The HQL object names include table names, field names, view names, and index names. It is recommended that the object name not exceed 30 bytes.

An error is reported if an object name of Oracle exceeds 30 bytes. PT also limits object names to 30 bytes.

Excessive long object names are not conductive to code reading, migration, and maintenance.

HQL - statistics of data records

To count the total number of records in a table, use select count(1) from table_name.

To count the number of valid records for a field in a table, use select count(column_name) from table_name.

JDBC - timeout limit

The JDBC provided by Hive supports timeout limit. The default value is 5 minutes. Users can use **java.sql.DriverManager.setLoginTimeout(int seconds)** to change the value. The unit of *seconds* is second.

UDF Management

It is recommended that the administrator creates permanent UDF. This is done to avoid repeated execution of the add jar statement and UDF redefining.

UDF of Hive has some default properties. For example, the default value of **deterministic** is **true** (indicating that the same result will be returned for the same input), and the default value of **stateful** is **true**. Corresponding annotates should be added when user-defined UDF conducts an internal data summary. The following is an example:

```
@UDFType(deterministic = false)
Public class MyGenericUDAFEvaluator implements Closeable {
```

Suggestions on Optimizing Table Partitions

1. It is advised to use partition tables and store data by day when data volume is large and statistics need to be collected on a daily basis.
2. In order to avoid excessive small files, add **distribute by** to the partition field during dynamic partition data insertion.

Suggestions on Optimizing Storage File Formats

Hive supports multiple storage formats, including TextFile, RCFile, ORC, Sequence, and Parquet. If you want to save storage space or query certain fields for the most of time, use columnar storage, for example, ORC files, to create tables.

7 Hudi

7.1 Hudi Development Specifications Overview

Scope

This document describes the design and development rules for the integrated lake warehouse and batch processing solution based on the MRS-Hudi component. It mainly includes the following specifications:

- Data table design
- Resource configuration
- Performance Tuning
- Common Troubleshooting
- Common parameter settings

Terms Conventions

This specification is described in the following terms:

- **Rule:** The rule must be observed during programming.
- **Recommendation:** Principles that must be considered when programming.
- **Description:** The interpretation of the rule or recommendation.
- **Example:** This rule or suggestion is given from both positive and negative aspects.

Scope of application

- Design, develop, test, and maintain data storage and processing based on MRS-Hudi.
- This design specification is based on MRS 3.3.0.

7.2 Hudi Data Sheet Design Specification

7.2.1 Hudi Table Model Design Specifications

rules

- A proper primary key must be set for the Hudi table.

  Hudi tables provide the data update and idempotent write capabilities. This capability requires that primary keys must be set for Hudi tables. Improper primary keys will cause duplicate data. The primary key can be a single primary key or a composite primary key. The primary key cannot have null or null values. To set the primary key, see the following example:

  SparkSQL:

  ```
  //Use primaryKey to specify the primary key. If the primary key is a composite primary key, separate it with commas (,).
  create table hudi_table (
  id1 int,
  id2 int,
  name string,
  price double
  ) using hudi
  options (
  primaryKey = 'id1,id2',
  preCombineField = 'price'
  );
  ```

  SparkDatasource:

  ```
  //Use hoodie.datasource.write.recordkey.field to specify the primary key.
  df.write.format("hudi").
  option("hoodie.datasource.write.table.type", COPY_ON_WRITE).
  option("hoodie.datasource.write.precombine.field", "price").
  option("hoodie.datasource.write.recordkey.field", "id1, id2").
  ```

  FlinkSQL:

  ```
  //Use hoodie.datasource.write.recordkey.field to specify the primary key.
  create table hudi_table(
  id1 int,
  id2 int,
  name string,
  price double
  ) partitioned by (name) with (
  'connector'='hudi',
  'hoodie.datasource.write.recordkey.field' = 'id1,id2',
  'write.precombine.field'='price')
  ```

- The precombine field must be set in the Hudi table.

  During data synchronization, data may be repeatedly written and disordered, for example, abnormal data recovery and abnormal restart of the writer program. You can set the precombine field to a proper value to ensure data accuracy. Old data will not overwrite new data, that is, the idempotent write capability. This field can be of the following types: service table update timestamp, database submission timestamp, and so on. The value of the precombine field cannot be null or null. You can set the precombine field by referring to the following example:

  SparkSQL:

  //Specify the precombine field by using the preCombineField field.

  ```
  create table hudi_table (
  id1 int,
  id2 int,
  name string,
  price double
  ) using hudi
  ```



```
options (
primaryKey = 'id1,id2',
preCombineField = 'price'
);
```

SparkDatasource:

```
//Specify the precombine field by using hoodie.datasource.write.precombine.field.
df.write.format("hudi").
option("hoodie.datasource.write.table.type", COPY_ON_WRITE).
option("hoodie.datasource.write.precombine.field", "price").
option("hoodie.datasource.write.recordkey.field", "id1, id2").
```

Flink:

```
//Specify the precombine field by using write.precombine.field.
create table hudi_table(
id1 int,
id2 int,
name string,
price double
) partitioned by (name) with (
'connector'='hudi',
'hoodie.datasource.write.recordkey.field' = 'id1,id2',
'write.precombine.field'='price')
```

- Flow calculation uses the MOR table.

  Streaming computing is real-time computing with low latency and requires high-performance streaming read and write capabilities. Among the MOR and COW models in Hudi tables, the MOR table has better streaming read and write performance. Therefore, the MOR table model is used in streaming computing scenarios. The following table lists the comparison between the read and write performance of MOR tables.

Comparison Dimension	MOR table	COW Table
Stream write	High	Low
Streaming Read	High	Low
Batch write	High	Low
Batch Read	Low	High

- Real-time into the lake, the table model adopts MOR table.

  Generally, the performance requirements for real-time lake entry are within minutes or at the minute level. Based on the comparison between Hudi and Hudi table models, the MOR table model needs to be selected in the real-time lake entry scenario.

- Use lowercase letters for Hudi table names and column names.

  When multiple engines read and write the same Hudi table, lowercase letters are used to avoid case difference between engines.

The suggestion

- In the Spark batch processing scenario, the COW table is used in the scenario where the write latency is not high.

  In the COW table model, the write speed is low because of write amplification. But COW has very good readability. In addition, batch computing is not sensitive to write latency, so COW tables can be used.

- The Hive metadata synchronization function must be enabled for the Hudi table write task.

  SparkSQL is naturally integrated with Hive, and metadata is not required. This suggestion is applicable to the scenario where the Hudi table is written through the Spark Datasource API or Flin. When the Hudi table is written through the Spark Datasource API or Flin, the configuration item for synchronizing metadata to the Hive needs to be added. This configuration is used to uniformly host the metadata of Hudi tables in the Hive metadata service, facilitating cross-engine data operations and data management.

7.2.2 Hudi Table Index Design Specifications

rules

- Do not modify the table index type.

  The index of the Hudi table determines the data storage mode. If the index type is changed randomly, duplicate existing data and new data in the table will occur and data accuracy will be affected. Common index types are as follows:

  – Bloom index: unique index of the Spark engine. The bloomfiter mechanism is used to write the Bloom index content to the footer of the Parquet file.

  – Bucket index: During data writing, the primary key is used to perform hash calculation and write data into buckets. This index has the fastest write speed, but the number of buckets needs to be properly configured. Both Flink and Spark support this index.

  – Status index: It is unique to the Flink engine. It records the storage location of row records to the status backend. During the cold start of a job, all data storage files are traversed to generate index information.

- If the Flink status index is used, Spark cannot continue to write data after data is written to Flink.

  When writing data to the MOR table of Hudi, Flink generates only log files. The log files will be converted into parquet files by performing the compaction operation. When updating the Hudi table, Spark depends on whether the parquet file exists. If the current Hudi table is written into a log file, duplicate data will be generated if Spark is used to write the Hudi table. In the batch initialization phase, Spark is used to write data to Hudi tables in batches. When Flink is used to write data based on Flink status indexes, the cause is that all data files are traversed to generate status indexes during Flink cold startup.

- In the real-time lake access scenario, the Spark engine uses bucket indexes, and the Flink engine can use bucket indexes or status indexes.

  In real time, the high performance data needs to be imported within minutes or at the minute level. Index selection affects the performance of writing Hudi tables. The performance differences between indexes are as follows:

  – Bucket index.

    Advantage: During the write process, the primary key is written in hash buckets, which provides high performance and is not limited by the data volume of the table. Both the Flink and Spark engines support this function. The Flink and Spark engines can cross-write the same table.

Disadvantage: The number of buckets cannot be dynamically adjusted. If the data volume fluctuates and the data volume of the entire table keeps increasing, a large data file is generated due to a large data volume in a single bucket. Balance improvement needs to be performed in conjunction with partition tables.

– Flink status index.

Advantages: The index information of the primary key exists in the state backend. Data update only needs to check the state backend, which is fast. In addition, the size of the generated data files is stable, and the problem of small files and oversized files is not generated.

Disadvantage: This index is specific to Flink. When the total number of data rows in a table reaches hundreds of millions, state backend parameters need to be optimized to maintain write performance. This index does not support cross-write of Flink and Spark.

- For a table whose data volume keeps increasing, the bucket index must be partitioned by time, and the partition key must be the data creation time.

  According to the characteristics of Flink status index, if the Hudi table exceeds a certain amount of data, the Flink job status backend is under great pressure. To maintain performance, you need to optimize the status backend parameters. In addition, the entire table data needs to be traversed during the cold start of Flink. As a result, the Flink job startup is slow due to a large amount of data. Therefore, for a table with a large amount of data, bucket indexes can be used to avoid complex state backend optimization.

  If the bucket index + partition table mode cannot balance the problem of large Bueckt buckets, you can use the Flink state index and optimize the corresponding configuration parameters according to the specifications.

The suggestion

- If the number of data records in a Flink-based streaming table exceeds 200 million, the bucket index is used. If the number of data records does not exceed 200 million, the Flink status index can be used.

  According to the characteristics of Flink status index, when the Hudi table exceeds a certain amount of data, the Flink job status backend is under great pressure. To maintain performance, you need to optimize the status backend parameters. In addition, the entire table data needs to be traversed during the cold start of Flink. A large amount of data may cause slow startup of Flink jobs. Therefore, for tables with a large amount of data, bucket indexes can be used to avoid complex state backend optimization.

  If the bucket index + partition table mode cannot balance the problem of large Bueckt buckets, you can use the Flink state index and optimize the corresponding configuration parameters according to the specifications.

- Bucket index-based tables are designed based on the data volume of a single bucket, which is 2 GB.

  To prevent a single bucket from being too large, it is recommended that the data volume of a single bucket not exceed 2 GB. (The 2 GB indicates the size of the data content, not the number of data rows or the size of the parquet data file.) to limit the Parquet file size of the corresponding bucket to 256 MB. Balance read and write memory consumption and HDFS storage utilization. Therefore, the limit of 2 GB is only an empirical value because the size of different service data after column-store compression is different.

Why is 2 GB recommended?

- After 2 GB data is stored as a column-store Parquet file, the size of the data file is about 150 MB to 256 MB. Data varies depending on the service. The size of a single HDFS data block is usually 128 MB, which effectively utilizes the storage space.
- The memory space occupied by data read and write is the size of original data (including null values). During big data computing, 2 GB is within the acceptable range for single-task read and write.

If the data volume of a single bucket exceeds the value range, what are the possible impacts?

- OOM may occur in read and write tasks. To solve this problem, increase the memory usage of a single task.
- The read and write performance deteriorates because the amount of data processed by a single task increases, which increases the processing time.

7.2.3 Hudi Table Partition Design Specifications

rules

The partition key cannot be updated:

Hudi has the primary key uniqueness mechanism. However, in a partitioned table scenario, only the primary key in the partition can be unique. Therefore, if the value of the partitioned key changes, multiple rows with the same primary key may exist. In the scenario where data is partitioned by date, you can use the data creation time as the partition field. Do not use the data update time as the partition field.

NOTE

When the Hudi index type is set to Global, Hudi supports cross-partition data update. However, Global index performance is poor and is not recommended.

The suggestion

- The fact table uses the date partition table, and the dimension table uses the non-partition or large-granularity date partition.

  Whether to use a partitioned table depends on the total data volume, increment, and usage mode of the table. From the table attributes, the fact table and dimension table have the following characteristics:

  - Fact table: The data volume is large and the increment is large. Data is read by date and data in a certain period is read.
  - Dimension table: The total amount is small and the increment is small. Most of the dimension table is updated. Data is read in the entire table or filtered by service ID.

  Based on the preceding considerations, if the dimension table is partitioned by day, the number of files is too large. In addition, the full table is read, which causes a large number of file reading tasks. If the large-granularity date partition, such as year partition, is used, the number of partitions and the number of files can be effectively reduced. For dimension tables with small increments, you can also use non-partitioned tables. If a dimension table

contains a large amount of total data or a large amount of incremental data, you can use a service ID for partitioning. In most data processing logic, service conditions are used to filter large dimension tables to improve processing performance. Such tables must be optimized based on service scenarios. You cannot optimize from a date partition alone. The fact table is read by time segment. The number of files read in the last year, month, or day is relatively stable and controllable. Therefore, the date partition table is preferred for fact tables.

- The partition uses the date field and the granularity of the partition table. The granularity must be determined based on the data update scope. The size must be neither too large nor too small.

  The partitioning granularity can be year, month, or day. The goal of the partitioning granularity is to reduce the number of file buckets that are concurrently written, especially when the data volume is updated and the updated data has a regular time range. For example, if the data update proportion in the last month is the largest, partitions can be created by month. If the data updated in the last day accounts for a large proportion, partition the data by day.

  Bucket index is used. Data is written to each bucket in the partition by using the hash algorithm of the primary key. The data volume in each partition fluctuates. Therefore, the number of buckets in a partition is calculated based on the maximum data volume in the partition. In this case, the more fine-grained partitions are, the more redundant buckets are. For example:

  If day-level partitions are used, the average daily data volume is 3 GB, and the maximum daily logs are 8 GB. In this case, tables are created using the number of buckets = 8 GB/2 GB = 4. Daily updates account for a large number of data, and are mainly scattered in the last month. As a result, data is written to the buckets of the whole month, that is, 4 x 30 = 120 buckets. If monthly partitioning is used, the number of buckets in the partition = 3 GB x 30/2 GB = 45 buckets. In this way, the number of buckets to be written is reduced to 45. With limited computing resources, the fewer buckets are written to, the higher the performance.

7.3 Hudi Data Table Management Operation Specifications

7.3.1 Hudi Data Table Compaction Specifications

Updated data in the mor table is written in the form of row-store logs. When the logs are read, they need to be combined based on the primary key and are row-stored. As a result, the log reading efficiency is much lower than that of the parquet. Hudi uses compaction to compress logs into parquet files to improve the log read performance.

rules

- If data is continuously written to a table, perform the compaction at least once within 24 hours.

  For the MOR table, the compaction operation must be performed at least once a day, regardless of whether the MOR table is written in streaming

mode or in batches. If compaction is not performed for a long time, the size of logs in the Hudi table will increase. In this case, the following problems will occur:

- Hudi table reading is slow and requires a lot of resources. This is because reading MOR tables involves log merging, which consumes a lot of resources and is slow.
- A long-term compaction process consumes a lot of resources and is prone to OOM.
- If no Compaction operation is performed to generate Parquet files of the new version, the files of the old version cannot be cleared by the Clean operation, increasing the storage pressure.

- The ratio of CPU to memory is 1:4 to 1:8.

  A Compaction job merges the data in the existing parquet file and the data in the new log file, which consumes a lot of memory resources. Based on the previous table design specifications and the actual traffic fluctuation, it is recommended that the ratio of CPU to memory for a Compaction job be 1. 4 to 1:8, ensuring stable running of Compaction jobs. If the OOM problem occurs in the Compaction, increase the memory usage.

The suggestion

- The Compaction performance is improved by increasing the number of concurrent requests.

  If the ratio of CPU to memory is proper, compaction jobs are stable and a single compaction task can run stably. However, the overall running duration of the Compaction depends on the number of files processed by the Compaction and the number of allocated CPU cores (concurrency capability). Therefore, the Compaction performance can be improved by increasing the number of CPU cores for the Compaciton job. (Note that the ratio of CPU to memory must be ensured when the CPU is increased.)

- Hudi tables use asynchronous compaction.

  To ensure the stable running of the streaming data import job, ensure that the streaming data import job does not perform other tasks during the real-time data import process. For example, Flink performs Compaction when writing data to Hudi. This seems to be a good solution, that is, completing the storage and compaction. However, the compaction operation consumes a lot of memory and I/O, and has the following impact on the streaming database import job:

  - Increased end-to-end latency: Compaction increases the write latency because Compaction is more time-consuming than importing data to the database.
  - Job instability: Compaction brings more instability to the stock-in job. Compaction OOM will cause the entire job to fail.

- You are advised to perform the compaction every 2 to 4 hours.

  Compaction is an important and mandatory maintenance method for MOR tables. For real-time tasks, the compaction process must be decoupled from the real-time tasks. Spark tasks are scheduled periodically to complete asynchronous compaction. The key point of this solution is how to set the period properly. If the period is too short, Spark tasks may be idle. If the period is too long, too many Compaction Plans may be accumulated and not

executed. As a result, Spark tasks take a long time and downstream read job latency is long. In this scenario, the following suggestions are provided: Based on the cluster resource usage, asynchronous Compaction jobs can be scheduled and executed every two or four hours. This is a basic solution for maintaining MOR tables.

- Spark is used to perform compaction asynchronously. Flink is not used to perform compaction.

  The recommended solution for Flink to write hudi is that Flink only writes data and generates compaction plans. A separate Spark job asynchronously executes compaction, clean, and archive. Compaction plan generation is lightweight and has negligible impact on Flink write jobs.

  The procedure for implementing the preceding solution is as follows:

  - **Flink only writes data and generates compaction plans.**

    Add the following parameters to the table creation statement of the Flink stream task to ensure that only the compaction plan is generated when the Flink task writes data to the Hudi.

    ```
    'compaction.async.enabled'='false' //Disable the Flink from executing the Compaction task.
    'compaction.schedule.enabled' ='true' //Enable compaction plan generation.
    'compaction.delta_commits' = '5' //By default, the MOR table attempts to generate a compaction plan after five checkpoints. This parameter needs to be adjusted based on the site requirements.
    'clean.async.enabled' = 'false' //Disable the Clean operation.
    'hoodie.archive.automatic' = 'false' //Disable the Archive operation.
    ```

  - **Spark executes the Compaction plan offline, and performs the Clean and Archive operations.**

    On the scheduling platform (Huawei DataArts can be used), run a scheduled offline task to enable Spark to execute the Compaction plan and clean and archive the Hudi table.

    ```
    set hoodie.archive.automatic = false;
    set hoodie.clean.automatic = false;
    set hoodie.compact.inline = true;
    set hoodie.run.compact.only.inline=true;
    The set hoodie.cleaner.commits.retained = 500; // clean retains the data files corresponding to the latest 500 deltacommits on the timeline. The files corresponding to the deltacommits in the earlier version will be deleted. The value must be greater than the value of compaction.delta_commits. Adjust the value based on the site requirements.
    set hoodie.keep.max.commits = 700; // timeline retains a maximum of 700 deltacommits.
    The set hoodie.keep.min.commits = 501; // timeline retains at least 500 deltacommits. The value must be greater than the value of hoodie.cleaner.commits.retained. Adjust the value based on the site requirements.
    run compaction on <database name>. <table name>; // Executes the Compaction plan.
    run clean on <database name>. <table name>; // Performing the Clean Operation
    The run archivelog on <database name>.<table name>; // performs the Archive operation.
    ```

- Asynchronous Compaction can serialize multiple tables into a job. Tables with similar resource configuration are put into a group. The resource configuration of the group of jobs is the resource required by the table that consumes the most resources.

  For the people who are in the **·Asynchronous compaction is used for the HDI table.** And to the **· Use Spark to asynchronously execute Compaction, not...** For the asynchronous Compaction task mentioned in, the following suggestions are provided:

  - You do not need to develop an asynchronous compaction task for each Hudi table, which leads to high job development costs, explosion of cluster jobs, and ineffective use and release of cluster resources.

– Asynchronous compaction tasks can be completed by executing SparkSQL. Compaction, Clean, and Archive tasks of multiple Hudi tables can be executed in the same task. For example, asynchronous maintenance operations can be performed on table1 and table2.

```
set hoodie.clean.async = true;
set hoodie.clean.automatic = false;
set hoodie.compact.inline = true;
set hoodie.run.compact.only.inline=true;
set hoodie.cleaner.commits.retained = 500;
set hoodie.keep.min.commits = 501;
set hoodie.keep.max.commits = 700;
run compaction on <database name>. <table1>;
run clean on <database name>. <table1>;
run archivelog on <database name>.<table1>;
run compaction on <database name>.<table2>;
run clean on <database name>.<table2>;
run archivelog on <database name>.<table2>;
```

7.3.2 Hudi Data Table Clean Specifications

Clean is also one of the maintenance operations of the Hudi table. This operation needs to be performed on both the MOR and COW tables. The Clean operation is used to clear the files of the old version (data files that are not used by Hudi anymore). This not only saves the time for the Hudi table list process, but also relieves the storage pressure.

rules

The Hudi table must be cleaned.

Clean must be enabled for the MOR and COW tables of Hudi.

- When data is written to the Hudi table, the system automatically determines whether to clean the Hudi table because the clean function is enabled by default **(hoodie.clean.automatic** is set to **true** by default).
- The Clean operation is not triggered every time data is written. At least two conditions must be met:
  a. The Hudi table requires an old version of the file. For COW tables, files of earlier versions must exist as long as the data is updated. For the MOR table, ensure that the data has been updated and compaction has been performed so that the file of the earlier version can be available.
  b. The Hudi table meets the threshold specified by hoodie.cleaner.commits.retained. If Flink writes hudi, the number of checkpoints submitted must exceed the threshold. For batch Hudi, the number of batch write times must exceed the threshold.

The suggestion

- The downstream MOR table uses the batch read mode. The number of clean versions is the number of compaction versions plus 1.

  The MOR table must ensure that the compaction plan can be successfully executed. The compaction plan only records the log files in the Hudi table and the Parquet files to be merged. Therefore, the most important point is to ensure that all the files to be merged exist when the compaction plan is executed. In the Hudi table, only the Clean operation can clear files. Therefore, it is recommended that the Clean triggering threshold (the value of

hoodie.cleaner.commits.retained) be at least greater than the Compaction triggering threshold. (For the Flink task, the value is the value of compaction.delta_commits.)

- Flow calculation is used in the downstream direction of the MOR table. In earlier versions, hour-level calculation is retained.

  If the downstream of the MOR table uses streaming computing, such as Flink streaming read, the historical version can be stored in hours based on service requirements. In this way, incremental data in the last few hours can be read from log files. If the retention duration is too short, When the downstream Flink job is restarted or blocked due to abnormal interruption, the upstream incremental data has been cleaned. Flink needs to read the incremental data from the parquet file, and the performance deteriorates. If the retention period is too long, historical data in logs will be redundantly stored.

  You can reserve the historical version data for two hours according to the following formula:

  Set the number of versions to 3600 x 2/Version interval. The version interval is obtained from the checkpoint period of the Flink job or the upstream batch write period.

- If the service does not have special requirements for storing historical version data in the COW table, set the number of versions to 1.

  Each version of a COW table contains full data of the table. The number of versions that are retained depends on the number of versions that are redundant. Therefore, if the service does not require historical data backtracking, set the number of retained versions to 1, that is, retain the latest version.

- The clean job must be executed at least once a day, which can be executed every 2 to 4 hours.

  The MOR and COW tables of Hudi must be cleaned at least once a day. For details about how to clean the MOR and COW tables, see section 2.2.1.6. The clean function of the COW can automatically determine whether to perform the clean operation when writing data.

7.3.3 Hudi Data Table Archive Specifications

Archive is used to reduce the pressure on Hudi to read and write metadata. All metadata is stored in the following path: Hudi table root directory/.hoodie. If the number of files in the .hoodie directory exceeds 10000, the Hudi table has obvious read and write latency.

rules

Archive must be executed for the Hudi table.

For the MOR and COW tables of Hudi, the Archive function must be enabled.

- When data is written to the Hudi table, the system automatically determines whether to perform the Archive operation because the Archive function is enabled by default (hoodie.archive.automatic is set to true by default).
- The Archive operation is not triggered every time data is written. At least the following conditions must be met:

a. The Hudi table meets the threshold specified by hoodie.keep.max.commits. If the Flink writes data to the hudi, the number of checkpoints submitted must exceed the threshold. If Spark writes data to the hudi, the number of times that the hudi is written must exceed the threshold.

b. The Hudi table has been cleaned. If the Hudi table is not cleaned, the Archive operation will not be executed. (Ignore this condition in MRS 3.3.1-LTS and later versions.)

The suggestion

The Archive job must be executed at least once a day, which can be executed every two to four hours.

The MOR and COW tables of Hudi must be archived at least once a day. For details about how to archive the MOR and COW tables, see section 2.2.1.6. The archive function of the COW can automatically determine whether to execute the data write operation.

7.4 Spark on Hudi Development Specifications

7.4.1 Spark Read/Write Hudi Development Specifications

Specifications of the parameters in various write modes for the Spark write Hudi

Type	Description	Enable parameter	Scenario Selection	Features

upsert	update + insert
	Hudi default write type, which has the update capability.
		This parameter is set by default and does not need to be set to Enabled.
		• SparkSQL:
		`set hoodie.datasource.write.operation=upsert;`
		• DataSource Api:
		`df.write .format("hudi") .options(xxx) .option("hoodie.datasource.write.operation", "upsert") .mode("append") .save("/tmp/tablePath")`
			The default value is selected.	Pros:
				• Small files can be merged.
				• Updates are supported.
				Disadvantages:
				• Write speed go by the rules to the letter.

Type	Description	Enable parameter	Scenario Selection	Features

append	Directly write data without updates	• Spark: Spark does not have the pure append mode. You can use the bulk insert mode instead.
		• SparkSQL:
		`set hoodie.datasource.write.operation = bulk_insert;`
		`set hoodie.datasource.write.row.writer.enable = true;`
		• DataSource Api:
		`df.write`
		`.format("hudi")`
		`.options(xxx)`
		`.option("hoodie.datasource.write.operation", "bulk_insert")`
		`.option("hoodie.datasource.write.row.writer.enable", "true")`
		`.mode("append")`
		`.save("/tmp/tablePath")`
			High throughput and no data update scenario.	Pros:
				• Fastest write speed.
				Disadvantages:
				• Small files cannot be merged.
				• No update capability.
				• Clustering is required to merge small files.

delete	Delete operation	No parameter is required. You can directly use the delete syntax. `delete from tableName where primaryKey='id1';`	The SQL statement deletes data.	Same as the upsert type.

Insert overwrite	Override partition	No parameter is required. Use the insert overwrite syntax directly.
		`insert overwrite table tableName partition (dt = '2021-01-04')`
		`select * from srcTable;`
			Partition level again.	Overwrite the partition.

Insert overwrite table	Override the entire table	No parameter is required. Use the insert overwrite syntax directly.
		`insert overwrite table tableName`
		`select * from srcTable;`
			Rewrite it all.	Overwrite the entire table.

Type	Description	Enable parameter	Scenario Selection	Features

Bulk_insert	Batch Import	• SparkSQL:
		`set hoodie.datasource.write.operation = bulk_insert;`
		`set hoodie.datasource.write.row.writer.enable = true;`
		• DataSource Api:
		`df.write`
		`.format("hudi")`
		`.options(xxx)`
		`.option("hoodie.datasource.write.operation", "bulk_insert")`
		`.option("hoodie.datasource.write.row.writer.enable", "true")`
		`.mode("append")`
		`.save("/tmp/tablePath")`
			You are advised to use this tool during table initialization and migration.	The mode is the same as the append mode.

Specifications for Spark to read Hudi parameters in incremental mode

Type	Description	Enable parameter	Scenario Selection	Features

snapshot	Real-time data reading.	Default value. No parameter is required to enable this function.
		SparkSQL:
		`set hoodie.datasource.query.type=snapshot;`
		DataSource Api:
		`val df = spark.read`
		`.format("hudi")`
		`.option("hoodie.datasource.query.type", "snapshot")`
		`.load("tablePath")`
			Default selection	Each read data is the latest, and the data is visible immediately after being written.

Type	Description	Enable parameter	Scenario Selection	Features

incremental	Incremental query. Only the data between two commit operations is queried.	• SparkSQL:
		set hoodie.tableName.consume.mode=INCREMENTAL;// The current table must be read in incremental mode.
		set hoodie.tableName.consume.start.timestamp=20201227153030;// Specify the initial incremental pull. commit
		set hoodie.tableName.consume.end.timestamp=20210308212318; // specifies the end of the incremental pull. If the end of the commit operation is not specified, the latest commit operation is used.
		select * from tableName where `_hoodie_commit_time`>'20201227153030'and `_hoodie_commit_time`<='20210308212318'; //The results must be filtered by start.timestamp and end.timestamp. If end.timestamp is not specified, the results are filtered by start.timestamp.
		set hoodie.tableName.consume.mode=SNAPSHOT; // After using the incremental mode, the query mode must be reset.
		• DataSource Api:
		val df = spark.read
		.format("hudi")
		.option("hoodie.tableName.consume.mode", "INCREMENTAL")
		.option("hoodie.tableName.consume.start.timestamp", "20201227153030")
		.option("hoodie.tableName.consume.end.timestamp", "20210308212318")
		.load("tablePath")
		.where ("`_hoodie_commit_time`>'20201227153030' and `_hoodie_commit_time`<='20210308212318 '")
			In the streaming processing scenario, only incremental data is obtained each time.	Reads only the data between two commit operations. It is not a full table scan, which is much more efficient than obtaining the data before the commit operation by using the where condition.

Type	Description	Enable parameter	Scenario Selection	Features

read_optimized	Read-optimized view.
	Only the data in the parquet file in the table is read. For the mor table, new data is written to the log. Therefore, the data read in this mode is not the latest.
		• SparkSQL:
		When the Mor table is synchronized to Hive, three tables are generated: primary table, ro table, and rt table. The ro table is the read optimization table. You can directly read the ro table.
		`select * from tableName_ro;`
		• DataSource Api:
		`val df = spark.read .format("hudi") .option("hoodie.datasource.query.type", "read_optimized") .load("tablePath")`
			The query performance is required, but the data delay is acceptable.	For the mor table, the performance of this read mode is much faster than that of the real-time table. In this mode, log data is not read and can be read only after data is compacted. Therefore, data reading in this mode has a certain data latency.

7.4.1.1 SparkSQL table creation parameter specifications

The rules

- When creating a table, you must specify primaryKey and preCombineField.

  Hudi tables provide the data update and idempotent write capabilities. This capability requires that primary keys must be set for data records to identify duplicate data and update operations. If the primary key is not specified, the table will lose the data update capability. If the preCombineField parameter is not specified, duplicate primary keys will occur.

Parameter name	Parameter Description	Input Value	Description
primaryKey	primary key of hudi	On Demand	It must be specified. It can be a composite primary key but must be globally unique.

Parameter name	Parameter Description	Input Value	Description
preCombineField	Pre-combination key. Multiple data records with the same primary key are merged based on this field.	On demand	This parameter is mandatory. Data with the same primary key will be merged by this field. You cannot specify multiple fields.

- Do not set hoodie.datasource.hive_sync.enable to false during table creation.

  If this parameter is set to false, newly written partitions cannot be synchronized to Hive Metastore. The query engine loses data when reading the data because the newly written partition information is missing.
- Do not set the Hudi index type to INMEMORY.

  This index is for test use only. Using the index in the production environment will cause duplicate data.

Creating an example

```
create table data_partition(id int, comb int, col0 int, yy int, mm int, dd int)
using hudi -- Specify the hudi data source.
partitioned by(yyy, mm, dd) --Specify the partition. Multi-level partitioning is supported.
location '/opt/log/data_partition' --Specify the path. If the table is not created in Hive Warehouse, the table is created.
options(
type='mor', --Table type: mor or cow
primaryKey='id', --primary key, which can be a compound primary key but must be globally unique.
preCombineField='comb' --Pre-combined field. Data with the same primary key will be merged by this field. Currently, only one field cannot be specified.
)
```

7.4.1.2 Specifications for Spark to read Hudi parameters in incremental mode

rules

Before the incremental query, you must specify the query mode of the current table as the incremental query mode and rewrite the query mode of the table after the query.

If the incremental query is complete and the table query mode is not set again, subsequent real-time query will be affected.

Example

```
set hoodie.tableName.consume.mode=INCREMENTAL;// The current table must be read in incremental mode.
set hoodie.tableName.consume.start.timestamp=20201227153030;// Specify the initial incremental pull. commit
set hoodie.tableName.consume.end.timestamp=20210308212318; // specifies the end of incremental pulling. If this parameter is not specified, the latest commit command is used.
select * from tableName where `_hoodie_commit_time`>' 20201227153030'and `_hoodie_commit_time`<=' 20210308212318'; //The results must be filtered based on start.timestamp and end.timestamp. If end.timestamp is not specified, the results must be filtered based only on start.timestamp.
```



```
The set hoodie.tableName.consume.mode=SNAPSHOT; // has used the incremental mode, and the query mode must be reset.
```

7.4.1.3 Specifications for setting the compaction parameter in the Spark asynchronous task execution table

- Do not manually run the run schedule command to generate a compaction plan if the write job is not stopped.

  Error example:

  ```
  run schedule on dsrTable
  ```

  If other tasks are writing data to the table, data loss will occur after this operation is performed.

- When running the run compaction command, do not set hoodie.run.compact.only.inline to false. Set hoodie.run.compact.only.inline to true.

  Error example:

  ```
  set hoodie.run.compact.only.inline=false;
  run compaction on dsrTable;
  ```

  If other tasks are writing data to the table, performing the preceding operations will cause data loss.

  Correct example: Asynchronous Compaction

  ```
  set hoodie.compact.inline = true;
  set hoodie.run.compact.only.inline=true;
  run compaction on dsrTable;
  ```

7.4.1.4 Spark Table Data Maintenance Specifications

Do not run the Alter command to modify the key attributes of a table: type/primaryKey/preCombineField/hoodie.index.type.

Run the following statement to modify the key attributes of the table:

```
alter table dsrTable set tblproperties('type'='xx');
alter table dsrTable set tblproperties('primaryKey'='xx');
alter table dsrTable set tblproperties('preCombineField'='xx');
alter table dsrTable set tblproperties('hoodie.index.type'='xx');
```

Engines such as Hive and Presto can directly modify table attributes. However, such modification will cause duplicate data in the entire Hudi table or even data damage. Therefore, do not modify the preceding attributes.

7.4.1.5 Suggestions for Spark Concurrently Write Hudi Data

- In concurrent scenarios, the inter-partition concurrent write mode is recommended. That is, different write tasks are written to different partitions.

  Partition concurrency parameter control:

  - SQL mode:

    ```
    set hoodie.support.partition.lock=true;
    ```

  - DataSource API mode:

    ```
    df.write
    .format("hudi")
    .options(xxx)
    .option("hoodie.support.partition.lock", "true")
    .mode(xxx)
    .save("/tmp/tablePath")
    ```


NOTE

The preceding parameters must be set for all inter-partition concurrent write tasks.

- Do not perform concurrent writes in the same partition. To perform concurrent writes, you need to enable the Hudi OCC mode and strictly comply with the concurrency parameter settings. Otherwise, table data may be damaged.

  Concurrent OCC parameter control:

  - SQL mode:

    ```
    //Enable the OCC.
    set hoodie.write.concurrency.mode=optimistic_concurrency_control;
    set hoodie.cleaner.policy.failed.writes=LAZY;
    //Enable the concurrent lock ZooKeeper.
    set
    hoodie.write.lock.provider=org.apache.hudi.client.transaction.lock.ZookeeperBasedLockProvider; /
    / Setting the ZooKeeper Lock
    set hoodie.write.lock.zookeeper.url=<zookeeper_url>; // Setting the ZooKeeper Address
    set hoodie.write.lock.zookeeper.port=<zookeeper_port>; // Setting the Use of the ZooKeeper Port
    set hoodie.write.lock.zookeeper.lock_key=<table_name>; // Set lock name
    set hoodie.write.lock.zookeeper.base_path=<table_path>; // Setting the ZooKeeper Lock Path
    ```

  - DataSource API mode:

    ```
    df.write
    .format("hudi")
    .options(xxx)
    .option("hoodie.write.concurrency.mode", "optimistic_concurrency_control")
    .option("hoodie.cleaner.policy.failed.writes", "LAZY")
    .option("hoodie.write.lock.zookeeper.url", "zookeeper_url")
    .option("hoodie.write.lock.zookeeper.port", "zookeeper_port")
    .option("hoodie.write.lock.zookeeper.lock_key", "table_name")
    .option("hoodie.write.lock.zookeeper.base_path", "table_path")
    .mode(xxx)
    .save("/tmp/tablePath")
    ```

NOTE

1. The preceding parameters must be set for all concurrent write tasks. The OCC does not ensure that all concurrent write tasks are successfully executed. When multiple write tasks update the same file, only one task succeeds.
2. In the concurrent scenario, the cleaner policy must be set to Lazy. Therefore, junk files cannot be automatically deleted.

7.4.2 Suggestions on configuring resources for Spark read and write Hudi resources

- According to the resource configuration rules for the Hudi task of Spark, the ratio of memory to CPU cores is 2:1, and the ratio of off-heap memory to CPU cores is 0.5:1. That is, one core, requiring 2 GB heap memory and 0.5 GB non-heap memory.

  NOTE

  In the Spark initialization and import scenario, the preceding resource ratio needs to be adjusted because the amount of data to be processed is large. The recommended ratio of memory to core is 4:1 and the ratio of non-heap memory to core is 1:1.

  Example:

  ```
  spark-submit
  --master yarn-cluster
  --executor-cores 2 //Core
  --executor-memory 4g //Heap memory
  --conf spark.executor.memoryOverhead=1024 // off-heap memory
  ```


- Spark-based ETL calculation: The recommended ratio of CPU core to memory is greater than 1:2, and the recommended ratio is from 1:4 to 1:8.

  The previous rule refers to the resource ratio of pure read and write. If Spark jobs have service logic calculation in addition to read and write, this process will cause memory increase. Therefore, it is recommended that the ratio of CPU cores to memory be greater than 1:2. If the logic is complex, increase the memory. This should be adjusted based on the actual situation. Generally, the default value range is 1:4 to 1:8.

- It is recommended that the number of CPU cores be greater than or equal to the number of buckets. (The partition table may be written to multiple partitions each time. In ideal conditions, the recommended number of CPU cores = Number of write partitions x Number of buckets. If the actual number of cores is less than the value, the write performance decreases linearly.)

  Example:

  The number of buckets in the current table is three, and the number of partitions that are written to the table is two. It is recommended that the number of cores configured for the Spark import task be greater than or equal to 3 x 2.

  ```
  spark-submit
  --master yarn-cluster
  --executor-cores 2
  --executor-memory 4g
  --excutor-num 3
  ```

  The preceding configuration indicates that the number of excutor-num*executor-cores=6 >= partitions multiplied by the number of buckets is 6.

7.4.3 Spark On Hudi Performance Optimization

Optimizing Spark Shuffle Parameters to Improve Hudi Write Efficiency

- If spark.shuffle.readHostLocalDisk is set to true, the local disk reads shuffle data, reducing network transmission overhead.
- If spark.io.encryption.enabled is set to false, the shuffle process is disabled from writing encrypted disks, improving the shuffle efficiency.
- Set spark.shuffle.service.enabled to true to enable the shuffle service and improve the stability of the shuffle task.

Configuration Item	Cluster Default Value	After adjustment
--conf spark.shuffle.readHostLocalDisk	false	true
--conf spark.io.encryption.enabled	true	false
--conf spark.shuffle.service.enabled	false	true

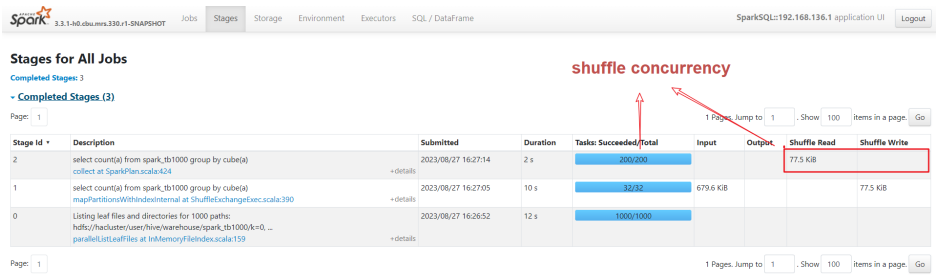
Adjusting Spark Scheduling Parameters to Optimize the Spark Scheduling Delay in OBS Scenarios

- If OBS storage is enabled, Spark locality can be disabled to improve Spark scheduling efficiency.

Configuration Item	Cluster Default Value	After adjustment
--conf spark.locality.wait	3s	0s
--conf spark.locality.wait.process	3s	0s
--conf spark.locality.wait.node	3s	0s
--conf spark.locality.wait.rack	3s	0s

Optimizes the shuffle parallelism and improves the Spark processing efficiency.

The following figure shows the shuffle concurrency.



The default cluster size is 200. You can set the job size separately. If the bottleneck stage (long execution time) is found and the number of cores allocated to the current job is greater than the number of concurrent jobs, the concurrency is insufficient. Optimize the configuration as follows:

Scenario	Configuration Item	Cluster Default	After adjustment
Jar Job	spark.default.parallelism	200	Set this parameter to twice the number of available resources in the actual job.
SQL Job	spark.sql.shuffle.partitions	200	Set this parameter to twice the number of available resources in the actual job.
Hudi Warehousing Operation	hoodie.upsert.shuffle.parallelism	200	Used by non-bucket tables. Set this parameter to twice the number of available resources.

CAUTION

When **spark.dynamicAllocation.enabled** is set to **true**, resources are evaluated based on **spark.dynamicAllocation.maxExecutors**.

Bucket table. Bucket tailoring can be enabled to improve primary key click query efficiency.

Example:

The service uses the primary key ID as the query condition to perform the point query. For example, select xxx where id = idx...

When creating a table, you can add the following attributes to improve the query efficiency. By default, the attribute value is primaryKey, that is, primary key.

```
hoodie.bucket.index.hash.field=id
```

When initializing a Hudi table, you can quickly write data in BulkInsert mode.

Example:

```
set hoodie.combine.before.insert=true; //: deduplicates the data before the database is imported. If the data is not duplicate, you do not need to set this parameter.
set hoodie.datasource.write.operation = bulk_insert; // specifies the bulk insert mode.
set hoodie.bulkinsert.shuffle.parallelism = 4; // specifies the degree of parallelism during bulk_insert data writing, which is equal to the number of partition parquet files saved after data writing.
insert into dsrTable select * from srcTabble
```

Enable log column tailoring to improve the query efficiency of the mor table.

When the mor table is read, logs and Parquet are combined, and the performance is not satisfactory. Log column tailoring can be enabled to reduce the I/O read overhead during combination.

Run the following command to query the SparkSQL database:

```
set hoodie.enable.log.column.prune=true;
```

Other parameters are optimized when Spark processes Hudi tables.

- Set spark.sql.enableToString to false to reduce the memory usage when Spark parses complex SQL statements and improve the parsing efficiency.
- If spark.speculation is set to false, speculative execution is disabled. Enabling this parameter consumes extra CPU resources. In addition, Hudi does not support this parameter. If this parameter is enabled, files may be damaged.

Configuration Item	Cluster Default	After adjustment
--conf spark.sql.enableToString	true	false
--conf spark.speculation	false	false

7.5 Bucket Tuning Example

7.5.1 Creating a Bucket Index Table

Common parameters for setting bucket indexes are as follows:

- Spark:
  ```
  hoodie.index.type=BUCKET
  hoodie.bucket.index.num.buckets=5
  ```
- Flink
  ```
  index.type=BUCKET
  hoodie.bucket.index.num.buckets=5
  ```

Determine whether to use a partitioned table or a non-partitioned table.

Tables are classified into fact tables and dimension tables based on their application scenarios.

- A fact table usually has a large amount of data, mainly new data, and a small proportion of updated data. In addition, most updated data is within a recent period (year, month, or day). When the downstream reads the table for ETL calculation, the downstream usually uses the time range (for example, the latest day, month, or year) for tailoring. Such a table can be partitioned based on the data creation time to ensure the optimal read and write performance.
- Generally, the data volume of a dimension table is small. The data volume of the entire table is mainly updated and few new data is added. The table data volume is stable. In addition, full data needs to be read for ETL calculation such as join. Therefore, the performance of non-partitioned tables is better.
- **Do not update the partition key of a partitioned table. Otherwise, duplicate data will be generated.**

Exception scenario: ultra-large dimension table and ultra-small fact table

In special cases, for example, a dimension table with a large number of data continuously added exists. (The table data volume is greater than 200 GB or the daily growth volume exceeds 60 MB.) or a fact table with very small data volume. (The table data volume is less than 10 GB and will not exceed 10 GB in the next three to five years.) Exception handling needs to be performed based on specific scenarios:

- Dimension table with a large number of data continuously added

  Method 1: Reserve the number of buckets. If a non-partitioned table is used, you need to estimate the data increment in a long period of time to increase the number of buckets in advance. The disadvantage is that files will continue to expand as data increases.

  Method 2: Large-granularity partitioning (recommended). If a partition table is used, calculate the data based on the data growth. For example, yearly partitioning is used. This method is more troublesome, but the table does not need to be imported again after several years.

  Method 3: Data aging: Analyze whether invalid dimension data can be deleted from large dimension tables based on service logic to reduce the data scale.
- A fact table with a very small amount of data

  In this way, the non-partitioned table can be used to reserve more buckets to improve the read and write performance while estimating the data growth over a long period of time.

Check the number of buckets in the table.

The number of buckets in the Hudi table is related to the table performance. Therefore, pay special attention to the setting.

The following are the key information for setting the number of buckets. You need to confirm the information before creating a table.

- Non-partition table
  a. Total number of data records in a single table = select count(1) from tablename (required when entering the lake)
  b. Size of a single data record = 1 KB on average (Huawei recommends that you run the select * from tablename limit 100 command to paste the query result in Notepad++ to obtain the size of 100 records and divide the size by 100 to obtain the average size of a single record.)
  c. Data volume in a single table (GB) = Total number of data records in a single table x Data size in a single table/1024/1024
  d. Number of buckets for non-partitioned tables = MAX(Data volume of a single table (GB)/2 GB x 2) Roundup, 4)
- Partition table
  a. Maximum number of partition data records in the last month = Consult the product line before entering the lake
  b. Size of a single data record = 1 KB on average (Huawei recommends that you run the select * from tablename limit 100 command to paste the query result in Notepad++ to obtain the size of 100 data records and divide the size by 100 to obtain the average size of a single data record.)
  c. Data volume of a single partition (GB) = Maximum number of data records in the partition in the last month x Data size of a single table/1024/1024
  d. Number of buckets in the partition table = MAX(Data volume of a single partition (GB)/2 GB, rounded up, 1)

CAUTION

The total data size of the table is used, not the size of the compressed file.

An even number of buckets is recommended. Set the minimum number of buckets for a non-partitioned table to 4 and that for a partitioned table to 1.

Confirm the SQL statement for creating a table.

DataArts supports operations on Hudi tables using Spark JDBC and Spark APIs.

- In Spark JDBC mode, public resources are used and Spark jobs do not need to be started independently. However, resources and configuration parameters required for SQL execution cannot be specified. Therefore, you are advised to create tables or query a small amount of data.
- When SQL statements executed in Spark API mode are used, it takes a long time to start a Spark job independently. However, you can set running

program parameters to specify parameters such as resources required by the job. Batch import is recommended.

A job uses APIs to specify resources to prevent other tasks from occupying JDBC resources for a long time.

CAUTION

If DataArts uses Spark APIs to operate Hudi tables, you must add the parameter **--conf spark.support.hudi=true** and run jobs by scheduling.

Creating Hudi Tables Using DataArts

DataArts supports operations on Hudi tables using Spark JDBC and Spark APIs.

- In Spark JDBC mode, public resources are used and Spark jobs do not need to be started independently. However, resources and configuration parameters required for SQL execution cannot be specified. Therefore, you are advised to create tables or query a small amount of data.
- When SQL statements executed in Spark API mode start a Spark job independently, it takes a certain period of time. However, you can specify parameters such as resources required by the job by configuring running program parameters. Batch import is recommended.

A job uses APIs to specify resources to prevent other tasks from occupying JDBC resources for a long time.

CAUTION

If DataArts uses Spark APIs to operate Hudi tables, you must add the parameter **--conf spark.support.hudi=true** and run jobs by scheduling.

7.5.2 Hudi table initialization

1. Usually, a Spark job is used to initialize and import inventory data. The initialization data volume is large. Therefore, you are advised to use APIs to provide sufficient resources for the initialization.
2. In the scenario where Flink or Spark stream jobs need to be written in real time after batch initialization, you are advised to filter messages on the and consume the messages from a specified time range to control the repeated data access volume. (For example, after Spark initialization is complete, Flink filters out data generated two hours ago when consuming Kafka.). If Kafka messages cannot be filtered, you can access the data in real time to generate offsets, truncate tables, import historical data, and enable real-time data.

NOTE

1. If the table already contains data and no truncate table exists before batch initialization, the batch data will be written into a large log file, which poses great pressure on subsequent compaction and requires more resources.
2. Hudi tables are stored in Hive metadata. There should be one internal table (manually created) and two external tables (automatically created after data is written).
3. Two external tables named _ro (Users read only the merged parquet file, that is, read the optimized view chart.), _rt (Read the latest version of data written in real time, i.e. real-time view chart).

7.5.3 Real-time Task Access

Real-time jobs are usually completed by Flink SQL or SparkStreaming. For real-time streaming tasks, compaction plans are generated synchronously and asynchronously.

- Configure the Hudi table on the sink end in a Flink SQL job as follows:

```
create table denza_hudi_sink (
$HUDI_SINK_SQL_REPLACEABLE$
) PARTITIONED BY (
years,
months,
days
) with (
'connector' = 'hudi', //Specifies that the Hudi table is written.
'path'='obs://XXXXXXXXXXXXXXXXXX/', //Specify the path for storing the Hudi table.
'table.type'='MAKEED_ON_READ', //Hudi table type
'hoodie.datasource.write.recordkey.field' = 'id', //Primary key
'write.precombine.field'='vin', //Combined field
'write.tasks' = '10', //Flink write parallelism
'hoodie.datasource.write.keygenerator.type' = 'COMPLEX', //Specify the KeyGenerator, which is the same as the Hudi table type created by Spark.
' hoodie.datasource.write.hive_style_partitioning '='true', //Use the partition format supported by Hive.
'read.streaming.enabled' = 'true', //Enable stream reading.
'read.streaming.check-interval'='60', //checkpoint interval, in seconds.
'index.type' = 'BACKET', //Specify the index type of the Hudi table as BUCKET.
'hoodie.bucket.index.num.buckets' = '10', //Specify the number of buckets.
'compaction.delta_commits' = '3', //Interval for the commit file generated by the compaction
'compaction.async.enabled' = 'false', //Disable the asynchronous execution of the compaction.
'compaction.schedule.enabled' ='true', //compaction synchronously generates a plan.
'clean.async.enabled'='false', //Disable asynchronous clean.
'hoodie.archive.automatic'='false', //Automatic archive disabled
'hoodie.clean.automatic'='false', //Automatic clean is disabled.
'hive_sync.enable' = 'true', // Automatically synchronize Hive tables.
'hive_sync.mode' = 'jdbc', //The Hive table synchronization mode is jdbc.
'hive_sync.jdbc_url'='', //Jdbc URL for synchronizing Hive tables
'hive_sync.db' = 'hudi_cars_byd', //Database for synchronizing Hive tables
'hive_sync.table'='byd_hudi_denza_1s_mor', //Synchronize the table name of the Hive table.
'hive_sync.metastore.uris' = 'thrift://XXXXX:9083', //Metastore URI for synchronizing Hive tables
'hive_sync.support_timestamp' = 'true', //The Hive table supports the timestamp format.
'hive_sync.partition_extractor_class' = 'org.apache.hudi.hive.MultiPartKeysValueExtractor' / /
Synchronize the extractor class of the Hive table.
);
```

- The following table lists the common parameters used by the Spark Streaming to write data to the Hudi table. (The meaning of the parameter is similar to that of flink and is not commented out.)

```
hoodie.table.name=
hoodie.index.type=BUCKET
hoodie.bucket.index.num.buckets=3
hoodie.datasource.write.precombine.field=
hoodie.datasource.write.recordkey.field=
hoodie.datasource.write.partitionpath.field=
hoodie.datasource.write.table.type= MERGE_ON_READ
```



```
hoodie.datasource.write.hive_style_partitioning=true
hoodie.compact.inline=true
hoodie.schedule.compact.only.inline=true
hoodie.run.compact.only.inline=false
hoodie.clean.automatic=false
hoodie.clean.async=false
hoodie.archive.async=false
hoodie.archive.automatic=false
hoodie.compact.inline.max.delta.commits=50
hoodie.datasource.hive_sync.enable=true
hoodie.datasource.hive_sync.partition_fields=
hoodie.datasource.hive_sync.database=
hoodie.datasource.hive_sync.table=
hoodie.datasource.hive_sync.partition_extractor_class=org.apache.hudi.hive.MultiPartKeysValueExtracto
r
```

7.5.4 Offline Compaction Configuration

For real-time services of MOR tables, compaction plans are generated during data write. Therefore, DataArts or scripts need to be used to schedule SparkSQL to execute the generated compaction plans.

- Execution parameter

```
set hoodie.compact.inline = true; // Enable compaction.
The set hoodie.run.compact.only.inline = true; //compaction executes only the generated plans and
does not generate new plans.
set hoodie.cleaner.commits.retained = 120; // Clearing and Retaining 120 Commits
A maximum of 140 commit records can be retained in the set hoodie.keep.max.commits = 140; //
archive.
At least 121 commit records can be retained in the set hoodie.keep.min.commits = 121; // archive.
set hoodie.clean.async = false; // Enable asynchronous cleanup.
set hoodie.clean.automatic = false; //Disable the automatic cleaning function to prevent the
compaction operation from starting the clean operation.
run compaction on $tablename; //Execute the compaction plan.
run clean on $tablename; //Run the clean operation to delete redundant versions.
run archivelog on $tablename; //Merge and clear metadata files by running archivelog.
```

  NOTE

  Do not set the clearance and archiving parameters to a large value. Otherwise, the Hudi table performance will be affected. Therefore, you are advised to:

  Two times the number of commit operations required by the hoodie.cleaner.commits.retained = compaction

  hoodie.keep.min.commits = hoodie.cleaner.commits.retained + 1

  hoodie.keep.max.commits = hoodie.keep.min.commits + 20

  Run the clean and archive commands after the compaction command is executed. The clean and archive logs have low requirements on resources. To avoid resource waste, you can configure the compaction task as a task when DataArts is used to schedule the clean and archive logs as a task and configure different resources for the clean and archive logs to save resources.

- Execution resource
  a. The interval for scheduling compaction plans must be smaller than the interval for generating compaction plans. For example, if a compaction plan is generated about one hour, the scheduling task for executing the compaction plan must be scheduled at least every half an hour.
  b. For the resources configured for the Compaction job, the number of vcores must be at least equal to or greater than the number of buckets in a single partition. The ratio of the number of vcores to the memory must be 1:4, that is, one vcore is configured with 4 GB memory.

8 IoTDB

8.1 IoTDB Application Development Rules

Set a Proper Number of Storage Groups

A proper number of storage groups can improve performance. The system processes I/O requests first in the memory cache and then switch to the stored files or folders if no result is found. In this regard, a large number of storage groups will store too many files or folders, take up large amounts of memory, and slow down the system I/O speed as a result. A small number will reduce the concurrency and block write commands.

Set a balanced number of storage groups based on your data scale and usage scenarios to achieve better system performance.

All Time Series Must Start with root and End with the Sensor

The time series can be considered as the complete path of the sensor that generates the time series data. In IoTDB, all time series must start with **root** and end with the sensor.

8.2 IoTDB Application Development Suggestions

Use the Native Session API to Avoid SQL Concatenation

For the sample of calling the IoTDB Session API in a cluster in security mode, see **IoTDB Session**. For that in a cluster in common mode, see **IoTDB Session**.

Use Write APIs with High Performance Based on Service Requirements

The write APIs are ranked as follows in descending order of their performance:

insertTablets (inserts multiple tablets, from multiple rows of the same column on multiple devices) >

insertTablet (inserts a tablet, from multiple rows of the same column on a single device) >

insertRecordsOfOneDevice (inserts multiple records, from multiple rows of different columns on a single device) >

insertRecords(Object value) (inserts multiple records, from multiple rows of different columns on multiple devices) >

insertRecords(String value) (inserts multiple records, from multiple rows of different columns on multiple devices) >

insertRecord (inserts a record, from only one row on a single device)

Do Not Use the Same Client to Initiate Concurrent Connections

An IoTDB client can connect to only one IoTDBServer. A large number of concurrent connections from the same client to IoTDBServer will deteriorate the connection performance. You can use multiple clients to connect to IoTDBServer based on service requirements to achieve load balancing.

Use SessionPool to Reuse Connections

Use the distributed technology to cache sessions so that the client does not need to create sessions for each read or write request. Alternatively, use SessionPool to reuse connections.

Close ResultSet and SessionDataSet in a Timely Manner

Close ResultSet and SessionDataSet in a timely manner to avoid resource wastes.

9 Kafka

9.1 Kafka Application Development Rules

Create Topics by calling Kafka APIs (AdminZkClient.createTopic)

- For Java programming languages, correct examples are as follows:

```
import kafka.zk.AdminZkClient;
import kafka.zk.KafkaZkClient;
import kafka.admin.RackAwareMode;
…
KafkaZkClient kafkaZkClient = KafkaZkClient.apply(zkUrl, JaasUtils.isZkSecurityEnabled(),
zkSessionTimeoutMs, zkConnectionTimeoutMs, Int.MaxValue(), Time.SYSTEM, "", "", null);
AdminZkClient adminZkClient = new AdminZkClient(kafkaZkClient);
adminZkClient.createTopic(topic, partitions, replicas, new Properties(), RackAwareMode.Enforced
$.MODULE$);
…
```

- For Scala programming languages, correct examples are as follows:

```
import kafka.zk.AdminZkClient;
import kafka.zk.KafkaZkClient;
…
val kafkaZkClient: KafkaZkClient = KafkaZkClient.apply(zkUrl, JaasUtils.isZkSecurityEnabled(),
zkSessionTimeoutMs, zkConnectionTimeoutMs, Int.MaxValue, Time.SYSTEM, "", "")
val adminZkClient: AdminZkClient = new AdminZkClient(kafkaZkClient)
adminZkClient.createTopic(topic, partitions, replicas)
```

The number of Partition copies must be less than or equal to the number of nodes

Copies of Topic Partitions in Kafka are used for improving data reliability. Copies of the same Partition are distributed on different nodes. Therefore, the number of Partition copies must be less than or equal to the number of nodes.

Set the fetch.message.max.bytes parameter of the Consumer client

The value of fetch.message.max.bytes muster be equal to or greater than the maximum number of bytes of messages that the Producer client generates each time. If the value is too small, the messages generated by the Producer client cannot be consumed successfully by the Consumer client.

9.2 Kafka Application Development Suggestions

In the same group, the number of consumers and that of Topic Partitions to be consumed should be the same

If the number of consumers is greater than that of Topic Partitions, some consumers cannot consume Topics. If the number of consumers is smaller than that of Topic Partitions, concurrent consumption cannot be fully represented. Therefore, the number of consumers and that of Topic Partitions to be consumed should be the same.

Avoid writing data with single ultra-large log

Data with single ultra-large log can affect efficiency and writing. Under such circumstance, modify the values of the **max.request.size** and **max.partition.fetch.bytes** configuration items when initializing Kafka producer instances and consumer instances, respectively.

For example, set **max.request.size** and **max.partition.fetch.bytes** to **5252880**.

```
// Protcol type:configuration SASL_PLAINTEXT or PLAINTEXT
props.put(securityProtocol, kafkaProc.getValues(securityProtocol, "SASL_PLAINTEXT"));
// service name
props.put(saslKerberosServiceName, "kafka");
props.put("max.request.size", "5252880");
// Security protocol type
props.put(securityProtocol, kafkaProc.getValues(securityProtocol, "SASL_PLAINTEXT"));
// service name
props.put(saslKerberosServiceName, "kafka");
props.put("max.partition.fetch.bytes","5252880");
```


10 Mapreduce

10.1 MapReduce Application Development Rules

Inherit the Mapper abstract class.

The map() and setup() methods are called during the Map procedure of a MapReduce task.

Example:

```
public static class MapperClass extends

Mapper<Object, Text, Text, IntWritable> {
/**
* map input. The key indicates the offset of the original file, and the value is a row of characters in the original file.
* The map input key and value are provided by InputFormat. You do not need to set them. By default, TextInputFormat is used.
*/
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
//Custom implementation
}
/**
* The setup() method is called only once before the map() method of a map task or the reduce() method of a reduce task is called.*/
public void setup(Context context) throws IOException,
InterruptedException {
// Custom implementation
}
}
```

Inherit the Reducer abstract class.

The reduce() and setup() methods are called during the Reduce procedure of a MapReduce task.

Example:

```
public static class ReducerClass extends

Reducer<Text, IntWritable, Text, IntWritable> {

/**
```



```
* @param The input is a collection iterator consisting of (key, value) pairs.
* Each map puts together all the pairs with the same key. The reduce method sums the number of the same keys.
* Call context.write(key, value) to write the output to the specified directory.
* Outputformat writes the (key, value) pairs output by reduce to the file system.
* By default, TextOutputFormat is used to write the reduce output to the HDFS.
*/

public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
// Custom implementation
}

/**
* The setup() method is called only once before the map() method of a map task or the reduce() method of a reduce task is called.
*/

public void setup(Context context) throws IOException,
InterruptedException {

// Custom implementation. Context obtains the configuration information.

}
}
```

Submit a MapReduce task.

Use the main() method to create a job, set parameters, and submit the job to the Hadoop cluster.

Example:

```
public static void main(String[] args) throws Exception {
Configuration conf = getConfiguration();
// Input parameters for the main method: args[0] indicates the input path of the MR job. args[1] indicates the output path of the MR job.
String[] otherArgs = new GenericOptionsParser(conf, args)
.getRemainingArgs();
if (otherArgs.length != 2) {
System.err.println("Usage: <in> <out>");
System.exit(2);
}
Job job = new Job(conf, "job name");
// Locate the jar package of the major task.
job.setJar("D:\\job-examples.jar");
// job.setJarByClass(TestWordCount.class);
// Set the map and reduce classes to be executed. You can also specify them in the configuration file.
job.setMapperClass(TokenizerMapperV1.class);
job.setReducerClass(IntSumReducerV1.class);
// Set the combiner class. By default, it is not used. If it is used, it runs the same classes as reduce. Exercise care when using the Combiner class. You can also specify the combiner class in the configuration file.
job.setCombinerClass(IntSumReducerV1.class);
// Set the output type of the job. You can also specify it in the configuration file.
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
// Set the input and output paths for the job. You can also specify them in the configuration file.
Path outputPath = new Path(otherArgs[1]);
FileSystem fs = outputPath.getFileSystem(conf);
// If the output path already exists, delete it.
if (fs.exists(outputPath)) {
fs.delete(outputPath, true);
}
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```


10.2 MapReduce Application Development Suggestions

Globally used configuration items, which are specified in the mapred-site.xml configuration file.

The following provides the configuration items in the mapred-site.xml file corresponding to the interface.

Example:

```
setMapperClass(Class <extends Mapper> cls) ->"mapreduce.job.map.class"
setReducerClass(Class<extends Reducer> cls) ->"mapreduce.job.reduce.class"
setCombinerClass(Class<extends Reducer> cls) ->"mapreduce.job.combine.class"
setInputFormatClass(Class<extends InputFormat> cls) ->"mapreduce.job.inputformat.class"
setJar(String jar) ->"mapreduce.job.jar"
setOutputFormat(Class< extends OutputFormat> theClass) ->"mapred.output.format.class"
setOutputKeyClass(Class<> theClass) ->"mapreduce.job.output.key.class"
setOutputValueClass(Class<> theClass) ->"mapreduce.job.output.value.class"
setPartitionerClass(Class<extends Partitioner> theClass) ->"mapred.partitioner.class"
setMapOutputCompressorClass(Class<extends CompressionCodec> codecClass)
->"mapreduce.map.output.compress"&"mapreduce.map.output.compress.codec"
setJobPriority(JobPriority prio) ->"mapreduce.job.priority"
setQueueName(String queueName) ->"mapreduce.job.queuename"
setNumMapTasks(int n) ->"mapreduce.job.maps"
setNumReduceTasks(int n) ->"mapreduce.job.reduces"
```


11 Spark

11.1 Spark Application Development Rules

Import the Spark class in Spark applications

- Example in Java:

```
//Class imported when SparkContext is created.
import org.apache.spark.api.java.JavaSparkContext
//Class imported for the RDD operation.
import org.apache.spark.api.java.JavaRDD
//Class imported when SparkConf is created.
import org.apache.spark.SparkConf
```

- Example in Scala:

```
//Class imported when SparkContext is created.
import org.apache.spark.SparkContext
//Class imported for the RDD operatin.
import org.apache.spark.SparkContext._
//Class imported when SparkConf is created.
import org.apache.spark.SparkConf
```

Pay attention to the parameter transfer between the Driver and Executor nodes in distributed cluster

When Spark is used for programming, certain code logic needs to be determined based on the parameter entered. Generally, the parameter is specified as a global variable and assigned a null value. The actual value is assigned before the SparkContext object is instantiated using the main function. However, in the distributed cluster mode, the jar package of the executable program will be sent to each Executor. If the global variable values are changed only for the nodes in the main function and are not sent to the functions executing tasks, an error of null pointer will be reported.

Correct:

```
object Test
{
  private var testArg: String = null;
  def main(args: Array[String])
  {
    testArg = ¡;
    val sc: SparkContext = new SparkContext(¡);
```



```
  sc.textFile(¡)
  .map(x => testFun(x, testArg));
 }

 private def testFun(line: String, testArg: String): String =
 {
  testArg.split(¡);
  return ¡;
 }
}
```

Incorrect:

```
//Define an object.
object Test
{
 // Define a global variable and set it to null. Assign a value to this variable before the SparkContext object is instantiated using the main function.
 private var testArg: String = null;
 //main function
 def main(args: Array[String])
 {
  pair
  testArg = ¡;
  val sc: SparkContext = new SparkContext(¡);

  sc.textFile(¡)
  .map(x => testFun(x));
 }

 private def testFun(line: String): String =
 {
  testArg.split(...);
  return ¡;
 }
}
```

No error will be reported in the local mode of Spark. However, in the distributed cluster mode, an error of null pointer will be reported. In the cluster mode, the jar package of the executable program is sent to each Executor for running. When the testFun function is executed, the system queries the value of testArg from the memory. The value of testArg, however, is changed only when the nodes of the main function are started and other nodes are unaware of the change. Therefore, the value returned by the memory is null, which causes an error of null pointer.

SparkContext.stop must be added before an application program stops

When Spark is used in secondary development, SparkContext.stop() must be added before an application program stops.

NOTE

When Java is used in application development, JavaSparkContext.stop() must be added before an application program stops.

When Scala is used in application development, SparkContext.stop() must be added before an application program stops.

The following use Scala as an example to describe correct and incorrect examples.

Correct:

```
//Submit a spark job.
val sc = new SparkContext(conf)
```



```
//Specific task
...

//The application program stops.
sc.stop()
```

Incorrect:

```
//Submit a spark job.
val sc = new SparkContext(conf)

//Specific task
...
```

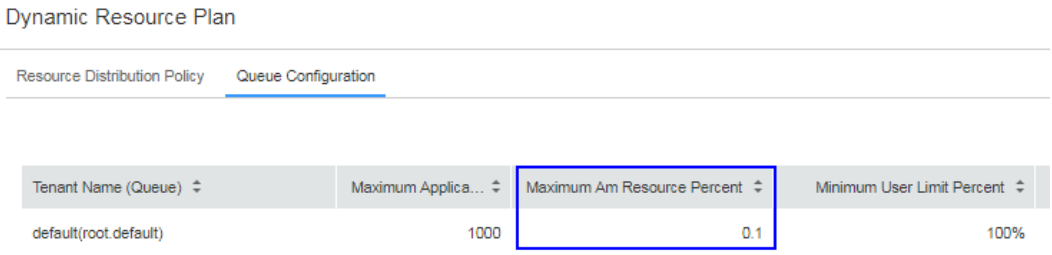
If you do not add SparkContext.stop, the YARN page displays the failure information. In the same task, as shown in **Figure 11-1**, the first program does not add SparkContext.stop(), while the second program adds SparkContext.stop.

Figure 11-1 Difference when SparkContext.stop() is added

Appropriately plan the proportion of resources for AM

When there are many tasks and each task occupies few resources, the tasks may fail to start even if the cluster resources are sufficient and the tasks are submitted successfully. To address this issue, you can increase the value of **Max AM Resource Percent**.

Figure 11-2 Modify Max AM Resource Percent



11.2 Spark Application Development Suggestions

Persist the RDD that will be frequently used

The default RDD storage level is StorageLevel.NONE, which means that the RDD is not stored on disks or in memory. If an RDD is frequently used, persist the RDD as follows:

Call cache(), persist(), or persist(newLevel: StorageLevel) of spark.RDD to persist the RDD. The cache() and persist() functions set the RDD storage level to StorageLevel.MEMORY_ONLY. The persist(newLevel: StorageLevel) function allows you to set other storage level for the RDD. However, before calling this function, ensure that the RDD storage level is StorageLevel.NONE or the same as the

newLevel. That is, once the RDD storage level is set to a value other than StorageLevel.NONE, the storage level cannot be changed.

To unpersist an RDD, call unpersist(blocking: Boolean = true). The function can:

1. Remove the RDD from the persistence list. The corresponding RDD data becomes recyclable.

2. Set the storage level of the RDD to StorageLevel.NONE.

Carefully select the the shuffle operator

This type of operator features wide dependency. That is, a partition of the parent RDD affects multiple partitions of the child RDD. The elements in an RDD are <key, value> pairs. During the execution process, the partitions of the RDD will be sequenced again. This operation is called shuffle.

Network transmission between nodes is involved in the shuffle operators. Therefore, for an RDD with large data volume, you are advised to extract information as much as possible to minimize the size of each piece of data and then call the shuffle operators.

The following methods are often used:

- combineByKey() : RDD[(K, V)] => RDD[(K, C)]

  This method is used to convert all the keys that have the same value in RDD[(K, V)] to a value with type of C.
- groupByKey() and reduceByKey() are two types of implementation of combineByKey. If groupByKey and reduceByKey cannot meet requirements in complex data aggregation, you can use customized aggregation functions as the parameters of combineByKey.
- distinct(): RDD[T] => RDD[T]

  This method is used to remove repeated elements. The code is as follows:

  ```
  map(x => (x, null)).reduceByKey((x, y) => x, numPartitions).map(_._1)
  ```

  This process is time-consuming, especially when the data volume is high. Therefore, it is not recommended for the RDD generated from large files.
- join() : (RDD[(K, V)], RDD[(K, W)]) => RDD[(K, (V, W))]

  This method is used to combine two RDDs through key.

  If a key in RDD[(K, V)] has X values and the same key in RDD[(K, W)] has Y values, a total of (X * Y) data records will be generated in RDD[(K, (V, W))].

Use high-performance operators if the service permits

1. Using reduceByKey/aggregateByKey to replace groupByKey

   The map-side pre-aggregation refers to that each local node performs the aggregation operation on the same key, which is similar to the local combiner in MapReduce. The map-side pre-aggregation ensures that each key on a node is unique. When a node is collecting the data of the same key in the processing results of the previous nodes, data that needs to be obtained will be significantly reduced, decreasing disk I/O and Internet transmission cost. Generally speaking, it is advised to replace groupByKey operator with reduceByKey or aggregateByKey operator if possible because they will pre-aggregate the local same key on each node by using user-defined functions.

However, the groupByKey operator does not support pre-aggregation and delivers lower performance than reduceByKey or aggregateByKey because all data are distributed and transmitted on all the nodes.

2. Using mapPartitions to replace ordinary map operators

   During a function invocation, mapPartitions operators will process all the data in a partition instead of only one piece of data, and therefore delivers higher performance than the ordinary map operators. However, mapPartitions may occasionally result in Out of Memory (OOM). If memory is insufficient, some objects cannot be recycled during memory recycling. Therefore, exercise caution when using mapPartitions.

3. Performing the coalesce operation after filtering

   After filtering a large portion of data (for example, above 30%) by using the filter operator in an RDD, you are advised to manually decrease the number of partitions by using coalesce in order to compress the data in RDD to fewer partitions. This is because after filtering, much data in each partition is filtered out, leaving little data to be processed. If the computing is continued, resources can be wasted. The task handling speed decreases as the number of tasks increases. Therefore, decreasing the number of partitions by using coalesce to compress the RDD data to fewer partitions can ensure that all the partitions are handled with fewer tasks. The performance can also be enhanced in some scenarios.

4. Using repartitionAndSortWithinPartitions to replace repartition and sort

   repartitionAndSortWithinPartitions is recommended by Spark official website. It is advised to use repartitionAndSortWithinPartitions for sorting after repartitioning. This operator can sort and shuffle repartitions at the same time, delivering higher performance.

5. Using foreachPartitions to replace foreach

   Similar to "Using mapPartitions to replace ordinary map operators", this mechanism handles all the data in a partition during a function invocation instead of one piece of data. In practice, foreachPartitions is proved to be helpful in improving performance. For example, the foreach function can be used to write all the data in RDD into MySQL. Ordinary foreach operators, write data piece by piece, and a database connection is established for each function invocation. Frequent connection establishments and destructions cause low performance. foreachPartitions, however, processes all the data in a partition at a time. Only one database connection is required for each partition. Batch insertion delivers higher performance.

RDD Shared Variables

In application development, when a function is transferred to a Spark operation(such as map and reduce) and runs on a remote cluster, the operation is actually performed on the independent copies of all the variables involved in the function. These variables will be copied to each machine. In general, reading and writing shared variables across tasks is apparently inefficient. Spark provides two shared variables that are commonly used: broadcast variable and accumulator.

Kryo can be used to optimize serialization performance in performance-demanding scenarios.

Spark offers two serializers:

org.apache.spark.serializer.KryoSerializer: high-performance but low compatibility

org.apache.spark.serializer.JavaSerializer: average performance and high compatibility

Method: conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

NOTE

The following are reasons why Spark does not use Kryo-based serialization by default:

Spark uses Java serialization by default, that is, uses the ObjectOutputStream and ObjectInputStream API to perform serialization and deserialization. Spark can also use Kryo serialization library, which delivers higher performance than Java serialization library. According to official statistics, Kryo-based serialization is 10 times more efficient than Java-based serialization. Kryo-based serialization requires the registration of all the user-defined types to be serialized, which is a burden for developers.

Suggestions on Optimizing Spark Streaming Performance

1. Set an appropriate batch processing duration (batchDuration).
2. Set concurrent data receiving appropriately.
   - Set multiple receivers to receive data.
   - Set an appropriate receiver congestion duration.
3. Set concurrent data processing appropriately.
4. Use Kryo-based serialization.
5. Optimize memory.
   - Set the persistence level to reduce GC costs.
   - Use concurrent Mark Sweep GC algorithms to shorten GC pauses.

Suggestions for a Running PySpark

To run a PySpark application, you must install the Python environment and upload the necessary Python dependency package to the HDFS. The Python environment provided by the cluster cannot be used.

