MapReduce Service

# Component Development Specifications

**Issue** 01

**Date** 2024-05-11

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 Security Authentication

## 1.1 Rules

### Only one account is used in each process and a process needs explicit authentication only once.

This rule must be met in the following scenarios (pay attention to this in the design phase):

1. A process can access multiple clusters at the same time (Each cluster has independent KrbServer and LdapServer.)
2. All applications running in a container (for example, Tomcat) belong to the same process.

## 1.2 Suggestions

### Account Management Principles

1. Service application should apply for new accounts instead of using original system accounts.
2. The new accounts should meet the principle of least privilege.

# 2 ClickHouse

## 2.1 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.

# 2.2 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.

# 3 Doris

## 3.1 Table Creation Rules

This section describes the rules and suggestions for creating a Doris table.

### Doris Table Creation Rules

- When creating a Doris table and specifying bucket buckets, ensure that the data size of each bucket ranges from 100 MB to 3 GB and the maximum number of buckets in a single partition does not exceed 5000.
- If the number of data records in a table exceeds 500 million, you must set a bucket policy.
- Do not set too many bucket columns in a table. Generally, you only need to set one or two columns. In addition, you need to ensure even data distribution and balanced query throughput.
  - Data is evenly distributed to prevent data skew in some buckets from affecting data balancing and query efficiency.
  - The query throughput uses the bucket tailoring optimization of query SQL statements to avoid full bucket scanning and improve query performance.
  - Bucket column selection: Columns with even data and commonly used as query conditions are preferentially used as bucket columns.

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

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

    After the command is executed, check whether the difference between the number of data records in each group is small. If the difference exceeds 2/3 or 1/2, select another bucket field.
- Do not use dynamic partitions for less than 20 million data records. Dynamic partitioning automatically creates partitions, but users cannot pay attention to small tables. As a result, a large number of unused partitions are created to distinguish buckets.
- When creating a table, ensure that there are three to five sort keys. If there are too many sort keys, data writing will be slow and data import performance will be affected.

- If Auto Bucket is not used, buckets need to be divided based on the existing data volume to improve the import and query performance. Auto Bucket causes a large number of tablets. As a result, a large number of small files exist.
- The number of copies for creating a table must be at least 2. The default value is 3. Do not use a single backup.

## Doris Table Creation Suggestions

- The number of materialized views in a single table cannot exceed six. It is not recommended that materialized views be nested. You are not advised to use materialized views to perform ETL tasks such as heavy aggregation and join calculation during data writing.
- If there is a large amount of historical partition data but the historical data is small, the data is unbalanced, or the data query probability is low, you can create historical partitions (such as yearly and monthly partitions) and store all historical data in the corresponding partitions.

  The method of creating a history partition is FROM ("2000-01-01") TO ("2022-01-01") 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 bucket field, 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* ...
- During table creation, the first field must be the column that is most frequently queried. By default, the prefix index quick query capability is provided. The column that is most frequently queried and has a high cardinality is selected as the prefix index, by default, the first 36 bytes of a row are used as the prefix index of the row. (A column of the varchar type can match only 20 bytes, and the prefix index will be truncated if less than 36 bytes are matched.)
- For more than 100 million data records, if fuzzy match or equivalent/in conditions are used, you can use inverted indexes (supported since Doris 2.x) or Bloomfilter. For orthogonal queries with low cardinality columns, bitmap indexes are recommended. (The cardinality of bitmap indexes ranges from 10000 to 100000.)
- When creating a table, you need to plan the number of fields to be used in the future. You can reserve dozens of fields of the integer or character type. If fields are insufficient in the future, you need to add fields temporarily at a high cost.

# 3.2 Data Change

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

## Doris Data Change Rules

- Applications cannot directly use the **delete** or **update** statement to change data. You can use the **upsert** mode of the CDC to change data.

- You are not advised to frequently add or delete fields in tables during peak hours. You are advised to reserve fields to be used in the early stage of table creation. If fields must be added or deleted, or field types and comments must be modified, stop writing and modifying related tables during off-peak hours, and then re-create the tables.

  a. Create a table. The structure of the table is the same as that of the table whose fields need to be added, deleted, or modified. Add new fields to the new table, delete unnecessary fields, or modify fields whose types need to be changed.

  b. Select specified fields and insert them to the newly created table.

  **INSERT INTO** *Newly created table* **SELECT** *Specified column* **FROM** *Existing table whose column needs to be modified*;

  NOTE

  If the table contains a large amount of data, you can import the data to the new table in batches by time to reduce the instantaneous high CPU or MEM memory usage and affect the query service. 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. Therefore, you need to set the query timeout parameter query_timeout at the SQL or user level.

### Doris Data Change Suggestions

When performing special large SQL operations, you can use a method similar to **SELECT /*+ SET_VAR(query_timeout = xxx*/ from table** to set session variables in hint mode. Do not set global system variables.

## 3.3 Naming Conventions

This section describes the rules and suggestions for naming a database or table when you create a Doris database or table.

### Doris Naming Rules

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

### Doris Naming Suggestions

- The database name is in lowercase and separated by underscores (_). The length of the database name is less than 62 bytes.
- The name of the Doris table is case sensitive. The name is in lowercase and separated by underscores (_). The length of the name is less than 64 bytes.

# 3.4 Data Query

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

## Doris data query rules

- In the data query service code, you are advised to retry the query when the query fails and issue the query again.
- If the enumerated value of the constant **in** exceeds 1000, the constant must be changed to a subquery.
- Do not use REST API (Statement Execution Action) to execute a large number of SQL queries. This interface is used only for cluster maintenance.
- If the number of query results exceeds 50,000, use JDBC Catalog or OUTFILE to export the query data. Otherwise, a large amount of data on the FE will occupy FE resources, affecting cluster stability.
  - For interactive query, you are advised to export data in pagination mode (offset limit). The pagination command is Order by.
  - If data is exported for a third party, the outfile or export mode is recommended.
- Colocation Join is used for JOIN of more than two tables with more than 300 million records.
- Do not use select * to query data in hundreds of millions of large tables. Specify the fields to be queried during query.
  - Use the SQL Block mode to forbid the select * operation.
  - For high-concurrency point queries, you are advised to enable row-based storage (supported by Doris 2.x) and use PreparedStatement for queries.
- Bucket conditions must be set for querying hundreds of millions of tables.
- Do not perform full-partition data scanning on a partitioned table.

## Doris Data Query Suggestions

- If the number of **insert into select** statements exceeds 100 million, you are advised to split the statements into multiple **insert into select** statements and execute them in multiple batches.
- Do not use OR as a JOIN condition.
- You are not advised to frequently delete and modify data. You can save the data to be deleted in batches and delete the data in batches occasionally. In addition, you need to specify conditions to improve system stability and deletion efficiency.
- Some data is returned after a large amount of data (more than 500 million) is sorted. You are advised to reduce the data range before performing the sorting. Otherwise, the performance will be affected if a large amount of data is sorted. The following is an example.

  Instead of **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 query task performance optimization parameters:

  This parameter is set at the session level and indicates the number of fragments that can be concurrently executed. This parameter consumes a large number of CPU resources. Therefore, you do not need to set this parameter. If you need to set this parameter to accelerate query performance, comply with the following rules:

  - Do not set this parameter to take effect globally. Do not use the **set global** command to set this parameter.
  - You are advised to 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.
  - When setting this parameter, you need to observe the CPU usage. You can set this parameter only when the CPU usage is less than 50%.
  - If the query SQL statement is **insert into select** with a large amount of data, you are advised not to set this parameter.

# 3.5 Data Import

This section describes the specifications for importing Doris data.

## Doris Data Import Suggestions

- Do not frequently perform the **update**, **delete**, or **truncate** operation. You are advised to perform the operation every several minutes. To use the **delete** operation, you must set the partition or primary key column condition.
- Do not use **INSERT INTO** *tbl1* **VALUES("1"),("a");** to import data. If a small amount of data needs to be written, use StreamLoad, BrokerLoad, SparkLoad, or Flink Connector provided by Doris.
- When Flink writes data to Doris in real time, the time set for CheckPoint must consider 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.
- You are advised not to use insert values as the main data write mode. StreamLoad, BrokerLoad, or SparkLoad is recommended for batch data import.
- When data is imported in INSERT INTO WITH LABEL XXX SELECT mode, if downstream dependency or query exists, you need to check whether the imported data is visible.

  Run the show load where label='xxx' SQL command to check whether the current INSERT task status 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 importing data of less than 100 GB. If the data volume is too large, SparkLoad can be used.
- Do not use Routine Load of Doris to import data. You are advised to use Flink to query Kafka data and then write the data to Doris. This facilitates the control of the amount of data to be imported in a single batch and prevents a

large number of small files from being generated. If Routine Load has been used to import data, set max_tolerable_backend_down_num to 1 on the FE before rectification to improve data import reliability.

- You are advised to import data in batches at a low frequency. The average interval for importing a single table must be greater than 30s. The recommended interval is 60s. 1000 to 100000 rows of data are imported at a time.

# 3.6 UDF Development

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

## Doris UDF Development Rules

- The method invocation in the UDF must be thread-safe.
- Do not read external large files to the memory in the UDF implementation. If the file size is too large, the memory may 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 may be used up.
- The Java UDF should capture and process possible exceptions. Do not send exceptions to services for processing to avoid unknown exceptions in programs. You can 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 package in the class does not conflict with the package on the server. You can run the **grep -lr** *"Full restriction class name"* command to check the JAR packages that conflict with each other. If a class name conflict occurs, you can fully restrict the class name to avoid the conflict.

## Doris UDF Development Suggestions

- Do not copy a large amount of data to prevent stack memory overflow.
- 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. You are advised to use the camel-case naming method and end it with UDF, for example, MyFunctionUDF.
- The 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 by default or when an exception occurs. You are advised to use basic data types or Java classes as return value types.

# 3.7 Connection and Running

Comply with the following specifications when connecting to the Doris and running Doris tasks:

- You are advised to use the ELB to connect to the Doris to ensure that services can be provided when the connected FE is faulty.
- When a single Doris instance or hardware fault occurs, newly submitted tasks can be successfully executed, but running tasks cannot be successfully executed when the fault occurs. Therefore, you need to retry a task when connecting to the Doris to execute the task. When the task fails due to unknown reasons, you can ensure that the newly submitted task can be successfully executed.

# 4 Flink

## 4.1 Applicable Scenarios

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

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

## 4.2 Rules

### Delete Residual Directories If a Flink Task Stops Unexpectedly

After a FlinkServer instance is installed, the residual Flink directories are automatically deleted.

By default, only residual directories in the **/flink_base** directory of ZooKeeper and those in the **/flink/recovery** directory of HDFS are deleted.

## 4.3 Suggestions

### FlinkServer Usage

You are advised to submit Flink jobs using FlinkServer. FlinkServer supports the submission of Flink SQL jobs and Flink Jar jobs. .

# 5 HBase

## 5.1 Application Scenarios

Hadoop database (HBase) is a reliable, high-performance, column-oriented and scalable distributed storage system. Different from traditional relational databases, HBase is suitable for massive data process.

The HBase is applicable to the following scenarios:

- Massive data processing (higher than the TB or PB level).
- High-throughput demanding scenarios
- Scenarios that require efficient random read of massive data.
- Good-scalability demanding scenarios
- Concurrent processing of structured and unstructured data.
- Scenarios that do not require the Atomicity, Consistency, Isolation, Durability (ACID) feature provided by traditional relational databases.

HBase tables have the following features:

- Large: Each table contains a hundred million rows and one million columns.
- Column-oriented: Storage and rights control is implemented based on columns (families), and columns (families) are independently retrieved.
- Sparse: Null columns do not occupy storage space.

## 5.2 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.

## 5.3 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.

## 5.4 Examples

### Set Configuration parameters

To set up a connection between an HBase Client and the HBase Server, set the following parameters:

- hbase.zookeeper.quorum: IP address of Zookeeper. If there are multiple Zookeeper nodes, separate multiple IP addresses by a comma (,).
- hbase.zookeeper.property.clientPort: Port of Zookeeper.

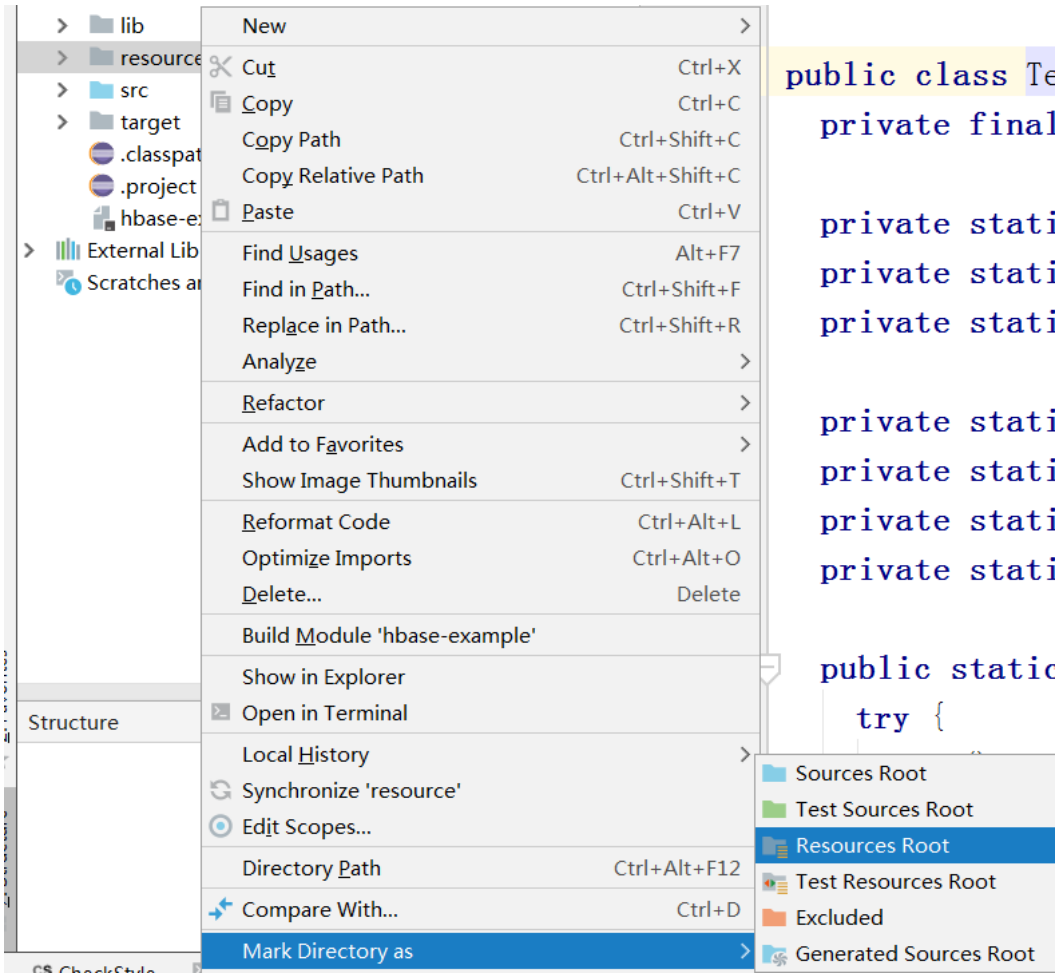
NOTE

The Configuration instance created by using HBaseConfiguration.create() will be automatically loaded with the configuration items in the following files:

- core-default.xml
- core-site.xml
- hbase-default.xml
- hbase-site.xml

Save these configuration files in **Source Folder**. To create a **Source Folder**, create a **resource** folder in the project, right-click the folder, and choose **Mark Directory as** > **Resources Root**.




The following table describes the parameters to be configured on the client.

NOTE

Do not change the values of these parameters.

| Parameter | Description |
| --- | --- |
| hbase.client.pause | Specifies the time to wait before a retry is performed when an exception occurs. The actual time is calculated based on this value and the number of retires. |
| hbase.client.retries.number | Specifies the number of retries to be performed when an exception occurs. |
| hbase.client.retries.longer.multiplier | This parameter is related to the number of retires. |
| hbase.client.rpc.maxattempts | Specifies the number of retires when the RPC request is not successfully sent. |
| hbase.regionserver.lease.period | This parameter (in ms) is related to the Scanner timeout period. |

| Parameter | Description |
| --- | --- |
| hbase.client.write.buffer | This parameter is invalid if AutoFlush is enabled. If AutoFlush is disabled, the HBase Client caches the data to be written. When the size of the data cached reaches the specified limit, the HBase Client initiates a write operation to the HBase cluster. |
| hbase.client.scanner.caching | Specifies the number of rows allowed for a next request during a Scan. |
| hbase.client.keyvalue.maxsize | Specifies the maximum value of a keyvalue. |
| hbase.htable.threads.max | Specifies the maximum number of threads related to data operations in an HTable instance. |
| hbase.client.prefetch.limit | Before reading or writing data, the client must obtain the address of the Region. Therefore, a client can have some Region addresses pre-cached. This parameter is related to the configuration of the number of Region addresses pre-cached. |

**Example:**

```
hbaseConfig = HBaseConfiguration.create();
//You do not need to set the following parameters if they are specified in the configuration files.
hbaseConfig.set("hbase.zookeeper.quorum", "172.16.100.1,172.16.100.2,172.16.100.3");
hbaseConfig.set("hbase.zookeeper.property.clientPort", "2181");
```

## Use HTablePool in multi-thread write operations

Use HTablePool for multiple data write threads. Observe the following when using HTablePool to perform multi-thread write operations:

1. Enable multiple date write threads to share the same HTablePool instance.
2. Specify maxSize of the HTableInterface instance when instantiating HTablePool. That is, instantiate the class using the following constructor function:

   public HTablePool(final Configuration config, final int maxSize)

   The value of maxSize can be determined based on Threads (the number of data write threads) and Tables (the number of user tables). Generally, maxSize cannot be greater than the product of Threads and Tables. (maxSize <= Threads x Tables)
3. The client thread obtains an HTableInterface instance with the table name of tableName using HTablePool#getTable(tableName).
4. An HTableInterface instance can be used by only one thread at a time.
5. If HTableInterface is not used, call HTablePool#putTable(HTableInterface table) to release it.

**Example:**

```
/**
* A certain number of retries is required after a data write failure. The time to wait before each retry is
```

```
determined based on the number of retries performed.
*/
private static final int[] RETRIES_WAITTIME = {1, 1, 1, 2, 2, 4, 4, 8, 16, 32};
/**
* Specify the number of retries.
*/
private static final int RETRIES = 10;
/**
* The unit of the time to wait after a failure.
*/
private static final int PAUSE_UNIT = 1000;
private static Configuration hadoopConfig;
private static HTablePool tablePool;
private static String[] tables;
/**
* <Initialize HTablePool>
* <Function description>
* @param config
* @see [class, class#method, class#member]
*/
public static void initTablePool()
{
DemoConfig config = DemoConfig.getInstance();
if (hadoopConfig == null)
{
hadoopConfig = HBaseConfiguration.create();
hadoopConfig.set("hbase.zookeeper.quorum", config.getZookeepers());
hadoopConfig.set("hbase.zookeeper.property.clientPort", config.getZookeeperPort());
}
if (tablePool == null)
{
tablePool = new HTablePool(hadoopConfig, config.getTablePoolMaxSize());
tables = config.getTables().split(",");
}
}
public void run()
{
// Initialize HTablePool. Initialize this instance only once because it is shared by multiple threads.
initTablePool();
for (;;)
{
Map<String, Object> data = DataStorage.takeList();
String tableName = tables[(Integer)data.get("table")];
List<Put> list = (List)data.get("list");
// Use Row as the Key and save all puts in the List. This set is used only for querying the data failed to be
written when a write operation fails, because the Server only returns the Row of the data failed.
Map<byte[], Put> rowPutMap = null;
// Perform the operation again if it fails (even if some of the data failed to be written). Only the data failed
to be written is submitted each time.
INNER_LOOP :
for (int retry = 0; retry < RETRIES; retry++)
{
// Obtain an HTableInterface instance from HTablePool. Release the instance if it is not required.
HTableInterface table = tablePool.getTable(tableName);
try
{
table.put(list);
// The operation is successful.
break INNER_LOOP;
}
catch (IOException e)
{
// If the exception type is RetriesExhaustedWithDetailsException, some of the data failed to be written. The
exception occurs because the processes in the HBase cluster are abnormal or a large number of Regions are
being migrated.
// If the exception type is not RetriesExhaustedWithDetailsException, insert all the data in the list again.
if (e instanceof RetriesExhaustedWithDetailsException)
{
RetriesExhaustedWithDetailsException ree =
```

```
(RetriesExhaustedWithDetailsException)e;
int failures = ree.getNumExceptions();
System.out.println("In this operation, [" + failures + "] data records failed to be inserted.");
// Instantiate the Map when a retry is performed upon the first failure.
if (rowPutMap == null)
{
rowPutMap = new HashMap<byte[], Put>(failures);
for (int m = 0; m < list.size(); m++)
{
Put put = list.get(m);
rowPutMap.put(put.getRow(), put);
}
}
//Clear the original data and then add the data failed to be written.
list.clear();
for (int m = 0; m < failures; m++)
{
list.add(rowPutMap.get(ree.getRow(m)));
}
}
}
finally
{
// Release the instance after using it.
tablePool.putTable(table);
}
// If an exception occurs, wait some time after releasing the HTableInterface instance.
try
{
sleep(getWaitTime(retry));
}
catch (InterruptedException e1)
{
System.out.println("Interruped");
}
}
}
}
```

## Create a Put instance

HBase is a column-oriented database. One column of data may correspond to multiple column families, and one column family may correspond to multiple columns. Before data is written, the column (column family name and column name) must be specified.

| | ColumnFamily01 | | | | | ColumnFamily02 | | | |
|---|---|---|---|---|---|---|---|---|---|
| | column01 | column02 | column03 | column04 | column05 | column01 | column02 | column03 | column04 |
| Row--01 | | | | | | | | | |
| Row--02 | | | | | | | | | |
| Row--03 | | | | | | | | | |
| Row--04 | | | | | | | | | |
| Row--05 | | | | | | | | | |
| Row--06 | | | | | | | | | |
| Row--07 | | | | | | | | | |
| Row--08 | | | | | | | | | |

A Put instance must be created before a row of data is written in an HBase table. The Put instance data consists of (Key, Value). The Value can contain multiple columns of values.

When a (Key, Value) record is added to a Put instance, the family, qualifier, and value added are byte sets. Use the Bytes.toBytes method to convert character strings to byte sets. Do not use the String.toBytes method, because this method

cannot ensure correct data coding. Errors occur when the Key or Value contains Chinese characters.

**Example:**

```
//The column family name is privateInfo.
private final static byte[] FAMILY_PRIVATE = Bytes.toBytes("privateInfo");
//The privateInfo column family has two columns: "name" and "address".
private final static byte[] COLUMN_NAME = Bytes.toBytes("name");
private final static byte[] COLUMN_ADDR = Bytes.toBytes("address");
/**
* <Create a Put instance. >
* <A put instance with one column family and two columns of data is created. >
* @param rowKey Key key value
* @param name name
* @param address address
* @return
* @see [class, class#method, class#member]
*/
public Put createPut(String rowKey, String name, String address)
{
Put put = new Put(Bytes.toBytes(rowKey));
put.add(FAMILY_PRIVATE, COLUMN_NAME, Bytes.toBytes(name));
        put.add(FAMILY_PRIVATE, COLUMN_ADDR, Bytes.toBytes(address));
return put;
}
```

## Create an HBaseAdmin instance

**Example:**

```
private Configuration demoConf = null;
private HBaseAdmin hbaseAdmin = null;
/**
* <Constructor function>
* Import the Configuration instances.
*/
public HBaseAdminDemo(Configuration conf)
{
this.demoConf = conf;
try
{
// Instantiate HBaseAdmin
hbaseAdmin = new HBaseAdmin(this.demoConf);
}
catch (MasterNotRunningException e)
{
e.printStackTrace();
}
catch (ZooKeeperConnectionException e)
{
e.printStackTrace();
}
}
/**
* <Examples of method using>
* <For details about more methods, see the HBase interface documents. >
* @throws IOException
* @throws ZooKeeperConnectionException
* @throws MasterNotRunningException
* @see [Class, class#method, class#member]
*/
public void demo() throws MasterNotRunningException, ZooKeeperConnectionException, IOException
{
byte[] regionName = Bytes.toBytes("mrtest,jjj,1315449869513.fc41d70b84e9f6e91f9f01affdb06703.");
byte[] encodeName = Bytes.toBytes("fc41d70b84e9f6e91f9f01affdb06703");
// Reallocate a Region.
hbaseAdmin.unassign(regionName, false);
```

```
// Actively initiate Balance.
hbaseAdmin.balancer();
// Move a Region. The second parameter is HostName+StartCode of RegionServer, for example,
// host187.example.com,60020,1289493121758. If this parameter is set to null, the Region will be moved at
random.
hbaseAdmin.move(encodeName, null);
// Check whether a table exists.
hbaseAdmin.tableExists("tableName");
// Check whether a table is activated.
hbaseAdmin.isTableEnabled("tableName");
}
/**
* <Method used to rapidly create a table >
* <Create an HTableDescriptor instance, which contains description of the HTable to be creased. Create the
column families, which are associated with the HColumnDescriptor instance. In this example, the column
family name is "columnName">.
* @param tableName table name
* @return
* @see [Class, class#method, class#member]
*/
public boolean createTable(String tableName)
{
try {
if (hbaseAdmin.tableExists(tableName)) {
return false;
}
HTableDescriptor tableDesc = new HTableDescriptor(tableName);
HColumnDescriptor fieldADesc = new HColumnDescriptor("columnName".getBytes());
fieldADesc.setBlocksize(640 * 1024);
tableDesc.addFamily(fieldADesc);
hbaseAdmin.createTable(tableDesc);
} catch (Exception e) {
e.printStackTrace();
return false;
}
return true;
}
```

# 5.5 Appendix

## Parameters Batch and Caching for Scan

Batch: specifies the maximum number of data records returned each time when scan calls the next interface. It is related to the number of **columns** read each time.

Caching: specifies the maximum number of next values returned for an RPC request. It is related to the number of **rows** obtained by each RPC.

The following examples explain the functions of these two parameters in Scan:

A Region contains two rows (rowkey) of data in table A. Each row has 1000 columns, and each column has only one version, that is, each row has 1000 key values.

| - | Colu A1 | Colu A2 | Colu A3 | Colu A4 | … | Colu N1 | Colu N2 | Colu N3 | Colu N4 |
|---|---|---|---|---|---|---|---|---|---|
| Row1 | - | - | - | - | … | - | - | - | - |
| Row2 | - | - | - | - | … | - | - | - | - |

- **Example 1:** If Batch is not specified and Caching is 2,

  2000 (Key, Value) records will be returned for each RPC request.
- **Example 2:** If Batch is set to 500 and Caching is 2,

  1000 (Key, Value) records will be returned for each RPC request.
- **Example 3:** If Batch is set to 300 and Caching is 4,

  1000 (Key, Value) records will be returned for each RPC request.

**Further explanation of Batch and Caching**

- Each Caching indicates a chance of data request.
- The value of Batch determines whether a row of data can be read in a Caching. If the value of Batch is smaller than the total columns in a row, this row of data can be read in at least two Caching operations (the next Caching starts from the data where the previous caching stops).
- Each Caching cannot cross rows. That is, if the value of Batch is not reached after a row of data is read, data of the next row will not be read.

This can further explain the results of the previous examples.

- **Example 1:**

  Since Batch is not set, all columns of that row will be read by default. As Caching is 2, 2000 (Key, Value) records will be returned for each RPC request.
- **Example 2:**

  Because Batch is 500 and Caching is 2, a maximum of 500 columns of data will be read in each Caching. Therefore, 1000 (Key, Value) records will be returned after two times of caching.
- **Example 3:**

  Because Batch is 300 and Caching is 4, four times of caching are required to read 1000 data records. Therefore, only 1000 (Key, Value) records will be returned.

**Code example:**

```
Scan s = new Scan();
//Set the start and end keys for data query.
s.setStartRow(Bytes.toBytes("01001686138100001"));
s.setStopRow(Bytes.toBytes("01001686138100002"));
s.setBatch(1000);
s.setCaching(100);
ResultScanner scanner = null;
try {
scanner = tb.getScanner(s);
for (Result rr = scanner.next(); rr != null; rr = scanner.next()) {
for (KeyValue kv : rr.raw()) {
//Display the query results.
System.out.println("key:" + Bytes.toString(kv.getRow())
+ "getQualifier:" + Bytes.toString(kv.getQualifier())
+ "value" + Bytes.toString(kv.getValue()));
}
}
} catch (IOException e) {
System.out.println("error!" + e.toString());
} finally {
scanner.close();
}
```

# 6 HDFS

## 6.1 Application Scenarios

Hadoop distribute file system (HDFS) runs on commodity hardware. It provides high error tolerance. In addition, it supports high data access throughput and is suitable for applications that involve large-scale data sets.

The HDFS is applicable to the following scenarios:

- Massive data processing (higher than the TB or PB level).
- High-throughput demanding scenarios.
- High-reliability demanding scenarios.
- Good-scalability demanding scenarios.

The HDFS is not applicable to the scenarios that involve a large number of small files, random write, and low-latency read.

## 6.2 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.

# 6.3 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.

# 7 Hive

## 7.1 Application Scenarios

Hive is an open-source data warehouse framework built on Hadoop. It provides storage of structured data and basic data analysis services using the Hive query language (HQL), a language like the structured query language (SQL). Hive converts HQL statements to MapReduce or Spark tasks to query and analyze massive data stored in Hadoop clusters.

Hive provides the following features:

- Extracts, transforms, and loads (ETL) data using HQL.
- Analyzes massive structured data using HQL.
- Supports multiple data storage formats, including JavaScript object notation (JSON), comma separated values (CSV), TextFile, RCFile, ORCFILE, and SequenceFile.
- Multiple client connection modes. JDBC interfaces are supported.

Hive is applicable to offline massive data analysis (such as log and cluster status analysis), large scale data mining (such as user behavior analysis, interest region analysis, and region display), and other scenarios.

To ensure Hive high availability (HA), user data security, and service access security, Huawei MRS incorporates the following features based on Hive 3.1.0:

- Kerberos security authentication.
- Data file encryption.
- Comprehensive rights management.

## 7.2 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.

## 7.3 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.

# 7.4 Examples

## JDBC Secondary Development Code Example 1

The following code example provides the following functions:

1. Provides the username and key file path in the JDBC URL address so that programs can automatically perform security logins and create Hive connections.
2. Runs HQL statements for creating, querying, and deleting tables.

```
package com.huawei.bigdata.hive.example;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Properties;


import org.apache.hadoop.conf.Configuration;
import com.huawei.bigdata.security.LoginUtil;

public class JDBCExample {
  private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";

  private static final String ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME = "Client";
  private static final String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";
  private static final String ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = "zookeeper/hadoop.hadoop.com";

  private static Configuration CONF = null;
  private static String KRB5_FILE = null;
  private static String USER_NAME = null;
  private static String USER_KEYTAB_FILE = null;

  private static String zkQuorum = null;//IP address and port list of a ZooKeeper node
  private static String auth = null;
  private static String sasl_qop = null;
  private static String zooKeeperNamespace = null;
  private static String serviceDiscoveryMode = null;
  private static String principal = null;
  private static String auditAddition = null;
  private static void init() throws IOException{
    CONF = new Configuration();

    Properties clientInfo = null;
    String userdir = System.getProperty("user.dir") + File.separator
        + "conf" + File.separator;
```

```
    InputStream fileInputStream = null;
    try{
      clientInfo = new Properties();
      //"hiveclient.properties" is the client configuration file. If the multi-instance feature is used, you need to
replace the file with "hiveclient.properties" of the corresponding instance client.
      //"hiveclient.properties" file is stored in the config directory of the decompressed installation package of
the corresponding instance client.
      String hiveclientProp = userdir + "hiveclient.properties" ;
      File propertiesFile = new File(hiveclientProp);
      fileInputStream = new FileInputStream(propertiesFile);
      clientInfo.load(fileInputStream);
    }catch (Exception e) {
      throw new IOException(e);
    }finally{
      if(fileInputStream != null){
        fileInputStream.close();
        fileInputStream = null;
      }
    }
    //The format of zkQuorum is "xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181";
    //"xxx.xxx.xxx.xxx" of zkQuorum indicates the IP address of the node where ZooKeeper locates. The
default port is 2181.
    zkQuorum =  clientInfo.getProperty("zk.quorum");
    auth = clientInfo.getProperty("auth");
    sasl_qop = clientInfo.getProperty("sasl.qop");
    zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
    principal = clientInfo.getProperty("principal");
    auditAddition = clientInfo.getProperty("auditAddition");
    // Set a user name for the newly created user, where xxx indicates the username created previously. For
example, if the created user is user, USER_NAME is user.
    USER_NAME = "xxx";

    if ("KERBEROS".equalsIgnoreCase(auth)) {
      // Set the keytab and krb5 file path on the client.
      USER_KEYTAB_FILE = userdir + "user.keytab";
      KRB5_FILE = userdir + "krb5.conf";
      System.setProperty("java.security.krb5.conf", KRB5_FILE);
      System.setProperty(ZOOKEEPER_SERVER_PRINCIPAL_KEY, ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);
    }
  }

  /**
   * This example shows how to use the Hive JDBC interface to run the HQL command <br>.
   * <br>
   *
   * @throws ClassNotFoundException
   * @throws IllegalAccessException
   * @throws InstantiationException
   * @throws SQLException
   * @throws IOException
   */
  public static void main(String[] args) throws InstantiationException,
      IllegalAccessException, ClassNotFoundException, SQLException, IOException{
    // Parameter Initialization
    init();

    // Define HQL. HQL must be a single statement and cannot contain ";".
    String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
        "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};

    // Build JDBC URL
    StringBuilder sBuilder = new StringBuilder(
        "jdbc:hive2://").append(zkQuorum).append("/");

    if ("KERBEROS".equalsIgnoreCase(auth)) {
      sBuilder.append(";serviceDiscoveryMode=")
              .append(serviceDiscoveryMode)
              .append(";zooKeeperNamespace=")
```

```
            .append(zooKeeperNamespace)
            .append(";sasl.qop=")
            .append(sasl_qop)
            .append(";auth=")
            .append(auth)
            .append(";principal=")
            .append(principal)
            .append(";user.principal=")
            .append(USER_NAME)
            .append(";user.keytab=")
            .append(USER_KEYTAB_FILE);
  } else {
 //Normal mode
    sBuilder.append(";serviceDiscoveryMode=")
            .append(serviceDiscoveryMode)
            .append(";zooKeeperNamespace=")
            .append(zooKeeperNamespace)
            .append(";auth=none");
  }
  if (auditAddition != null && !auditAddition.isEmpty()) {
     strBuilder.append(";auditAddition=").append(auditAddition);
  }
  String url = sBuilder.toString();

  // Load the Hive JDBC driver.
  Class.forName(HIVE_DRIVER);

  Connection connection = null;
  try {
    // Obtain the JDBC connection.
    // If the normal mode is used, the second parameter needs to be set to a correct username. Otherwise,
the anonymous user will be used for login.
    connection = DriverManager.getConnection(url, "", "");

    // Create a table.
    // If data needs to be imported to the table, use the load statement to import data to the table, for
example, import data from the HDFS to the table.
    // load data inpath '/tmp/employees.txt' overwrite into table employees_info;
    execDDL(connection,sqls[0]);
    System.out.println("Create table success!");

    // Query the table.
    execDML(connection,sqls[1]);

    // Delete the table
    execDDL(connection,sqls[2]);
    System.out.println("Delete table success!");
  }catch (Exception e) {
    System.out.println("Create connection failed : "  + e.getMessage());
  }
  finally {
    // Close the JDBC connection.
    if (null != connection) {
      connection.close();
    }
  }
 }

 public static void execDDL(Connection connection, String sql)
 throws SQLException {
  PreparedStatement statement = null;
  try {
    statement = connection.prepareStatement(sql);
    statement.execute();
  }
  finally {
    if (null != statement) {
      statement.close();
    }
```

```
  }
 }


 public static void execDML(Connection connection, String sql) throws SQLException {
   PreparedStatement statement = null;
   ResultSet resultSet = null;
   ResultSetMetaData resultMetaData = null;

   try {
     // Execute HQL.
     statement = connection.prepareStatement(sql);
     resultSet = statement.executeQuery();

     // Export the queried column names to the console.
     resultMetaData = resultSet.getMetaData();
     int columnCount = resultMetaData.getColumnCount();
     for (int i = 1; i <= columnCount; i++) {
       System.out.print(resultMetaData.getColumnLabel(i) + '\t');
     }
     System.out.println();

     // Export the query results to the console.
     while (resultSet.next()) {
       for (int i = 1; i <= columnCount; i++) {
         System.out.print(resultSet.getString(i) + '\t');
       }
       System.out.println();
     }
   }
   finally {
     if (null != resultSet) {
       resultSet.close();
     }

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

}
```

## JDBC Secondary Development Code Example 2

The following code example provides the following functions:

1. Does not provide the username and key file path in the JDBC URL address to create Hive connections. Users perform security logins by themselves.
2. Runs HQL statements for creating, querying, and deleting tables.

**NOTE**

When accessing ZooKeeper, programs need to use the jaas configuration file, for example, **user.hive.jaas.conf**. The details are as follows:

```
Client {
 com.sun.security.auth.module.Krb5LoginModule required
 useKeyTab=true
 keyTab="D:\\workspace\\jdbc-examples\\conf\\user.keytab"
 principal="xxx@HADOOP.COM"
 useTicketCache=false
 storeKey=true
 debug=true;
 };
```

You need to modify the keyTab path (absolute path) and principal in the configuration file based on the actual environment, and set environment variable **java.security.auth.login.config** to the file path.

```
package com.huawei.bigdata.hive.example;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Properties;


import org.apache.hadoop.conf.Configuration;
import com.huawei.bigdata.security.LoginUtil;

public class JDBCExamplePreLogin {
  private static final String HIVE_DRIVER = "org.apache.hive.jdbc.HiveDriver";

  private static final String ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME = "Client";
  private static final String ZOOKEEPER_SERVER_PRINCIPAL_KEY = "zookeeper.server.principal";
  private static final String ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL = "zookeeper/hadoop";

  private static Configuration CONF = null;
  private static String KRB5_FILE = null;
  private static String USER_NAME = null;
  private static String USER_KEYTAB_FILE = null;

  private static String zkQuorum = null;//IP address and port list of a ZooKeeper node
  private static String auth = null;
  private static String sasl_qop = null;
  private static String zooKeeperNamespace = null;
  private static String serviceDiscoveryMode = null;
  private static String principal = null;
  private static String auditAddition = null;
  private static void init() throws IOException{
    CONF = new Configuration();

    Properties clientInfo = null;
    String userdir = System.getProperty("user.dir") + File.separator
        + "conf" + File.separator;
    InputStream fileInputStream = null;
    try{
      clientInfo = new Properties();
      //"hiveclient.properties" is the client configuration file. If the multi-instance feature is used, you
need to replace the file with "hiveclient.properties" of the corresponding instance client.
      //"hiveclient.properties" file is stored in the config directory of the decompressed installation
package of the corresponding instance client.
      String hiveclientProp = userdir + "hiveclient.properties" ;
```

```
      File propertiesFile = new File(hiveclientProp);
      fileInputStream = new FileInputStream(propertiesFile);
      clientInfo.load(fileInputStream);
    }catch (Exception e) {
      throw new IOException(e);
    }finally{
      if(fileInputStream != null){
        fileInputStream.close();
        fileInputStream = null;
      }
    }
    //The format of zkQuorum is "xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181";
    //"xxx.xxx.xxx.xxx" of zkQuorum indicates the IP address of the node where ZooKeeper locates. The
default port is 2181.
    zkQuorum =  clientInfo.getProperty("zk.quorum");
    auth = clientInfo.getProperty("auth");
    sasl_qop = clientInfo.getProperty("sasl.qop");
    zooKeeperNamespace = clientInfo.getProperty("zooKeeperNamespace");
    serviceDiscoveryMode = clientInfo.getProperty("serviceDiscoveryMode");
    principal = clientInfo.getProperty("principal");
    auditAddition = clientInfo.getProperty("auditAddition");
    // Set a user name for the newly created user, where xxx indicates the username created previously.
For example, if the created user is user, USER_NAME is user.
    USER_NAME = "xxx";

    if ("KERBEROS".equalsIgnoreCase(auth)) {
      // Set the keytab and krb5 file path on the client.
      USER_KEYTAB_FILE = userdir + "user.keytab";
      KRB5_FILE = userdir + "krb5.conf";

      LoginUtil.setJaasConf(ZOOKEEPER_DEFAULT_LOGIN_CONTEXT_NAME, USER_NAME,
USER_KEYTAB_FILE);
      LoginUtil.setZookeeperServerPrincipal(ZOOKEEPER_SERVER_PRINCIPAL_KEY,
ZOOKEEPER_DEFAULT_SERVER_PRINCIPAL);

      // Security mode
      // Zookeeper Login Authentication
      LoginUtil.login(USER_NAME, USER_KEYTAB_FILE, KRB5_FILE, CONF);
    }
  }

  /**
   * This example shows how to use the Hive JDBC interface to run the HQL command <br>.
   * <br>
   *
   * @throws ClassNotFoundException
   * @throws IllegalAccessException
   * @throws InstantiationException
   * @throws SQLException
   * @throws IOException
   */
  public static void main(String[] args) throws InstantiationException,
      IllegalAccessException, ClassNotFoundException, SQLException, IOException{
    // Parameter Initialization
    init();

    // Define HQL. HQL must be a single statement and cannot contain ";".
    String[] sqls = {"CREATE TABLE IF NOT EXISTS employees_info(id INT,name STRING)",
        "SELECT COUNT(*) FROM employees_info", "DROP TABLE employees_info"};

    // Build JDBC URL
    StringBuilder sBuilder = new StringBuilder(
        "jdbc:hive2://").append(zkQuorum).append("/");

    if ("KERBEROS".equalsIgnoreCase(auth)) {
      sBuilder.append(";serviceDiscoveryMode=")
              .append(serviceDiscoveryMode)
              .append(";zooKeeperNamespace=")
              .append(zooKeeperNamespace)
```

```
            .append(";sasl.qop=")
            .append(sasl_qop)
            .append(";auth=")
            .append(auth)
            .append(";principal=")
            .append(principal);
    } else {
      // Normal mode
      sBuilder.append(";serviceDiscoveryMode=")
            .append(serviceDiscoveryMode)
            .append(";zooKeeperNamespace=")
            .append(zooKeeperNamespace)
            .append(";auth=none");
    }
    if (auditAddition != null && !auditAddition.isEmpty()) {
        strBuilder.append(";auditAddition=").append(auditAddition);
    }
    String url = sBuilder.toString();

    // Load the Hive JDBC driver.
    Class.forName(HIVE_DRIVER);

    Connection connection = null;
    try {
      // Obtain the JDBC connection.
      // If the normal mode is used, the second parameter needs to be set to a correct username.
Otherwise, the anonymous user will be used for login.
      connection = DriverManager.getConnection(url, "", "");

      // Create a table.
      // If data needs to be imported to the table, use the load statement to import data to the table,
for example, import data from the HDFS to the table.
      // load data inpath '/tmp/employees.txt' overwrite into table employees_info;
      execDDL(connection,sqls[0]);
      System.out.println("Create table success!");

      // Query the table.
      execDML(connection,sqls[1]);

      // Delete the table
      execDDL(connection,sqls[2]);
      System.out.println("Delete table success!");
    }
    finally {
      // Close the JDBC connection.
      if (null != connection) {
        connection.close();
      }
    }
  }

  public static void execDDL(Connection connection, String sql)
  throws SQLException {
    PreparedStatement statement = null;
    try {
      statement = connection.prepareStatement(sql);
      statement.execute();
    }
    finally {
      if (null != statement) {
        statement.close();
      }
    }
  }


  public static void execDML(Connection connection, String sql) throws SQLException {
    PreparedStatement statement = null;
    ResultSet resultSet = null;
```

```
    ResultSetMetaData resultMetaData = null;

    try {
      // Execute HQL.
      statement = connection.prepareStatement(sql);
      resultSet = statement.executeQuery();

      // Export the queried column names to the console.
      resultMetaData = resultSet.getMetaData();
      int columnCount = resultMetaData.getColumnCount();
      for (int i = 1; i <= columnCount; i++) {
        System.out.print(resultMetaData.getColumnLabel(i) + '\t');
      }
      System.out.println();

      // Export the query results to the console.
      while (resultSet.next()) {
        for (int i = 1; i <= columnCount; i++) {
          System.out.print(resultSet.getString(i) + '\t');
        }
        System.out.println();
      }
    }
    finally {
      if (null != resultSet) {
        resultSet.close();
      }

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

}
```

## HCatalog Secondary Development Code Example

The following code example demonstrates how to use the HCatInputFormat and HCatOutputFormat interfaces provided by HCatalog to submit MapReduce jobs.

```
public class HCatalogExample extends Configured implements Tool {

    public static class Map extends
            Mapper<LongWritable, HCatRecord, IntWritable, IntWritable> {
        int age;
        @Override
        protected void map(
                LongWritable key,
                HCatRecord value,
                org.apache.hadoop.mapreduce.Mapper<LongWritable, HCatRecord,
                        IntWritable, IntWritable>.Context context)
                throws IOException, InterruptedException {
            age = (Integer) value.get(0);
            context.write(new IntWritable(age), new IntWritable(1));
        }
    }

    public static class Reduce extends Reducer<IntWritable, IntWritable,
    IntWritable, HCatRecord> {
      @Override
      protected void reduce(
              IntWritable key,
              java.lang.Iterable<IntWritable> values,
              org.apache.hadoop.mapreduce.Reducer<IntWritable, IntWritable,
              IntWritable, HCatRecord>.Context context)
              throws IOException, InterruptedException {
          int sum = 0;
          Iterator<IntWritable> iter = values.iterator();
```

```
            while (iter.hasNext()) {
                sum++;
                iter.next();
            }
            HCatRecord record = new DefaultHCatRecord(2);
            record.set(0, key.get());
            record.set(1, sum);

            context.write(null, record);
        }
    }

    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        String[] otherArgs = args;

        String inputTableName = otherArgs[0];
        String outputTableName = otherArgs[1];
        String dbName = "default";

        @SuppressWarnings("deprecation")
        Job job = new Job(conf, "GroupByDemo");

        HCatInputFormat.setInput(job, dbName, inputTableName);
        job.setInputFormatClass(HCatInputFormat.class);
        job.setJarByClass(HCatalogExample.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(WritableComparable.class);
        job.setOutputValueClass(DefaultHCatRecord.class);

        OutputJobInfo outputjobInfo = OutputJobInfo.create(dbName,outputTableName, null);
        HCatOutputFormat.setOutput(job, outputjobInfo);
        HCatSchema schema = outputjobInfo.getOutputSchema();
        HCatOutputFormat.setSchema(job, schema);
        job.setOutputFormatClass(HCatOutputFormat.class);

        return (job.waitForCompletion(true) ? 0 : 1);
    }
    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new HCatalogExample(), args);
        System.exit(exitCode);
    }
}
```

# 8 Hudi

## 8.1 Applicable Scenarios

**Full data analysis**: When an analysis needs to read full data in a table, you can use the real-time view of Hudi to provide the latest full data for the analysis engine.

**Quick data analysis**: When an analysis poses higher requirements on analysis performance than eventually consistent or full data, you can use the read-optimized view of Hudi to improve read efficiency.

**Incremental data analysis**: In incremental data extract, transform and load (ETL) and online analytical processing (OLAP) scenarios, you can use the incremental view of Hudi to read the latest incremental data or the incremental data submitted at a specified time, eliminating the need to search the entire full data and significantly improving the read performance.

**Historical image data analysis**: To analyze data at a historical time point, you can use Multiversion Concurrency Control (MVCC) of Hudi to read image data of a specific version.

## 8.2 Suggestions

Currently, Hudi is mainly applicable to real-time data import to the lake and incremental data ETL. Stored historical data can be imported to Hudi tables in batches.

Copy on write (COW) tables apply to scenarios where the incremental data is basically new data and that have high requirements on data read performance.

Merge on read (MOR) tables apply to scenarios that have high requirements on data import performance and where the incremental data contains a large amount of added and updated data.

You are advised to use the date field to set partition paths in hoodie keys.

Configure Hudi resources for real-time data importing to the data lake in based on the number of Kafka partitions. One Kafka partition can be consumed by only one executor-core. Therefore, setting excessive executor-cores wastes resources.

Set the consumption batch parameters for Spark Streaming to write data to the data lake based on site requirements. Ensure that the interval between two batches is slightly smaller than the time required for consuming a batch of messages to write data into the Hudi table.

The degree of parallelism (DOP) of Hudi write operations cannot be too large. A proper DOP helps shorten the processing time.

# 9 Kafka

## 9.1 Application Scenarios

Kafka is a distributed message releasing and subscription system. It provides features such as message persistence, high-throughput, multi-client support, and real-time message processing. Kafka is applicable to online and offline message consumption as well as Internet service massive data collection scenarios, such as conventional data collection, active website tracking, aggregation of operation data in statistics systems (monitoring data), and log collection.

Reasons for using the message system

- Decoupling: The message system inserts a hidden, data-based interface layer during data processing.
- Redundancy: Message queues are persistent, preventing data loss.
- Scalability: Message queues are decoupled from data processing, facilitating the expansion of data processing.
- Recoverability: Data processing can be recovered in case of failures.
- Sequence guarantee: Message queues help ensure the message sequence and keep the messages in a partition in order.
- Asynchronous communication: Messages can join the queue for further processing when necessary.

## 9.2 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.

## 9.3 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 Application Scenarios

Files of Mapreduce are stored in the HDFS. MapReduce is a programming model used for parallel computation of large data sets (larger than 1 TB). It is advised to use MapReduce when the file being processed cannot be loaded to memory.

It is advised to use Spark if MapReduce is not required.

## 10.2 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);
}
```

# 10.3 Suggestions

## Specify the global configuration items in mapred-site.xml

The mapred-site.xml file contains the following configuration items:

```
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"
```

# 10.4 Examples

## Count the number of female netizens who dwell on online shopping for more than 2 hours at a weekend.

The operation involves three steps:

1. Filter the online time of female netizens in log files using the MapperClass inherited from the Mapper abstract class.
2. Calculate the online time of each female netizen and output information about the female netizens who dwell online for more than 2 hours using the ReducerClass inherited from the Reducer abstract class.
3. Use the main method to create a MapReduce job and then submit the MapReduce job to the Hadoop cluster.

### Step 1: Use MapperClass to define the map() and setup() methods of the Mapper abstract class.

```
public static class MapperClass extends

Mapper<Object, Text, Text, IntWritable> {
// Separator
String delim;
```

```
// Filter the sex.
String sexFilter;
private final static IntWritable timeInfo = new IntWritable(1);
private Text nameInfo = new Text();
/**
* 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 {
// A row of characters read.
String line = value.toString();
if (line.contains(sexFilter)) {
// Obtain the names.
String name = line.substring(0, line.indexOf(delim));
nameInfo.set(name);
// Obtain information about the online time.
String time = line.substring(line.lastIndexOf(delim),
line.length());
timeInfo.set(Integer.parseInt(time));
// map outputs (key, value) pairs.
context.write(nameInfo, timeInfo);
}
}
/**
* 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 {
// Obtain configuration information using Context.
sexFilter = delim + context.getConfiguration().get("log.sex.filter", "female") + delim;
}
}
```

### Step 2: Use CReducerClass to define the reduce() method of the Reducer abstract class.

```
public static class ReducerClass extends
Reducer<Text, IntWritable, Text, IntWritable> {
private IntWritable result = new IntWritable();
// Total time limit.
private int timeThreshold;
/**
* @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 {
int sum = 0;
for (IntWritable val : values) {
sum += val.get();
}
// If the time is smaller than the time limit, no information will be returned.
if (sum < timeThreshold) {
return;
}
result.set(sum);
// reduce output: key: female netizen information, value: online time
context.write(key, result);
}
/**
* 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 {
// Obtain configuration information using Context.
timeThreshold = context.getConfiguration().getInt(
"log.time.threshold", 120);
}
}
```

### Step 3: Use the main() method to create a job, set parameters, and submit the job to the Hadoop cluster.

```
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, "Collect Female Info");
// Locate the jar package of the major task.
job.setJar("D:\\mapreduce-examples\\hadoop-mapreduce-examples\\mapreduce-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);
}
```

# 11 Spark

## 11.1 Application Scenarios

Spark is a distributed batch processing framework. It provides analysis and mining and iterative memory computing capabilities and supports application development in multiple programming languages, including Scala, Java, and Python. Spark is applicable to the following scenarios:

- Data processing: Spark can process data quickly and has fault tolerance and scalability.
- Iterative computation: Spark supports iterative computation to meet the requirements of multi-step data processing logic.
- Data mining: Spark supports complex mining and analysis of massive data and supports multiple data mining and machine learning algorithms.
- Streaming Processing: Spark supports streaming processing with a second-level delay and supports multiple external data sources.
- Query Analysis: Spark supports standard SQL query analysis, provides the DSL (DataFrame), and supports multiple external input types.

## 11.2 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.

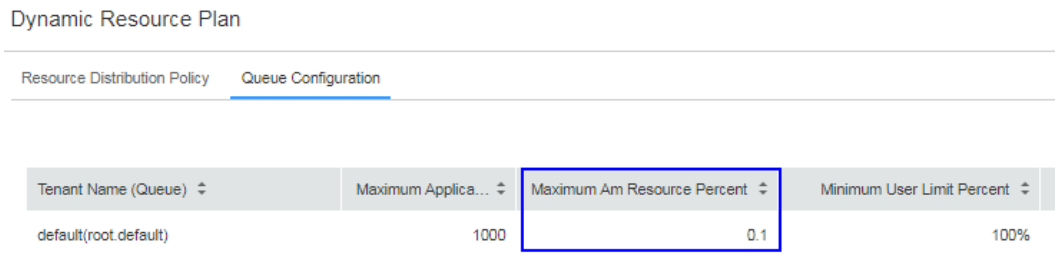
**Figure 11-1** Difference when SparkContext.stop() is added


## 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




# 11.3 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.

# 12 Yarn

## 12.1 Application Scenarios

YARN is a distributed resource management system that is used to improve resource usage in the distributed cluster environment. Resources include memory, I/O, network resources, and disk resources. YARN is developed to address the shortage of the original MapReduce framework. Initially, MapReduce's committers were able to modify the existing codes periodically. As codes increase and due to the problems in the design of the original MapReduce framework, the modification becomes increasingly difficult. Therefore, MapReduce's committers decides to redesign the framework with enhancements to scalability, availability, reliability, and compatibility, increasing the resource usage of next-generation MapReduce(MRv2/Yarn) and supporting more computing frameworks apart from MapReduce.

## 12.2 Rules

### Use YarnClient. createYarnClient() to create a client

Do not directly use the protocol interface ClientRMProxy.createRMProxy(config,ApplicationClientProtocol.class) to create a client.

### The Application Master uses the asynchronous interface AMRMClientAsync.createAMRMClientAsync() to interact with the ResourceManager

Do not directly use the protocol interface ClientRMProxy.createRMProxy(config,ApplicationMasterProtocol.class) to create a client used by the Application Master to interact with the ResourceManager.

## The Application Master uses the asynchronous interface AMRMClientAsync.createAMRMClientAsync() to interact with the NodeMabager

Do not directly use the ContainerManagementProtocolProxy interface to create a client used by the Application Master to interact with the NodeManager.

## 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;
}
```