Mapreduce 服务

组件开发规范

文档版本 01

发布日期 2025-02-14

华为云计算技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为云计算技术有限公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为云计算技术有限公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为云计算技术有限公司

地址： 贵州省贵安新区黔中大道交兴功路华为云数据中心 邮编：550029

网址： https://www.huaweicloud.com/

目 录

1 ClickHouse 应用开发规范

1.1 ClickHouse 设计规范概述

内容介绍

本文主要描述ClickHouse数据管理全生命周期过程中，数据库规划、建模设计、开发、调优、运维的规则建议和指导。

通过这些约束和建议，指导开发者在ClickHouse数据库开发使用过程中能够最大化发挥数据库的优势，保障ClickHouse数据库高性能、稳定可靠运行。用户可更专注于上层业务，释放数据更大的价值。

表 1-1 ClickHouse 设计规范说明

项目	描述
数据库规划	集群业务规划、容量规划、数据分布。
数据库设计	Database设计、宽表设计、分布式表设计、本地表设计、分区设计、索引设计、物化视图设计。
数据库开发	简单查询、聚合查询、join查询、数据增/删/改等SQL开发。
数据库调优	调优思路、参数调优、系统调优、SQL改写调优。
数据库运维	监控、告警、日志、系统表/视图。

适用范围

规范适用于ClickHouse数据库设计、数据库开发、数据库测试、数据库运维以及DBA和业务使用人员。

1.2 ClickHouse 集群规划

1.2.1 ClickHouse 集群业务规划

- 集群规模

  建议单集群不超过256节点规模。

- 集群负载

  对于不同业务负载的业务，需要分开集群部署，便于不同负载的业务进行资源隔离。

- 集群并发

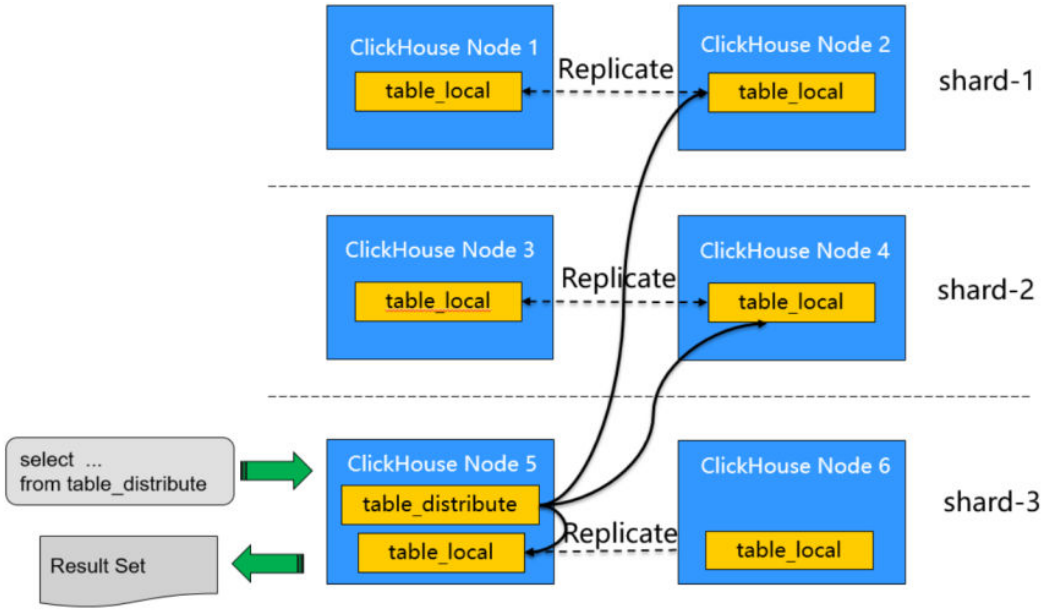
  由于ClickHouse单个SQL会最大化使用每个主机上的CPU/内存/IO资源，对于复杂SQL查询（复杂聚合、复杂join计算）能够支持50~100并发，对于简单的SQL查询，支持100~200左右查询。

  如果集群有混合负载（要求极致性能的点查/范围查询和有大数据量聚合及join查询），建议将不同类型的负载拆分到不同集群；对于集群规划有远远超过100个并发业务系统，也需要设计将业务分摊到不同的集群。

1.2.2 ClickHouse 数据分布设计

Shard 和副本概念介绍

图 1-1 ClickHouse 集群架构图



从横向来看ClickHouse数据库集群，所有数据都会平均分布到多个shard分片中进行保存，数据平均分布后，保证了查询的高度并行性，以提升数据的查询性能。

从纵向来看，每个shard内部有多个副本组成，保证分片数据的高可靠性，以及计算的高可靠性。

数据分布设计

- Shard数据分片均匀分布

  建议用户的数据均匀分布到集群中的多个shard分片，如图1-1所示有3个分片。

假如有30 GB数据需要写入到集群中，需要将30 GB数据均匀切分后分别放到shard-1、shard-2和shard-3的3个分片节点中，以充分发挥MPP查询时并行计算能力，避免数据在shard间倾斜计算出现木桶效应，导致SQL查询性能较差。

可通过弹性负载均衡（Elastic Load Balance，简称ELB）访问ClickHouse，来实现数据均匀。

- Shard内数据副本高可靠存储

  数据写入单shard中的一个副本后，ClickHouse会自动异步将数据同步到其他副本，如图1-1中的shard-3。

  如果将10GB数据导入ClickHouse Node 5节点副本，ClickHouse会自动异步将数据同步到ClickHouse Node 6节点副本，保证shard-3分片数据的高可靠性存储。

1.2.3 ClickHouse 容量规划设计

为了能够更好的发挥ClickHouse分布式查询能力，在集群规划阶段需要合理设计集群数据分布存储。

当前ClickHouse能力为单机磁盘容量达到80%后会上报告警信息，磁盘容量达90%后集群会处于只读状态。

出现磁盘告警信息后需要考虑是否是容量不足问题，如果是容量不足问题需要尽快考虑集群扩容，提升集群整体容量存储。

ClickHouse节点及容量规划如下：

- 磁盘规划

  由于ClickHouseServer业务数据主要存储在本地磁盘上，数据量可能会随着集群使用时间增长而增长，通常建议ClickHouse数据盘单独挂载，元数据盘共享第一个数据盘目录。

- 磁盘实际容量

  由于磁盘存在1MB = 1024KB或者1000KB的不同算法，一般来说，磁盘实际可用容量 = 磁盘标注容量 * 0.9。

  例如磁盘标注容量为1.2 TB，实际容量为1200 * 0.9 = 1080 GB。

- 计算公式

  假设历史数据量为H，每日增量为A，单节点磁盘容量为C，数据保留M天，集群副本数为R，则ClickHouseServer物理节点数计算公式如下：

  ClickHouseServer物理节点数N = [R * (H + A * M)] / C

1.2.4 ClickHouse 依赖服务设计

为了保证ClickHouse服务的稳定，需要提早规划好对于底层依赖服务的设计，主要是ZooKeeper，尤其是在使用replicated*系列表引擎的场景下。

1. ZooKeeper默认部署在MRS集群的Master节点，根据节点CPU和内存规格，调整ZooKeeper实例的最大可用内存。

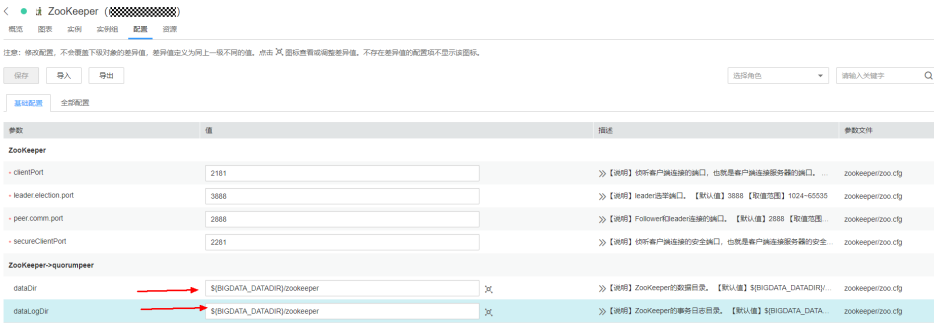
   登录MRS集群的FusionInsight Manager界面，单击“集群 > 服务 > ZooKeeper > 配置 > 全部配置 > quorumpeer > 系统”，调整“GC_OPTS”参数：-Xmx*最大内存数*GB。

   最大内存数参考值：master节点内存-16GB * 0.65（保守估计值）

   说明

   修改完成后需要重启ZooKeeper服务。

2. 修改ZooKeeper的数据盘和日志盘默认配置，改为不同磁盘。



3. 完成后同步修改ClickHouse服务的ZooKeeper相关配置。

   登录MRS集群的FusionInsight Manager界面，单击“集群 > 服务 > ClickHouse > 配置 > 全部配置 > ClickHouse > Zookeeper”。配置调整后通常不需要重启Clickhouse服务。

1.3 ClickHouse 数据库设计

1.3.1 ClickHouse DataBase 设计

业务隔离设计-各业务分库设计

在业务规划时，不同业务归属于不同数据库，便于后续对应用户关联的数据库下表、视图等数据库对象权限的分离管理和维护。

业务隔离设计-不要在 system 库中创建业务表

system数据库是ClickHouse默认的系统数据库，默认数据库中的系统表记录的是系统的配置、元数据等的信息数据。

业务在使用ClickHouse的时候，需要指定自己业务的数据库进行连接和使用，业务相关的表创建在自己业务库中，不要将业务的表创建在系统数据库中，避免对系统数据库造成不必要的影响。

命名规范设计规则

- 所有命名采用26个英文字母和0~9这10个自然数，加上下划线_组成，一般不要出现其他符号。
- 对象名尽量的短，能表达业务所使用数据库含义即可，以英文单词、单词组合或英文单词缩写组成，不以数字或下划线_开头。
- 命名尽量不要使用SQL保留字，请注意大小写敏感。如果必须要使用一些保留关键字，请使用双引号（""）或者反引号（`）进行转义。

1.3.2 ClickHouse 表引擎适用场景说明

ClickHouse中最强大的表引擎当属MergeTree（合并树）引擎及该系列其他引擎，根据业务场景选择合适的引擎。

表引擎选择建议

- 自助报表分析、行为数据分析，在不涉及重复数据聚合的情况下，建议使用ReplicatedMergeTree表引擎。
- 涉及到物化视图等聚合函数的场景，建议使用ReplicatedAggregatingMergeTree表引擎。
- 经常有数据去重或有update修改数据的场景下，建议使用ReplacingMergeTree表引擎，配合使用argMax函数获取最新数据。

表 1-2 应用场景列表

引擎名称	应用场景
MergeTree	ClickHouse中最重要的引擎，基于分区键（partitioning key）的数据分区分块存储、前缀稀疏索引（order by和primary key）。
ReplacingMergeTree	相对于MergeTree，它会用最新的数据覆盖具有相同主键的重复项。 删除老数据的操作是在分区异步merge的时候进行处理，只有同一个分区的数据才会被去重，分区间及shard间重复数据不会被去重，所以应用侧想要获取到最新数据，需要配合argMax函数一起使用。
SummingMergeTree	当合并SummingMergeTree表的数据片段时，ClickHouse会把所有具有相同主键的行进行汇总，将同一主键的行替换为包含sum后的一行记录。 如果主键的组合方式使得单个键值对应于大量的行，则可以显著地减少存储空间并加快数据查询的速度。
AggregatingMergeTree	该引擎继承自MergeTree，并改变了数据片段的合并逻辑。 ClickHouse会将一个数据片段内所有具有相同主键（准确的说是排序键）的行替换成一行，这一行会存储一系列聚合函数的状态。可以使用AggregatingMergeTree表引擎来做增量数据的聚合统计，包括物化视图的数据聚合。
CollapsingMergeTree	在创建时与MergeTree基本一样，除了最后多了一个参数，需要指定Sign位（必须是Int8类型）。 CollapsingMergeTree会异步地删除（折叠）除了特定列Sign1和-1值以外的所有字段的值重复的行。
VersionedCollapsingMergeTree	是CollapsingMergeTree的升级，使用不同的collapsing算法，该算法允许使用多个线程以任何顺序插入数据。

引擎名称	应用场景
Replicated* MergeTree	只有Replicated*MergeTree系列引擎是上面介绍的引擎的多副本版本，为了提升数据和服务的可靠性，建议使用副本引擎： • ReplicatedMergeTree • ReplicatedSummingMergeTree • ReplicatedReplacingMergeTree • ReplicatedAggregatingMergeTree • ReplicatedCollapsingMergeTree • ReplicatedVersionedCollapsingMergeTree • ReplicatedGraphiteMergeTree

1.4 ClickHouse 宽表设计

1.4.1 ClickHouse 宽表设计原则

宽表设计原则

由于ClickHouse的宽表查询性能较优，且当前ClickHouse可支持上万列的宽表横向扩展。

在大部分场景下，有大表两表join以及多表join的场景，且多个join的表数据变化更新频率较低，这种情况，建议对多个表join查询逻辑提前进行加工处理，将处理后的数据写入到一个宽表中，宽表中包含所有要查询的数据字段，以供后续应用完全自助OLAP的高性能查询。

表命名规范

数据库表名称命名规则：

- 在数据库中，表名命名要求在当前数据库内唯一。
- 表名要求以字符开始，可以包含字符（a～z，A～Z）、数字（0～9）及下划线（_）。

1.4.2 ClickHouse 表字段设计

规则

- 不允许用字符类型存放时间或日期类数据，尤其是需要对该日期字段进行运算或者比较的时候。
- 不允许用字符类型存放数值类型的数据，尤其是需要对该数值字段进行运算或者比较的时候。字符串的过滤效率相对于整型或者特定时间类型有下降。

建议

- 不建议表中存储过多的Nullable列，可以考虑字符串使用“NA”，数值型用0作为缺省值。过多使用Nullable将消耗更多内存。

- 建议规划好业务所需的列，必要时可提前预置一些属性列，避免频繁的增删列。
- 数值类型：UInt8/UInt16/UInt32/UInt64、Int8/Int16/Int32/Int64, Float32/Float64等，选择不同长度，性能差别较大。

  建议根据业务场景所需选择最小满足的类型使用。

  - 示例

    ```
    CREATE TABLE counter ON CLUSTER default_cluster
    (
    `when`     DateTime DEFAULT now(),
    `device`    UInt32,
    `value`     Float32,
    `value64`   Float64
    )
    ENGINE = MergeTree
    PARTITION BY toYYYYMM(when)
    ORDER BY (device, when)
    ```

    表中有Float32类型的字段value和Float64的字段value64插入数据的查询表现如下：

    ```
    INSERT INTO counter
    SELECT
    toDateTime('2019-01-01 00:00:00') + toInt64(number / 10) AS when,
    (number % 10) + 1 AS device,
    (device * 3) + (number / 10000) AS value,
    value
    FROM system.numbers
    LIMIT 100000000;
    ```

    往value和value64插入相同的数据，总数据量1亿条。

  - 查询Float32字段

    ```
    SELECT countDistinct(value)
    FROM counter
    WHERE device = 1

    ┌─uniqExact(value)─┐
    │         10000000 │
    └──────────────────┘

    1 rows in set. Elapsed: 0.750 sec. Processed 10.04 million rows, 80.35 MB (13.39 million rows/s., 107.14 MB/s.)
    ```

    耗时：0.750秒。

  - 查询Float64字段

    ```
    SELECT countDistinct(value64)
    FROM counter
    WHERE device = 1

    ┌─uniqExact(value64)─┐
    │           10000000 │
    └────────────────────┘

    1 rows in set. Elapsed: 0.929 sec. Processed 10.04 million rows, 120.52 MB (10.81 million rows/s., 129.76 MB/s.)
    ```

    耗时：0.929秒。

    结果：Float32类型的查询时间比Float64更快。

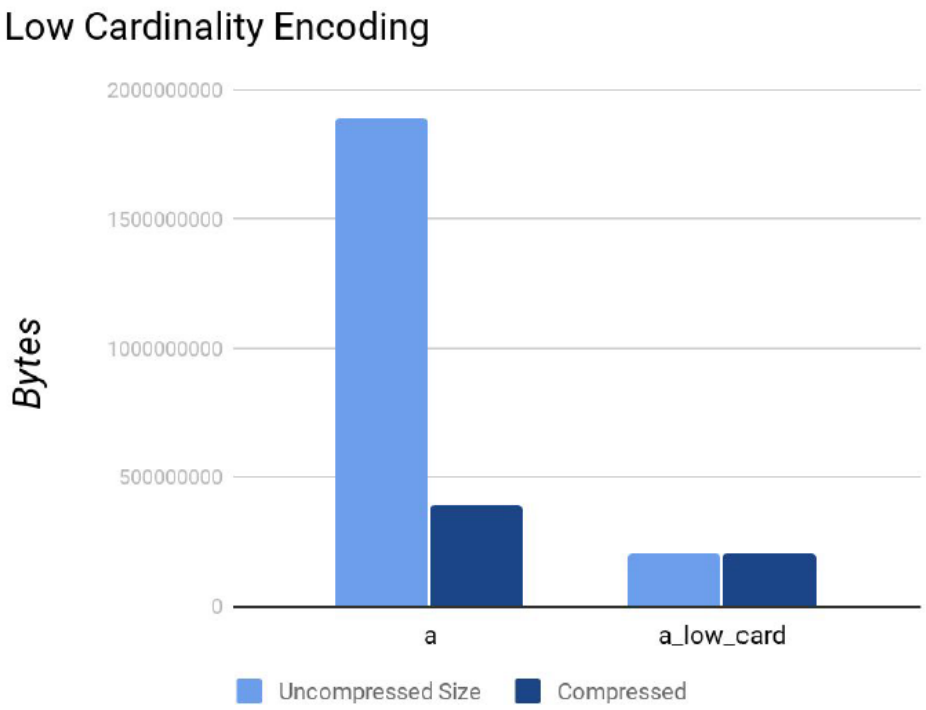
- 低基数维度（基数1万内），建议使用LowCardinality修饰符，提升查询性能。
  - 维度的基数（Cardinality）：指的是该维度在数据集中出现的不同值的个数。例如“国家”是一个维度，如果有200个不同的值，那么此维度的基数就是200。
  - 根据官方建议和实践经验，在维度基数小于1万的时候，对维度字段做LowCardinality编码，导入性能会有略微下降，查询性能提升明显，数据存储空间下降明显。
  - 在默认的情况下，声明了LowCardinality的字段会基于数据生成一个全局字典，并利用倒排索引建立Key和位置的对应关系。如果数据的基数大于8192，也就是说不同的值多于8192个，则会将一个全局字典拆分成多个局部字典（low_cardinality_max_dictionary_size参数控制，默认8192）。

- 示例

```
CREATE TABLE test_codecs ON CLUSTER default_cluster
(
`a` String,
`a_low_card` LowCardinality(String) DEFAULT a
)
ENGINE = MergeTree
PARTITION BY tuple()
ORDER BY tuple();
```

其中，字段a是原生字符串，字段a_low_card基于a做了低基维编码。

- 数据存储的对比



- 查询性能对比

```
SELECT a AS a, count(*) AS c FROM test_codecs
GROUP BY a ORDER BY c ASC LIMIT 10
. . .
10 rows in set. Elapsed: 0.681 sec. Processed 100.00 million
rows, 2.69 GB (146.81 million rows/s., 3.95 GB/s.)
```

Faster

```
SELECT a_lc AS a, count(*) AS c FROM test_codecs
GROUP BY a ORDER BY c ASC LIMIT 10
. . .
10 rows in set. Elapsed: 0.148 sec. Processed 100.00 million
rows, 241.16 MB (675.55 million rows/s., 1.63 GB/s.)
```

查询性能有5倍的提升。

1.4.3 ClickHouse 本地表设计

规则

- 单表（分布式表）的记录数不要超过万亿，对于万亿以上表的查询，性能较差，且集群维护难度变大。单表（本地表）不超过百亿。
- 表的设计都要考虑到数据的生命周期管理，需要进行TTL表属性设置或定期老化清理表分区数据。
- 单表的字段建议不要超过5000列。

  因为当一次插入的数据大小超过“min_bytes_for_wide_part”（默认值:10485760），ClickHouse写入会按每列1 MB（Nullable类型2MB）来预申请内存，容易出现内存超限的错误：

  ```
  Received exception from server (version 22.3.4):
  Code:241. DB::Exception: Received from localhost:9000. DB::Exception: Memory limit (for query) exceeded: would use 9.31 Gib (attempt to allocate chunk of 1048591 bytes), maximum: 9.31 GiB
  ```

  可以通过调大“min_bytes_for_wide_part”来规避。

参考案例

- MergeTree引擎在建表的时候支持列字段和表级的TTL。

  当列字段中的值过期时，ClickHouse会将其替换成数据类型的默认值。如果分区内，某一列的所有值均已过期，则ClickHouse会从文件系统中删除这个分区目录下的列文件。当表内的数据过期时，ClickHouse会删除所有对应的行。

  在列上配置TTL：

  ```
  CREATE TABLE default.t_column_ttl ON CLUSTER default_cluster
  (
  `did` Int32,
  `app_id` Int32,
  `region` Int32,
  `pt_d` Date,
  `create_time` Datetime,
  `product_desc1` String TTL create_time + toIntervalSecond(10),
  `product_desc2` String TTL create_time + toIntervalMonth(10),
  `product_desc3` String TTL create_time + toIntervalHour(10)
  )
  ENGINE = MergeTree()
  PARTITION BY toYYYYMMDD(pt_d)
  ORDER BY (app_id, region);
  ```

  在表上配置TTL：

  ```
  CREATE TABLE default.t_table_ttl ON CLUSTER default_cluster
  (
  `did` Int32,
  `app_id` Int32,
  `region` Int32,
  `pt_d` Date,
  `create_time` Datetime
  )
  ENGINE = MergeTree()
  PARTITION BY toYYYYMMDD(pt_d)
  ORDER BY (app_id, region)
  TTL create_time + toIntervalMonth(12);
  ```

  TTL详细使用见官网链接：

  https://clickhouse.tech/docs/en/engines/table-engines/mergetree-family/mergetree/#table_engine-mergetree-ttl

- 通过外部系统管理数据的生命周期，定时清理过期数据。

  清理数据SQL命令示例：

DROP TABLE default.table_with_non_default_policy ON CLUSTER default_cluster NO delay; #删除表

ALTER TABLE default.table_with_non_default_policy ON CLUSTER default_cluster drop partition 201901; #删除分区

本地表建表参考：

```
CREATE TABLE default.my_table_local ON CLUSTER default_cluster
(
`did`      Int32,
`app_id`   Int32,
`region`   Int32,
`pt_d`     Date
)
ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/default/my_table_local', '{replica}')
PARTITION BY toYYYYMMDD(pt_d)
PRIMARY KEY(app_id)
ORDER BY (app_id, region)
SETTINGS index_granularity = 8192;
```

- 表引擎选择：

  ReplicatedMergeTree：支持副本特性的MergeTree引擎，也是最常用的表引擎，其他表引擎参考使用场景介绍进行选择。

- ZooKeeper上的表元数据信息存储路径“/clickhouse/tables/{shard}/default/my_table_local”：

  {cluster}表示集群名称，{shard}是分片名称，{replica}是分片中的副本编号，这几个宏变量直接写即可，建表时不需要替换为常量值。

  default：表示创建的表名放到哪个数据库下面，在创建表时需要根据实际情况进行替换。

- on cluster：创建的集群

  建表会创建到集群中所有节点上，否则需要自己手动一个个节点去创建，一个个节点创建过程比较繁琐，创建比较慢；如果在集群中部分节点未创建表，在查询时会遇到无表信息的错误提示。

- no delay：立刻生效

  在删除表或修改表语法中加上no delay，表示立即删除，否则会等8分钟以后进行删除，如果未加no delay语法，删除表后需要立即创建同名的表名可能会遇到错误，创建不成功。

- order by：排序字段

  查询时最常使用且过滤性最高的字段作为排序字段。依次按照访问频度从高到低、维度基数从小到大来排。排序字段不宜太多，建议不超过4个，否则merge的压力会较大。排序字段不允许为null，如果存在null值，需要做数据转换。

- primary key：主键字段

  创建主键索引，值为排序字段的前导列，否则不允许创建表，为访问频率最高的字段创建索引，提升查询性能，查询时会通过索引数据快速的找到数据文件中的数据块所在位置信息。

- partition by：分区字段

  分区键不允许为null，如果字段中有null值，需要做数据转换处理。

- 表级别的参数配置：

  index_granularity：稀疏索引粒度配置，默认是8192，一般不需要修改。

建表定义，参考链接：

https://clickhouse.tech/docs/en/engines/table-engines/mergetree-family/mergetree/

1.4.4 ClickHouse 分布式表设计

建议

分布式表建表参考：

```
CREATE TABLE default.my_table_dis ON CLUSTER default_cluster
AS mybase.my_table_local
ENGINE = Distributed(default_cluster, default, my_table_local, rand());
```

使用说明

- 分布式表名称：default.my_table_dis。
- 本地表名称：default.my_table_local。
- 通过“AS”关联分布式表和本地表，保证分布式表的字段定义跟本地表一致。
- 分布式表引擎的参数说明：
  - default_cluster：集群名称。
  - default：本地表所在库名。
  - my_table_local：本地表名。
  - rand()：可选参数，分片键（sharding key），可以是表中一列的原始数据（如did），也可以是函数调用的结果。

    如轮训方式：rand()，表示在写入数据时直接将数据插入到分布式表，分布式表引擎会按轮训算法将数据发送到各个分片。

    ⚠ 注意

    该键是写分布式表保证数据均匀分布在各分片的唯一方式。

规则

不建议写分布式表。

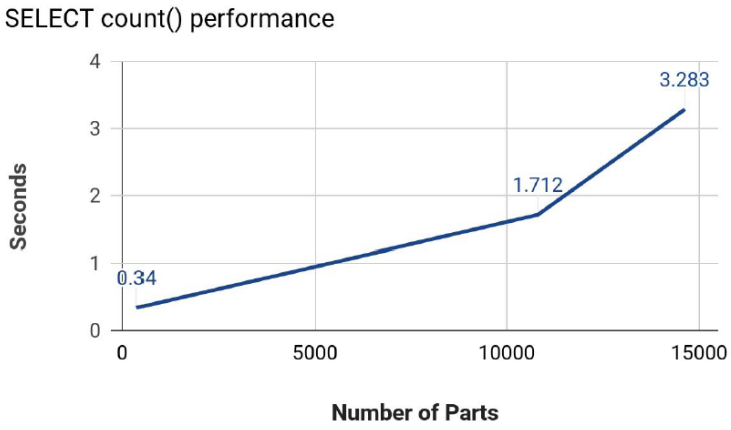
由于分布式表写数据是异步方式，客户端SQL由Balancer路由到一个节点之后，一批写入数据会先落入写入的节点，随后根据分布式表schema定义数据分布规则，将数据异步发送到各个shard的各个副本。整个过程数据异步发送，且数据会在一个节点临时存储，会导致网络、磁盘都会成为瓶颈，且写入成功后不一定能查询到最新一致性数据等问题。

1.4.5 ClickHouse 分区设计

合理设置分区键，控制分区数在一千以内，分区字段使用整型。

分区 part 数与查询性能关系

图 1-2 分区 part 数与查询性能关系图



分区建议

- 建议使用toYYYYMMDD（pt_d）作为分区键，pt_d是date类型。
- 如果业务场景需要做小时分区，使用pt_d、pt_h做联合分区键，其中pt_h是整型小时数。
- 如果保存多年数据，建议考虑使用月做分区，toYYYYMM（pt_d）。
- 综合考虑数据分区粒度、每个批次提交的数据量、数据的保存周期等因素，合理控制part数量。

1.4.6 ClickHouse 索引设计

一级索引设计

- 在建表设计时指定主键字段的建议：按查询时最常使用且过滤性最高的字段作为主键。依次按照访问频度从高到低、维度基数从小到大来排列。数据是按照主键排序存储的，查询的时候，通过主键可以快速筛选数据，合理的主键设计，能够大大减少读取的数据量，提升查询性能。例如所有的分析，都需要指定业务的id，则可以将业务id字段作为主键的第一个字段顺序。
- 根据业务场景合理设计稀疏索引粒度

  ClickHouse的主键索引采用的是稀疏索引存储，稀疏索引的默认采样粒度是8192行，即每8192行取一条记录在索引文件中，实践建议：
  - 索引粒度越小，对于小范围的查询更有效，避免查询资源的浪费；
  - 索引粒度越大，则索引文件越小，索引文件的处理会更快；
  - 超过10亿的表索引粒度可设为16384，其他设为8192或者更小值。

二级跳数索引设计

跳数索引使用参考：

- 使用说明

  对于*MergeTree引擎，支持配置跳数索引，即一种数据局部聚合的粗糙索引，对数据块创建索引，选择性的保留一部分原始数据（minmax、set）， 或者是保留

计算后的中间数据（bloomfilter）。在查询时，选择忽略加载不会包含结果的数据块，从而达到加速查询的效果。

- 索引定义

  INDEX *index_name expr* **TYPE** *type（...）* **GRANULARITY** *granularity_value*

  - Expr：属性表达式，基于字段或者字段的表达式来创建索引；
  - type（...）：支持的索引类型，minmax、set等；
  - Granularity：创建索引的记录粒度。比如index_granularity = 8192，granularity配置为3，则使用8192*3条记录创建一条索引数据。

- 创建索引样例

```
CREATE TABLE skip_index_test ON CLUSTER default_cluster
(
ID     String,
URL    String,
Code   String,
EventTime Date,
INDEX a ID TYPE minmax GRANULARITY 5,
INDEX b (length(ID) * 8) TYPE set(100) GRANULARITY 5,
INDEX c (ID, Code) TYPE ngrambf_v1(3, 256, 2, 0) GRANULARITY 5,
INDEX d ID TYPE tokenbf_v1(256, 2, 0) GRANULARITY 5,
INDEX e ID TYPE bloom_filter(0.025) GRANULARITY 5
) ENGINE = MergeTree()
ORDER BY ID ;
```

  - **minmax索引**

    记录了一段数据范围内的最小和最大极值，其索引的作用类似分区目录的minmax索引，能够快速跳过无用的数据区间。

    ```
    INDEX a ID TYPE minmax GRANULARITY 5
    ```

    上述示例中minmax索引会记录这段数据区间内ID字段的极值。极值的计算涉及每5个index_granularity区间中的数据。

  - **set索引**

    直接记录了声明字段或表达式的取值（唯一值，无重复），其完整形式为set（max_rows），其中max_rows是一个阈值，表示在一个index_granularity内，索引最多记录的数据行数。如果max_rows=0，则表示无限制。

    ```
    INDEX b (length(ID) * 8) TYPE set(100) GRANULARITY 5
    ```

    上述示例中set索引会记录数据中ID的长度*8后的取值。其中，每个index_granularity内最多记录100条。

  - **布隆过滤器**

    - **bloom_filter索引**

      为指定的列存储布隆过滤器。

      可选的参数false_positive用来指定从布隆过滤器收到错误响应的几率。取值范围是 (0,1)，默认值：0.025。

      支持的数据类型：Int*，UInt*，Float*，Enum，Date，DateTime，String，FixedString，Array，LowCardinality，Nullable。

    - **ngrambf_v1索引**

      记录的是数据短语的布隆表过滤器，只支持String和FixedString数据类型。只能够提升in、notIn、like、equals和notEquals查询的性能，其完整形式为：

      ngrambf_v1(n, size_of_bloom_filter_in_bytes, number_of_hash_functions, random_seed)

      这些参数是一个布隆过滤器的标准输入，如果接触过布隆过滤器，应该会对此十分熟悉。

具体的含义如下：

- n：token长度，依据n的长度将数据切割为token短语。
- size_of_bloom_filter_in_bytes：布隆过滤器的大小。
- number_of_hash_functions：布隆过滤器中使用Hash函数的个数。
- random_seed：Hash函数的随机种子。

- **tokenbf_v1索引**

  是ngrambf_v1的变种，同样也是一种布隆过滤器索引。tokenbf_v1除了短语token的处理方法外，其他与ngrambf_v1是完全一样的。tokenbf_v1会自动按照非字符的、数字的字符串分割token。

  INDEX d ID TYPE tokenbf_v1(256,2,0) GRANULARITY 5

- 索引创建详见官方文档

  https://clickhouse.tech/docs/en/engines/table-engines/mergetree-family/mergetree/#table_engine-mergetree-data_skipping-indexes

- 建表后再创建索引

  ALTER TABLE table_name add INDEX min_max_index (etl_time) TYPE minmax GRANULARITY 3;

- 删除索引

  ALTER TABLE table_name DROP INDEX min_max_index;

- 单表跳数索引数量

  由于索引的创建对数据导入性能有影响，建议单表跳数索引的总数量控制在5个以内。

1.5 ClickHouse 物化视图设计

1.5.1 ClickHouse 物化视图概述

由于TTL规则不会从原始表中同步到物化视图表，因此源表中带有TTL规则时，物化视图表同样需要配置TTL规则，并且建议与源表保持一致。

表 1-3 普通物化视图与 projection 对比

物化视图类型	原表数据与物化视图一致性	灵活性	物化视图开发及维护复杂度
普通物化视图	数据从原表同步到物化视图需要时间窗。	• 灵活性较高，有新的业务可开发新的物化视图。 • 可开发复杂逻辑SQL语句的物化视图。	复杂度较高，需要开发很多物化视图，每个物化视图都需要单独去管理和维护。
projection	数据实时同步，数据写入即可查询到物化视图最新数据。	创建表时指定的物化视图语法，新的SQL业务需要修改表结构。	不需要开发很多物化视图，任意查询SQL会自动重写命中物化视图。

说明

Projection仅在MRS 3.2.0及以上的版本集群中支持。

1.5.2 ClickHouse 普通物化视图设计

建议

- 在查询方式固定的场景，建议使用物化视图加速。

  物化视图创建参考如下：

  a. 明细表创建

  ```
  CREATE TABLE counter ON CLUSTER default_cluster
  (
  when DateTime DEFAULT now(),
  device UInt32,
  value Float32
  ) ENGINE=MergeTree
  PARTITION BY toYYYYMM(when)
  ORDER BY (device, when);
  ```

  b. 聚合表创建

  ```
  CREATE TABLE counter_daily_agg ON CLUSTER default_cluster
  (
  day DateTime,
  device UInt32,
  count UInt64,
  max_value_state AggregateFunction(max, Float32),
  min_value_state AggregateFunction(min, Float32),
  avg_value_state AggregateFunction(avg, Float32)
  )
  ENGINE = SummingMergeTree()
  PARTITION BY tuple()
  ORDER BY (device, day);
  ```

  说明

  AggregateFunction类型的字段使用二进制存储，在写入数据时，需要调用*State函数；而在查询数据时，则需要调用相应的*Merge函数。其中，*表示定义时使用的聚合函数。

  c. 物化视图创建

  ```
  CREATE MATERIALIZED VIEW counter_daily_mv ON CLUSTER default_cluster
  TO counter_daily_agg
  AS
  SELECT
  toStartOfDay(when) as day,
  device,
  count(*) as count,
  maxState(value) AS max_value_state,
  minState(value) AS min_value_state,
  avgState(value) AS avg_value_state
  FROM counter
  WHERE when >= toDate('2019-01-01 00:00:00')
  GROUP BY device, day
  ORDER BY device, day;
  ```

  说明

  创建物化视图counter_daily_mv，数据存储到表counter_daily_agg中，数据源来自counter。

- 聚合表在明细表名后加上_{type}_agg后缀；物化视图添加 _{type}_mv后缀。
- 物化视图、聚合表保持与明细表同样的分区类型及ttl时间。
- 物化视图中的group by字段名称与明细表对应字段名称一致；select子句返回列名称与聚合表中列的名称保持一致。
- 物化视图创建时不会进行语法校验，只有发生实际数据插入与查询时才会出错。
- 物化视图上线前，需做好充分验证。

规则

- 物化视图（Materialized View）显式指定聚合表。

  在创建物化视图时，使用TO关键字为物化视图指定数据存储表。

  如果不显示指定聚合表，则会创建隐式表.inner.mv1，与物化视图绑定。
- 用于数据预聚合的物化视图，聚合表使用聚合引擎。

  如果不用聚合引擎，则每次数据插入，会对明细表的全量数据重新计算，而不是只处理增量数据。
- 聚合表中，聚合指标定义成聚合类型（AggregateFunction）。

  物化视图的指标列与聚合表中对应字段名称一致，命名规范如下：

  {aggrateFunction}_{columnName}_state

  聚合表创建样例：

```
CREATE TABLE counter_daily_agg ON CLUSTER default_cluster
(
day DateTime,
device UInt32,
count UInt64,
max_value_state AggregateFunction(max, Float32),
min_value_state AggregateFunction(min, Float32),
avg_value_state AggregateFunction(avg, Float32)
)
ENGINE = SummingMergeTree()
PARTITION BY tuple()
ORDER BY (device, day);
```

- 在创建物化视图时，如果用到了多表联查，只有左表发生数据插入时才会触发物化视图数据修改。
- 禁止在创建物化视图时使用POPULATE关键字。

  使用POPULATE方式创建物化视图期间，如果有数据插入，则可能丢失。
- 推荐的历史数据同步方式：

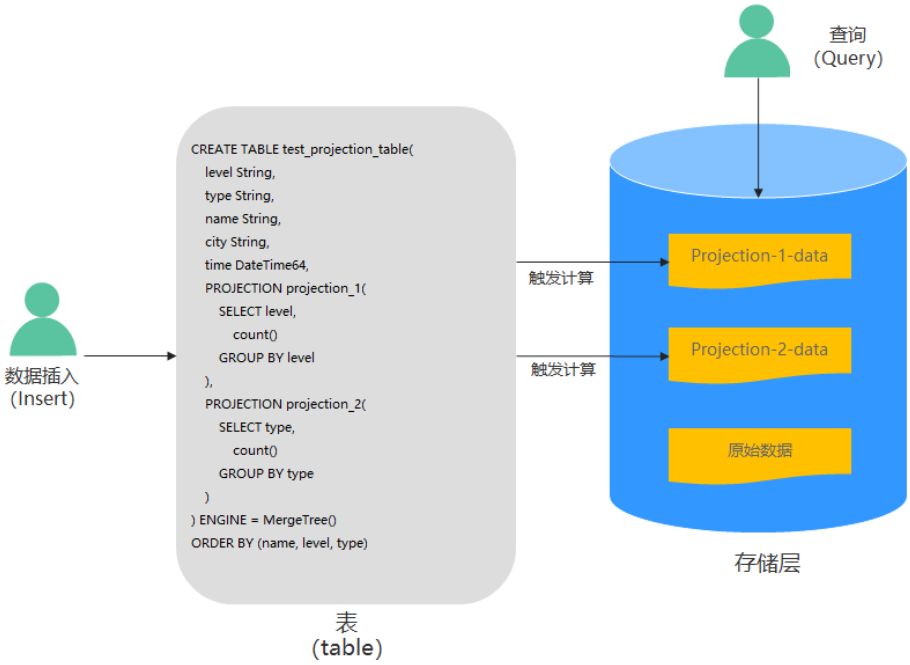
```
-- create MV c where date >= in_the_future
CREATE MATERIALIZED VIEW mv1 ON CLUSTER default_cluster
TO dest
AS
SELECT  a,  d,  count()  AS cnt
FROM source
WHERE d >= '2020-11-01'
GROUP BY a, d;
-- arrives 2020-11-01
INSERT INTO dest -- insert all for before in_the_future
SELECT  a,  d,  count()  AS cnt
FROM source
WHERE d < '2020-11-01' -- piece by piece by 1 month (or .. day)
GROUP BY a, d;
```

- 修改明细表、聚合表结构，严格按照以下步骤实施：

  a. 停止明细表数据插入。

b. 修改聚合表结构设计。
c. 删除物化视图表。
d. 重新创建新转化关系的物化视图。

1.5.3 ClickHouse Projection 设计



说明

Projection仅在MRS 3.2.0及以上的版本集群中支持。

projection 定义

```
CREATE TABLE test_projection_table(
    level String,
    type String,
    name String,
    city String,
    time DateTime64,
    PROJECTION projection_1(
        SELECT level,
            count()
        GROUP BY level
    ),
    PROJECTION projection_2(
        SELECT type,
            count()
        GROUP BY type
    )
) ENGINE = MergeTree()
ORDER BY (name, level, type)
```

通过表属性修改方式创建 projection

在创建好projection后还可以对projection进行修改，具体语句如下：

```
ALTER TABLE test_projection_table
ADD PROJECTION projection_3(
```



```
    SELECT type,
       level
    GROUP BY type,
       level
)
```

Projection 的使用

- 如下SQL查询的时候会走表达式：
  SELECT type, count() FROM test_projection_table WHERE type = 'A' GROUP BY type;
- 而如下SQL不会走projection，因为city不在projection的定义中。
  SELECT city, count() FROM test_projection_table WHERE type = 'A' GROUP BY city;
- 具体可以通过explain查看执行计划，如果出现ReadFromStorage (MergeTree(with projection))，表示命中projection。

命中 projection 使用规则

- Where条件必须是Projection定义中Group By的子集。
- Group By必须是Projection定义中Group By的子集。
- Select必须是Projection定义中Select的子集。
- 多表join场景不支持Projection特性，此种场景建议用普通物化视图实现。

1.6 ClickHouse 逻辑视图设计

建议如下：

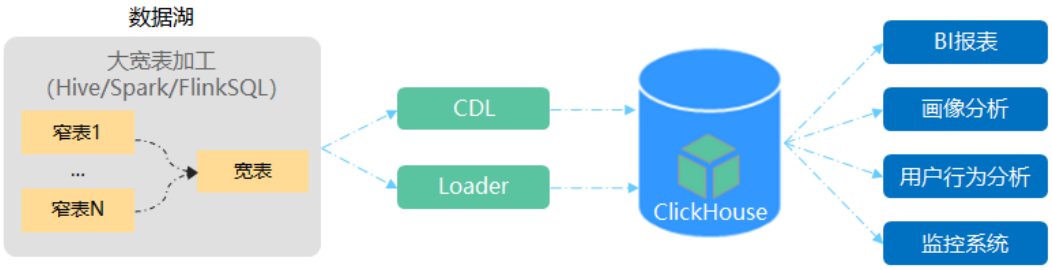
- 业务逻辑上有很多比较复杂的SQL运算，可以封装为一个视图，后续查询时只查询视图，简化业务查询使用。
- 如果业务间有权限隔离诉求，可将部分数据查询封装到视图中，使用视图方只能看到视图下有限行及列的数据。

1.7 ClickHouse 数据库开发

1.7.1 ClickHouse 数据入库工具

最佳实践方案

ClickHouse数据加工流程最佳实践：在数据湖中通过Hive&Spark（批量）/FlinkSQL（增量）加工成大宽表后，通过CDL/Loader工具实时同步到ClickHouse，下游BI工具和应用进行实时OLAP分析。



数据加工

建议使用Hive/Spark进行数据批量加工，FilkSQL进行数据增量加工。

数据入库

建议使用CDL（增量实时同步）和Loader（批量同步）工具进行数据同步，也可选择HDFS外表（CK集群只支持X86平台）用户自己写调度程序进行数据导入。

1.7.2 ClickHouse 数据入库规范

规则

- 写本地表，查询分布式表，提升写入和查询性能，保证写入和查询的数据一致性。
- 只有在去重诉求的场景下，可以使用分布式表插入，通过sharding key将要去重的数据转发到同一个shard，便于后续去重查询。
- 外部模块保证数据导入的幂等性。

  ClickHouse不支持数据写入的事务保证。通过外部导入数据模块控制数据的幂等性，比如某个批次的数据导入异常，则drop对应的分区数据或清理掉导入的数据后，重新导入该分区或批次数据。
- 大批量少频次的写入。

  ClickHouse的每次数据插入，都会生成一到多个part文件，如果data part过多，merge压力会变大，甚至出现各种异常影响数据插入。建议每个批次5k到100k行，写入字段不能太多，太多字段情况下要减少写入行数，以降低对写入节点的内存和CPU压力，每秒不超过1次插入。
- 多副本并行导入。

  有大数据的导入场景，建议将数据提前拆分成多份，在一个shard内的多个副本同时导入，以分摊一个节点导入数据的压力，同时能提升数据入库的性能，缩短入库时间。

  常见错误：

  Too many parts(304). Merges are processing significantly slower than inserts

  原因分析：MergeTree的merge的速度跟不上目录生成的速度，数据目录越来越多就会抛出这个异常。

建议

- 一次只插入一个分区内的数据

  如果数据属于不同的分区，则每次插入，不同分区的数据会独立生成part文件，导致part总数量膨胀，建议一批插入的数据属于同一个分区。
- 写入速率

  单节点写入速度为50~200MB/S，如果写入的数据每行为1Kb，那么写入的速度为50，000到200，000行每秒，如果行数据容量更小，那么写入速度将更高，如果写入性能不够，可以使用多个副本同时写入，同一时间每个副本写入的数据保持均衡。
- 慎用分布式表批量插入
  - 写分布式表，数据会分发到集群的所有本地表，每个本地表插入的数据量是总插入量的1/N，batch size可能比较小，导致data part过多，merge压力变大，甚至出现异常影响数据插入；

  - 数据的一致性问题：数据先在分布式表写入节点的主机落盘，然后数据被异步地发送到本地表所在主机进行存储，中间没有一致性的校验，如果分布式表写入数据的主机出现异常，会存在数据丢失风险；
  - 对于数据写分布式表和数据写本地表相比，分布式表数据写入性能也会变慢，单批次分布式表写，写入节点的磁盘和网络IO会成为性能瓶颈点。
  - 分布式表转发给各个shard成功与否，插入数据的客户端无法感知，转发失败的数据会不断重试转发，消耗CPU。

- 大批量数据导入要分时、分节点、扩容

  如果数据盘为SATA盘，当大批量数据集中插入时候，会抢占磁盘，使得磁盘长时间处于繁忙状态，影响其他alter类操作的效率。

  尽量避免批量导数据的SQL并发执行，会给磁盘和ClickHouse并发能力带来冲击。

- Kafka数据入库

  不建议建ClickHouse kafka表引擎，进行数据同步到ClickHouse中，当前CK的kafka引擎有会导致kafka引擎数据入库产生性能等诸多问题，通过用户使用经验，需要应用侧自己写kafka的数据消费，攒批写入ClickHouse，提升ClickHouse的入库性能。

- 使用分区替换或增加的方式写入数据

  为避免目标表写入脏数据导致的删改，先将数据写入临时表，再从临时表写入目标表。

  操作步骤如下：

  a. 创建一张与目标表table_dest结构、分区键、排序键、主键、存储策略、引擎都一致的临时表table_source。

  b. 先把数据写到临时表，一次只写入一个分区的数据，检查临时表的数据准确无误。

  c. 使用以下SQL查看目标表的分区：

     SELECT partition AS `partition`,sum(rows) AS `count` FROM system.parts WHERE active AND database=='*数据库名*' AND table=='*表名*' GROUP BY partition ORDER BY partition ASC;

  d. 如果目标表存在该分区，将分区替换到目标表，到集群的每个节点上执行如下语法：

     ALTER TABLE table_dest REPLACE PARTITION partition_expr FROM table_source;

  e. 如果目标表不存在该分区，将分区增加到目标表，到集群的每个节点上执行如下语法：

     ALTER TABLE table_dest REPLACE PARTITION tuple() partition_expr FROM table_source;

1.7.3 ClickHouse 数据查询

数据查询规则

- 禁止select *查询

  只查询需要的字段可以减少磁盘io和网络io，提升查询性能。

- 使用uniqCombined替代distinct

  uniqCombined对去重逻辑进行了优化，通过近似去重提升十倍查询性能，如果对查询允许有误差，可以使用uniqCombined替代，否则还继续使用distinct语法。

- 降低对表的修改频次

  默认场景下ClickHouse执行alter语句是异步执行，对同一张表频繁执行alter操作可能导致业务失败。

- 多表复杂join拆分为两表join或子查询

  多表复杂join场景，建议拆分为两两表join，且两表join为大小表join，小小表join，尽量避免大大表join。也可以将多表复杂join拆分为子查询模式。

  SELECT name FROM tab_a WHERE id IN (SELECT id FROM tab_b WHERE name = 'xx');

  注意

  这里说的大表为条件过滤后的总数据量，千万级以上的数据量可定义为大表。

- 关联查询必须大表join小表

  对于ClickHouse来说，原则上需要把多表join模型提前加工为宽表模型，但是在一些情况下，多个表，甚至是维度表变化比较频繁情况下，不太适合进行宽表加工处理，不得已必须使用Join模型以实时查询到最新数据。那么join，建议2表join，大表join小表，小表在后（大表join小表），并必须有关联条件。小表的数据量控制在百万~千万行级别，且需要在join前尽量把小表数据通过条件进行有效过滤。

- join/in/not in需要添加Global关键字

  在通常的join/in/not in时候，需要在前面添加Global关键字，避免查询放大问题。

数据查询建议

- 建议查询指定分区

  通过指定分区字段会减少底层数据库扫描的文件数量，提升查询性能，实际经验：700个分区的千列大表，需要查询一个分区中有7000万数据，其他699个分区中无数据，虽然只有一个分区有数据，其他分区无数据，但是查询指定分区为百毫秒级性能，没有指定分区查询性能为1~2秒左右，性能相差20倍。

- 慎用final查询

  在查询语句的最后跟上final，通常是对于ReplacingMergeTree引擎，数据不能完全去重情况下，有些开发人员习惯写final关键字进行实时合并去重操作（merge-on-read），保证查询数据无重复数据。可以通过argMax函数或其他方式规避此问题。

数据修改

- 建议慎用delete、update的mutation操作

  标准SQL的更新、删除操作是同步的，即客户端要等服务端返回执行结果（通常是int值）；而ClickHouse的update、delete是通过异步方式实现的，当执行update语句时，服务端立即返回执行成功还是失败结果，但是实际上此时数据还没有修改完成，而是在后台排队等着进行真正的修改，可能会出现操作覆盖的情况，也无法保证操作的原子性。

  a. 业务场景要求有update、delete等操作，建议使用ReplacingMergeTree、CollapsingMergeTree、VersionedCollapsingMergeTree引擎，使用方式参见：**https://clickhouse.tech/docs/zh/engines/table-engines/mergetree-family/collapsingmergetree/**。

- 建议少或不增删数据列

  业务提前规划列个数，如果将来有更多列要使用，可以规划预留多列，避免在生产系统跑业务过程中进行大量的alter table modify列操作，导致不可以预知的性能、数据一致性问题。

- 对于批量数据清理，建议根据分区来操作：

  ALTER TABLE table_name DROP PARTITION partition_name;

- 禁止修改索引列

  对索引列的修改会导致现有索引失效，触发重建索引，期间查询数据不准确。

  如果业务场景必须修改索引列，推荐用ReplacingMergeTree引擎建表，使用数据写入+去重引擎代替数据更新场景：**https://clickhouse.tech/docs/zh/engines/table-engines/mergetree-family/collapsingmergetree/**。

数据 merge

建议谨慎执行optimize操作，Optimize一般会对表做重写操作，建议在业务压力小时候进行操作，否则对IO/MEM/CPU资源有较大消耗，导致业务查询变慢或不可用。

1.7.4 ClickHouse 数据库应用开发

在ClickHouse的使用过程中，由于使用不规范的方式访问和查询，导致业务失败的情况时有发生。此外，偶尔也会发生因为网络闪断等导致连接和查询失败的情况。

MRS提供了ClickHouse的样例代码工程，旨在提供连接重试机制和规范化用户连接和查询的方法，从而减少业务失败的风险，提升系统的稳定性和可靠性。

本样例代码工程包含了连接、查询和插入相关规则和建议，以及相关的代码示例，可以帮助客户更好地理解和实践这些方法。通过使用本代码样例，客户可以有效地降低业务失败的概率，提升用户体验和业务质量。

操作步骤

步骤1 先获取clickhouse-example样例代码工程。

代码获取地址：**https://github.com/huaweicloud/huaweicloud-mrs-example/blob/mrs-3.1.2/src/clickhouse-examples/**。

步骤2 在样例工程“conf”目录下有一个“clickhouse-example.properties”配置文件，其中各项的配置的作用如下所示：

```
#连接节点或Balancer的ip列表，ip之间用逗号隔开
loadBalancerIPList=
#是否需要开启ssl,如果取值为true，则loadBalancerHttpsPort必填
sslUsed=true
#端口号
loadBalancerHttpPort=
loadBalancerHttpsPort=
#ClickHouse安全模式开关，安全模式集群时该参数固定为true。
CLICKHOUSE_SECURITY_ENABLED=true
#连接的用户名
user=
#连接的用户的密码
password=
#集群名称
clusterName=
#数据库名称
databaseName=
#表名称
tableName=
```



```
#一个批次写入的条数
batchRows=10000
#写入数据的总批次
batchNum=10
#ip:port。安全模式下https端口，普通模式下http端口
clickhouse_dataSource_ip_list=
#ip:tcp port
native_dataSource_ip_list=ip:port,ip:port,ip:port
```

步骤3 在Demo.java有三种连接JDBC的样例：节点的JDBC连接、banlancer的JDBC连接和tcp端口的banlancer的JDBC连接。

步骤4 Demo提供了createDatabase、createTable、insertData和queryData的样例。

----结束

规则

- 大批量少频次的插入。

  内容要求：ClickHouse的每次数据插入都会生成一到多个part文件，如果data part过多则会导致merge压力变大，甚至出现服务异常影响数据插入。建议一次插入10万行，每秒不超过1次插入。

- 一次只插入一个分区内的数据。

  内容要求：如果数据属于不同的分区，则每次插入，不同分区的数据会独立生成part文件，导致part总数量膨胀。甚至写入报错“Merges are processing significantly slower than inserts”。一批次写入的数据，对应的分区数太多。ClickHouse建表之后insert batch时，会对不同的分区创建一个目录。如果一个batch里面的数据对应了过多的分区，那么一次insert就会生成较多的分区目录，后台merge线程处理速度跟不上分区增加的速度，社区规格是每秒不超过一个数据目录。

  具体的操作：确认一个batch的数据对应了多少个分区，insert的时候，尽量保证一个batch包含的分区数是1。

- 慎用delete、update操作。

  内容要求：建议使用CollapsingMergeTree、VersionedCollapsingMergeTree引擎或根据分区批量清理。

- ClickHouse需要写本地表。

  内容要求：连接balancer写入报错Request Entity Too Large。这是由于Nginx对http请求体大小有限制，而一次写入的数据量超过了这个限制。

  规避：修改Nginx配置项client_max_body_size为一个较大的值。

  解决：写本地表，不要通过balancer写入数据。

- 禁用实验特性

  内容要求：ClickHouse实验特性可能存在设计或功能缺陷，不具备商用能力。如果在生产环境上使用实验特性，会给生产环境带来数据准确性、集群稳定性等多个方面的系统风险。

- JDBC攒批入库禁用函数

  内容要求：使用jdbc攒批方式写数据到ClickHouse，对ClickHouse函数（例如：时间函数now()）会解析成String类型，而数据库里是**DateTime**类型，导致类型不匹配，数据入库异常。

  解决：在代码中生成时间，并生成字段传入或者在ClickHouse中修改表结构，给对应字段默认值。

建议

- 查询增加重试机制

  clickhouse-example.properties的配置文件的loadBalancerIPList可以配置多个ip，在二次样例代码中已经实现从第一个ip开始连接查询，查询失败时，继续连接下一个ip进行查询。

- 每个应用配置的loadBalancerIPList顺序不要一致，以免对balancer ip产生访问热点

  例如应用一配置loadBalancerIPList=ip1，ip2，ip3，应用二配置loadBalancerIPList=ip3，ip1，ip2。

- 根据连接方式选择端口

  普通集群默认开启8123端口，安全集群默认开启8443端口。

  端口号查看方式：在集群的Manager界面选择“集群 > 服务 > ClickHouse > 配置”。

  - 用于通过HTTP连接到ClickHouse server的端口默认为8123。
  - 用于通过HTTPS连接到ClickHouse server的端口默认为8443。
  - 用户客户端通过TCP连接到ClickHouse server的端口默认为9000。
  - 用户客户端通过TCP ssl连接到ClickHouse server的端口默认为9440。
  - ClickHouseBalancer的HTTP端口默认为21425。
  - ClickHouseBalancer的HTTPS端口默认为21426。

1.8 ClickHouse 数据库调优

1.8.1 ClickHouse 调优思路

ClickHouse的总体性能调优思路为性能瓶颈点分析、关键参数调整以及SQL调优。在调优过程中，需要综合系统资源、吞吐量、集群负载等各种因素来分析，定位性能问题，设定调优目标，调优达到客户所需目标即可。

ClickHouse调优人员需要系统软件架构、软硬件配置、数据库架构原理及配置参数、并发控制、查询处理和数据库应用有广泛而深刻的理解和认识，才能在调优过程中找到关键瓶颈点，解决性能问题。

图 1-3 调优流程

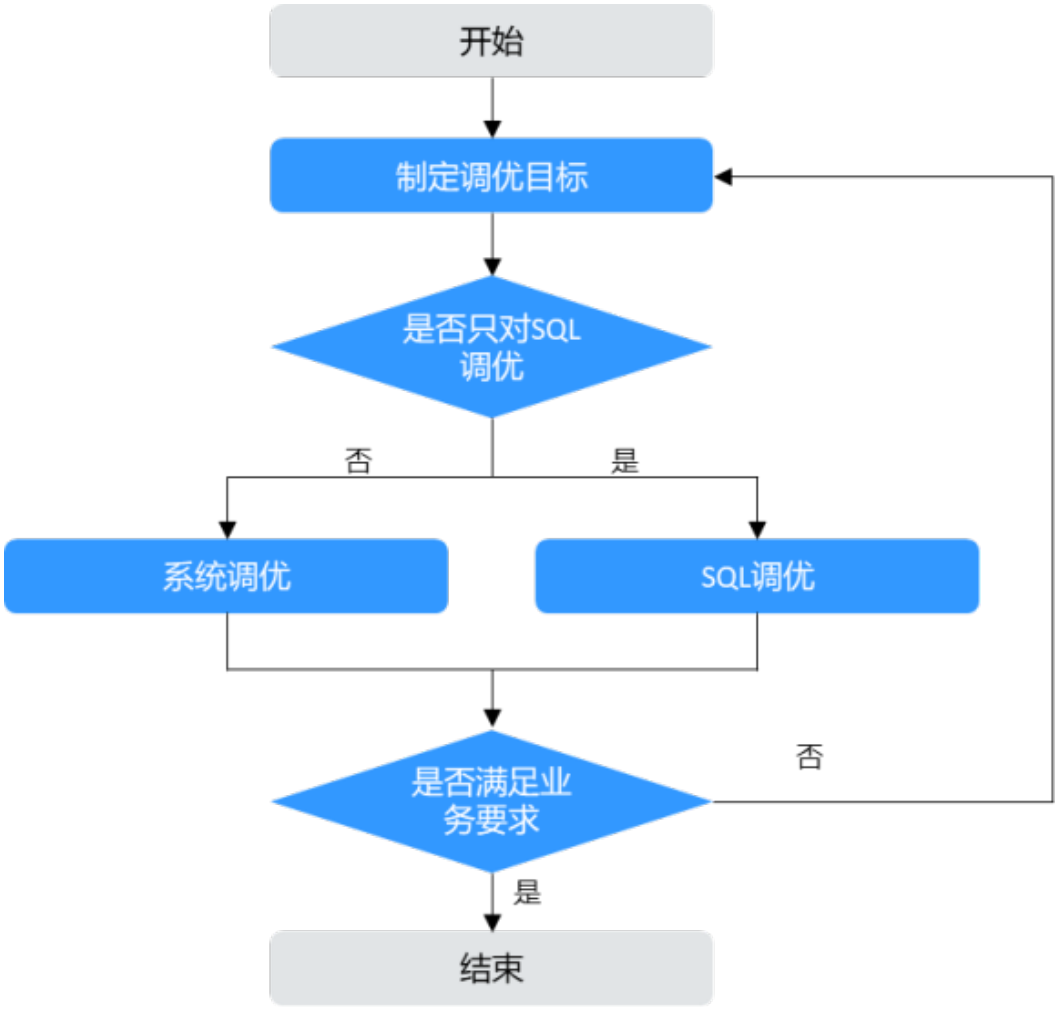


表 1-4 调优流程说明

流程	描述
系统调优	对OS操作系统级参数和数据库的调优，充分地利用主机的CPU、内存、I/O和网络资源，提升整个系统查询的吞吐量，同时数据库参数也调整到最优状态。
SQL调优	审视业务所用SQL语句是否存在可优化空间，包括： • 分析数据分布是否有倾斜，对于大表数据是否平均分布在各个shard。 • 分析建表语句，查看是否有建立分区、一级索引、二级索引、排序键是否指定等。 • 分析查询SQL是否使用了分区和索引，检查查询过滤条件比较频繁的列是否安排在建表时指定的索引及排序键的靠前位置。
数据库参数调优	通过调优数据参数，提升数据库性能，保障数据库稳定运行。

更多信息可参考ClickHouse社区文档相关调优内容**https://clickhouse.com/docs/en/intro**。

1.8.2 ClickHouse 系统调优

通过FusionInsight Manager查看主机上的CPU、内存、I/O和网络资源使用情况，确认这些资源是否已被充分利用，分以下几种情况：

- 每个节点资源占用都比较均匀

  通过观察资源在每个节点都使用比较均匀，说明系统资源使用比较正常，可以先不关注，可以去分析SQL语句是否有进一步优化的余地。

- 有个别节点资源占用比较高

  如果观察到个别节点占用资源较高，需要针对占用资源较高的节点分析，分析当前的SQL语句是什么原因导致部分节点占用比其他节点更多资源，是计算还是数据存储倾斜导致，或者是软件bug导致。

- 每个节点资源占用都比较高

  如果集群所有节点资源占用都比较高，说明集群整体比较忙，需要单独确认需要调优的SQL语句，单独调优。如果SQL也无调优余地，集群资源达到瓶颈，需要通过扩容来提升查询性能，达到调优目标。

1.8.3 ClickHouse SQL 调优

规则

1. 合理使用数据表的分区字段和索引字段。

   MergeTree引擎，数据是以分区目录的形式进行组织存储的，在进行的数据查询时，使用分区可以有效跳过无用的数据文件，减少数据的读取。

   MergeTree引擎会根据索引字段进行数据排序，并且根据index_granularity的配置生成稀疏索引。根据索引字段查询，能快速过滤数据，减少数据的读取，大大提升查询性能。

2. 不要用**select ***，只查询需要的字段，减少机器负载，提升查询性能。

   OLAP分析场景，一张大宽表通常能有几百上千列，选择其中少数的几列做维度列、指标列计算。匹配这种场景下，ClickHouse的数据也是按照列存储的。如果使用**select ***，会大大加重系统的压力。

3. 通过limit限制查询返回的数据量，节省计算资源、减少网络开销。

   如果返回的数据量过大，客户端有可能出现内存溢出等服务异常。

   对于前端使用ClickHouse的场景，如果要查询的数据量比较大，建议每次可适当地进行分页查询返回数据，以减少查询数据量对网络带宽和计算资源的占用。

   【不做limit限制】

   ```
   SELECT dict_value
   FROM zeus.did_mapping
   ```

   ```
   Showed first 10000.

   10002340 rows in set. Elapsed: 1.124 sec. Processed 10.00 million rows, 190.10 MB (8.90 million rows/s., 169.10 MB/s.)
   ```

   耗时：1.124

   【做limit限制】

   ```
   SELECT dict_value
   FROM zeus.did_mapping
   LIMIT 10
   ```

   ```
   10 rows in set. Elapsed: 0.002 sec.
   ```


耗时：0.002

```
SELECT dict_value
FROM zeus.did_mapping
LIMIT 10

┌─dict_value─────────────────────────┐
│ 0012f9f3-3183-497b-839b-174adb45199f │
│ 002625ac-c1a1-47a3-9e6e-6f31f4b7a7c7 │
│ 007db765-8dc7-46ac-a7c2-067dbd5b9611 │
│ 009721be-3e9f-4137-84b5-0dbdd3f2cd52 │
│ 00f1fc5a-2194-4927-88f6-00288ca6fcf9 │
│ 010f8c3f-2049-450b-8edc-af70f5ac89b0 │
│ 0151d5a3-22b7-4bee-886b-d00129526001 │
│ 015b3c69-4fbb-4175-a313-eb56dfd4f38c │
│ 017d4a43-1957-4057-8e63-87de66d0fbb8 │
└────────────────────────────────────┘

10 rows in set. Elapsed: 0.002 sec.
```

4. join查询时小表在右。

   两表JOIN时，会将右表数据加载到内存中，再根据右表数据遍历左表做匹配，将小表放在右边，减少匹配查询的次数。根据使用的情况，大表join小表的性能比小表join大表的性能有数量级的提升。

   【大表在左小表在右】

   ```
   SELECT count(a.id)
   FROM
   (
   SELECT id
   FROM mytable
   WHERE id < 100000000
   ) AS a
   INNER JOIN
   (
   SELECT id
   FROM mytable
   WHERE id < 1000000
   ) AS b ON a.id = b.id;
   ```

   耗时：0.145 sec。

   【大表在右小表在左】

   ```
   SELECT count(a.id)
   FROM
   (
   SELECT id
   FROM mytable
   WHERE id < 1000000
   ) AS a
   INNER JOIN
   (
   SELECT id
   FROM mytable
   WHERE id < 100000000
   ) AS b ON a.id = b.id;
   ```

   耗时：0.996 sec。

5. ClickHouse不支持limit下推，SQL生成时需要优化，以免SQL性能受影响。

   【错误示例】

   ```
   select did from (select did from tableA) limit 10;
   ```

   【正确示例】

   ```
   select did from (select did from tableA limit 10);
   ```


6. 基于大宽表做数据分析，尽量不要使用大表join大表的操作。

   ClickHouse分布式join的性能较差，建议在模型侧将数据聚合成大宽表再导入ClickHouse。

   【两表join查询】

```
SELECT
col1,
col2
FROM
(
SELECT
t1.col1 AS col1,
t2.col2 AS col2
FROM
(
SELECT
did,
col1
FROM table1
WHERE cc_pt_d = '2020-03-30'
) AS t1
LEFT JOIN
(
SELECT
did AS did_v2,
col2
FROM table2
WHERE pt_d = '2020-03-30'
) AS t2 ON t2.did_v2 = t1.did
) AS t
GROUP BY
col1,
col2
LIMIT 10;
耗时: 40秒。
```

   【大宽表查询】

```
SELECT
col1,
col2
FROM
table1
GROUP BY
col1,
col2
LIMIT 10;
耗时: 8秒。
```

建议

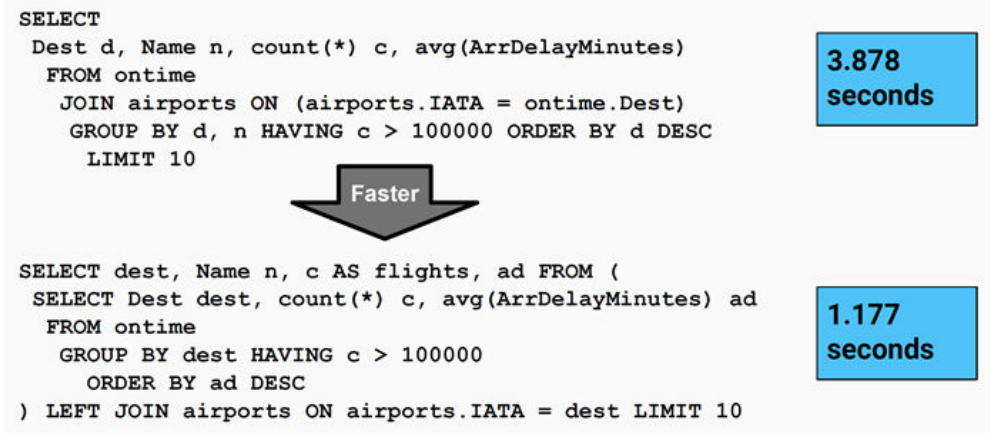
1. 明确数据查询的范围，增加条件过滤和查询的数据周期过滤，缩小数据查询范围。

   【示例】

```
SELECT uniqCombined(did) from pp.scene_model where pt_d <‘2020-11-10’ and pt_d >
‘2020-11-03’;
```

2. 在分组、join等操前做数据过滤，减少计算的数据量。

   【效果对比】



3. 用PREWHERE替代WHERE，优先过滤数据，加速查询。

   PREWHERE相对于WHERE在执行时的区别：首先只读取PREWHERE表达式所指定的列，根据条件做数据过滤，再根据过滤后的数据读取其他列。这通常会减少磁盘读取数据的压力。

   PREWHERE只支持*MergeTree系列的表。系统配置**optimize_move_to_prewhere**默认开启，将WHERE转成PREWHERE，可以根据自己的业务场景调整这个配置。

   查询语句中同时有PREWHERE和WHERE，在这种情况下，PREWHERE先于WHERE执行。

4. 合理配置最大并发数。

   Clickhouse快是因为采用了并行处理机制，即使一个查询，默认也会用服务器一半的CPU去执行，所以ClickHouse对高并发查询的场景支持的不够。

   官方默认的最大并发数是100，可以根据实际场景调整并发配置，实际使用中并发数配置的是150，建议不超过200。

5. 部署负载均衡组件，查询基于负载均衡组件进行，避免单点查询压力太大影响性能。

   ClickHouse支持连接集群中的任意节点查询，如果查询集中到一台节点，可能会导致该节点的压力过大并且可靠性不高。建议使用ClickHouseBalancer或者其他负载均衡服务，均衡查询负载，提升可靠性。

6. 用近似去重（uniqCombined、uniq）替代精确去重。

   ClickHouse提供多种近似去重算法，通过count_distinct_implementation配置，支持将countDistinct语法转成所配置的近似算法。查询性能有数量级的提升。

   近似算法的误差一般在1%以内。在数据准确度要求不高，比如趋势分析等，建议使用近似去重提升用户体验。

   【使用精确去重查询】

```
SELECT countDistinct(dict_value)
FROM zeus.did_mapping

┌─uniqExact(dict_value)─┐
│              10002340 │
└───────────────────────┘

1 rows in set. Elapsed: 1.280 sec. Processed 10.00 million rows, 190.10 MB (7.81 million rows/s., 148.49 MB/s.)
```

   耗时：1.280秒。

   【使用近似查询】

```
SELECT uniq(dict_value)
FROM zeus.did_mapping

┌─uniq(dict_value)─┐
│         10046324 │
└──────────────────┘

1 rows in set. Elapsed: 0.061 sec. Processed 10.00 million rows, 190.10 MB (162.85 million rows/s., 3.10 GB/s.)
```

耗时：0.061秒。

7. 对于字符串类型的字段做复杂计算，建议先编码成整数类型，以提升计算性能。

   【 字符编码前，32字节的String类型字段did 】

```
CREATE TABLE default.Test_String ON Cluster default_cluster
(
  `EventDate` DateTime,
  `did`       String,
  `UserID`    UInt32,
  `ver`       UInt16
)
ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/default/Test_String', '{replica}')
PARTITION BY toYYYYMM(EventDate)
ORDER BY (EventDate, intHash32(UserID))
SETTINGS index_granularity = 8192;
select count(distinct did) from dws_wallet_xxx_mlb_ds;
执行耗时：142秒。
```

   【 字符编码后，将32位长String转码成int类型 】

```
CREATE TABLE default.Test_Int ON Cluster default_cluster
(
  `EventDate` DateTime,
  `did`       UInt32,
  `UserID`    UInt32,
  `ver`       UInt16
)
ENGINE = ReplicatedMergeTree('/clickhouse/tables/{shard}/default/Test_Int', '{replica}')
PARTITION BY toYYYYMM(EventDate)
ORDER BY (EventDate, intHash32(UserID))
SETTINGS index_granularity = 8192;
select count(distinct did_int) from dws_wallesst_xxx_mlb_ds;
执行耗时：34秒。
```

8. 高基数（大于10W）字段（int类型），使用bitmap做精确去重。

   【 countDistinct做精确去重 】

```
select count(distinct did_int) from dws_wallet_xxx_mlb_ds;
执行耗时：34秒。
```

   【 countDistinct做精确去重 】

```
select groupBitmapMergeState(arrayReduce('groupBitmapState', [toUInt64(did)])) as user1 from
t_r_309;
执行耗时：8秒。
```

9. 使用物化视图加速查询。

   对于查询方式比较固定的场景，建议使用物化视图，提前做好数据聚合，相对于查询明细表，性能有数量级的提升。

   【 物化视图创建 】

   明细表、物化视图创建参见 【 建议 】 物化视图创建参考。

   【 明细表插入数据 】

```
INSERT INTO counter SELECT
toDateTime('2019-01-01 00:00:00') + toInt64(number / 10) AS when,
(number % 10) + 1 AS device,
((device * 3) + (number / 10000)) + ((rand() % 53) * 0.1) AS value
FROM system.numbers
LIMIT 100000000;
```

   【 查询明细表 】


```
SELECT
    device,
    count(*) AS count,
    max(value) AS max,
    min(value) AS min,
    avg(value) AS avg
FROM counter
GROUP BY device
ORDER BY device ASC

┌─device─┬─count─┬──────max─┬──min─┬──────────────avg─┐
│      1 │ 10000000 │ 10008.164 │  3.008 │ 5005.6002815747115 │
│      2 │ 10000000 │ 10011.185 │ 6.0251 │  5008.600215326505 │
│      3 │ 10000000 │ 10014.112 │ 9.0512 │  5011.599968641471 │
│      4 │ 10000000 │ 10017.152 │ 12.0163 │   5014.59998214844 │
│      5 │ 10000000 │ 10020.165 │ 15.1274 │  5017.600102860719 │
│      6 │ 10000000 │ 10023.101 │ 18.0385 │  5020.600059550402 │
│      7 │ 10000000 │  10026.19 │ 21.0416 │  5023.599433079225 │
│      8 │ 10000000 │ 10029.195 │ 24.0457 │  5026.600215895193 │
│      9 │ 10000000 │ 10032.197 │ 27.0668 │  5029.600555278349 │
│     10 │ 10000000 │ 10035.196 │ 30.0029 │  5032.600203040115 │
└────────┴──────┴──────────┴──────┴─────────────────┘

10 rows in set. Elapsed: 0.211 sec. Processed 100.00 million rows, 800.00 MB (474.79 million rows/s., 3.80 GB/s.)
```

【查询物化视图】

```
SELECT
    device,
    sum(count) AS count,
    maxMerge(max_value_state) AS max,
    minMerge(min_value_state) AS min,
    avgMerge(avg_value_state) AS avg
FROM counter_daily
GROUP BY device
ORDER BY device ASC

┌─device─┬─count─┬──────max─┬──min─┬──────────────avg─┐
│      1 │ 10000000 │ 10008.164 │  3.008 │ 5005.6002815747115 │
│      2 │ 10000000 │ 10011.185 │ 6.0251 │  5008.600215326504 │
│      3 │ 10000000 │ 10014.112 │ 9.0512 │  5011.599968641471 │
│      4 │ 10000000 │ 10017.152 │ 12.0163 │   5014.59998214844 │
│      5 │ 10000000 │ 10020.165 │ 15.1274 │  5017.600102860719 │
│      6 │ 10000000 │ 10023.101 │ 18.0385 │  5020.600059550402 │
│      7 │ 10000000 │  10026.19 │ 21.0416 │  5023.599433079225 │
│      8 │ 10000000 │ 10029.195 │ 24.0457 │  5026.600215895193 │
│      9 │ 10000000 │ 10032.197 │ 27.0668 │  5029.600555278349 │
│     10 │ 10000000 │ 10035.196 │ 30.0029 │  5032.600203040115 │
└────────┴──────┴──────────┴──────┴─────────────────┘

10 rows in set. Elapsed: 0.002 sec. Processed 2.11 thousand rows, 194.74 KB (1.05 million rows/s., 96.96 MB/s.)
```

【效果对比】

使用物化视图后，遍历的数据量从1亿下降到2000，耗时从0.211秒下降到0.002秒，性能提升100倍。

10. 使用bitmap做跨表预估计算。

【场景】

用户画像，用户数预估：计算t_r_309和t_r_308 join后，did字段的基数。

【表join示例】

```
SELECT countDistinct(a.did)
FROM
(
SELECT DISTINCT did
FROM t_r_309
) AS a
INNER JOIN
(
SELECT DISTINCT did
FROM t_r_308
) AS b ON a.did = b.did;
```

【bitmap实现示例】

```
SELECT bitmapAndCardinality(user1, user2)
FROM
(
SELECT
1 AS join_id,
groupBitmapMergeState(arrayReduce('groupBitmapState', [toUInt32(did)])) AS user1
FROM t_r_309
) AS a
INNER JOIN
(
```

```
SELECT
1 AS join_id,
groupBitmapMergeState(arrayReduce('groupBitmapState', [toUInt32(did)])) AS user2
FROM t_r_308
) AS b ON a.join_id = b.join_id;
```

【效果对比】

多张表join后计算，随着join数越多，时延越大，基本在几十秒以上。使用bitmap计算预估，耗时在3秒以内。

11. 使用GLOBAL JOIN/IN替换普通的JOIN。

    ClickHouse基于分布式表的查询会转换成所有分片的本地表的操作，再汇总结果。实际使用中，join和global join的执行逻辑差别很大，建议使用global join做分布式表查询。

    【场景说明】

    a. 查询的集群有N个分片（shard）
    b. A_all是分布式表，对应的本地表是A_local
    c. B_all是分布式表，对应的本地表是B_local

    【分布式表直接join示例】

    ```
    SELECT * FROM A_all AS t1 JOIN B_all AS t2 ON t1.id = t2.id;
    ```

    执行逻辑如下：

    a. 在发起查询的节点，将查询分发到所有分片，转成A_all Join B_local。
    b. 在收到a中每个请求的分片，再将请求分发到所有分片，转成A_local Join B_local。
    c. 可以看到，分布式表的join操作，存在查询放大的问题。

    【分布式表global join示例】

    ```
    SELECT * FROM A_all AS t1 GLOBAL JOIN B_all AS t2 ON t1.id = t2.id;
    ```

    执行逻辑如下：

    a. 在查询发起的节点，查询B_all的所有数据到本地的缓存表T中，并将T分发到所有节点。
    b. 查询发起的节点，将本地缓存表T分发到所有分片。
    c. 每个分片执行A_local join T。
    d. 在收到a中每个请求的分片，再将请求分发到所有分片，转成A_local Join B_local。

    【效果对比】

    可以看到，使用GLOBAL关键字后，查询的放大减少了很多。不过，由于需要将右表汇总再分发到所有机器，如果右表的数据量很大，需要考虑机器的内存，避免内存溢出。

12. 数据压缩算法的选择，建议使用默认的lz4压缩算法。

    ClickHouse提供了两种数据压缩方式供选择：LZ4和ZSTD。

    默认的LZ4压缩方式，会提供更快的执行效率，但是同时，要付出较多的磁盘容量占用的代价。

13. ReplacingMergeTree表引擎数据查询，需要先做数据去重合并提升性能。

    如果使用去重引擎进行数据查询，且使用argMax函数和final关键字，会导致整个查询性能较差，需要提前对重复数据做合并去重optimize操作，查询时候直接查询不需要使用argMax函数和final关键字，提升查询性能。

1.8.4 ClickHouse 参数调优实践

表 1-5 ClickHouse 参数调优汇总

参数名	参数描述	默认值	建议值	是否需要重启生效
max_memory_usage_for_all_queries	单台服务器上所有查询的内存使用量，默认没有限制。建议根据机器的总内存，预留一部分空间，防止内存不够导致服务或者机器异常。	0	机器总内存的80%	否
max_memory_usage	单个查询在单台服务器的能使用的最大内存。	10G	50GB	否（新版本可通过多租户方式配置）
max_bytes_before_external_group_by	确定了在GROUP BY中启动将临时数据转存到磁盘上的内存阈值。默认值为0表示这项功能将被禁用。一般：设置为max_memory_usage/2。	0	25GB	否
max_execution_time	单次查询耗时的最长时间，单位为秒。默认没有限制。	0	300	否
max_threads	执行请求的最大线程数。默认情况下是按照机器CPU核数自动确定的。单并发情况下线程数越大越好（该值要小于CPU核数），多并发情况建议设置为CPU核数/2的值。	CPU核数/2	64	否
max_result_rows	限制返回结果行数，默认为0不限制。	0	100000	否
distributed_product_mode	默认SQL中的子查询不允许使用分布式表，修改为local表示将子查询中对分布式表的查询转换为对应的本地表。	deny	根据场景定： deny/ local/ global/ allow	否

参数名	参数描述	默认值	建议值	是否需要重启生效
background_pool_size	后台用于merge的线程池大小。	16	• 4u16G：cpu核数 • 8u32G：cpu核数 • 16u64G及以上：CPU内核数 * 2	否
log_queries	system.query_log表的开关。默认值为0，不存在该表。修改为1，系统会自动创建system.query_log表，并记录每次query的日志信息。	0	1	否
skip_unavailable_shards	当通过分布式表查询时，遇到无效的shard是否跳过。默认值为0表示不跳过，抛异常。设置值为1表示跳过无效shard。	0	建议使用默认值。异常时，调整为1，提供有损服务。	否
max_bytes_before_external_sort	如果没有足够的内存，可以使用该参数来设置外部排序（在磁盘中创建一些临时文件）。默认为0表示禁用外部排序功能，当内存不够时直接抛错，设置了该值order by可以正常完成，但是速度非常慢。	0	25GB	否
keep_alive_timeout	服务端与客户端保持长连接的时长，单位为秒。	10	600	否
max_concurrent_queries	最大支持的查询并发。	100	150	否
session_timeout_ms	ClickHouse服务和ZooKeeper保持的会话时长，超过该时间ZooKeeper还收不到Clickhouse的心跳信息，会将与Clickhouse的session断开。	3000	120000	否

参数名	参数描述	默认值	建议值	是否需要重启生效
max_server_memory_usage_to_ram_ratio	ClickHosueServer默认可使用的系统最大内存比例，小数表示，如0.9表示系统内存的90%。配置不合理可能出现节点OOM导致业务受损。	0.8	• 4u16G：0.6 • 8u32G：0.7 • 16u64G：0.8 • 32u128G及以上：0.9	是

1.9 ClickHouse 数据库运维

1.9.1 ClickHouse 日志管理

1. 日志级别、日志文件大小、日志文件数目的修改设置。
   - ClickHouse支持日志级别的动态调整。

     登录FusionInsight Manager界面，访问“集群 > 服务 > ClickHouse > 配置 > 全部配置 > ClickHouseServer > 日志 > logger.level”，可进行日志级别动态调整。日志级别优先级从低到高分别是trace、debug、information、warning、error、fatal，程序会打印高于或等于所设置级别的日志，设置的日志等级越低，打印出来的日志就越详细。
   - ClickHouse支持日志文件大小和文件数目的调整。

     登录FusionInsight Manager界面，访问“集群 > 服务 > ClickHouse > 配置 > 全部配置 > ClickHouseServer > 日志”，可修改ClickHouseServer审计日志和运行日志的文件大小和文件数目。
   - ClickHouse支持ClickHouseBalancer日志文件大小和文件数目的调整。

     登录FusionInsight Manager界面，访问“集群 > 服务 > ClickHouse > 配置 > 全部配置 > ClickHouseBalancer > 日志”，可修改ClickHouseBalancer日志文件的大小和文件数目。
2. 支持日志在线检索和日志收集。
   - 支持在线检索ClickHouse日志内容。

     登录FusionInsight Manager界面，访问“运维 > 日志 > 在线检索”，在“服务”中选择“ClickHouse”，“检索内容”填写日志检索关键字，通过“检索”在线检索ClickHouse日志内容。
   - 支持ClickHouse日志内容收集。

     登录FusionInsight Manager界面，访问“运维 > 日志 > 下载”，在“服务”中选择“ClickHouse”，“主机”中选择主机节点或默认所有主机节点，通过“下载”收集ClickHouse对应的日志文件。

1.9.2 ClickHouse 日志管理规则

日志路径

- ClickHouse相关日志的默认存储路径为：“${BIGDATA_LOG_HOME}/clickhouse”。
- ClickHouseServer运行相关日志：“/var/log/Bigdata/clickhouse/clickhouseServer/ *.log”。
- ClickHouseBalancer运行日志：“/var/log/Bigdata/clickhouse/balance/*.log”。
- ClickHouseServer审计日志：“/var/log/Bigdata/audit/clickhouse/clickhouse-server-audit.log”。
- ClickHouse数据迁移日志：“/var/log/Bigdata/clickhouse/migration/${task_name}/clickhouse-copier_{timestamp}_{processId}/copier.log”。

日志归档规则

- ClickHouse日志启动了自动压缩归档功能，缺省情况下，当日志大小超过100MB的时（此日志文件大小可进行配置），会自动压缩。
- 压缩后的日志文件名规则为：“*<原有日志名>*.[*编号*].gz”。
- 默认最多保留最近的10个压缩文件，压缩文件保留个数可以在Manager界面中配置。

1.9.3 ClickHouse 日志详细信息

日志类型	日志文件名	描述
ClickHouse相关日志	/var/log/Bigdata/clickhouse/clickhouseServer/clickhouse-server.err.log	ClickHouseServer服务运行错误日志文件路径。
	/var/log/Bigdata/clickhouse/clickhouseServer/checkService.log	ClickHouseServer服务运行关键日志文件路径。
	/var/log/Bigdata/clickhouse/clickhouseServer/clickhouse-server.log	
	/var/log/Bigdata/clickhouse/clickhouseServer/ugsync.log	用户角色同步工具打印日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/prestart.log	ClickHouse预启动日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/start.log	ClickHouse启动日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/checkServiceHealthCheck.log	ClickHouse健康检查日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/checkugsync.log	用户角色同步检查日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/checkDisk.log	ClickHouse磁盘检测日志文件路径。

日志类型	日志文件名	描述
	/var/log/Bigdata/clickhouse/clickhouseServer/backup.log	ClickHouse在Manager上执行备份恢复操作的日志文件路径。
	/var/log/Bigdata/clickhouse/clickhouseServer/stop.log	ClickHouse停止日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/postinstall.log	ClickHouse的postinstall.sh脚本调用日志。
	/var/log/Bigdata/clickhouse/balance/start.log	ClickHouseBalancer服务启动日志文件路径。
	/var/log/Bigdata/clickhouse/balance/error.log	ClickHouseBalancer服务运行错误日志文件路径。
	/var/log/Bigdata/clickhouse/balance/access_http.log	ClickHouseBalancer服务运行http日志文件路径。
	/var/log/Bigdata/clickhouse/balance/access_tcp.log	ClickHouseBalancer服务运行tcp日志文件路径。
	/var/log/Bigdata/clickhouse/balance/checkService.log	ClickHouseBalancer服务检查日志。
	/var/log/Bigdata/clickhouse/balance/postinstall.log	ClickHouseBalancer的postinstall.sh脚本调用日志。
	/var/log/Bigdata/clickhouse/balance/prestart.log	ClickHouseBalancer服务预启动日志文件路径。
	/var/log/Bigdata/clickhouse/balance/stop.log	ClickHouseBalancer服务关闭日志文件路径。
	/var/log/Bigdata/clickhouse/clickhouseServer/auth.log	ClickHouse服务认证日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/cleanService.log	重装实例异常产生的记录日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/offline_shard_table_manager.log	ClickHouse入服/退服日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/traffic_control.log	ClickHouse主备容灾流量控制日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/clickhouse_migrate_metadata.log	ClickHouse元数据搬迁日志。

日志类型	日志文件名	描述
	/var/log/Bigdata/clickhouse/clickhouseServer/clickhouse_migrate_data.log	ClickHouse业务数据搬迁日志。
	/var/log/Bigdata/clickhouse/clickhouseServer/changePassword.log	ClickHouse修改用户密码日志。
数据迁移日志	/var/log/Bigdata/clickhouse/migration/*数据迁移任务名*/clickhouse-copier_*{timestamp}_{processId}*/copier.log	参考使用ClickHouse数据迁移工具，使用迁移工具时产生的运行日志。
	/var/log/Bigdata/clickhouse/migration/*数据迁移任务名*/clickhouse-copier_*{timestamp}_{processId}*/copier.err.log	参考使用ClickHouse数据迁移工具，使用迁移工具时产生的错误日志。
	/var/log/Bigdata/tomcat/clickhouse/auto_balance/*数据迁移任务名*/balance_manager.log	参考使用ClickHouse数据迁移工具，勾选一键均衡产生的运行日志。
clickhouse-tomcat日志	/var/log/Bigdata/tomcat/clickhouse/web_clickhouse.log	ClickHouse自定义UI运行日志。
	/var/log/Bigdata/tomcat/audit/clickhouse/clickhouse_web_audit.log	clickhouse的数据迁移审计日志。
ClickHouse审计日志	/var/log/Bigdata/audit/clickhouse/clickhouse-server-audit.log	ClickHouse的审计日志文件路径。

1.9.4 表运维

1.9.4.1 TTL 变更

场景1：TTL周期由小变大方案：

方案1：新建一张TTL时间为最新时间的表结构相同但名不同的表，把原表的数据导入新表，交换表名字；

方案2：业务代码中异步下发CK的修改TTL语句，下发之后业务代码不需要等待执行结果

1）类似在shell中，nohup sh xx.sh & --xx.sh中为修改TTL语句：alter table default.test_auto modify ttl EventDate + toIntervalMonth(2);

2）TTL放在代码流程中的最后一步执行，类似DDL表结构变更语句在TTL修改之前执行；

场景2：TTL周期由大变小方案：

方案1：新建一张ttl时间为最新需要修改时间TTL属性的表，表结构相同但名不同的表，把原表的数据导入到新表，交换表名字；

方案2：配置加上延迟物化参数，修改表TTL为最新时间，具体步骤如下：

1）在SQL级配置参数并修改TTL：alter table default.test_auto modify ttl EventDate + toIntervalMonth(2) SETTINGS materialize_ttl_after_modify=0;

2）删除过期数据，直接删除过期数据分区：alter table default.test_auto drop partation xxx; --多个分区逐一删除

2 Doris 应用开发规范

2.1 Doris 建表规范

该章节主要介绍创建Doris表时需遵循的规则和建议。

Doris 建表规则

- 在创建Doris表指定分桶buckets时，每个桶的数据大小应保持在100MB~3GB之间，单分区中最大分桶数量不超过5000。
- 表数据超过5亿条以上必须设置分区分桶策略。
- 表的分桶列不要设置太多，一般情况下设置1或2个列即可，同时需要兼顾数据分布均匀和查询吞吐均衡。
  - 数据均匀是为了避免某些桶的数据存在倾斜影响数据均衡和查询效率。
  - 查询吞吐利用查询SQL的分桶剪裁优化避免了全桶扫描，以提升查询性能。
  - 分桶列的选取：优先考虑数据较为均匀且常用于查询条件的列作为分桶列。

    可使用以下方法分析是否会导致数据倾斜：

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

    命令执行后查看各个分组的数据条数是否相差不大，如果相差超过2/3或1/2，则需要重新选择分桶字段。
- 2千万以内数据禁止使用动态分区。动态分区会自动创建分区，而小表用户关注不到，会创建出大量不使用的分区分桶。
- 创建表时，排序键key不能太多，一般建议3~5个；太多key会导致数据写入较慢，影响数据导入性能。
- 不使用Auto Bucket，需按照已有的数据量来进行分区分桶，能更好的提升导入及查询性能。Auto Bucket会造成Tablet数量过多，最终导致有大量的小文件。
- 创建表时的副本数必须至少为2，默认是3，禁止使用单副本。
- 没有聚合函数列的表不应该被创建为AGGREGATE表。
- 创建主键表时需保持主键的列唯一，不建议将所有列都设置为主键列，且主键表需设置value列。主键表不建议用于数据去重场景。

Doris 建表建议

- 单表物化视图不能超过6个，物化视图不建议嵌套，不建议数据写入时通过物化视图进行重型聚合和Join计算等ETL任务。
- 对于有大量历史分区数据，但是历史数据比较少，或者数据不均衡，或者数据查询概率较小的情况，可以创建历史分区（比如年分区，月分区），将所有历史数据放到对应分区里。

  创建历史分区方式为：**FROM ("***2000-01-01***") TO ("***2022-01-01***") INTERVAL 1 YEAR**
- 1千万~2亿以内数据为了方便可以不设置分区（Doris内部有一个默认分区），直接用分桶策略即可。
- 如果分桶字段存在30%以上的数据倾斜，则禁止使用Hash分桶策略，改为使用Random分桶策略，相关命令为：

  Create table ... **DISTRIBUTED BY RANDOM BUCKETS** *10* ...
- 建表时第一个字段一定是最常查询使用的列，默认有前缀索引快速查询能力，选取最常查询且高基数的列作为前缀索引，默认将一行数据的前36个字节作为这行数据的前缀索引（varchar类型的列只能匹配20个字节，并且会匹配不足36个字节截断前缀索引）。
- 超过亿级别的数据，如果有模糊匹配或者等值/in条件，可以使用倒排索引（Doris 2.x版本开始支持）或者Bloomfilter。如果是低基数列的正交查询适合使用bitmap索引（bitmap索引的基数在10000~100000之间效果较好）。
- 建表时需要提前规划将来要使用的字段个数，可以多预留几十个字段，类型包括整型、字符型等。避免将来字段不够使用，需要较高代价临时去添加字段。

2.2 Doris 数据变更规范

该章节主要介绍Doris数据变更时需遵循的规则和建议。

Doris 数据变更规则

- 应用程序不能直接使用**delete**或者**update**语句变更数据，可以使用CDC的**upsert**方式来实现。
- 不建议业务高峰期或在表上频繁地进行加减字段，建议在业务前期规划建表时预留将来要使用的字段。如果必须添加或删除字段，及修改字段类型和注释，需在业务低峰期，停止相关表的写入和修改业务后，通过重建表方式实现以上操作：
  a. 新建一个表，该表结构和需进行增删改字段的表结构相同。在新建表中增加需要添加的新字段、删除不需要的字段、或修改需改变类型的字段。
  b. 选取指定字段数据插入到新创建的表中：

     INSERT INTO *新创建的表* **SELECT** *指定的字段* **FROM** *已存在需要修改列的表*;

     说明

     如果表数据量较大，可按时间过滤分批次将数据导入到新表，减小CPU或MEM内存瞬时冲高占用问题，影响查询业务，命令为：

     insert into *tab1* **select** *col* **from** *tab* **where** *date <= xx*;
  c. 交换两个表的名称：

     ALTER TABLE *[db.]tbl1* **REPLACE WITH TABLE** *tbl2* **[PROPERTIES('swap' = 'true')];**

- 对于部分查询，可能执行时间比较长，查询比较耗费内存和CPU等资源，需要在SQL或user级别设置查询超时时间参数：query_timeout

Doris 数据变更建议

执行特殊的大SQL操作时，可以使用类似**SELECT /*+ SET_VAR(query_timeout = xxx*/ from table**通过Hint方式设置Session会话变量，不要设置全局的系统变量。

2.3 Doris 命名规范

该章节主要介绍创建Doris数据库或表时，数据库名或表名需遵循的规则和建议。

Doris 命名规则

数据库字符集需指定UTF-8，并且只支持UTF-8。

Doris 命名建议

- 数据库名称统一使用小写方式，中间使用下划线（ _ ）分隔，长度为62字节以内。
- Doris表名称大小写敏感，统一使用小写方式，中间使用下划线（ _ ）分隔，长度为64字节以内。

2.4 Doris 数据查询规范

该章节主要介绍Doris数据查询时需遵循的规则和建议。

Doris 数据查询规则

- 在数据查询业务代码中建议查询失败时进行重试，再次下发查询。
- **in**中常量枚举值超过1000后，必须修改为子查询。
- 禁止使用REST API（Statement Execution Action）执行大量SQL查询，该接口仅用于集群维护。
- query查询条件返回结果超过5万条，则使用JDBC Catalog或者OUTFILE方式导出查询数据，否则FE上大量数据传输将占用FE资源，影响集群稳定性。
  - 如果是交互式查询，建议使用分页方式（offset limit）导出数据，分页命令为Order by。
  - 如果数据导出提供给第三方使用，建议使用outfile或者export方式
- 2个以上大于3亿的表JOIN使用Colocation Join。
- 亿级别大表禁止使用**select ***查询数据，查询时需明确要查询的字段。
  - 使用SQL Block方式禁止**select ***操作。
  - 如果是高并发点查询，建议开启行存储（Doris 2.x版本支持），并且使用PreparedStatement查询。
- 亿级以上表数据查询必须设置分区分桶条件。
- 禁止对分区表执行全分区数据扫描操作。

Doris 数据查询建议

- 一次**insert into select**数据超过1亿条后，建议拆分为多个**insert into select**语句执行，分成多个批次来执行。
- 不要使用OR作为JOIN条件。
- 不建议频繁的数据delete修改，将要删除的数据攒批，偶尔进行批量删除，且需要带上条件，提升系统稳定性和删除效率。
- 大量数据排序（5亿以上）后返回部分数据，建议先减少数据范围再执行排序，否则大量排序会影响性能。例如：

  将**from table order by datatime desc limit** *10*优化为**from table where datatime='***2023-10-20***' order by datatime desc limit** *10*。
- 查询任务性能调优参数**parallel_fragment_exec_instance_num**使用注意事项：

  此参数是session级别设置，表示可并发执行的fragment数量，对CPU消耗较大，因此一般情况下不需要设置此参数。如果需要设置此参数来加速查询性能，必须遵循以下规则：
  - 切勿设置该参数为全局生效，禁止使用**set global**方式进行设置。
  - 设置参数值建议为偶数2或4（最大值不要超过单节点CPU核数的一半）。
  - 设置此参数值时需要观察CPU使用率，CPU使用率小于50%时方可考虑设置。
  - 如果查询SQL是**insert into select**大数据量的方式，不建议设置此参数。

2.5 Doris 数据导入规范

该章节主要介绍Doris数据导入规范。

Doris 数据导入建议

- 禁止高频执行**update**、**delete**或**truncate**操作，推荐几分钟执行一次，使用**delete**必须设置分区或主键列条件。
- 禁止使用**INSERT INTO** *tbl1* **VALUES (“1”), (“a”);**方式导入数据，少量少次写可以，多量多频次时需使用Doris提供的StreamLoad、BrokerLoad、SparkLoad或者Flink Connector方式。
- 在Flink实时写入数据到Doris的场景下，CheckPoint设置的时间需要考虑每批次数据量，如果每批次数据太小会造成大量小文件，推荐值为60s。
- 建议不使用**insert values**作为数据写入的主要方式，批量数据导入推荐使用StreamLoad、BrokerLoad或SparkLoad。
- 使用INSERT INTO WITH LABEL *XXX* SELECT方式进行数据导入，如果有下游依赖或查询，需要先查看导入的数据是否为可见状态。

  具体查看方法：通过**show load where label='***xxx***'** SQL命令查询当前INSERT任务状态（status）是否为“VISIBLE”，如果为“VISIBLE”导入的数据才可见。
- Streamload数据导入适合10 GB以内的数据量、Brokerload适合百GB以内数据，数据过大时可考虑使用SparkLoad。
- 禁止使用Doris的Routine Load进行导入数据操作，推荐使用Flink查询Kafka数据再写入Doris，更容易控制导入数据单批次数据量，避免大量小文件产生。如果确实已经使用了Routine Load进行导数，在没整改前请配置FE “max_tolerable_backend_down_num”参数值为“1”，以提升导入数据可靠性。

- 建议低频攒批导入数据，平均单表导入批次间隔需大于30s，推荐间隔60s，一次导入1000~100000行数据。

2.6 Doris UDF 开发规范

本章节主要介绍开发Doris UDF程序时应遵循的规则和建议。

Doris UDF 开发规则

- UDF中方法调用必须是线程安全的。
- UDF实现中禁止读取外部大文件到内存中，如果文件过大可能会导致内存耗尽。
- 需避免大量递归调用，否则容易造成栈溢出或oom。
- 需避免不断创建对象或数组，否则容易造成内存耗尽。
- Java UDF应该捕获和处理可能发生的异常，不能将异常给服务处理，以避免程序出现未知异常。可以使用try-catch块来处理异常，并在必要时记录异常信息。
- UDF中应避免定义静态集合类用于临时数据的存储，或查询外部数据存在较大对象，否则会导致内存占用过高。
- 应该避免类中**import**的包和服务侧包冲突，可通过**grep -lr** "*完全限定类名*"命令来检查冲突的Jar包。如果发生类名冲突，可通过完全限定类名方式来避免。

Doris UDF 开发建议

- 不要执行大量数据的复制操作，防止堆栈内存溢出。
- 应避免使用大量字符串拼接操作，否则会导致内存占用过高。
- Java UDF应该使用有意义的名称，以便其他开发人员能够轻松理解其用途。建议使用驼峰式命名法，并以UDF结尾，例如：MyFunctionUDF。
- Java UDF应该指定返回值的数据类型，并且必须具有返回值，返回值默认或异常时不要设置为NULL。建议使用基本数据类型或Java类作为返回值类型。

2.7 Doris 连接运行规范

连接Doris和运行Doris任务时需遵循的规范如下：

- 推荐使用ELB连接Doris，避免当连接的FE故障时，无法对外提供服务。
- 当Doris单实例或硬件故障时，新提交的任务能运行成功，但不能确保故障时正在运行的任务能执行成功。因此，需要用户连接Doris执行任务时进行失败重试，当任务遇到未知原因失败时，能保证重试新提交的任务能运行成功。

3 Flink 应用开发规范

3.1 Flink 开发规范概述

范围

本规范主要描述基于MRS-Flink组件进行湖仓一体、流批一体方案的设计与开发方面的规则。其主要包括以下方面的规范：

- 数据表设计
- 资源配置
- 性能调优
- 常见故障处理
- 常用参数配置

术语约定

本规范采用以下的术语描述：

- **规则**：编程时必须遵守的原则。
- **建议**：编程时必须加以考虑的原则。
- **说明**：对此规则或建议进行的解释。
- **示例**：对此规则或建议给出示例。

适用范围

- 基于MRS-Flink数据存储进行数据存储、数据加工作业的设计、开发、测试和维护。
- 该设计开发规范是基于MRS 3.2.0及以后版本。
- 参数优化部分适配于MRS 3.2.0及以后版本。
- 该规范中与开源社区不一致的点，以本文档为准。

参考资料

Flink开源社区开发文档：https://nightlies.apache.org/flink/flink-docs-stable/。

3.2 FlinkSQL Connector 开发规范

3.2.1 FlinkSQL ClickHouse 表开发规则

提前在 ClickHouse 中创建表

Flink作业在ClickHouse中找不到对应表会报错，所以需提前在ClickHouse中创建好对应的表。

Flink 写 ClickHouse 不支持删除操作

由于不支持删除操作，Flink无法对ClickHouse的数据进行回撤。在Flink处理更新数据的时候产生的回撤流就无法在ClickHouse中执行，导致数据结果不对。

同时通过Flink CDC对接上游数据库写ClickHouse的场景也受限，上游数据库如果进行了物理操作，那么ClickHouse中数据无法进行同步删除。

3.2.2 FlinkSQL ClickHouse 表开发建议

配置多个 ClickHouseBalancer 实例 IP

配置多个ClickHouseBalancer实例IP可以避免ClickHouseBalancer实例单点故障。相关配置（with属性）如下：

```
'url' = 'jdbc:clickhouse://ClickHouseBalancer实例IP1:ClickHouseBalancer端口,ClickHouseBalancer实例IP2:ClickHouseBalancer端口/default',
```

Sink 表配置合适的攒批参数

攒批写参数：

Flink会将数据先放入内存，到达触发条件时再flush到数据库表中。

相关配置如下：

- sink.buffer-flush.max-rows：攒批写ClickHouse的行数，默认100。
- sink.buffer-flush.interval：攒批写入的间隔时间，默认1s。

两个条件只要有一个满足，就会触发一次sink，即到达触发条件时再flush到数据库表中。

- 示例1：60秒sink一次
  ```
  'sink.buffer-flush.max-rows' = '0',
  'sink.buffer-flush.interval' = '60s'
  ```
- 示例2：100条sink一次
  ```
  'sink.buffer-flush.max-rows' = '100',
  'sink.buffer-flush.interval' = '0s'
  ```
- 示例3：数据不sink
  ```
  'sink.buffer-flush.max-rows' = '0',
  'sink.buffer-flush.interval' = '0s'
  ```


配置去重需在 ClickHouse 中创建 ReplacingMergeTree 表

由于Flink写入ClickHouseBalancer无法保证同key数据写入同一个ClickHouseServer中，所以同key数据的合并需要依赖ClickHouse的ReplacingMergeTree引擎。

3.2.3 FlinkSQL Doris 数据表开发规则

提前在Doris中创建表：

Flink作业在Doris中找不到对应表会报错，所以需要提前在Doris中创建好对应的表。

Doris作为Sink表时需开启CheckPoint：

Flink作业在触发CheckPoint时才会往Doris表中写数据。

3.2.4 FlinkSQL Kafka 表开发规则

Kafka 作为 sink 表时必须指定“topic”配置项

【示例】向Kafka的“test_sink”主题插入一条消息：

```
CREATE TABLE KafkaSink(
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `age` INT
) WITH (
  'connector' = 'kafka',
  'topic' = 'test_sink',
  'properties.bootstrap.servers' = 'Kafka的Broker实例业务IP:Kafka端口号',
  'scan.startup.mode' = 'latest-offset',
  'value.format' = 'csv',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.kerberos.domain.name' = 'hadoop.系统域名
);
INSERT INTO KafkaSink (`user_id`, `user_name`, `age`)VALUES ('1', 'John Smith', 35);
```

Kafka 作为 source 表时必须指定“properties.group.id”配置项

【示例】以“testGroup”为用户组读取主题为“test_sink”的Kafka消息：

```
CREATE TABLE KafkaSource(
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `age` INT
) WITH (
  'connector' = 'kafka',
  'topic' = 'test_sink',
  'properties.bootstrap.servers' = 'Kafka的Broker实例业务IP:Kafka端口号',
  'scan.startup.mode' = 'latest-offset',
  'properties.group.id' = 'testGroup',
  'value.format' = 'csv',
  'properties.sasl.kerberos.service.name' = 'kafka',
  'properties.security.protocol' = 'SASL_PLAINTEXT',
  'properties.kerberos.domain.name' = 'hadoop.系统域名
);
SELECT * FROM KafkaSource;
```

不能同时设置“topic-pattern”和“topic”配置项

topic-pattern：主题模式，用于source表，可使用正则表达式的主题名称。

【示例】以下source表将订阅所有以“test-topic-”开头，单个数字结尾的主题消息：

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

3.2.5 FlinkSQL Kafka 表开发建议

Kafka 作为 source 表时应设置限流

本章节适用于MRS 3.3.0及以后版本。

防止上限超过流量峰值，导致作业异常带来不稳定因素。因此建议设置限流，限流上限应该为业务上线压测的峰值。

【示例】

```
#如下参数作用在每个并行度
'scan.records-per-second.limit' = '1000'
#真实的限流流量如下
min( parallelism * scan.records-per-second.limit，partitions num * scan.records-per-second.limit)
```

为保证数据准确性将同 key 数据写入 Kafka 的同一个分区

Flink写Kafka使用fixed策略，并在写入之前根据key进行Hash。

【示例】

```
CREATE TABLE kafka (
 f_sequence INT,
 f_sequence1 INT,
 f_sequence2 INT,
 f_sequence3 INT
 ) WITH (
 'connector' = 'kafka',
 'topic' = 'yxtest123',
 'properties.bootstrap.servers' = '192.168.0.104:9092',
 'properties.group.id' = 'testGroup1',
 'scan.startup.mode' = 'latest-offset',
 'format' = 'json',
 'sink.partitioner'='fixed'
 );

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

为提升 Kafka 消费速度可将 Kafka Source 并行度与 Topic 分区数保持一致

当Kafka Source并行度大于Topic分区数时，多余的并行度不能消费数据。

3.2.6 FlinkSQL HBase 数据表开发规则

提前在 HBase 中创建表

Flink作业在HBase中找不到对应表会报错，所以需要提前在HBase中创建好对应的表。

HBase 与 Flink 不在同一集群时只支持 Flink 和 HBase 均为普通模式集群的对接

当HBase与Flink为同一集群或互信的集群，支持FlinkServer对接HBase。

当HBase与Flink不在同一集群或不互信的集群，则只支持Flink和HBase均为普通模式集群的对接。

FlinkServer 对接 HBase 时需要配置 HBASE_CONF_DIR 参数

步骤1 以客户端安装用户登录安装客户端的节点，复制HBase的“/opt/client/HBase/hbase/conf/”目录下的所有配置文件至部署FlinkServer的所有节点的一个空目录，如“/tmp/client/HBase/hbase/conf/”。

修改FlinkServer节点上面配置文件目录及其上层目录属主为omm。

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

说明

- FlinkServer节点：
  登录Manager，选择“集群 > 服务 > Flink > 实例”，查看FlinkServer所在的“业务IP”。
- 若FlinkServer实例所在节点与包含HBase服务客户端的安装节点相同，则该节点不执行此步骤。

步骤2 登录Manager，选择“集群 > 服务 > Flink > 配置 > 全部配置”，搜索“HBASE_CONF_DIR”参数，在该参数的“值”中填写**步骤1**中复制了HBase配置文件的FlinkServer的目录，如“/tmp/client/HBase/hbase/conf/”。

说明

若FlinkServer实例所在节点与包含HBase服务客户端的安装节点相同，则在HBASE_CONF_DIR”参数的“值”填写HBase的“/opt/client/HBase/hbase/conf/”目录。

步骤3 填写完成后单击“保存”，确认修改配置后单击“确定”。

步骤4 单击“实例”，勾选所有FlinkServer实例，选择“更多 > 重启实例”，根据界面提示重启实例。

----结束

3.2.7 FlinkSQL HBase 数据表开发建议

客户端提交作业时通过 with 属性添加 HBase 配置信息

Flink客户端提交作业，如SQL client提交，在建表语句中添加如下配置：

表 3-1 Flink 作业 with 属性

配置	说明
'properties.hbase.rpc.protection' = 'authentication'	需和HBase服务端的配置一致。
'properties.zookeeper.znode.parent' = '/hbase'	多服务场景中，会存在hbase1，hbase2，需明确要访问的集群。

配置	说明
'properties.hbase.security.authorization' = 'true'	开启鉴权。
'properties.hbase.security.authentication' = 'kerberos'	开启Kerberos认证。

【示例】

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

开启异步 Lookup Join 提升维表 Join 性能

在HBase维表with中添加如下属性：

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

调大 Lookup Join 算子并行度提升维表 Join 性能

在HBase维表with中添加如下属性：

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

调大 Sink HBase 算子并行度提升写入性能

在HBase sink表with中添加如下属性：

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

3.2.8 FlinkSQL Elasticsearch 表开发规则

Flink支持1.12.2及以后版本，Elasticsearch支持7.10.2及以后版本。

安全模式的 Flink 对接普通模式的 Elasticsearch 集群需设置参数“es.security.indication”的值为“false”

- 安全模式的Flink集群支持对接安全模式和普通模式的Elasticsearch集群。

  当安全模式的Flink集群对接普通模式的Elasticsearch集群时需设置如下参数：

  a. 登录FusionInsight Manager页面，选择“集群 > 服务 > Flink > 配置 > 全部配置”，搜索参数“es.security.indication”，并将FlinkResource角色和FlinkServer角色下该参数的值配置为“false” 。

  b. 重启Flink服务，在“概览”页签，选择“更多 > 重启服务”等待Flink服务重启成功。

- 普通模式的Flink集群支持对接普通模式的Elasticsearch集群。

3.2.9 FlinkSQL Elasticsearch 表开发建议

FlinkSQL Elasticsearch 作业，参数配置如下：

表 3-2 Flink 作业 With 属性

参数	是否必选	数据类型	描述
connector	必选	String	指定要使用的连接器，如elasticsearch-7，即连接到Elasticsearch 7.x或更高版本的集群。
hosts	必选	String	要连接的一台或多台Elasticsearch主机地址。 例如：'http://*10.10.10.10*:24100;http://*10.10.10.10*:24100'
index	必选	String	Elasticsearch中每条记录的索引。可以是一个静态索引（如 'myIndex' ）或一个动态索引（如 'index-{log_ts|*yyyy-MM-dd*}' ）。
document-id.key-delimiter	可选	String	复合键的分隔符，默认为“_”。若指定为“$”，则文档ID为“KEY1$KEY2$KEY3”。
username	可选	String	用于连接Elasticsearch实例的用户名。
password	可选	String	用于连接Elasticsearch实例的用户名密码。若配置了username，则必须配置为非空字符串。
failure-handler	可选	String	对Elasticsearch请求失败情况的失败处理策略，有效策略如下： ● fail（默认值）：如果请求失败并因此导致作业失败，则发生异常。 ● ignore：忽略失败并放弃请求。 ● retry-rejected：重新添加由于队列容量饱和而失败的请求。 ● 自定义类名称：使用ActionRequestFailureHandler的子类进行失败处理。
sink.flush-on-checkpoint	可选	Boolean	● true：确保在进行CheckPoint时读出缓冲区中的数据，默认值为“true”。 ● false：Sink将不对请求的一致性提供保证。在进行CheckPoint时，对于进行中的请求，Sink将不再等待Elasticsearch的执行完成确认。

参数	是否必选	数据类型	描述
sink.bulk-flush.max-actions	可选	Integer	每个批量请求的最大缓冲操作数，默认值为“1000”，可设置为“0”禁用该功能。
sink.bulk-flush.max-size	可选	MemorySize	每个批量请求的缓冲操作在内存中的最大值，默认值为“2MB”，单位必须为MB，可设置为“0”禁用该功能。
sink.bulk-flush.interval	可选	Duration	缓冲操作的间隔时间，默认值为“1s”，可设置为“0”禁用该功能。
sink.bulk-flush.backoff.strategy	可选	String	指定在由于临时请求错误导致任何flush操作失败时如何执行重试。有效策略为： ● DISABLED（默认值）：不执行重试，即第一次请求错误后失败。 ● CONSTANT：常量回退，即每次回退等待时间相同。 ● EXPONENTIAL：指数回退，即每次回退等待时间指数递增。
sink.bulk-flush.backoff.max-retries	可选	Integer	最大回退重试次数。
sink.bulk-flush.backoff.delay	可选	Duration	每次退避重试之间的延迟，退避策略如下： ● CONSTANT：每次重试之间的延迟。 ● EXPONENTIAL：初始的延迟。
connection.path-prefix	可选	String	添加到每个REST通信中的前缀字符串，例如：'/v1'。
format	可选	String	Elasticsearch连接器支持的指定格式，默认值为“json”。

3.2.10 FlinkSQL JDBC 表开发规则

提前在对应数据库中创建表

- JDBC作为sink表时，需要提前在对应数据库（如MySQL）中创建好用于接收数据的空表。
- JDBC作为维表时，需要提前在对应数据库（如MySQL）中创建好维度表。

3.2.11 FlinkSQL JDBC 表开发建议

Flink SQL 与 JDBC 数据类型对应关系

参考**表3-3**开发Flink SQL作业。

表 3-3 Flink SQL 与 JDBC 数据类型对应关系

Flink SQL数据类型	MySQL数据类型	Oracle数据类型	PostgreSQL数据类型	SQL Server数据类型
BOOLEAN	BOOLEAN TINYINT(1)	-	BOOLEAN	BIT
TINYINT	TINYINT	-	-	TINYINT
SMALLINT	SMALLINT TINYINT UNSIGNED	-	SMALLINT INT2 SMALLSERIAL SERIAL2	SMALLINT
INT	INT MEDIUMINT SMALLINT UNSIGNED	-	INTEGER SERIAL	INT
BIGINT	BIGINT INT UNSIGNED	-	BIGINT BIGSERIAL	BIGINT
FLOAT	FLOAT	BINARY_FLOAT	REAL FLOAT4	REAL
DOUBLE	DOUBLE DOUBLE PRECISION	BINARY_DOUBLE	FLOAT8 DOUBLE PRECISION	FLOAT
STRING	CHAR(n) VARCHAR(n) TEXT	CHAR(n) VARCHAR(n) CLOB	CHAR(n) CHARACTER(n) VARCHAR(n) CHARACTER VARYING(n) TEXT	CHAR(n) NCHAR(n) VARCHAR(n) NVARCHAR(n) TEXT NTEXT
BYTES	BINARY VARBINARY BLOB	RAW(s) BLOB	BYTEA	BINARY(n) VARBINARY(n)
ARRAY	-	-	ARRAY	-
DATE	DATE	DATE	DATE	DATE
TIME [(p)] [WITHOUT TIMEZONE]	TIME [(p)]	DATE	TIME [(p)] [WITHOUT TIMEZONE]	TIME(0)

Flink SQL数据类型	MySQL数据类型	Oracle数据类型	PostgreSQL数据类型	SQL Server数据类型
TIMESTAMP [(p)] [WITHOUT TIMEZONE]	DATETIME [(p)]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	DATETIME DATETIME2
DECIMAL(20, 0)	BIGINT UNSIGNED	-	-	-
DECIMAL(p, s)	NUMERIC(p, s) DECIMAL(p, s)	SMALLINT FLOAT(s) DOUBLE PRECISION REAL NUMBER(p, s)	NUMERIC(p, s) DECIMAL(p, s)	DECIMAL(p, s)

3.2.12 FlinkSQL DWS 表开发规则

提前在 DWS 中创建表

若开发FlinkSQL DWS表作业，需要在DWS中创建数据表。

由于Flink作业在DWS中找不到对应表会报错，所以需要提前在DWS中创建好用于接收数据的空表。

3.2.13 FlinkSQL DWS 表开发建议

FlinkSQL DWS 表开发建议

开发FlinkSQL DWS作业，DWS可以作为源表、结果表和维表。

开发FlinkSQL DWS表请参考**Flink SQL概述**。

3.2.14 FlinkSQL Redis 表开发规则

Flink Redis 作业参数规范

Flink Redis作业参数配置规范如下表所示。

表 3-4 Flink Redis 作业参数规范

配置项	是否必选	类型	描述
zSetScoreColumn	可选	String	Redis作为维表时，ZSet格式score字段对应的列名。

配置项	是否必选	类型	描述
hashKeyColumn	可选	String	Hash格式，Hash字段对应的列名。
host	必选	String	Redis集群连接IP，为Redis集群的实例IP（业务平面）。
port	必选	String	端口为对应的Redis实例的端口。 Redis实例的端口计算方式为：22400+该实例的ID-1。 实例ID可以通过在FusionInsight Manager中选择“集群 > 服务 > Redis > Redis管理”，单击Redis集群名称查看。 例如Redis集群内角色R1对应的Redis实例的端口为22400+1-1=22400。
separator	可选	String	Redis作为维表时，value中的字段分割符，示例：“(,)”、“(\u200b)”。
key-ttl-mode	可选	String	Redis数据过期策略： ● no-ttl：数据不过期。 ● expire-msec： 指定多长时间之后数据过期，以毫秒为单位。 ● expire-at-date：到指定时间数据过期，精确到秒。 ● expire-at-timestamp：到指定时间数据过期，精确到毫秒。
key-ttl	可选	String	配置“key-ttl-mode”参数为非“no-ttl”时需设置该值，该值不需要带单位。
isSSLMode	可选	String	是否开启SSL模式： ● true：开启SSL模式。 ● false：不开启SSL模式。
keyPrefix	可选	String	Redis key的前缀。

3.2.15 FlinkSQL Redis 表开发建议

Sink 表设置合适的批写参数

- sink.batch.max-size：开启批写Redis并设置批写数量（正整数），单位：条。“-1”表示不开启批写Redis。
  - 开启该功能可提升大数据场景下性能表现，但不适合对实时性要求过高的场景，建议批写数量不超过30000。

  - 开启该参数需同步开启CheckPoint。
- sink.flush-buffer.timeout：开启批写Redis后，可按照指定时间将队列里面的数据刷新到Redis。单位：ms。
- 示例1
```
# 开启批写Redis并设置批写数量为5，开启批写需同步开启CheckPoint
'sink.batch.max-size' = '5'
# 数据在缓冲区的最大等待时间为1s
'sink.flush-buffer.timeout' = '1000'
```
- 示例2
```
# '-1'表示不开启批写Redis
'sink.batch.max-size' = '-1'
'sink.flush-buffer.timeout' = '1000'
```

3.2.16 FlinkSQL Hive 表开发规则

提前在 Hive 中创建表

- Flink作业在Hive中找不到对应表会报错，所以需要提前在Hive客户端创建好对应的表。
- FlinkServer对接Hive使用对接MetaStore的方式，故需要Hive开启MetaStore功能。

说明

查看Hive是否开启MetaStore功能：

登录FusionInsight Manager，选择“集群 > 服务 > Hive > 配置 > 全部配置”，搜索参数“hive-ext.dlcatalog.metastore.client.enable”，查看该参数的值是否为“false”。

若为“false”，表示当前Hive已开启MetaStore功能；若为“true”，表示当前Hive已开启LakeFormation，未使用MetaStore功能。

3.2.17 FlinkSQL Hive 表开发建议

FlinkServer 对接 Hive 时创建集群连接

步骤1 以具有FlinkServer管理员权限的用户访问FlinkServer WebUI界面，选择“系统管理 > 集群连接管理”，进入集群连接管理页面。

步骤2 单击“创建集群连接”，在弹出的页面中填写集群连接信息，单击“测试”，测试连接成功后单击“确定”，完成集群连接创建。

例如集群连接名称为“flink_hive”，创建hive Catalog时配置集群连接名称“cluster.name”为“flink_hive”。
```
CREATE CATALOG myhive WITH (
  'type' = 'hive',
  'hive-version' = '3.1.0',
  'default-database' = 'default',
  'cluster.name' = 'flink_hive'
);
```

----结束

3.3 Flink on Hudi 开发规范

3.3.1 Flink 流式读 Hudi 表规则

Flink流式读Hudi表参数规范如下所示：

表 3-5 Flink 流式读 Hudi 表参数规范

参数名称	是否必填	参数描述	示例
Connector	必填	读取表类型。	hudi
Path	必填	表存储的路径。	根据实际情况填写
table.type	必填	Hudi表类型，默认值为COPY_ON_WRITE。	MERGE_ON_READ
hoodie.datasource.write.recordkey.field	必填	表的主键。	根据实际填写
write.precombine.field	必填	数据合并字段。	根据实际填写
read.tasks	选填	读Hudi表task并行度，默认值为4。	4
read.streaming.enabled	必填	• true：开启流式增量模式。 • false：批量读。	根据实际填写，流读场景下为true
read.streaming.start-commit	选填	指定‘yyyyMMddHHmmss’格式的起始commit（闭区间），默认从最新commit。	-
hoodie.datasource.write.keygenerator.type	选填	上游表主键生成类型。	COMPLEX
read.streaming.check-interval	选填	流读检测上游新提交的周期，默认值为1分钟。	5（流量大建议使用默认值）
read.end-commit	选填	• Stream增量消费，通过参数read.streaming.start-commit指定起始消费位置； • Batch增量消费，通过参数read.streaming.start-commit指定起始消费位置，通过参数read.end-commit指定结束消费位置（闭区间），即包含起始、结束的commit。默认到最新commit。	-

参数名称	是否必填	参数描述	示例
changelog.enabled	选填	是否写入changelog消息。默认值为false，CDC场景填写为true。	false

3.3.2 Flink 流式读 Hudi 表建议

设置合理的消费参数避免 File Not Found 问题

当下游消费Hudi过慢，上游写入端会把Hudi文件归档，导致File Not Found问题。优化建议如下：

- 调大read.tasks。
- 如果有限流则调大限流参数。
- 调大上游compaction、archive、clean参数。

3.3.3 Flink 流式写 Hudi 表规则

Flink 流式写 Hudi 表参数规范

Flink流式写Hudi表参数规范如下表所示。

表 3-6 Flink 流式写 Hudi 表参数规范

参数名称	是否必填	参数描述	建议值
Connector	必填	读取表类型。	hudi
Path	必填	表存储的路径。	根据实际填写
hoodie.datasource.write.recordkey.field	必填	表的主键。	根据实际填写
write.precombine.field	必填	数据合并字段。	根据实际填写
write.tasks	选填	写Hudi表task并行度，默认值为4。	4
index.bootstrap.enabled	选填	Flink采用的是内存索引，需要将数据的主键缓存到内存中，保证目标表的数据唯一，因此需要配置该值，否则会导致数据重复。默认值为FALSE。Bueckt索引时不配置该参数。	TRUE

参数名称	是否必填	参数描述	建议值
write.index_bootstrap.tasks	选填	index.bootstrap.enabled开启后有效，增加任务数提升启动速度。	4
index.state.ttl	选填	索引数据保存时长，默认值为0，表示永久不失效，可根据业务调整。	0
compaction.delta_commits	选填	MOR表Compaction计划触发条件。	200
compaction.async.enabled	必填	是否开启在线压缩。将compaction操作转移到sparksql运行，提升写性能。	FALSE
hive_sync.enable	选填	是否向Hive同步表信息。	True
hive_sync.metastore.uris	选填	Hivemeta uri信息。	根据实际填写
hive_sync.jdbc_url	选填	Hive jdbc链接。	根据实际填写
hive_sync.table	选填	Hive的表名。	根据实际填写
hive_sync.db	选填	Hive的数据库名，默认为default。	根据实际填写
hive_sync.support_timestamp	选填	是否支持时间戳。	True
changelog.enabled	选填	是否写入changelog消息。默认值为false，CDC场景填写为true。	false

表名必须满足 Hive 格式要求

- 表名必须以字母或下划线开头，不能以数字开头。
- 表名只能包含字母、数字、下划线。
- 表名长度不能超过128个字符。
- 表名中不能包含空格和特殊字符，如冒号、分号、斜杠等。
- 表名不区分大小写，但建议使用小写字母。
- Hive保留关键字不能作为表名，如select、from、where等。

【示例】

```
my_table、customer_info、sales_data
```

3.3.4 Flink 流式写 Hudi 表建议

- 使用SparkSQL统一建表。
- 推荐使用Spark异步任务对Hudi表进行Compaction。

- 表名必须以字母或下划线开头，不能以数字开头。
- 表名只能包含字母、数字、下划线。
- 表名长度不能超过128个字符。
- 表名中不能包含空格和特殊字符，如冒号、分号、斜杠等。
- 表名不区分大小写，但建议使用小写字母。
- Hive保留关键字不能作为表名，如select、from、where等。

3.3.5 Flink on Hudi 作业参数规则

Flink 作业参数配置规范

Flink作业参数配置规范如下表所示。

表 3-7 Flink 作业参数配置规范

参数名称	是否必填	参数描述	建议值
-c	必填	指定主类名。	根据实际情况而定
-ynm	必填	Flink Yarn作业名称。	根据实际情况而定
execution.checkpointing.interval	必填	Checkpoint触发间隔（毫秒），通过-yD添加，单位毫秒。	60000
execution.checkpointing.timeout	必填	Checkpoint超时时长，通过-yD添加，默认值为30min。	30min
parallelism.default	选填	作业并行度，例如join算子，通过-yD添加，默认值为1。	根据实际情况而定
table.exec.state.ttl	必填	Flink状态ttl（join ttl），通过-yD添加，默认值为0。	根据实际情况而定

Checkpoint 间隔时长大于 Checkpoint 执行时长

checkpoint执行时长视checkpoint的数据量相关，数据量越大实行耗时越大

Checkpoint 超时时长大于 Checkpoint 间隔时长

Checkpoint间隔时长是指多长时间触发一次Checkpoint操作，启动Checkpoint后执行时长超过Checkpoint超时时长会导致作业失败。

CDC 场景下 Hudi 读写表需要开启 Changelog

CDC场景下为保障Flink计算的准确，需要在Hudi表中保留+I、+U、-U、-D。所以同一个Hudi表在写入、流读时都需要开启Changelog。

3.3.6 Flink on Hudi 作业参数建议

Hudi 表作为 Source 表时建议设置限流

Hudi表作为Source表，防止上限超过流量峰值，导致作业出现异常带来不稳定因素，因此建议设置限流，限流上限应该为业务上线压测的峰值。

使用时需添加如下参数：

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

设置 execution.checkpointing.tolerable-failed-checkpoints

Flink On Hudi作业建议设置Checkpoint容忍次数多次，如100。

3.4 Flink 任务开发规范

3.4.1 Flink 任务开发规则

对有更新操作的数据流进行聚合计算时要注意数据准确性问题

在针对更新数据进行聚合需要选择合适的解决方案，否则聚合结果会是错误的。

例如：

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

- 第一批数据：[1,1,10],[2,1,11],[3,2,8]
  聚合结果：[1,21],[2,8]
- 第二批数据：[2,1,12] //对ID=2的记录进行更新。
  错误结果：[1,33],[2,8] //若是无法识别是对ID=2的数据进行了更新。
  聚合结果：[1,22],[2,8] //识别为更新操作可以得到正确结果。

对于如何识别是更新数据有三种方式：

- 通过状态后端解决
  通过状态后端存储所有原始数据，新来的数据根据状态来判断是否是更新操作，进而通过Flink聚合回撤机制实现聚合结果数据的更新。
  优点：可以解决聚合准确性问题，而且对用户友好，对数据没有要求。
  缺点：大数据量情况下状态后端存储的数据比较多。
- 通过CDC格式数据解决
  CDC格式数据是指更新操作记录中会同时包含更新前数据和更新后数据。通过更新前的内容来回撤掉之前的聚合结果，通过更新后的数据更新最新的计算结果。
  优点：不需要有大的状态后端存储，整体计算资源压力要小于基于状态后端的方案。

缺点：需要依赖于数据格式，常见的方式通过CDC采集工具，将数据采集到Kafka，然后Flink读Kafka数据进行计算。

- 通过changelog数据解决

  changelog与CDC格式的数据类似，只不过存储的方式不同，CDC格式数据会将更新前和更新后的数据在一行记录，而changelog数据会将更新数据拆分成两行，一行是对更新前数据的删除操作，一行是更新后的数据插入操作记录。Flink在计算的时候会将基于更新数据的聚合结果删除，再将基于更新后数据的计算结果插入。changelog可以基于Hudi表实现，基于CDC格式的数据可以转为changelog数据存储到Hudi的MOR表的log文件中，也可以基于状态后端生成Hudi的changelog数据。

  优点：可以基于湖存储实现更新数据聚合一致性保证。

  缺点：

  - Hudi的MOR表中仅在log文件中存在changelog数据，如果Flink作业计算延迟导致上游数据积压，而Hudi又清理了log文件，就会导致changelog丢失。针对这种情况需要保留版本数多一点，且给Flink作业合理的资源配置避免数据积压周期超过了清理周期。
  - 基于状态后端生成changelog也是依赖于状态后端的，状态后端通常是会配置TTL时间的，不会永久保留。这种场景下更新操作是任意更新，没有一定时间周期限制。例如更新近一个月的数据，TTL设置大于一个月即可；若更新全部数据，就需要设置TTL为永久，不适用于大表。
  - 目前changelog的MOR表，仅支持Flink引擎进行compaction处理，不支持Spark引擎。

3.4.2 Flink 任务开发建议

高可用性下考虑提高 Checkpoint 保存数

Checkpoint保存数默认是1，也就是只保存最新的Checkpoint的状态文件，当进行状态恢复时，如果最新的Checkpoint文件不可用（比如HDFS文件所有副本都损坏或者其他原因），那么状态恢复就会失败。如果设置Checkpoint保存数为2，即使最新的Checkpoint恢复失败，那么Flink会回滚到之前那一次Checkpoint的状态文件进行恢复。所以可以增加Checkpoint保存数。

【示例】配置Checkpoint文件保存数为2：

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

生产环境使用增量 Rocksdb 作为 State Backend

Flink提供了三种状态后端：MemoryStateBackend，FsStateBackend，和RocksDBStateBackend。

- MemoryStateBackend是将state存储在JobManager的Java堆上，每个状态的大小不能超过akka帧的大小，且总量不能超过JobManager的堆内存大小。所以只适合于本地开发调试，或状态大小有限的一些小状态的场景。
- FsStateBackend是文件系统状态后端，正常情况下将state存储在TaskManager堆内存中，当Checkpoint时将state存储在文件系统上，而JobManager内存中存储极少的元数据（高可用场景下存储在ZooKeeper）。因为文件系统的存储空间足够，适合于大状态，长窗口，或大键值状态的有状态处理任务，也适合于高可用方案。
- RocksDBStateBackend是内嵌数据库后端，正常情况下state存储在RocksDB数据库中，该数据库数据放在本地磁盘上，在Checkpoint时将state存储在配置的文件

系统上而JobManager内存中存储极少的元数据（高可用场景下存储在ZooKeeper），同时是唯一一个可以增量Checkpoint的状态后端，除了适合于FsStateBackend的场景，还适用于超大状态的场景。

表 3-8 Flink 状态后端

类别	MemoryStateBackend	FsStateBackend	RocksDBStateBackend
方式	Checkpoint数据直接返回给Master节点，不落盘	数据写入文件，将文件路径传给Master	数据写入文件，将文件路径传给Master
存储	堆内存	堆内存	Rocksdb（本地磁盘）
性能	相比最好（一般不用）	性能好	性能不好
缺点	数据量小、易丢失	容易OOM风险	需要读写、序列化、IO等耗时
是否支持增量	不支持	不支持	支持

【示例】配置RockDBStateBackend（flink-conf.yaml）：

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

使用 EXACTLY ONCE 流处理语义保证端到端的一致性

流处理语义有三种：EXACTLY ONCE、AT LEAST ONCE、AT MOST ONCE。

- AT MOST ONCE：无法保证数据处理的完整性，但性能相比最好。
- AT LEAST ONCE：可以保证数据处理的完整性，但无法保证数据处理的准确性，性能适中。
- EXACTLY ONCE：可以保证数据处理的准确性，但性能最差。

首先需要确认能否保证EXACTLY_ONCE（严格一次），因为端到端EXACTLY ONCE语义需要输入数据源的可回放（例如Kafka可回放数据），输出数据源的事务性（例如MySQL可原子性写入数据）。在无法满足这些条件的情况下，可以视情况将其降级为AT LEAST ONCE或者AT MOST ONCE。

- 在无法满足输入源的可回放时，只能保证AT MOST ONCE。
- 在无法满足输出目的的原子性写入时，只能保证AT LEAST ONCE。

【示例】API方式设置Exactly once语义：

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

【示例】资源文件方式设置Exactly once语义：

```
# checkpoint的语义
execution.checkpointing.mode: EXACTLY_ONCE
```


通过查看监控信息定位 Back Pressure 点

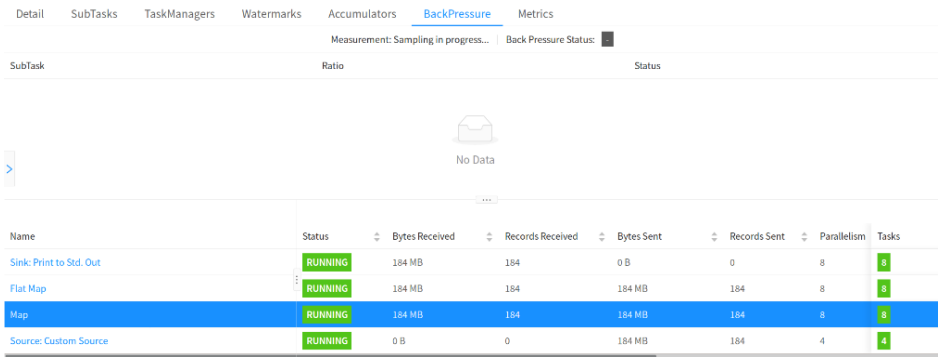
Flink提供了很多的监控指标，根据这些指标可以分析任务过程中的性能状况及瓶颈。

【示例】配置采样的样本数和时间间隔：

```
# 有效的反压结果被废弃并重新进行采样的时间，单位ms
web.backpressure.refresh-interval: 60000
# 用于确定反压采样的样本数
web.backpressure.num-samples: 100
# 用于确定反压采样的间隔时间，单位ms
web.backpressure.delay-between-samples: 50
```

可以在Job的Overview选项卡后面查看BackPressure，如下图表示采样进行中，默认情况下，大约需要5秒完成采样。

图 3-1 采样进行中



如下图显示“OK”表示没有反压，“HIGH”表示对应SubTask被反压。

图 3-2 无反压状态

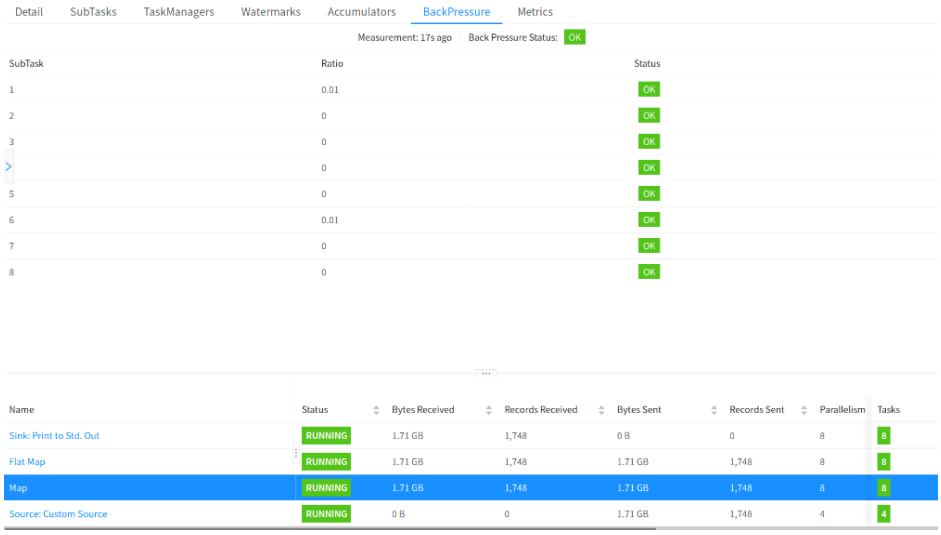
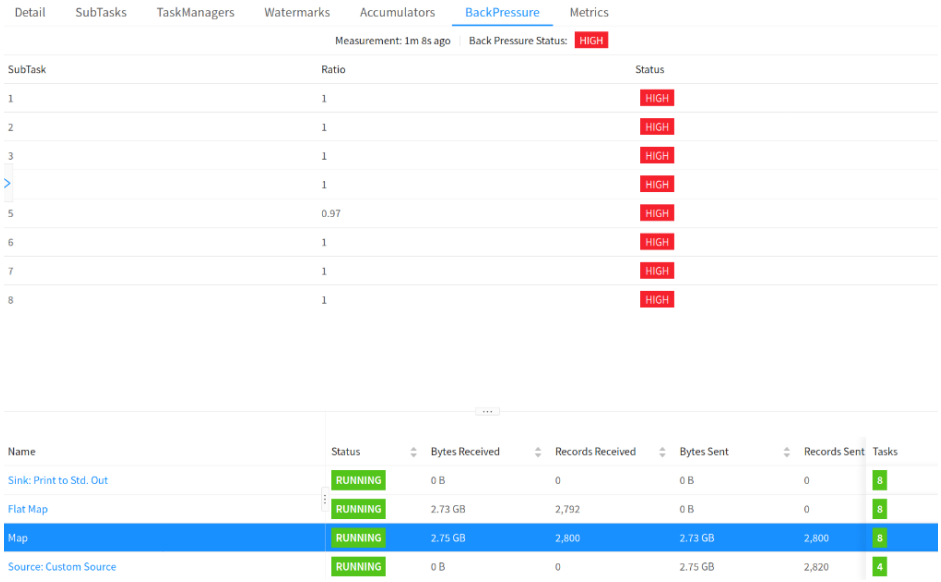


图 3-3 反压状态



使用 Hive SQL 时如果 Flink 语法不兼容则可切换 Hive 方言

当前Flink支持的SQL语法解析引擎有default和Hive两种，第一种为Flink原生SQL语言，第二种是Hive SQL语言。因为部分Hive语法的DDL和DML无法用Flink SQL运行，所以遇到这种SQL可直接切换成Hive的dialect。使用Hive dialect需要注意：

- Hive dialect只能用于操作Hive表，不能用于普通表。Hive方言应与HiveCatalog一起使用。
- 虽然所有Hive版本都支持相同的语法，但是是否有特定功能仍然取决于使用的Hive版本。例如仅在Hive-2.7.0或更高版本中支持更新数据库位置。
- Hive和Calcite具有不同的保留关键字。例如default在Calcite中是保留关键字，在Hive中是非保留关键字。所以在使用Hive dialect时，必须使用反引号（`）引用此类关键字，才能将其用作标识符。
- 在Hive中不能查询在Flink中创建的视图。

【示例】修改SQL解析为Hive语法（sql-submit-defaults.yaml）：

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

中小规模数据量维度表可以采用内存维度表（如 Hudi）

- 内存维度表：将维度数据加载到内存当中，每个TM都会加载全量的数据，在内存内实现数据点查关联。若数据量过大，需要给TM分配大的内存空间，否则容易导致作业异常。
- 外置维度表：将维度数据存在高速的K-V数据库中，通过远程的K-V查询实现点查关联，常用的开源K-V库有HBase。
- 状态维度表：将维度表数据当做流表，实时读入到流式作业当中，通过数据的回撤流能力实现维度更新和数据不对齐场景下的数据一致性保证。维度表保存时间比较长，当前Flink on Hudi能力可以针对Hudi作为维度表单独设置TTL时长。

表 3-9 维度表实现方式对比

维度	内存维度表（hive/hudi表）	外置维度表（HBase）	状态维度表
性能	非常高（毫秒内）	中（毫秒级）	高（毫秒内~毫秒级）
数据量	小，建议单个TM保持1GB以内	大，TB级	中，GB级
存储资源	内存消耗大，单个TM全量存储	外置存储，无存储资源消耗	各TM分散存储，内存+磁盘存储
时效性	周期性数据加载，时效低	相对高	高
关联数据结果	低	中	-

大数据量的维度表建议采用 HBase

数据量比较大，而且不要数据高一致的场景，可以采用HBase类的KV库提供维度表点查关联能力。

由于K-V库的数据需由另外的作业写入，与当前的Flink作业会存在一定的时差，容易导致当前Flink作业查询K-V库时不是最新的数据，且由于lookup查询不支持回撤，关联的结果存在一致性问题。

维度表要求高数据一致性采用流表作为维度表

基于Hudi作为维度source表，可以实现维度表单独设置TTL时长，不跟随作业的整体TTL时间进行数据老化，从而保证维度数据可以长期保存在状态后端中。而且基于流表作为维度表可以基于Flink回撤机制实现数据的一致性。

3.5 Flink SQL 逻辑开发规范

3.5.1 Flink SQL 逻辑开发规则

维表 lookup join 场景维度表个数不超过五个

Hudi维度表都在TM heap中，当维表过多时heap中保存的维表数据过多，TM会不断GC，导致作业性能下降。

【示例】lookup join维表数5个：

```
CREATE TABLE table1(id  int, param1 string) with(...);
CREATE TABLE table2(id  int, param2 string) with(...);
CREATE TABLE table3(id  int, param3 string) with(...);
CREATE TABLE table4(id  int, param4 string) with(...);
CREATE TABLE table5(id  int, param5 string) with(...);
CREATE TABLE orders (
    order_id    STRING,
    price       DECIMAL(32,2),
    currency    STRING,
```

```
    order_time  TIMESTAMP(3),
    WATERMARK FOR order_time AS order_time
) WITH (/* ... */);

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

多流 Join 场景流表个数不超过三个

当Join表过多时，状态后端压力太大会导致端到端时延增加。

【示例】实时Join维表数3个：

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

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

关联嵌套层级不超过三层

嵌套层级越多，回撤流的的数据量越大。

【示例】关联嵌套3层：

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

基于 Hudi 表的 lookup join 单表数据量不超过 1GB

Hudi维度表都在TM heap中，当维表过大时heap中保存的维表数据过多，TM会不断GC导致作业性能下降。

流流关联中不能加入批 Source 算子

流流关联中不能加入批Source算子，根据业务情况将该Source算子调整为维表算子。

3.5.2 Flink SQL 逻辑开发建议

在 aggregate 和 join 等操作前将数据过滤来减少计算的数据量

提前过滤可以减少在shuffle阶段前的数据量，减少网络IO，从而提升查询效率。

比如在表join前先过滤数据比在ON和WHERE时过滤可以有效较少join数据量。因为执行顺序从发生shuffle再filter变成了先发生filter再shuffle。

【示例】优化后将谓词条件A.userid>10提前到了子查询语句中，减少了shuffle的数据量：

- 优化前SQL：

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

- 优化后SQL：

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

慎用正则表达式函数 REGEXP

正则表达式是非常耗时的操作，对比加减乘除通常有百倍的性能开销，而且正则表达式在某些极端情况下可能会进入无限循环，导致作业阻塞。推荐首先使用LIKE。正则函数包括：

- REGEXP
- REGEXP_EXTRACT
- REGEXP_REPLACE

【示例】

- 使用正则表达式：

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

- 使用like模糊查询：

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

UDF 嵌套不可过长

多个UDF嵌套时表达式长度很长，Flink优化生成的代码超过64KB导致编译错误。建议UDF嵌套不超过6个。

【示例】UDF嵌套：

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

聚合函数中 case when 语法改写成 filter 语法

在聚合函数中，FILTER是更符合SQL标准用于过滤的语法，并且能获得更多的性能提升。FILTER是用于聚合函数的修饰符，用于限制聚合中使用的值。

【示例】在某些场景下需要从不同维度来统计UV，如Android中的UV，iPhone中的UV，Web中的UV和总UV，这时可能会使用如下CASE WHEN语法。

- 修改前：
  ```
  SELECT
  day,
  COUNT(DISTINCT user_id) AS total_uv,
  COUNT(DISTINCT CASE WHEN flag IN (android', "iphone'") THEN user_id ELSE NULL END) AS app_uv,
  COUNT(DISTINCT CASE WHEN flag IN(wap', 'other') THEN user_id ELSE NULL END) AS web_uv
  FROM T
  GROUP BY day
  ```
- 修改后：
  ```
  SELECT
  day,
  COUNT(DISTINCT user_id) AS total_uv,
  COUNT(DISTINCT user_id) FILTER (WHERE flag IN ('android', 'iphone')) AS app_uv,
  COUNT(DISTINCT user_id) FILTER(WHERE flag IN ('wap', 'other'))AS web_uv
  FROM T
  GROUP BY day
  ```

Flink SQL优化器可以识别相同的distinct key上的不同过滤器参数。例如示例中三个COUNT DISTINCT都在user_id列上。Flink可以只使用一个共享状态实例，而不是三个状态实例，以减少状态访问和状态大小，在某些工作负载下可以获得显著的性能提升。

拆分 distinct 聚合优化聚合中数据倾斜

通过两阶段聚合能消除常规的数据倾斜，但是处理distinct聚合时性能并不好。因为即使启动了两阶段聚合，distinct key也不能combine消除重复值，累加器中仍然包含所有的原始记录。

可以将不同的聚合（例如 COUNT(DISTINCT col)）分为两个级别：

第一次聚合由group key和额外的bucket key进行shuffle。bucket key是使用HASH_CODE(distinct_key) % BUCKET_NUM计算的，BUCKET_NUM默认为1024，可以通过table.optimizer.distinct-agg.split.bucket-num选项进行配置。

第二次聚合是由原始group key进行shuffle，并使用SUM聚合来自不同buckets的COUNT DISTINCT值。由于相同的distinct key将仅在同一bucket中计算，因此转换是等效的。bucket key充当附加group key的角色，以分担group key中热点的负担。bucket key使Job具有可伸缩性来解决不同聚合中的数据倾斜/热点。

【示例】

- 资源文件配置：
  ```
  table.optimizer.distinct-agg.split.enabled: true
  table.optimizer.distinct-agg.split.bucket-num: 1024
  ```

- 查询今天有多少唯一用户登录：

```
SELECT day, COUNT(DISTINCT user_id)
FROM T
GROUP BY day
```

- 自动改写查询：

```
SELECT day, SUM(cnt)
FROM(
    SELECT day, COUNT(DISTINCT user_id) as cnt
FROM T
GROUP BY day, MOD(HASH_CODE(user_id), 1024)
    )
GROUP BY day
```

多流 join 场景建议 join 字段设置为主键

如果join字段不为主键，会导致Flink shuffle task按照hash进行数据处理，导致在Flink中无法保序。同时状态后端中同一个join key字段会保留多份，join时会产生笛卡尔积。

比如A表字段为“id, field1”，B表字段为“id, field2”。A表和B表根据“id”进行join，A表有历史数据（1, a1），B表有历史数据（1, b1）。当A表发生变化（1, a1）>（1, a2），同时B表发生变化（1, b1）>（1, b2）时，join结果如下，且join结果的顺序无法保证。

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

- 优化前SQL：

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

- 优化后SQL：

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

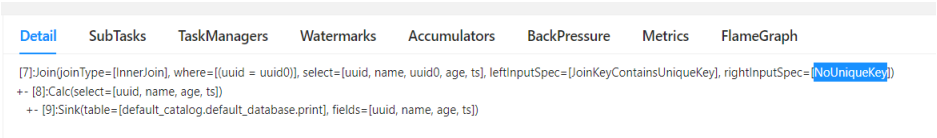
多表 join 场景且 join key 是联合主键时 select 字段要显示添加联合主键所有字段

如果不显示select联合主键所有字段，join算子会丢弃部分主键，导致join spec为NoUniqueKey。

- 优化前SQL：

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

图 3-4 join spec 为 NoUniqueKey

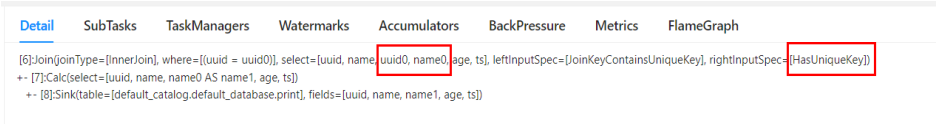


- 优化后SQL：

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

```
) with (
  'connector' = 'datagen',
  'rows-per-second' = '1'
);
create table print(
  uuid varchar(20),
  name varchar(10),
  name1 varchar(10),
  age int,
  ts timestamp
) with ('connector' = 'print');
insert into
  print
select
  t1.uuid,
  t1.name,
  t2.name as name1,
  t2.age,
  t2.ts
from
  table1 t1
  join table2 t2 on t1.uuid = t2.uuid;
```

图 3-5 优化后



多表 left join 场景下关联键发生改变使用雪花模型代替星型模型

多表left join关联键发生更新时会发生数据乱序，建议右表先关联成一个view，然后再与左表关联。

关联键group_id改变导致“-D”和“+I”乱序，下游根据user_id哈希时虽然进入同一并行度，但是“+I”消息先到，“-D”消息后到，最终写入宽表时记录就会被删除。

- 优化前SQL：

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

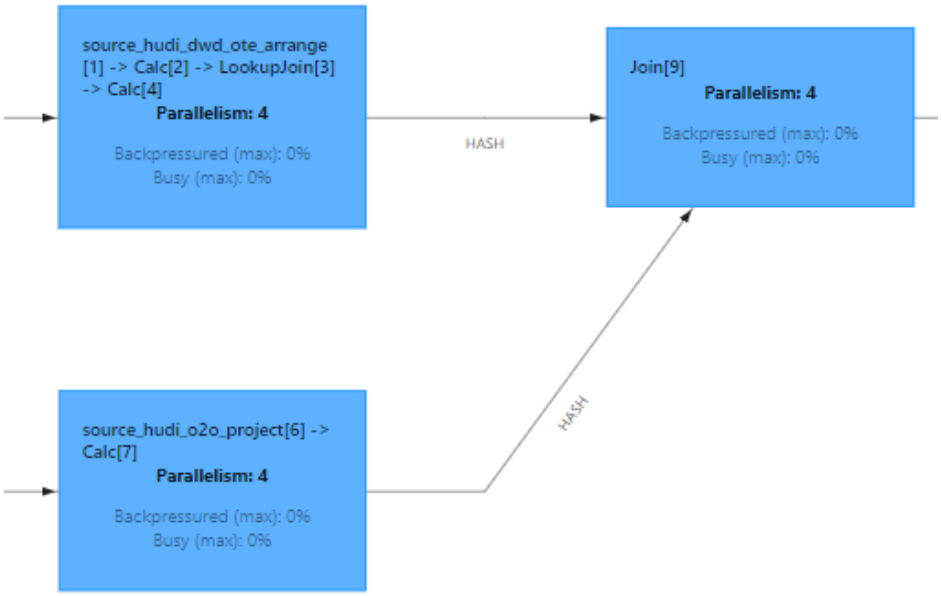
- 优化后SQL：

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

多表 left join 时建议 lookup join 在所有双流 join 后

多表left join时建议lookup join在所有双流join后，否则下游有left join LATERAL TABLE时会发生乱序。

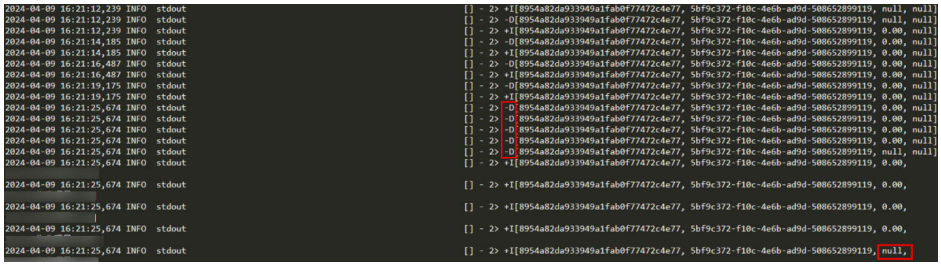
图 3-6 多表 left join



虽然左表已经定义主键，但是经过lookup join后下游left join时无法推断左流主键，导致左流所有历史数据都存储在状态，右流数据到达后会从最新的状态开始依次回撤左流状态中的每一条数据，经过LATERAL TABLE每一条source数据又与lateral table自关联，数据乱序。

查看打印结果可以看到连续多条“-D”消息，并且最后一条数据错误，因此建议lookup join放在双流join后执行。

图 3-7 连续多条“-D”消息



- 优化前SQL：

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

- 优化后SQL：

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

使用 char 数据类型时指定精度或者改用 string 类型

使用“cast(id as char)”数据类型转换时，结果只截取第一位，导致数据错误。如果转换字段正好是主键字段则会丢失大量数据。

配置“table.exec.legacy-cast-behaviour=ENABLED”也可以解决转换发生错误的问题，但是不建议使用。

在Flink 1.15之前，可以通过将“table.exec.legacy-cast-behaviour”设置为“enabled”来启用旧版本的类型转换行为。但在Flink 1.15及之后版本中，默认情况下该标志被禁用，将导致以下行为：

- 转换为CHAR/VARCHAR/BINARY/VARBINARY时禁用修剪/填充操作。
- CAST操作永远不会失败，而是返回NULL，类似于TRY_CAST，但不会推断正确的类型。
- 对于某些转换为CHAR/VARCHAR/STRING的格式化操作，结果可能略有不同。

我们不建议使用此标志，并强烈建议新项目保持禁用该标志并使用新的类型转换行为。该标志将在未来的Flink版本中被移除。

- 优化前SQL：

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

- 优化后SQL：

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

多个 Flink 作业或者 insert into 语句写同一张 Gauss for MySQL 时建议过滤回撤数据

当有多个Flink作业写同一张MySQL表时，其中一个Flink作业发送回撤数据（-D、-U）到目标表删除整行数据，再插入本次更新的数据，导致其他作业写入的字段全部丢失。

- 优化前SQL：

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



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

- 优化后SQL：

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

3.6 Flink 性能调优开发规范

3.6.1 Flink 性能调优规则

及时对 Hudi 表进行 compaction 防止 Hudi Source 算子 Checkpoint 完成时间过长

当Hudi Source算子Checkpoint完成时间长时，可检查该Hudi表compaction是否正常。因为当长时间不做compaction时list性能会变差。

在事实表与维度表关联场景中可以按表设置 TTL 降低状态后端数据量

具体使用指导参考**通过表级TTL进行状态后端优化**。

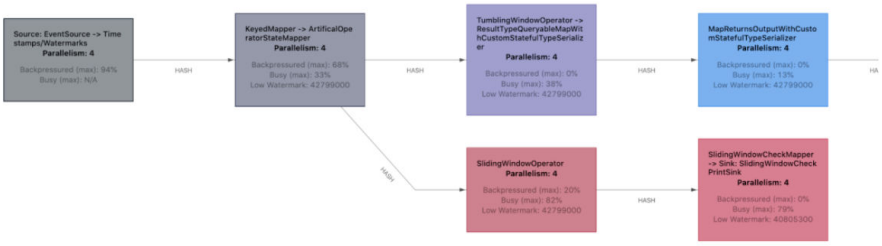
合理设置并行度

任务运行的速度和并行度相关，一般来说提升并行度能有效提升读取的速度，但是过大的并行度可能导致部分节点资源的浪费，过小的并行度可能导致部分节点运行缓

慢。对于SQL当前不能手动指定每个Task的并行度，指定的是所有Task统一的并行度。

推荐Source的并行度由上游组件推断设置，对于流系统，与上游的分区数相同（例如Kafka的Topic分区数）；对于批系统，与上游的切片数相同（例如HDFS的block数量）。

Flink作业中有Source、Sink、中间计算算子的并行度可以调整。通过分析作业流图，如果发现是中间计算Busy就需要通过调整整个作业并行度来调整这类算子的并行度，常见的如join算子。



3.6.2 Flink 性能调优建议

Hudi MOR 流表开启 log Index 特性提升 Flink 流读 Mor 表性能

Hudi的Mor表可以通过log index提升读写性能，在Sink和Source表添加属性'hoodie.log.index.enabled'='true'。

通过调整对应算子并行度提升性能

- 读写Hudi可以通过配置读写并发提升读写性能。

  读算子的并行度调整参数：read.tasks

  写算子的并行度调整参数：write.tasks
- 采用状态索引在作业重启的时候（非Checkpoint重启），需要读目标表重建索引，可以增大该算子并行度提升性能。

  加载索引的并行度调整参数：write.index_bootstrap.tasks
- 采用状态索引写数据需要进行主键唯一性检查，分配具体写入文件，提升该算子并行度提升性能。

  写算子索引检测算子调整参数：write.bucket_assign.tasks

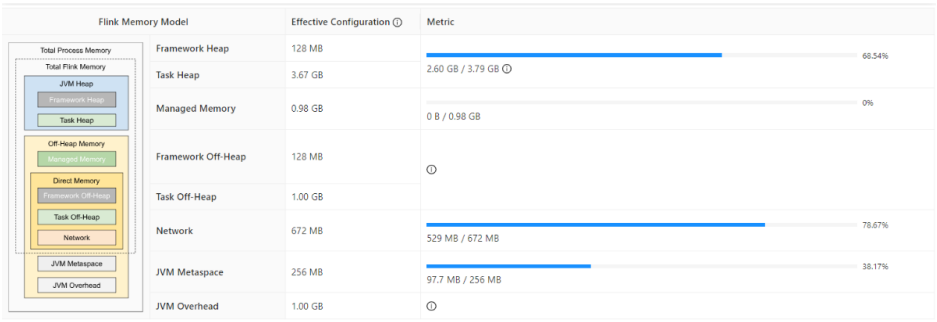
非状态计算提升性能的资源优化

Flink计算操作分为如下两类：

- 无状态计算操作：该部分算子不需要保存计算状态，例如：filter、union all、lookup join。
- 有状态计算操作：该部分算子要根据数据前后状态变化进行计算，例如：join，union、window、group by、聚合算子等。

对于非状态计算主要调优为TaskManager的Heap Size与NetWork。

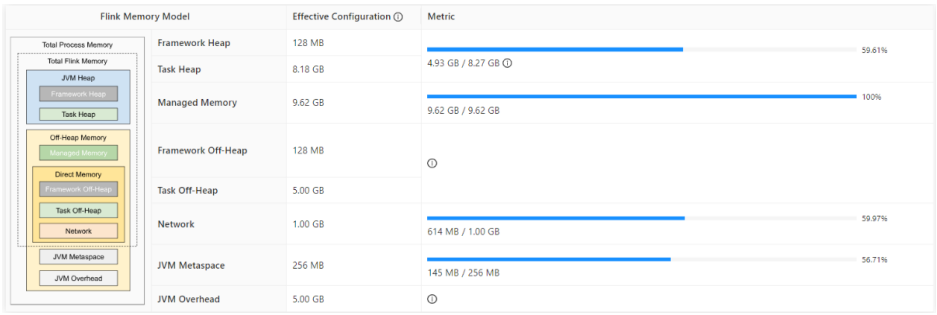
例如作业仅进行数据的读和写，TaskManage无需增加额外的vCore，off-Heap和Overhead默认为1GB，内存主要给Heap和Network。



状态计算提升性能的资源优化

SQL逻辑包含较多join、卷积计算等操作。主要调优状态后端性能、vCore、Manage Memory。

例如作业做三表多表关联，性能要求高，单个TaskManage增加额外的6个vCore，off-Heap和Overhead提高到5GB，用于Flink状态管理的Manage Memory为9.6GB。



通过表级 TTL 进行状态后端优化

本章节适用于MRS 3.3.0及以后版本。

在Flink双流Join场景下，若Join的左表和右表其中一个表数据变化快，需要较短时间的过期时间，而另一个表数据变化较慢，需要较长时间的过期时间。目前Flink只有表级别的TTL（Time To Live：生存时间），为了保证Join的准确性，需要将表级别的TTL设置为较长时间的过期时间，此时状态后端中保存了大量的已经过期的数据，给状态后端造成了较大的压力。为了减少状态后端的压力，可以单独为左表和右表设置不同的过期时间。不支持where子句。

可通过使用Hint方式单独为左表和右表设置不同的过期时间，如左表（state.ttl.left）设置TTL为60秒，右表（state.ttl.right）设置TTL为120秒：

- Hint方式格式：

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

key:
    stringLiteral
val:
    stringLiteral
```

- 在SQL语句中配置示例：

```
CREATE TABLE user_info (`user_id` VARCHAR, `user_name` VARCHAR) WITH (
  'connector' = 'kafka',
  'topic' = 'user_info_001',
  'properties.bootstrap.servers' = '192.168.64.138:21005',
  'properties.group.id' = 'testGroup',
  'scan.startup.mode' = 'latest-offset',
```

```
  'value.format' = 'csv'
);
CREATE table print(
  `user_id` VARCHAR,
  `user_name` VARCHAR,
  `score` INT
) WITH ('connector' = 'print');
CREATE TABLE user_score (user_id VARCHAR, score INT) WITH (
  'connector' = 'kafka',
  'topic' = 'user_score_001',
  'properties.bootstrap.servers' = '192.168.64.138:21005',
  'properties.group.id' = 'testGroup',
  'scan.startup.mode' = 'latest-offset',
  'value.format' = 'csv'
);
INSERT INTO
  print
SELECT
  t.user_id,
  t.user_name,
  d.score
FROM
  user_info as t
  LEFT JOIN
  -- 为左表和右表设置不同的TTL时间
  /*+ OPTIONS('state.ttl.left'='60S', 'state.ttl.right'='120S') */
  user_score as d ON t.user_id = d.user_id;
```

通过表级 JTL 进行状态后端优化

本章节适用于MRS 3.3.0及以后版本。

在Flink双流inner Join场景下，若Join业务允许join一次就可以剔除后端中的数据时，可以使用该特性。

该特性只适用于流流inner join。

可通过使用Hint方式单独为左表和右表设置不同join次数：

- Hint方式格式：

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

key:
    stringLiteral
val:
    stringLiteral
```

- 在SQL语句中配置示例：

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

```
);
INSERT INTO
  print
SELECT
  t.user_id,
  t.user_name,
  d.score
FROM
  user_info as t
  JOIN
  -- 为左表和右表设置不同的JTL关联次数
  /*+ OPTIONS('eliminate-state.left.threshold'='1','eliminate-state.right.threshold'='1') */
  user_score as d ON t.user_id = d.user_id;
```

TM 的 Slot 数和 TM 的 CPU 数成倍数关系

在Flink中，每个Task被分解成SubTask，SubTask作为执行的线程单位运行在TM上，在不开启Slot Sharing Group的情况下，一个SubTask是部署在一个slot上的。即使开启了Slot Sharing Group，大部分情况下Slot中拥有的SubTask也是负载均衡的。所以可以理解为TM上的Slot个数代表了上面运行的任务线程数。

合理的Slots数量应该和CPU核数相同，在使用超线程时，每个Slot将占用2个或更多的硬件线程。

【示例】建议配置TM Slot个数为CPU Core个数的2~4倍：

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

数据量大并发数高且有 Shuffle 时可调整网络内存

在并发数高和数据量大时，发生shuffle后会发生大量的网络IO，提升网络缓存内存可以扩大一次性读取的数据量，从而提升IO速度。

【示例】

```
# 网络占用内存占整个进程内存的比例
taskmanager.memory.network.fraction: 0.6
# 网络缓存内存的最小值
taskmanager.memory.network.min: 1g
# 网络缓存内存的最大值（MRS 3.3.1及之后版本无需修改该值，默认值已为Long#MAX_VALUE）
taskmanager.memory.network.max: 20g
```

基于序列化性能尽量使用 POJO 和 Avro 等简单的数据类型

使用API编写Flink程序时需要考虑Java对象的序列化，大多数情况下Flink都可以高效的处理序列化。SQL中无需考虑，SQL中数据都为ROW类型，都采用了Flink内置的序列化器，能很高效的进行序列化。

表 3-10 序列化

序列化器	Opts/s
PojoSeriallizer	813
Kryo	294
Avro(Reflect API)	114
Avro(SpecificRecord API)	632

网络通信调优

Flink通信主要依赖Netty网络，所以在Flink应用执行过程中，Netty的设置尤为重要，网络通信的好坏直接决定着数据交换的速度以及任务执行的效率。

【 示例 】

```
# netty的服务端线程数目(-1表示默认参数numOfSlot)
taskmanager.network.netty.server.numThreads -1(numOfSlot)
# netty的客户端线程数目(-1表示默认参数numofSlot)
taskmanager.network.netty.client.numThreads : -1
# netty的客户端连接超时时间
taskmanager.network.netty.client.connectTimeoutSec：120s
# netty的发送和接受缓冲区的大小(0表示netty默认参数，4MB)
taskmanager.network.netty.sendReceiveBufferSize: 0
# netty的传输方式，默认方式会根据运行的平台选择合适的方式
taskmanager.network.netty.transport：auto
```

内存总体调优

Flink内部对内存进行了划分，整体上划分成为了堆内存和堆外内存两部分。Java堆内存是通过Java程序创建时指定的，这也是JVM可自动GC的部分内存。堆外内存可细分为可被JVM管理的和不可被JVM管理的，可被JVM管理的有Managed Memory、Direct Memory，这部分是调优的重点，不可被JVM管理的有JVM Metaspace、JVM Overhead，这部分是native memory。

图 3-8 内存

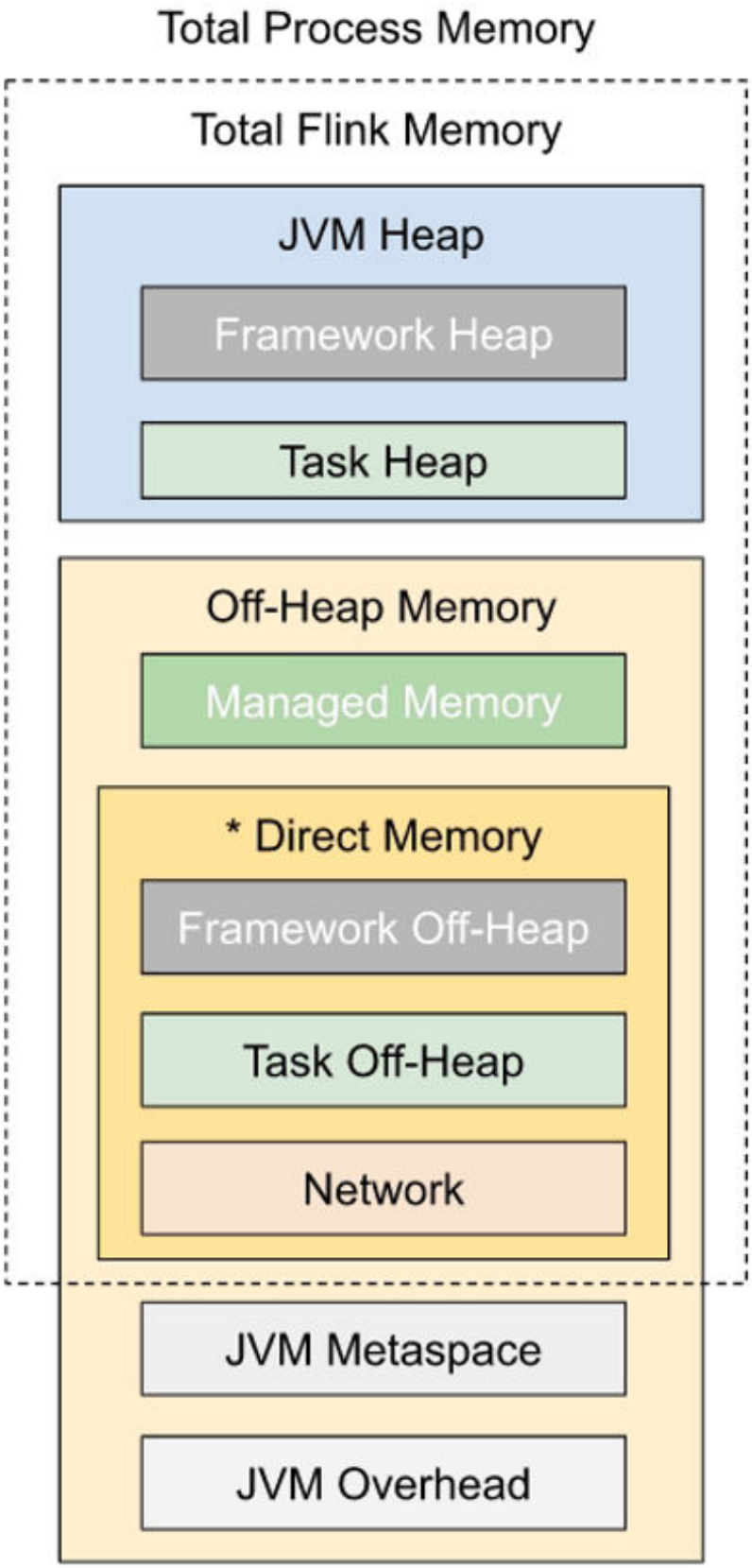


表 3-11 相关参数

参数	配置	注释	说明
Total Memory	taskmanager.memory.flink.size: none	总体Flink管理的内存大小，没有默认值，不包含Metaspace和Overhead，Standalone模式时设置。	整体内存。
	taskmanager.memory.process.size: none	整个Flink进程使用的内存大小，容器模式时设置。	
FrameWork	taskmanager.memory.framework.heap.size: 128mb	runtime占用的heap的大小，一般来说不用修改，占用空间相对固定。	RUNTIME底层占用的内存，一般不用做较大改变。
	taskmanager.memory.framework.off-heap.size: 128mb	runtime占用的off-heap的大小，一般来说不用修改，占用空间相对固定。	
Task	taskmanager.memory.task.heap.size：none	没有默认值，flink.size减去框架、托管、网络等得到。	算子逻辑，用户代码（如UDF）正常对象占用内存的地方。
	taskmanager.memory.task.off-heap.size：0	默认值为0，task使用的off heap内存。	
Managed Memory	taskmanager.memory.managed.fraction: 0.4	托管内存占taskmanager.memory.flink.size的比例，默认0.4。	managed内存用于中间结果缓存、排序、哈希等（批计算），以及RocksDB state backend（流计算），该内存在批模式下一开始就申请固定大小内存，而流模式下会按需申请。
	taskmanager.memory.managed.size: 0	托管内存大小，一般不指定，默认为0，内存大小由上面计算出来。若指定了则覆盖比例计算的内存。	
Network	taskmanager.memory.network.min：64mb	网络缓存的最小值。	用于taskmanager之间shuffle、广播以及与network buffer。
	taskmanager.memory.network.max：1gb	网络缓存的最大值。（MRS 3.3.1及之后版本无需修改该值，默认值已为Long#MAX_VALUE）	
	taskmanager.memory.network.fraction：0.1	network memory占用taskmanager.memory.flink.size的大小，默认0.1，会被限制在network.min和network.max之间。	用于taskmanager之间shuffle、广播以及与network buffer。

参数	配置	注释	说明
Others	taskmanager.memory.jvm-metaspace.size：256M	metaspace空间的最大值，默认值256MB。	用户自己管理的内存。
	taskmanager.memory.jvm-overhead.min：192M	jvm额外开销的最小值，默认192MB。	
	taskmanager.memory.jvm-overhead.max：1G	jvm额外开销的最大值，默认1GB。	
	taskmanager.memory.jvm-overhead.fraction：0.1	jvm额外开销占taskmanager.memory.process.size的比例，默认0.1，算出来后会被限制在jvm-overhead.min和jvm-overhead.max之间。	

说明

3.3.1及之后版本无需修改taskmanager.memory.network.max网络缓存的最大值

如果不能使用 broardcast join 应该尽量减少 shuffle 数据

不能broadcast join那么必定会发生shuffle，可通过各种手段来减少发生shuffle的数据量，例如谓词下推，Runtime Filter等等。

【示例】

```
# Runtime filter配置
table.exec.runtime-filter.enabled: true
# 下推
table.optimizer.source.predicate-pushdown-enabled: true
```

数据倾斜状态下可以使用 localglobal 优化策略

【示例】

```
#开启mini-batch优化
table.exec.mini-batch.enabled：true
#最长等待时间
table.exec.mini-batch.allow-latency: 20ms
#最大缓存记录数
table.exec.mini-batch.size：8000
#开启两阶段聚合
table.optimizer.agg-phase-strategy：TWO_PHASE
```

吞吐量大场景下使用 MiniBatch 聚合增加吞吐量

MiniBatch聚合的核心思想是将一组输入的数据缓存在聚合算子内部的缓冲区中。当输入的数据被触发处理时，每个key只需一个操作即可访问状态，可以很大程度减少状态

开销并获得更好的吞吐量。但是可能会增加一些延迟，因为它会缓冲一些记录而不是立即处理，这是吞吐量和延迟之间的权衡。默认未开启该功能。

- API方式：

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

- 资源文件方式（flink-conf.yaml）：

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

使用 local-global 两阶段聚合减少数据倾斜

Local-Global聚合是为解决数据倾斜问题提出的，通过将一组聚合分为两个阶段，首先在上游进行本地聚合，然后在下游进行全局聚合，类似于MapReduce中的 Combine + Reduce模式。

数据流中的记录可能会倾斜，因此某些聚合算子的实例必须比其他实例处理更多的记录，这会产生热点问题。本地聚合可以将一定数量具有相同key的输入数据累加到单个累加器中。全局聚合将仅接收reduce后的累加器，而不是大量的原始输入数据，这可以很大程度减少网络shuffle和状态访问的成本。每次本地聚合累积的输入数据量基于mini-batch间隔，这意味着local-global聚合依赖于启用了mini-batch优化。

- API方式：

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

- 资源文件方式：

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

RocksDB 作为状态后端时通过多块磁盘提升 IO 性能

RocksDB使用内存加磁盘的方式存储数据，当状态比较大时，磁盘占用空间会比较大。如果对RocksDB有频繁的读取请求，那么磁盘IO会成为Flink任务瓶颈。当一个TaskManager包含三个slot时，那么单个服务器上的三个并行度都对磁盘造成频繁读写，从而导致三个并行度的之间相互争抢同一个磁盘IO，导致三个并行度的吞吐量都会下降。可以通过指定多个不同的硬盘从而减少IO竞争。

【示例】Rockdb配置Checkpoint目录放在不同磁盘（flink-conf.yaml）：

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


RocksDB 作为状态后端时尽量使用 MapState 或 ListState 替换 ValueState 存储容器

RocksDB场景下，由于RocksDB是一个内嵌式的KV数据库，它的数据都是根据key和value进行存放的。对于map类数据，若使用ValueState，在RocksDB中作为一条记录存储，value是整个map，而使用MapState，在RocksDB中作为N条记录存储，这样做的好处是当进行查询或者修改可以只序列化一小部分数据，当将map作为整体存储时每次增删改都会产生很大的序列化开销。对于List数据，使用ListState可以无需序列化动态添加元素。

另外Flink中的State支持设置TTL，TTL实际上是将时间戳与userValue封装起来，ValueState的TTL基于整个Key，MapState<UK, UV>的TTL是基于UK，它的粒度更小，可支持更丰富的TTL语义。

Checkpoint 配置压缩减少 Checkpoint 大小

在IO密集型应用中，可以通过开启Checkpoint压缩，牺牲极小部分CPU性能，提升IO性能。

【示例】配置Checkpoint时开启压缩（flink-conf.yaml）：

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

大状态 Checkpoint 优先从本地状态恢复

为了快速的状态恢复，每个task会同时写Checkpoint数据到本地磁盘和远程分布式存储，也就是说这是一份双复制。只要task本地的Checkpoint数据没有被破坏，系统在应用恢复时会首先加载本地的Checkpoint数据，这样就很大程度减少了远程拉取状态数据的过程。

【示例】配置Checkpoint优先从本地恢复（flink-conf.yaml）：

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

3.7 Flink 开发样例

Flink支持对接ClickHouse、HBase、HDFS等多个服务，具体支持版本及样例详情可参考如下：

- **FlinkServer对接ClickHouse**
- **FlinkServer对接HBase**
- **FlinkServer对接HDFS**
- **FlinkServer对接Hive**
- **FlinkServer对接Hudi**
- **FlinkServer对接Kafka**

3.8 Flink 常见开发问题

3.8.1 Flink 作业提交时报错端口范围不足

问题现象

Flink作业提交时，没有足够的端口分配给actor system，导致作业启动失败，报错：Could not start actor system on any port in port range 32326-32390。

解决方法

MRS集群上服务众多，如果不限制端口范围可能导致其他服务端口被占用而导致异常，因此MRS集群给每个服务分配的端口范围是固定的，Flink端口范围是[32326-32390]。

当Flink作业单个taskmanager分配的slot数过多时会导致分配端口不足，可通过在“*客户端安装路径*/Flink/flink/conf/flink-conf.yaml”中配置“taskmanager.data.port”的值为“0”取消Flink端口分配限制。

说明

取消Flink端口分配限制后可能会占用其他服务的端口而导致集群异常，请谨慎配置。

3.8.2 Flink 对接 Elasticsearch 作业运行一段时间后 Checkpoint 失败

问题现象

Flink对接Elasticsearch作业，运行一段时间（TGT的有效期一般为24小时）后，写Elasticsearch失败，Checkpoint超时报错。

TGT（Ticket Granting ticket）：票据授权票据。

解决方法

Flink对接Elasticsearch作业在运行过程中，TGT超期后会重新进行认证，此时Elasticsearch缓存的票据信息没有更新，导致认证失败。可通过在“*客户端安装路径*/Flink/flink/conf/flink-conf.yaml”的“env.java.opts”配置项中添加如下参数解决。

```
-Djavax.security.auth.useSubjectCredsOnly=true
```

3.8.3 Flink Jar 包冲突报错 ClassCastException 类型转换异常

问题现象

Flink lib中引入第三方依赖包后，启动作业报错：

```
ClassCastException: X Cannot be cast to X
```

问题原因

引入的第三方依赖包与Flink中的依赖包有冲突，第三方依赖包中的类不兼容，导致某些函数实例化失败。

解决方法

方法一：排包，排除引入的第三方jar中的冲突包，建议优先使用该方法。

方法二：将flink-conf.yaml配置文件中配置项“classloader.resolve-order”的值修改为“parent-first”。修改Flink类加载顺序，使其优先加载Flink自带的依赖包。

3.8.4 如何设置开源 Flink 中的 znode 存储目录

问题现象

如何将开源Flink中的znode存储目录设置为自定义目录。

解决方法

如设置目录为/flink_base/flink，在flink-conf.yaml配置文件中将“high-availability.zookeeper.path.under.quota”的值设置为“/flink_base/flink”。

【示例】

```
high-availability.zookeeper.path.under.quota: /flink_base/flink
```

3.8.5 DGC 方式如何创建 Flink Hive Sql 作业

问题现象

使用DGC方式如何创建Flink Hive Sql作业。

解决方法

若通过DGC方式创建提交Flink Hive作业，以读Kafka写Hive作业为例，步骤如下：

1. 提前在Hive客户端中创建Hive表。例如：

```
create table user_behavior_hive_tbl_no_partition(
    user_id STRING,
    item_id STRING,
    cat_id STRING,
    ts timestamp
  ) PARTITIONED BY (dy STRING, ho STRING, mi STRING)
stored as textfile TBLPROPERTIES (
 'partition.time-extractor.timestamp-pattern' = '$dy $ho:$mi:00',
 'sink.partition-commit.trigger' ='process-time',
 'sink.partition-commit.delay' = '0S',
 'sink.partition-commit.policy.kind' = 'metastore,success-file'
 );
```

2. 创建Flink Hive Sql作业，在DGC提交运行。Sql示例如下：

```
CREATE TABLE test_kafka (
  user_id varchar,
  item_id varchar,
  cat_id varchar,
  zw_test timestamp
) WITH (
  'connector' = 'kafka',
  'topic' = 'zw_test_kafka',
  'format' = 'json',
  'properties.bootstrap.servers' = 'Kafka的Broker实例业务IP:Kafka端口号',
  'properties.group.id' = 'example-group1',
  'scan.startup.mode' = 'latest-offset'
);
CREATE CATALOG myhive WITH (
```

```
  'type' = 'hive',
  'hive-version' = '3.1.0',
  'default-database' = 'default'
);
use catalog myhive;
INSERT into
  user_behavior_hive_tbl_no_partition
SELECT
  user_id,
  item_id,
  cat_id,
  zw_test,
  DATE_FORMAT(zw_test, 'yyyy-MM-dd'),
  DATE_FORMAT(zw_test, 'HH'),
  DATE_FORMAT(zw_test, 'mm')
FROM
  default_catalog.default_database.test_kafka;
```

4 HBase 应用开发规范

4.1 HBase 应用开发规则

Configuration 实例的创建

该类应该通过调用HBaseConfiguration的create()方法来实例化。否则，将无法正确加载HBase中的相关配置项。

正确示例：

```
//该部分，应该是在类成员变量的声明区域声明
private Configuration hbaseConfig = null;
//建议在类的构造函数中，或者初始化方法中实例化该类
hbaseConfig = HBaseConfiguration.create();
```

错误示例：

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

共享 Configuration 实例

HBase客户端代码通过创建一个与ZooKeeper之间的HConnection，来获取与一个HBase集群进行交互的权限。一个ZooKeeper的HConnection连接，对应着一个Configuration实例，已经创建的HConnection实例，会被缓存起来。也就是说，如果客户端需要与HBase集群进行交互的时候，会传递一个Configuration实例到缓存中去，HBase Client部分通过已缓存的HConnection实例，来判断属于这个Configuration实例的HConnection实例是否存在，如果不存在，会创建一个新的HConnection，如果存在，则会直接返回相应的实例。

因此，如果频繁地创建Configuration实例，会导致创建很多不必要的HConnection实例，很容易达到ZooKeeper的连接数上限。

建议在整个客户端代码范围内，都共用同一个Configuration对象实例。

Table 实例的创建

```
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();
  }
}
```

不允许多个线程在同一时间共用同一个 Table 实例

Table是一个非线程安全类，因此，同一个Table实例，不应该被多个线程同时使用，否则可能会出现并发问题。

Table 实例缓存

如果一个Table实例可能长时间会被同一个线程固定且频繁地用到，例如，通过一个线程不断地往一个表内写入数据，那么这个Table在实例化后，就需要缓存下来，而不是每一次插入操作，都要实例化一个Table对象（尽管提倡实例缓存，但也不是在一个线程中一直沿用一个实例，个别场景下依然需要重构，可参见下一条规则）。

正确示例：

说明

注意该实例中提供的以Map形式缓存Table实例的方法，未必通用。这与多线程多Table实例的设计方案有关。如果确定一个Table实例仅仅可能会被用于一个线程，而且该线程也仅有一个Table实例的话，就无须使用Map。这里提供的思路仅供参考。

```
//该Map中以TableName为Key值，缓存所有已经实例化的Table
private Map<String, Table> demoTables = new HashMap<String, Table>();
//所有的Table实例，都将共享这个Configuration实例
private Configuration demoConf = null;
/**
* <初始化一个HTable类>
```



```
* <功能详细描述>
* @param tableName
* @return
* @throws IOException
* @see [类、类#方法、类#成员]
*/
private Table initNewTable(String tableName) throws IOException
{
try (Connection conn = ConnectionFactory.createConnection(demoConf)){
      return conn.getTable(tableName);
    }
}
/**
* <获取Table实例>
* <功能详细描述>
* @see [类、类#方法、类#成员]
*/
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;
}
}
/**
* <写数据>
* <这里未涉及到多线程多Table实例在设计模式上的优化.这里采用同步方法，
* 主要是是考虑到同一个Table是非线程安全的.通常，建议一个Table实例，在同一
*  时间只能被用在一个写数据的线程中>
* @param dataList
* @param tableName
* @see [类、类#方法、类#成员]
*/
public void putData(List<Put> dataList, String tableName)
{Table table = getTable(tableName);
//关于这里的同步：如果在采用的设计方案中，不存在多线程共用同一个Table实例
//的可能的话，就无须同步了。这里需要注意Table实例是非线程安全的
synchronized (table)
{
try
{
table.put(dataList);
table.notifyAll();
}
catch (IOException e)
{
                // 在捕获到IOE时，需要将缓存的实例重构。
try {
    // 关闭之前的Connection.
      table.close();
                // 重新创建这个实例.
                table = initNewTable(tableName);
} catch (IOException e1) {
// TODO
}
}
```

```
}
}
```

错误示例：

```
public void putDataIncorrect(List<Put> dataList, String tableName)
{Table table = null;
try
{
//每次写数据，都创建一个HTable实例
table = initNewTable(tableName);
table.put(dataList);
}
catch (IOException e1)
{
// TODO Auto-generated catch block
e1.printStackTrace();
}
finally
{
table.close();
}
}
```

Table 实例写数据的异常处理

尽管在前一条规则中提到了提倡Table实例的重构，但是，并非提倡一个线程自始至终要沿用同一个Table实例，当捕获到IOException时，依然需要重构Table实例。示例代码可参考上一个规则的示例。

另外，请谨慎调用如下两个方法：

- **Configuration#clear：**

  这个方法，会清理所有已加载的属性，对于已经在使用这个Configuration的类或线程而言，可能会带来潜在的问题（例如，假如Table还在使用这个Configuration，那么，调用这个方法后，Table中的这个Configuration的所有的参数，都被清理掉了），也就是说：只要还有对象或者线程在使用这个Configuration，就不应该调用这个clear方法，除非所有的类或线程，都已经确定不用这个Configuration了。

  因此，这个方法，应该要放在进程退出时执行，而不是每一次Table要重构的时候执行。

- **HConnectionManager#deleteAllConnections：**

  这个可能会导致现有的正在使用的连接被从**连接集合**中清理掉，同时，因为在HTable中保存了原有连接的引用，可能会导致这个连接无法关闭，进而可能会造成泄漏。因此，这个方法不建议使用。

写入失败的数据要做相应的处理

在写数据的过程中，如果进程异常或一些其它的短暂的异常，可能会导致一些写入操作失败。因此，对于操作的数据，需要将其记录下来。在集群恢复正常后，重新将其写入到HBase数据表中。

另外，有一点需要注意：HBase Client返回写入失败的数据，是不会自动重试的，仅仅会告诉接口调用者哪些数据写入失败了。对于写入失败的数据，一定要做一些安全的处理，例如可以考虑将这些失败的数据，暂时写在文件中，或者，直接缓存在内存中。

正确示例：


```
private List<Row> errorList = new ArrayList<Row>();
/**
* <采用PutList的模式插入数据>
* <如果不是多线程调用该方法，可不采用同步>
* @param put 一条数据记录
* @throws IOException
* @see [类、类#方法、类#成员]
*/
public synchronized void putData(Put put)
{
// 暂时将数据缓存在该List中
dataList.add(put);
// 当dataList的大小达到PUT_LIST_SIZE之后，就执行一次Put操作
if (dataList.size() >= PUT_LIST_SIZE)
{
try
{
demoTable.put(dataList);
}
catch (IOException e)
{
// 如果是RetriesExhaustedWithDetailsException类型的异常，
// 说明这些数据中有部分是写入失败的这通常都是因为
// HBase集群的进程异常引起，有时也会因为有大量
// 的Region正在被转移，导致尝试一定的次数后失败
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();
}
}
```

资源释放

关于ResultScanner和Table实例，在用完之后，需要调用它们的Close方法，将资源释放掉。Close方法，要放在finally块中，来确保一定会被调用到。

正确示例：

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

错误示例：

1. 在代码中未调用scanner.close()方法释放相关资源。
2. scanner.close()方法未放置在finally块中。

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


Scan 时的容错处理

Scan时不排除会遇到异常，例如，租约过期。在遇到异常时，建议Scan应该有重试的操作。

事实上，重试在各类异常的容错处理中，都是一种优秀的实践，这一点，可以应用在各类与HBase操作相关的接口方法的容错处理过程中。

不用 Admin 时，要及时关闭，Admin 实例不应常驻内存

Admin的实例应尽量遵循 “用时创建，用完关闭”的原则。不应该长时间缓存同一个Admin实例。

4.2 HBase 应用开发建议

不要调用 Admin 的 closeRegion 方法关闭一个 Region

Admin中，提供了关闭一个Region的接口：

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

通过该方法关闭一个Region，HBase Client端会直接发RPC请求到Region所在的RegionServer上，整个流程对Master而言，是不感知的。也就是说，尽管RegionServer关闭了这个Region，但是，在Master侧，还以为该Region是在该RegionServer上面打开的。假如，在执行Balance的时候，Master计算出恰好要转移这个Region，那么，这个Region将无法被关闭，本次转移操作将无法完成（关于这个问题，在当前的HBase版本中的处理的确还欠缺妥当）。

因此，暂时不建议使用该方法关闭一个Region。

采用 PutList 模式写数据

Table类中提供了两种写数据的接口：

1. public void put(final Put put) throws IOException
2. public void put(final List<Put> puts) throws IOException

第1种方法较之第2种方法，在性能上有明显的弱势。因此，写数据时应该采用第2种方法。

Scan 时指定 StartKey 和 EndKey

一个有确切范围的Scan，在性能上会带来较大的好处。

代码示例：

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

不要关闭 WAL

WAL是Write-Ahead-Log的简称，是指数据在入库之前，首先会写入到日志文件中，借此来确保数据的安全性。

WAL功能默认是开启的，但是，在Put类中提供了关闭WAL功能的接口：

public void setWriteToWAL(boolean write)

因此，不建议调用该方法将WAL关闭（即将writeToWAL设置为False），因为可能会造成最近1S（该值由RegionServer端的配置参数“hbase.regionserver.optionallogflushinterval”决定，默认为1S）内的数据丢失。但如果在实际应用中，对写入的速率要求很高，并且可以容忍丢失最近1S内的数据的话，可以将该功能关闭。

创建一张表或 Scan 时设定 blockcache 为 true

HBase客户端建表和scan时，设置blockcache=true。需要根据具体的应用需求来设定它的值，这取决于有些数据是否会被反复的查询到，如果存在较多的重复记录，将这个值设置为true可以提升效率，否则，建议关闭。

建议按默认配置，默认就是true，只要不强制设置成false就可以，例如：

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

HBase 不支持条件查询和 Orderby 等查询方法，存储按照字典排序，读取只支持 Rowkey 扫描

设计时应避免HBase随机查找、排序的应用场景。

业务表设计建议

1. 预分Region，使Region分布均匀，提高并发
2. 避免过多的热点Region。根据应用场景，可考虑将时间因素引入Rowkey。
3. 同时访问的数据尽量连续存储。同时读取的数据相邻存储；同时读取的数据存放在同一行；同时读取的数据存放在同一cell。
4. 查询频繁属性放在Rowkey前面部分。Rowkey的设计在排序上必须与主要的查询条件契合。
5. 离散度较好的属性作为RowKey组成部分。分析数据离散度特点以及查询场景，综合各种场景进行设计。
6. 存储冗余信息，提高检索性能。使用二级索引，适应更多查询场景。
7. 利用过期时间、版本个数设置等操作，让表能自动清除过期数据。

说明

在HBase中，一直在繁忙写数据的Region被称为热点Region。

5 HDFS 应用开发规范

5.1 HDFS 应用开发规则

HDFS NameNode 元数据存储路径

NameNode元数据信息的默认存储路径为“${BIGDATA_DATA_HOME}/namenode/data”，该参数用于确定HDFS文件系统的元数据信息的保存路径。

HDFS 需要开启 NameNode 镜像备份

NameNode的镜像备份参数为“fs.namenode.image.backup.enable”，需要设置该值为“true”，系统即可定期备份NameNode的数据。

HDFS 需要开启 DataNode 数据存储路径

DataNode默认存储路径配置为：${BIGDATA_DATA_HOME}/hadoop/dataN/dn/datadir（N≥1），N为数据存放的目录个数。

例如：${BIGDATA_DATA_HOME}/hadoop/data1/dn/datadir、${BIGDATA_DATA_HOME}/hadoop/data2/dn/datadir

设置后，数据会存储到节点上每个挂载磁盘的对应目录下面。

HDFS 提高读取写入性能方式

写入数据流程：HDFS Client收到业务数据后，从NameNode获取到数据块编号、位置信息后，联系DataNode，并将需要写入数据的DataNode建立起流水线，完成后，客户端再通过自有协议写入数据到Datanode1，再由DataNode1复制到DataNode2、DataNode3（三备份）。写完的数据，将返回确认信息给HDFS Client。

1. 合理设置块大小，如设置dfs.blocksize为 268435456（即256MB）。
2. 对于一些不可能重用的大数据，缓存在操作系统的缓存区是无用的。可将以下两参数设置为false：

   dfs.datanode.drop.cache.behind.reads和dfs.datanode.drop.cache.behind.writes

MapReduce 中间文件存放路径

MapReduce默认中间文件夹存放路径只有一个，${hadoop.tmp.dir}/mapred/local，建议修改为每个磁盘下均可存放中间文件。

例如：/hadoop/hdfs/data1/mapred/local、/hadoop/hdfs/data2/mapred/local、/hadoop/hdfs/data3/mapred/local等，不存在的目录会自动忽略。

JAVA 开发时，申请资源须在 finally 释放

申请的HDFS资源需要在try/finally中释放，而不能只在try语句之外释放，否则会导致异常情况下的资源泄漏。

HDFS 文件操作 API 概述

Hadoop中关于文件操作类基本上全部是在“org.apache.hadoop.fs”包中，这些API能够支持的操作包含：打开文件，读写文件，删除文件等。Hadoop类库中最终面向用户提供的接口类是FileSystem，该类是个抽象类，只能通过来类的get方法得到具体类。get方法存在几个重载版本，常用的是这个：

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

该类封装了几乎所有的文件操作，例如mkdir，delete等。综上基本可以得出操作文件的程序库框架：

```
operator()
{
    得到Configuration对象
    得到FileSystem对象
    进行文件操作
 }
```

HDFS 初始化方法

HDFS初始化是指在使用HDFS提供的API之前，需要做的必要工作。

大致过程为：加载HDFS服务配置文件，并进行Kerberos安全认证，认证通过后再实例化Filesystem，之后使用HDFS的API。此处Kerberos安全认证需要使用到的keytab文件，请提前准备。

正确示例：

```
private  void init() throws IOException {
    Configuration conf = new Configuration();
    // 读取配置文件
    conf.addResource("user-hdfs.xml");
   // 安全模式下，先进行安全认证
    if ("kerberos".equalsIgnoreCase(conf.get("hadoop.security.authentication"))) {
        String PRINCIPAL = "username.client.kerberos.principal";
        String KEYTAB = "username.client.keytab.file";
        // 设置keytab密钥文件
        conf.set(KEYTAB, System.getProperty("user.dir") + File.separator + "conf" + File.separator +
conf.get(KEYTAB));
        // 设置kerberos配置文件路径 */
        String krbfilepath = System.getProperty("user.dir") + File.separator + "conf" + File.separator +
"krb5.conf";
        System.setProperty("java.security.krb5.conf", krbfilepath);
        // 进行登录认证 */
        SecurityUtil.login(conf, KEYTAB, PRINCIPAL);
    }

    // 实例化文件系统对象
```

```
    fSystem = FileSystem.get(conf);
 }
```

HDFS 上传本地文件

通过FileSystem.copyFromLocalFile（Path src，Patch dst）可将本地文件上传到HDFS的指定位置上，其中src和dst均为文件的完整路径。

正确示例：

```
public class CopyFile {
       public static void main(String[] args) throws Exception {
       Configuration conf=new Configuration();
       FileSystem hdfs=FileSystem.get(conf);
            //本地文件
              Path src =new Path("D:\\HebutWinOS");
               //HDFS为止
       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());
            }
       }
 }
```

HDFS 创建文件

通过"FileSystem.mkdirs（Path f）"可在HDFS上创建文件夹，其中f为文件夹的完整路径。

正确示例：

```
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);
     }
 }
```

查看 HDFS 文件的最后修改时间

通过FileSystem.getModificationTime()可查看指定HDFS文件的修改时间。

正确示例：

```
    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的修改时间是"+modiTime);
   }
```

读取 HDFS 某个目录下的所有文件

通过FileStatus.getPath（ ）可查看指定HDFS中某个目录下所有文件。

正确示例：

```
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();
 }
```

查找某个文件在 HDFS 集群的位置

通过FileSystem.getFileBlockLocation（FileStatus file，long start，long len）可查找指定文件在HDFS集群上的位置，其中file为文件的完整路径，start和len来标识查找文件的路径。

正确示例：

```
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]);
        }
   }
```

获取 HDFS 集群上所有节点名称信息

通过DatanodeInfo.getHostName（ ）可获取HDFS集群上的所有节点名称。

正确示例：

```
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());
       }
  }
```

多线程安全登录方式

如果有多线程进行login的操作，当应用程序第一次登录成功后，所有线程再次登录时应该使用relogin的方式。

login的代码样例：

```
 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的代码样例：

```
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;
    }
```

警告

多次重复登录会导致后建立的会话对象覆盖掉之前登录建立的，将会导致之前建立的会话无法被维护监控，最终导致会话超期后部分功能不可用。

5.2 HDFS 应用开发建议

HDFS 的读写文件注意点

HDFS不支持随机读和写。

HDFS追加文件内容只能在文件末尾添加，不能随机添加。

只有存储在HDFS文件系统中的数据才支持append，edit.log以及数据元文件不支持Append。Append追加文件时，需要将“hdfs-site.xml”中的“dfs.support.append”参数值设置为true。

说明

- “dfs.support.append”参数在开源社区版本中默认值是关闭，在FusionInsight版本默认值是开启。
- 该参数为服务器端参数。建议开启，开启后才能使用Append功能。
- 不适用HDFS场景可以考虑使用其他方式来存储数据，如HBase。

HDFS 不适用于存储大量小文件

HDFS不适用于存储大量的小文件，因为大量小文件的元数据会占用NameNode的大量内存。

HDFS 中数据的备份数量 3 份即可

DataNode数据备份数量3份即可，增加备份数量不能提升系统效率，只会提升系统数据的安全系数；在某个节点损坏时，该节点上的数据会被均衡到其他节点上。

HDFS 定期镜像备份

NameNode的镜像备份参数为“fs.namenode.image.backup.enable”，将设置该值为“true”，系统即可定期备份NameNode的数据。

提供数据可靠性相关操作

在调用write函数写入数据时，HDFS客户端并不会将数据写入HDFS，而是缓存在客户端内存中，此时若客户端异常、断电，则数据丢失。对于有高可靠要求的数据，应该写完后，调用hflush将数据刷新到HDFS侧。

6 Hive 应用开发规范

6.1 Hive 应用开发规则

Hive JDBC 驱动的加载

客户端程序以JDBC的形式连接HiveServer时，需要首先加载Hive的JDBC驱动类org.apache.hive.jdbc.HiveDriver。

故在客户端程序的开始，必须先使用当前类加载器加载该驱动类。

如果classpath下没有相应的jar包，则客户端程序抛出Class Not Found异常并退出。

如下：

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

获取数据库连接

使用JDK的驱动管理类java.sql.DriverManager来获取一个Hive的数据库连接。

Hive的数据库URL为url="jdbc:hive2://xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181,xxx.xxx.xxx.xxx:2181/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver;sasl.qop=auth-conf;auth=KERBEROS;principal=hive/hadoop.hadoop.com@HADOOP.COM;user.principal=hive/hadoop.hadoop.com;user.keytab=conf/hive.keytab";

以上已经经过安全认证，所以Hive数据库的用户名和密码为null或者空。

如下：

```
// 建立连接
connection = DriverManager.getConnection(url, "", "");
```

执行 HQL

执行HQL，注意HQL不能以";"结尾。

正确示例：

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



```
PreparedStatement statement = connection.prepareStatement(sql);
resultSet = statement.executeQuery();
```

错误示例：

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

关闭数据库连接

客户端程序在执行完HQL之后，注意关闭数据库连接，以免内存泄露，同时这是一个良好的编程习惯。

需要关闭JDK的两个对象statement和connection。

如下：

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

        // 关闭JDBC连接
        if (null != connection) {
            connection.close();
        }
    }
```

HQL 语法规则之判空

判断字段是否为“空”，即没有值，使用“is null”；判断不为空，即有值，使用“is not null”。

要注意的是，在HQL中String类型的字段若是空字符串， 即长度为0，那么对它进行IS NULL的判断结果是False。此时应该使用“col = '' ”来判断空字符串；使用“col != '' ”来判断非空字符串。

正确示例：

```
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 != '';
```

错误示例：

```
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;
```

注：表tbl_src的id字段为Int类型，name字段为String类型。

客户端配置参数需要与服务端保持一致

当集群的Hive、YARN、HDFS服务端配置参数发生变化时，客户端程序对应的参数会被改变，用户需要重新审视在配置参数变更之前提交到HiveServer的配置参数是否和服务端配置参数一致，如果不一致，需要用户在客户端重新调整并提交到HiveServer。例如下面的示例中，如果修改了集群中的YARN配置参数时，Hive客户端、示例程序都需要审视并修改之前已经提交到HiveServer的配置参数：

初始状态：

集群YARN的参数配置如下：

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

客户端的参数配置如下：

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

集群YARN修改后，参数配置如下：

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

如果此时客户端程序不做调整修改，则客户端参数仍旧有效，会导致Reducer内存不足而使任务运行失败。

多线程安全登录方式

如果有多线程进行login的操作，当应用程序第一次登录成功后，所有线程再次登录时应该使用relogin的方式。

login的代码样例：

```
 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的代码样例：

```
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;
   }
```

使用 WebHCat 的 REST 接口以 Streaming 方式提交 MR 任务的前置条件

本接口需要依赖hadoop的streaming包，在以Streaming方式提交MR任务给WebHCat前，需要将“hadoop-streaming-2.7.0.jar”包上传到HDFS的指定路径下：“hdfs:///apps/templeton/hadoop-streaming-2.7.0.jar”。首先登录到安装有客户端和Hive服务的节点上，以客户端安装路径为“/opt/client”为例：

source /opt/client/bigdata_env

使用kinit登录人机用户或者机机用户。

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

其中**/apps/templeton**/需要根据不同的实例进行修改，默认实例使用**/apps/templeton/**，Hive1实例使用**/apps1/templeton/**，以此类推。

避免对同一张表同时进行读写操作

目前的版本中，Hive不支持并发操作，需要避免对同一张表同时进行读写操作，否则会出现查询结果不准确，甚至任务失败的情况。

分桶表不支持 insert into

分桶表（bucket table）不支持insert into，仅支持insert overwrite，否则会导致文件个数与桶数不一致。

使用 WebHCat 的部分 REST 接口的前置条件

WebHCat的部分REST接口使用依赖于MapReduce的JobHistoryServer实例，具体接口如下：

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

Hive 授权说明

Hive授权（数据库、表或者视图）推荐通过Manager授权界面进行授权，不推荐使用命令行授权，除了“alter databases databases_name set owner='user_name'”场景以外。

不允许创建 Hive on HBase 的分区表

Hive on HBase表将实际数据存储在HBase上。由于HBase会将表划分为多个分区，将分区散列在RegionServer上，因此不允许在Hive中创建Hive on HBase分区表。

Hive on HBase 表不支持 INSERT OVERWRITE

HBase中使用rowkey作为一行记录的唯一标识。在插入数据时，如果rowkey相同，则HBase会覆盖该行的数据。如果在Hive中对一张Hive on HBase表执行INSERT OVERWRITE，会将相同rowkey的行进行覆盖，不相关的数据不会被覆盖。

6.2 Hive 应用开发建议

HQL 编写之隐式类型转换

查询语句使用字段的值做过滤时，不建议通过Hive自身的隐式类型转换来编写HQL。因为隐式类型转换不利于代码的阅读和移植。

建议示例：

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

不建议示例：

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

说明

表tbl_src的id字段为Int类型，name字段为String类型。

HQL 编写之对象名称长度

HQL的对象名称，包括表名、字段名、视图名、索引名等，其长度建议不要超过30个字节。

Oracle中任何对象名称长度不允许超过30个字节，超过时会报错。PT为了兼容Oracle，对对象的名称进行了限制，不允许超过30个字节。

太长不利于阅读、维护、移植。

HQL 编写之记录个数统计

统计某个表所有的记录个数，建议使用“select count(1) from table_name”。

统计某个表某个字段有效的记录个数，建议使用“select count(column_name) from table_name”。

JDBC 超时限制

Hive提供的JDBC实现有超时限制，默认是5分钟，用户可以通过**java.sql.DriverManager.setLoginTimeout(int seconds)**设置，*seconds*的单位为秒。

UDF 管理

建议由管理员创建永久UDF，避免每次使用时都去add jar，和重新定义UDF。

Hive的UDF会有一些默认属性，比如“deterministic”默认为“true”（同一个输入会返回同一个结果），“stateful”（是否有状态，默认为“true”）。当用户实现的自定义UDF内部实现了汇总等，需要在类上加上相应的注解，例如如下类：

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


表分区优化建议

1. 当数据量较大，且经常需要按天统计时，建议使用分区表，按天存放数据。
2. 为了避免在插入动态分区数据的过程中，产生过多的小文件，在执行插入时，在分区字段上加上distribute by。

存储文件格式优化建议

Hive支持多种存储格式，比如TextFile，RCFile，ORC，Sequence，Parquet等。为了节省存储空间，或者大部分时间只查询其中的一部分字段时，可以在建表时使用列式存储(比如ORC文件)。

7 Hudi 应用开发规范

7.1 Hudi 开发规范概述

范围

本规范主要描述基于MRS-Hudi组件进行湖仓一体、流批一体方案的设计与开发方面的规则。其主要包括以下方面的规范：

- 数据表设计
- 资源配置
- 性能调优
- 常见故障处理
- 常用参数配置

术语约定

本规范采用以下的术语描述：

- **规则**：编程时强制必须遵守的原则。
- **建议**：编程时必须加以考虑的原则。
- **说明**：对此规则或建议进行的解释。
- **示例**：对此规则或建议从正、反两个方面给出。

适用范围

- 基于MRS-Hudi进行数据存储、数据加工作业的设计、开发、测试和维护。
- 该设计开发规范是基于MRS 3.3.0版本。

7.2 Hudi 数据表设计规范

7.2.1 Hudi 表模型设计规范

规则

- Hudi表必须设置合理的主键。

  Hudi表提供了数据更新和幂等写入能力，该能力要求Hudi表必须设置主键，主键设置不合理会导致数据重复。主键可以为单一主键也可以为复合主键，两种主键类型均要求主键不能有null值和空值，可以参考以下示例设置主键：

  SparkSQL：

  ```
  -- 通过primaryKey指定主键，如果是复合主键需要用逗号分隔。
  create table hudi_table (
  id1 int,
  id2 int,
  name string,
  price double
  ) using hudi
  options (
  primaryKey = 'id1,id2',
  preCombineField = 'price'
  );
  ```

  SparkDatasource：

  ```
  --通过hoodie.datasource.write.recordkey.field指定主键。
  df.write.format("hudi").
  option("hoodie.datasource.write.table.type", COPY_ON_WRITE).
  option("hoodie.datasource.write.precombine.field", "price").
  option("hoodie.datasource.write.recordkey.field", "id1,id2").
  ```

  FlinkSQL：

  ```
  --通过hoodie.datasource.write.recordkey.field指定主键。
  create table hudi_table(
  id1 int,
  id2 int,
  name string,
  price double
  ) partitioned by (name) with (
  'connector' = 'hudi',
  'hoodie.datasource.write.recordkey.field' = 'id1,id2',
  'write.precombine.field' = 'price')
  ```

- Hudi表必须配置precombine字段。

  在数据同步过程中不可避免会出现数据重复写入、数据乱序问题，例如：异常数据恢复、写入程序异常重启等场景。通过设置合理precombine字段值可以保证数据的准确性，老数据不会覆盖新数据，也就是幂等写入能力。该字段可用选择的类型包括：业务表中更新时间戳、数据库的提交时间戳等。precombine字段不能有null值和空值，可以参考以下示例设置precombine字段：

  SparkSQL:

  ```
  --通过preCombineField指定precombine字段。
  create table hudi_table (
  id1 int,
  id2 int,
  name string,
  price double
  ) using hudi
  options (
  primaryKey = 'id1,id2',
  preCombineField = 'price'
  );
  ```

  SparkDatasource:

  ```
  --通过hoodie.datasource.write.precombine.field指定precombine字段。
  df.write.format("hudi").
  ```



```
option("hoodie.datasource.write.table.type", COPY_ON_WRITE).
option("hoodie.datasource.write.precombine.field", "price").
option("hoodie.datasource.write.recordkey.field", "id1,id2").
```

Flink:

```
--通过write.precombine.field指定precombine字段。
create table hudi_table(
id1 int,
id2 int,
name string,
price double
) partitioned by (name) with (
'connector' = 'hudi',
'hoodie.datasource.write.recordkey.field' = 'id1,id2',
'write.precombine.field' = 'price')
```

- 流式计算采用MOR表。

  流式计算为低时延的实时计算，需要高性能的流式读写能力，在Hudi表中存在的MOR和COW两种模型中，MOR表的流式读写性能相对较好，因此在流式计算场景下采用MOR表模型。关于MOR表在读写性能的对比关系如下：

对比维度	MOR表	COW表
流式写	高	低
流式读	高	低
批量写	高	低
批量读	低	高

- 实时入湖，表模型采用MOR表。

  实时入湖一般的性能要求都在分钟内或者分钟级，结合Hudi两种表模型的对比，因此在实时入湖场景中需要选择MOR表模型。

- Hudi表名以及列名采用小写字母。

  多引擎读写同一张Hudi表时，为了规避引擎之间大小写的支持不同，统一采用小写字母。

建议

- Spark批处理场景，对写入时延要求不高的场景，采用COW表。

  COW表模型中，写入数据存在写放大问题，因此写入速度较慢；但COW具有非常好的读取性能力。而且批量计算对写入时延不是很敏感，因此可以采用COW表。

- Hudi表的写任务要开启Hive元数据同步功能。

  SparkSQL天然与Hive集成，无需考虑元数据问题。该条建议针对的是通过Spark Datasource API或者Flin写Hudi表的场景，通过这两种方式写Hudi时需要增加向Hive同步元数据的配置项；该配置的目的是将Hudi表的元数据统一托管到Hive元数据服务中，为后续的跨引擎操作数据以及数据管理提供便利。

7.2.2 Hudi 表索引设计规范

规则

- 禁止修改表索引类型。

  Hudi表的索引会决定数据存储方式，随意修改索引类型会导致表中已有的存量数据与新增数据之间出现数据重复和数据准确性问题。常见的索引类型如下：

  - 布隆索引：Spark引擎独有索引，采用bloomfiter机制，将布隆索引内容写入到Parquet文件的footer中。
  - Bucket索引：在写入数据过程中，通过主键进行Hash计算，将数据进行分桶写入；该索引写入速度最快，但是需要合理配置分桶数目；Flink、Spark均支持该索引写入。
  - 状态索引：Flink引擎独有索引，是将行记录的存储位置记录到状态后端的一种索引形式，在作业冷启动过程中会遍历所有数据存储文件生成索引信息。

- 用Flink状态索引，Flink写入后，不支持Spark继续写入。

  Flink在写Hudi的MOR表只会生成log文件，后续通过compaction操作，将log文件转为parquet文件。Spark在更新Hudi表时严重依赖parquet文件是否存在，如果当前Hudi表写的是log文件，采用Spark写入就会导致重复数据的产生。在批量初始化阶段 ，先采用Spark批量写入Hudi表，再用Flink基于Flink状态索引写入不会有问题，原因是Flink冷启动的时候会遍历所有的数据文件生成状态索引。

- 实时入湖场景中，Spark引擎采用Bucket索引，Flink引擎可以用Bucket索引或者状态索引。

  实时入湖都是需要分钟内或者分钟级的高性能入湖，索引的选择会影响到写Hudi表的性能。在性能方面各个索引的区别如下：

  - Bucket索引

    优点：写入过程中对主键进行hash分桶写入，性能比较高，不受表的数据量限制。Flink和Spark引擎都支持，Flink和Spark引擎可以实现交叉混写同一张表。

    缺点：Bucket个数不能动态调整，数据量波动和整表数据量持续上涨会导致单个Bucket数据量过大出现大数据文件。需要结合分区表来进行平衡改善。

  - Flink状态索引

    优点：主键的索引信息存在状态后端，数据更新只需要点查状态后端即可，速度较快；同时生成的数据文件大小稳定，不会产生小文件、超大文件问题。

    缺点：该索引为Flink特有索引。在表的总数据行数达到数亿级别，需要优化状态后端参数来保持写入的性能。使用该索引无法支持Flink和Spark交叉混写。

- 对于数据总量持续上涨的表，采用Bucket索引时，须使用时间分区，分区键采用数据创建时间。

  参照Flink状态索引的特点，Hudi表超过一定数据量后，Flink作业状态后端压力很大，需要优化状态后端参数才能维持性能；同时由于Flink冷启动的时候需要遍历全表数据，大数据量也会导致Flink作业启动缓慢。因此基于简化使用的角度，针对大数据量的表，可以通过采用Bucket索引来避免状态后端的复杂调优。

  如果Bucket索引+分区表的模式无法平衡Bueckt桶过大的问题，还是可以继续采用Flink状态索引，按照规范去优化对应的配置参数即可。

建议

- 基于Flink的流式写入的表，在数据量超过2亿条记录，采用Bucket索引，2亿以内可以采用Flink状态索引。

  参照Flink状态索引的特点，Hudi表超过一定数据量后，Flink作业状态后端压力很大，需要优化状态后端参数才能维持性能；同时由于Flink冷启动的时候需要遍历全表数据，大数据量也会导致Flink作业启动缓慢。因此基于简化使用的角度，针对大数据量的表，可以通过采用Bucket索引来避免状态后端的复杂调优。

如果Bucket索引+分区表的模式无法平衡Bueckt桶过大的问题，还是可以继续采用Flink状态索引，按照规范去优化对应的配置参数即可。

- 基于Bucket索引的表，按照单个Bucket 2GB数据量进行设计。

  为了规避单个Bucket过大，建议单个Bucket的数据量不要超过2GB（该2GB是指数据内容大小，不是指数据行数也不是parquet的数据文件大小），目的是将对应的桶的Parquet文件大小控制在256MB范围内（平衡读写内存消耗和HDFS存储有效利用），因此可以看出2GB的这个限制只是一个经验值，因为不同的业务数据经过列存压缩后大小是不一样的。

  为什么建议是2GB?

  - 2GB的数据存储成列存Parquet文件后，大概的数据文件大小是150MB ~ 256MB左右。不同业务数据会有出入。而HDFS单个数据块一般会是128MB，这样可以有效地利用存储空间。
  - 数据读写占用的内存空间都是原始数据大小（包括空值也是会占用内存的），2GB在大数据计算过程中，处于单task读写可接受范围之内。

  如果是单个Bucket的数据量超过了该值范围，可能会有什么影响?

  - 读写任务可能会出现OOM的问题，解决方法就是提升单个task的内存占比。
  - 读写性能下降，因为单个task的处理的数据量变大，导致处理耗时变大。

7.2.3 Hudi 表分区设计规范

规则

分区键不可以被更新：

Hudi具有主键唯一性机制，但在分区表的场景下通常只能保证分区内主键唯一，因此如果分区键的值发生变更后，会导致相同主键的行记录出现多条的情况。在以日期分区的场景，可采用数据的创建时间为分区字段，切记不要采用数据更新时间做分区。

说明

当指定Hudi的索引类型为Global索引类型时，Hudi支持跨分区进行数据更新，但Global索引性能较差一般不建议使用。

建议

- 事实表采用日期分区表，维度表采用非分区或者大颗粒度的日期分区

  是否采用分区表要根据表的总数据量、增量和使用方式来决定。从表的使用属性看事实表和维度表具有的特点：

  - 事实表：数据总量大，增量大，数据读取多以日期做切分，读取一定时间段的数据。
  - 维度表：总量相对小，增量小，多以更新操作为主，数据读取会是全表读取，或者按照对应业务ID过滤。

  基于以上考虑，维度表采用天分区会导致文件数过多，而且是全表读取，会导致所需要的文件读取Task过多，采用大颗粒度的日期分区，例如年分区，可以有效降低分区个数和文件数量；对于增量不是很大的维度表，也可以采用非分区表。如果维度表的总数据量很大或者增量也很大，可以考虑采用某个业务ID进行分区，在大部分数据处理逻辑中针对大维度表，会有一定的业务条件进行过滤来提升处理性能，这类表要结合一定的业务场景来进行优化，无法从单纯的日期分区进行优化。事实表读取方式都会按照时间段切分，近一年、近一个月或者近一天，读取的文件数相对稳定可控，所以事实表优先考虑日期分区表。

- 分区采用日期字段，分区表粒度，要基于数据更新范围确定，不要过大也不要过小。

  分区粒度可以采用年、月、日，分区粒度的目标是减少同时写入的文件桶数，尤其是在有数据量更新，且更新数据有一定时间范围规律的，比如：近一个月的数据更新占比最大，可以按照月份创建分区；近一天内的数据更新占比大，可以按照天进行分区。

  采用Bucket索引，写入是通过主键Hash打散的，数据会均匀的写入到分区下每个桶。因为各个分区的数据量是会有波动的，分区下桶的个数设计一般会按照最大分区数据量计算，这样会出现越细粒度的分区，桶的个数会冗余越多。例如：

  采用天级分区，平均的日增数据量是3GB，最多一天的日志是8GB，这个会采用Bucket桶数= 8GB/2GB = 4 来创建表；每天的更新数据占比较高，且主要分散到近一个月。这样会导致结果是，每天的数据会写入到全月的Bucket桶中，那就是4*30 = 120个桶。如果采用月分区，分区桶的个数= 3GB * 30 /2GB = 45个桶 ，这样写入的数据桶数减少到了45个桶。在有限的计算资源下，写入的桶数越少，性能越高。

7.3 Hudi 数据表管理操作规范

7.3.1 Hudi 数据表 Compaction 规范

mor表更新数据以行存log的形式写入，log读取时需要按主键合并，并且是行存的，导致log读取效率比parquet低很多。为了解决log读取的性能问题，Hudi通过compaction将log压缩成parquet文件，大幅提升读取性能。

规则

- 有数据持续写入的表，24小时内至少执行一次compaction。

  对于MOR表，不管是流式写入还是批量写入，需要保证每天至少完成1次Compaction操作。如果长时间不做compaction，Hudi表的log将会越来越大，这必将会出现以下问题：

  - Hudi表读取很慢，且需要很大的资源。 这是由于读MOR表涉及到log合并，大log合并需要消耗大量的资源并且速度很慢。
  - 长时间进行一次Compaction需要耗费很多资源才能完成，且容易出现OOM。
  - 阻塞Clean，如果没有Compaction操作来产生新版本的Parquet文件，那旧版本的文件就不能被Clean清理，增加存储压力。

- CPU与内存比例为1:4~1:8。

  Compaction作业是将存量的parquet文件内的数据与新增的log中的数据进行合并，需要消耗较高的内存资源，按照之前的表设计规范以及实际流量的波动结合考虑，建议Compaction作业CPU与内存的比例按照1:4~1:8配置，保证Compaction作业稳定运行。当Compaction出现OOM问题，可以通过调大内存占比解决。

建议

- 通过增加并发数提升Compaction性能。

  CPU和内存比例配置合理会保证Compaction作业是稳定的，实现单个Compaction task的稳定运行。但是Compaction整体的运行时长取决于本次Compaction处理文件数以及分配的cpu核数（ 并发能力 ），因此可以通过增加

Compaciton作业的CPU核的个数来提升Compaction性能（注意增加cpu也要保证CPU与内存的比例）。

- Hudi表采用异步Compaction。

  为了保证流式入库作业的稳定运行，就需要保证流式作业不在实时入库的过程中做其它任务，比如Flink写Hudi的同时会做Compaction。这看似是一个不错的方案，即完成了入库又完成Compaction。但是Compaction操作是非常消耗内存和IO的，它会给流式入库作业带来以下影响：

  - 增加端到端时延：Compaction会放大写入时延，因为Compaction比入库更耗时。
  - 作业不稳定：Compaction会给入库作业带来更多的不稳定性，Compaction OOM将会导致整个作业直接失败。

- 建议2~4小时进行一次compaction。

  Compaction是MOR表非常重要且必须执行的维护手段，对于实时任务来说，要求Compaction执行合并的过程必须和实时任务解耦，通过周期调度Spark任务来完成异步Compaction，这个方案的关键之处在于如何合理的设置这个周期，周期如果太短意味着Spark任务可能会空跑，周期如果太长可能会积压太多的Compaction Plan没有去执行而导致Spark任务耗时长并且也会导致下游的读作业时延高。对此场景，在这里给出以下建议：按照集群资源使用情况，可以每2小时或每4个小时去调度执行一次异步Compaction作业，这是一个基本的维护MOR表的方案。

- 采用Spark异步执行Compaction，不采用Flink进行Compaction。

  Flink写hudi建议的方案是Flink只负责写数据和生成Compaction计划，由单独的Spark作业异步执行compaction、clean和archive。Compaction计划的生成是轻量级的对Flink写入作业影响可以忽略。

  上述方案落地的具体步骤参考如下：

  - **Flink只负责写数据和生成Compaction计划**

    Flink流任务建表语句中添加如下参数，控制Flink任务写Hudi时只会生成Compaction plan

    ```
    'compaction.async.enabled' = 'false'       -- 关闭Flink 执行Compaction任务
    'compaction.schedule.enabled' = 'true'     -- 开启Compaction计划生成
    'compaction.delta_commits' = '5'           -- MOR表默认5次checkpoint尝试生成compaction plan，该参数需要根据具体业务调整
    'clean.async.enabled' = 'false'            -- 关闭Clean操作
    'hoodie.archive.automatic' = 'false'       -- 关闭Archive操作
    ```

  - **Spark离线完成Compaction计划的执行，以及Clean和Archive操作**

    在调度平台（可以使用华为的DataArts）运行一个定时调度的离线任务来让Spark完成Hudi表的Compaction计划执行以及Clean和Archive操作。

    ```
    set hoodie.archive.automatic = false;
    set hoodie.clean.automatic = false;
    set hoodie.archive.async = false;
    set hoodie.clean.async = false;
    set hoodie.compact.inline = true;
    set hoodie.run.compact.only.inline=true;
    set hoodie.cleaner.commits.retained = 500;  -- clean保留timeline上最新的500个deltacommit对应的数据文件，之前的deltacommit所对应的旧版本文件会被清理。该值需要大于compaction.delta_commits设置的值，需要根据具体业务调整。
    set hoodie.keep.max.commits = 700;  -- timeline最多保留700个deltacommit
    set hoodie.keep.min.commits = 501;  -- timeline最少保留500个deltacommit。该值需要大于hoodie.cleaner.commits.retained设置的值，需要根据具体业务调整。
    run compaction on <database name>. <table name>;   -- 执行Compaction计划
    run clean on <database name>. <table name>;        -- 执行Clean操作
    run archivelog on <database name>.<table name>;    -- 执行Archive操作
    ```

- 异步Compaction可以将多个表串行到一个作业，资源配置相近的表放到一组，该组作业的资源配置为最大消耗资源的表所需的资源

对于在**•Hudi表采用异步Compaction**和**•采用Spark异步执行Compaction，不...**中提到的异步Compaction任务，这里给出以下开发建议：

- 不需要对每张Hudi表都开发异步Compaction任务，这样会导致作业开发成本高，集群作业爆炸，集群资源不能有效的利用和释放。
- 异步Compaction任务可以通过执行SparkSQL来完成，多个Hudi表的Compaction、Clean和Archive可以放在同一个任务来执行，比如对table1和table2用同一个任务来执行异步维护操作：

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

7.3.2 Hudi 数据表 Clean 规范

Clean也是Hudi表的维护操作之一，该操作对于MOR表和COW表都需要执行。Clean操作的目的是为了清理旧版本文件（Hudi不再使用的数据文件），这不但可以节省Hudi表List过程的时间，也可以缓解存储压力。

规则

Hudi表必须执行Clean。

对于Hudi的MOR、COW表，都需要开启Clean。

- Hudi表在写入数据时会自动判断是否需要执行Clean，因为Clean的开关默认打开(hoodie.clean.automatic默认为true)。
- Clean操作并不是每次写数据时都会触发，至少需要满足两个条件：
  a. Hudi表中需要有旧版本的文件。对于COW表来说，只要保证数据被更新过就一定存在旧版本的文件。对于MOR表来说，要保证数据被更新过并且做过Compaction才能有旧版本的文件。
  b. Hudi表满足hoodie.cleaner.commits.retained设置的阈值。如果是Flink写hudi，则至少提交的checkpoint要超过这个阈值；如果是批写Hudi，则批写次数要超过这个阈值。

建议

- MOR表下游采用批量读模式，采用clean的版本数为compaction版本数+1。
  MOR表一定要保证Compaction Plan能够被成功执行，Compaction Plan只是记录了Hudi表中哪些Log文件要和哪些Parquet文件合并，所以最重要的地方在于保证Compaction Plan在被执行的时候它需要合并的文件都存在。而Hudi表中只有Clean操作可以清理文件，所以建议Clean的触发阈值（hoodie.cleaner.commits.retained的值）至少要大于Compaction的触发阈值（对于Flink任务来说就是compaction.delta_commits的值）。
- MOR表下游采用流式计算，历史版本保留小时级。

如果MOR表的下游是流式计算，例如Flink流读，可以按照业务需要保留小时级的历史版本，这样的话近几个小时之内的增量数据可以通过log文件读出，如果保留时长过短，下游flink作业在重启或者异常中断阻塞的情况下，上游增量数据已经Clean掉了，flink需要从parquet文件读增量数据，性能会有下降；如果保留时间过长，会导致log里面的历史数据冗余存储。

具体可以按照下面的计算公式来保留2个小时的历史版本数据：

版本数设置为3600*2/版本interval时间，版本interval时间来自于flink作业的checkpoint周期，或者上游批量写入的周期。

- COW表如果业务没有历史版本数据保留的特殊要求，保留版本数设置为1。

  COW表的每个版本都是表的全量数据，保留几个版本就会冗余多少个版本。因此如果业务无历史数据回溯的需求，保留版本数设置为1，也就是保留当前最新版本

- clean作业每天至少执行一次，可以2~4小时执行一次。

  Hudi的MOR表和COW表都需要保证每天至少1次Clean，MOR表的Clean可以参考2.2.1.6小节和Compaction放在一起异步去执行。COW的Clean可以在写数据时自动判断是否执行。

7.3.3 Hudi 数据表 Archive 规范

Archive（归档）是为了减轻Hudi读写元数据的压力，所有的元数据都存放在这个路径：Hudi表根目录/.hoodie目录，如果.hoodie目录下的文件数量超过10000就会发现Hudi表有非常明显的读写时延。

规则

Hudi表必须执行Archive。

对于Hudi的MOR类型和COW类型的表，都需要开启Archive。

- Hudi表在写入数据时会自动判断是否需要执行Archive，因为Archive的开关默认打开(hoodie.archive.automatic默认为true)。
- Archive操作并不是每次写数据时都会触发，至少需要满足以下两个条件：
  a. Hudi表满足hoodie.keep.max.commits设置的阈值。如果是Flink写hudi至少提交的checkpoint要超过这个阈值；如果是Spark写hudi，写Hudi的次数要超过这个阈值。
  b. Hudi表做过Clean，如果没有做过Clean就不会执行Archive（MRS 3.3.1-LTS及以后版本，忽略此项条件）。

建议

Archive作业每天至少执行一次，可以2~4小时执行一次。

Hudi的MOR表和COW表都需要保证每天至少1次Archive，MOR表的Archive可以参考2.2.1.6小节和Compaction放在一起异步去执行。COW的Archive可以在写数据时自动判断是否执行。

7.4 Spark on Hudi 开发规范

7.4.1 SparkSQL 建表参数规范

规则

- 建表必须指定primaryKey和preCombineField。

  Hudi表提供了数据更新的能力和幂等写入的能力，该能力要求数据记录必须设置主键用来识别重复数据和更新操作。不指定主键会导致表丢失数据更新能力，不指定preCombineField会导致主键重复。

参数名称	参数描述	输入值	说明
primaryKey	hudi主键	按需	必须指定，可以是复合主键但是必须全局唯一。
preCombineField	预合并键，相同主键的多条数据按该字段进行合并	按需	必须指定，相同主键的数据会按该字段合并，不能指定多个字段。

- 禁止建表时将hoodie.datasource.hive_sync.enable指定为false。

  指定为false将导致新写入的分区无法同步到Hive Metastore中。由于缺失新写入的分区信息，查询引擎读取该时会丢数。

- 禁止指定Hudi的索引类型为INMEMORY类型。

  该索引仅是为了测试使用。生产环境上使用该索引将导致数据重复。

建表示例

```
create table data_partition(id int, comb int, col0 int,yy int, mm int, dd int)
using hudi                              --指定hudi 数据源
partitioned by(yy,mm,dd)                --指定分区， 支持多级分区
location '/opt/log/data_partition'      --指定路径，如果不指定建表在hive warehouse里
options(
type='mor',                             --表类型 mor 或者 cow
primaryKey='id',                        --主键，可以是复合主键但是必须全局唯一
preCombineField='comb'                  --预合并字段，相同主键的数据会按该字段合并，当前不能指定多个字段
)
```

7.4.2 Spark 增量读取 Hudi 参数规范

规则

增量查询之前必须指定当前表的查询为增量查询模式，并且查询后重写设置表的查询模式

如果增量查询完，不重新将表查询模式设置回去，将影响后续的实时查询

示例

```
set hoodie.tableName.consume.mode=INCREMENTAL;--必须设置当前表读取为增量读取模式。
set hoodie.tableName.consume.start.timestamp=20201227153030;--指定初始增量拉取commit。
set hoodie.tableName.consume.end.timestamp=20210308212318;  --指定增量拉取结束commit，如果不指定的话采用最新的commit。
select * from tableName where `_hoodie_commit_time`>'20201227153030' and
`_hoodie_commit_time`<='20210308212318'; --结果必须根据start.timestamp和end.timestamp进行过滤，如果
```



```
没有指定end.timestamp，则只需要根据start.timestamp进行过滤。
set hoodie.tableName.consume.mode=SNAPSHOT；--使用完增量模式，必须把查询模式重新设置回来。
```

7.4.3 Spark 异步任务执行表 compaction 参数设置规范

- 写作业未停止情况下，禁止手动执行run schedule命令生成compaction计划。

  错误示例：

  ```
  run schedule on dsrTable
  ```

  如果还有别的任务在写这张表，执行该操作会导致数据丢失。

- 执行run compaction命令时，禁止将hoodie.run.compact.only.inline设置成false，该值需要设置成true。

  错误示例：

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

  如果还有别的任务在写这张表，执行上述操作会导致数据丢失。

  正确示例：异步Compaction

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

7.4.4 Spark on Hudi 表数据维护规范

禁止通过Alter命令修改表关键属性信息：type/primaryKey/preCombineField/hoodie.index.type

错误示例，执行如下语句修改表关键属性：

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

Hive/Presto等引擎可以直接修改表属性，但是这种修改会导致整个Hudi表出现数据重复，甚至数据损坏；因此禁止修改上述属性。

7.4.5 Spark 并发写 Hudi 建议

- 涉及到并发场景，推荐采用分区间并发写的方式：即不同的写入任务写不同的分区

  分区并发参数控制：

  - SQL方式：

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

  - DataSource Api方式：

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

  说明

  所有参与分区间并发写入的任务，都必须配置上述参数。

- 不建议同分区内并发写，这种并发写入需要开启Hudi OCC方式并发写入，必须严格遵守并发参数配置，否则会出现表数据损坏的问题。

并发OCC参数控制：

- SQL方式：

```
--开启OCC。
set hoodie.write.concurrency.mode=optimistic_concurrency_control;
set hoodie.cleaner.policy.failed.writes=LAZY;

--开启并发ZooKeeper锁。
set
hoodie.write.lock.provider=org.apache.hudi.client.transaction.lock.ZookeeperBasedLockProvider;
--设置使用ZooKeeper锁。
set hoodie.write.lock.zookeeper.url=<zookeeper_url>;  --设置使用ZooKeeper地址。
set hoodie.write.lock.zookeeper.port=<zookeeper_port>; --设置使用ZooKeeper端口。
set hoodie.write.lock.zookeeper.lock_key=<table_name>;  --设置锁名称。
set hoodie.write.lock.zookeeper.base_path=<table_path>; --设置zk锁路径。
```

- DataSource Api方式：

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

说明

1. 所有参与并发写入的任务，都必须配置上述参数。OCC不会保证所有参与并发写入的任务都执行成功;当出现多个写任务更新同一个文件时，只有一个任务可以成功，其余失败。
2. 并发场景下，需要设置cleaner policy为Lazy，因此无法自动清理垃圾文件。

7.4.6 Spark 读写 Hudi 资源配置建议

- Spark读写Hudi任务资源配置规则，内存和CPU核心的比例2:1，堆外内存和CPU核心比例0.5:1；即一个核心，需要2G堆内存，0.5G堆外内存

  说明

  Spark初始化入库场景，由于处理的数据量比较大，上述资源配比需要调整，内存和Core的比例推荐4:1，堆外内存和Core的比例1:1。

  示例：

```
spark-submit
--master yarn-cluster
--executor-cores  2                         --核心
--executor-memory 4g                          --堆内存
--conf spark.executor.memoryOverhead=1024    --堆外内存
```

- 基于Spark进行ETL计算，CPU核心 ： 内存比例建议>1:2，推荐1：4~1：8

  上一个规则是指纯读写的资源配比，如果Spark的作业除了读写还有业务逻辑计算，该过程会导致需要内存增加，因此建议CPU核心与内存的比例大于1：2，如果逻辑比较复杂适当调大内存，这要基于实际情况进行调整。一般默认推荐配置为1：4~1：8。

- 针对bucket表的写入资源配置，建议给的CPU核心数量不小于桶数目（分区表每次可能写入多个分区，理想情况下建议给的CPU核心数量=写入分区*分桶数；实际配置的core小于这个值，写入性能线性下降）。

  示例：

当前表bucket数为3，同时写入分区数为2， 建议入库Spark任务配置的core数量大于等于3*2。

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

以上配置代表excutor-num*executor-cores=6 >=分区数*分桶数=6。

7.4.7 Spark On Hudi 性能调优

优化 Spark Shuffle 参数提升 Hudi 写入效率

- 开启spark.shuffle.readHostLocalDisk=true，本地磁盘读取shuffle数据，减少网络传输的开销。
- 开启spark.io.encryption.enabled=false，关闭shuffle过程写加密磁盘，提升shuffle效率。
- 开启spark.shuffle.service.enabled=true，启动shuffle服务，提升任务shuffle的稳定性。

配置项	集群默认值	调整后
--conf spark.shuffle.readHostLocalDisk	false	true
--conf spark.io.encryption.enabled	true	false
--conf spark.shuffle.service.enabled	false	true

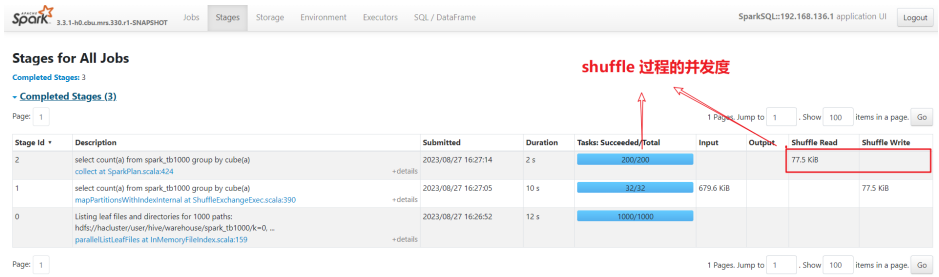
调整 Spark 调度参数优化 OBS 场景下 Spark 调度时延

- 开启对于OBS存储，可以关闭Spark的本地性进行优化，尽可能提升Spark调度效率

配置项	集群默认值	调整后
--conf spark.locality.wait	3s	0s
--conf spark.locality.wait.process	3s	0s
--conf spark.locality.wait.node	3s	0s
--conf spark.locality.wait.rack	3s	0s

优化 shuffle 并行度，提升 Spark 加工效率

所谓的shuffle并发度如下图所示：



集群默认是200，作业可以单独设置。如果发现瓶颈stage（执行时间长），且分配给当前作业的核数大于当前的并发数，说明并发度不足。通过以下配置优化。

场景	配置项	集群默认值	调整后
Jar作业	spark.default.parallelism	200	按实际作业可用资源2倍设置
SQL作业	spark.sql.shuffle.partitions	200	按实际作业可用资源2倍设置
hudi入库作业	hoodie.upsert.shuffle.parallelism	200	非bucket表使用，按实际作业可用资源2倍设置

注意

动态资源调度情况下（spark.dynamicAllocation.enabled= true）时，资源按照spark.dynamicAllocation.maxExecutors评估。

Bucket 表，可以开启桶裁剪提升主键点查效率

示例：

业务经常使用主键id作为查询条件，执行点查；比如select xxx where id = idx ... 。

建表时，可以加入如下属性，提升查询效率。默认配置下属性值等于primaryKey，即主键。

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

初始化 Hudi 表时，可以使用 BulkInsert 方式快速写入数据

示例：

```
set hoodie.combine.before.insert=true;                --入库前去重，如果数据没有重复 该参数无需设置。
set hoodie.datasource.write.operation = bulk_insert;  --指定写入方式为bulk insert方式。
set hoodie.bulkinsert.shuffle.parallelism = 4;        --指定bulk_insert写入时的并行度，等于写入完成后保存的分区parquet文件数。
insert into dsrTable select * from srcTabble
```

开启 log 列裁剪，提升 mor 表查询效率

mor表读取的时候涉及到Log和Parquet的合并，性能不是很理想。可以开启log列裁剪减少合并时IO读取开销

SparkSQL执行查询，先执行：

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

Spark 加工 Hudi 表时其他参数优化

- 设置spark.sql.enableToString=false，降低Spark解析复杂SQL时候内存使用，提升解析效率。
- 设置spark.speculation=false，关闭推测执行，开启该参数会带来额外的cpu消耗，同时Hudi不支持启动该参数，启用该参数写Hudi有概率导致文件损坏。

配置项	集群默认值	调整后
--conf spark.sql.enableToString	true	false
--conf spark.speculation	false	false

7.5 Bucket 调优示例

7.5.1 创建 Bucket 索引表调优

Bucket索引常用设置参数：

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

判断使用分区表还是非分区表

根据表的使用场景一般将表分为事实表和维度表：

- 事实表通常整表数据规模较大，以新增数据为主，更新数据占比小，且更新数据大多落在近一段时间范围内（年或月或天），下游读取该表进行ETL计算时通常会使用时间范围进行裁剪（例如最近一天、一月、一年），这种表通常可以通过数据的创建时间来做分区以保证最佳读写性能。
- 维度表数据量一般整表数据规模较小，以更新数据为主，新增较少，表数据量比较稳定，且读取时通常需要全量读取做join之类的ETL计算，因此通常使用非分区表性能更好。
- **分区表的分区键不允许更新，否则会产生重复数据。**

例外场景：超大维度表和超小事实表

特殊情况如存在**持续大量新增数据的维度表**（表数据量在200G以上或日增长量超过60M）或**数据量非常小的事实表**（表数据量小于10G且未来三至五年增长后也不会超过10G）需要针对具体场景来进行例外处理：

- 持续大量新增数据的维度表

  方法一：预留桶数，如使用非分区表则需通过预估较长一段时间内的数据增量来预先增加桶数，**缺点是随着数据的增长，文件依然会持续膨胀**；

方法二：大粒度分区（推荐），如果使用分区表则需要根据数据增长情况来计算，例如使用年分区，**这种方式相对麻烦些但是多年后表无需重新导入**。

方法三：数据老化，按照业务逻辑分析大的维度表是否可以通过数据老化清理无效的维度数据从而降低数据规模。

- 数据量非常小的事实表

  这种可以在预估很长一段时间的数据增长量的前提下使用非分区表预留稍宽裕一些的桶数来提升读写性能。

确认表内桶数

Hudi表的桶数设置，关系到表的性能，需要格外引起注意。

以下几点，是设置桶数的关键信息，需要建表前确认。

- 非分区表
  a. 单表数据总条数 = select count(1) from tablename（入湖时需提供）；
  b. 单条数据大小 = 平均 1KB（华为建议通过select * from tablename limit 100 将查询结果粘贴在notepad++中得出100条数据的大小再除以100得到单条平均大小）
  c. 单表数据量大小(G) = 单表数据总条数*单表数据大小/1024/1024
  d. 非分区表桶数 = MAX（单表数据量大小(G)/2G*2，再向上取整，4）
- 分区表
  a. 最近一个月最大数据量分区数据**总条数** = 入湖前咨询产品线
  b. 单条数据大小 = 平均 1KB（华为建议通过select * from tablename limit 100 将查询结果粘贴在notepad++中得出100条数据的大小再除以100得到单条平均大小）
  c. 单分区数据量大小(G) = 最近一个月最大数据量分区数据总条数*单表数据大小/1024/1024
  d. 分区表桶数 = MAX（单分区数据量大小(G)/2G，再后向上取整，1）

⚠ 注意

1. 需要使用的是表的总数据大小，而不是压缩以后的文件大小
2. 桶的设置以偶数最佳，非分区表最小桶数请设置4个，分区表最小桶数请设置1个。

确认建表 SQL

DataArts支持通过Spark JDBC方式和Spark API方式操作Hudi表：

- Spark JDBC方式使用公用资源，不用单独起Spark作业，但是不能指定执行SQL所需要的资源以及配置参数，因此建议用来做建表操作或小数据量的查询操作。
- Spark API方式执行的SQL独立起Spark作业，有一定的耗时，但是可以通过配置运行程序参数来指定作业所需要的资源等参数，建议批量导入等

作业使用API方式来指定资源运行，防止占用jdbc资源长时间阻塞其他任务。

注意

DataArts使用Spark API方式操作Hudi表，必须要添加参数--conf spark.support.hudi=true，并且通过执行调度来运行作业。

使用 DataArts 创建 Hudi 表

DataArts支持通过Spark JDBC方式和Spark API方式操作Hudi表：

- Spark JDBC方式使用公用资源，不用单独起Spark作业，但是不能指定执行SQL所需要的资源以及配置参数，因此建议用来做建表操作或小数据量的查询操作。
- Spark API方式执行的SQL独立起Spark作业，有一定的耗时，但是可以通过配置运行程序参数来指定作业所需要的资源等参数，建议批量导入等

作业使用API方式来指定资源运行，防止占用jdbc资源长时间阻塞其他任务。

注意

DataArts使用Spark API方式操作Hudi表，必须要添加参数--conf spark.support.hudi=true，并且通过执行调度来运行作业。

7.5.2 Hudi 表初始化

1. 初始化导入存量数据通常由Spark作业来完成，由于初始化数据量通常较大，因此推荐使用API方式给充足资源来完成。
2. 对于批量初始化后需要接Flink或Spark流作业实时写入的场景，一般建议通过对上有消息进行过滤，从一个指定的时间范围开始消费来控制数据的重复接入量（例如Spark初始化完成后，Flink消费Kafka时过滤掉2小时之前的数据），如果无法对kafka消息进行过滤，则可以考虑先实时接入生成offset，再truncate table，再历史导入，再开启实时。

说明

1. 如果批量初始化前表里已经存在数据且没有truncate table，则会导致批量数据写成非常大的log文件，对后续compaction形成很大压力需要更多资源才能完成
2. Hudi表在Hive元数据中，应该会存在1张内部表（手动创建），2张外部表（写入数据后自动创建）。
3. 2张外部表，表名_ro（用户只读合并后的parquet文件，即读优化视图表），_rt（读实时写入的最新版本数据，即实时视图表）。

7.5.3 实时任务接入

实时作业一般由Flink Sql或Sparkstreaming来完成，流式实时任务通常配置同步生成compaction计划，异步执行计划。

- Flink SQL作业中sink端Hudi表相关配置如下：

```
create table denza_hudi_sink (
$HUDI_SINK_SQL_REPLACEABLE$
) PARTITIONED BY (
years,
months,
days
```

```
) with (
'connector' = 'hudi',                                          --指定写入的是Hudi表。
'path' = 'obs://XXXXXXXXXXXXXXXXXX/',                          --指定Hudi表的存储路径。
'table.type' = 'MERGE_ON_READ',                                --Hudi表类型。
'hoodie.datasource.write.recordkey.field' = 'id',              --主键。
'write.precombine.field' = 'vin',                              --合并字段。
'write.tasks' = '10',                                          --flink写入并行度。
'hoodie.datasource.write.keygenerator.type' = 'COMPLEX',       --指定KeyGenerator，与Spark创建的Hudi表类型一致。
'hoodie.datasource.write.hive_style_partitioning' = 'true',    --使用hive支持的分区格式。
'read.streaming.enabled' = 'true',                             --开启流读。
'read.streaming.check-interval' = '60',                        --checkpoint间隔，单位为秒。
'index.type'='BUCKET',                                         --指定Hudi表索引类型为BUCKET。
'hoodie.bucket.index.num.buckets'='10',                        --指定bucket桶数。
'compaction.delta_commits' = '3',                              --compaction生成的commit间隔。
'compaction.async.enabled' = 'false',                          --compaction异步执行关闭。
'compaction.schedule.enabled' = 'true',                        --compaction同步生成计划。
'clean.async.enabled' = 'false',                               --异步clean关闭。
'hoodie.archive.automatic' = 'false',                          --自动archive关闭。
'hoodie.clean.automatic' = 'false',                            --自动clean关闭。
'hive_sync.enable' = 'true',                                   --自动同步hive表。
'hive_sync.mode' = 'jdbc',                                     --同步hive表方式为jdbc。
'hive_sync.jdbc_url' = '',                                     --同步hive表的jdbc url。
'hive_sync.db' = 'hudi_cars_byd',                              --同步hive表的database。
'hive_sync.table' = 'byd_hudi_denza_1s_mor',                   --同步hive表的tablename。
'hive_sync.metastore.uris' = 'thrift://XXXXX:9083 ',           --同步hive表的metastore uri。
'hive_sync.support_timestamp' = 'true',                        --同步hive表支持timestamp格式。
'hive_sync.partition_extractor_class' = 'org.apache.hudi.hive.MultiPartKeysValueExtractor'    --同步hive表的extractor类。
);
```

- Spark streaming写入Hudi表常用的参数如下（参数意义与上面flink类似，不再做注释）：

```
hoodie.table.name=
hoodie.index.type=BUCKET
hoodie.bucket.index.num.buckets=3
hoodie.datasource.write.precombine.field=
hoodie.datasource.write.recordkey.field=
hoodie.datasource.write.partitionpath.field=
hoodie.datasource.write.table.type= MERGE_ON_READ
hoodie.datasource.write.hive_style_partitioning=true
hoodie.compact.inline=true
hoodie.schedule.compact.only.inline=true
hoodie.run.compact.only.inline=false
hoodie.clean.automatic=false
hoodie.clean.async=false
hoodie.archive.async=false
hoodie.archive.automatic=false
hoodie.compact.inline.max.delta.commits=50
hoodie.datasource.hive_sync.enable=true
hoodie.datasource.hive_sync.partition_fields=
hoodie.datasource.hive_sync.database=
hoodie.datasource.hive_sync.table=
hoodie.datasource.hive_sync.partition_extractor_class=org.apache.hudi.hive.MultiPartKeysValueExtractor
```

7.5.4 离线 Compaction 配置

对于MOR表的实时业务，通常设置在写入中同步生成compaction计划，因此需要额外通过DataArts或者脚本调度SparkSQL去执行已经产生的compaction计划。

- 执行参数

```
set hoodie.compact.inline = true;                --打开compaction操作。
set hoodie.run.compact.only.inline = true;       --compaction只执行已生成的计划，不产生新计划。
set hoodie.cleaner.commits.retained = 120;       --清理保留120个commit。
set hoodie.keep.max.commits = 140;               --归档最大保留140个commit。
set hoodie.keep.min.commits = 121;               --归档最小保留121个commit。
set hoodie.clean.async = false;                  --关闭异步清理。
```

```
set hoodie.clean.automatic = false;          --关闭自动清理，防止compaction操作触发clean。
set hoodie.archive.async = false;           --关闭异步归档。
set hoodie.archive.automatic = false;        --关闭自动归档。

run compaction on $tablename;                --执行compaction计划。
run clean on $tablename;                    --执行clean操作清理冗余版本。
run archivelog on $tablename;                --执行archivelog合并清理元数据文件。
```

⚠ 注意

1. 关于清理、归档参数的值不宜设置过大，会影响Hudi表的性能，通常建议：
   hoodie.cleaner.commits.retained = compaction所需要的commit数的2倍
   hoodie.keep.min.commits = hoodie.cleaner.commits.retained + 1
   hoodie.keep.max.commits = hoodie.keep.min.commits + 20
2. 执行compaction后再执行clean和archive，由于clean和archivelog对资源要求较小，为避免资源浪费，使用DataArts调度的话可以compaction作为一个任务，clean、archive作为一个任务分别配置不同的资源执行来节省资源使用。

- 执行资源
  a. Compaction调度的间隔应小于Compaction计划生成的间隔，例如1小时左右生成一个Compaction计划的话，执行Compaction计划的调度任务应该至少半小时调度一次。
  b. Compaction作业配置的资源，vcore数至少要大于等于单个分区的桶数，vcore数与内存的比例应为1：4即1个vcore配4G内存。

8 Impala 应用开发规范

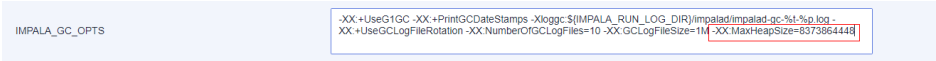
8.1 Impala 应用开发规则

创建集群时只需指定一个 Catalog 和一个 StoreStore

如果已经创建了两个Catalog和StateStore，Impalad角色需要指定--catalog_service_host和--state_store_host，Catalog角色需要指定--state_store_host。

Impalad（Coordinator）角色的 jvm 内存要大于或等于 Catalog 角色的 jvm 内存

Impala的元数据存放在内存中，Impalad需要从Catalog同步全量元数据，要保证Impala的jvm内存大于Catalog的jvm内存，才可以容纳下这些元数据。



建表时分区不要超过 10 万个，分区太多会影响元数据加载速度，阻塞查询

Impala元数据和分区、文件数量正相关，太多分区会导致Impala元数据占用内存过大，刷新元数据时需要扫描的分区文件就越多，极大地降低查询效率。

建表时整数类型的分区键不补前置 0，例如'hour=01'等分区

整数类型分区使用补齐前缀0的方式，会导致Impala解析分区不准确，影响元数据刷新。

列名、别名无特殊情况使用英文，不使用中文

除注释外，由于中文编码存在特殊字符，使用中文会导致impala解析时遇到不能识别的符号，从而出现解析失败或进入死循环。

包含 case when 子句的 view 视图或子查询，不应嵌套超过 3 层，避免出现嵌套过深导致 Impala 内存溢出

case when子句包含多个判断分支，在多层view视图或子查询嵌套场景下，复杂度呈指数增长，通过实测该场景下嵌套层数不能超过3层，否则会出现内存溢出。可使用临时表替代view或子查询，将一个多重嵌套拆分成多个查询执行。

分区表 select * 必须带上分区键

分区表查询select * 不带分区键，会Impala触发全表，极大地占用计算资源，非必要场景下请按分区查询。

8.2 Impala 应用开发建议

Coordinator 和 Executor 分离部署，Coordinator 根据集群规模部署 2-5 个

Coordinator承担缓存元数据，解析SQL执行计划，和响应客户端请求的功能主要使用jvm内存，而Executor承担数据读写，算子计算等功能，主要使用offheap内存；拆分后可有效提升内存使用率；另外，所有的SQL执行统计均在Coordinator中记录，分离后可通过访问几个Coordinator节点获取整个集群的SQL运行情况，可减少运维压力。

根据业务需求，配置 impala 资源池和资源队列，核心业务使用单独的队列隔离，并配置 mem_limit 和 exec_time_limit_s 避免大查询

使用资源队列可避免不同业务相互抢占资源，相互影响，具体请参考**Impala启用并配置动态资源池**。

OBS 存储开启本地缓存

OBS数据存储场景可根据业务需求配置本地缓存，提升读取速率，配置单盘100GB本地缓存示例：—data_cache=/srv/BigData/data1/impala:100GB

HDFS 存储开启短路读

HDFS存储场景下可开启短路读，提升读取速率，具体请参考：https://impala.apache.org/docs/build/html/topics/impala_config_performance.html

新建表，新增分区等表结构变动操作后，执行 Invalidate metadata <table>, 在数据入库/湖后，对于发生变化的表/分区进行主动 refresh 更新 impala 元数据

在非Impala引擎（Hive，Spark等）新建、修改表，需要在Impala侧执行Invalidate metadata <table>同步表schema信息，需要查询该表时才会同步全量元数据；而新增分区，插入数据等场景可主动执行refresh即可增量更新元数据。

定时使用 compute increment stats <table_name>刷新常用表的统计信息，加速查询

Impala依赖表统计信息对查询消耗的资源做预估，准确的统计信息有利于Impala更合理地解析执行计划，分配资源。

定时进行小文件合并，减少单表的文件数量，提升元数据加载速率

Impala元数据和分区、文件数量正相关，太多分区会导致Impala元数据占用内存过大，刷新元数据时需要扫描的分区文件就越多，极大地降低查询效率。

建表时存储类型建议选择 orc 或者 parquet

orc和parquet是列式存储格式，读取效率更高，而且有更高的压缩率，可有效降低数据存储空间。

9 IoTDB 应用开发规范

9.1 IoTDB 应用开发规则

设置合理数量的存储组

设置合理数量的存储组可以带来性能的提升。既不会因为产生过多的存储文件（夹）导致频繁切换IO降低系统速度（并且会占用大量内存且出现频繁的内存-文件切换），也不会因为过少的存储文件夹（降低了并发度从而）导致写入命令阻塞。

应根据自己的数据规模和使用场景，平衡存储文件的存储组设置，以达到更好的系统性能。

所有的时间序列必须以 root 开始、以传感器作为结尾。

时间序列可以被看作产生时序数据的传感器所在的完整路径，在IoTDB中所有的时间序列必须以root开始、以传感器作为结尾。

9.2 IoTDB 应用开发建议

推荐使用原生接口 Session，避免 SQL 拼接

关于IoTDB Session接口样例，安全模式集群可参考**IoTDB Session程序**章节，普通模式集群可参考**IoTDB Session程序**章节。

根据业务情况推荐优先使用性能高的写入接口

写入接口性能由高到低排序如下：

insertTablets（多设备多行同列）>

insertTablet（单设备多行同列）>

insertRecordsOfOneDevice （单设备多行不同列）>

insertRecords(Object value) （多设备多行不同列）>

insertRecords(String value) （多设备多行不同列）>

insertRecord（单设备一行）

避免并发使用同一个客户端连接

IoTDB客户端只能连接一个IoTDBServer，大量并发使用同一个客户端会对该客户端连接的IoTDBServer造成压力，可以根据业务需求连接多个不同的客户端来达到负载均衡。

使用 SessionPool 复用连接

分布式在Session内部做了缓存，实现客户端时避免每次读写都新建Session，或者使用SessionPool进行复用连接。

查询结果集 ResultSet、SessionDataSet 使用完成后注意关闭

查询结果集ResultSet、SessionDataSet使用完成后需要关闭，否则会造成服务资源浪费。

10 Kafka 应用开发规范

10.1 Kafka 应用开发规则

调用 Kafka API（AdminZkClient.createTopic）创建 Topic

- 对于Java开发语言，正确示例：

```
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
$.MODULE$);
...
```

- 对于Scala开发语言，正确示例：

```
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)
```

Partition 的副本数不要超过节点个数

Kafka中Topic的Partition的副本是为了提升数据的可靠性而存在的，同一个Partition的副本会分布在不同的节点，因此副本数不允许超过节点个数。

Consumer 客户端的配置参数“fetch.message.max.bytes”大小

Consumer客户端的配置参数“fetch.message.max.bytes”必须大于等于Producer客户端每次产生的消息最大字节数。如果参数的值太小，可能导致Producer产生的消息无法被Consumer成功消费。

10.2 Kafka 应用开发建议

同一个组的消费者的数量建议与待消费的 Topic 下的 Partition 数保持一致

若同一个组的消费者数量多于Topic的Partition数时，会有多余的消费者一直无法消费该Topic的消息，若消费者数量少于Topic的Partition数时，并发消费得不到完全体现，因此建议两者相等。

避免写入单条记录超大的数据

单条记录超大的数据在影响处理效率的同时还可能写入失败，此时需要在初始化Kafka生产者实例时根据情况调整“max.request.size ”值，在初始化消费者实例时调整“max.partition.fetch.bytes”值。

例如，参考本例，可以将max.request.size 、max.partition.fetch.bytes配置项设置为“5252880”：

```
// 协议类型:当前支持配置为SASL_PLAINTEXT或者PLAINTEXT
props.put(securityProtocol, kafkaProc.getValues(securityProtocol, "SASL_PLAINTEXT"));
// 服务名
props.put(saslKerberosServiceName, "kafka");
props.put("max.request.size", "5252880");
// 安全协议类型
props.put(securityProtocol, kafkaProc.getValues(securityProtocol, "SASL_PLAINTEXT"));
// 服务名
props.put(saslKerberosServiceName, "kafka");
props.put("max.partition.fetch.bytes","5252880");
```


11 Mapreduce 应用开发规范

11.1 Mapreduce 应用开发规则

继承 Mapper 抽象类实现

在Mapreduce任务的Map阶段，会执行map()及setup()方法。

正确示例：

```
public static class MapperClass extends

Mapper<Object, Text, Text, IntWritable> {
/**
* map的输入，key为原文件位置偏移量，value为原文件的一行字符数据。
* 其map的输入key，value为文件分割方法InputFormat提供，用户不设置，默认 * 使用TextInputFormat。
*/
public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
//自定义的实现
}
/**
* setup()方法只在进入map任务的map()方法之前或者reduce任务的reduce()方法之前调用一次
*/
public void setup(Context context) throws IOException,
InterruptedException {
//自定义的实现
}
}
```

继承 Reducer 抽象类实现

在Mapreduce任务的Reduce阶段，会执行reduce()及setup()方法。

正确示例：

```
public static class ReducerClass extends

Reducer<Text, IntWritable, Text, IntWritable> {

/**
* @param 输入为一个key和value值集合迭代器。
* 由各个map汇总相同的key而来。reduce方法汇总相同key的个数。
* 并调用context.write(key, value)输出到指定目录。
* 其reduce的输出的key，value由Outputformat写入文件系统。
```



```
* 默认使用TextOutputFormat写入HDFS。
*/

public void reduce(Text key, Iterable<IntWritable> values,
Context context) throws IOException, InterruptedException {
//自定义实现
}

/**
* setup()方法只在进入map任务的map()方法之前或者reduce任务的reduce()方法之前调用一次。
*/

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

// 自定义实现，Context可以获得配置信息。

}
}
```

提交一个 Mapreduce 任务

main()方法创建一个job，指定参数，提交作业到hadoop集群。

正确示例：

```
public static void main(String[] args) throws Exception {
Configuration conf = getConfiguration();
// main方法输入参数：args[0]为样例MR作业输入路径，args[1]为样例MR作业输出路径
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");
// 设置找到主任务所在的jar包。
job.setJar("D:\\job-examples.jar");
// job.setJarByClass(TestWordCount.class);
// 设置运行时执行map，reduce的类，也可以通过配置文件指定。
job.setMapperClass(TokenizerMapperV1.class);
job.setReducerClass(IntSumReducerV1.class);
// 设置combiner类，默认不使用，使用时通常使用和reduce一样的类，Combiner类需要谨慎使用，也可以通过
配置文件指定。
job.setCombinerClass(IntSumReducerV1.class);
// 设置作业的输出类型，也可以通过配置文件指定。
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
// 设置该job的输入输出路径，也可以通过配置文件指定。
Path outputPath = new Path(otherArgs[1]);
FileSystem fs = outputPath.getFileSystem(conf);
// 如果输出路径已存在，删除该路径。
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);
}
```

11.2 Mapreduce 应用开发建议

全局使用的配置项，在“mapred-site.xml”配置文件中指定。

如下示例给出接口所对应的“mapred-site.xml”中的配置项。

示例：


```
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”
```


12 Spark 应用开发规范

12.1 Spark 应用开发规则

Spark 应用中，需引入 Spark 的类

- 对于Java开发语言，正确示例：

```
// 创建SparkContext时所需引入的类。
import org.apache.spark.api.java.JavaSparkContext
// RDD操作时引入的类。
import org.apache.spark.api.java.JavaRDD
// 创建SparkConf时引入的类。
import org.apache.spark.SparkConf
```

- 对于Scala开发语言，正确示例：

```
// 创建SparkContext时所需引入的类。
import org.apache.spark.SparkContext
// RDD操作时引入的类。
import org.apache.spark.SparkContext._
// 创建SparkConf时引入的类。
import org.apache.spark.SparkConf
```

分布式模式下，应注意 Driver 和 Executor 之间的参数传递

在Spark编程时，总是有一些代码逻辑中需要根据输入参数来判断，这种时候往往会使用这种方式，将参数设置为全局变量，先给定一个空值（null），在main函数中，实例化SparkContext对象之前对这个变量赋值。然而，在分布式模式下，执行程序的jar包会被发送到每个Executor上执行。而该变量只在main函数的节点改变了，并未传给执行任务的函数中，因此Executor将会报空指针异常。

正确示例：

```
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 …;
  }
}
```

错误示例：

```
//定义对象。
object Test
{
  // 定义全局变量，赋为空值（null）；在main函数中，实例化SparkContext对象之前对这个变量赋值。
  private var testArg: String = null;
  // main函数
  def main(args: Array[String])
  {

    testArg = …;
    val sc: SparkContext = new SparkContext(…);

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

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

运行错误示例，在Spark的local模式下能正常运行，而在分布式模式情况下，会在蓝色代码处报错，提示空指针异常，这是由于在分布式模式下，执行程序的jar包会被发送到每个Executor上执行，当执行到testFun函数时，需要从内存中取出testArg的值，但是testArg的值只在启动main函数的节点改变了，其他节点无法获取这些变化，因此它们从内存中取出的就是初始化这个变量时的值null，这就是空指针异常的原因。

应用程序结束之前必须调用 SparkContext.stop

利用spark做二次开发时，当应用程序结束之前必须调用SparkContext.stop()。

说明

利用Java语言开发时，应用程序结束之前必须调用JavaSparkContext.stop()。

利用Scala语言开发时，应用程序结束之前必须调用SparkContext.stop()。

以Scala语言开发应用程序为例，分别介绍下正确示例与错误示例。

正确示例：

```
//提交spark作业
val sc = new SparkContext(conf)

//具体的任务
...

//应用程序结束
sc.stop()
```

错误示例：

```
//提交spark作业
val sc = new SparkContext(conf)

//具体的任务
...
```


如果不添加SparkContext.stop，YARN界面会显示失败。如图12-1，同样的任务，前一个程序是没有添加SparkContext.stop，后一个程序添加了SparkContext.stop()。

图 12-1 添加 SparkContext.stop()和不添加的区别

合理规划 AM 资源占比

任务数量较多且每个任务占用的资源较少时，可能会出现集群资源足够，提交的任务成功但是无法启动，此时可以提高AM的最大资源占比。

图 12-2 修改 AM 最大资源百分比

12.2 Spark 应用开发建议

RDD 多次使用时，建议将 RDD 持久化

RDD在默认情况下的存储级别是StorageLevel.NONE，即既不存磁盘也不放在内存中，如果某个RDD需要多次使用，可以考虑将该RDD持久化，方法如下：

调用spark.RDD中的cache()、persist()、persist(newLevel:StorageLevel)函数均可将RDD持久化，cache()和persist()都是将RDD的存储级别设置为StorageLevel.MEMORY_ONLY，persist(newLevel:StorageLevel)可以为RDD设置其他存储级别，但是要求调用该方法之前RDD的存储级别为StorageLevel.NONE或者与newLevel相同，也就是说，RDD的存储级别一旦设置为StorageLevel.NONE之外的级别，则无法改变。

如果想要将RDD去持久化，那么可以调用unpersist(blocking:Boolean = true)，该函数功能如下：

1. 将该RDD从持久化列表中移除，RDD对应的数据进入可回收状态；
2. 将RDD的存储级别重新设置为StorageLevel.NONE。

慎重选择 shuffle 过程的算子

该类算子称为宽依赖算子，其特点是父RDD的一个partition影响子RDD的多个partition，RDD中的元素一般都是<key, value>对。执行过程中都会涉及到RDD的partition重排，这个操作称为shuffle。

由于shuffle类算子存在节点之间的网络传输，因此对于数据量很大的RDD，应该尽量提取需要使用的信息，减小其单条数据的大小，然后再调用shuffle类算子。

常用的有如下几种：

- combineByKey() : RDD[(K, V)] => RDD[(K, C)]，是将RDD[(K, V)]中key相同的数据的所有value转化成为一个类型为C的值。
- groupByKey() 和reduceByKey()是combineByKey的两种具体实现，对于数据聚合比较复杂而groupByKey和reduceByKey不能满足使用需求的场景，可以使用自己定义的聚合函数作为combineByKey的参数来实现。
- distinct(): RDD[T] => RDD[T]，作用是去除重复元素的算子。其处理过程代码如下：

  ```
  map(x => (x, null)).reduceByKey((x, y) => x, numPartitions).map(_._1)
  ```

  这个过程比较耗时，尤其是数据量很大时，建议不要直接对大文件生成的RDD使用。
- join() : (RDD[(K, V)], RDD[(K, W)]) => RDD[(K, (V, W))]，作用是将两个RDD通过key做连接。

  如果RDD[(K, V)]中某个key有X个value，而RDD[(K, W)]中相同key有Y个value，那么最终在RDD[(K, (V, W))]中会生成X*Y条记录。

在业务情况允许的情况下使用高性能算子

1. 使用reduceByKey/aggregateByKey替代groupByKey。

   所谓的map-side预聚合，说的是在每个节点本地对相同的key进行一次聚合操作，类似于MapReduce中的本地combiner。 map-side预聚合之后，每个节点本地就只会有一条相同的key，因为多条相同的key都被聚合起来了。其他节点在拉取所有节点上的相同key时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘IO以及网络传输开销。通常来说，在可能的情况下，建议使用reduceByKey或aggregateByKey算子来替代掉groupByKey算子。因为reduceByKey和aggregateByKey算子都会使用用户自定义的函数对每个节点本地的相同key进行预聚合。而groupByKey算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。
2. 使用mapPartitions替代普通map。

   mapPartitions类的算子，一次函数调用会处理一个partition所有的数据，而不是一次函数调用处理一条，性能相对来说会高一些。 但是有的时候，使用mapPartitions会出现OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个partition所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现OOM异常。所以使用这类操作时要慎重！
3. 使用filter之后进行coalesce操作。

   通常对一个RDD执行filter算子过滤掉RDD中较多数据后（比如30%以上的数据），建议使用coalesce算子，手动减少RDD的partition数量，将RDD中的数据压缩到更少的partition中去。因为filter之后，RDD的每个partition中都会有很多数据被过滤掉，此时如果照常进行后续的计算，其实每个task处理的partition中的数据量并不是很多，有一点资源浪费，而且此时处理的task越多，可能速度反而越慢。因此用coalesce减少partition数量，将RDD中的数据压缩到更少的partition之后，只要使用更少的task即 可处理完所有的partition。在某些场景下，对于性能的提升会有一定的帮助。
4. 使用repartitionAndSortWithinPartitions替代repartition与sort类操作。

   repartitionAndSortWithinPartitions是Spark官网推荐的一个算子，官方建议，如果需要在 repartition重分区之后，还要进行排序，建议直接使用

repartitionAndSortWithinPartitions 算子。因为该算子 可以一边进行重分区的shuffle操作，一边进行排序。shuffle与sort两个操作同时进行，比先shuffle再sort来说，性能可能是要高的。

5. 使用foreachPartitions替代foreach。

   原理类似于“使用mapPartitions替代map”，也是一次函数调用处理一个partition的所有数据，而不是一次函数调用处理一条数 据。在实践中发现，foreachPartitions类的算子，对性能的提升还是很有帮助的。比如在foreach函数中，将RDD中所有数据写 MySQL，那么如果是普通的foreach算子，就会一条数据一条数据地写，每次函数调用可能就会创建一个数据库连接，此时就势必会频繁地创建和销毁数据库连接，性能是非常低下；但是如果用foreachPartitions算子一次性处理一个partition的数据，那么对于每个 partition，只要创建一个数据库连接即可，然后执行批量插入操作，此时性能是比较高的。

RDD 共享变量

在应用开发中，一个函数被传递给Spark操作（ 例如map和reduce ），在一个远程集群上运行，它实际上操作的是这个函数用到的所有变量的独立复制。这些变量会被复制到每一台机器。通常看来，在任务之间中，读写共享变量显然不够高效。Spark为两种常见的使用模式，提供了两种有限的共享变量：广播变量、累加器。

在对性能要求比较高的场景下，可以使用 Kryo 优化序列化性能

Spark提供了两种序列化实现：

org.apache.spark.serializer.KryoSerializer：性能好，兼容性差

org.apache.spark.serializer.JavaSerializer：性能一般，兼容性好

使用：conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

说明

为什么不默认使用Kryo序列化?

Spark默认使用的是Java的序列化机制，也就是ObjectOutputStream/ObjectInputStream API来进行序列化和反序列化。但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。官方介 绍，Kryo序列化机制比Java序列化机制，性能高10倍左右。Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求要注册所有需要进行序列化的自定义类型，因此对于开发者来说，这种方式比较麻烦。

Spark Streaming 性能优化建议

1. 设置合理的批处理时间(batchDuration)。
2. 设置合理的数据接收并行度。
   - 设置多个Receiver接收数据。
   - 设置合理的Receiver阻塞时间。
3. 设置合理的数据处理并行度。
4. 使用Kryo系列化。
5. 内存调优。
   - 设置持久化级别减少GC开销。
   - 使用并发的标记-清理GC算法减少GC暂停时间。

运行 pyspark 建议

运行pyspark应用时，不能使用集群自带的python环境，需要用户自行安装python环境，并将python相关依赖包打包上传到HDFS。