腾讯云数据仓库 TCHouse-D

开发指南

产品文档

腾讯云

文档目录

开发指南
数据表设计
数据表和数据模型

最近更新时间：2024-06-27 10:54:20

本文档主要从逻辑层面，描述 Doris 的表和数据模型，以帮助用户更好地应对不同的业务场景。

数据表

在 Doris 中，数据以表（Table）的形式进行逻辑上的描述。表是具有相同模式的同质数据的集合。 一张表包括行（Row）和列（Column）。Row 即用户的一行数据。Column 用于描述一行数据中不同的字段，可以根据实际情况采用不同的数据类型（如整形、字符串、布尔型等）。
从 OLAP 场景看，Column 可以分为两大类：Key 和 Value。Key 表示维度列，value 表示指标列。

数据模型

Doris 的数据模型主要分为3类：Aggregate、Unique 和 Duplicate。

Aggregate 模型

我们以实际的例子来说明什么是聚合模型，以及如何正确的使用聚合模型。

示例1：导入数据聚合

假设业务有如下数据表模式：

ColumnName	Type	AggregationType	Comment
user_id	LARGEINT	-	用户 ID
date	DATE	-	数据导入日期
city	VARCHAR(20)	-	用户所在城市
age	SMALLINT	-	用户年龄
sex	TINYINT	-	用户性别
last_visit_date	DATETIME	REPLACE	用户最后一次访问时间

cost	BIGINT	SUM	用户总消费
max_dwell_time	INT	MAX	用户最大停留时间
min_dwell_time	INT	MIN	用户最小停留时间

如果转换成建表语句则如下（省略建表语句中的 Partition 和 Distribution 信息）：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户id",
    `date` DATE NOT NULL COMMENT "数据导入日期时间",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00" COMMENT "用户最
    `cost` BIGINT SUM DEFAULT "0" COMMENT "用户总消费",
    `max_dwell_time` INT MAX DEFAULT "0" COMMENT "用户最大停留时间",
    `min_dwell_time` INT MIN DEFAULT "99999" COMMENT "用户最小停留时间"
)
AGGREGATE KEY(`user_id`, `date`, `city`, `age`, `sex`)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 1
PROPERTIES (
"replication_allocation" = "tag.location.default: 1"
);
```

可以看到，这是一个典型的用户信息和访问行为的事实表。在一般星型模型中，用户信息和访问行为一般分别存放在维度表和事实表中。这里我们为了更加方便的解释 Doris 的数据模型，将两部分信息统一存放在一张表中。

表中的列按照是否设置了 `AggregationType`，分为 Key (维度列) 和 Value（指标列）。没有设置 `AggregationType` 的，如 `user_id`、`date`、`age`、`sex` 称为 **Key**，而设置了 `AggregationType` 的称为 **Value**。

当我们导入数据时，对于 Key 列相同的行会聚合成一行，而 Value 列会按照设置的 `AggregationType` 进行聚合。`AggregationType` 目前有以下四种聚合方式：

1. SUM：求和，多行的 Value 进行累加。
2. REPLACE：替代，下一批数据中的 Value 会替换之前导入过的行中的 Value。
3. MAX：保留最大值。
4. MIN：保留最小值。

假设我们有以下导入数据（原始数据）：

user_id	date	city	age	sex	last_visit_date	cost	max_dwell_time	min_dwell_time
10000	2017-10-01	北京	20	0	2017-10-01 06:00:00	20	10	10

10000	2017-10-01	北京	20	0	2017-10-01 07:00:00	15	2	2
10001	2017-10-01	北京	30	1	2017-10-01 17:05:45	2	22	22
10002	2017-10-02	上海	20	1	2017-10-02 12:59:12	200	5	5
10003	2017-10-02	广州	32	0	2017-10-02 11:20:00	30	11	11
10004	2017-10-01	深圳	35	0	2017-10-01 10:00:15	100	3	3
10004	2017-10-03	深圳	35	0	2017-10-03 10:20:22	11	6	6

我们假设这是一张记录用户访问某商品页面行为的表。我们以第一行数据为例，解释如下：

数据	说明
10000	用户 ID，每个用户唯一识别 ID
2017-10-01	数据入库时间，精确到日期
北京	用户所在城市
20	用户年龄
0	性别男（1 代表女性）
2017-10-01 06:00:00	用户本次访问该页面的时间，精确到秒
20	用户本次访问产生的消费
10	用户本次访问，驻留该页面的时间
10	用户本次访问，驻留该页面的时间（冗余）

那么当这批数据正确导入到 Doris 中后，Doris 中最终存储如下：

user_id	date	city	age	sex	last_visit_date	cost	max_dwell_time	min_dwell_time
10000	2017-10-01	北京	20	0	2017-10-01 07:00:00	35	10	2
10001	2017-	北	30	1	2017-10-01	2	22	22

	10-01	京			17:05:45			
10002	2017-10-02	上海	20	1	2017-10-02 12:59:12	200	5	5
10003	2017-10-02	广州	32	0	2017-10-02 11:20:00	30	11	11
10004	2017-10-01	深圳	35	0	2017-10-01 10:00:15	100	3	3
10004	2017-10-03	深圳	35	0	2017-10-03 10:20:22	11	6	6

可以看到，用户 10000 只剩下了一行**聚合后**的数据。而其余用户的数据和原始数据保持一致。这里先解释下用户 10000 聚合后的数据：

前5列没有变化，从第6列 `last_visit_date` 开始：

`2017-10-01 07:00:00` ：因为 `last_visit_date` 列的聚合方式为 REPLACE，所以 `2017-10-01 07:00:00` 替换了 `2017-10-01 06:00:00` 保存了下来。

说明

在同一个导入批次中的数据，对于 REPLACE 这种聚合方式，替换顺序不做保证。如在这个例子中，最终保存下来的，也有可能是 `2017-10-01 06:00:00` 。而对于不同导入批次中的数据，可以保证，后一批次的数据会替换前一批次。

`35` ：因为 `cost` 列的聚合类型为 SUM，所以由 20 + 15 累加获得 35。

`10` ：因为 `max_dwell_time` 列的聚合类型为 MAX，所以 10 和 2 取最大值，获得 10。

`2` ：因为 `min_dwell_time` 列的聚合类型为 MIN，所以 10 和 2 取最小值，获得 2。

经过聚合，Doris 中最终只会存储聚合后的数据。换句话说，即明细数据会丢失，用户不能够再查询到聚合前的明细数据了。

示例2：保留明细数据

接示例1，我们将表结构修改如下：

ColumnName	Type	AggregationType	Comment
user_id	LARGEINT	-	用户 ID
date	DATE	-	数据导入日期
timestamp	DATETIME	-	数据导入时间，精确到秒
city	VARCHAR(20)	-	用户所在城市
age	SMALLINT	-	用户年龄

sex	TINYINT	-	用户性别
last_visit_date	DATETIME	REPLACE	用户最后一次访问时间
cost	BIGINT	SUM	用户总消费
max_dwell_time	INT	MAX	用户最大停留时间
min_dwell_time	INT	MIN	用户最小停留时间

即增加了一列 `timestamp`，记录精确到秒的数据导入时间。

同时，将 `AGGREGATE KEY` 设置为 `AGGREGATE KEY(user_id, date, timestamp, city, age, sex)`

导入数据如下：

user_id	date	timestamp	city	age	sex	last_visit_date	cost	max_dwell_time	min_
10000	2017-10-01	2017-10-01 08:00:05	北京	20	0	2017-10-01 06:00:00	20	10	10
10000	2017-10-01	2017-10-01 09:00:05	北京	20	0	2017-10-01 07:00:00	15	2	2
10001	2017-10-01	2017-10-01 18:12:10	北京	30	1	2017-10-01 17:05:45	2	22	22
10002	2017-10-02	2017-10-02 13:10:00	上海	20	1	2017-10-02 12:59:12	200	5	5
10003	2017-10-02	2017-10-02 13:15:00	广州	32	0	2017-10-02 11:20:00	30	11	11
10004	2017-10-01	2017-10-01 12:12:48	深圳	35	0	2017-10-01 10:00:15	100	3	3
10004	2017-10-03	2017-10-03 12:38:20	深圳	35	0	2017-10-03 10:20:22	11	6	6

那么当这批数据正确导入到 Doris 中后，Doris 中最终存储如下：

user_id	date	timestamp	city	age	sex	last_visit_date	cost	max_dwell_time	min_

10000	2017-10-01	2017-10-01 08:00:05	北京	20	0	2017-10-01 06:00:00	20	10	10
10000	2017-10-01	2017-10-01 09:00:05	北京	20	0	2017-10-01 07:00:00	15	2	2
10001	2017-10-01	2017-10-01 18:12:10	北京	30	1	2017-10-01 17:05:45	2	22	22
10002	2017-10-02	2017-10-02 13:10:00	上海	20	1	2017-10-02 12:59:12	200	5	5
10003	2017-10-02	2017-10-02 13:15:00	广州	32	0	2017-10-02 11:20:00	30	11	11
10004	2017-10-01	2017-10-01 12:12:48	深圳	35	0	2017-10-01 10:00:15	100	3	3
10004	2017-10-03	2017-10-03 12:38:20	深圳	35	0	2017-10-03 10:20:22	11	6	6

我们可以看到，存储的数据，和导入数据完全一样，没有发生任何聚合。这是因为，这批数据中，因为加入了 `timestamp` 列，所有行的 Key 都**不完全相同**。也就是说，只要保证导入的数据中，每一行的 Key 都不完全相同，那么即使在聚合模型下，Doris 也可以保存完整的明细数据。

示例3：导入数据与已有数据聚合

接示例1。假设现在表中已有数据如下：

user_id	date	city	age	sex	last_visit_date	cost	max_dwell_time	min_dwell_time
10000	2017-10-01	北京	20	0	2017-10-01 07:00:00	35	10	2
10001	2017-10-01	北京	30	1	2017-10-01 17:05:45	2	22	22
10002	2017-10-02	上海	20	1	2017-10-02 12:59:12	200	5	5

10003	2017-10-02	广州	32	0	2017-10-02 11:20:00	30	11	11
10004	2017-10-01	深圳	35	0	2017-10-01 10:00:15	100	3	3
10004	2017-10-03	深圳	35	0	2017-10-03 10:20:22	11	6	6

我们再导入一批新的数据：

user_id	date	city	age	sex	last_visit_date	cost	max_dwell_time	min_dwell_time
10004	2017-10-03	深圳	35	0	2017-10-03 11:22:00	44	19	19
10005	2017-10-03	长沙	29	1	2017-10-03 18:11:02	3	1	1

那么当这批数据正确导入到 Doris 中后，Doris 中最终存储如下：

user_id	date	city	age	sex	last_visit_date	cost	max_dwell_time	min_dwell_time
10000	2017-10-01	北京	20	0	2017-10-01 07:00:00	35	10	2
10001	2017-10-01	北京	30	1	2017-10-01 17:05:45	2	22	22
10002	2017-10-02	上海	20	1	2017-10-02 12:59:12	200	5	5
10003	2017-10-02	广州	32	0	2017-10-02 11:20:00	30	11	11
10004	2017-10-01	深圳	35	0	2017-10-01 10:00:15	100	3	3
10004	2017-10-03	深圳	35	0	2017-10-03 11:22:00	55	19	6
10005	2017-10-03	长沙	29	1	2017-10-03 18:11:02	3	1	1

可以看到，用户 10004 的已有数据和新导入的数据发生了聚合。同时新增了10005用户的数据。
数据的聚合，在 Doris 中有如下三个阶段发生：

1. 每一批次数据导入的 ETL 阶段。该阶段会在每一批次导入的数据内部进行聚合。
2. 底层 BE 进行数据 Compaction 的阶段。该阶段，BE 会对已导入的不同批次的数据进行进一步的聚合。
3. 数据查询阶段。在数据查询时，对于查询涉及到的数据，会进行对应的聚合。

数据在不同时间，可能聚合的程度不一致。例如一批数据刚导入时，可能还未与之前已存在的数据进行聚合。但是对于用户而言，用户**只能查询到**聚合后的数据。即不同的聚合程度对于用户查询而言是透明的。用户需始终认为数据以**最终的完成的聚合程度**存在，而**不应假设某些聚合还未发生**。

Unique 模型

在某些多维分析场景下，用户更关注的是如何保证 Key 的唯一性，即如何获得 Primary Key 唯一性约束。因此，我们引入了 Unique 的数据模型。该模型本质上是聚合模型的一个特例，也是一种简化的表结构表示方式。举例说明：

ColumnName	Type	IsKey	Comment
user_id	BIGINT	Yes	用户 ID
username	VARCHAR(50)	Yes	用户昵称
city	VARCHAR(20)	No	用户所在城市
age	SMALLINT	No	用户年龄
sex	TINYINT	No	用户性别
phone	LARGEINT	No	用户电话
address	VARCHAR(500)	No	用户住址
register_time	DATETIME	No	用户注册时间

这是一个典型的用户基础信息表。这类数据没有聚合需求，只需保证主键唯一性。（这里的主键为 user_id + username）。那么我们的建表语句如下：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户id",
    `username` VARCHAR(50) NOT NULL COMMENT "用户昵称",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `phone` LARGEINT COMMENT "用户电话",
    `address` VARCHAR(500) COMMENT "用户地址",
    `register_time` DATETIME COMMENT "用户注册时间"
)
UNIQUE KEY(`user_id`, `username`)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 1
```

```
PROPERTIES (
"replication_allocation" = "tag.location.default: 1"
);
```

而这个表结构，完全同等于以下使用聚合模型描述的表结构：

ColumnName	Type	AggregationType	Comment
user_id	BIGINT	-	用户 ID
username	VARCHAR(50)	-	用户昵称
city	VARCHAR(20)	REPLACE	用户所在城市
age	SMALLINT	REPLACE	用户年龄
sex	TINYINT	REPLACE	用户性别
phone	LARGEINT	REPLACE	用户电话
address	VARCHAR(500)	REPLACE	用户住址
register_time	DATETIME	REPLACE	用户注册时间

及建表语句：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户id",
    `username` VARCHAR(50) NOT NULL COMMENT "用户昵称",
    `city` VARCHAR(20) REPLACE COMMENT "用户所在城市",
    `age` SMALLINT REPLACE COMMENT "用户年龄",
    `sex` TINYINT REPLACE COMMENT "用户性别",
    `phone` LARGEINT REPLACE COMMENT "用户电话",
    `address` VARCHAR(500) REPLACE COMMENT "用户地址",
    `register_time` DATETIME REPLACE COMMENT "用户注册时间"
)
AGGREGATE KEY(`user_id`, `username`)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 1
PROPERTIES (
"replication_allocation" = "tag.location.default: 1"
);
```

即 Unique 模型完全可以用聚合模型中的 REPLACE 方式替代。其内部的实现方式和数据存储方式也完全一样。这里不再继续举例说明。

Duplicate 模型

在某些多维分析场景下，数据既没有主键，也没有聚合需求。因此，我们引入 Duplicate 数据模型来满足这类需求。举例说明。

ColumnName	Type	SortKey	Comment
timestamp	DATETIME	Yes	日志时间
type	INT	Yes	日志类型
error_code	INT	Yes	错误码
error_msg	VARCHAR(1024)	No	错误详细信息
op_id	BIGINT	No	负责人 ID
op_time	DATETIME	No	处理时间

建表语句如下：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `timestamp` DATETIME NOT NULL COMMENT "日志时间",
    `type` INT NOT NULL COMMENT "日志类型",
    `error_code` INT COMMENT "错误码",
    `error_msg` VARCHAR(1024) COMMENT "错误详细信息",
    `op_id` BIGINT COMMENT "负责人id",
    `op_time` DATETIME COMMENT "处理时间"
)
DUPLICATE KEY(`timestamp`, `type`)
DISTRIBUTED BY HASH(`type`) BUCKETS 1
PROPERTIES (
"replication_allocation" = "tag.location.default: 1"
);
```

这种数据模型区别于 Aggregate 和 Unique 模型。数据完全按照导入文件中的数据进行存储，不会有任何聚合，即使两行数据完全相同也都会保留。 而在建表语句中指定的 DUPLICATE KEY，只是用来指明底层数据按照那些列进行 `排序` 。在 DUPLICATE KEY 的选择上，我们建议适当的选择前 2-4 列就可以。这种数据模型适用于既没有聚合需求，又没有主键唯一性约束的原始数据的存储。

聚合模型的局限性

Aggregate 模型 & Unique 模型

这里我们针对 Aggregate 模型（包括 Unique 模型），来介绍下聚合模型的局限性。

在聚合模型中，模型对外展现的，是**最终聚合后的**数据。也就是说，对于任何还未聚合的数据（例如说两个不同导入批次的数据），必须通过某种方式保证对外展示的一致性。

假设表结构如下：

ColumnName	Type	AggregationType	Comment
user_id	LARGEINT	-	用户 ID
date	DATE	-	数据导入日期
cost	BIGINT	SUM	用户总消费

假设存储引擎中有如下两个已经导入完成的批次的数据：

batch 1

user_id	date	cost
10001	2017-11-20	50
10002	2017-11-21	39

batch 2

user_id	date	cost
10001	2017-11-20	1
10001	2017-11-21	5
10003	2017-11-22	22

可以看到，用户 10001 分属在两个导入批次中的数据还没有聚合。但是为了保证用户只能查询到如下最终聚合后的数据：

user_id	date	cost
10001	2017-11-20	51
10001	2017-11-21	5
10002	2017-11-21	39
10003	2017-11-22	22

我们在查询引擎中加入了聚合算子，来保证数据对外的一致性。

另外，在聚合列（Value）上，执行与聚合类型不一致的聚合类查询时，要注意语意。例如我们在如上示例中执行如下查询：

```
SELECT MIN(cost) FROM table;
```

得到的结果是 5，而不是 1。

同时，这种一致性保证，在某些查询中，会极大的降低查询效率。

我们以最基本的 count(*) 查询为例：

```
SELECT COUNT(*) FROM table;
```

在其他数据库中，这类查询都会很快的返回结果。因为在实现上，我们可以通过如**导入时对行进行计数，保存 count 的统计信息**，或者在查询时**仅扫描某一列数据，获得 count 值**的方式，只需很小的开销，即可获得查询结果。但是在 Doris 的聚合模型中，这种查询的开销**非常大**。

我们以刚才的数据为例：

batch 1

user_id	date	cost
10001	2017-11-20	50
10002	2017-11-21	39

batch 2

user_id	date	cost
10001	2017-11-20	1
10001	2017-11-21	5
10003	2017-11-22	22

因为最终的聚合结果为：

user_id	date	cost
10001	2017-11-20	51
10001	2017-11-21	5
10002	2017-11-21	39
10003	2017-11-22	22

所以，`select count(*) from table;` 的正确结果应该为 **4**。但如果我们只扫描 `user_id` 这一列，如果加上查询时聚合，最终得到的结果是 **3**（10001, 10002, 10003）。而如果不加查询时聚合，则得到的结果是 **5**（两批次一共5行数据）。可见这两个结果都是不对的。

为了得到正确的结果，我们必须同时读取 `user_id` 和 `date` 这两列的数据，**再加上查询时聚合**，才能返回 **4** 这个正确的结果。也就是说，在 count( ) *查询中，Doris 必须扫描所有的 AGGREGATE KEY 列（这里就是* `user_id` 和 `date` ），并且聚合后，才能得到语意正确的结果。当聚合列非常多时，count( ) 查询需要扫描大量的数据。

因此，当业务上有频繁的 count( ) *查询时，我们建议用户通过增加一个**值恒为 1 的，聚合类型为 SUM 的列来模拟 count( )*。如刚才的例子中的表结构，我们修改如下：

ColumnName	Type	AggregateType	Comment
user_id	BIGINT	-	用户 ID
date	DATE	-	数据导入日期
cost	BIGINT	SUM	用户总消费
count	BIGINT	SUM	用于计算 count

增加一个 count 列，并且导入数据中，该列值**恒为 1**。则 `select count(*) from table;` 的结果等价于 `select sum(count) from table;` 。而后者的查询效率将远高于前者。不过这种方式也有使用限制，就是用户需要自行保证，不会重复导入 AGGREGATE KEY 列都相同的行。否则，`select sum(count) from table;` 只能表述原始导入的行数，而不是 `select count(*) from table;` 的语义，前者值会错误的增大。

另一种方式，就是 **将如上的 `count` 列的聚合类型改为 REPLACE，且依然值恒为 1**。那么 `select sum(count) from table;` 和 `select count(*) from table;` 的结果将是一致的。并且这种方式，没有导入重复行的限制。

Duplicate 模型

Duplicate 模型没有聚合模型的这个局限性。因为该模型不涉及聚合语意，在做 count(*) 查询时，任意选择一列查询，即可得到语意正确的结果。

最佳实践

因为数据模型在建表时就已经确定，且**无法修改**。所以，选择一个合适的数据模型**非常重要**。

数据模型选择

Doris 数据模型目前分为三类:：AGGREGATE KEY，UNIQUE KEY，DUPLICATE KEY。三种模型中数据都是按KEY进行排序。

Aggregate 模型

Aggregate 模型可以通过预聚合，极大地降低聚合查询时所需扫描的数据量和查询的计算量，非常适合有固定模式的报表类查询场景。但是该模型对 count(*) 查询很不友好。同时因为固定了 Value 列上的聚合方式，在进行其他类型的聚合查询时，需要考虑语意正确性。
Aggregate Key 相同时，新旧记录进行聚合，目前支持的聚合函数有SUM，MIN，MAX，REPLACE。

```
CREATE TABLE site_visit
(
    siteid      INT,
    city        SMALLINT,
    username    VARCHAR(32),
    pv BIGINT   SUM DEFAULT '0'
)
AGGREGATE KEY(siteid, city, username)
DISTRIBUTED BY HASH(siteid) BUCKETS 10;
```

Unique 模型

Unique 模型针对需要唯一主键约束的场景，Unique key 相同时，新记录覆盖旧记录，可以保证主键唯一性约束。适用于有更新需求的分析业务。目前 Unique key 实现上和 Aggregate key 的 REPLACE 聚合方法一样，二者本质上相同。但是无法利用 ROLLUP 等预聚合带来的查询优势（因为本质是 REPLACE，没有 SUM 这种聚合方式）。
注意
Unique 模型仅支持整行更新，如果用户既需要唯一主键约束，又需要仅更新部分列（例如将多张源表的列合并后导入到一张 doris 表的情形），则可以考虑使用 Aggregate 模型，同时将非主键列的聚合类型设置为 REPLACE_IF_NOT_NULL。具体的用法可以参考 数据模型。

```
CREATE TABLE sales_order
(
    orderid     BIGINT,
    status      TINYINT,
    username    VARCHAR(32),
    amount      BIGINT DEFAULT '0'
)
UNIQUE KEY(orderid)
DISTRIBUTED BY HASH(orderid) BUCKETS 10;
```

Duplicate 模型

Duplicate 模型相同的行不会合并，适合任意维度的 Ad-hoc 查询。虽然无法利用预聚合的特性，但是不受聚合模型的约束，可以发挥列存模型的优势（列裁剪、向量执行等）。

```
CREATE TABLE session_data
(
    visitorid   SMALLINT,
    sessionid   BIGINT,
    visittime   DATETIME,
    city        CHAR(20),
    province    CHAR(20),
    ip          varchar(32),
    brower      CHAR(20),
    url         VARCHAR(1024)
)
DUPLICATE KEY(visitorid, sessionid)
DISTRIBUTED BY HASH(sessionid, visitorid) BUCKETS 10;
```

大宽表与 Star Schema（星形模型）

业务方建表时，为了和前端业务适配，通常不对维度信息和指标信息加以区分，而将 Schema 定义成大宽表。对于 Doris 而言， 这类大宽表的性能不尽如人意：

Schema 中字段数比较多，聚合模型中可能 key 列比较多，导入过程中需要排序的列会增加。

维度信息更新会反应到整张表中，而更新的频率直接影响查询的效率。

使用过程中，建议用户尽量使用 Star Schema 区分维度表和指标表。频繁更新的维度表也可以放在 MySQL 外部表中。而如果只有少量更新， 可以直接放在 Doris 中。在 Doris 中存储维度表时，可对维度表设置更多的副本，提升 Join 的性能。

数据分区和分桶

最近更新时间：2024-06-27 10:54:36

为了能高效处理大数据量的存储和计算，Doris 按分治思想对数据进行分割处理。
Doris 支持两层的数据划分。第一层是 Partition（分区），支持 Range （按范围）和 List（按枚举值） 的划分方式。
第二层是 Bucket（分桶），仅支持 Hash 的划分方式。分区和分桶都是对数据进行横向分割。
也可以仅使用一层分区。使用一层分区时，只支持 Bucket 划分。下面我们来分别介绍下分区以及分桶。

分区（Partition）

分区用于将数据划分成不同区间, 逻辑上可以理解为将原始表划分成了多个子表。可以方便的按分区对数据进行管理，例如，删除数据时更加迅速。
Partition 列可以指定一列或多列，分区列必须为 KEY 列。多列分区的使用方式在后面 **多列分区** 小结介绍。
不论分区列是什么类型，在写分区值时，都需要加双引号。
分区数量理论上没有上限。
当不使用 Partition 建表时，系统会自动生成一个和表名同名的，全值范围的 Partition。该 Partition 对用户不可见，并且不可删改。
创建分区时**不可添加范围重叠**的分区。

Range 分区

Partition 支持通过 `VALUES LESS THAN (...)` 仅指定上界，系统会将前一个分区的上界作为该分区的下界，生成一个左闭右开的区间。同时，也支持通过 `VALUES [...)` 指定上下界，生成一个左闭右开的区间。
通过 `VALUES [...)` 同时指定上下界比较容易理解。这里举例说明，当使用 `VALUES LESS THAN (...)` 语句进行分区的增删操作时，分区范围的变化情况。

```
-- Range Partition

CREATE TABLE IF NOT EXISTS example_db.expamle_range_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户id",
    `date` DATE NOT NULL COMMENT "数据灌入日期时间",
    `timestamp` DATETIME NOT NULL COMMENT "数据灌入的时间戳",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00" COMMENT "用
    `cost` BIGINT SUM DEFAULT "0" COMMENT "用户总消费",
    `max_dwell_time` INT MAX DEFAULT "0" COMMENT "用户最大停留时间",
    `min_dwell_time` INT MIN DEFAULT "99999" COMMENT "用户最小停留时间"
```

```
    )
    ENGINE=OLAP
    AGGREGATE KEY(`user_id`, `date`, `timestamp`, `city`, `age`, `sex`)
    PARTITION BY RANGE(`date`)
    (
        PARTITION `p201701` VALUES LESS THAN ("2017-02-01"),
        PARTITION `p201702` VALUES LESS THAN ("2017-03-01"),
        PARTITION `p201703` VALUES LESS THAN ("2017-04-01")
    )
    DISTRIBUTED BY HASH(`user_id`) BUCKETS 16
    PROPERTIES
    (
        "replication_num" = "3"
    );
```

如上 `expamle_range_tbl` 示例，当建表完成后，查看分区会发现自动生成了如下3个分区：

```
show partitions from expamle_range_tbl;
```

```
p201701: [MIN_VALUE,  2017-02-01)
p201702: [2017-02-01, 2017-03-01)
p201703: [2017-03-01, 2017-04-01)
```

当我们使用 sql 增加一个分区 p201705 VALUES LESS THAN ("2017-06-01")：

```
alter table expamle_range_tbl add partition p201705 VALUES LESS THAN ("2017-06-01")
show partitions from expamle_range_tbl;
```

分区结果如下：

```
p201701: [MIN_VALUE,  2017-02-01)
p201702: [2017-02-01, 2017-03-01)
p201703: [2017-03-01, 2017-04-01)
p201705: [2017-04-01, 2017-06-01)
```

此时我们删除分区 p201703：

```
alter table expamle_range_tbl drop partition p201703;
show partitions from expamle_range_tbl;
```

则分区结果如下：

```
p201701: [MIN_VALUE,  2017-02-01)
p201702: [2017-02-01, 2017-03-01)
p201705: [2017-04-01, 2017-06-01)
```

注意

注意到 p201702 和 p201705 的分区范围并没有发生变化，而这两个分区之间，出现了一个空洞：[2017-03-01, 2017-04-01)。即如果导入的数据范围在这个空洞范围内，是无法导入的。

继续删除分区 p201702：

```
alter table expamle_range_tbl drop partition p201702;
show partitions from expamle_range_tbl;
```

分区结果如下：

```
p201701: [MIN_VALUE,  2017-02-01)
p201705: [2017-04-01, 2017-06-01)
```

说明

空洞范围变为：[2017-02-01, 2017-04-01)。

现在增加一个分区 p201702new VALUES LESS THAN ("2017-03-01")：

```
alter table expamle_range_tbl add partition p201702new VALUES LESS THAN ("2017-03-0
show partitions from expamle_range_tbl;
```

分区结果如下：

```
p201701:    [MIN_VALUE,  2017-02-01)
p201702new: [2017-02-01, 2017-03-01)
p201705:    [2017-04-01, 2017-06-01)
```

说明

可以看到空洞范围缩小为：[2017-03-01, 2017-04-01)。

现在删除分区 p201701，并添加分区 p201612 VALUES LESS THAN ("2017-01-01")：

```
alter table expamle_range_tbl drop partition p201701;
alter table expamle_range_tbl add partition p201612 VALUES LESS THAN ("2017-01-01")
show partitions from expamle_range_tbl;
```

分区结果如下：

```
p201612:    [MIN_VALUE,  2017-01-01)
p201702new: [2017-02-01, 2017-03-01)
p201705:    [2017-04-01, 2017-06-01)
```

说明

即出现了一个新的空洞：[2017-01-01, 2017-02-01)。

综上，分区的删除不会改变已存在分区的范围。删除分区可能出现空洞。通过 `VALUES LESS THAN` 语句增加分区时，分区的下界紧接上一个分区的上界。

多列分区

Range 分区除了上述我们看到的单列分区，也支持**多列分区**，示例如下：

```
-- Range Partition

CREATE TABLE IF NOT EXISTS example_db.expamle_range_multi_partiton_key_tbl
(
`user_id` LARGEINT NOT NULL COMMENT "用户id",
`date` DATE NOT NULL COMMENT "数据灌入日期时间",
`timestamp` DATETIME NOT NULL COMMENT "数据灌入的时间戳",
`city` VARCHAR(20) COMMENT "用户所在城市",
`age` SMALLINT COMMENT "用户年龄",
`sex` TINYINT COMMENT "用户性别",
`last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00" COMMENT "用户最后一次
`cost` BIGINT SUM DEFAULT "0" COMMENT "用户总消费",
`max_dwell_time` INT MAX DEFAULT "0" COMMENT "用户最大停留时间",
`min_dwell_time` INT MIN DEFAULT "99999" COMMENT "用户最小停留时间"
)
ENGINE=OLAP
AGGREGATE KEY(`user_id`, `date`, `timestamp`, `city`, `age`, `sex`)
PARTITION BY RANGE(`date`, `user_id`)
(
PARTITION `p201701_1000` VALUES LESS THAN ("2017-02-01", "1000"),
PARTITION `p201702_2000` VALUES LESS THAN ("2017-03-01", "2000"),
PARTITION `p201703_all`  VALUES LESS THAN ("2017-04-01")
)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 16
PROPERTIES
(
"replication_num" = "3"
);
```

在以上示例中，我们指定 `date` （DATE 类型）和 `user_id` （INT 类型）作为分区列。以上示例最终得到的分区如下：

```
show partitions from expamle_range_multi_partiton_key_tbl;
```

```
* p201701_1000:    [(MIN_VALUE,  MIN_VALUE), ("2017-02-01", "1000")   )
* p201702_2000:    [("2017-02-01", "1000"),  ("2017-03-01", "2000")   )
* p201703_all:     [("2017-03-01", "2000"),  ("2017-04-01", MIN_VALUE))
```

注意

最后一个分区用户缺省只指定了 `date` 列的分区值，所以 `user_id` 列的分区值会默认填充 `MIN_VALUE` 。

当用户插入数据时，分区列值会按照顺序依次比较，最终得到对应的分区。

举例如下：

```
* 数据  -->  分区
```

```
* 2017-01-01, 200     --> p201701_1000
* 2017-01-01, 2000    --> p201701_1000
* 2017-02-01, 100     --> p201701_1000
* 2017-02-01, 2000    --> p201702_2000
* 2017-02-15, 5000    --> p201702_2000
* 2017-03-01, 2000    --> p201703_all
* 2017-03-10, 1       --> p201703_all
* 2017-04-01, 1000    --> 无法导入
* 2017-05-01, 1000    --> 无法导入
```

验证方法：

插入一条数据并检查存入到哪个分区。分区字段 VisibleVersionTime、VisibleVersion 刚刚有更新的分区即为刚插入数据所在的分区。

```
insert into expamle_range_multi_partiton_key_tbl values (200, '2017-01-01', '2017-0
insert into expamle_range_multi_partiton_key_tbl values (2000, '2017-01-01', '2017-
insert into expamle_range_multi_partiton_key_tbl values (200, '2017-02-01', '2017-0
show partitions from expamle_range_multi_partiton_key_tbl\\G
```

List 分区

分区列支持 `BOOLEAN, TINYINT, SMALLINT, INT, BIGINT, LARGEINT, DATE, DATETIME, CHAR, VARCHAR` 数据类型，分区值为枚举值。只有当数据为目标分区枚举值其中之一时，才可以命中分区。Partition 支持通过 `VALUES IN (...)` 来指定每个分区包含的枚举值。

下面通过示例说明，进行分区的增删操作时，分区的变化。

```
-- List Partition

CREATE TABLE IF NOT EXISTS example_db.expamle_list_tbl
(
`user_id` LARGEINT NOT NULL COMMENT "用户id",
`date` DATE NOT NULL COMMENT "数据灌入日期时间",
`timestamp` DATETIME NOT NULL COMMENT "数据灌入的时间戳",
`city` VARCHAR(20) NOT NULL COMMENT "用户所在城市",
`age` SMALLINT COMMENT "用户年龄",
`sex` TINYINT COMMENT "用户性别",
`last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00" COMMENT "用户最后一次
`cost` BIGINT SUM DEFAULT "0" COMMENT "用户总消费",
`max_dwell_time` INT MAX DEFAULT "0" COMMENT "用户最大停留时间",
`min_dwell_time` INT MIN DEFAULT "99999" COMMENT "用户最小停留时间"
)
ENGINE=olap
AGGREGATE KEY(`user_id`, `date`, `timestamp`, `city`, `age`, `sex`)
PARTITION BY LIST(`city`)
(
```

```
PARTITION `p_cn` VALUES IN ("Beijing", "Shanghai", "Hong Kong"),
PARTITION `p_usa` VALUES IN ("New York", "San Francisco"),
PARTITION `p_jp` VALUES IN ("Tokyo")
)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 16
PROPERTIES
(
"replication_num" = "3"
);
```

如上 `example_list_tbl` 示例，当建表完成后，会自动生成如下3个分区：

```
show partitions from expamle_list_tbl;
```

```
p_cn: ("Beijing", "Shanghai", "Hong Kong")
p_usa: ("New York", "San Francisco")
p_jp: ("Tokyo")
```

当我们增加一个分区 p_uk VALUES IN ("London")

```
alter table expamle_list_tbl add partition p_uk VALUES IN ("London");
show partitions from expamle_list_tbl;
```

分区结果如下：

```
p_cn: ("Beijing", "Shanghai", "Hong Kong")
p_usa: ("New York", "San Francisco")
p_jp: ("Tokyo")
p_uk: ("London")
```

当我们删除分区 p_jp，分区结果如下：

```
alter table expamle_list_tbl drop partition p_jp;
show partitions from expamle_list_tbl;
```

```
p_cn: ("Beijing", "Shanghai", "Hong Kong")
p_usa: ("New York", "San Francisco")
p_uk: ("London")
```

多列分区

List 分区也支持**多列分区**，示例如下：

```
-- List Partition

CREATE TABLE IF NOT EXISTS example_db.expamle_list_multi_partiton_key_tbl
```

```
(
`user_id` LARGEINT NOT NULL COMMENT "用户id",
`date` DATE NOT NULL COMMENT "数据灌入日期时间",
`timestamp` DATETIME NOT NULL COMMENT "数据灌入的时间戳",
`city` VARCHAR(20) NOT NULL COMMENT "用户所在城市",
`age` SMALLINT COMMENT "用户年龄",
`sex` TINYINT COMMENT "用户性别",
`last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00" COMMENT "用户最后一次
`cost` BIGINT SUM DEFAULT "0" COMMENT "用户总消费",
`max_dwell_time` INT MAX DEFAULT "0" COMMENT "用户最大停留时间",
`min_dwell_time` INT MIN DEFAULT "99999" COMMENT "用户最小停留时间"
)
ENGINE=olap
AGGREGATE KEY(`user_id`, `date`, `timestamp`, `city`, `age`, `sex`)
PARTITION BY LIST(`user_id`, `city`)
(
PARTITION `p1_city` VALUES IN (("1", "Beijing"), ("1", "Shanghai")),
PARTITION `p2_city` VALUES IN (("2", "Beijing"), ("2", "Shanghai")),
PARTITION `p3_city` VALUES IN (("3", "Beijing"), ("3", "Shanghai"))
)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 16
PROPERTIES
(
"replication_num" = "3"
);
```

在以上示例中，我们指定 `user_id` (INT 类型) 和 `city` (VARCHAR 类型) 作为分区列。以上示例最终得到的分区如下：

```
show partitions from expamle_list_multi_partiton_key_tbl;
```

```
* p1_city: [("1", "Beijing"), ("1", "Shanghai")]
* p2_city: [("2", "Beijing"), ("2", "Shanghai")]
* p3_city: [("3", "Beijing"), ("3", "Shanghai")]
```

当用户插入数据时，分区列值会按照顺序依次比较，最终得到对应的分区。举例如下：

```
* 数据  --->  分区
* 1, Beijing     ---> p1_city
* 1, Shanghai    ---> p1_city
* 2, Shanghai    ---> p2_city
* 3, Beijing     ---> p3_city
* 1, Tianjin     ---> 无法导入
* 4, Beijing     ---> 无法导入
```

验证方法：
插入一条数据并检查存入到哪个分区。分区字段 VisibleVersionTime、VisibleVersion 刚刚有更新的分区即为刚插入数据所在的分区。

```
insert into expamle_list_multi_partiton_key_tbl values (1, '2017-01-01', '2017-01-0
show partitions from expamle_list_multi_partiton_key_tbl\\G
```

分桶（Bucket）

根据分桶列的 hash 值将数据划分成不同的 Bucket。
如果使用了 Partition，则 `DISTRIBUTED ...` 语句描述的是数据在**各个分区内**的划分规则。如果不使用 Partition，则描述的是对整个表的数据的划分规则。
分桶列可以选择多列，但必须为 Key 列。分桶列可以和 Partition 列相同或不同。
分桶列的选择，是在 **查询吞吐** 和 **查询并发** 之间的一种权衡：
1.1 如果选择多个分桶列，则数据分布更均匀。如果一个查询条件不包含所有分桶列的等值条件，那么该查询会触发所有分桶同时扫描，这样查询的吞吐会增加，单个查询的延迟随之降低。这个方式适合大吞吐低并发的查询场景。
1.2 如果仅选择一个或少数分桶列，则对应的点查询可以仅触发一个分桶扫描。此时，当多个点查询并发时，这些查询有较大的概率分别触发不同的分桶扫描，各个查询之间的 IO 影响较小（尤其当不同桶分布在不同磁盘上时），所以这种方式适合高并发的点查询场景。
分桶的数量理论上没有上限。

最佳实践

关于分桶列的选择

建议采用区分度大的列做分桶, 避免出现数据倾斜。
为方便数据恢复, 建议单个 Bucket 的 size 不要太大, 保持在10GB以内, 所以建表或增加 Partition 时请合理考虑 Bucket 数目, 其中不同 Partition 可指定不同的 Bucket 数。

关于 Partition 和 Bucket 的数量和数据量的建议

一个表的 Tablet 总数量等于（Partition num * Bucket num）。
一个表的 Tablet 数量，在不考虑扩容的情况下，推荐略多于整个集群的磁盘数量。
单个 Tablet 的数据量理论上没有上下界，但建议在 1G - 10G 的范围内。如果单个 Tablet 数据量过小，则数据的聚合效果不佳，且元数据管理压力大。如果数据量过大，则不利于副本的迁移、补齐，且会增加 Schema Change 或者 Rollup 操作失败重试的代价（这些操作失败重试的粒度是 Tablet）。
当 Tablet 的数据量原则和数量原则冲突时，建议优先考虑数据量原则。

在建表时，每个分区的 Bucket 数量统一指定。但是在动态增加分区时（`ADD PARTITION`），可以单独指定新分区的 Bucket 数量。可以利用这个功能方便的应对数据缩小或膨胀。
一个 Partition 的 Bucket 数量一旦指定，不可更改。所以在确定 Bucket 数量时，需要预先考虑集群扩容的情况。例如当前只有 3 台 host，每台 host 有 1 块盘。如果 Bucket 的数量只设置为 3 或更小，那么后期即使再增加机器，也不能提高并发度。
举一些例子：假设在有10台 BE，每台BE一块磁盘的情况下。如果一个表总大小为 500MB，则可以考虑4-8个分片。5GB：8-16个分片。50GB：32个分片。500GB：建议分区，每个分区大小在 50GB 左右，每个分区16-32个分片。5TB：建议分区，每个分区大小在 50GB 左右，每个分区16-32个分片。
说明
表的数据量可以通过 `SHOW DATA` 命令查看，结果除以副本数，即表的数据量。

复合分区与单分区

复合分区

第一级称为 Partition，即分区。用户可以指定某一维度列作为分区列（当前只支持整型和时间类型的列），并指定每个分区的取值范围。
第二级称为 Distribution，即分桶。用户可以指定一个或多个维度列以及桶数对数据进行 HASH 分布。
以下场景推荐使用复合分区:
有时间维度或类似带有有序值的维度，可以以这类维度列作为分区列。分区粒度可以根据导入频次、分区数据量等进行评估。
历史数据删除需求：如有删除历史数据的需求（例如仅保留最近 N 天的数据）。使用复合分区，可以通过删除历史分区来达到目的。也可以通过在指定分区内发送 DELETE 语句进行数据删除。
解决数据倾斜问题：每个分区可以单独指定分桶数量。如按天分区，当每天的数据量差异很大时，可以通过指定分区的分桶数，合理划分不同分区的数据,分桶列建议选择区分度大的列。

单分区

用户也可以不使用复合分区（不选择分区列），即使用单分区。则数据只做 HASH 分布。

常见问题

`Failed to create partition [xxx] . Timeout`

Doris 建表是按照 Partition 粒度依次创建的。当一个 Partition 创建失败时，可能会报这个错误。即使不使用 Partition，当建表出现问题时，也会报 `Failed to create partition`，因为如前文所述，Doris 会为没有指定 Partition 的表创建一个不可更改的默认的 Partition。
当遇到这个错误时，通常是 BE 在创建数据分片时遇到了问题。可以参照以下步骤排查：
1. 在 fe.log 中，查找对应时间点的 `Failed to create partition` 日志。在该日志中，会出现一系列类似 `{10001-10010}` 字样的数字对。数字对的第一个数字表示 Backend ID，第二个数字表示 Tablet ID。如上这个数

字对，表示 ID 为 10001 的 Backend 上，创建 ID 为 10010 的 Tablet 失败了。
2. 前往对应 Backend 的 be.INFO 日志，查找对应时间段内，tablet id 相关的日志，可以找到错误信息。
3. 以下罗列一些常见的 tablet 创建失败错误，包括但不限于：
4. BE 没有收到相关 task，此时无法在 be.INFO 中找到 tablet id 相关日志或者 BE 创建成功，但汇报失败。以上问题，请检查 FE 和 BE 的连通性。
5. 预分配内存失败。可能是表中一行的字节长度超过了 100KB。
6. `Too many open files` 。打开的文件句柄数超过了 Linux 系统限制。需修改 Linux 系统的句柄数限制。
如果创建数据分片时超时，也可以通过在 fe.conf 中设置 `tablet_create_timeout_second=xxx` 以及 `max_create_table_timeout_second=xxx` 来延长超时时间。其中 `tablet_create_timeout_second` 默认是1秒， `max_create_table_timeout_second` 默认是60秒，总体的超时时间为 min(tablet_create_timeout_second * replication_num, max_create_table_timeout_second)，具体参数设置参见 FE 配置项 。

分区数量和分桶数量是否有上限？

分区数量和分表数量没有上限，但过多的分区和分桶数量可能会对doris性能造成影响。
为避免一次创建过多分区，FE上 `max_dynamic_partition_num` 和 `max_multi_partition_num` 参数，分别对**自动分区创建分区**和**批量创建分区**的数量进行了限制。

分区的分桶数量和数据副本数量如何设置？

手工创建分区时，可以指定创建分区的副本数量和分桶数量。如果没有指定，则使用表的 `replication_allocation` 和 `buckets` 参数。
自动创建分区时，创建的分区的副本数量和分桶数量使用表的 `dynamic_partition.replication_allocation` 和 `dynamic_partition.buckets` 参数。

分桶数量是否能修改？

已创建的分区的分桶数量不可修改。
对于使用Range分区的表，可以修改表的bucket数量。对于修改之后手工创建的表的分区，将使用修改后的bucket分桶数量，对于已创建的分区不生效。如果开启了自动分区，这个修改不会影响自动分区新创建的分区的bucket分桶数量，如需要修改自动分区分桶数量，需要修改 `dynamic_partition.buckets` ，修改后新自动分区创建的bucket分桶数量创建分区，对于已创建的分区不生效。
如需修改已有表的分区分桶数量。（**操作过程涉及数据，注意做好数据备份，关注doris系统负载**）
对于未使用Range分区的表，只能通过新建表的方式修改分桶数量，再将数据导入新表的方式进行修改。
对于已有分区，可以通过创建临时分区，将对应分区导入临时分区，之后再将临时分区设置为已有分区名称。

分桶 key 是否能修改？

不可以，分桶key只能在建表时设置，后续不能更改。

是否可以在已有表上开启自动分桶功能？

不可以，自动分桶功能只能在创建表时开启。

替换分区后原有分区是否存在？

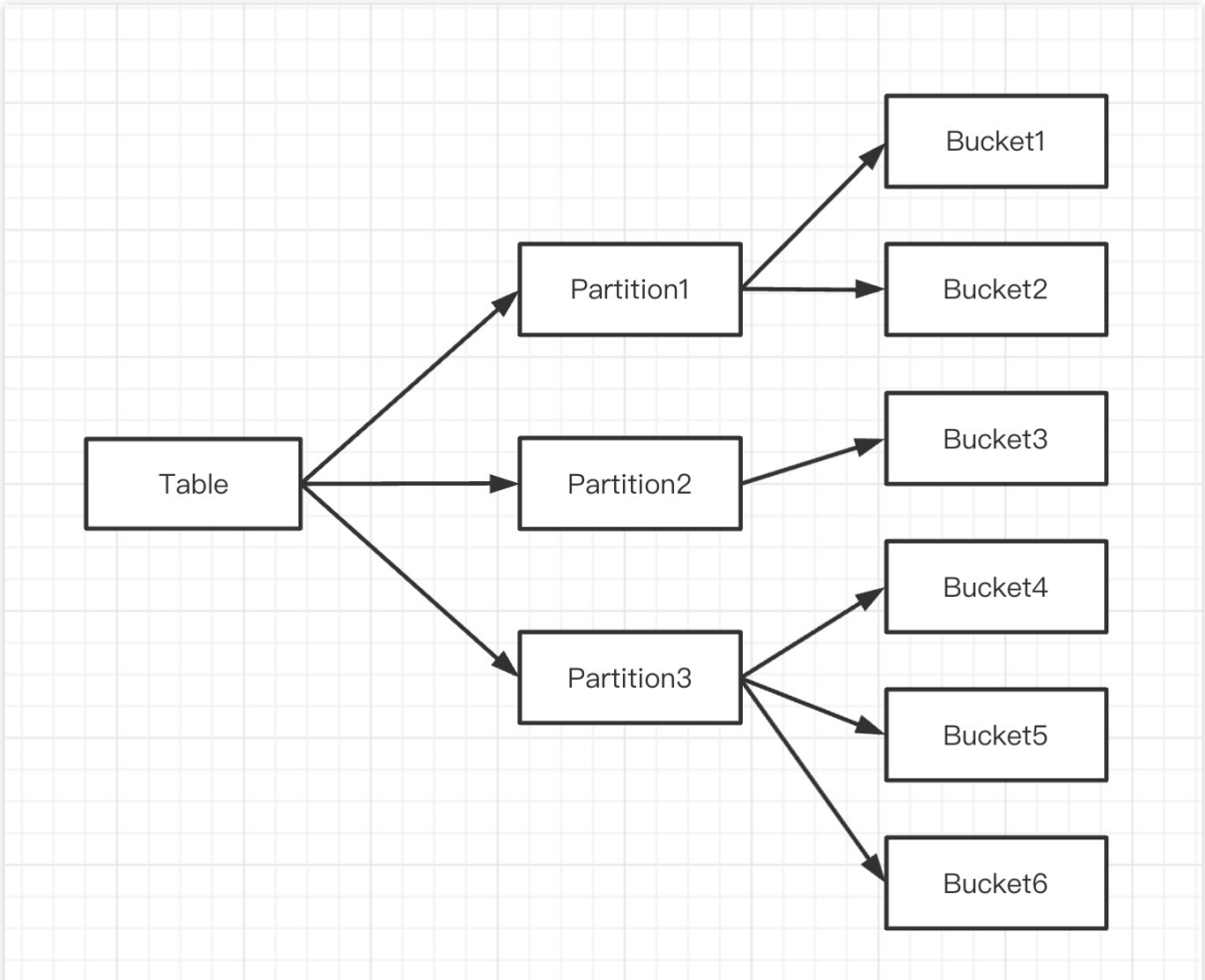
分区替换成功后，被替换的分区将被删除且不可恢复。

数据分布和副本

最近更新时间：2024-06-27 10:55:00

数据分片

Doris 表按两层结构进行数据划分，分别是分区和分桶。示意如下：



每个分桶文件就是一个数据分片（Tablet），Tablet是数据划分的最小**逻辑**单元。每个 Tablet 包含若干数据行。各个 Tablet 之间的数据没有交集，并且在物理上是独立存储的。

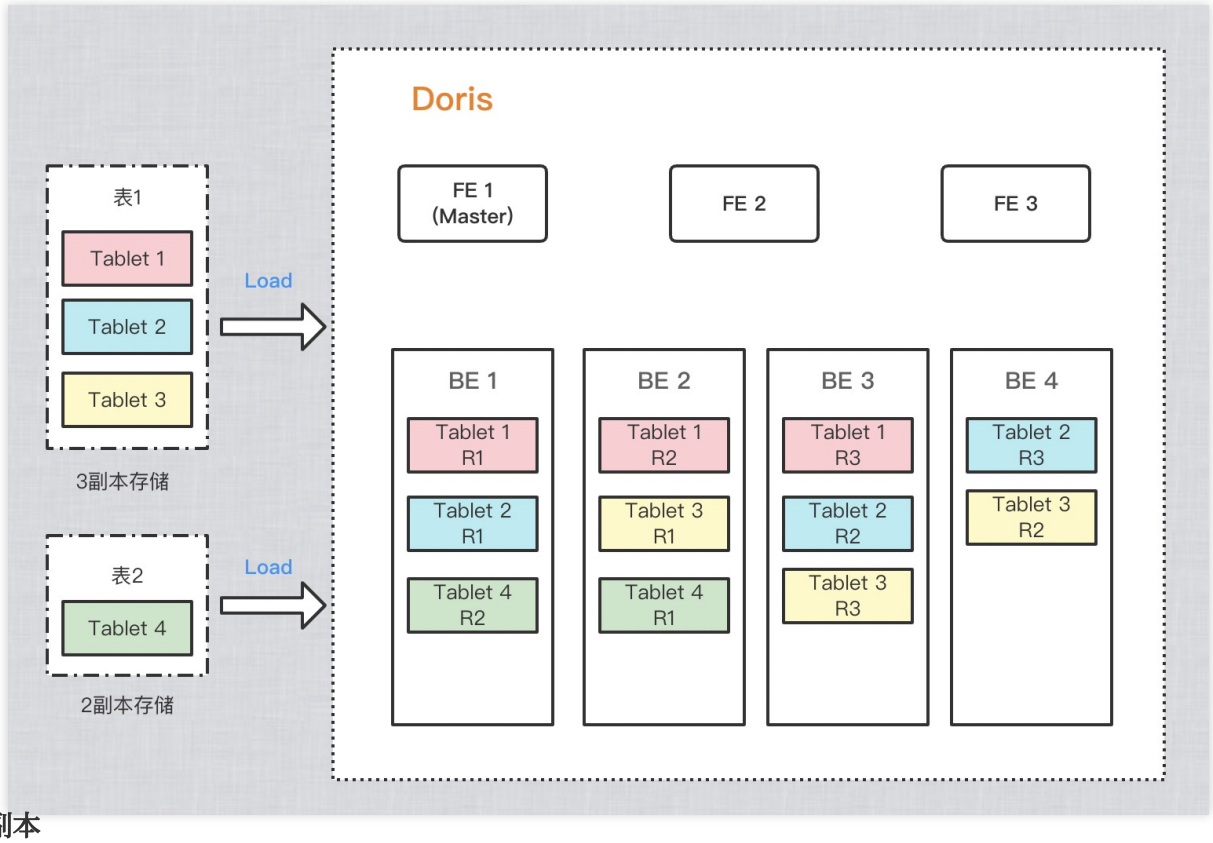
一个 Tablet 只属于一个 Partition，相应的多个 Tablet 在逻辑上归属于不同的分区（Partition）。而一个 Partition 包含若干个 Tablet。因为 Tablet 在物理上是独立存储的，所以可以视为 Partition 在物理上也是独立。Tablet 是数据移动、复制等操作的最小物理存储单元。

副本

为了提高保存数据的可靠性和计算时的性能，Doris 对每个表复制多份进行存储。数据的每份复制就叫做一个副本。Doris 按 Tablet 为基本单元对数据进行副本存储，默认一个分片有3个副本。建表时可在 PROPERTIES 中设置副本的数量：

```
PROPERTIES
    (
        "replication_num" = "3"
    );
```

下图示例，有两个表分别导入 Doris，表1导入后按3副本存储，表2导入后按2副本存储。



关于副本

每个分片的副本数量默认为3，建议保持默认即可。在建表语句中，所有 Partition 中的 Tablet 副本数量统一指定。而在增加新分区时，可以单独指定新分区中分片的副本数量。

最大副本数量取决于集群中部署BE服务的独立 IP 的数量（注意不是 BE 数量）。Doris 中副本分布的原则是：不允许同一个 Tablet 的副本分布在同一台物理机上，而识别物理机即通过 IP。所以，即使在同一台物理机上部署了 3 个或更多 BE 实例，如果这些 BE 的 IP 相同，则依然只能设置副本数为 1。

副本数量可以在运行时修改。

副本数量强烈建议保持为奇数。

Doris 负载均衡策略

在 FE 的 Master 中 tablet_rebalance_type 配置项中设置，值为 BeLoad、Partition。 如果类型解析失败，默认使用 BeLoad。

BeLoad：根据集群 BE 节点存储负载情况进行负载均衡，从负载高的 BE 节点上迁移 Replica 至负载低的节点。

Partition：均衡每个节点上 Partition 的 Replica 数量，从 Replica 数量高的BE节点上迁移 Replica 至 Replica 数量低的节点，不考虑磁盘使用情况。

横向扩容/缩容都会触发整个 Doris 集群进行负载均衡 Replica 迁移，消耗大量的 CPU 及 IO 资源，负载均衡完成前，对于系统查询/写入性能会有很大影响，需要谨慎评估处理。

最佳实践

查看数据表的数据分布信息

创建表

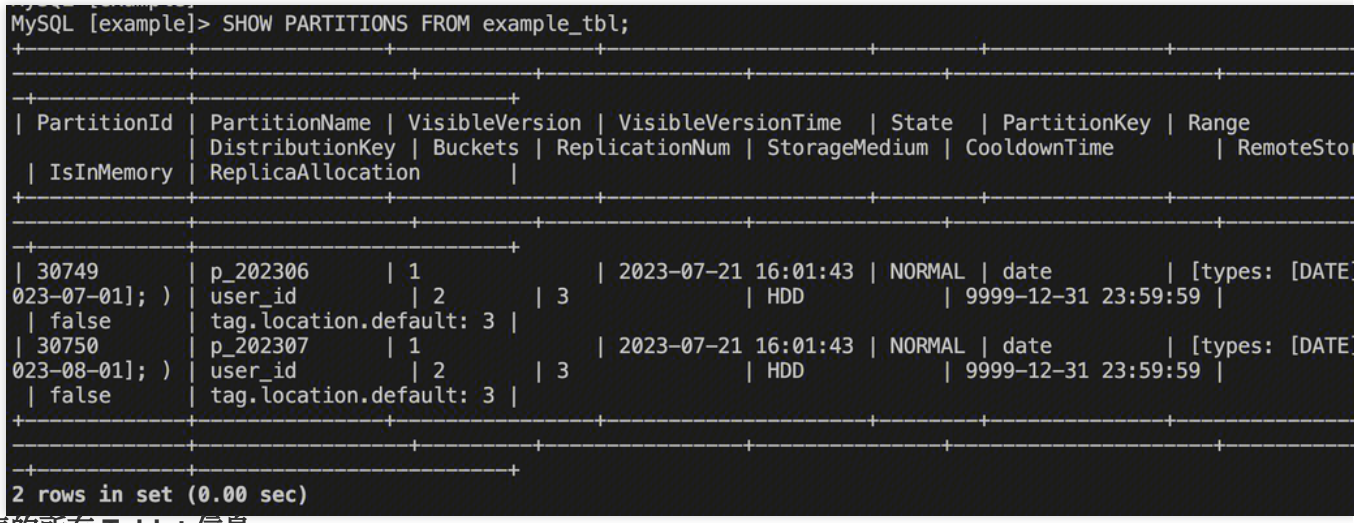
```
CREATE TABLE `example_tbl` (
  `user_id` varchar(128) NOT NULL COMMENT '用户id',
  `date` date NOT NULL COMMENT '数据灌入日期时间'
) ENGINE=OLAP
DUPLICATE KEY(`user_id`)
COMMENT 'OLAP'
PARTITION BY RANGE(`date`)
(
PARTITION p_202306 VALUES [('2023-06-01'), ('2023-07-01')),
PARTITION p_202307 VALUES [('2023-07-01'), ('2023-08-01')))
DISTRIBUTED BY HASH(`user_id`) BUCKETS 2
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
);
```

创建一个测试表，3个副本，2个分区，每个分区2个分桶。

查看表的所有 Partition 信息

```
SHOW PARTITIONS FROM example_tbl;
```

可以看到表中两个 Partition 的具体信息。



查看表的所有 Tablet 信息

```
SHOW TABLETS FROM example_tbl;
```

```
+----------+-----------+-----------+-------------+---------+-----------------+-----------------+----------
-----------------------------------+
| TabletId | ReplicaId | BackendId | SchemaHash | Version | LstSuccessVersion | LstFailedVersion | LstFailed
 | State  | LstConsistencyCheckTime | CheckVersion | VersionCount | PathHash           | MetaUrl
                                    |
+----------+-----------+-----------+-------------+---------+-----------------+-----------------+----------
-+--------+-------------------------+--------------+--------------+--------------------+-----------------
-----------------------------------+
| 30753    | 30754     | 10004     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 5949996882349618933 | http://10.0.1.11:8
0/api/compaction/show?tablet_id=30753 |
| 30753    | 30755     | 10005     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 2273776146434132252 | http://10.0.1.9:80
/api/compaction/show?tablet_id=30753  |
| 30753    | 30756     | 10003     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 3228276829640821460 | http://10.0.1.10:8
0/api/compaction/show?tablet_id=30753 |
| 30757    | 30758     | 10003     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 3228276829640821460 | http://10.0.1.10:8
0/api/compaction/show?tablet_id=30757 |
| 30757    | 30759     | 10004     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 5949996882349618933 | http://10.0.1.11:8
0/api/compaction/show?tablet_id=30757 |
| 30757    | 30760     | 10005     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 2273776146434132252 | http://10.0.1.9:80
/api/compaction/show?tablet_id=30757  |
| 30761    | 30762     | 10005     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 2273776146434132252 | http://10.0.1.9:80
/api/compaction/show?tablet_id=30761  |
| 30761    | 30763     | 10004     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 5949996882349618933 | http://10.0.1.11:8
0/api/compaction/show?tablet_id=30761 |
| 30761    | 30764     | 10003     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 3228276829640821460 | http://10.0.1.10:8
0/api/compaction/show?tablet_id=30761 |
| 30765    | 30766     | 10005     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 2273776146434132252 | http://10.0.1.9:80
/api/compaction/show?tablet_id=30765  |
| 30765    | 30767     | 10003     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 3228276829640821460 | http://10.0.1.10:8
0/api/compaction/show?tablet_id=30765 |
| 30765    | 30768     | 10004     | 1727234318 | 1       | 1                 | -1               | NULL
 | NORMAL | NULL                    | -1           | 1            | 5949996882349618933 | http://10.0.1.11:8
0/api/compaction/show?tablet_id=30765 |
+----------+-----------+-----------+-------------+---------+-----------------+-----------------+----------
-+--------+-------------------------+--------------+--------------+--------------------+-----------------
-----------------------------------+
12 rows in set (0.00 sec)
```

可以看到目前一共有2 *2* 3 = 12个 Tablet，通过 TabletId 和 ReplicaId 可以看出，ReplicaId 是唯一的，一个 TabletId 对应着3个 ReplicaId，即3个副本。

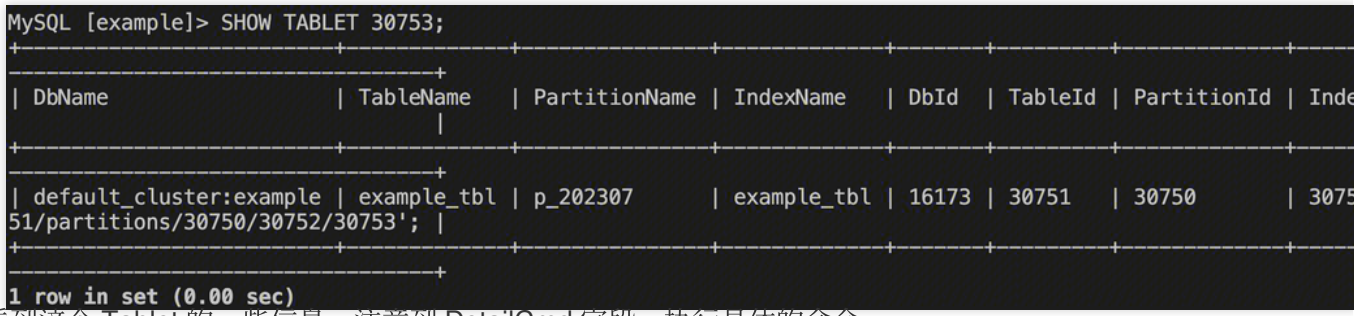
Tablet 信息中各列的具体含义如下：

列名	字段含义
TabletId	Tablet 的 ID
ReplicaId	Tablet 副本 ID
BackendId	Tablet 所在 BE 的 ID
SchemaHash	表的 schema 的哈希值，用于确保 schema 的一致性
Version	版本
LstSuccessVersion	上一次任务调度成功数据版本

LstFailedVersion	上一次任务调度失败数据版本
LstFailedTime	上一次任务调度失败时间
LocalDataSize	本地数据大小
RemoteDataSize	从远程节点获取的数据大小
RowCount	副本中的行数
State	副本的当前状态
LstConsistencyCheckTime	上一次tablet副本一致性检查时间
CheckVersion	上一次tablet副本一致性检查数据版本
VersionCount	Tablet 内的数据版本数量
PathHash	Tablet 存储路径的 hash 值
MetaUrl	查看 Tablet 的 Meta 信息的 url 地址
CompactionStatus	查看 Tablet 的 Compaction 信息的 url 地址

查看 Tablet 具体信息

```
SHOW TABLET {TabletId};
```



可以看到这个 Tablet 的一些信息，注意到 DetailCmd 字段，执行具体的命令。

```
MySQL [example]> SHOW PROC '/dbs/16173/30751/partitions/30750/30752/30753';
+-----------+-----------+---------+-------------------+-------------------+---------------+------------+-----
-------+--------------+-------------------+-----------------------------------------------------------+-----------------
| ReplicaId | BackendId | Version | LstSuccessVersion | LstFailedVersion | LstFailedTime | SchemaHash | Loca
 IsBad | VersionCount | PathHash          | MetaUrl                                                   | CompactionStatus
+-----------+-----------+---------+-------------------+-------------------+---------------+------------+-----
-------+--------------+-------------------+-----------------------------------------------------------+-----------------
| 30754     | 10004     | 1       | 1                 | -1                | NULL          | 1727234318 | 0
 false | 1            | 5949996882349618933 | http://10.0.1.11:8040/api/meta/header/30753 | http://10.0.1.11
| 30755     | 10005     | 1       | 1                 | -1                | NULL          | 1727234318 | 0
 false | 1            | 2273776146434132252 | http://10.0.1.9:8040/api/meta/header/30753  | http://10.0.1.9:
| 30756     | 10003     | 1       | 1                 | -1                | NULL          | 1727234318 | 0
 false | 1            | 3228276829640821460 | http://10.0.1.10:8040/api/meta/header/30753 | http://10.0.1.10
+-----------+-----------+---------+-------------------+-------------------+---------------+------------+-----
-------+--------------+-------------------+-----------------------------------------------------------+-----------------
3 rows in set (0.01 sec)
```

可以看到这个 Tablet 所有的 Replica 的具体信息。

查看 Tablet 健康状态

查看所有 db 的 Tablet 状态

```
SHOW PROC '/cluster_health/tablet_health'\\G
```

```
MySQL [example]> show proc '/cluster_health/tablet_health'\G
*************************** 1. row ***************************
                     DbId: 10006
                   DbName: default_cluster:doris_audit_db__
                TabletNum: 29
               HealthyNum: 29
         ReplicaMissingNum: 0
       VersionIncompleteNum: 0
       ReplicaRelocatingNum: 0
              RedundantNum: 0
 ReplicaMissingInClusterNum: 0
    ReplicaMissingForTagNum: 0
          ForceRedundantNum: 0
        ColocateMismatchNum: 0
       ColocateRedundantNum: 0
       NeedFurtherRepairNum: 0
           UnrecoverableNum: 0
ReplicaCompactionTooSlowNum: 0
           InconsistentNum: 0
               OversizeNum: 0
                CloningNum: 0
*************************** 2. row ***************************
                     DbId: 16173
                   DbName: default_cluster:example
                TabletNum: 326
               HealthyNum: 326
         ReplicaMissingNum: 0
       VersionIncompleteNum: 0
       ReplicaRelocatingNum: 0
              RedundantNum: 0
 ReplicaMissingInClusterNum: 0
    ReplicaMissingForTagNum: 0
          ForceRedundantNum: 0
        ColocateMismatchNum: 0
       ColocateRedundantNum: 0
       NeedFurtherRepairNum: 0
           UnrecoverableNum: 0
ReplicaCompactionTooSlowNum: 0
           InconsistentNum: 0
               OversizeNum: 22
                CloningNum: 0
```

主要查看 TabletNum 和 HealthyNum 是否相等，在 db 健康的情况下，预期这两个值需要是相等的。

这里显示 example 库有22个 Tablet 是超出了预期大小。

查看 db 的 Tablet 具体状态

```
SHOW PROC '/cluster_health/tablet_health/{DbId}'\\G
```

```
MySQL [example]> show proc '/cluster_health/tablet_health/16173'\G
*************************** 1. row ***************************
         ReplicaMissingTablets:
       VersionIncompleteTablets:
       ReplicaRelocatingTablets:
              RedundantTablets:
 ReplicaMissingInClusterTablets:
    ReplicaMissingForTagTablets:
          ForceRedundantTablets:
        ColocateMismatchTablets:
       ColocateRedundantTablets:
       NeedFurtherRepairTablets:
           UnrecoverableTablets:
ReplicaCompactionTooSlowTablets:
            InconsistentTablets:
              OversizeTablets: 29120,25344,25346,29125,25349,25351,29105,25329,25331,29108,29110,2533
5341,29118,25278
1 row in set (0.01 sec)
```

这里展示所有问题的 TabletId，知道 TabletId 后，可以通过上面展示过的 SHOW TABLTE 语句查看具体的 Tablet 信息，接着进行进一步的处理。

索引、排序列和前缀索引

最近更新时间：2024-06-27 10:55:17

索引

Doris 支持比较丰富的索引结构来减少数据的扫描和提高查询效率，目前支持的索引类型有：
Sorted Compound Key Index，可以最多指定三个列组成复合排序键，通过该索引，能够有效进行数据裁剪，从而能够更好支持高并发的报表场景 。
Z-order Index ：可以高效对数据模型中的任意字段组合进行范围查询。
Min/Max ：有效过滤数值类型的等值和范围查询。
Bloom Filter ：对高基数列的等值过滤裁剪非常有效。
Invert Index ：能够对任意字段实现快速检索。
不同于传统的数据库设计，Doris 不支持在任意列上创建索引。Doris 这类 MPP 架构的 OLAP 数据库，通常都是通过提高并发来处理大量数据的。

排序列（Sort Key）

为了提高查询性能，Doris 优化了数据存储的组织结构。本质上，Doris 的数据存储在类似 SSTable（Sorted String Table）的数据结构中。该结构是一种有序的数据结构，可以按照指定的列进行排序存储（可以是一列或者多列），这些列即称为排序列。在这种数据结构上，以排序列作为条件进行查找，会非常的高效。
在 Aggregate、Unique 和 Duplicate 三种数据模型中。底层的数据存储，是按照各自建表语句中，AGGREGATE KEY、UNIQUE KEY 和 DUPLICATE KEY 中指定的列进行排序存储的。Rollup 可以指定自己的排序列，但排序列必须是 Rollup 列顺序的前缀。
注意
在建表语句中的列定义中，排序列的定义必须出现在其他列的定义之前。
排序列的顺序是由 create table 语句中的列顺序决定的。

前缀索引

即在排序的数据结构（SSTable）基础上，实现的一种根据给定前缀列，快速查询数据的索引方式。对于能使用上排序结构的查询，Doris 采用二分查找算法定位到目标数据的区间。但如果表中数据行数很多，直接对排序列进行二分查找需要把所有 filter 列的数据都加载到内存中，这会消耗大量内存空间。为优化这个细节，Doris 在 Sort Key 的基础上引入稀疏的 Shortkey Index（前缀索引），Sortkey Index 的内容会比数据量少1024倍（Doris 把每1024行数据

组成一个逻辑数据块 (称作 Data Block)，每个 Data Block 在前缀索引中存储一行索引），因此会全量缓存在内存中，实际查找的过程中可以有效加速查询。

当 Sort Key 列数非常多时，会占用大量内存，为了性能考虑，对前缀索引项做了限制：

最多可选取 3 列作为 Shortkey 列。

不能使用 FLOAT / DOUBLE 类型的列。

只能按排序键的顺序来构造前缀索引。

VARCHAR / CHAR 类型列只能出现一次，并且只能是最后位置。

所有列字节数不超过36字节，VARCHAR / CHAR 列按剩余字节数折断。

当用户在建表语句中指定 short_key 属性时，例如"short_key" = "4"指定4个列作为 short_key， 可突破上述限制。

示例

1. 以下表结构的前缀索引为 user_id(8 Bytes) + age(4 Bytes) + message(prefix 20 Bytes)。

ColumnName	Type
user_id	BIGINT
age	INT
message	VARCHAR(100)
max_dwell_time	DATETIME
min_dwell_time	DATETIME

2. 以下表结构的前缀索引为 user_name(20 Bytes)。即使没有达到 36 个字节，因为遇到 VARCHAR，所以直接截断，不再往后继续。

ColumnName	Type
user_name	VARCHAR(20)
age	INT
message	VARCHAR(100)
max_dwell_time	DATETIME
min_dwell_time	DATETIME

当我们的查询条件是**前缀索引的前缀**时，可以极大的加快查询速度。例如在第一个例子中，我们执行如下查询：

```
SELECT * FROM table WHERE user_id=1829239 and age=20；
```

该查询的效率会**远高于**如下查询：

```
SELECT * FROM table WHERE age=20；
```

所以在建表时，**正确的选择列顺序，能够极大地提高查询效率**。

最佳实践

通过 ROLLUP 来调整前缀索引

因为建表时已经指定了列顺序，所以一个表只有一种前缀索引。这对于使用其他不能命中前缀索引的列作为条件进行的查询来说，效率上可能无法满足需求。因此，我们可以通过创建 ROLLUP 来人为的调整列顺序。

优化排序列的先后顺序来提高查询性能

当 Sort Key 涉及多个列的时候，需要注意先后顺序，区分度高、经常查询的列建议放在前面。
注意排序列的数量：
1. 如果选择了大量的列用于排序列，那么数据导入时排序的开销会增大整个导入过程的耗时。
2. 设计良好的少量排序列也能快速定位到数据行所在的位置，增加更多列进行排序也不会带来查询的提升。

数据导入
导入总览

最近更新时间：2024-06-27 10:55:32

导入（Load）功能就是将用户的原始数据导入到 Doris 中。导入成功后，用户即可通过 Mysql 客户端查询数据。
Doris 支持多种导入方式。建议先完整阅读本文档，再根据所选择的导入方式，查看各自导入方式的详细文档。

基本概念

1. Frontend（FE）：Doris 系统的元数据和调度节点。在导入流程中主要负责导入规划生成和导入任务的调度工作。
2. Backend（BE）：Doris 系统的计算和存储节点。在导入流程中主要负责数据的 ETL 和存储。
3. Broker：Broker 为一个独立的无状态进程。封装了文件系统接口，提供 Doris 读取远端存储系统中文件的能力。
4. 导入作业（Load job）：导入作业读取用户提交的源数据，转换或清洗后，将数据导入到 Doris 系统中。导入完成后，数据即可被用户查询到。
5. Label：所有导入作业都有一个 Label。Label 在一个数据库内唯一，可由用户指定或系统自动生成，用于标识一个导入作业。相同的 Label 仅可用于一个成功的导入作业。
6. MySQL 协议/HTTP 协议：Doris 提供两种访问协议接口。 MySQL 协议和 HTTP 协议。部分导入方式使用 MySQL 协议接口提交作业，部分导入方式使用 HTTP 协议接口提交作业。

导入方式

为适配不同的数据导入需求，Doris 系统提供了6种不同的导入方式。每种导入方式支持不同的数据源，存在不同的使用方式（异步，同步）。所有导入方式都支持 csv 数据格式。其中 Broker load 还支持 parquet 和 orc 数据格式。
每个导入方式的说明请参阅单个导入方式的操作手册。

Broker load

通过 Broker 进程访问并读取外部数据源（如 HDFS）导入到 Doris。用户通过 Mysql 协议提交导入作业后，异步执行。通过 `SHOW LOAD` 命令查看导入结果。

Stream load

用户通过 HTTP 协议提交请求并携带原始数据创建导入。主要用于快速将本地文件或数据流中的数据导入到 Doris。导入命令同步返回导入结果。

Insert

类似 MySQL 中的 Insert 语句，Doris 提供 `INSERT INTO tbl SELECT ...;` 的方式从 Doris 的表中读取数据并导入到另一张表。或者通过 `INSERT INTO tbl VALUES(...);` 插入单条数据。

Multi load

用户通过 HTTP 协议提交多个导入作业。Multi Load 可以保证多个导入作业的原子生效。

Routine load

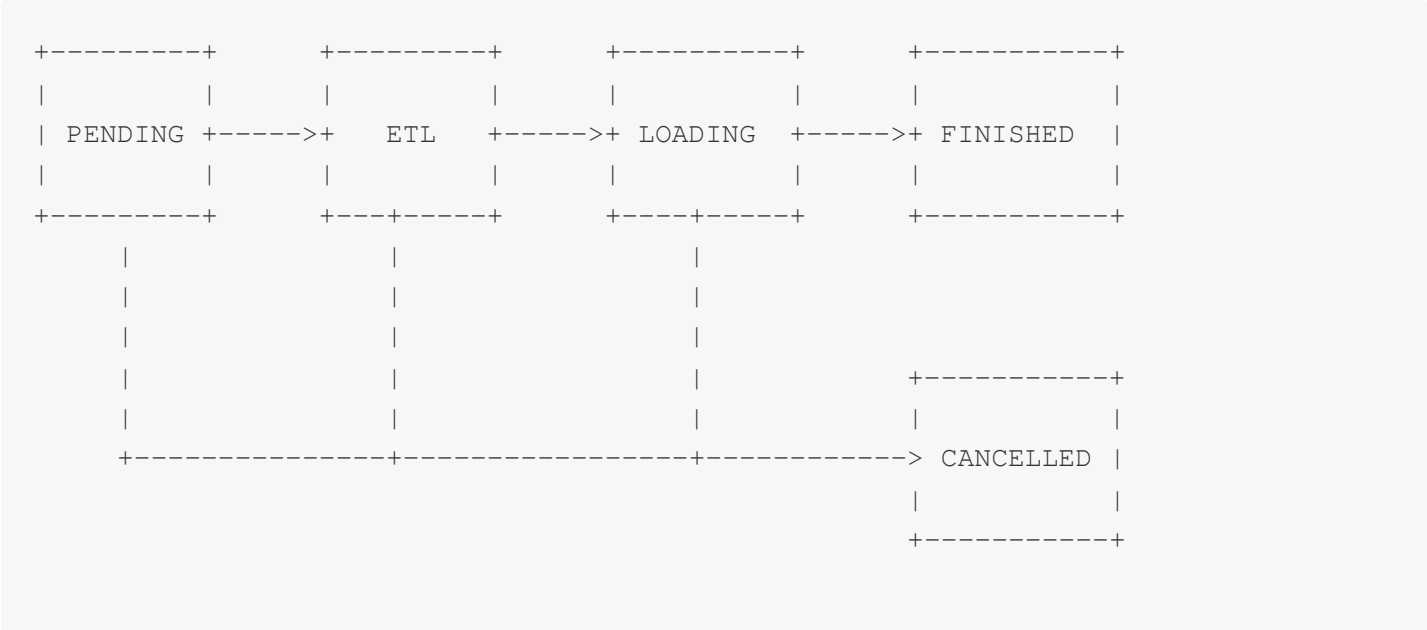
用户通过 MySQL 协议提交例行导入作业，生成一个常驻线程，不间断的从数据源（如 Kafka）中读取数据并导入到 Doris 中。

通过 S3协议直接导入

用户通过 S3协议直接导入数据，用法和 Broker Load 类似

基本原理

导入执行流程



如上图，一个导入作业主要经过上面4个阶段。

PENDING（非必须）: 该阶段只有 Broker Load 才有。Broker Load 被用户提交后会短暂停留在这个阶段，直到被 FE 中的 Scheduler 调度。 其中 Scheduler 的调度间隔为5秒。

ETL（非必须）： 该阶段在版本 0.10.0(包含) 之前存在，主要是用于将原始数据按照用户声明的方式进行变换，并且过滤不满足条件的原始数据。在 0.10.0 后的版本，ETL 阶段不再存在，其中数据 transform 的工作被合并到 LOADING 阶段。

LOADING： 该阶段在版本 0.10.0（包含）之前主要用于将变换后的数据推到对应的 BE 存储中。在 0.10.0 后的版本，该阶段先对数据进行清洗和变换，然后将数据发送到 BE 存储中。当所有导入数据均完成导入后，进入等待生效过程，此时 Load job 依旧是 LOADING。

FINISHED： 在 Load job 涉及的所有数据均生效后，Load job 的状态变成 FINISHED。FINISHED 后导入的数据均可查询。

CANCELLED: 在作业 FINISH 的之前，作业都可能被取消并进入 CANCELLED 状态。如用户手动取消，或导入出现错误等。CANCELLED 也是 Load Job 的最终状态，不可被再次执行。

上述阶段，除了 PENDING 到 LOADING 阶段是 Scheduler 轮训调度的，其他阶段之前的转移都是回调机制实现。

Label 和 原子性

Doris 对所有导入方式提供原子性保证。既保证同一个导入作业内的数据，原子生效。不会出现仅导入部分数据的情况。
同时，每一个导入作业都有一个由用户指定或者系统自动生成的 Label。Label 在一个 Database 内唯一。当一个 Label 对应的导入作业成功后，不可再重复使用该 Label 提交导入作业。如果 Label 对应的导入作业失败，则可以重复使用。
用户可以通过 Label 机制，来保证 Label 对应的数据最多被导入一次，即 At-Most-Once 语义。

同步和异步

Doris 目前的导入方式分为两类，同步和异步。如果是外部程序接入 Doris 的导入功能，需要判断使用导入方式是哪类再确定接入逻辑。

同步

同步导入方式即用户创建导入任务，Doris 同步执行导入，执行完成后返回用户导入结果。用户可直接根据创建导入任务命令返回的结果同步判断导入是否成功。
同步类型的导入方式有: **Stream load**，**Insert**。
操作步骤：
1. 用户（外部系统）创建导入任务。
2. Doris 返回导入结果。
3. 用户（外部系统）判断导入结果，如果失败可以再次提交导入任务。

注意
如果用户使用的导入方式是同步返回的，且导入的数据量过大，则创建导入请求可能会花很长时间才能返回结果。

异步

异步导入方式即用户创建导入任务后，Doris 直接返回创建成功。**创建成功不代表数据已经导入**。导入任务会被异步执行，用户在创建成功后，需要通过轮询的方式发送查看命令查看导入作业的状态。如果创建失败，则可以根据失败信息，判断是否需要再次创建。
异步类型的导入方式有：**Broker load**，**Multi load**。
操作步骤：
1. 用户（外部系统）创建导入任务。
2. Doris 返回导入创建结果。
3. 用户（外部系统）判断导入创建结果，成功则进入4，失败回到重试创建导入，回到1。
4. 用户（外部系统）轮询查看导入任务，直到状态变为 FINISHED 或 CANCELLED。

注意事项

无论是异步还是同步的导入类型，都不应该在 Doris 返回导入失败或导入创建失败后，无休止的重试。**外部系统在有限次数重试并失败后，保留失败信息，大部分多次重试均失败问题都是使用方法问题或数据本身问题。**

内存限制

用户可以通过设置参数来限制单个导入的内存使用，以防止导入占用过多的内存而导致系统OOM。
不同导入方式限制内存的方式略有不同，可以参阅各自的导入手册查看。
一个导入作业通常会分布在多个 Backend 上执行，导入内存限制的是一个导入作业，在单个 Backend 上的内存使用，而不是在整个集群的内存使用。
同时，每个 Backend 会设置可用于导入的内存的总体上限。具体配置参阅下面的通用系统配置小节。这个配置限制了所有在该 Backend 上运行的导入任务的总体内存使用上限。
较小的内存限制可能会影响导入效率，因为导入流程可能会因为内存达到上限而频繁的将内存中的数据写回磁盘。而过大的内存限制可能导致当导入并发较高时，系统 OOM。所以，需要根据需求，合理的设置导入的内存限制。

最佳实践

用户在接入 Doris 导入时，一般会采用程序接入的方式，这样可以保证数据被定期的导入到 Doris 中。下面主要说明了程序接入 Doris 的最佳实践。
1. 选择合适的导入方式：根据数据源所在位置选择导入方式。例如：如果原始数据存放在 HDFS 上，则使用 Broker load 导入。
2. 确定导入方式的协议：如果选择了 Broker load 导入方式，则外部系统需要能使用 MySQL 协议定期提交和查看导入作业。
3. 确定导入方式的类型：导入方式为同步或异步。例如 Broker load 为异步导入方式，则外部系统在提交创建导入后，必须调用查看导入命令，根据查看导入命令的结果来判断导入是否成功。
4. 制定 Label 生成策略：Label 生成策略需满足，每一批次数据唯一且固定的原则。这样 Doris 就可以保证 At-Most-Once。
5. 程序自身保证 At-Least-Once：外部系统需要保证自身的 At-Least-Once，这样就可以保证导入流程的 Exactly-Once。

通用系统配置

下面主要解释了几个所有导入方式均通用的系统级别的配置。

FE 配置

以下配置属于 FE 的系统配置，可以通过修改 FE 的配置文件 `fe.conf` 来修改配置。

max_load_timeout_second 和 min_load_timeout_second

这两个配置含义为：最大的导入超时时间，最小的导入超时时间，以秒为单位。默认的最大超时时间为3天, 默认的最小超时时间为1秒。用户自定义的导入超时时间不可超过这个范围。该参数通用于所有的导入方式。

desired_max_waiting_jobs

在等待队列中的导入任务个数最大值，默认为100。当在 FE 中处于 PENDING 状态（也就是等待执行的）导入个数超过该值，新的导入请求则会被拒绝。

此配置仅对异步执行的导入有效，当异步执行的导入等待个数超过默认值，则后续的创建导入请求会被拒绝。

max_running_txn_num_per_db

这个配置的含义是说，每个 Database 中正在运行的导入最大个数（不区分导入类型，统一计数）。默认的最大导入并发为 100。当前 Database 正在运行的导入个数超过最大值时，后续的导入不会被执行。如果是同步导入作业，则导入会被拒绝。如果是异步导入作业。则作业会在队列中等待。

BE 配置

以下配置属于 BE 的系统配置，可以通过修改 BE 的配置文件 `be.conf` 来修改配置。

push_write_mbytes_per_sec

BE 上单个 Tablet 的写入速度限制。默认是 10，即 10MB/s。通常 BE 对单个 Tablet 的最大写入速度，根据 Schema 以及系统的不同，大约在 10-30MB/s 之间。可以适当调整这个参数来控制导入速度。

write_buffer_size

导入数据在 BE 上会先写入一个 memtable，memtable 达到阈值后才会写回磁盘。默认大小是 100MB。过小的阈值可能导致 BE 上存在大量的小文件。可以适当提高这个阈值减少文件数量。但过大的阈值可能导致 RPC 超时，见下面的配置说明。

tablet_writer_rpc_timeout_sec

导入过程中，发送一个 Batch（1024行）的 RPC 超时时间。默认 600 秒。因为该 RPC 可能涉及多个 memtable 的写盘操作，所以可能会因为写盘导致 RPC 超时，可以适当调整这个超时时间来减少超时错误（如 `send batch fail` 错误）。同时，如果调大 `write_buffer_size` 配置，也需要适当调大这个参数。

streaming_load_rpc_max_alive_time_sec

在导入过程中，Doris 会为每一个 Tablet 开启一个 Writer，用于接收数据并写入。这个参数指定了 Writer 的等待超时时间。如果在这个时间内，Writer 没有收到任何数据，则 Writer 会被自动销毁。当系统处理速度较慢时，Writer 可能长时间接收不到下一批数据，导致导入报错： `TabletWriter add batch with unknown id` 。此时可适当增大这个配置。默认为 600 秒。

load_process_max_memory_limit_bytes 和 load_process_max_memory_limit_percent

这两个参数，限制了单个 Backend 上，可用于导入任务的内存上限。分别是最大内存和最大内存百分比。 `load_process_max_memory_limit_percent` 默认为 80，表示对 Backend 总内存限制的百分比（总内存限制 `mem_limit` 默认为 80%，表示对物理内存的百分比）。即假设物理内存为 M，则默认导入内存限制为 M *80%* 80%。

`load_process_max_memory_limit_bytes` 默认为 100GB。系统会在两个参数中取较小者，作为最终的 Backend 导入内存使用上限。

label_keep_max_second

设置导入任务记录保留时间。已经完成的（ FINISHED or CANCELLED ）导入任务记录会保留在 Doris 系统中一段时间，时间由此参数决定。参数默认值时间为3天。该参数通用与所有类型的导入任务。

列映射

假设导入数据有 `1，2，3` ，表有 `c1,c2,c3` 三列，如果数据直接导入表中可以使用如下语句 `COLUMNS(c1,c2,c3)` 此语句等价于 `COLUMNS(tmp_c1,tmp_c2,tmp_c3,c1=tmp_c1,c2=tmp_c2,c3=tmp_c3)` 。

如果想再导入数据时执行变换或者使用临时变量，则变换或者临时变量一定要按照使用的顺序指定， 例如 `COLUMNS(tmp_c1,tmp_c2,tmp_c3, c1 = tmp_c1 +1, c2= c1+1, c3 =c2+1)` ,这样的语句等价于 `COLUMNS(tmp_c1,tmp_c2,tmp_c3, c1 = tmp_c1 +1, c2= tmp_c1 +1+1, c3 =tmp_c1 +1+1+1)` 。

在使用某个表达式时这个表达式一定要在前面定义，例如如下语句则不合法 `COLUMNS(tmp_c1,tmp_c2,tmp_c3, c1 = c1+1, c2 = temp + 1, temp = tmp_c1 +1, c3 =c2+1)` 。

导入方式

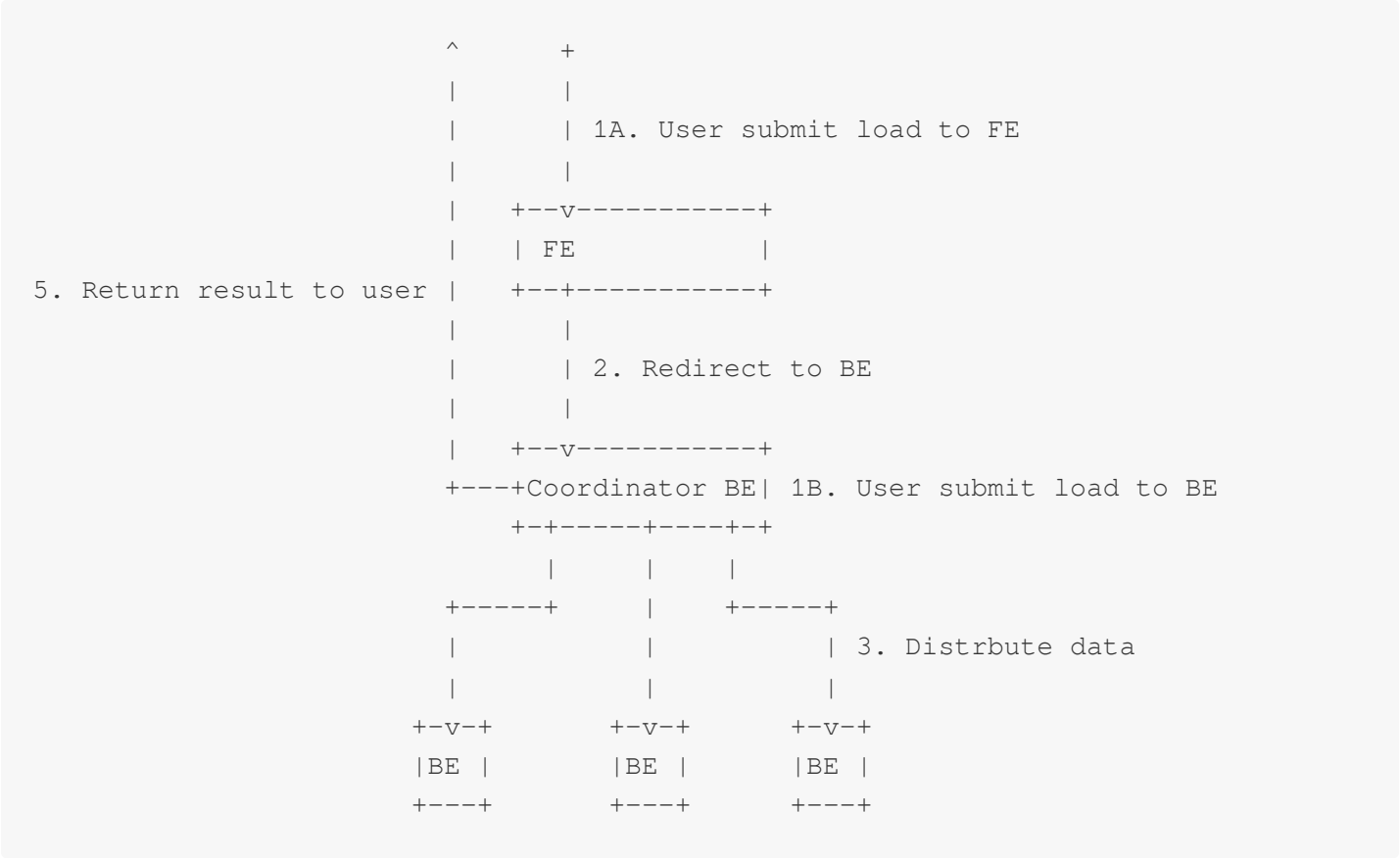
Stream load（本地文件）

最近更新时间：2024-06-27 10:55:46

Stream load 是一个同步的导入方式，用户通过发送 HTTP 协议发送请求将本地文件或数据流导入到 Doris 中。
Stream load 同步执行导入并返回导入结果。用户可直接通过请求的返回体判断本次导入是否成功。
Stream load 主要适用于导入本地文件，或通过程序导入数据流中的数据。

基本原理

下图展示了 Stream load 的主要流程，省略了一些导入细节。



Stream load 中，Doris 会选定一个节点作为 Coordinator 节点。该节点负责接数据并分发数据到其他数据节点。
用户通过 HTTP 协议提交导入命令。如果提交到 FE，则 FE 会通过 HTTP redirect 指令将请求转发给某一个 BE。用户也可以直接提交导入命令给某一指定 BE。导入的最终结果由 Coordinator BE 返回给用户。因此，发起导入请求的机器需要有访问 BE 节点 http 服务端口的权限。
FE 节点默认 http 服务端口为8030，BE 节点默认 http 服务端口为8040（可通过show frontends;show backends;等语句查看）。

支持数据格式

目前 Stream load 支持两个数据格式：CSV（文本） 和 JSON。

基本操作

创建导入

Stream load 通过 HTTP 协议提交和传输数据。这里通过 `curl` 命令展示如何提交导入。
用户也可以通过其他 HTTP client 进行操作。

```
curl --location-trusted -u user:passwd [-H ""...] -T data.file -XPUT http://fe_host

# Header 中支持属性见下面的 ‘导入任务参数’ 说明
# 格式为: -H "key1:value1"
```

示例：

```
curl --location-trusted -u root -T date -H "label:123" http://abc.com:8030/api/test
```

创建导入的详细语法帮助执行 `HELP STREAM LOAD` 查看，下面主要介绍创建 Stream load 的部分参数意义。
签名参数
user/passwd
Stream load 由于创建导入的协议使用的是 HTTP 协议，通过 Basic access authentication 进行签名。Doris 系统会根据签名验证用户身份和导入权限。
导入任务参数
Stream load 由于使用的是 HTTP 协议，所以所有导入任务有关的参数均设置在 Header 中。下面主要介绍了 Stream load 导入任务参数的部分参数意义。
label
导入任务的标识。每个导入任务，都有一个在单 database 内部唯一的 label。label 是用户在导入命令中自定义的名称。通过这个 label，用户可以查看对应导入任务的执行情况。
label 的另一个作用，是防止用户重复导入相同的数据。**强烈推荐用户同一批次数据使用相同的 label。这样同一批次数据的重复请求只会被接受一次，保证了 At-Most-Once**。
当 label 对应的导入作业状态为 CANCELLED 时，该 label 可以再次被使用。
column_separator
用于指定导入文件中的列分隔符，默认为\\t。如果是不可见字符，则需要加\\x作为前缀，使用十六进制来表示分隔符。
如 hive 文件的分隔符\\x01，需要指定为-H "column_separator:\\x01"。
可以使用多个字符的组合作为列分隔符。

line_delimiter

用于指定导入文件中的换行符，默认为\\n。

可以使用做多个字符的组合作为换行符。

max_filter_ratio

导入任务的最大容忍率，默认为0容忍，取值范围是0~1。当导入的错误率超过该值，则导入失败。

如果用户希望忽略错误的行，可以通过设置这个参数大于 0，来保证导入可以成功。

计算公式为：

`(dpp.abnorm.ALL / (dpp.abnorm.ALL + dpp.norm.ALL)) > max_filter_ratio``dpp.abnorm.ALL` 表示数据质量不合格的行数。如类型不匹配，列数不匹配，长度不匹配等。

`dpp.norm.ALL` 指的是导入过程中正确数据的条数。可以通过 `SHOW LOAD` 命令查询导入任务的正确数据量。

原始文件的行数 = `dpp.abnorm.ALL + dpp.norm.ALL`

where

导入任务指定的过滤条件。Stream load 支持对原始数据指定 where 语句进行过滤。被过滤的数据将不会被导入，也不会参与 filter ratio 的计算，但会被计入 `num_rows_unselected` 。

Partitions

待导入表的 Partition 信息，如果待导入数据不属于指定的 Partition 则不会被导入。这些数据将计入 `dpp.abnorm.ALL` 。

columns

待导入数据的函数变换配置，目前 Stream load 支持的函数变换方法包含列的顺序变化以及表达式变换，其中表达式变换的方法与查询语句的一致。

```
列顺序变换例子：原始数据有三列(src_c1,src_c2,src_c3)， 目前doris表也有三列（dst_c1,dst_c2,ds

如果原始表的src_c1列对应目标表dst_c1列，原始表的src_c2列对应目标表dst_c2列，原始表的src_c3列对
columns: dst_c1, dst_c2, dst_c3

如果原始表的src_c1列对应目标表dst_c2列，原始表的src_c2列对应目标表dst_c3列，原始表的src_c3列对
columns: dst_c2, dst_c3, dst_c1

表达式变换例子：原始文件有两列，目标表也有两列（c1,c2）但是原始文件的两列均需要经过函数变换才能对应
columns: tmp_c1, tmp_c2, c1 = year(tmp_c1), c2 = month(tmp_c2)
其中 tmp_*是一个占位符，代表的是原始文件中的两个原始列。
```

exec_mem_limit

导入内存限制。默认为 2GB，单位为字节。

strict_mode

Stream Load 导入可以开启 strict mode 模式。开启方式为在 HEADER 中声明 `strict_mode=true` 。默认的 strict mode 为关闭。

strict mode 模式的意思是：对于导入过程中的列类型转换进行严格过滤。严格过滤的策略如下：

1.1 对于列类型转换来说，如果 strict mode 为true，则错误的数据将被 filter。这里的错误数据是指：原始数据并不为空值，在参与列类型转换后结果为空值的这一类数据。

1.2 对于导入的某列由函数变换生成时，strict mode 对其不产生影响。

1.3 对于导入的某列类型包含范围限制的，如果原始数据能正常通过类型转换，但无法通过范围限制的，strict mode 对其也不产生影响。例如：如果类型是 decimal(1,0), 原始数据为 10，则属于可以通过类型转换但不在列声明的范围内。这种数据 strict 对其不产生影响。

merge_type

数据的合并类型，一共支持三种类型 APPEND、DELETE、MERGE 其中，APPEND 是默认值，表示这批数据全部需要追加到现有数据中，DELETE 表示删除与这批数据 key 相同的所有行，MERGE 语义 需要与 delete 条件联合使用，表示满足 delete 条件的数据按照 DELETE 语义处理其余的按照 APPEND 语义处理

two_phase_commit

Stream load 导入可以开启两阶段事务提交模式：在 Stream load 过程中，数据写入完成即会返回信息给用户，此时数据不可见，事务状态为 `PRECOMMITTED`，用户手动触发 commit 操作之后，数据才可见。

默认的两阶段批量事务提交为关闭。

说明

开启方式： 在 be.conf 中配置 `disable_stream_load_2pc=false` 并且 在 HEADER 中声明 `two_phase_commit=true` 。

示例：

1. 发起 stream load 预提交操作：

```
curl  --location-trusted -u user:passwd -H "two_phase_commit:true" -T test.txt http
{
"TxnId": 18036,
"Label": "55c8ffc9-1c40-4d51-b75e-f2265b3602ef",
"TwoPhaseCommit": "true",
"Status": "Success",
"Message": "OK",
"NumberTotalRows": 100,
"NumberLoadedRows": 100,
"NumberFilteredRows": 0,
"NumberUnselectedRows": 0,
"LoadBytes": 1031,
"LoadTimeMs": 77,
"BeginTxnTimeMs": 1,
"StreamLoadPutTimeMs": 1,
"ReadDataTimeMs": 0,
"WriteDataTimeMs": 58,
"CommitAndPublishTimeMs": 0
}
```

2. 对事务触发 commit 操作：

```
curl -X PUT --location-trusted -u user:passwd  -H "txn_id:18036" -H "txn_operation:
```

```
{
"status": "Success",
"msg": "transaction [18036] commit successfully."
}
```

3. 对事务触发 abort 操作：

```
curl -X PUT --location-trusted -u user:passwd  -H "txn_id:18037" -H "txn_operation:
{
"status": "Success",
"msg": "transaction [18037] abort successfully."
}
```

返回结果

由于 Stream load 是一种同步的导入方式，所以导入的结果会通过创建导入的返回值直接返回给用户。

示例：

```
{
    "TxnId": 1003,
    "Label": "b6f3bc78-0d2c-45d9-9e4c-faa0a0149bee",
    "Status": "Success",
    "ExistingJobStatus": "FINISHED", // optional
    "Message": "OK",
    "NumberTotalRows": 1000000,
    "NumberLoadedRows": 1000000,
    "NumberFilteredRows": 1,
    "NumberUnselectedRows": 0,
    "LoadBytes": 40888898,
    "LoadTimeMs": 2144,
    "BeginTxnTimeMs": 1,
    "StreamLoadPutTimeMs": 2,
    "ReadDataTimeMs": 325,
    "WriteDataTimeMs": 1933,
    "CommitAndPublishTimeMs": 106,
    "ErrorURL": "http://192.168.1.1:8042/api/_load_error_log?file=__shard_0/error_l
}
```

下面主要解释了 Stream load 导入结果参数：

TxnId：导入的事务ID。用户可不感知。

Label：导入 Label。由用户指定或系统自动生成。

Status：导入完成状态。

"Success"：表示导入成功。

"Publish Timeout"：该状态也表示导入已经完成，只是数据可能会延迟可见，无需重试。

"Label Already Exists"：Label 重复，需更换 Label。

"Fail"：导入失败。
ExistingJobStatus：已存在的 Label 对应的导入作业的状态。
这个字段只有在当 Status 为 "Label Already Exists" 时才会显示。用户可以通过这个状态，知晓已存在 Label 对应的导入作业的状态。"RUNNING" 表示作业还在执行，"FINISHED" 表示作业成功。
Message：导入错误信息。
NumberTotalRows：导入总处理的行数。
NumberLoadedRows：成功导入的行数。
NumberFilteredRows：数据质量不合格的行数。
NumberUnselectedRows：被 where 条件过滤的行数。
LoadBytes：导入的字节数。
LoadTimeMs：导入完成时间。单位毫秒。
BeginTxnTimeMs：向Fe请求开始一个事务所花费的时间，单位毫秒。
StreamLoadPutTimeMs：向Fe请求获取导入数据执行计划所花费的时间，单位毫秒。
ReadDataTimeMs：读取数据所花费的时间，单位毫秒。
WriteDataTimeMs：执行写入数据操作所花费的时间，单位毫秒。
CommitAndPublishTimeMs：向Fe请求提交并且发布事务所花费的时间，单位毫秒。
ErrorURL：如果有数据质量问题，通过访问这个 URL 查看具体错误行。
注意
由于 Stream load 是同步的导入方式，所以并不会在 Doris 系统中记录导入信息，用户无法异步的通过查看导入命令看到 Stream load。使用时需监听创建导入请求的返回值获取导入结果。

取消导入

用户无法手动取消 Stream load，Stream load 在超时或者导入错误后会被系统自动取消。

查看 Stream Load

用户可以通过 `show stream load` 来查看已经完成的 stream load 任务。
默认 BE 是不记录 Stream Load 的记录，如果您要查看需要在 BE 上启用记录，配置参数是：`enable_stream_load_record=true` ，具体配置请参见 BE 配置项。

相关系统配置

FE 配置

stream_load_default_timeout_second
导入任务的超时时间(以秒为单位)，导入任务在设定的 timeout 时间内未完成则会被系统取消，变成 CANCELLED。默认的 timeout 时间为 600 秒。如果导入的源文件无法在规定时间内完成导入，用户可以在 stream load 请求中设置单独的超时时间。
或者调整 FE 的参数 `stream_load_default_timeout_second` 来设置全局的默认超时时间。

BE 配置

streaming_load_max_mb

Stream Load 的最大导入大小，默认为 10G，单位是 MB。如果用户的原始文件超过这个值，则需要调整 BE 的参数 `streaming_load_max_mb` 。

最佳实践

应用场景

使用 Stream load 的最合适场景就是原始文件在内存中或者在磁盘中。其次，由于 Stream load 是一种同步的导入方式，所以用户如果希望用同步方式获取导入结果，也可以使用这种导入。

数据量

由于 Stream load 的原理是由 BE 发起的导入并分发数据，建议的导入数据量在 1G 到 10G 之间。由于默认的最大 Stream load 导入数据量为 10G，所以如果要导入超过 10G 的文件需要修改 BE 的配置 `streaming_load_max_mb`

```
比如：待导入文件大小为15G
修改 BE 配置 streaming_load_max_mb 为 16000 即可。
```

Stream load 的默认超时为 300秒，按照 Doris 目前最大的导入限速来看，约超过 3G 的文件就需要修改导入任务默认超时时间了。

```
导入任务超时时间 = 导入数据量 / 10M/s （具体的平均导入速度需要用户根据自己的集群情况计算）
例如：导入一个 10G 的文件
timeout = 1000s 等于 10G / 10M/s
```

完整例子

数据情况： 数据在发送导入请求端的本地磁盘路径 /home/store_sales 中，导入的数据量约为 15G，希望导入到数据库 bj_sales 的表 store_sales 中。

集群情况：Stream load 的并发数不受集群大小影响。

step1：导入文件大小是否超过默认的最大导入大小10G。

```
修改 BE conf
streaming_load_max_mb = 16000
```

step2：计算大概的导入时间是否超过默认 timeout 值。

```
导入时间 ≈ 15000 / 10 = 1500s
超过了默认的 timeout 时间，需要修改 FE 的配置
stream_load_default_timeout_second = 1500
```

step3：创建导入任务。

```
curl --location-trusted -u user:password -T /home/store_sales -H "label:abc" http:/
```

常见问题

出现 Label Already Exists 该如何解决?

Stream load 的 Label 重复排查步骤如下：

1. 是否和其他导入方式已经存在的导入 Label 冲突：

由于 Doris 系统中导入的 Label 不区分导入方式，所以存在其他导入方式使用了相同 Label 的问题。

通过 `SHOW LOAD WHERE LABEL = “xxx”` ，其中 xxx 为重复的 Label 字符串，查看是否已经存在一个 FINISHED 导入的 Label 和用户申请创建的 Label 相同。

2. 是否 Stream load 同一个作业被重复提交了：

由于 Stream load 是 HTTP 协议提交创建导入任务，一般各个语言的 HTTP Client 均会自带请求重试逻辑。Doris 系统在接受到第一个请求后，已经开始操作 Stream load，但是由于没有及时返回给 Client 端结果， Client 端会发生再次重试创建请求的情况。这时候 Doris 系统由于已经在操作第一个请求，所以第二个请求已经就会被报 Label Already Exists 的情况。

排查上述可能的方法：使用 Label 搜索 FE Master 的日志，看是否存在同一个 Label 出现了两次 `redirect load action to destination=` 的情况。如果有就说明，请求被 Client 端重复提交了。

建议用户根据当前请求的数据量，计算出大致导入的时间，并根据导入超时时间，将 Client 端请求超时的时间改成大于导入超时的时间值，避免请求被 Client 端多次提交。

3. Connection reset 异常

在0.14.0及之前的版本启用 HTTP V2之后出现 connection reset 异常，因为 Web 容器内置的是 tomcat，对这个协议实现是有问题的。在使用 Stream load 导入大数据量的情况下会出现 connect reset 异常是因为 tomcat 在做307跳转之前就开始了数据传输，造成 BE 收到的数据请求时缺少认证信息。后续版本已将内置容器改成了 Jetty 解决了这个问题，如果您遇到这个问题，请升级版本或者禁用 HTTP V2（ `enable_http_server_v2=false` ）。

升级以后同时升级您程序的 httpclient 版本到 `4.5.13` ，在您的 pom.xml 文件中引入下面的依赖：

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.13</version>
</dependency>
```

更多帮助

您可以在 MySql 客户端命令行下输入 `HELP STREAM LOAD` 获取更多帮助信息。

Broker Load（HDFS 数据）

最近更新时间：2024-06-27 10:56:01

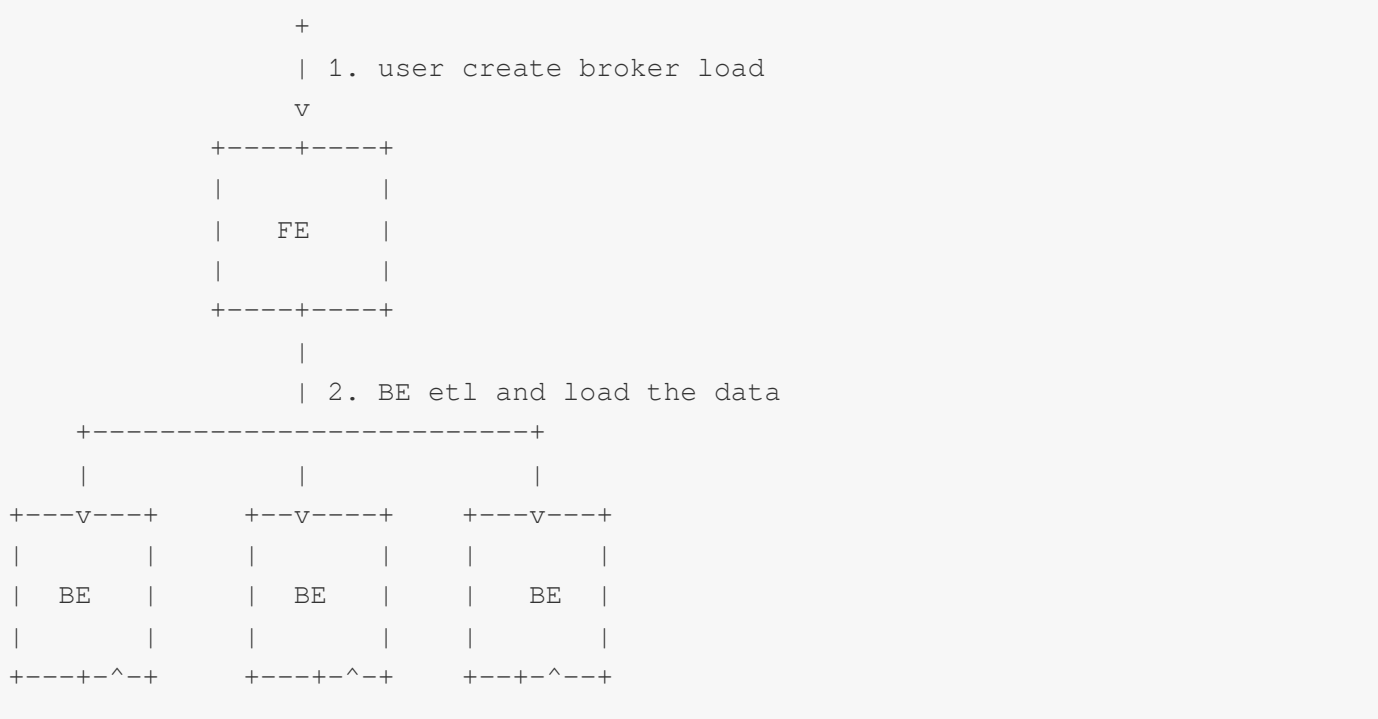
Broker load 是一种异步的导入方式，支持的数据源取决于 Broker 进程支持的数据源。而一般有支持社区版 HDFS 的 Broker 和支持 S3协议对象存储的 Broker。本文讲解如何使用 Broker load 导入 HDFS 数据。
因为 Doris 表里的数据是有序的，所以 Broker load 在导入数据时，要利用 Doris 集群资源对数据进行排序，相对于 Spark load 来完成海量历史数据迁移，对 Doris 的集群资源占用较大。因此，这种方式多是在用户没有 Spark 这种计算资源的情况下才使用，如果有 Spark 计算资源建议使用 Spark load。
用户需要通过 MySQL 协议 创建 Broker load 导入，并通过查看导入命令检查导入结果。

适用场景

源数据在 Broker 可以访问的存储系统中，如 HDFS。
数据量在几十到百 GB 级别。

基本原理

用户在提交导入任务后，FE 会生成对应的 Plan 并根据目前 BE 的个数和文件的大小，将 Plan 分给多个 BE 执行，每个 BE 执行一部分导入数据。
BE 在执行的过程中会从 Broker 拉取数据，在对数据 transform 之后将数据导入系统。所有 BE 均完成导入后，由 FE 最终决定导入是否成功。



```
    | |           | |         | |
    | |           | |         | | 3. pull data from broker
+---v-+-+     +---v-+-+    +--v-+--+
|       |     |       |    |       |
|Broker |     |Broker |    |Broker |
|       |     |       |    |       |
+---+-^-+     +---+-^-+    +---+-^-+
    | |           | |          | |
+---v-+-----------v-+----------v-+-+
|       HDFS/BOS/AFS cluster       |
|                                  |
+----------------------------------+
```

开始导入

下面我们通过几个实际的场景示例来看 Broker load 的使用。

Hive 分区表的数据导入

1. 创建 Hive 表。

```
##数据格式是：默认，分区字段是：day
CREATE TABLE `ods_demo_detail`(
  `id` string,
  `store_id` string,
  `company_id` string,
  `tower_id` string,
  `commodity_id` string,
  `commodity_name` string,
  `commodity_price` double,
  `member_price` double,
  `cost_price` double,
  `unit` string,
  `quantity` double,
  `actual_price` double
)
PARTITIONED BY (day string)
row format delimited fields terminated by ','
lines terminated by '\\n'
```

使用 Hive 的 Load 命令将您的数据导入到 Hive 表中：

```
load data local inpath '/opt/custorm' into table ods_demo_detail;
```

2. 创建 Doris 表，具体建表语法参照 CREATE TABLE。

```
CREATE TABLE `doris_ods_test_detail` (
  `rq` date NULL,
  `id` varchar(32) NOT NULL,
  `store_id` varchar(32) NULL,
  `company_id` varchar(32) NULL,
  `tower_id` varchar(32) NULL,
  `commodity_id` varchar(32) NULL,
  `commodity_name` varchar(500) NULL,
  `commodity_price` decimal(10, 2) NULL,
  `member_price` decimal(10, 2) NULL,
  `cost_price` decimal(10, 2) NULL,
  `unit` varchar(50) NULL,
  `quantity` int(11) NULL,
  `actual_price` decimal(10, 2) NULL
) ENGINE=OLAP
UNIQUE KEY(`rq`, `id`, `store_id`)
PARTITION BY RANGE(`rq`)
(
PARTITION P_202204 VALUES [('2022-04-01'), ('2022-05-01')))
DISTRIBUTED BY HASH(`store_id`) BUCKETS 1
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"dynamic_partition.enable" = "true",
"dynamic_partition.time_unit" = "MONTH",
"dynamic_partition.start" = "-2147483648",
"dynamic_partition.end" = "2",
"dynamic_partition.prefix" = "P_",
"dynamic_partition.buckets" = "1",
"in_memory" = "false",
"storage_format" = "V2"
);
```

3. 开始导入数据，具体语法参照 Broker Load 。

```
LOAD LABEL broker_load_2022_03_23
(
    DATA INFILE("hdfs://192.168.20.123:8020/user/hive/warehouse/ods.db/ods_demo_det
    INTO TABLE doris_ods_test_detail
    COLUMNS TERMINATED BY ","
  (id,store_id,company_id,tower_id,commodity_id,commodity_name,commodity_price,memb
    COLUMNS FROM PATH AS (`day`)
   SET
   (rq = str_to_date(`day`,'%Y-%m-%d'),id=id,store_id=store_id,company_id=company_i
    )
WITH BROKER "broker_name_1"
```

```
    (
      "username" = "hdfs",
      "password" = ""
    )
PROPERTIES
(
    "timeout"="1200",
    "max_filter_ratio"="0.1"
);
```

说明

`192.168.20.123:8020` 是 hive 表所用 HDFS 集群的 active namenode 的 IP 和 Port。

Hive 分区表导入(ORC格式)

1. 创建 Hive 分区表，ORC 格式。

```
#数据格式:ORC 分区:day
CREATE TABLE `ods_demo_orc_detail`(
  `id` string,
  `store_id` string,
  `company_id` string,
  `tower_id` string,
  `commodity_id` string,
  `commodity_name` string,
  `commodity_price` double,
  `member_price` double,
  `cost_price` double,
  `unit` string,
  `quantity` double,
  `actual_price` double
)
PARTITIONED BY (day string)
row format delimited fields terminated by ','
lines terminated by '\\n'
STORED AS ORC
```

2. 创建 Doris 表，建表语句和上面的 Doris 建表语句一样。

3. 使用 Broker load 导入数据。

```
LOAD LABEL dish_2022_03_23
(
  DATA INFILE("hdfs://10.220.147.151:8020/user/hive/warehouse/ods.db/ods_demo_orc_d
  INTO TABLE doris_ods_test_detail
  COLUMNS TERMINATED BY ","
  FORMAT AS "orc"
(id,store_id,company_id,tower_id,commodity_id,commodity_name,commodity_price,member
```

```
   COLUMNS FROM PATH AS (`day`)
  SET
  (rq = str_to_date(`day`,'%Y-%m-%d'),id=id,store_id=store_id,company_id=company_id,
)
WITH BROKER "broker_name_1"
(
 "username" = "hdfs",
 "password" = ""
)
PROPERTIES
(
  "timeout"="1200",
  "max_filter_ratio"="0.1"
);
```

注意

`FORMAT AS "orc"` ：指定了要导入的数据的格式。

`SET` ：定义了 Hive 表和 Doris 表之间的字段映射关系及字段转换的一些操作。

HDFS 文件系统数据导入

继续以上面创建好的 Doris 表为例，演示通过 Broker Load 从 HDFS 上导入数据。

导入作业的语句如下：

```
LOAD LABEL demo.label_20220402
        (
            DATA INFILE("hdfs://10.220.147.151:8020/tmp/test_hdfs.txt")
            INTO TABLE `ods_dish_detail_test`
            COLUMNS TERMINATED BY "\\t"            (id,store_id,company_id,tower_id
        )
        with HDFS (
            "fs.defaultFS"="hdfs://10.220.147.151:8020",
            "hdfs_user"="root"
        )
        PROPERTIES
        (
            "timeout"="1200",
            "max_filter_ratio"="0.1"
        );
```

这里的具体 参数可以参照：Broker 及 Broker Load 文档。

查看导入状态

我们可以通过下面的命令查看上面导入任务的状态信息，具体的查看导入状态的语法参考 SHOW LOAD。

```
mysql> show load order by createtime desc limit 1\\G;
*************************** 1. row ***************************
         JobId: 41326624
         Label: broker_load_2022_03_23
         State: FINISHED
      Progress: ETL:100%; LOAD:100%
          Type: BROKER
       EtlInfo: unselected.rows=0; dpp.abnorm.ALL=0; dpp.norm.ALL=27
      TaskInfo: cluster:N/A; timeout(s):1200; max_filter_ratio:0.1
      ErrorMsg: NULL
    CreateTime: 2022-04-01 18:59:06
  EtlStartTime: 2022-04-01 18:59:11
 EtlFinishTime: 2022-04-01 18:59:11
 LoadStartTime: 2022-04-01 18:59:11
LoadFinishTime: 2022-04-01 18:59:11
           URL: NULL
    JobDetails: {"Unfinished backends":{"5072bde59b74b65-8d2c0ee5b029adc0":[]},"Sca
1 row in set (0.01 sec)
```

取消导入

当 Broker load 作业状态不为 CANCELLED 或 FINISHED 时，可以被用户手动取消。取消时需要指定待取消导入任务的 Label 。取消导入命令语法可执行 CANCEL LOAD 查看。
例如：撤销数据库 demo 上， label 为 broker_load_2022_03_23 的导入作业：

```
CANCEL LOAD FROM demo WHERE LABEL = "broker_load_2022_03_23";
```

相关系统配置

Broker 参数

Broker load 需要借助 Broker 进程访问远端存储，不同的 Broker 需要提供不同的参数，具体请参阅 Broker 。

FE 配置

下面几个配置属于 Broker load 的系统级别配置，也就是作用于所有 Broker load 导入任务的配置。主要通过修改 `fe.conf` 来调整配置值。
min_bytes_per_broker_scanner/max_bytes_per_broker_scanner/max_broker_concurrency
前两个配置限制了单个 BE 处理的数据量的最小和最大值。第三个配置限制了一个作业的最大的导入并发数。最小处理的数据量，最大并发数，源文件的大小和当前集群 BE 的个数 **共同决定了本次导入的并发数**。

```
本次导入并发数 = Math.min(源文件大小/最小处理量，最大并发数，当前BE节点个数)
本次导入单个BE的处理量 = 源文件大小/本次导入的并发数
```

通常一个导入作业支持的最大数据量为 `max_bytes_per_broker_scanner * BE 节点数` 。如果需要导入更大数据量，则需要适当调整 `max_bytes_per_broker_scanner` 参数的大小。

默认配置：

```
参数名：min_bytes_per_broker_scanner， 默认 64MB，单位bytes。
参数名：max_broker_concurrency， 默认 10。
参数名：max_bytes_per_broker_scanner，默认 3G，单位bytes。
```

最佳实践

应用场景

使用 Broker load 最适合的场景就是原始数据在文件系统（HDFS，BOS，AFS）中的场景。其次，由于 Broker load 是单次导入中唯一的一种异步导入的方式，所以如果用户在导入大文件中，需要使用异步接入，也可以考虑使用 Broker load。

数据量

这里仅讨论单个 BE 的情况，如果用户集群有多个 BE 则下面标题中的数据量应该乘以 BE 个数来计算。例如：如果用户有3个 BE，则 3G 以下（包含）则应该乘以 3，也就是 9G 以下（包含）。

3G 以下（包含）：用户可以直接提交 Broker load 创建导入请求。

3G 以上：由于单个导入 BE 最大的处理量为 3G，超过 3G 的待导入文件就需要通过调整 Broker load 的导入参数来实现大文件的导入。

1.1 根据当前 BE 的个数和原始文件的大小修改单个 BE 的最大扫描量和最大并发数。

```
修改 fe.conf 中配置
max_broker_concurrency = BE 个数
当前导入任务单个 BE 处理的数据量 = 原始文件大小 / max_broker_concurrency
max_bytes_per_broker_scanner >= 当前导入任务单个 BE 处理的数据量

例如一个 100G 的文件，集群的 BE 个数为 10 个
max_broker_concurrency = 10
max_bytes_per_broker_scanner >= 10G = 100G / 10
```

修改后，所有的 BE 会并发的处理导入任务，每个 BE 处理原始文件的一部分。

注意

上述两个 FE 中的配置均为系统配置，也就是说其修改是作用于所有的 Broker load 任务的。

1.2 在创建导入的时候自定义当前导入任务的 timeout 时间。

```
当前导入任务单个 BE 处理的数据量 / 用户 Doris 集群最慢导入速度(MB/s) >= 当前导入任务的 timeout

例如一个 100G 的文件，集群的 BE 个数为 10个
timeout >= 1000s = 10G / 10M/s
```

1.3 当用户发现第二步计算出的 timeout 时间超过系统默认的导入最大超时时间4小时。
这时候不推荐用户将导入最大超时时间直接改大来解决问题。单个导入时间如果超过默认的导入最大超时时间4小时，最好是通过切分待导入文件并且分多次导入来解决问题。主要原因是：单次导入超过4小时的话，导入失败后重试的时间成本很高。
可以通过如下公式计算出 Doris 集群期望最大导入文件数据量：

```
期望最大导入文件数据量 = 14400s * 10M/s * BE 个数
例如：集群的 BE 个数为 10个
期望最大导入文件数据量 = 14400s * 10M/s * 10 = 1440000M ≈ 1440G

注意：一般用户的环境可能达不到 10M/s 的速度，所以建议超过 500G 的文件都进行文件切分，再导入。
```

作业调度

系统会限制一个集群内正在运行的 Broker load 作业数量，以防止同时运行过多的 Load 作业。
首先， FE 的配置参数： `desired_max_waiting_jobs` 会限制一个集群内，未开始或正在运行（作业状态为 PENDING 或 LOADING）的 Broker load 作业数量。默认为 100。如果超过这个阈值，新提交的作业将会被直接拒绝。
一个 Broker load 作业会被分为 pending task 和 loading task 阶段。其中 pending task 负责获取导入文件的信息，而 loading task 会发送给BE执行具体的导入任务。
FE 的配置参数 `async_pending_load_task_pool_size` 用于限制同时运行的 pending task 的任务数量。也相当于控制了实际正在运行的导入任务数量。该参数默认为 10。也就是说，假设用户提交了100个Load作业，同时只会有10个作业会进入 LOADING 状态开始执行，而其他作业处于 PENDING 等待状态。
FE 的配置参数 `async_loading_load_task_pool_size` 用于限制同时运行的 loading task 的任务数量。一个 Broker load 作业会有 1 个 pending task 和多个 loading task （等于 LOAD 语句中 DATA INFILE 子句的个数）。所以 `async_loading_load_task_pool_size` 应该大于等于 `async_pending_load_task_pool_size` 。

性能分析

可以在提交 LOAD 作业前，先执行 `set enable_profile=true` 打开会话变量。然后提交导入作业。待导入作业完成后，可以在 FE 的 web 页面的 `Queris` 标签中查看到导入作业的 Profile。可以查看 SHOW LOAD PROFILE 帮助文档，获取更多使用帮助信息。
这个 Profile 可以帮助分析导入作业的运行状态。当前只有作业成功执行后，才能查看 Profile。

常见问题

导入报错：`Scan bytes per broker scanner exceed limit:xxx`。

请参照文档中最佳实践部分，修改 FE 配置项 `max_bytes_per_broker_scanner` 和 `max_broker_concurrency`。

导入报错：`failed to send batch` 或 `TabletWriter add batch with unknown id`。

适当修改 `query_timeout` 和 `streaming_load_rpc_max_alive_time_sec`。

streaming_load_rpc_max_alive_time_sec：在导入过程中，Doris 会为每一个 Tablet 开启一个 Writer，用于接收数据并写入。这个参数指定了 Writer 的等待超时时间。如果在这个时间内，Writer 没有收到任何数据，则 Writer 会被自动销毁。当系统处理速度较慢时，Writer 可能长时间接收不到下一批数据，导致导入报错：`TabletWriter add batch with unknown id`。此时可适当增大这个配置。默认为 600 秒。

导入报错：`LOAD_RUN_FAIL; msg:Invalid Column Name:xxx`。

如果是 PARQUET 或者 ORC 格式的数据，需要在文件头的列名与 Doris 表中的列名一致，如：

```
(tmp_c1,tmp_c2)
SET
(
  id=tmp_c2,
  name=tmp_c1
)
```

代表获取在 parquet 或 orc 中以(tmp_c1, tmp_c2)为列名的列，映射到 doris 表中的(id, name)列。如果没有设置 set，则以 column 中的列作为映射。

说明

如果使用某些 hive 版本直接生成的 orc 文件，orc 文件中的表头并非 hive meta 数据，而是（_col0, _col1, _col2, ...）, 可能导致 Invalid Column Name 错误，需要使用 set 进行映射。

S3 Load（对象存储 COS）

最近更新时间：2024-06-27 10:56:17

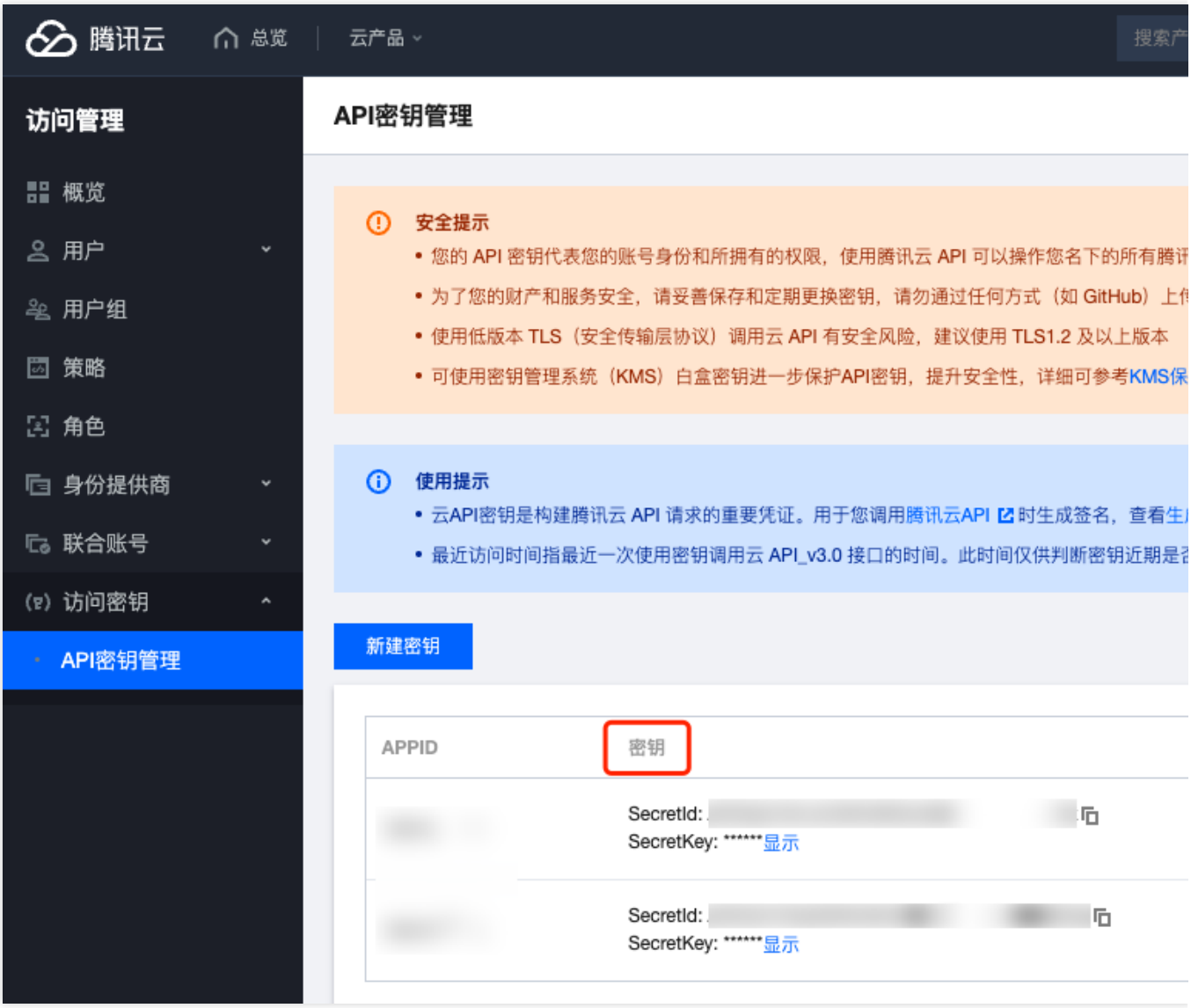
Doris 可通过 S3 协议直接从支持 S3 协议的在线存储系统导入数据。
本文档主要介绍如何导入腾讯云对象存储（兼容 S3 协议） 中存储的数据。也支持导入其他支持 S3 协议的对象存储系统导入，如 AWS S3 、百度云的 BOS 和阿里云的 OSS 等。

适用场景

源数据在支持 S3 协议的存储系统中，如 COS、S3、BOS、OSS 等。
数据量在几十到百 GB 级别。

准备工作

1. 准备 AWS_ACCESS_KEY 和 AWS_SECRET_KEY。
首先需要找到或者添加腾讯云的访问密钥。路径是：在腾讯云搜索访问密钥，使用已有密钥或单击**新建密钥**。然后获取其中的 SecretId，SecretKey，SecretId 为 AWS_ACCESS_KEY，SecretKey 为 AWS_SECRET_KEY，如下图所示：



2. 准备 REGION 和 ENDPOINT。

REGION 可以在创建桶的时候选择也可以在桶列表中查看到，与存储桶所在地域相关，如 ap-beijing，ap-guangzhou。ENDPOINT 的格式就是 `https://cos.<REGION>.myqcloud.com` 。其他云存储系统可以从相应的文档中找到与 S3 兼容的相关信息。

开始导入

导入方式和 Broker Load（HDFS 数据） 基本相同，只需要将 `WITH BROKER broker_name ()` 语句替换成如下部分：

```
    WITH S3
    (
        "AWS_ENDPOINT" = "http://cos.<REGION>.myqcloud.com",
        "AWS_ACCESS_KEY" = "AWS_ACCESS_KEY",
```

```
        "AWS_SECRET_KEY"="AWS_SECRET_KEY",
        "AWS_REGION" = "<REGION>"
    )
```

完整示例如下：

```
    LOAD LABEL example_db.exmpale_label_1
    (
        DATA INFILE("s3://your_bucket_name/your_path/your_file.txt")
        INTO TABLE load_test
        COLUMNS TERMINATED BY ","
    )
    WITH S3
    (
        "AWS_ENDPOINT" = "http://cos.<REGION>.myqcloud.com",
        "AWS_ACCESS_KEY" = "AWS_ACCESS_KEY",
        "AWS_SECRET_KEY"="AWS_SECRET_KEY",
        "AWS_REGION" = "<REGION>"
    )
    PROPERTIES
    (
        "timeout" = "3600"
    );
```

常见问题

S3 SDK 默认使用 virtual-hosted style 方式。但某些对象存储系统可能没开启或没支持 virtual-hosted style 方式的访问，此时我们可以添加 `use_path_style` 参数来强制使用 path style 方式：

```
   WITH S3
   (
        "AWS_ENDPOINT" = "http://cos.<REGION>.myqcloud.com",
        "AWS_ACCESS_KEY" = "AWS_ACCESS_KEY",
        "AWS_SECRET_KEY"="AWS_SECRET_KEY",
        "AWS_REGION" = "<REGION>",
        "use_path_style" = "true"
   )
```

Spark load

最近更新时间：2024-06-27 10:56:31

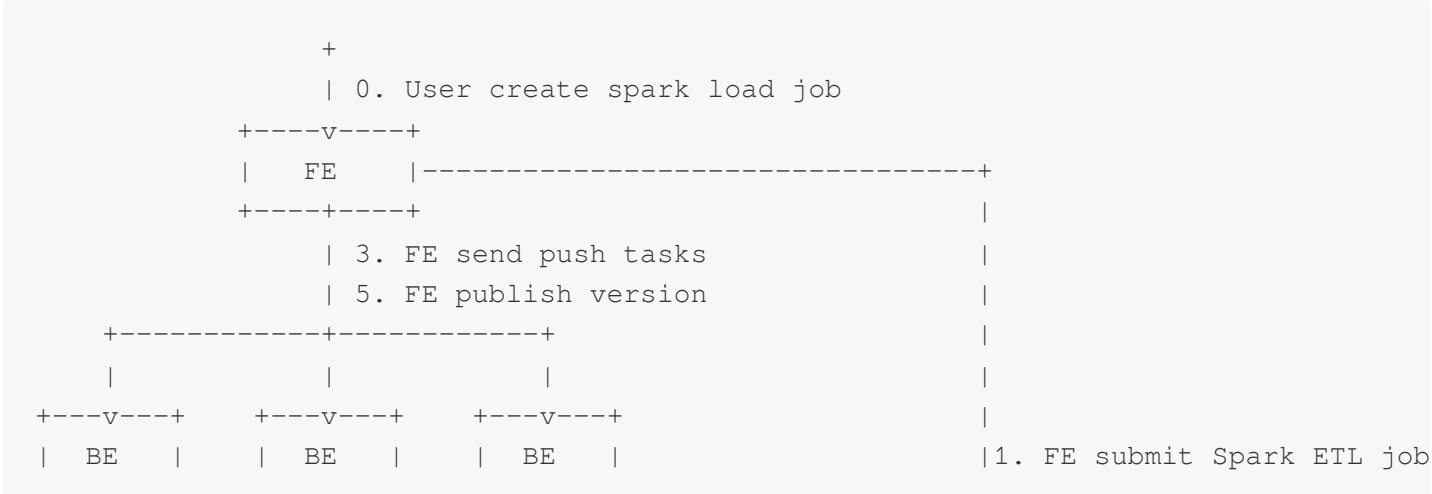
Spark Load 通过外部的 Spark 资源实现对导入数据的预处理，提高 Doris 大数据量的导入性能并且节省 Doris 集群的计算资源。主要用于初次迁移，大数据量导入 Doris 的场景。
Spark Load 是利用了 Spark 集群的资源对要导入的数据进行了排序，Doris BE 直接写文件，这样能大大降低 Doris 集群的资源使用，对于历史海量数据迁移降低 Doris 集群资源使用及负载有很好的效果。
如果用户在没有 Spark 集群这种资源的情况下，又想方便、快速的完成外部存储历史数据的迁移，可以使用 Broker Load（HDFS 数据） 。相对 Spark Load 导入，Broker Load 对 Doris 集群的资源占用会更高。
Spark Load 是一种异步导入方式，用户需要通过 MySQL 协议创建 Spark 类型导入任务，并通过 SHOW LOAD 查看导入结果。

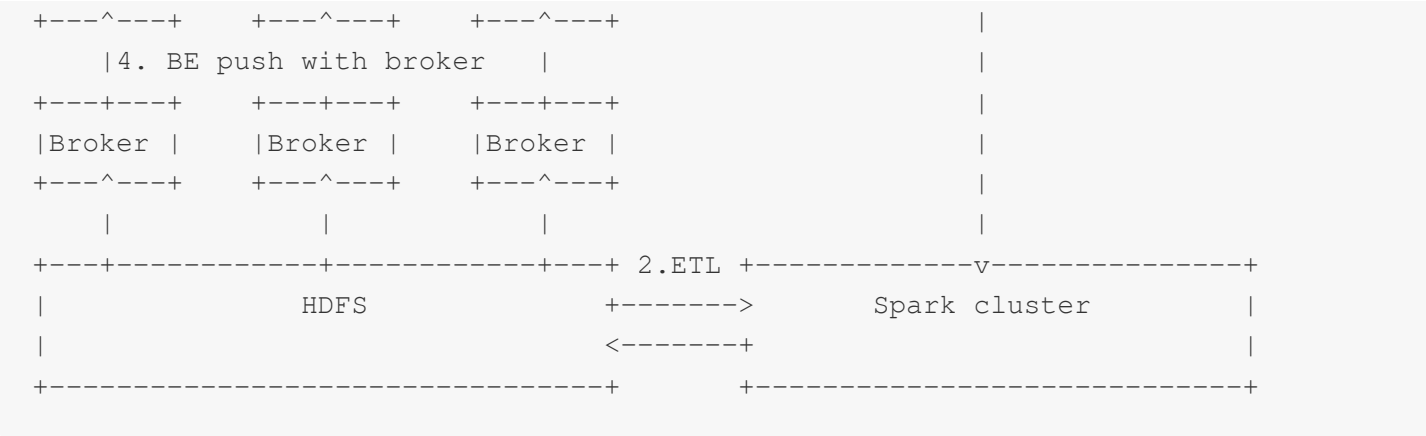
适用场景

源数据在 Spark 可以访问的存储系统中，如 HDFS。
数据量在 几十 GB 到 TB 级别。

基本原理

用户通过 MySQL 客户端提交 Spark 类型导入任务，FE 记录元数据并返回用户提交成功。
Spark Load 任务的执行主要分为以下 5 个阶段。
1. FE 调度提交 ETL 任务到 Spark 集群执行。
2. Spark 集群执行 ETL 完成对导入数据的预处理，包括全局字典构建（ Bitmap 类型）、分区、排序、聚合等。
3. ETL 任务完成后，FE 获取预处理过的每个分片的数据路径，并调度相关的 BE 执行 Push 任务。
4. BE 通过 Broker 读取数据，转化为 Doris 底层存储格式。
5. FE 调度生效版本，完成导入任务。





配置环境

使用前需要配置 FE Follower 节点的环境。只需在 Follwer 节点配置，Observer 节点不需要。

说明：

配置环境部分操作需要腾讯云技术人员支持，请在使用时 提交工单 与我们联系。

打通 Doris 到 EMR 集群的网络

最好保证同 VPC 同子网同安全组，这样网络联通问题最少。如果不满足，需要联系腾讯云网络运维共同解决。最终需通过 ping / telnet 确认连通性。

配置 Hadoop 客户端

FE 底层通过执行 Yarn 命令去获取正在运行的 Application 的状态以及杀死 Application，因此需要为 FE 配置 Yarn 客户端。建议使用 2.5.2 或以上的版本。

需要客户提供要连接的 Yarn 集群的 Hadoop 客户端，如此可保证和 EMR 集群版本一致，避免出现兼容性问题，且其中包含必需的 hdfs-site.xml，yarn-site.xm，core-site.xml 等配置文件。

获取到 Hadoop 客户端后，让其放置到 /usr/local/service下，改名为 Hadoop 或 建一个指向此目录的名为 Hadoop 的软链。

修改路径/usr/local/service/hadoop/etc/hadoop的权限，保证 Doris FE 启动用户有写权限：

```
chown -R doris:doris /usr/local/service/hadoop/etc/hadoop
```

说明：

如果要使用的 Spark 是在腾讯云 EMR，请 提交工单 联系 EMR 运维，将 Follower FE 的 IP 加入体外客户端白名单。

使用 Doris FE 启动用户（1.2版本之前为 root，1.2及之后为 doris）执行 `hdfs dfs -ls /` 命令，验证 Hadoop 客户端是否安装成功。

配置 Spark 客户端

FE 底层通过执行 spark-submit 的命令去提交 Spark 任务，因此需要为 FE 配置 Spark 客户端，建议使用 2.4.5 或以上的 Spark2 官方版本。

最好让客户提供要使用的yarn集群的 spark 客户端，这样版本一致，且其中包含必需的 spark-defaults.conf, hive-site.xml 等配置文件。
获取到 Spark 客户端后，让其放置到/usr/local/service下，改名为 spark 或 建一个名为 spark 的软链指向其。
然后执行以下命令：

```
cd /usr/local/service/spark/jars/
zip spark_jars.zip *.jar
```

修改 spark 的日志级别(log4j.rootLogger)，改为 INFO。

```
  vim /usr/local/service/spark/conf/log4j.properties
```

使用 doris FE 启动用户（1.2版本之前为 root，1.2及之后为 doris）执行以下命令，验证 Hadoop 及 Spark 客户端是否安装成功。

```
spark-submit --queue default --master yarn --deploy-mode cluster --class org.apache
```

配置 Doris

配置 spark load 任务的目录，每个任务生成一个日志文件，Doris 通过监控日志文件获取任务 ID 和状态。

```
mkdir -p /usr/local/service/doris/log/spark_launcher_log;
chmod 777 /usr/local/service/doris/log/spark_launcher_log
```

通过管控，在 FE.conf 中添加配置：

```
spark_home_default_dir=/usr/local/service/spark
spark_resource_path=/usr/local/service/spark/jars/spark_jars.zip
yarn_client_path=/usr/local/service/hadoop/bin/yarn
yarn_config_dir=/usr/local/service/hadoop/etc/hadoop
spark_dpp_version=1.2-SNAPSHOT
```

配置完毕后重启 Master FE。
说明：
添加配置中，若需要腾讯云技术人员支持，请在使用时 提交工单 与我们联系。

使用 Spark load

创建 Spark Resource

一个典型的创建语句模板如下：

```
CREATE EXTERNAL RESOURCE spark_resource_xxx
PROPERTIES
```

```
(
  "type" = "spark",
  "spark.master" = "yarn",
  "spark.submit.deployMode" = "cluster",
  "spark.yarn.queue" = "<xxx_queue>",
  "spark.hadoop.yarn.resourcemanager.ha.enabled" = "true",
  "spark.hadoop.yarn.resourcemanager.ha.rm-ids" = "rm1,rm2",
  "spark.hadoop.yarn.resourcemanager.address.rm1" = "<rm1_host>:<rm1_port>",
  "spark.hadoop.yarn.resourcemanager.address.rm2" = "<rm2_host>:<rm2_port>",
  "spark.hadoop.fs.defaultFS" = "hdfs://<hdfs_defaultFS>",
  "spark.hadoop.dfs.nameservices" = "<hdfs_defaultFS>",
  "spark.hadoop.dfs.ha.namenodes.<hdfs_defaultFS>" = "nn1,nn2",
  "spark.hadoop.dfs.namenode.rpc-address.<hdfs_defaultFS>.nn1" = "<nn1_host>:<nn1_p
  "spark.hadoop.dfs.namenode.rpc-address.<hdfs_defaultFS>.nn2" = "<nn2_host>:<nn2_p
  "spark.hadoop.dfs.client.failover.proxy.provider" = "org.apache.hadoop.hdfs.serve
  "working_dir" = "hdfs://<hdfs_defaultFS>/doris/spark_load",
  "broker" = "<doris_broker_name>",
  "broker.username" = "hadoop",
  "broker.password" = "",
  "broker.dfs.nameservices" = "<hdfs_defaultFS>",
  "broker.dfs.ha.namenodes.<hdfs_defaultFS>" = "nn1, nn2",
  "broker.dfs.namenode.rpc-address.HDFS4001273.nn1" = "<nn1_host>:<nn1_port>",
  "broker.dfs.namenode.rpc-address.HDFS4001273.nn2" = "<nn2_host>:<nn2_port>",
  "broker.dfs.client.failover.proxy.provider" = "org.apache.hadoop.hdfs.server.name
);
```

上述配置适用于绝大多数 EMR 集群，即开启了 RM HA 和 HDFS HA 的集群。Doris 也支持使用非 HA 集群、或开启了 Kerberos 认证的集群，具体的配置方法参见 Spark Load。

说明：

腾讯云 EMR 的常见端口：<rm1_port>: 5000、<nn1_port>: 4007。

授权对 Spark Resource 的使用

普通账户无法创建 resource，只能看到自己有 USAGE_PRIV 使用权限的资源。因此如果有普通账户需要使用某资源，需要给他授权。授予权限后也可撤销。具体命令如下：

```
-- 授予spark0资源的使用权限给用户user0
GRANT USAGE_PRIV ON RESOURCE "spark_resource_xxx" TO "user0"@"%";

-- 授予spark0资源的使用权限给角色role0
GRANT USAGE_PRIV ON RESOURCE "spark_resource_xxx" TO ROLE "role0";

-- 授予所有资源的使用权限给用户user0
GRANT USAGE_PRIV ON RESOURCE * TO "user0"@"%";

-- 授予所有资源的使用权限给角色role0
```

```
GRANT USAGE_PRIV ON RESOURCE * TO ROLE "role0";

-- 撤销用户user0的spark0资源使用权限
REVOKE USAGE_PRIV ON RESOURCE "spark_resource_xxx" FROM "user0"@"%";
```

执行 Spark load 任务

```
LOAD LABEL test_label_01    --label
(
 DATA INFILE ("hdfs://HDFS4001234/warehouse/ods.db/user_events/ds=2023-04-15/*"  --
 INTO TABLE user_events   --doris table
 FORMAT AS "parquet"      --data format
 ( event_time, user_id, op_code)  --columns in file
 COLUMNS FROM PATH AS ( `ds` )     --partition column
 SET
 (                                  --column mapping
  ds = ds,
  event_time = event_time,
  user_id = user_id,
  op_code = op_code
 )
)
WITH RESOURCE 'spark_resource_xxx'
(                                          --spark job params
 "spark.executor.memory" = "4g",
 "spark.default.parallelism" = "400",
 "spark.executor.cores" = '5',
 "spark.executor.instances" = '10'
)
PROPERTIES
(                                              --doris load task params
 "timeout" = "259200"
);
```

需要根据实际情况修改 label，file path，doris table，分区列、列映射关系、spark 任务参数等的值。一个 label 如果任务成功则不可重复使用。

管理任务

需要先进入到导入目标表所在的库：`use xxx_db;` 。

查看任务

`show load where label='test_label_01';` 。

重点关注结果中的 STATE 和 PROGRESS 列。作业的状态转换路径为：

```
- State
    导入任务当前所处的阶段。任务提交之后状态为 PENDING，提交 Spark ETL 之后状态变为 ETL，ETL 
    导入任务的最终阶段有两个：CANCELLED 和 FINISHED，当 Load Job 处于这两个阶段时导入完成。其

- Progress
    导入任务的进度描述。分为两种进度：ETL 和 LOAD，对应了导入流程的两个阶段 ETL 和 LOADING。
    LOAD 的进度范围为：0~100%。
     `LOAD 进度 = 当前已完成所有 Replica 导入的 Tablet 个数 / 本次导入任务的总 Tablet 个数
    **如果所有导入表均完成导入，此时 LOAD 的进度为 99%** 导入进入到最后生效阶段，整个导入完成后，
    导入进度并不是线性的。所以如果一段时间内进度没有变化，并不代表导入没有在执行。
```

取消任务

如果任务未结束，可执行命令将其结束：`cancel load where label='test_label_01';` 。

问题排查

一个 spark load 任务失败可能是 doris 的问题，也可能是外部使用的 yarn 集群的问题。因此两边都需要排查。

先从 Doris 侧排查：

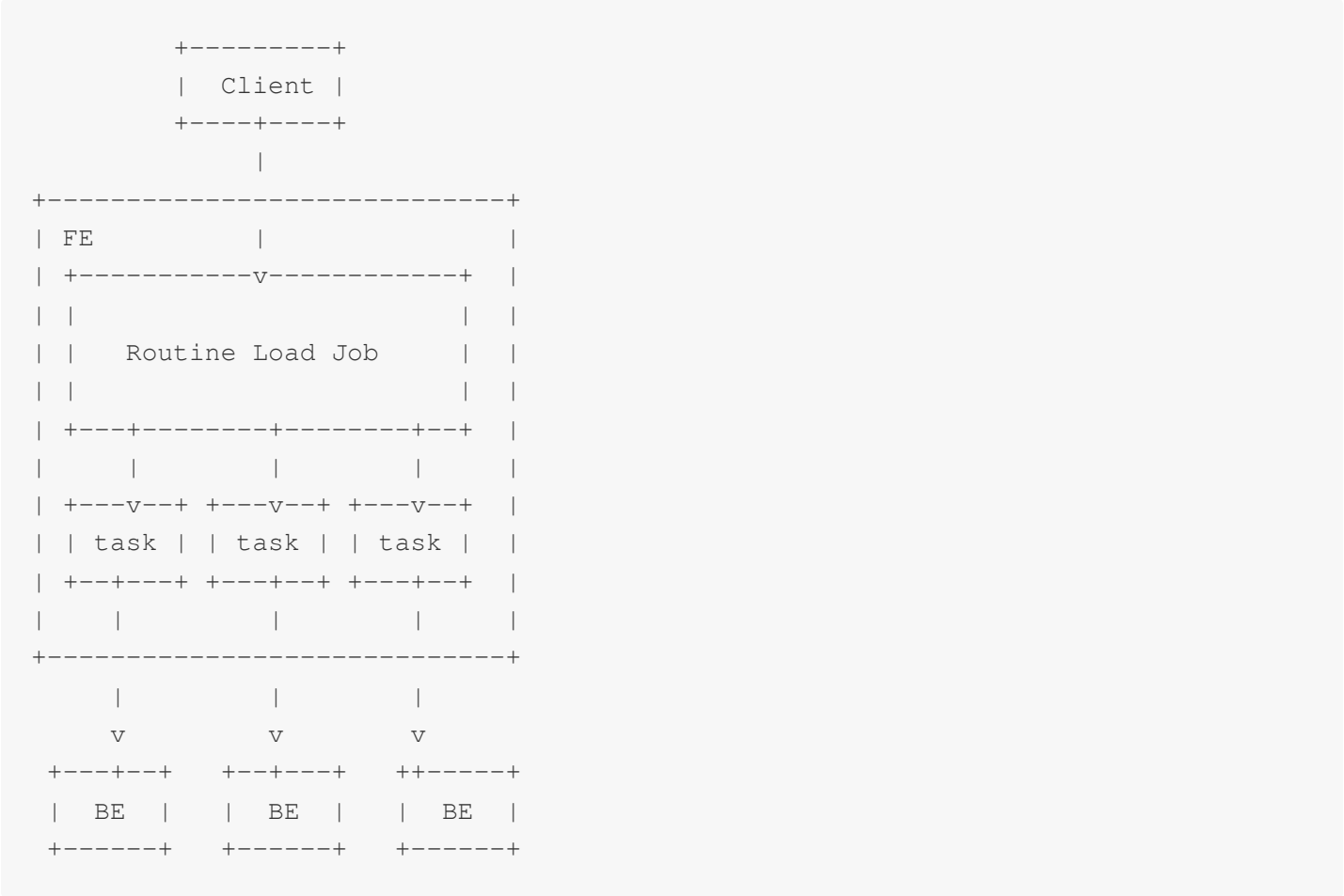
1. 使用 `show load` 命令根据 label 查看load 任务信息。
2. 每个任务会在目录/usr/local/service/doris/log/spark_launcher_log下生成一个日志文件，默认保留3天。进入此目录后执行 `ls *<load_task_label>*` 进行模糊搜索。
3. 如果找不到上述任务日志文件或文件为空，需要查看 doris fe 日志 fe.log / fe.warn.log：`cd /data/cdw/doris/fe/log/` 。
4. 如果日志显示 Spark Job 执行失败，则需让用户查看 Spark Job 日志确认具体报错。

Routine Load（Kafka 数据）

最近更新时间：2024-06-27 10:56:49

例行导入（Routine Load）功能，支持用户提交一个常驻的导入任务，通过不断的从指定的数据源读取数据，将数据导入到 Doris 中，例如自动持续从Kafka导入数据。
本文档主要介绍该功能的实现原理、使用方式以及最佳实践。

基本原理



如上图，Client 向 FE 提交一个 Routine Load 作业。
1. FE 通过 JobScheduler 将一个导入作业拆分成若干个 Task。每个 Task 负责导入指定的一部分数据。Task 被 TaskScheduler 分配到指定的 BE 上执行。
2. 在 BE 上，一个 Task 被视为一个普通的导入任务，通过 Stream Load 的导入机制进行导入。导入完成后向 FE 汇报。
3. FE 中的 JobScheduler 根据汇报结果，继续生成后续新的 Task，或者对失败的 Task 进行重试。
4. 整个 Routine Load 作业通过不断的产生新的 Task，来完成数据不间断的导入。

Kafka 例行导入

当前我们仅支持从 Kafka 进行例行导入。该部分会详细介绍 Kafka 例行导入使用方式和最佳实践。

使用限制

1. 支持无认证的 Kafka 访问，以及通过 SSL 方式认证的 Kafka 集群。
2. 支持的消息格式为 csv， json 文本格式。csv 每一个 message 为一行，且行尾**不包含**换行符。
3. 默认支持 Kafka 0.10.0.0(含) 以上版本。如果要使用 Kafka 0.10.0.0 以下版本 (0.9.0, 0.8.2, 0.8.1, 0.8.0)，需要修改 BE 的配置，将 kafka_broker_version_fallback 的值设置为要兼容的旧版本，或者在创建routine load的时候直接设置 property.broker.version.fallback的值为要兼容的旧版本，使用旧版本的代价是routine load 的部分新特性可能无法使用，如根据时间设置 kafka 分区的 offset。

创建任务

创建例行导入任务的详细语法可以连接到 Doris 后，查看 CREATE ROUTINE LOAD 命令手册，或者执行 `HELP ROUTINE LOAD;` 查看语法帮助。

下面我们以几个例子说明如何创建 Routine Load 任务：

1. 为 example_db 的 example_tbl 创建一个名为 test1 的 Kafka 例行导入任务。指定列分隔符和 group.id 和 client.id，并且自动默认消费所有分区，且从有数据的位置（OFFSET_BEGINNING）开始订阅。

```
CREATE ROUTINE LOAD example_db.test1 ON example_tbl
     COLUMNS TERMINATED BY ",",
     COLUMNS(k1, k2, k3, v1, v2, v3 = k1 * 100)
     PROPERTIES
     (
         "desired_concurrent_number"="3",
         "max_batch_interval" = "20",
         "max_batch_rows" = "300000",
         "max_batch_size" = "209715200",
         "strict_mode" = "false"
     )
     FROM KAFKA
     (
         "kafka_broker_list" = "broker1:9092,broker2:9092,broker3:9092",
         "kafka_topic" = "my_topic",
         "property.group.id" = "xxx",
         "property.client.id" = "xxx",
         "property.kafka_default_offsets" = "OFFSET_BEGINNING"
     );
```

2. 以 **严格模式** 为 example_db 的 example_tbl 创建一个名为 test1 的 Kafka 例行导入任务：

```
CREATE ROUTINE LOAD example_db.test1 ON example_tbl
     COLUMNS(k1, k2, k3, v1, v2, v3 = k1 * 100),
```

```
    WHERE k1 > 100 and k2 like "%doris%"
    PROPERTIES
    (
        "desired_concurrent_number"="3",
        "max_batch_interval" = "20",
        "max_batch_rows" = "300000",
        "max_batch_size" = "209715200",
        "strict_mode" = "true"
    )
    FROM KAFKA
    (
        "kafka_broker_list" = "broker1:9092,broker2:9092,broker3:9092",
        "kafka_topic" = "my_topic",
        "kafka_partitions" = "0,1,2,3",
        "kafka_offsets" = "101,0,0,200"
    );
```

3. 导入 Json 格式数据使用示例：

Routine Load 导入的 json 格式仅支持以下两种：

第一种只有一条记录，且为 json 对象：

```
{"category":"a9jadhx","author":"test","price":895}
```

第二种为 json 数组，数组中可含多条记录：

```
[
  {"category":"11",
          "title":"SayingsoftheCentury",
          "price":895,
          "timestamp":1589191587
  },
  {
          "category":"22",
          "author":"2avc",
          "price":895,
          "timestamp":1589191487
  },
  {
          "category":"33",
          "author":"3avc",
          "title":"SayingsoftheCentury",
          "timestamp":1589191387
  }
]
```

创建待导入的 Doris 数据表：

```
CREATE TABLE `example_tbl` (
 `category` varchar(24) NULL COMMENT "",
 `author` varchar(24) NULL COMMENT "",
 `timestamp` bigint(20) NULL COMMENT "",
 `dt` int(11) NULL COMMENT "",
 `price` double REPLACE
) ENGINE=OLAP
AGGREGATE KEY(`category`,`author`,`timestamp`,`dt`)
COMMENT "OLAP"
PARTITION BY RANGE(`dt`)
(
 PARTITION p0 VALUES [("-2147483648"), ("20200509")),
PARTITION p20200509 VALUES [("20200509"), ("20200510")),
PARTITION p20200510 VALUES [("20200510"), ("20200511")),
PARTITION p20200511 VALUES [("20200511"), ("20200512"))
)
DISTRIBUTED BY HASH(`category`,`author`,`timestamp`) BUCKETS 4
PROPERTIES (
  "replication_num" = "1"
);
```

以简单模式导入 json 数据：

```
CREATE ROUTINE LOAD example_db.test_json_label_1 ON table1
COLUMNS(category,price,author)
PROPERTIES
(
"desired_concurrent_number"="3",
"max_batch_interval" = "20",
"max_batch_rows" = "300000",
"max_batch_size" = "209715200",
"strict_mode" = "false",
"format" = "json"
)
FROM KAFKA
(
"kafka_broker_list" = "broker1:9092,broker2:9092,broker3:9092",
"kafka_topic" = "my_topic",
"kafka_partitions" = "0,1,2",
"kafka_offsets" = "0,0,0"
);
```

精准导入 json 格式数据：

```
CREATE ROUTINE LOAD example_db.test1 ON example_tbl
COLUMNS(category, author, price, timestamp, dt=from_unixtime(timestamp, '%Y%m%d'))
PROPERTIES
```

```
(
  "desired_concurrent_number"="3",
  "max_batch_interval" = "20",
  "max_batch_rows" = "300000",
  "max_batch_size" = "209715200",
  "strict_mode" = "false",
  "format" = "json",
  "jsonpaths" = "[\\"$.category\\",\\"$.author\\",\\"$.price\\",\\"$.timestamp\\"]"
  "strip_outer_array" = "true"
)
FROM KAFKA
(
  "kafka_broker_list" = "broker1:9092,broker2:9092,broker3:9092",
  "kafka_topic" = "my_topic",
  "kafka_partitions" = "0,1,2",
  "kafka_offsets" = "0,0,0"
);
```

strict mode 与 source data 的导入关系

这里以列类型为 TinyInt 来举例。

当表中的列允许导入空值时：

source data	source data example	string to int	strict_mode	result
空值	\\N	N/A	true or false	NULL
not null	aaa or 2000	NULL	true	invalid data(filtered)
not null	aaa	NULL	false	NULL
not null	1	1	true or false	correct data

这里以列类型为 Decimal(1,0) 举例。

当表中的列允许导入空值时：

source data	source data example	string to int	strict_mode	result
空值	\\N	N/A	true or false	NULL
not null	aaa	NULL	true	invalid data(filtered)
not null	aaa	NULL	false	NULL
not null	1 or 10	1	true or false	correct data

注意

10 虽然是一个超过范围的值，但是因为其类型符合 decimal 的要求，所以 strict mode 对其不产生影响。10 最后会在其他 ETL 处理流程中被过滤。但不会被 strict mode 过滤。

访问 SSL 认证的 Kafka 集群

访问 SSL 认证的 Kafka 集群需要用户提供用于认证 Kafka Broker 公钥的证书文件（ca.pem）。如果 Kafka 集群同时开启了客户端认证，则还需提供客户端的公钥（client.pem）、密钥文件（client.key），以及密钥密码。这里所需的文件需要先通过 `CREAE FILE` 命令上传到 Doris 中，**并且 catalog 名称为 `kafka` **。 `CREATE FILE` 命令的具体帮助可以参见 `HELP CREATE FILE;` 。这里给出示例：

1. 上传文件

```
CREATE FILE "ca.pem" PROPERTIES("url" = "https://example_url/kafka-key/ca.pem", "ca
CREATE FILE "client.key" PROPERTIES("url" = "https://example_urlkafka-key/client.ke
CREATE FILE "client.pem" PROPERTIES("url" = "https://example_url/kafka-key/client.p
```

2. 创建例行导入作业

```
CREATE ROUTINE LOAD db1.job1 on tbl1
PROPERTIES
(
  "desired_concurrent_number"="1"
)
FROM KAFKA
(
  "kafka_broker_list"= "broker1:9091,broker2:9091",
  "kafka_topic" = "my_topic",
  "property.security.protocol" = "ssl",
  "property.ssl.ca.location" = "FILE:ca.pem",
  "property.ssl.certificate.location" = "FILE:client.pem",
  "property.ssl.key.location" = "FILE:client.key",
  "property.ssl.key.password" = "abcdefg"
);
```

说明

Doris 通过 Kafka 的 C++ API `librdkafka` 来访问 Kafka 集群。 `librdkafka` 所支持的参数可以参阅 Configuration properties。

查看作业状态

查看**作业**状态的具体命令和示例可以通过 `HELP SHOW ROUTINE LOAD;` 命令查看。

查看**任务**运行状态的具体命令和示例可以通过 `HELP SHOW ROUTINE LOAD TASK;` 命令查看。

只能查看当前正在运行中的任务，已结束和未开始的任务无法查看。

修改作业属性

用户可以修改已经创建的作业。具体说明可以通过 `HELP ALTER ROUTINE LOAD;` 命令查看或参阅 ALTER ROUTINE LOAD。

作业控制

用户可以通过 `STOP/PAUSE/RESUME` 三个命令来控制作业的停止，暂停和重启。可以通过 `HELP STOP ROUTINE LOAD;``HELP PAUSE ROUTINE LOAD;` 以及 `HELP RESUME ROUTINE LOAD;` 三个命令查看帮助和示例。

其他说明

1. 例行导入作业和 ALTER TABLE 操作的关系。
例行导入不会阻塞 SCHEMA CHANGE 和 ROLLUP 操作。但是注意如果 SCHEMA CHANGE 完成后，列映射关系无法匹配，则会导致作业的错误数据激增，最终导致作业暂停。建议通过在例行导入作业中显示指定列映射关系，以及通过增加 Nullable 列或带 Default 值的列来减少这类问题。
删除表的 Partition 可能会导致导入数据无法找到对应的 Partition，作业进入暂停。
2. 例行导入作业和其他导入作业的关系（LOAD，DELETE，INSERT）。
例行导入和其他 LOAD 作业以及 INSERT 操作没有冲突。
当执行 DELETE 操作时，对应表分区不能有任何正在执行的导入任务。所以在执行 DELETE 操作前，可能需要先暂停例行导入作业，并等待已下发的 task 全部完成后，才可以执行 DELETE。
3. 例行导入作业和 DROP DATABASE/TABLE 操作的关系。
当例行导入对应的 database 或 table 被删除后，作业会自动 CANCEL。
4. kafka 类型的例行导入作业和 kafka topic 的关系。
当用户在创建例行导入声明的 `kafka_topic` 在kafka集群中不存在时。
如果用户 kafka 集群的 broker 设置了 `auto.create.topics.enable = true`，则 `kafka_topic` 会先被自动创建，自动创建的 partition 个数是由**用户方的kafka集群**中的 broker 配置 `num.partitions` 决定的。例行作业会正常的不断读取该 topic 的数据。
如果用户 kafka 集群的 broker 设置了 `auto.create.topics.enable = false`, 则 topic 不会被自动创建，例行作业会在没有读取任何数据之前就被暂停，状态为 `PAUSED` 。
所以，如果用户希望当 kafka topic 不存在的时候，被例行作业自动创建的话，只需要将**用户方的 kafka 集群**中的 broker 设置 `auto.create.topics.enable = true` 即可。
5. 在网络隔离的环境中可能出现的问题 在有些环境中存在网段和域名解析的隔离措施，所以需要注意：
1. 创建 Routine load 任务中指定的 Broker list 必须能够被 Doris 服务访问。
2. Kafka 中如果配置了 `advertised.listeners`，`advertised.listeners` 中的地址必须能够被 Doris 服务访问。
3. 关于指定消费的 Partition 和 Offset。
Doris 支持指定 Partition 和 Offset 开始消费。新版中还支持了指定时间点进行消费的功能。这里说明下对应参数的配置关系。
有三个相关参数：
`kafka_partitions` ：指定待消费的 partition 列表，如："0, 1, 2, 3"。

`kafka_offsets` ：指定每个分区的起始offset，必须和 `kafka_partitions` 列表个数对应。如："1000, 1000, 2000, 2000"

`property.kafka_default_offset` ：指定分区默认的起始offset。

在创建导入作业时，这三个参数可以有以下组合：

组合	`kafka_partitions`	`kafka_offsets`	`property.kafka_default_offset`	行为
1	No	No	No	系统会 找topi 所有分 OFFS 开始消
2	No	No	Yes	系统会 找topi 所有分 defaul 指定的 始消费
3	Yes	No	No	系统会 分区的 OFFS 开始消
4	Yes	Yes	No	系统会 分区的 offset 消费
5	Yes	No	Yes	系统会 分区， offset 位置开

4. STOP 和 PAUSE 的区别。

FE 会自动定期清理 STOP 状态的 ROUTINE LOAD，而 PAUSE 状态的则可以再次被恢复启用。

相关参数

一些系统配置参数会影响例行导入的使用。

1. max_routine_load_task_concurrent_num
FE 配置项，默认为 5，可以运行时修改。该参数限制了一个例行导入作业最大的子任务并发数。建议维持默认值。设置过大，可能导致同时并发的任务数过多，占用集群资源。
2. max_routine_load_task_num_per_be
FE 配置项，默认为5，可以运行时修改。该参数限制了每个 BE 节点最多并发执行的子任务个数。建议维持默认值。如果设置过大，可能导致并发任务数过多，占用集群资源。
3. max_routine_load_job_num
FE 配置项，默认为100，可以运行时修改。该参数限制的例行导入作业的总数，包括 NEED_SCHEDULED，RUNNING，PAUSE 这些状态。超过后，不能在提交新的作业。
4. max_consumer_num_per_group
BE 配置项，默认为 3。该参数表示一个子任务中最多生成几个 consumer 进行数据消费。对于 Kafka 数据源，一个 consumer 可能消费一个或多个 kafka partition。假设一个任务需要消费 6 个 kafka partition，则会生成 3 个 consumer，每个 consumer 消费 2 个 partition。如果只有 2 个 partition，则只会生成 2 个 consumer，每个 consumer 消费 1 个 partition。
5. push_write_mbytes_per_sec
BE 配置项。默认为 10，即 10MB/s。该参数为导入通用参数，不限于例行导入作业。该参数限制了导入数据写入磁盘的速度。对于 SSD 等高性能存储设备，可以适当增加这个限速。
6. max_tolerable_backend_down_num FE 配置项，默认值是0。在满足某些条件下，Doris 可 PAUSED 的任务重新调度，即变成 RUNNING。该参数为0代表只有所有 BE 节点是 alive 状态才允许重新调度。
7. period_of_auto_resume_min FE 配置项，默认是5分钟。Doris 重新调度，只会在5分钟这个周期内，最多尝试3次。如果3次都失败则锁定当前任务，后续不在进行调度。但可通过人为干预，进行手动恢复。

最佳实践

这里以持续从kafka导入TPC-H数据集中的lineitem表为例子展示routine load的最佳实践

确定kafka msg格式和目标表结构

写入到 kafka msg 中的格式：

```
1992-01-02      3273383 3       2508983 508984  42      83658.12        0.06    0.0
1992-01-02      26696039        2       8782384 532409  50      73297.5 0.02    0.0
1992-01-02      47726080        1       5950048 950049  24      26346   0.02    0.0
1992-01-02      77918949        2       7276493 526501  23      33789.99        0.0
1992-01-02      87026306        4       9061545 811573  11      16566.99        0.0
1992-01-02      135925030       4       14097984        598013  30      59438.4 0.0
1992-01-02      189122402       2       13648194        398234  5       5707.55 0.1
1992-01-02      235359552       4       8148971 898996  29      58567.53        0.1
1992-01-02      298717351       3       4078182 78183   42      48719.16        0.0
1992-01-02      305288709       3       14997743        747786  33      60720   0.0
```

doris 中routine 目标表lineitem 的表结构，实际在doris中的表名称为：lineitem_rtload。

```
CREATE TABLE lineitem_rtload (
  l_shipdate date NOT NULL,
  l_orderkey bigint(20) NOT NULL,
  l_linenumber int(11) NOT NULL,
  l_partkey int(11) NOT NULL,
  l_suppkey int(11) NOT NULL,
  l_quantity decimalv3(15, 2) NOT NULL,
  l_extendedprice decimal(15, 2) NOT NULL,
  l_discount decimalv3(15, 2) NOT NULL,
  l_tax decimalv3(15, 2) NOT NULL,
  l_returnflag varchar(1) NOT NULL,
  l_linestatus varchar(1) NOT NULL,
  l_commitdate date NOT NULL,
  l_receiptdate date NOT NULL,
  l_shipinstruct varchar(25) NOT NULL,
  l_shipmode varchar(10) NOT NULL,
  l_comment varchar(44) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(l_shipdate, l_orderkey)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(l_orderkey) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 1",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
);
```

基于写入格式和目标表信息，创建 routine load 任务。

```
CREATE ROUTINE LOAD tpch_100_d.rtl_20230809 ON lineitem_rtload
COLUMNS TERMINATED BY "\\t"
PROPERTIES
(
    "desired_concurrent_number"="3",
    "max_batch_interval" = "20",
    "max_batch_rows" = "300000",
    "max_batch_size" = "209715200",
    "max_error_number" = "100",
    "strict_mode" = "true"
)
FROM KAFKA
(
    "kafka_broker_list" = "10.0.1.138:9092",
    "kafka_topic" = "doris_routine_load_test",
```

```
    "property.group.id" = "routine_test",
    "property.kafka_default_offsets" = "OFFSET_BEGINNING"
);
```

COLUMNS TERMINATED BY "\\t"

指定kafka msg中的字段分隔符，默认为 "\\t"

max_batch_interval/max_batch_rows/max_batch_size

这三个参数分别表示：

每个子任务最大执行时间，单位是秒。范围为 5 到 60。默认为10。

每个子任务最多读取的行数。必须大于等于200000。默认是200000。

每个子任务最多读取的字节数。单位是字节，范围是 100MB 到 1GB。默认是 100MB。

这三个参数，用于控制一个子任务的执行时间和处理量。当任意一个达到阈值，则任务结束。

max_error_number

采样窗口内，允许的最大错误行数。必须大于等于0。默认是 0，即不允许有错误行。

采样窗口为 max_batch_rows * 10。即如果在采样窗口内，错误行数大于 max_error_number，则会导致例行作业被暂停，需要人工介入检查数据质量问题。

被 where 条件过滤掉的行不算错误行。

strict_mode

是否开启严格模式，默认为关闭。如果开启后，非空原始数据的列类型变换如果结果为 NULL，则会被过滤。指定方式为：

"strict_mode" = "true"

strict mode 模式的意思是：对于导入过程中的列类型转换进行严格过滤。严格过滤的策略如下：

对于列类型转换来说，如果 strict mode 为true，则错误的数据将被 filter。这里的错误数据是指：原始数据并不为空值，在参与列类型转换后结果为空值的这一类数据。

对于导入的某列由函数变换生成时，strict mode 对其不产生影响。

对于导入的某列类型包含范围限制的，如果原始数据能正常通过类型转换，但无法通过范围限制的，strict mode 对其也不产生影响。例如：如果类型是 decimal(1,0)，原始数据为 10，则属于可以通过类型转换但不在列声明的范围内。这种数据 strict 对其不产生影响。

查看任务运行情况

创建 routine load 任务之后,可以通过show routine load命令查看运行状态的例行任务，如果在show routine load中没有找到对应的例行任务，则可能因为例行任务失败或者错误数过多被停止或者暂停，使用show all routine load查看所有状态的例行任务。

```
MySQL [tpch_100_d]> show routine load\\G;
*************************** 1. row ***************************
                  Id: 21619
                Name: rtl_20230809
          CreateTime: 2023-08-09 19:17:16
           PauseTime: NULL
```

```
              EndTime: NULL
               DbName: default_cluster:tpch_100_d
            TableName: lineitem_rtload
                State: RUNNING
       DataSourceType: KAFKA
       CurrentTaskNum: 3
        JobProperties: {"timezone":"Asia/Shanghai","send_batch_parallelism":"1","col
DataSourceProperties: {"topic":"doris_routine_load_test","currentKafkaPartitions":"
    CustomProperties: {"kafka_default_offsets":"OFFSET_BEGINNING","group.id":"ryanz
           Statistic: {"receivedBytes":568128,"runningTxns":[],"errorRows":0,"commi
            Progress: {"0":"1599","1":"1316","2":"1482"}
                 Lag: {"0":0,"1":0,"2":0}
ReasonOfStateChanged:
        ErrorLogUrls:
            OtherMsg:
```

其中 每个字段说明如下：

```
                 Id: 作业ID
               Name: 作业名称
         CreateTime: 作业创建时间
          PauseTime: 最近一次作业暂停时间
            EndTime: 作业结束时间
             DbName: 对应数据库名称
          TableName: 对应表名称
              State: 作业运行状态
       DataSourceType: 数据源类型：KAFKA
       CurrentTaskNum: 当前子任务数量
        JobProperties: 作业配置详情
DataSourceProperties: 数据源配置详情
    CustomProperties: 自定义配置
           Statistic: 作业运行状态统计信息
            Progress: 作业运行进度
                 Lag: 作业延迟状态
ReasonOfStateChanged: 作业状态变更的原因
        ErrorLogUrls: 被过滤的质量不合格的数据的查看地址
            OtherMsg: 其他错误信息
```

State

有以下4种 State：

NEED_SCHEDULE：作业等待被调度

RUNNING：作业运行中

PAUSED：作业被暂停

STOPPED：作业已结束

CANCELLED：作业已取消

Kafka开启写入任务

这里通过脚本模式kafka写入，每隔5秒写入200条msg。

之后通过show routine load命令可以在process字段看到kafka的3个offset不断的在推进。

```
Progress: {"0":"2061","1":"2135","2":"2254"}
Progress: {"0":"2279","1":"2293","2":"2321"}
```

修改routine load内容

可以通过以下sql 对routine load 进行修改

```
ALTER ROUTINE LOAD FOR [db.]job_name
[job_properties]
FROM data_source
[data_source_properties]
```

注意：

能修改处于 PAUSED 状态的作业。

例如，需要修改 max_batch_interval 为 10s。

先停止对应的任务

使用以下命令暂停任务：

```
PAUSE [ALL] ROUTINE LOAD FOR job_name
```

```
MySQL [tpch_100_d]> PAUSE  ROUTINE LOAD FOR rtl_20230809;
Query OK, 0 rows affected (0.00 sec)
MySQL [tpch_100_d]> show ROUTINE LOAD FOR rtl_20230809\\G;
*************************** 1. row ***************************
                  Id: 21619
                Name: rtl_20230809
          CreateTime: 2023-08-09 19:17:16
           PauseTime: 2023-08-09 21:03:21
             EndTime: NULL
              DbName: default_cluster:tpch_100_d
           TableName: lineitem_rtload
               State: PAUSED
      DataSourceType: KAFKA
      CurrentTaskNum: 0
       JobProperties: {"timezone":"Asia/Shanghai","send_batch_parallelism":"1","col
DataSourceProperties: {"topic":"doris_routine_load_test","currentKafkaPartitions":"
    CustomProperties: {"kafka_default_offsets":"OFFSET_BEGINNING","group.id":"ryanz
           Statistic: {"receivedBytes":1123678,"runningTxns":[],"errorRows":0,"comm
            Progress: {"0":"2917","1":"2754","2":"3029"}
                 Lag: {"0":0,"1":0,"2":0}
ReasonOfStateChanged: ErrorReason{code=errCode = 100, msg='User root pauses routine
        ErrorLogUrls:
```

```
             OtherMsg:
1 row in set (0.00 sec)
```

可以看到状态已经为暂停：State: PAUSED

修改任务参数

```
ALTER ROUTINE LOAD FOR tpch_100_d.rtl_20230809
    PROPERTIES
    (
        "max_batch_interval" = "10"
    );
```

重启任务

```
RESUME  ROUTINE LOAD FOR rtl_20230809;
```

通过 show ROUTINE LOAD FOR rtl_20230809\\G; 命令可以看到任务状态变为 running。并且 Progress 字段刷新的频率变为5s左右，证明变更生效。

更多帮助

关于 Routine Load 使用的更多详细语法，可以在 Mysql 客户端命令行下输入 `HELP ROUTINE LOAD` 获取更多帮助信息。

Flink Connector（Flink 实时或批量数据）

最近更新时间：2024-06-27 11:02:34

基于 Flink Connector 实现数据实时或批量导入 Doris

说明

本文档适用于 flink-doris-connector 1.1.0之后的版本。

基本介绍

Flink Doris Connector 支持通过 Flink 操作（读取、插入、修改、删除） Doris 中存储的数据。不只是导入。由于 Flink 是批流一体的计算引擎，因此实时的增量数据和存量的批量数据都可通过 Flink Doris Connector 导入 Doris。

代码库地址： `https://github.com/apache/doris-flink-connector`

本质是将 `Doris` 表映射为 `DataStream` 或者 `Table` 。

注意

修改和删除只支持在 Unique Key 模型上。

-n 目前的删除是支持 Flink CDC 的方式接入数据实现自动删除，如果是其他数据接入的方式，删除需要自己实现。

Flink CDC 的数据删除使用方式参照本文档最后一节。

版本兼容

Connector Version	Flink Version	Doris Version	Java Version	Scala Version
1.0.3	1.11+	0.15+	8	2.11,2.12
1.1.0	1.14+	1.0+	8	2.11,2.12
1.2.0	1.15+	1.0+	8	-

使用方法

Flink 读写 Doris 数据主要有两种方式：

SQL

DataStream

参数配置

Flink Doris Connector Sink 的内部实现是通过 `Stream Load` 服务向 Doris 写入数据, 同时也支持 `Stream Load` 请求参数的配置设置，具体参数可参考 `Stream Load手册` ，配置方法如下：

SQL 使用 `WITH` 参数 `sink.properties.` 配置

DataStream 使用方法 `DorisExecutionOptions.builder().setStreamLoadProp(Properties)` 配置

SQL

Source（Doris 表作为数据源）

```
CREATE TABLE flink_doris_source (
  name STRING,
  age INT,
  price DECIMAL(5,2),
  sale DOUBLE
  )
  WITH (
    'connector' = 'doris',
    'fenodes' = 'FE_IP:8030',
    'table.identifier' = 'database.table',
    'username' = 'root',
    'password' = 'password'
);
```

Sink（Doris 表作为导入目标表）

```
-- enable checkpoint
SET 'execution.checkpointing.interval' = '10s';
CREATE TABLE flink_doris_sink (
  name STRING,
  age INT,
  price DECIMAL(5,2),
  sale DOUBLE
  )
  WITH (
    'connector' = 'doris',
    'fenodes' = 'FE_IP:8030',
    'table.identifier' = 'db.table',
    'username' = 'root',
    'password' = 'password',
    'sink.label-prefix' = 'doris_label'
);
```

Insert

```
INSERT INTO flink_doris_sink select name,age,price,sale from flink_doris_source
```

DataStream

Source

```
DorisOptions.Builder builder = DorisOptions.builder()
      .setFenodes("FE_IP:8030")
```

```
        .setTableIdentifier("db.table")
        .setUsername("root")
        .setPassword("password");

DorisSource<List<?>> dorisSource = DorisSourceBuilder.<List<?>>builder()
        .setDorisOptions(builder.build())
        .setDorisReadOptions(DorisReadOptions.builder().build())
        .setDeserializer(new SimpleListDeserializationSchema())
        .build();

env.fromSource(dorisSource, WatermarkStrategy.noWatermarks(), "doris source").print
```

Sink

String 数据流

```
// enable checkpoint
env.enableCheckpointing(10000);

DorisSink.Builder<String> builder = DorisSink.builder();
DorisOptions.Builder dorisBuilder = DorisOptions.builder();
dorisBuilder.setFenodes("FE_IP:8030")
        .setTableIdentifier("db.table")
        .setUsername("root")
        .setPassword("password");


DorisExecutionOptions.Builder  executionBuilder = DorisExecutionOptions.builder();
executionBuilder.setLabelPrefix("label-doris"); //streamload label prefix

builder.setDorisReadOptions(DorisReadOptions.builder().build())
        .setDorisExecutionOptions(executionBuilder.build())
        .setSerializer(new SimpleStringSerializer()) //serialize according to string
        .setDorisOptions(dorisBuilder.build());


//mock string source
List<Tuple2<String, Integer>> data = new ArrayList<>();
data.add(new Tuple2<>("doris",1));
DataStreamSource<Tuple2<String, Integer>> source = env.fromCollection(data);

source.map((MapFunction<Tuple2<String, Integer>, String>) t -> t.f0 + "\\t" + t.f1)
    .sinkTo(builder.build());
```

RowData 数据流

```
// enable checkpoint
env.enableCheckpointing(10000);
```

```
//doris sink option
DorisSink.Builder<RowData> builder = DorisSink.builder();
DorisOptions.Builder dorisBuilder = DorisOptions.builder();
dorisBuilder.setFenodes("FE_IP:8030")
      .setTableIdentifier("db.table")
      .setUsername("root")
      .setPassword("password");

// json format to streamload
Properties properties = new Properties();
properties.setProperty("format", "json");
properties.setProperty("read_json_by_line", "true");
DorisExecutionOptions.Builder  executionBuilder = DorisExecutionOptions.builder();
executionBuilder.setLabelPrefix("label-doris") //streamload label prefix
              .setStreamLoadProp(properties); //streamload params

//flink rowdata‘s schema
String[] fields = {"city", "longitude", "latitude"};
DataType[] types = {DataTypes.VARCHAR(256), DataTypes.DOUBLE(), DataTypes.DOUBLE()};

builder.setDorisReadOptions(DorisReadOptions.builder().build())
      .setDorisExecutionOptions(executionBuilder.build())
      .setSerializer(RowDataSerializer.builder()    //serialize according to rowdat
                     .setFieldNames(fields)
                     .setType("json")           //json format
                     .setFieldType(types).build())
      .setDorisOptions(dorisBuilder.build());

//mock rowdata source
DataStream<RowData> source = env.fromElements("")
  .map(new MapFunction<String, RowData>() {
      @Override
      public RowData map(String value) throws Exception {
          GenericRowData genericRowData = new GenericRowData(3);
          genericRowData.setField(0, StringData.fromString("beijing"));
          genericRowData.setField(1, 116.405419);
          genericRowData.setField(2, 39.916927);
          return genericRowData;
      }
  });

source.sinkTo(builder.build());
```

配置

通用配置项

Key	Default Value	Required	Comment
fenodes	--	Y	Doris FE HTTP 地址
table.identifier	--	Y	Doris 表名，如：db.tbl
username	--	Y	访问 Doris 的用户名
password	--	Y	访问 Doris 的密码
doris.request.retries	3	N	向 Doris 发送请求的重试次数
doris.request.connect.timeout.ms	30000	N	向 Doris 发送请求的连接超时时间
doris.request.read.timeout.ms	30000	N	向 Doris 发送请求的读取超时时间
doris.request.query.timeout.s	3600	N	查询 Doris 的超时时间，默认值为1小时，-1表示无超时限制
doris.request.tablet.size	Integer. MAX_VALUE	N	一个 Partition 对应的 Doris Tablet 个数。此数值设置越小，则会生成越多的 Partition。从而提升 Flink 侧的并行度，但同时会对 Doris 造成更大的压力
doris.batch.size	1024	N	一次从 BE 读取数据的最大行数。增大此数值可减少 Flink 与 Doris 之间建立连接的次数。 从而减轻网络延迟所带来的额外时间开销
doris.exec.mem.limit	2147483648	N	单个查询的内存限制。默认为 2GB，单位为字节
doris.deserialize.arrow.async	FALSE	N	是否支持异步转换 Arrow 格式到 flink-doris-connector 迭代所需的 RowBatch
doris.deserialize.queue.size	64	N	异步转换 Arrow 格式的内部处理队列，当 doris.deserialize.arrow.async 为 true 时生效
doris.read.field	--	N	读取 Doris 表的列名列表，多列之间使用逗号分隔
doris.filter.query	--	N	过滤读取数据的表达式，此表达式透传给 Doris。Doris 使用此表达式完成源端数据过滤
sink.label-prefix	--	Y	Stream load 导入使用的 label 前缀。2pc

			场景下要求全局唯一，用来保证 Flink 的 EOS 语义
sink.properties.*	--	N	Stream Load 的导入参数。 例如: 'sink.properties.column_separator' = ', ' 定义列分隔符, 'sink.properties.escape_delimiters' = 'true' 特殊字符作为分隔符，'\\x01'会被转换为二进制的0x01 JSON 格式导入 'sink.properties.format' = 'json' 'sink.properties.read_json_by_line' = 'true'
sink.enable-delete	TRUE	N	是否启用删除。此选项需要 Doris 表开启批量删除功能(Doris0.15+版本默认开启)，只支持 Unique 模型
sink.enable-2pc	TRUE	N	是否开启两阶段提交(2pc)，默认为 true，保证 Exactly-Once 语义

Doris 和 Flink 列类型映射关系

Doris Type	Flink Type
NULL_TYPE	NULL
BOOLEAN	BOOLEAN
TINYINT	TINYINT
SMALLINT	SMALLINT
INT	INT
BIGINT	BIGINT
FLOAT	FLOAT
DOUBLE	DOUBLE
DATE	DATE
DATETIME	TIMESTAMP
DECIMAL	DECIMAL
CHAR	STRING

LARGEINT	STRING
VARCHAR	STRING
DECIMALV2	DECIMAL
TIME	DOUBLE
HLL	Unsupported datatype

使用 Flink CDC 接入 Doris 示例（支持 Insert / Update / Delete 事件）

```
CREATE TABLE cdc_mysql_source (
  id int
  ,name VARCHAR
  ,PRIMARY KEY (id) NOT ENFORCED
) WITH (
 'connector' = 'mysql-cdc',
 'hostname' = '127.0.0.1',
 'port' = '3306',
 'username' = 'root',
 'password' = 'password',
 'database-name' = 'database',
 'table-name' = 'table'
);

-- 支持删除事件同步(sink.enable-delete='true'),需要 Doris 表开启批量删除功能
CREATE TABLE doris_sink (
id INT,
name STRING
)
WITH (
  'connector' = 'doris',
  'fenodes' = '127.0.0.1:8030',
  'table.identifier' = 'database.table',
  'username' = 'root',
  'password' = '',
  'sink.properties.format' = 'json',
  'sink.properties.read_json_by_line' = 'true',
  'sink.enable-delete' = 'true',
  'sink.label-prefix' = 'doris_label'
);

insert into doris_sink select id,name from cdc_mysql_source;
```

Java 示例

`samples/doris-demo/fink-demo/` 下提供了 Java 版本的示例，可供参考，查看点击 这里。

最佳实践

应用场景

使用 Flink Doris Connector 最适合的场景就是实时/批量同步源数据（从 Mysql，Oracle，PostgreSQL 等）到 Doris，使用 Flink 对 Doris 中的数据和其他数据源进行联合分析，也可以使用 Flink Doris Connector。

其他

Flink Doris Connector 主要是依赖 Checkpoint 进行流式写入，所以 Checkpoint 的间隔即为数据的可见延迟时间。为了保证 Flink 的 Exactly Once 语义，Flink Doris Connector 默认开启两阶段提交，Doris 在1.1版本后默认开启两阶段提交。1.0可通过修改 BE 参数开启，可参考 Stream load（本地文件）。

常见问题

1. **Bitmap 类型写入**

```
CREATE TABLE bitmap_sink (
dt int,
page string,
user_id int
)
WITH (
  'connector' = 'doris',
  'fenodes' = '127.0.0.1:8030',
  'table.identifier' = 'test.bitmap_test',
  'username' = 'root',
  'password' = '',
  'sink.label-prefix' = 'doris_label',
  'sink.properties.columns' = 'dt,page,user_id,user_id=to_bitmap(user_id)'
)
```

2. **errCode = 2, detailMessage = Label [label_0_1] has already been used, relate to txn [19650]**

Exactly-Once 场景下，Flink Job 重启时必须从最新的 Checkpoint/Savepoint 启动，否则会报如上错误。

不要求 Exactly-Once 时，也可通过关闭2PC 提交（sink.enable-2pc=false） 或更换不同的 sink.label-prefix 解决。

INSERT INTO

最近更新时间：2024-06-27 11:03:41

Insert Into 语句的使用方式和 MySQL 等数据库中 Insert Into 语句的使用方式类似。但在 Doris 中，所有的数据写入都是一个独立的导入作业。所以这里将 Insert Into 也作为一种导入方式介绍。
主要的 Insert Into 命令包含以下两种：
INSERT INTO tbl SELECT ...
INSERT INTO tbl (col1, col2, ...) VALUES (1, 2, ...), (1,3, ...);
其中第二种命令仅用于 Demo，不要使用在测试或生产环境中。

基本操作

创建导入

Insert Into 命令需要通过 MySQL 协议提交，创建导入请求会同步返回导入结果。
语法：

```
INSERT INTO table_name [partition_info] [WITH LABEL label] [col_list] [query_stmt]
```

示例：

```
INSERT INTO tbl2 WITH LABEL label1 SELECT * FROM tbl3;
INSERT INTO tbl1 VALUES ("qweasdzxcqweasdzxc"), ("a");
```

注意
当需要使用 `CTE(Common Table Expressions)` 作为 insert 操作中的查询部分时，必须指定 `WITH LABEL` 和 column list 部分。
示例：

```
INSERT INTO tbl1 WITH LABEL label1
WITH cte1 AS (SELECT * FROM tbl1), cte2 AS (SELECT * FROM tbl2)
SELECT k1 FROM cte1 JOIN cte2 WHERE cte1.k1 = 1;


INSERT INTO tbl1 (k1)
WITH cte1 AS (SELECT * FROM tbl1), cte2 AS (SELECT * FROM tbl2)
SELECT k1 FROM cte1 JOIN cte2 WHERE cte1.k1 = 1;
```

下面主要介绍创建导入语句中使用到的参数：
partition_info
导入表的目标分区，如果指定目标分区，则只会导入符合目标分区的数据。如果没有指定，则默认值为这张表的所

有分区。

col_list

导入表的目标列，可以以任意的顺序存在。如果没有指定目标列，那么默认值是这张表的所有列。如果待表中的某个列没有存在目标列中，那么这个列需要有默认值，否则 Insert Into 就会执行失败。

如果查询语句的结果列类型与目标列的类型不一致，那么会调用隐式类型转化，如果不能够进行转化，那么 Insert Into 语句会报语法解析错误。

query_stmt

通过一个查询语句，将查询语句的结果导入到 Doris 系统中的其他表。查询语句支持任意 Doris 支持的 SQL 查询语法。

VALUES

用户可以通过 VALUES 语法插入一条或者多条数据。

注意

VALUES 方式仅适用于导入几条数据作为导入 DEMO 的情况，完全不适用于任何测试和生产环境。Doris 系统本身也不适合单条数据导入的场景。建议使用 INSERT INTO SELECT 的方式进行批量导入。

WITH LABEL

INSERT 操作作为一个导入任务，也可以指定一个 label。如果不指定，则系统会自动指定一个 UUID 作为 label。该功能需要 0.11+ 版本。

注意

建议指定 Label 而不是由系统自动分配。如果由系统自动分配，但在 Insert Into 语句执行过程中，因网络错误导致连接断开等，则无法得知 Insert Into 是否成功。而如果指定 Label，则可以再次通过 Label 查看任务结果。

导入结果

Insert Into 本身就是一个 SQL 命令，其返回结果会根据执行结果的不同，分为以下几种：

1. 结果集为空。

如果 insert 对应 select 语句的结果集为空，则返回如下：

```
mysql> insert into tbl1 select * from empty_tbl;
Query OK, 0 rows affected (0.02 sec)
```

`Query OK` 表示执行成功。 `0 rows affected` 表示没有数据被导入。

2. 结果集不为空。

在结果集不为空的情况下。返回结果分为如下几种情况：

2.1 Insert 执行成功并可见：

```
mysql> insert into tbl1 select * from tbl2;
Query OK, 4 rows affected (0.38 sec)
{'label':'insert_8510c568-9eda-4173-9e36-6adc7d35291c', 'status':'visible', 'txnId'

mysql> insert into tbl1 with label my_label1 select * from tbl2;
Query OK, 4 rows affected (0.38 sec)
{'label':'my_label1', 'status':'visible', 'txnId':'4005'}
```

```
mysql> insert into tbl1 select * from tbl2;
Query OK, 2 rows affected, 2 warnings (0.31 sec)
{'label':'insert_f0747f0e-7a35-46e2-affa-13a235f4020d', 'status':'visible', 'txnId'

mysql> insert into tbl1 select * from tbl2;
Query OK, 2 rows affected, 2 warnings (0.31 sec)
{'label':'insert_f0747f0e-7a35-46e2-affa-13a235f4020d', 'status':'committed', 'txnI
```

`Query OK` 表示执行成功。 `4 rows affected` 表示总共有4行数据被导入。 `2 warnings` 表示被过滤的行数。

同时会返回一个 json 串：

```
{'label':'my_label1', 'status':'visible', 'txnId':'4005'}
{'label':'insert_f0747f0e-7a35-46e2-affa-13a235f4020d', 'status':'committed', 'txnI
{'label':'my_label1', 'status':'visible', 'txnId':'4005', 'err':'some other error'}
```

`label` 为用户指定的 label 或自动生成的 label。Label 是该 Insert Into 导入作业的标识。每个导入作业，都有一个在单 database 内部唯一的 Label。

`status` 表示导入数据是否可见。如果可见，显示 `visible` ，如果不可见，显示 `committed` 。

`txnId` 为这个 insert 对应的导入事务的 id。

`err` 字段会显示一些其他非预期错误。

当需要查看被过滤的行时，用户可以通过如下语句：

```
show load where label="xxx";
```

返回结果中的 URL 可以用于查询错误的数据，具体见后面 **查看错误行** 小结。

数据不可见是一个临时状态，这批数据最终是一定可见的

可以通过如下语句查看这批数据的可见状态：

```
show transaction where id=4005;
```

返回结果中的 `TransactionStatus` 列若为 `visible` ，则表述数据可见。

2.2 Insert 执行失败

执行失败表示没有任何数据被成功导入，并返回如下：

```
mysql> insert into tbl1 select * from tbl2 where k1 = "a";
ERROR 1064 (HY000): all partitions have no load data. url: http://10.74.167.16:8042
```

其中 `ERROR 1064 (HY000): all partitions have no load data` 显示失败原因。后面的 url 可以用于查询错误的数据，具体见后面 **查看错误行** 小结。

综上，对于 insert 操作返回结果的正确处理逻辑应为：

如果返回结果为 `ERROR 1064 (HY000)` ，则表示导入失败。

如果返回结果为 `Query OK` ，则表示执行成功。

如果 `rows affected` 为 0，表示结果集为空，没有数据被导入。
如果 `rows affected` 大于 0：
如果 `status` 为 `committed`，表示数据还不可见。需要通过 `show transaction` 语句查看状态直到 `visible` 。
如果 `status` 为 `visible`，表示数据导入成功。
如果 `warnings` 大于 0，表示有数据被过滤，可以通过 `show load` 语句获取 url 查看被过滤的行。

SHOW LAST INSERT

在上一小节中我们介绍了如何根据 insert 操作的返回结果进行后续处理。但一些语言的mysql类库中很难获取返回结果的中的 json 字符串。因此，Doris 还提供了 `SHOW LAST INSERT` 命令来显式的获取最近一次 insert 操作的结果。
当执行完一个 insert 操作后，可以在同一 session 连接中执行 `SHOW LAST INSERT` 。该命令会返回最近一次 insert 操作的结果，如：

```
mysql> show last insert\\G
*************************** 1. row ***************************
    TransactionId: 64067
            Label: insert_ba8f33aea9544866-8ed77e2844d0cc9b
         Database: default_cluster:db1
            Table: t1
TransactionStatus: VISIBLE
       LoadedRows: 2
     FilteredRows: 0
```

该命令会返回 insert 以及对应事务的详细信息。因此，用户可以在每次执行完 insert 操作后，继续执行 `show last insert` 命令来获取 insert 的结果。
注意
该命令只会返回在同一 session 连接中，最近一次 insert 操作的结果。如果连接断开或更换了新的连接，则将返回空集。

相关系统配置

FE 配置

timeout
导入任务的超时时间(以秒为单位)，导入任务在设定的 timeout 时间内未完成则会被系统取消，变成 CANCELLED。目前 Insert Into 并不支持自定义导入的 timeout 时间，所有 Insert Into 导入的超时时间是统一的，默认的 timeout 时间为1小时。如果导入的源文件无法在规定时间内完成导入，则需要调整 FE 的参数 `insert_load_default_timeout_second` 。

同时 Insert Into 语句收到 Session 变量 `query_timeout` 的限制。可以通过 `SET query_timeout = xxx;` 来增加超时时间，单位是秒。

Session 变量

enable_insert_strict

Insert Into 导入本身不能控制导入可容忍的错误率。用户只能通过 `enable_insert_strict` 这个 Session 参数用来控制。

当该参数设置为 false 时，表示至少有一条数据被正确导入，则返回成功。如果有失败数据，则还会返回一个 Label。

当该参数设置为 true 时，表示如果有一条数据错误，则导入失败。

默认为 false。可通过 `SET enable_insert_strict = true;` 来设置。

query_timeout

Insert Into 本身也是一个 SQL 命令，因此 Insert Into 语句也受到 Session 变量 `query_timeout` 的限制。可以通过 `SET query_timeout = xxx;` 来增加超时时间，单位是秒。

最佳实践

应用场景

1. 用户希望仅导入几条假数据，验证一下 Doris 系统的功能。此时适合使用 INSERT 的语法。
2. 用户希望将已经在 Doris 表中的数据进行 ETL 转换并导入到一个新的 Doris 表中，此时适合使用 INSERT 语法。
3. 用户可以创建一种外部表，如 MySQL 外部表映射一张 MySQL 系统中的表。或者创建 Broker 外部表来映射 HDFS 上的数据文件。然后通过 INSERT 语法将外部表中的数据导入到 Doris 表中存储。

数据量

Insert Into 对数据量没有限制，大数据量导入也可以支持。但 Insert Into 有默认的超时时间，用户预估的导入数据量过大，就需要修改系统的 Insert Into 导入超时时间。

```
导入数据量 = 36G 约≤ 3600s * 10M/s
其中 10M/s 是最大导入限速，用户需要根据当前集群情况计算出平均的导入速度来替换公式中的 10M/s
```

完整例子

用户有一张表 store_sales 在数据库 sales 中，用户又创建了一张表叫 bj_store_sales 也在数据库 sales 中，用户希望将 store_sales 中销售记录在 bj 的数据导入到这张新建的表 bj_store_sales 中。导入的数据量约为：10G。

```
store_sales schema：
(id, total, user_id, sale_timestamp, region)

bj_store_sales schema:
```

```
(id, total, user_id, sale_timestamp)
```

集群情况：用户当前集群的平均导入速度约为 5M/s

Step1: 判断是否要修改 Insert Into 的默认超时时间

```
计算导入的大概时间
10G / 5M/s = 2000s

修改 FE 配置
insert_load_default_timeout_second = 2000
```

Step2：创建导入任务

由于用户是希望将一张表中的数据做 ETL 并导入到目标表中，所以应该使用 Insert into query_stmt 方式导入。

```
INSERT INTO bj_store_sales WITH LABEL `label` SELECT id, total, user_id, sale_times
```

常见问题

查看错误行

由于 Insert Into 无法控制错误率，只能通过 `enable_insert_strict` 设置为完全容忍错误数据或完全忽略错误数据。因此如果 `enable_insert_strict` 设为 true，则 Insert Into 可能会失败。而如果 `enable_insert_strict` 设为 false，则可能出现仅导入了部分合格数据的情况。

当返回结果中提供了 url 字段时，可以通过以下命令查看错误行：`SHOW LOAD WARNINGS ON "url";`

示例：

```
SHOW LOAD WARNINGS ON "http://ip:port/api/_load_error_log?file=__shard_13/error_log
```

错误的原因通常如：源数据列长度超过目的数据列长度、列类型不匹配、分区不匹配、列顺序不匹配等。

使用 JDBC 同步数据

最近更新时间：2024-06-27 11:04:00

用户可以通过 JDBC 协议，使用 INSERT 语句进行数据导入。
INSERT 语句的使用方式和 MySQL 等数据库中 INSERT 语句的使用方式类似。 INSERT 语句支持以下两种语法：

```
* INSERT INTO table SELECT ...
* INSERT INTO table VALUES(...)
```

这里我们仅介绍第二种方式。关于 INSERT 命令的详细说明，请参阅 INSERT 命令文档。

单次写入

单次写入是指用户直接执行一个 INSERT 命令。示例如下：

```
INSERT INTO example_tbl (col1, col2, col3) VALUES (1000, "test", 3.25);
```

对于 Doris 来说，一个 INSERT 命令就是一个完整的导入事务。因此不论是导入一条数据，还是多条数据，我们都不建议在生产环境使用这种方式进行数据导入。 高频次的 INSERT 操作会导致在存储层产生大量的小文件，会严重影响系统性能。
该方式仅用于线下简单测试或低频少量的操作。或者可以使用以下方式进行批量的插入操作：

```
INSERT INTO example_tbl VALUES
(1000, "baidu1", 3.25)
(2000, "baidu2", 4.25)
(3000, "baidu3", 5.25);
```

我们建议一批次插入条数尽量大，例如几千甚至一万条一次。或者可以通过下面的程序的方式，使用 PreparedStatement 来进行批量插入。

JDBC 示例

这里我们给出一个简单的 JDBC 批量 INSERT 代码示例：

```
package demo.doris;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
```

```
public class DorisJDBCDemo {

    private static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    private static final String DB_URL_PATTERN = "jdbc:mysql://%s:%d/%s?rewriteBatc
    private static final String HOST = "127.0.0.1"; // Leader Node host
    private static final int PORT = 9030;   // query_port of Leader Node
    private static final String DB = "demo";
    private static final String TBL = "test_1";
    private static final String USER = "admin";
    private static final String PASSWD = "my_pass";

    private static final int INSERT_BATCH_SIZE = 10000;

    public static void main(String[] args) {
        insert();
    }

    private static void insert() {
        // 注意末尾不要加 分号 ";"
        String query = "insert into " + TBL + " values(?, ?)";
        // 设置 Label 以做到幂等。
        // String query = "insert into " + TBL + " WITH LABEL my_label values(?, ?)

        Connection conn = null;
        PreparedStatement stmt = null;
        String dbUrl = String.format(DB_URL_PATTERN, HOST, PORT, DB);
        try {
            Class.forName(JDBC_DRIVER);
            conn = DriverManager.getConnection(dbUrl, USER, PASSWD);
            stmt = conn.prepareStatement(query);

            for (int i =0; i < INSERT_BATCH_SIZE; i++) {
                stmt.setInt(1, i);
                stmt.setInt(2, i * 100);
                stmt.addBatch();
            }

            int[] res = stmt.executeBatch();
            System.out.println(res);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (stmt != null) {
                    stmt.close();
                }
            } catch (SQLException se2) {
```

```
                se2.printStackTrace();
            }
            try {
                if (conn != null) conn.close();
            } catch (SQLException se) {
                se.printStackTrace();
            }
        }
    }
}
```

请注意以下几点：

1. JDBC 连接串需添加 `rewriteBatchedStatements=true` 参数，并使用 `PreparedStatement` 方式。

目前 Doris 暂不支持服务器端的 PrepareStatemnt，所以 JDBC Driver 会在客户端进行批量 Prepare。

`rewriteBatchedStatements=true` 会确保 Driver 执行批处理。并最终形成如下形式的 INSERT 语句发往 Doris：

```
INSERT INTO example_tbl VALUES
(1000, "tencent1", 3.25)
(2000, "tencent2", 4.25)
(3000, "tencent3", 5.25);
```

2. 批次大小

因为是在客户端进行批量处理，因此一批次如果过大的话，会多占用客户端的内存资源，需关注。

说明

Doris 后续会支持服务端的 PrepareStatemnt，敬请期待。

3. 导入原子性

和其他导入方式一样，INSERT 操作本身也支持原子性。每一个 INSERT 操作都是一个导入事务，能够保证一个 INSERT 中的所有数据原子性的写入。

前面提到，我们建议在使用 INSERT 导入数据时，采用“批”的方式进行导入，而不是单条插入。

同时，我们可以为每次 INSERT 操作设置一个 Label。通过 Label 机制 可以保证操作的幂等性和原子性，最终做到数据的不丢不重。关于 INSERT 中 Label 的具体用法，可以参阅 INSERT 文档。

通过外部表同步数据

最近更新时间：2024-06-27 11:04:20

说明：

本文档所示内容仅适用1.1及之前版本，1.2版本后不推荐使用外部表同步数据。

Doris 可以创建通过 ODBC 协议访问的外部表。创建完成后，可通过 SELECT 语句直接查询外部表的数据，也可以通过 `INSERT INTO SELECT` 的方式导入外部表的数据。

本文档主要介绍如何创建通过 ODBC 协议访问的外部表，以及如何导入这些外部表的数据。目前支持的数据源包括：

MySQL

Oracle

PostgreSQL

SQLServer

Hive(1.0版本支持)

创建外部表

1. 创建 ODBC Resource

ODBC Resource 的作用是用于统一管理外部表的连接信息。

```
CREATE EXTERNAL RESOURCE `oracle_test_odbc`
PROPERTIES (
  "type" = "odbc_catalog",
  "host" = "192.168.0.10",
  "port" = "8086",
  "user" = "oracle",
  "password" = "oracle",
  "database" = "oracle",
  "odbc_type" = "oracle",
  "driver" = "Oracle"
);
```

这里我们创建了一个名为 `oracle_test_odbc` 的 Resource，其类型为 `odbc_catalog`，表示这是一个用于存储 ODBC 信息的 Resource。 `odbc_type` 为 `oracle`，表示这个 ODBC Resource 是用于连接 Oracle 数据库的。

2. 创建外部表

```
CREATE EXTERNAL TABLE `ext_oracle_demo` (
  `k1` decimal(9, 3) NOT NULL COMMENT "",
  `k2` char(10) NOT NULL COMMENT "",
```

```
    `k3` datetime NOT NULL COMMENT "",
    `k5` varchar(20) NOT NULL COMMENT "",
    `k6` double NOT NULL COMMENT ""
) ENGINE=ODBC
COMMENT "ODBC"
PROPERTIES (
 "odbc_catalog_resource" = "oracle_test_odbc",
 "database" = "oracle",
 "table" = "baseall"
);
```

这里我们创建一个 `ext_oracle_demo` 外部表，并引用了之前创建的 `oracle_test_odbc` Resource。

导入数据

1. 创建 Doris 表

这里我们创建一张 Doris 的表，列信息和上一步创建的外部表 `ext_oracle_demo` 一样：

```
CREATE TABLE `doris_oralce_tbl` (
 `k1` decimal(9, 3) NOT NULL COMMENT "",
 `k2` char(10) NOT NULL COMMENT "",
 `k3` datetime NOT NULL COMMENT "",
 `k5` varchar(20) NOT NULL COMMENT "",
 `k6` double NOT NULL COMMENT ""
)
COMMENT "Doris Table"
DISTRIBUTED BY HASH(k1) BUCKETS 2;
PROPERTIES (
  "replication_num" = "1"
);
```

关于创建 Doris 表的详细说明，请参阅 CREATE-TABLE 语法帮助。

2. 导入数据 (从 `ext_oracle_demo` 表 导入到 `doris_oracle_tbl` 表)

```
INSERT INTO doris_oracle_tbl SELECT k1,k2,k3 FROM ext_oracle_demo limit 100;
```

INSERT 命令是同步命令，返回成功，即表示导入成功。

注意事项

必须保证外部数据源与 Doris 集群是可以互通，包括BE节点和外部数据源的网络是互通的。

ODBC 外部表本质上是通过单一 ODBC 客户端访问数据源，因此并不合适一次性导入大量的数据，建议分批多次导入。

MySQL 数据实时或者批量写入

最近更新时间：2024-06-27 11:04:41

MySQL 数据实时写入主要用于导入实时增量数据。批量写入主要用于导入表中存量数据。下面分别说明导入方法。

MySQL 数据实时写入

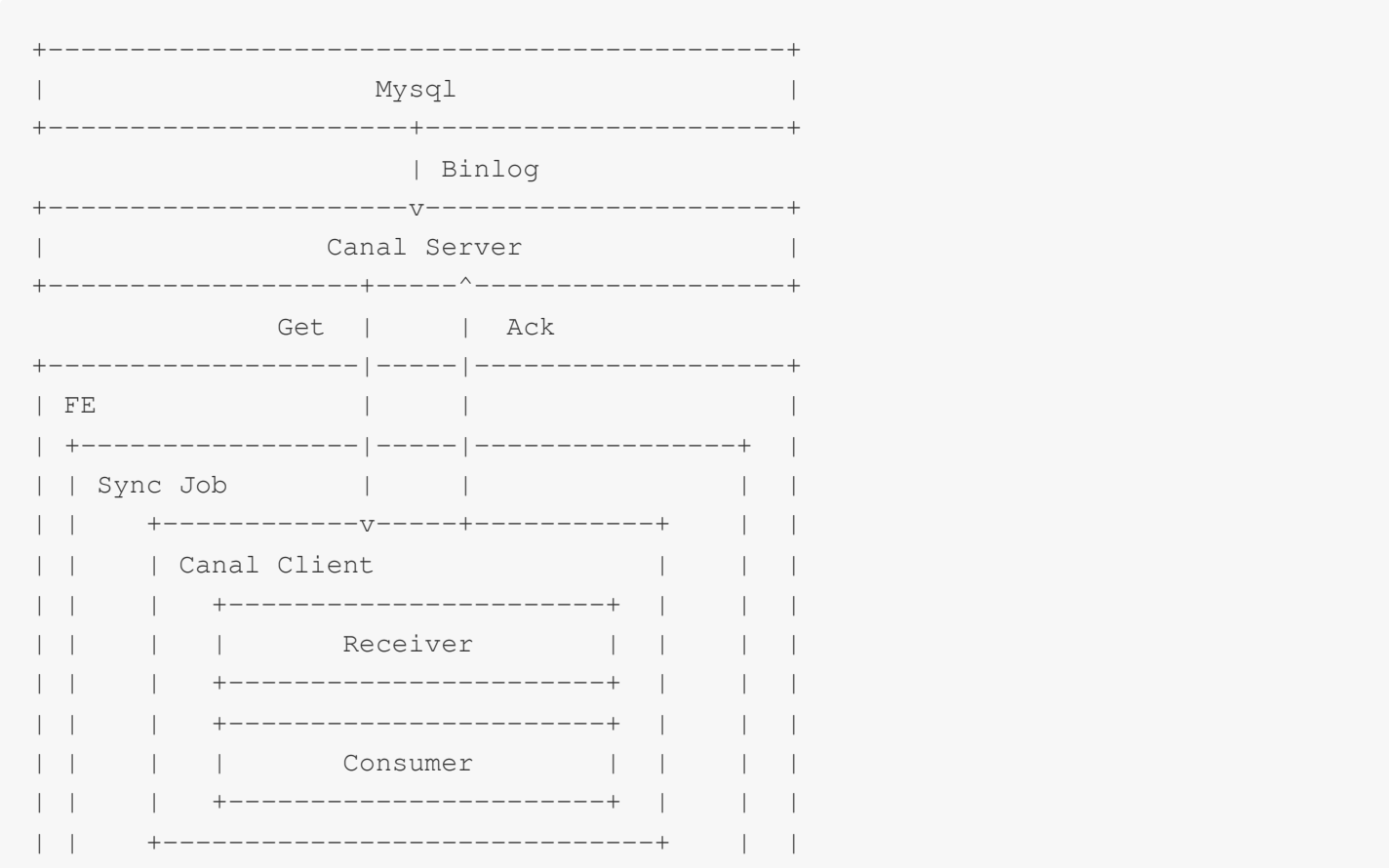
实时写入 Mysql 数据主要采用 Binlog Load 机制实现。Binlog Load 提供了一种使 Doris 增量同步用户对 Mysql 数据库的数据更新操作的 CDC(Change Data Capture)功能。

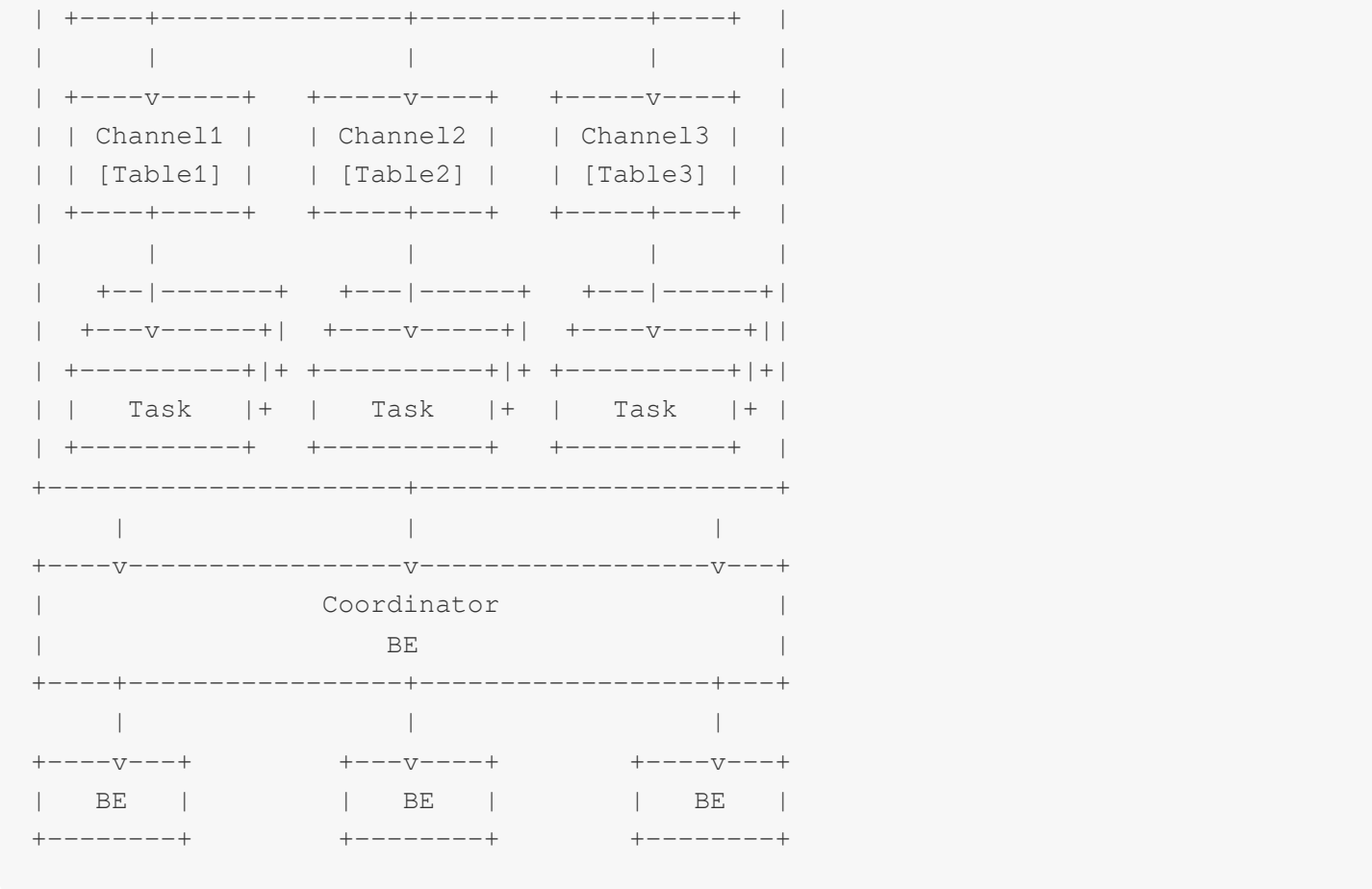
适用场景

INSERT/UPDATE/DELETE 支持
过滤 Query
暂不兼容 DDL 语句

基本原理

当前版本设计中，Binlog Load 需要依赖 canal 作为中间媒介，让 canal 伪造成一个从节点去获取 Mysql 主节点上的 Binlog 并解析，再由 Doris 去获取 Canal 上解析好的数据，主要涉及 Mysql 端、Canal 端以及 Doris 端，总体数据流向如下：





如上，用户向 FE 提交一个数据同步作业。

FE 会为每个数据同步作业启动一个 canal client，来向 canal server 端订阅并获取数据。

client 中的 receive r将负责通过 Get 命令接收数据，每获取到一个数据 batch，都会由 consumer 根据对应表分发到不同的 channel，每个 channel 都会为此数据 batch 产生一个发送数据的子任务 Task。

在 FE 上，一个 Task 是 channel 向 BE 发送数据的子任务，里面包含分发到当前 channel 的同一个 batch 的数据。

channel 控制着单个表事务的开始、提交、终止。一个事务周期内，一般会从 consumer 获取到多个 batch 的数据，因此会产生多个向BE发送数据的子任务 Task，在提交事务成功前，这些 Task 不会实际生效。

满足一定条件时（例如超过一定时间、达到提交最大数据大小），consumer 将会阻塞并通知各个 channel 提交事务。

当且仅当所有 channel 都提交成功，才会通过 Ack 命令通知 canal 并继续获取、消费数据。

如果有任意 channel 提交失败，将会重新从上一次消费成功的位置获取数据并再次提交（已提交成功的 channel 不会再次提交以保证幂等性）。

整个数据同步作业中，FE 通过以上流程不断的从 canal 获取数据并提交到 BE，来完成数据同步。

配置 Mysql 端

在 Mysql Cluster 模式的主从同步中，二进制日志文件（Binlog）记录了主节点上的所有数据变化，数据在 Cluster 的多个节点间同步、备份都要通过 Binlog 日志进行，从而提高集群的可用性。架构通常由一个主节点（负责写）和一个或多个从节点（负责读）构成，所有在主节点上发生的数据变更将会复制给从节点。

注意

目前必须要使用 Mysql 5.7及以上的版本才能支持 Binlog Load 功能。

要打开 mysql 的二进制 binlog 日志功能，则需要编辑 my.cnf 配置文件设置一下。

```
[mysqld]
log-bin = mysql-bin # 开启 binlog
binlog-format=ROW # 选择 ROW 模式
```

Mysql 端说明

在 Mysql 上，Binlog 命名格式为 mysql-bin.000001、mysql-bin.000002... ，满足一定条件时 mysql 会去自动切分 Binlog 日志：

mysql 重启了。

客户端输入命令 flush logs。

binlog 文件大小超过1G。

要定位 Binlog 的最新的消费位置，可以通过 binlog 文件名和 position(偏移量)。例如，各个从节点上会保存当前消费到的 binlog 位置，方便随时断开连接、重新连接和继续消费。

```
---------------------                                  ------------------
|        Slave        |               read              |      Master        |
| FileName/Position | <<<---------------------------- |    Binlog Files    |
---------------------                                  ------------------
```

对于主节点来说，它只负责写入 Binlog，多个从节点可以同时连接到一台主节点上，消费 Binlog 日志的不同部分，互相之间不会影响。

Binlog 日志支持两种主要格式（此外还有混合模式 mixed-based）：

```
statement-based格式：Binlog只保存主节点上执行的sql语句，从节点将其复制到本地重新执行
row-based格式：      Binlog会记录主节点的每一行所有列的数据的变更信息，从节点会复制并执行每一行
```

第一种格式只写入了执行的 sql 语句，虽然日志量会很少，但是有下列缺点：

1. 没有保存每一行实际的数据。
2. 在主节点上执行的 UDF、随机、时间函数会在从节点上结果不一致。
3. limit 语句执行顺序可能不一致。

因此我们需要选择第二种格式，才能从 Binlog 日志中解析出每一行数据。

在 row-based 格式下，Binlog 会记录每一条 binlog event 的时间戳、server id、偏移量等信息，如下面一条带有两条 insert 语句的事务：

```
begin;
insert into canal_test.test_tbl values (3, 300);
insert into canal_test.test_tbl values (4, 400);
commit;
```

对应将会有四条 binlog event，其中一条 begin event，两条 insert event，一条 commit event：

```
SET TIMESTAMP=1538238301/*!*/;
BEGIN
/*!*/.
# at 211935643
# at 211935698
#180930 0:25:01 server id 1 end_log_pos 211935698 Table_map: 'canal_test'.'test_tbl
#180930 0:25:01 server id 1 end_log_pos 211935744 Write_rows: table-id 25 flags: ST
...
'/*!*/;
### INSERT INTO canal_test.test_tbl
### SET
### @1=1
### @2=100
# at 211935744
#180930 0:25:01 server id 1 end_log_pos 211935771 Xid = 2681726641
...
'/*!*/;
### INSERT INTO canal_test.test_tbl
### SET
### @1=2
### @2=200
# at 211935771
#180930 0:25:01 server id 1 end_log_pos 211939510 Xid = 2681726641
COMMIT/*!*/;
```

如图所示，每条 Insert event 中包含了修改的数据。在进行 Delete/Update 操作时，一条 event 还能包含多行数据，使得 Binlog 日志更加的紧密。

开启 GTID 模式（可选）

一个全局事务 Id (global transaction identifier)标识出了一个曾在主节点上提交过的事务，在全局都是唯一有效的。开启了 Binlog 后，GTID 会被写入到 Binlog 文件中，与事务一一对应。
要打开 mysql 的 GTID 模式，则需要编辑 my.cnf 配置文件设置一下：

```
gtid-mode=on // 开启gtid模式
enforce-gtid-consistency=1 // 强制gtid和事务的一致性
```

在 GTID 模式下，主服务器可以不需要 Binlog 的文件名和偏移量，就能很方便的索引事务、恢复数据、复制副本。在 GTID 模式下，由于 GTID 的全局有效性，从节点将不再需要通过保存文件名和偏移量来定位主节点上的 Binlog 位置，而通过数据本身就可以定位了。在进行数据同步中，从节点会跳过执行任意被识别为已执行的 GTID 事务。GTID 的表现形式为一对坐标，`source_id` 标识出主节点，`transaction_id` 表示此事务在主节点上执行的顺序（最大263-1）。

```
GTID = source_id:transaction_id
```

例如，在同一主节点上执行的第23个事务的 gtid 为：

```
3E11FA47-71CA-11E1-9E33-C80AA9429562:23
```

配置 Canal 端

canal 是属于阿里巴巴 otter 项目下的一个子项目，主要用途是基于 MySQL 数据库增量日志解析，提供增量数据订阅和消费，用于解决跨机房同步的业务场景，建议使用 canal 1.1.5及以上版本，点击 下载地址，下载完成后，请按以下步骤完成部署。

1. 解压 canal deployer。
2. 在 conf 文件夹下新建目录并重命名，作为 instance 的根目录，目录名即后文提到的 destination。
3. 修改 instance 配置文件（可拷贝conf/example/instance.properties）：

```
vim conf/{your destination}/instance.properties
```

```
## canal instance serverId
canal.instance.mysql.slaveId = 1234
## mysql adress
canal.instance.master.address = 127.0.0.1:3306
## mysql username/password
canal.instance.dbUsername = canal
canal.instance.dbPassword = canal
```

4. 启动

```
sh bin/startup.sh
```

5. 验证启动成功

```
cat logs/{your destination}/{your destination}.log
```

```
2013-02-05 22:50:45.636 [main] INFO  c.a.o.c.i.spring.support.PropertyPlaceholderCo
2013-02-05 22:50:45.641 [main] INFO  c.a.o.c.i.spring.support.PropertyPlaceholderCo
2013-02-05 22:50:45.803 [main] INFO  c.a.otter.canal.instance.spring.CanalInstanceW
2013-02-05 22:50:45.810 [main] INFO  c.a.otter.canal.instance.spring.CanalInstanceW
```

Canal 端说明

canal 通过伪造自己的 mysql dump 协议，去伪装成一个从节点，获取主节点的 Binlog 日志并解析。

canal server 上可启动多个 instance，一个 instance 可看作一个从节点，每个 instance 由下面几个部分组成：

```
-------------------------------------------------
|  Server                                       |
|  --------------------------------------------- |
```

```
|   |  Instance 1                                  | |
|   |  -----------   -----------  -----------   | |
|   |  |  Parser |   |  Sink   |  | Store   |   | |
|   |  -----------   -----------  -----------   | |
|   |  ------------------------------------     | |
|   |  |             MetaManager          |     | |
|   |  ------------------------------------     | |
|   ---------------------------------------------- |
-------------------------------------------------
```

parser：数据源接入，模拟 slave 协议和 master 进行交互，协议解析。

sink：parser 和 store 链接器，进行数据过滤，加工，分发的工作。

store：数据存储。

meta manager：元数据管理模块。

每个 instance 都有自己在 cluster 内的唯一标识，即 server Id。

在 canal server 内，instance 用字符串表示，此唯一字符串被记为 destination，canal client 需要通过 destination 连接到对应的 instance。

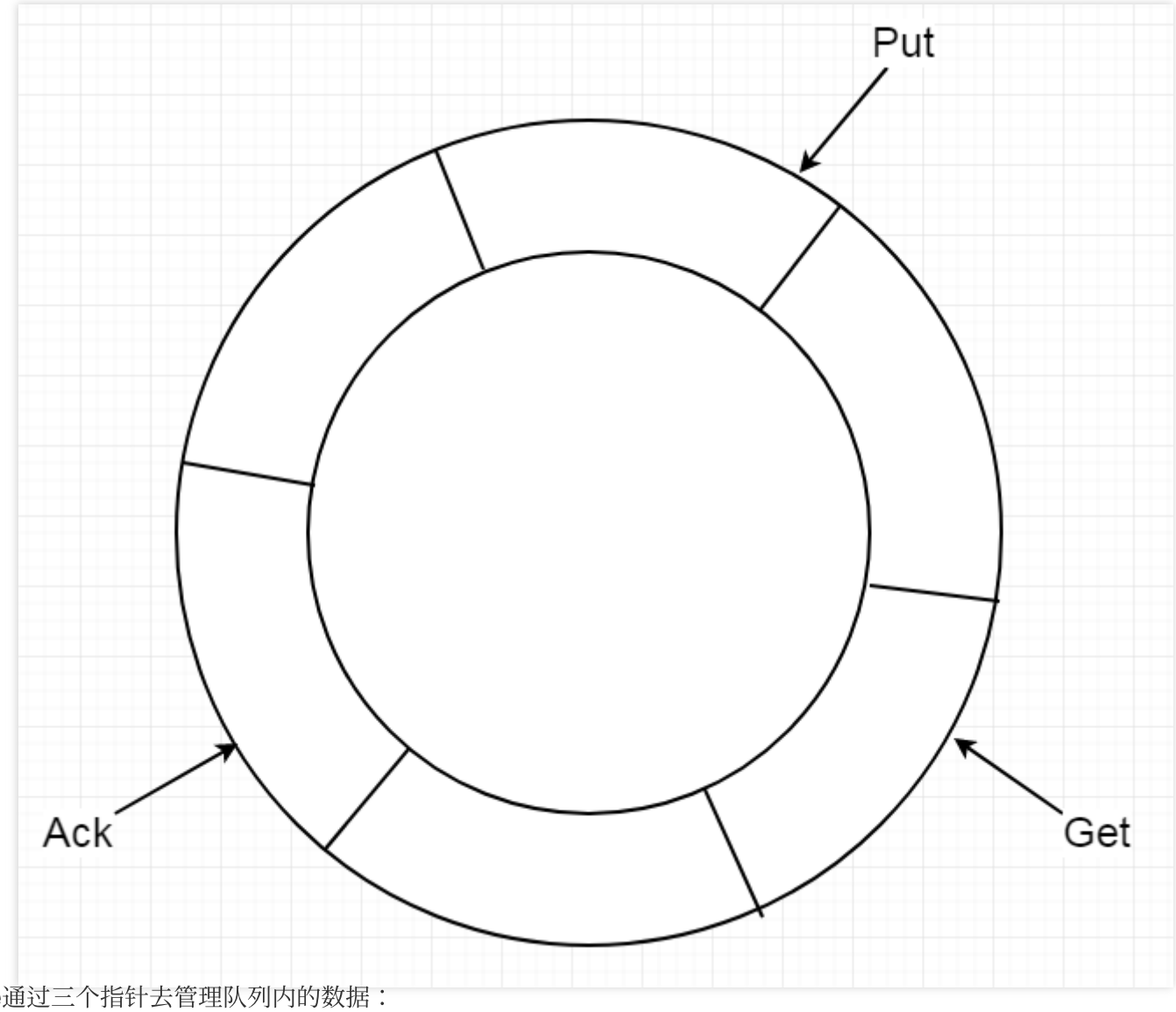
注意

canal client 和 canal instance 是一一对应的，Binlog Load 已限制多个数据同步作业不能连接到同一个 destination。

数据在 instance 内的流向是 **binlog -> parser -> sink -> store**。

instance 通过 parser 模块解析 binlog 日志，解析出来的数据缓存在 store 里面，当用户向FE提交一个数据同步作业时，会启动一个 canal client 订阅并获取对应 instance 中的 store 内的数据。

store 实际上是一个环形的队列，用户可以自行配置它的长度和存储空间。



store通过三个指针去管理队列内的数据：

1. get指针：get指针代表客户端最后获取到的位置。
2. ack指针：ack指针记录着最后消费成功的位置。
3. put指针：代表sink模块最后写入store成功的位置。

```
canal client异步获取store中数据

        get 0         get 1        get 2                    put
          |             |            |          ......        |
          v             v            v                        v
------------------------------------------------------------------------ store环形队列
          ^             ^
          |             |
        ack 0         ack 1
```

canal client 调用 get 命令时，canal server 会产生数据 batch 发送给 client，并右移 get 指针，client 可以获取多个 batch，直到 get 指针赶上 put 指针为止。

当消费数据成功时，client 会返回 ck + batch Id 通知已消费成功了，并右移 ack 指针，store 会从队列中删除此 batch 的数据，腾出空间来从上游 sink 模块获取数据，并右移 put 指针。
当数据消费失败时，client 会返回 rollback 通知消费失败，store 会将 get 指针重置左移到ack指针位置，使下一次 client 获取的数据能再次从 ack 指针处开始。
和 Mysql 中的从节点一样，canal 也需要去保存 client 最新消费到的位置。canal 中所有元数据（如 GTID、Binlog 位置）都是由 MetaManager 去管理的，目前元数据默认以 json 格式持久化在 instance 根目录下的 meta.dat 文件内。

基本操作

配置目标表属性

用户需要先在 Doris 端创建好与 Mysql 端对应的目标表。
Binlog Load 只能支持 Unique 类型的目标表，且必须激活目标表的 Batch Delete 功能。
开启 Batch Delete 的方法可以参考 ALTER TABLE PROPERTY 中的批量删除功能。
示例：

```
--create Mysql table
CREATE TABLE `source_test` (
  `id` int(11) NOT NULL COMMENT "",
  `name` int(11) NOT NULL COMMENT ""
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;


-- create Doris table
CREATE TABLE `target_test` (
  `id` int(11) NOT NULL COMMENT "",
  `name` int(11) NOT NULL COMMENT ""
) ENGINE=OLAP
UNIQUE KEY(`id`)
COMMENT "OLAP"
DISTRIBUTED BY HASH(`id`) BUCKETS 8;

-- enable batch delete
ALTER TABLE target_test ENABLE FEATURE "BATCH_DELETE";
```

注意

Doris 表结构和 Mysql 表结构字段顺序必须保持一致。

创建同步作业

```
CREATE SYNC `demo`.`job`
(
FROM `source_test` INTO `target_test`
(id,name)
)
```

```
FROM BINLOG
(
"type" = "canal",
"canal.server.ip" = "127.0.0.1",
"canal.server.port" = "11111",
"canal.destination" = "xxx",
"canal.username" = "canal",
"canal.password" = "canal"
);
```

创建数据同步作业的详细语法可以连接到 Doris 后，HELP CREATE SYNC JOB 查看语法帮助。这里主要详细介绍，创建作业时的注意事项。

语法：

```
CREATE SYNC [db.]job_name
 (
        channel_desc,
        channel_desc
        ...
 )
binlog_desc
```

job_name

`job_name` 是数据同步作业在当前数据库内的唯一标识，相同 `job_name` 的作业只能有一个在运行。

channel_desc

`channel_desc` 用来定义任务下的数据通道，可表示mysql源表到doris目标表的映射关系。在设置此项时，如果存在多个映射关系，必须满足 mysql 源表应该与 doris 目标表是一一对应关系，其他的任何映射关系（如一对多关系），检查语法时都被视为不合法。

column_mapping

`column_mapping` 主要指 mysql 源表和 doris 目标表的列之间的映射关系，如果不指定，FE会默认源表和目标表的列按顺序一一对应。但是我们依然建议显式的指定列的映射关系，这样当目标表的结构发生变化（例如增加一个 nullable 的列），数据同步作业依然可以进行。否则，当发生上述变动后，因为列映射关系不再一一对应，导入将报错。

binlog_desc

`binlog_desc` 中的属性定义了对接远端 Binlog 地址的一些必要信息，目前可支持的对接类型只有 canal 方式，所有的配置项前都需要加上 canal 前缀。

1.1 `canal.server.ip` ：canal server 的地址。

1.2 `canal.server.port` ：canal server 的端口。

1.3 `canal.destination` ：前文提到的 instance 的字符串标识。

1.4 `canal.batchSize` ：每批从 canal server 处获取的 batch 大小的最大值，默认8192。

1.5 `canal.username` ：instance 的用户名。

1.6 `canal.password` ：instance 的密码。

1.7 `canal.debug` ：设置为 true 时，会将 batch 和每一行数据的详细信息都打印出来，会影响性能。

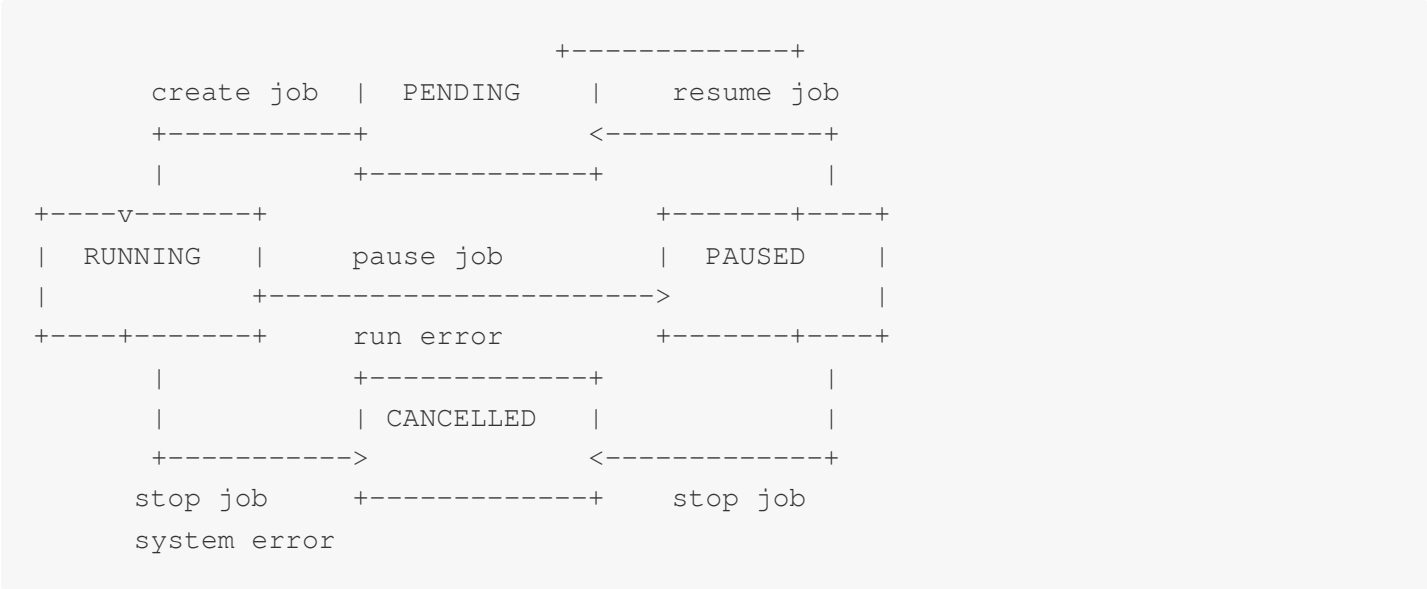
查看作业状态

查看作业状态的具体命令和示例可以通过 SHOW SYNC JOB 命令查看。

返回结果集的参数意义如下：

State

作业当前所处的阶段。作业状态之间的转换如下图所示：



作业提交之后状态为 PENDING，由 FE 调度执行启动 canal client 后状态变成 RUNNING，用户可以通过 STOP/PAUSE/RESUME 三个命令来控制作业的停止，暂停和恢复，操作后作业状态分别为 CANCELLED/PAUSED/RUNNING。

作业的最终阶段只有一个 CANCELLED，当作业状态变为 CANCELLED 后，将无法再次恢复。当作业发生了错误时，若错误是不可恢复的，状态会变成 CANCELLED，否则会变成 PAUSED。

Channel

作业所有源表到目标表的映射关系。

Status

当前 binlog 的消费位置（若设置了 GTID 模式，会显示 GTID），以及 doris 端执行时间相比 mysql 端的延迟时间。

JobConfig

对接的远端服务器信息，如 canal server 的地址与连接 instance 的 destination。

控制作业

用户可以通过 STOP/PAUSE/RESUME 三个命令来控制作业的停止，暂停和恢复。可以通过 STOP SYNC JOB, PAUSE SYNC JOB 以及 RESUME SYNC JOB。

相关参数

CANAL 配置

下面配置属于 canal 端的配置，主要通过修改 conf 目录下的 canal.properties 调整配置值。

`canal.ip`

canal server 的 IP 地址。

`canal.port`

canal server 的端口。

`canal.instance.memory.buffer.size`

canal 端的 store 环形队列的队列长度，必须设为2的幂次方，默认长度16384。此值等于 canal 端能缓存 event 数量的最大值，也直接决定了 Doris 端一个事务内所能容纳的最大 event 数量。建议将它改的足够大，防止 Doris 端一个事务内能容纳的数据量上限太小，导致提交事务太过频繁造成数据的版本堆积。

`canal.instance.memory.buffer.memunit`

canal 端默认一个 event 所占的空间，默认空间为1024 bytes。此值乘以 store 环形队列的队列长度等于 store 的空间最大值，例如 store 队列长度为16384，则 store 的空间为16MB。但是，一个 event 的实际大小并不等于此值，而是由这个 event 内有多少行数据和每行数据的长度决定的，例如一张只有两列的表的 insert event 只有30字节，但 delete event 可能达到数千字节，这是因为通常 delete event 的行数比 insert event 多。

FE 配置

下面配置属于数据同步作业的系统级别配置，主要通过修改 fe.conf 来调整配置值。

`sync_commit_interval_second`

提交事务的最大时间间隔。若超过了这个时间 channel 中还有数据没有提交，consumer 会通知 channel 提交事务。

`min_sync_commit_size`

提交事务需满足的最小event数量。若Fe接收到的event数量小于它，会继续等待下一批数据直到时间超过了`sy

`min_bytes_sync_commit`

提交事务需满足的最小数据大小。若 FE 接收到的数据大小小于它，会继续等待下一批数据直到时间超过了 `sync_commit_interval_second` 为止。默认值是15MB，如果您想修改此配置，请确保此值小于 canal 端的 `canal.instance.memory.buffer.size` 和 `canal.instance.memory.buffer.memunit` 的乘积（默认16MB），否则在 ack 前 Fe 会尝试获取比 store 空间更大的数据，导致 store 队列阻塞至超时为止。

`max_bytes_sync_commit`

提交事务时的数据大小的最大值。若 Fe 接收到的数据大小大于它，会立即提交事务并发送已积累的数据。默认值是64MB，如果您想修改此配置，请确保此值大于 canal 端的 `canal.instance.memory.buffer.size` 和 `canal.instance.memory.buffer.memunit` 的乘积（默认16MB）和 `min_bytes_sync_commit` 。

`max_sync_task_threads_num`

数据同步作业线程池中的最大线程数量。此线程池整个FE中只有一个，用于处理 FE 中所有数据同步作业向 BE 发送数据的任务 task，线程池的实现在 `SyncTaskPool` 类。

常见问题

1. 修改表结构是否会影响数据同步作业？
会影响。数据同步作业并不能禁止 `alter table` 的操作，当表结构发生了变化，如果列的映射无法匹配，可能导致作业发生错误暂停，建议通过在数据同步作业中显式指定列映射关系，或者通过增加 Nullable 列或带 Default 值的列来减少这类问题。
2. 删除了数据库后数据同步作业还会继续运行吗？
不会。删除数据库后的几秒日志中可能会出现找不到元数据的错误，之后该数据同步作业会被FE的定时调度检查时停止。
3. 多个数据同步作业可以配置相同的 `ip:port + destination` 吗？
不能。创建数据同步作业时会检查 `ip:port + destination` 与已存在的作业是否重复，防止出现多个作业连接到同一个 instance 的情况。
4. 为什么数据同步时浮点类型的数据精度在 Mysql 端和 Doris 端不一样？
Doris 本身浮点类型的精度与 Mysql 不一样。可以选择用 Decimal 类型代替。

MySQL 数据批量写入

批量写入 Mysql 数据可选使用之前介绍过的几种导入方式：
1. 使用 JDBC 同步数据：通过一个 JDBC 连接获取 mysql 中的数据，另一个 JDBC 连接 Doris 批量插入从 mysql 获取的数据。
2. 通过外部表同步数据。
3. mysql 表数据导出为 csv / json 格式的本地文件后使用 Stream load 导入。
4. mysql 表数据导出为 csv / json 格式的本地文件并上传至 HDFS / S3 存储系统后使用 Broker load 导入。
5. mysql 表数据导出为 csv / json 格式的本地文件并上传至 HDFS 存储系统后使用 Spark load 导入。

注意事项

上述每种导入方式都有各自的注意事项。
1. JDBC 同步方式注意一条 sql 中插入的数据的条数；另外如果插入条数过多则不适合用于线上环境。
2. ODBC 外部表不合适一次性导入大量的数据，建议分批多次导入。
3. Stream load 建议的一次导入数据量在 1G 到 10G 之间，如果历史数据很大，建议切分后多次导入。
4. 单次 Broker load 的数据量最好不要超过3G，如果历史数据很大，建议切分后多次导入。
5. 如果数据超过3G建议直接使用 Spark load。太小的数据使用 spark load 性能稍差。

从 Kafka 导入数据

最近更新时间：2024-06-27 11:04:55

用户可以通过提交例行导入作业，直接订阅 Kafka 中的消息数据，以近实时的方式进行数据同步。
Doris 自身能够保证不丢不重的订阅 Kafka 中的消息，即 `Exactly-Once` 消费语义。

订阅 Kafka 消息

订阅 Kafka 消息使用了 Doris 中的例行导入（Routine Load）功能。
用户首先需要创建一个**例行导入作业**。作业会通过例行调度，不断地发送一系列的**任务**，每个任务会消费一定数量 Kafka 中的消息。
请注意以下使用限制：
1. 支持无认证的 Kafka 访问，以及通过 SSL 方式认证的 Kafka 集群。
2. 支持的消息格式如下：
csv 文本格式。每一个 message 为一行，且行尾**不包含**换行符。
Json 格式。
3. 仅支持 Kafka 0.10.0.0(含) 以上版本。

访问 SSL 认证的 Kafka 集群

例行导入功能支持无认证的 Kafka 集群，以及通过 SSL 认证的 Kafka 集群。
访问 SSL 认证的 Kafka 集群需要用户提供用于认证 Kafka Broker 公钥的证书文件（ca.pem）。如果 Kafka 集群同时开启了客户端认证，则还需提供客户端的公钥（client.pem）、密钥文件（client.key），以及密钥密码。这里所需的文件需要先通过 `CREAE FILE` 命令上传到 Plao 中，并且 catalog 名称为 `kafka` 。`CREATE FILE` 命令的具体帮助可以参见 CREATE FILE 命令手册。这里给出示例：

上传文件

```
CREATE FILE "ca.pem" PROPERTIES("url" = "https://example_url/kafka-key/ca.pem", "ca
CREATE FILE "client.key" PROPERTIES("url" = "https://example_urlkafka-key/client.ke
CREATE FILE "client.pem" PROPERTIES("url" = "https://example_url/kafka-key/client.p
```

上传完成后，可以通过 SHOW FILES 命令查看已上传的文件。

创建例行导入作业

创建例行导入任务的具体命令，请参阅 ROUTINE LOAD 命令手册。这里给出示例：
1. 访问无认证的 Kafka 集群。

```
CREATE ROUTINE LOAD demo.my_first_routine_load_job ON test_1
```

```
COLUMNS TERMINATED BY ","
PROPERTIES
(
 "max_batch_interval" = "20",
 "max_batch_rows" = "300000",
 "max_batch_size" = "209715200",
)
FROM KAFKA
(
 "kafka_broker_list" = "broker1:9092,broker2:9092,broker3:9092",
 "kafka_topic" = "my_topic",
 "property.group.id" = "xxx",
 "property.client.id" = "xxx",
 "property.kafka_default_offsets" = "OFFSET_BEGINNING"
);
```

`max_batch_interval/max_batch_rows/max_batch_size` 用于控制一个子任务的运行周期。一个子任务的运行周期由最长运行时间、最多消费行数和最大消费数据量共同决定。

2. 访问 SSL 认证的 Kafka 集群。

```
CREATE ROUTINE LOAD demo.my_first_routine_load_job ON test_1
COLUMNS TERMINATED BY ",",
PROPERTIES
(
  "max_batch_interval" = "20",
  "max_batch_rows" = "300000",
  "max_batch_size" = "209715200",
)
FROM KAFKA
(
 "kafka_broker_list"= "broker1:9091,broker2:9091",
 "kafka_topic" = "my_topic",
 "property.security.protocol" = "ssl",
 "property.ssl.ca.location" = "FILE:ca.pem",
 "property.ssl.certificate.location" = "FILE:client.pem",
 "property.ssl.key.location" = "FILE:client.key",
 "property.ssl.key.password" = "abcdefg"
);
```

查看导入作业状态

查看**作业**状态的具体命令和示例请参阅 SHOW ROUTINE LOAD 命令文档。

查看某个作业的**任务**运行状态的具体命令和示例请参阅 SHOW ROUTINE LOAD TASK 命令文档。

只能查看当前正在运行中的任务，已结束和未开始的任务无法查看。

修改作业属性

用户可以修改已经创建的作业的部分属性。具体说明请参阅 ALTER ROUTINE LOAD 命令手册。

作业控制

用户可以通过 `STOP/PAUSE/RESUME` 三个命令来控制作业的停止，暂停和重启。
具体命令请参阅：STOP ROUTINE LOAD、 PAUSE ROUTINE LOAD 和 RESUME ROUTINE LOAD 命令文档。

更多帮助

关于 ROUTINE LOAD 的更多详细语法和最佳实践，请参阅 ROUTINE LOAD 命令手册。

使用 DataX 导入

最近更新时间：2024-06-27 11:05:09

关于 DataX

DataX 是阿里开源的通用离线数据导入工具，在业界有比较广泛的使用。DataX 实现了包括 MySQL、Oracle、SqlServer、Postgre、HDFS、Hive、ADS、HBase、TableStore(OTS)、MaxCompute(ODPS)、Hologres、DRDS 等各种异构数据源之间高效的数据同步功能。
使用 Datax 导入数据到 Doris 需要使用 DataX doriswriter 插件，这个插件是利用 Doris 的 Stream Load 功能进行数据导入的，需要配合 DataX 服务一起使用。

使用手册

DataX doriswriter 插件代码在 这里。这里包含插件代码以及 DataX 项目的开发环境。
doriswriter 插件依赖 DataX 代码中的一些模块。而这些模块并没有在 Maven 官方仓库中。所以我们在开发 doriswriter 插件时，需要下载完整的 DataX 代码库，才能进行插件的编译和开发。

目录结构

`doriswriter/`

这个目录是 doriswriter 插件的代码目录。这个目录中的所有代码，都托管在 Apache Doris 的代码库中。doriswriter 插件帮助文档在：`doriswriter/doc`。

`init-env.sh`

这个脚本主要用于构建 DataX 开发环境，主要进行了以下操作：
1. 将 DataX 代码库 clone 到本地。
2. 将 `doriswriter/` 目录软链到 `DataX/doriswriter` 目录。
3. 在 `DataX/pom.xml` 文件中添加 `<module>doriswriter</module>` 模块。
4. 将 `DataX/core/pom.xml` 文件中的 httpclient 版本从 4.5 改为 4.5.13。

说明

httpclient v4.5 在处理 307 转发时有 bug。
这个脚本执行后，开发者就可以进入 `DataX/` 目录开始开发或编译了。因为做了软链，所以任何对 `DataX/doriswriter` 目录中文件的修改，都会反映到 `doriswriter/` 目录中，方便开发者提交代码。

编译

1. 运行 `init-env.sh`。

2. 按需修改 `DataX/doriswriter` 中的代码。

3. 编译 doriswriter：

3.1 单独编译 doriswriter 插件：

```
mvn clean install -pl plugin-rdbms-util,doriswriter -DskipTests
```

3.2 编译整个 DataX 项目:

```
mvn package assembly:assembly -Dmaven.test.skip=true
```

产出在 `target/datax/datax/` .

说明

hdfsreader，hdfswriter 和 oscarwriter 这三个插件需要额外的 jar 包。如果您并不需要这些插件，可以在 `DataX/pom.xml` 中删除这些插件的模块。

3.3 编译错误

如遇到如下编译错误：

```
Could not find artifact com.alibaba.datax:datax-all:pom:0.0.1-SNAPSHOT ...
```

可尝试以下方式解决：

3.3.1 下载 alibaba-datax-maven-m2-20210928.tar.gz。

3.3.2 解压后，将得到的 `alibaba/datax/` 目录，拷贝到所使用的 maven 对应的 `.m2/repository/com/alibaba/` 下。

3.3.3 再次尝试编译。

3.3.4 按需提交修改。

示例

Stream 读取数据后导入至 Doris

该示例插件的使用说明请参阅 这里。

Mysql 读取数据后导入Doris

1. Mysql 表结构

```
CREATE TABLE `t_test`(
 `id`bigint(30) NOT NULL,
 `order_code` varchar(30) DEFAULT NULL COMMENT '',
 `line_code` varchar(30) DEFAULT NULL COMMENT '',
 `remark` varchar(30) DEFAULT NULL COMMENT '',
 `unit_no` varchar(30) DEFAULT NULL COMMENT '',
 `unit_name` varchar(30) DEFAULT NULL COMMENT '',
 `price` decimal(12,2) DEFAULT NULL COMMENT '',
 PRIMARY KEY(`id`) USING BTREE
```

```
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 ROW_FORMAT=DYNAMIC COMMENT='';
```

2. Doris 表结构

```
CREATE TABLE `ods_t_test` (
 `id`bigint(30) NOT NULL,
 `order_code` varchar(30) DEFAULT NULL COMMENT '',
 `line_code` varchar(30) DEFAULT NULL COMMENT '',
 `remark` varchar(30) DEFAULT NULL COMMENT '',
 `unit_no` varchar(30) DEFAULT NULL COMMENT '',
 `unit_name` varchar(30) DEFAULT NULL COMMENT '',
 `price` decimal(12,2) DEFAULT NULL COMMENT ''
)ENGINE=OLAP
UNIQUE KEY(`id`, `order_code`)
DISTRIBUTED BY HASH(`order_code`) BUCKETS 1
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false",
"storage_format" = "V2"
);
```

3. 创建 datax 脚本 `import_t_test.json`

```
{
 "job": {
     "setting": {
         "speed": {
             "channel": 1
         },
         "errorLimit": {
             "record": 0,
             "percentage": 0
         }
     },
     "content": [
         {
             "reader": {
                 "name": "mysqlreader",
                 "parameter": {
                     "username": "xxx",
                     "password": "xxx",
                     "column": ["id","order_code","line_code","remark","unit_no","u
                     "connection": [ { "table": [ "t_test" ], "jdbcUrl": [ "jdbc:my
             },
             "writer": {
                 "name": "doriswriter",
                 "parameter": {
```

```
                    "feLoadUrl": ["127.0.0.1:8030","127.0.0.2:8030"],
                    "beLoadUrl": ["127.0.0.3:8040","127.0.0.4:8040","127.0.0.5:804
                    "jdbcUrl": "jdbc:mysql://127.0.0.1:9030/",
                    "database": "demo",
                    "table": "ods_t_test",
                    "column": ["id","order_code","line_code","remark","unit_no","u
                    "username": "xxx",
                    "password": "xxx",
                    "postSql": [],
                    "preSql": [],
                    "loadProps": {
                    },
                    "maxBatchRows" : 300000,
                    "maxBatchByteSize" : 20971520
                }
            }
        }
    ]
  }
}
```

4. 执行 datax 任务：python datax.py import_t_test.json。具体用法参考 datax 官网。

从 Logstash 导入 Doris

最近更新时间：2024-06-27 11:05:28

从 Logstash 导入 Doris 需要使用 Logstash的Doris output plugin。该插件使用 HTTP 协议与 Doris FE HTTP 接口交互，并通过 Doris 的 stream load 的方式进行数据导入。

安装和编译

1. 下载源码

插件源码在 Doris 源码中，下载 Doris 源码。

2. 编译

在 doris 源码的 extension/logstash/ 目录下执行：

```
gem build logstash-output-doris.gemspec
```

您将在同目录下得到 logstash-output-doris-{version}.gem 文件。

3. 插件安装

`copy logstash-output-doris-{version}.gem` 到 logstash 安装目录下，执行命令：

```
./bin/logstash-plugin install logstash-output-doris-{version}.gem
```

安装 logstash-output-doris 插件。

配置

示例

在 config 目录下新建一个配置文件，命名为 logstash-doris.conf，具体配置如下：

```
    output {
        doris {
            http_hosts => [ "http://fehost:8030" ]
            user => user_name
            password => password
            db => "db_name"
            table => "table_name"
            label_prefix => "label_prefix"
            column_separator => ","
```

```
        }
    }
```

配置说明

连接相关配置：

配置	说明
`http_hosts`	FE的HTTP交互地址。 例如： ["http://fe1:8030", "http://fe2:8030"]
`user`	用户名，该用户需要有 doris 对应库表的导入权限
`password`	密码
`db`	数据库名
`table`	表名
`label_prefix`	导入标识前缀，最终生成的标识为 *{label*_prefix}_{db}_{table}_{time_stamp}

导入相关配置参考 Stream Load 手册。

配置	说明
`column_separator`	列分割符，默认为\\t。
`columns`	用于指定导入文件中的列和 table 中的列的对应关系。
`where`	导入任务指定的过滤条件。
`max_filter_ratio`	导入任务的最大容忍率，默认零容忍。
`partition`	待导入表的 Partition 信息。
`timeout`	超时时间，默认为600s。
`strict_mode`	严格模式，默认为 false。
`timezone`	指定本次导入所使用的时区，默认为东八区。
`exec_mem_limit`	导入内存限制，默认为 2GB，单位为字节。

其他配置：

配置	说明
`save_on_failure`	如果导入失败是否在本地保存，默认为 true

`save_dir`	本地保存目录，默认为 /tmp
`automatic_retries`	失败时重试最大次数，默认为3
`batch_size`	每批次最多处理的 event 数量，默认为100000
`idle_flush_time`	最大间隔时间，默认为20（秒）

启动

执行命令启动 doris output plugin：

```
{logstash-home}/bin/logstash -f {logstash-home}/config/logstash-doris.conf --config
```

完整使用示例

1. 编译 doris-output-plugin

1. 下载 ruby 压缩包，自行到 ruby 官网下载，这里使用的2.7.1版本。
2. 编译安装，配置 ruby 的环境变量.
3. 到 doris 源码 extension/logstash/ 目录下，执行：

```
gem build logstash-output-doris.gemspec
```

得到文件 logstash-output-doris-0.1.0.gem，至此编译完成。

2. 安装配置 filebeat

说明

此处使用 filebeat 作为 input。

1. es 官网 下载 filebeat tar 压缩包并解压。
2. 进入 filebeat 目录下，修改配置文件 filebeat.yml 如下：

```
filebeat.inputs:
- type: log
  paths:
    - /tmp/doris.data
output.logstash:
  hosts: ["localhost:5044"]
```

3. 启动 filebeat：

```
./filebeat -e -c filebeat.yml -d "publish"
```

3. 安装 logstash 及 doris-out-plugin

1. 从 es官网 下载 logstash tar 压缩包并解压。
2. 将步骤1中得到的 logstash-output-doris-0.1.0.gem copy 到 logstash 安装目录下。
3. 执行：

```
./bin/logstash-plugin install logstash-output-doris-0.1.0.gem
```

安装插件。

4. 在 config 目录下新建配置文件 logstash-doris.conf 内容如下：

```
input {
 beats {
     port => "5044"
 }
}

output {
 doris {
     http_hosts => [ "http://127.0.0.1:8030" ]
     user => doris
     password => doris
     db => "logstash_output_test"
     table => "output"
     label_prefix => "doris"
     column_separator => ","
     columns => "a,b,c,d,e"
 }
}
```

这里的配置需按照配置说明自行配置。

5. 启动 logstash：

```
./bin/logstash -f ./config/logstash-doris.conf --config.reload.automatic
```

4. 测试功能

向 /tmp/doris.data 追加写入数据：

```
echo a,b,c,d,e >> /tmp/doris.data
```

观察 logstash 日志，若返回 response的Status 为 Success，则导入成功，此时可在 logstash_output_test.output 表中查看已导入的数据。

导入的数据转换、列映射及过滤

最近更新时间：2024-06-27 11:05:44

支持的导入方式

Broker Load

```
LOAD LABEL example_db.label1
(
  DATA INFILE("bos://bucket/input/file")
  INTO TABLE `my_table`
  (k1, k2, tmpk3)
  PRECEDING FILTER k1 = 1
  SET (
          k3 = tmpk3 + 1
  )
  WHERE k1 > k2
)
WITH BROKER bos
(
  ...
);
```

Stream load

```
curl
--location-trusted
-u user:passwd
-H "columns: k1, k2, tmpk3, k3 = tmpk3 + 1"
-H "where: k1 > k2"
-T file.txt
http://host:port/api/testDb/testTbl/_stream_load
```

Routine Load

```
CREATE ROUTINE LOAD example_db.label1 ON my_table
COLUMNS(k1, k2, tmpk3, k3 = tmpk3 + 1),
PRECEDING FILTER k1 = 1,
WHERE k1 > k2
...
```

以上导入方式都支持对源数据进行列映射、转换和过滤操作：

前置过滤：对读取到的原始数据进行一次过滤。

```
PRECEDING FILTER k1 = 1
```

映射：定义源数据中的列。如果定义的列名和表中的列相同，则直接映射为表中的列。如果不同，则这个被定义的列可以用于之后的转换操作。如上面示例中的：

```
(k1, k2, tmpk3)
```

转换：将第一步中经过映射的列进行转换，可以使用内置表达式、函数、自定义函数进行转化，并重新映射到表中对应的列上。如上面示例中的：

```
k3 = tmpk3 + 1
```

后置过滤：对经过映射和转换后的列，通过表达式进行过滤。被过滤的数据行不会导入到系统中。如上面示例中的：

```
WHERE k1 > k2
```

列映射

列映射的目的主要是描述导入文件中各个列的信息，相当于为源数据中的列定义名称。通过描述列映射关系，我们可以将表中列顺序不同、列数量不同的源文件导入到 Doris 中。下面我们通过示例说明：
假设源文件有4列，内容如下（表头列名仅为方便表述，实际并无表头）：

列1	列2	列3	列4
1	100	beijing	1.1
2	200	shanghai	1.2
3	300	guangzhou	1.3
4	\\N	chongqing	1.4

说明

`\\N` 在源文件中表示 null。

1. 调整映射顺序

假设表中有 `k1,k2,k3,k4` 4列。我们希望的导入映射关系如下：

```
列1 -> k1
列2 -> k3
列3 -> k2
列4 -> k4
```

则列映射的书写顺序应如下：

```
(k1, k3, k2, k4)
```

2. 源文件中的列数量多于表中的列

假设表中有 `k1,k2,k3` 3列。我们希望的导入映射关系如下：

```
列1 -> k1
列2 -> k3
列3 -> k2
```

则列映射的书写顺序应如下：

```
(k1, k3, k2, tmpk4)
```

其中 `tmpk4` 为一个自定义的、表中不存在的列名。Doris 会忽略这个不存在的列名。

3. 源文件中的列数量少于表中的列，使用默认值填充

假设表中有 `k1,k2,k3,k4,k5` 5列。我们希望的导入映射关系如下：

```
列1 -> k1
列2 -> k3
列3 -> k2
```

这里我们仅使用源文件中的前3列。 `k4,k5` 两列希望使用默认值填充。

则列映射的书写顺序应如下：

```
(k1, k3, k2)
```

如果 `k4,k5` 列有默认值，则会填充默认值。否则如果是 `nullable` 的列，则会填充 `null` 值。否则，导入作业会报错。

列前置过滤

前置过滤是对读取到的原始数据进行一次过滤。目前仅支持 BROKER LOAD 和 ROUTINE LOAD。

前置过滤有以下应用场景：

1. 转换前做过滤。

希望在列映射和转换前做过滤的场景。能够先行过滤掉部分不需要的数据。

2. 过滤列不存在于表中，仅作为过滤标识。

例如源数据中存储了多张表的数据（或者多张表的数据写入了同一个 Kafka 消息队列）。数据中每行有一列表名来标识该行数据属于哪个表。用户可以通过前置过滤条件来筛选对应的表数据进行导入。

列转换

列转换功能允许用户对源文件中列值进行变换。目前 Doris 支持使用绝大部分内置函数、用户自定义函数进行转换。

说明

自定义函数隶属于某一数据库下，在使用自定义函数进行转换时，需要用户对这个数据库有读权限。

转换操作通常是和列映射一起定义的。即先对列进行映射，再进行转换。下面我们通过示例说明：

假设源文件有4列，内容如下（表头列名仅为方便表述，实际并无表头）：

列1	列2	列3	列4
1	100	beijing	1.1
2	200	shanghai	1.2
3	300	guangzhou	1.3
4	400	chongqing	1.4

1. 将源文件中的列值经转换后导入表中。

假设表中有 `k1,k2,k3,k4` 4列。我们希望的导入映射和转换关系如下：

```
列1       -> k1
列2 * 100 -> k3
列3       -> k2
列4       -> k4
```

则列映射的书写顺序应如下：

```
(k1, tmpk3, k2, k4, k3 = tmpk3 * 100)
```

这里相当于我们将源文件中的第2列命名为 `tmpk3` ，同时指定表中 `k3` 列的值为 `tmpk3 * 100` 。最终表中的数据如下：

k1	k2	k3	k4
1	beijing	10000	1.1
2	shanghai	20000	1.2
3	guangzhou	30000	1.3
null	chongqing	40000	1.4

2. 通过 case when 函数，有条件的进行列转换。

假设表中有 `k1,k2,k3,k4` 4列。我们希望对于源数据中的 `beijing, shanghai, guangzhou,`

`chongqing` 分别转换为对应的地区id后导入：

```
列1                    -> k1
列2                    -> k2
列3 进行地区id转换后     -> k3
列4                    -> k4
```

则列映射的书写顺序应如下：

```
(k1, k2, tmpk3, k4, k3 = case tmpk3 when "beijing" then 1 when "shanghai" then 2 wh
```

最终表中的数据如下：

k1	k2	k3	k4
1	100	1	1.1
2	200	2	1.2
3	300	3	1.3
null	400	4	1.4

3. 将源文件中的 null 值转换成 0 导入。同时也进行示例2中的地区 ID 转换。

假设表中有 `k1,k2,k3,k4` 4列。在对地区id转换的同时，我们也希望对于源数据中 k1 列的 null 值转换成 0 导入：

```
列1 如果为null 则转换成0    -> k1
列2                        -> k2
列3                        -> k3
列4                        -> k4
```

则列映射的书写顺序应如下：

```
(tmpk1, k2, tmpk3, k4, k1 = ifnull(tmpk1, 0), k3 = case tmpk3 when "beijing" then 1
```

最终表中的数据如下：

k1	k2	k3	k4
1	100	1	1.1
2	200	2	1.2
3	300	3	1.3
0	400	4	1.4

列过滤

经过列映射和转换后，我们可以通过过滤条件将不希望导入到Doris中的数据进行过滤。下面我们通过示例说明：

假设源文件有4列，内容如下（表头列名仅为方便表述，实际并无表头）：

列1	列2	列3	列4
1	100	beijing	1.1
2	200	shanghai	1.2
3	300	guangzhou	1.3
4	400	chongqing	1.4

1. 在列映射和转换缺省的情况下，直接过滤。

假设表中有 `k1,k2,k3,k4` 4列。我们可以在缺省列映射和转换的情况下，直接定义过滤条件。如我们希望只导入源文件中第4列为大于 1.2 的数据行，则过滤条件如下：

```
where k4 > 1.2
```

最终表中的数据如下：

k1	k2	k3	k4
3	300	guangzhou	1.3
null	400	chongqing	1.4

缺省情况下，Doris 会按照顺序进行列映射，因此源文件中的第4列自动被映射到表中的 `k4` 列。

2. 对经过列转换的数据进行过滤。

假设表中有 `k1,k2,k3,k4` 4列。在 **列转换** 示例中，我们将省份名称转换成了id。这里我们想过滤掉 id 为 3 的数据。则转换、过滤条件如下：

```
(k1, k2, tmpk3, k4, k3 = case tmpk3 when "beijing" then 1 when "shanghai" then 2 wh
where k3 != 3
```

最终表中的数据如下：

k1	k2	k3	k4
1	100	1	1.1
2	200	2	1.2
null	400	4	1.4

这里我们看到，执行过滤时的列值，为经过映射和转换后的最终列值，而不是原始数据。

3. 多条件过滤

假设表中有 `k1,k2,k3,k4` 4列。我们想过滤掉 `k1` 列为 `null` 的数据，同时过滤掉 `k4` 列小于 1.2 的数据，则过滤条件如下：

```
where k1 is null and k4 < 1.2
```

最终表中的数据如下：

k1	k2	k3	k4
2	200	2	1.2
3	300	3	1.3

数据质量问题和过滤阈值

导入作业中被处理的数据行可以分为如下三种：

1. Filtered Rows

因数据质量不合格而被过滤掉的数据。数据质量不合格包括类型错误、精度错误、字符串长度超长、文件列数不匹配等数据格式问题，以及因没有对应的分区而被过滤掉的数据行。

2. Unselected Rows

这部分为因 `preceding filter` 或 `where` 列过滤条件而被过滤掉的数据行。

3. Loaded Rows

被正确导入的数据行。

Doris 的导入任务允许用户设置最大错误率（`max_filter_ratio`）。如果导入的数据的错误率低于阈值，则这些错误行将被忽略，其他正确的数据将被导入。

错误率的计算方式为：

```
#Filtered Rows / (#Filtered Rows + #Loaded Rows)
```

也就是说 `Unselected Rows` 不会参与错误率的计算。

导入的严格模式

最近更新时间：2024-06-27 11:06:07

严格模式（strict_mode）为导入操作中的一个参数配置。该参数会影响某些数值的导入行为和最终导入的数据。
本文档主要说明如何设置严格模式，以及严格模式产生的影响。

如何设置

严格模式默认情况下都为 False，即关闭状态。
不同的导入方式设置严格模式的方式不尽相同。

1. Broker Load（HDFS 数据）

```
LOAD LABEL example_db.label1
(
  DATA INFILE("bos://my_bucket/input/file.txt")
  INTO TABLE `my_table`
  COLUMNS TERMINATED BY ","
)
WITH BROKER bos
(
  "bos_endpoint" = "http://bj.bcebos.com",
  "bos_accesskey" = "xxxxxxxxxxxxxxxxxxxxxxxxxxx",
  "bos_secret_accesskey"="yyyyyyyyyyyyyyyyyyyyyyyyyy"
)
PROPERTIES
(
  "strict_mode" = "true"
)
```

2. Stream load（本地文件）

```
curl --location-trusted -u user:passwd \\
-H "strict_mode: true" \\
-T 1.txt \\
http://host:port/api/example_db/my_table/_stream_load
```

3. Routine Load（Kafka 数据）

```
CREATE ROUTINE LOAD example_db.test_job ON my_table
PROPERTIES
(
  "strict_mode" = "true"
)
```

```
FROM KAFKA
(
  "kafka_broker_list" = "broker1:9092,broker2:9092,broker3:9092",
  "kafka_topic" = "my_topic"
);
```

4. INSERT INTO

通过 变量（variables） 设置：

```
SET enable_insert_strict = true;
INSERT INTO my_table ...;
```

严格模式的作用

严格模式的意思是，对于导入过程中的列类型转换进行严格过滤。严格过滤的策略如下：

对于列类型转换来说，如果开启严格模式，则错误的数据将被过滤。这里的错误数据是指：原始数据并不为 `null`，而在进行列类型转换后结果为 `null` 的这一类数据。

这里所指的 `列类型转换`，并不包括用函数计算得出的 `null` 值。

对于导入的某列类型包含范围限制的，如果原始数据能正常通过类型转换，但无法通过范围限制的，严格模式对其也不产生影响。例如：如果类型是 `decimal(1,0)`，原始数据为 10，则属于可以通过类型转换但不在列声明的范围内。这种数据 strict 对其不产生影响。

以列类型为 TinyInt 来举例：

原始数据类型	原始数据举例	转换为 TinyInt 后的值	严格模式	结果
空值	\\N	NULL	开启或关闭	NULL
非空值	"abc" or 2000	NULL	开启	非法值（被过滤）
非空值	"abc"	NULL	关闭	NULL
非空值	1	1	开启或关闭	正确导入

说明

表中的列允许导入空值。

`abc` 及 `2000` 在转换为 TinyInt 后，会因类型或精度问题变为 NULL。在严格模式开启的情况下，这类数据将会被过滤。而如果是关闭状态，则会导入 `null`。

以列类型为 Decimal(1,0) 举例：

原始数据类型	原始数据举例	转换为 Decimal 后的值	严格模式	结果
空值	\\N	null	开启或关闭	NULL

非空值	aaa	NULL	开启	非法值（被过滤）
非空值	aaa	NULL	关闭	NULL
非空值	1 or 10	1 or 10	开启或关闭	正确导入

说明

表中的列允许导入空值。

`abc` 在转换为 Decimal 后，会因类型问题变为 NULL。在严格模式开启的情况下，这类数据将会被过滤。而如果是关闭状态，则会导入 `null` 。

`10` 虽然是一个超过范围的值，但是因为其类型符合 decimal 的要求，所以严格模式对其不产生影响。 `10` 最后会在其他导入处理流程中被过滤。但不会被严格模式过滤。

导入 Json 格式数据

最近更新时间：2024-06-27 11:06:22

Doris 从0.12版本开始支持 Json 格式的数据导入。

支持的导入方式

目前只有以下导入方式支持 Json 格式的数据导入：

Stream Load

Routine Load

关于以上导入方式的具体说明，请参阅相关文档。本文档主要介绍在这些导入方式中关于 Json 部分的使用说明。

支持的 Json 格式

当前仅支持以下两种 Json 格式：

1. 以 Array 表示的多行数据。

以 Array 为根节点的 Json 格式。Array 中的每个元素表示要导入的一行数据，通常是一个 Object。示例如下：

```
[
    { "id": 123, "city" : "beijing"},
    { "id": 456, "city" : "shanghai"},
    ...
]
```

```
[
    { "id": 123, "city" : { "name" : "beijing", "region" : "haidian"}},
    { "id": 456, "city" : { "name" : "beijing", "region" : "chaoyang"}},
    ...
]
```

这种方式通常用于 Stream Load 导入方式，以便在一批导入数据中表示多行数据。

这种方式必须配合设置 `strip_outer_array=true` 使用。Doris 在解析时会将数组展开，然后依次解析其中的每一个 Object 作为一行数据。

2. 以 Object 表示的单行数据。

以 Object 为根节点的 Json 格式。整个 Object 即表示要导入的一行数据。示例如下：

```
{ "id": 123, "city" : "beijing"}
```

```
{ "id": 123, "city" : { "name" : "beijing", "region" : "haidian" }}
```

这种方式通常用于 Routine Load 导入方式，如表示 Kafka 中的一条消息，即一行数据。

3. 以固定分隔符分隔的多行 Object 数据。

Object 表示的一行数据即表示要导入的一行数据，示例如下：

```
{ "id": 123, "city" : "beijing"}
{ "id": 456, "city" : "shanghai"}
...
```

这种方式通常用于 Stream Load 导入方式，以便在一批导入数据中表示多行数据。

这种方式必须配合设置 `read_json_by_line=true` 使用，特殊分隔符还需要指定 `line_delimiter` 参数，默认 `\\n` 。Doris 在解析时会按照分隔符分隔，然后解析其中的每一行 Object 作为一行数据。

fuzzy_parse 参数

在 Stream load 中，可以添加 `fuzzy_parse` 参数来加速 JSON 数据的导入效率。

这个参数通常用于导入 **以 Array 表示的多行数据** 这种格式，所以一般要配合 `strip_outer_array=true` 使用。

这个功能要求 Array 中的每行数据的**字段顺序完全一致**。Doris 仅会根据第一行的字段顺序做解析，然后以下标的形式访问之后的数据。该方式可以提升 3-5X 的导入效率。

Json Path

Doris 支持通过 Json Path 抽取 Json 中指定的数据。

注意

因为对于 Array 类型的数据，Doris 会先进行数组展开，最终按照 Object 格式进行单行处理。所以本文档之后的示例都以单个 Object 格式的 Json 数据进行说明。

不指定 Json Path。

如果没有指定 Json Path，则 Doris 会默认使用表中的列名查找 Object 中的元素。示例如下：

表中包含两列: `id` , `city` 。

Json 数据如下：

```
{ "id": 123, "city" : "beijing"}
```

则 Doris 会使用 `id` , `city` 进行匹配，得到最终数据 `123` 和 `beijing` 。

如果 Json 数据如下：

```
{ "id": 123, "name" : "beijing"}
```

则使用 `id` , `city` 进行匹配，得到最终数据 `123` 和 `null` 。

指定 Json Path

通过一个 Json 数据的形式指定一组 Json Path。数组中的每个元素表示一个要抽取的列。示例如下：

```
["$.id", "$.name"]
```

```
["$.id.sub_id", "$.name[0]", "$.city[0]"]
```

Doris 会使用指定的 Json Path 进行数据匹配和抽取。

匹配非基本类型

前面的示例最终匹配到的数值都是基本类型，如整型、字符串等。Doris 当前暂不支持复合类型，如 Array、Map 等。所以当匹配到一个非基本类型时，Doris 会将该类型转换为 Json 格式的字符串，并以字符串类型进行导入。示例如下：

Json 数据为：

```
{ "id": 123, "city" : { "name" : "beijing", "region" : "haidian" }}
```

Json Path 为 `["$.city"]` 。则匹配到的元素为：

```
{ "name" : "beijing", "region" : "haidian" }
```

该元素会被转换为字符串进行后续导入操作：

```
"{'name':'beijing','region':'haidian'}"
```

匹配失败

当匹配失败时，将会返回 `null` 。示例如下：

Json 数据为：

```
{ "id": 123, "name" : "beijing"}
```

Json Path 为 `["$.id", "$.info"]` 。则匹配到的元素为 `123` 和 `null` 。

Doris 当前不区分 Json 数据中表示的 null 值，和匹配失败时产生的 null 值。假设 Json 数据为：

```
{ "id": 123, "name" : null }
```

则使用以下两种 Json Path 会获得相同的结果： `123` 和 `null` 。

```
["$.id", "$.name"]
```

```
["$.id", "$.info"]
```

完全匹配失败

为防止一些参数设置错误导致的误操作。Doris 在尝试匹配一行数据时，如果所有列都匹配失败，则会认为这个是一个错误行。假设 Json 数据为：

```
{ "id": 123, "city" : "beijing" }
```

如果 Json Path 错误的写为（或者不指定 Json Path 时，表中的列不包含 `id` 和 `city` ）：

```
["$.ad", "$.infa"]
```

则会导致完全匹配失败，则该行会标记为错误行，而不是产出 `null, null` 。

Json Path 和 Columns

Json Path 用于指定如何对 JSON 格式中的数据进行抽取，而 Columns 指定列的映射和转换关系。两者可以配合使用。

换句话说，相当于通过 Json Path，将一个 Json 格式的数据，按照 Json Path 中指定的列顺序进行了列的重排。之后，可以通过 Columns，将这个重排后的源数据和表的列进行映射。举例如下：

数据内容：

```
{"k1" : 1, "k2": 2}
```

表结构：

`k2 int, k1 int`

导入语句1（以 Stream Load 为例）：

```
curl -v --location-trusted -u root: -H "format: json" -H "jsonpaths: [\\"$.k2\\", \
```

导入语句1中，仅指定了 Json Path，没有指定 Columns。其中 Json Path 的作用是将 Json 数据按照 Json Path 中字段的顺序进行抽取，之后会按照表结构的顺序进行写入。最终导入的数据结果如下：

```
+------+------+
| k1   | k2   |
+------+------+
|    2 |    1 |
+------+------+
```

会看到，实际的 k1 列导入了 Json 数据中的 "k2" 列的值。这是因为，Json 中字段名称并不等同于表结构中字段的名称。我们需要显式的指定这两者之间的映射关系。

导入语句2：

```
curl -v --location-trusted -u root: -H "format: json" -H "jsonpaths: [\\"$.k2\\", \
```

相比于导入语句1，这里增加了 Columns 字段，用于描述列的映射关系，按 `k2, k1` 的顺序。即按Json Path 中字段的顺序抽取后，指定第一列为表中 k2 列的值，而第二列为表中 k1 列的值。最终导入的数据结果如下：

```
+------+------+
```

```
| k1   | k2   |
+------+------+
|    1 |    2 |
+------+------+
```

当然，如其他导入一样，可以在 Columns 中进行列的转换操作。示例如下：

```
curl -v --location-trusted -u root: -H "format: json" -H "jsonpaths: [\\"$.k2\\", \
```

上述示例会将 k1 的值乘以 100 后导入。最终导入的数据结果如下：

```
+------+------+
| k1   | k2   |
+------+------+
|  100 |    2 |
+------+------+
```

NULL 和 Default 值

示例数据如下：

```
[
    {"k1": 1, "k2": "a"},
    {"k1": 2},
    {"k1": 3, "k2": "c"},
]
```

表结构为：`k1 int null, k2 varchar(32) null default "x"`

导入语句如下：

```
curl -v --location-trusted -u root: -H "format: json" -H "strip_outer_array: true"
```

用户可能期望的导入结果如下，即对于缺失的列，填写默认值。

```
+------+------+
| k1   | k2   |
+------+------+
|    1 |    a |
+------+------+
|    2 |    x |
+------+------+
|    3 |    c |
+------+------+
```

但实际的导入结果如下，即对于缺失的列，补上了 NULL。

```
+------+------+
| k1   | k2   |
+------+------+
|    1 |    a |
+------+------+
|    2 | NULL |
+------+------+
|    3 |    c |
+------+------+
```

这是因为通过导入语句中的信息，Doris 并不知道 “缺失的列是表中的 k2 列”。

如果要对以上数据按照期望结果导入，则导入语句如下：

```
curl -v --location-trusted -u root: -H "format: json" -H "strip_outer_array: true"
```

LargeInt 与 Decimal

Doris支持LargeInt与Decimal等数据范围更大，数据精度更高的数据类型。但是由于Doris使用的Rapid Json库对于数字类型能够解析的最大范围为Int64与Double，这导致了通过Json导入LargeInt或Decimal时可能会出现：精度丢失，数据转换出错等问题。

示例数据如下：

```
[
    {"k1": 1, "k2":9999999999999.999999 }
]
```

导入 k2列类型为 `Decimal(16, 9)` ，数据为: `9999999999999.999999` 。在进行 Json 导入时，由于 Double 转换的精度丢失导致了导入的数据为： `10000000000000.0002` ，引发了导入出错。

为了解决这个问题，Doris 在导入时提供了 `num_as_string` 的开关。Doris 在解析 Json 数据时会将数字类型转为字符串，然后在确保不会出现精度丢失的情况下进行导入。

```
curl -v --location-trusted -u root: -H "format: json" -H "num_as_string: true" -T e
```

但是开启这个开关会引起一些意想不到的副作用。Doris 当前暂不支持复合类型，如 Array、Map 等。所以当匹配到一个非基本类型时，Doris 会将该类型转换为 Json 格式的字符串，而 `num_as_string` 会同样将复合类型的数字转换为字符串，举个例子：

Json 数据为： `{ "id": 123, "city" : { "name" : "beijing", "city_id" : 1 }}`

不开启 `num_as_string` 时，导入的city列的数据为: `{ "name" : "beijing", "city_id" : 1 }`

而开启了 `num_as_string` 时，导入的city列的数据为: `{ "name" : "beijing", "city_id" : "1" }`

注意

这里导致了复合类型原先为1的数字类型的 city_id 被作为字符串列处理并添加上了引号，与原始数据相比，产生了变化。
所以在使用 Json 导入时，要尽量避免 LargeInt 与 Decimal 与复合类型的同时导入。如果无法避免，则需要充分了解开启 `num_as_string` 后对复合类型导入的**副作用**。

应用示例

Stream Load

因为 Json 格式的不可拆分特性，所以在使用 Stream Load 导入 Json 格式的文件时，文件内容会被全部加载到内存后，才开始处理。因此，如果文件过大的话，可能会占用较多的内存。
假设表结构为：

```
id      INT     NOT NULL,
city    VARHCAR NULL,
code    INT     NULL
```

1. 导入单行数据1:

```
{"id": 100, "city": "beijing", "code" : 1}
```

不指定 Json Path：

```
curl --location-trusted -u user:passwd -H "format: json" -T data.json http://localh
```

导入结果：

```
100     beijing     1
```

指定 Json Path：

```
curl --location-trusted -u user:passwd -H "format: json" -H "jsonpaths: [\\"$.id\\"
```

导入结果：

```
100     beijing     1
```

2. 导入单行数据2：

```
{"id": 100, "content": {"city": "beijing", "code" : 1}}
```

指定 Json Path：

```
curl --location-trusted -u user:passwd -H "format: json" -H "jsonpaths: [\\"$.id\\"
```

导入结果：

```
100     beijing     1
```

3. 导入多行数据：

```
[
{"id": 100, "city": "beijing", "code" : 1},
{"id": 101, "city": "shanghai"},
{"id": 102, "city": "tianjin", "code" : 3},
{"id": 103, "city": "chongqing", "code" : 4},
{"id": 104, "city": ["zhejiang", "guangzhou"], "code" : 5},
{
     "id": 105,
     "city": {
             "order1": ["guangzhou"]
     },
     "code" : 6
}
]
```

指定 Json Path：

```
curl --location-trusted -u user:passwd -H "format: json" -H "jsonpaths: [\\"$.id\\"
```

导入结果：

```
100     beijing                     1
101     shanghai                    NULL
102     tianjin                     3
103     chongqing                   4
104     ["zhejiang","guangzhou"]    5
105     {"order1":["guangzhou"]}    6
```

4. 对导入数据进行转换

数据依然是示例3中的多行数据，现需要对导入数据中的 `code` 列加1后导入。

```
curl --location-trusted -u user:passwd -H "format: json" -H "jsonpaths: [\\"$.id\\"
```

导入结果：

```
100     beijing                     2
101     shanghai                    NULL
102     tianjin                     4
103     chongqing                   5
104     ["zhejiang","guangzhou"]    6
105     {"order1":["guangzhou"]}    7
```

Routine Load

Routine Load 对 Json 数据的处理原理和 Stream Load 相同。在此不再赘述。

对于 Kafka 数据源，每个 Massage 中的内容被视作一个完整的 Json 数据。如果一个 Massage 中是以 Array 格式的表示的多行数据，则会导入多行，而 Kafka 的 offset 只会增加 1。而如果一个 Array 格式的 Json 表示多行数据，但是因为 Json 格式错误导致解析 Json 失败，则错误行只会增加 1（因为解析失败，实际上 Doris 无法判断其中包含多少行数据，只能按一行错误数据记录）。

数据导出
通过 EXPORT 语句导出

最近更新时间：2024-06-27 11:06:41

数据导出（Export）是 Doris 提供的一种将数据导出的功能。该功能可以将用户指定的表或分区的数据，以文本的格式，通过 Broker 进程导出到远端存储上，如 HDFS/BOS 等。本文档主要介绍 Export 的基本原理、使用方式、最佳实践以及注意事项。

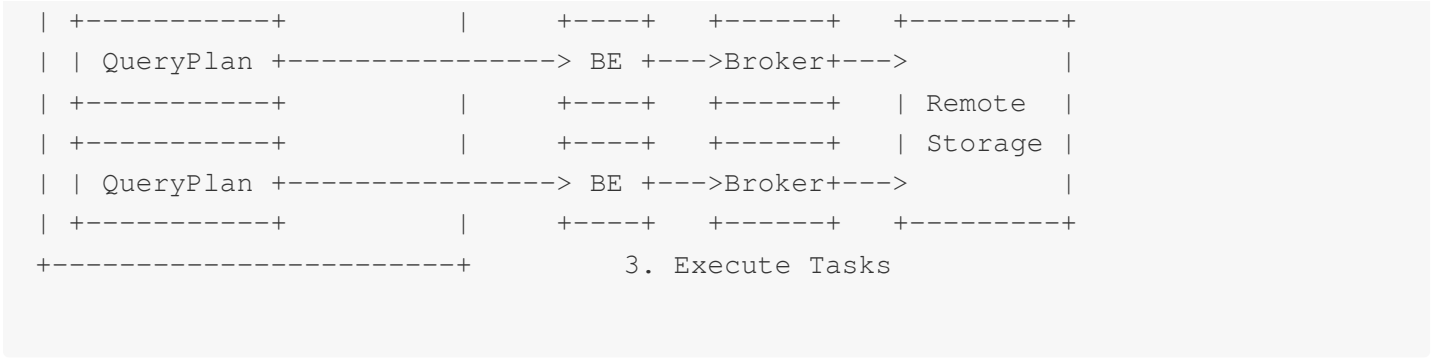
名词解释

FE：Frontend，Doris 的前端节点。负责元数据管理和请求接入。
BE：Backend，Doris 的后端节点。负责查询执行和数据存储。
Broker：Doris 可以通过 Broker 进程对远端存储进行文件操作。
Tablet：数据分片。一个表会划分成多个数据分片。

原理

用户提交一个 Export 作业后。Doris 会统计这个作业涉及的所有 Tablet。然后对这些 Tablet 进行分组，每组生成一个特殊的查询计划。该查询计划会读取所包含的 Tablet 上的数据，然后通过 Broker 将数据写到远端存储指定的路径中，也可以通过S3协议直接导出到支持S3协议的远端存储上。
总体的调度方式如下：

```
+--------+
| Client |
+---+----+
    |  1. Submit Job
    |
+---v--------------------+
| FE                     |
|                        |
| +--------------------+ |
| | ExportPendingTask | |
| +--------------------+ |
|                        | 2. Generate Tasks
| +--------------------+ |
| | ExportExporingTask | |
| +--------------------+ |
|                        |
```



1. 用户提交一个 Export 作业到 FE。

2. FE 的 Export 调度器会通过两阶段来执行一个 Export 作业：

PENDING：FE 生成 ExportPendingTask，向 BE 发送 snapshot 命令，对所有涉及到的 Tablet 做一个快照。并生成多个查询计划。

EXPORTING：FE 生成 ExportExportingTask，开始执行查询计划。

查询计划拆分

Export 作业会生成多个查询计划，每个查询计划负责扫描一部分 Tablet。每个查询计划扫描的 Tablet 个数由 FE 配置参数 `export_tablet_num_per_task` 指定，默认为 5。即假设一共 100 个 Tablet，则会生成 20 个查询计划。用户也可以在提交作业时，通过作业属性 `tablet_num_per_task` 指定这个数值。

一个作业的多个查询计划顺序执行。

查询计划执行

一个查询计划扫描多个分片，将读取的数据以行的形式组织，每 1024 行为一个 batch，调用 Broker 写入到远端存储上。

查询计划遇到错误会整体自动重试 3 次。如果一个查询计划重试 3 次依然失败，则整个作业失败。

Doris 会首先在指定的远端存储的路径中，建立一个名为 `__doris_export_tmp_12345` 的临时目录（其中 `12345` 为作业 id）。导出的数据首先会写入这个临时目录。每个查询计划会- 生成一个文件，文件名示例： `export-data-c69fcf2b6db5420f-a96b94c1ff8bccef-1561453713822` 。其中 `c69fcf2b6db5420f-a96b94c1ff8bccef` 为查询计划的 query id。 `1561453713822` 为文件生成的时间戳。

当所有数据都导出后，Doris 会将这些文件 rename 到用户指定的路径中。

Broker 参数

Export 需要借助 Broker 进程访问远端存储，不同的 Broker 需要提供不同的参数。

开始导出

导出到 HDFS

```
EXPORT TABLE db1.tbl1
PARTITION (p1,p2)
[WHERE [expr]]
TO "hdfs://host/path/to/export/"
PROPERTIES
(
    "label" = "mylabel",
    "column_separator"=",",
    "columns" = "col1,col2",
    "exec_mem_limit"="2147483648",
    "timeout" = "3600"
)
WITH BROKER "hdfs"
(
    "username" = "user",
    "password" = "passwd"
);
```

`label` ：本次导出作业的标识。后续可以使用这个标识查看作业状态。

`column_separator` ：列分隔符。默认为 `\\t` 。支持不可见字符，例如 '\\x07'。

`columns` ：要导出的列，使用英文状态逗号隔开，如果不填这个参数默认是导出表的所有列。

`line_delimiter` ：行分隔符。默认为 `\\n` 。支持不可见字符，例如 '\\x07'。

`exec_mem_limit` ： 表示 Export 作业中，一个查询计划在单个 BE 上的内存使用限制。默认 2GB。单位字节。

`timeout` ：作业超时时间。默认 2小时。单位秒。

`tablet_num_per_task` ：每个查询计划分配的最大分片数。默认为 5。

导出到对象存储

直接导出到云存储，而不通过broker。

```
EXPORT TABLE db.table
TO "s3://your_bucket/xx_path"
PROPERTIES
(
   "label"="your_label",
   "line_delimiter"="\\n",
   "column_separator"="\\001",
   "tablet_num_per_task"="10"
)
WITH S3
(
   "AWS_ENDPOINT" = "http://cos.ap-beijing.myqcloud.com",
   "AWS_ACCESS_KEY" = "AWS_ACCESS_KEY",
   "AWS_SECRET_KEY"="AWS_SECRET_KEY",
   "AWS_REGION"="AWS_REGION"
```

```
);
```

`AWS_ACCESS_KEY` / `AWS_SECRET_KEY` ：是您访问 OSS API 的密钥。

`AWS_ENDPOINT` ：表示 OSS 的数据中心所在的地域。

`AWS_REGION` ：Endpoint 表示 OSS 对外服务的访问域名。

查看导出状态

提交作业后，可以通过 SHOW EXPOR 命令查询导入作业状态。结果举例如下：

```
mysql> show EXPORT\\G;
*************************** 1. row ***************************
     JobId: 14008
     State: FINISHED
  Progress: 100%
  TaskInfo: {"partitions":["*"],"exec mem limit":2147483648,"column separator":",",
      Path: hdfs://host/path/to/export/
CreateTime: 2019-06-25 17:08:24
 StartTime: 2019-06-25 17:08:28
FinishTime: 2019-06-25 17:08:34
   Timeout: 3600
  ErrorMsg: NULL
1 row in set (0.01 sec)
```

JobId：作业的唯一 ID。

Label：自定义作业标识。

State：作业状态：

PENDING：作业待调度。

EXPORTING：数据导出中。

FINISHED：作业成功。

CANCELLED：作业失败。

Progress：作业进度。该进度以查询计划为单位。假设一共 10 个查询计划，当前已完成 3 个，则进度为 30%。

TaskInfo：以 Json 格式展示的作业信息：

db：数据库名。

tbl：表名。

partitions：指定导出的分区。 `*` 表示所有分区。

exec mem limit：查询计划内存使用限制。单位字节。

column separator：导出文件的列分隔符。

line delimiter：导出文件的行分隔符。

tablet num：涉及的总 Tablet 数量。

broker：使用的 broker 的名称。

coord num：查询计划的个数。

Path：远端存储上的导出路径。

CreateTime/StartTime/FinishTime：作业的创建时间、开始调度时间和结束时间。

Timeout：作业超时时间。单位是秒。该时间从 CreateTime 开始计算。

ErrorMsg：如果作业出现错误，这里会显示错误原因。

最佳实践

查询计划的拆分

一个 Export 作业有多少查询计划需要执行，取决于总共有多少 Tablet，以及一个查询计划最多可以分配多少个 Tablet。因为多个查询计划是串行执行的，所以如果让一个查询计划处理更多的分片，则可以减少作业的执行时间。但如果查询计划出错（例如调用 Broker 的 RPC 失败，远端存储出现抖动等），过多的 Tablet 会导致一个查询计划的重试成本变高。所以需要合理安排查询计划的个数以及每个查询计划所需要扫描的分片数，在执行时间和执行成功率之间做出平衡。一般建议一个查询计划扫描的数据量在 3-5 GB内（一个表的 Tablet 的大小以及个数可以通过 `SHOW TABLET FROM tbl_name;` 语句查看）。

exec_mem_limit

通常一个 Export 作业的查询计划只有 `扫描` - `导出` 两部分，不涉及需要太多内存的计算逻辑。所以通常 2GB 的默认内存限制可以满足需求。但在某些场景下，例如一个查询计划，在同一个 BE 上需要扫描的 Tablet 过多，或者 Tablet 的数据版本过多时，可能会导致内存不足。此时需要通过这个参数设置更大的内存，例如 4GB、8GB 等。

EXPORT 到腾讯云 COS

COS 域名获取

cos 桶有地域属性，指定 cos 上传地址时需要带上地域 ID，详细的地域 ID 可以从 地域和访问域名 查看，也可以在 COS 的列表页面上查看到：

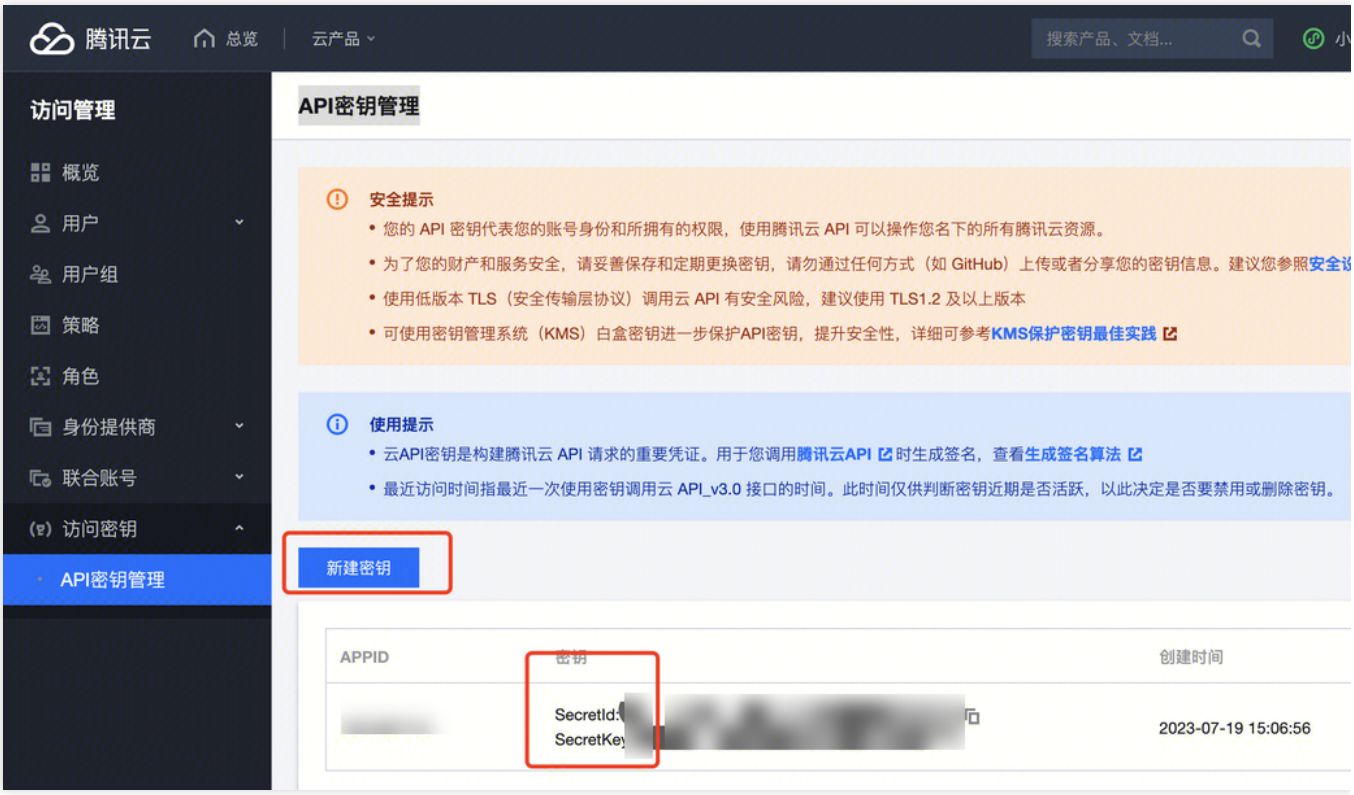
使用 EXPORT 语句时，需要指定 AWS_ENDPOINT 参数，这里即是 cos 的域名地址，格式为：

```
https://cos.[region_id].myqcloud.com
```

其中 region_id 为 cos 所属地域名称，例如上图中的：ap-guangzhou，其他字段不变。

COS 的 SecretId 和 SecretKey 获取

COS 的 SecretId 对应 S3 协议的 ACCESS_KEY，COS 的 SecretKey 对应 S3 协议的 SECRET_KEY，其中 COS 的 SecretId 和 SecretKey 获取需要从腾讯云 API 密钥管理获取如有现成密钥可以直接使用，如果没有则创建密钥。



开启导出任务

根据上面拿到的信息，可以创建 EXPORT 任务。这里以导出一个62G包含6亿行数据的 lineitem_ep 表为例子。

```
EXPORT TABLE lineitem TO "s3://doris-1301087413/doris-export/export_test"
WITH s3 (
"AWS_ENDPOINT" = "https://cos.ap-guangzhou.myqcloud.com",
"AWS_ACCESS_KEY" = "xxxSecretId",
"AWS_SECRET_KEY"="xxxSecretKey",
"AWS_REGION" = "ap-guangzhou"
);
```

其中，s3://doris-1301087413/doris-export/export_test 中。

s3是固定前缀，告知 doris 目标指定。

doris-1301087413为 cos 存储桶。

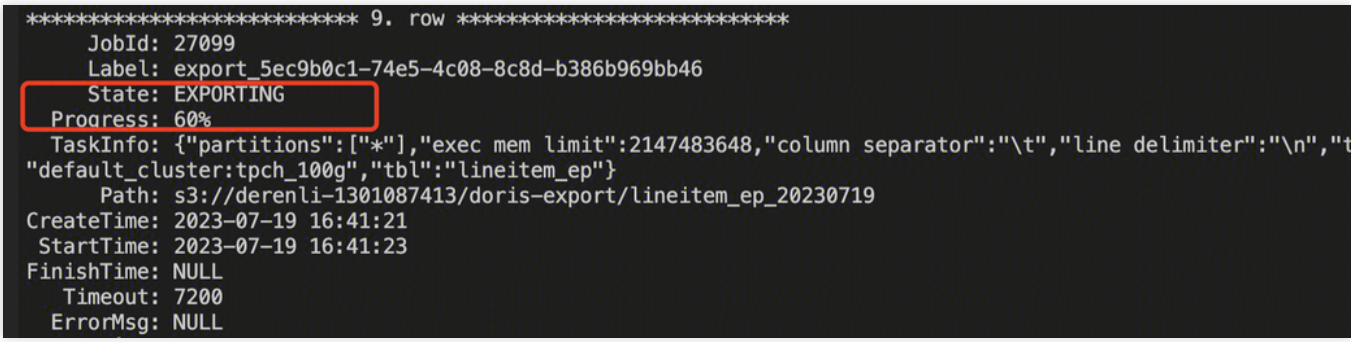
/doris-export/export_test 为指定导出位置的路径，如果路径不存在，doris 会自动创建。

"AWS_ENDPOINT" = "https://cos.ap-guangzhou.myqcloud.com" 是根据存储桶地域填写的 cos 服务器域名。

"AWS_REGION" = "ap-guangzhou" 是存储桶地域。

查看导出任务进度和信息

通过 show export; 命令可以查看 export 任务的进度和报错信息。



关注 State 和 Progress 字段，如果任务失败，ErrorMsg 字段会有对应的报错信息。

COS 数据文件导入到 DORIS

导入指令：

```
LOAD LABEL test_db.exmpale_label_4(
DATA INFILE("s3://doris-1301087413/doris-export/export_test/*")
INTO TABLE test_tb      COLUMNS TERMINATED BY "\\t"
)
WITH S3(
"AWS_ENDPOINT" = "https://cos.ap-guangzhou.myqcloud.com",
"AWS_ACCESS_KEY" = "xxxxx",     "AWS_SECRET_KEY"="xxxx",
"AWS_REGION" = "ap-guangzhou"
)
PROPERTIES(
"timeout" = "7600"
);
```

其中：

LABEL test_db.exmpale_label_4：test_db 为导入 db 名称。

exmpale_label_4为本次导入的标签，唯一标记一次导入任务 DATA INFILE("s3://doris-1301087413/doris-export/export_test/*") ：doris-1301087413/doris-export/export_test/ 为导入文件路径。

* 代表导入该路径下的所有文件都进行导入，也可以指定唯一文件名称。

EXPORT 到腾讯云 EMR 的 HDFS

网络确认

确认 Doris 集群与 EMR 集群在同一个 VPC 网络下。

创建导出任务

```
EXPORT TABLE orders TO "hdfs://hdfs_ip:hdfs_port/your_path"
PROPERTIES (
"column_separator"="\\t",
"line_delimiter" = "\\n"
```

```
)
WITH BROKER "Broker_Doris" (
"username"="-",
"password"="-"
)
```

其中：

Broker_Doris：Broker_Doris 为腾讯云 Doris 中 Broker 默认名称，无需修改。

username：访问 HDFS 的用户名。

password：访问 HDFS 的密码。

查看导出任务状态

通过 show export; 命令可以查看 export 任务的进度和报错信息。

```
MySQL [tpchdb]> show export\G
*************************** 1. row ***************************
     JobId: 38978
     Label: export_e40eb631-3eb8-40d3-b21c-f82452bd5361
     State: FINISHED
  Progress: 100%
  TaskInfo: {"partitions":["*"],"exec mem limit":2147483648,"column separator":"\t","line delimi
r_Doris","coord num":2,"db":"default_cluster:tpchdb","tbl":"orders"}
      Path: hdfs://10.0.1.134:4007/doris/tpch_100g_export/orders
CreateTime: 2023-08-11 11:37:03
 StartTime: 2023-08-11 11:37:08
FinishTime: 2023-08-11 11:37:17
   Timeout: 7200
  ErrorMsg: NULL
1 row in set (0.00 sec)
```

关注 State 和 Progress 字段，如果任务失败，ErrorMsg 字段会有对应的报错信息。

腾讯云 EMR 的 HDFS 导入 Doris

导入指令：

```
LOAD LABEL tpchdb.load_orders_recover(
DATA INFILE("hdfs://hdfs_ip:hdfs_port/your_path/*")
INTO TABLE orders_recover
COLUMNS TERMINATED BY "\\t"
)
WITH BROKER "Broker_Doris"(
"username" = "-",
"password" = "-!"
)
PROPERTIES
(
"timeout"="1200",
"max_filter_ratio"="0.1"
```

```
);
```

其中：

LABEL tpchdb.load_orders_recover：tpchdb 为导入 db 名称。

load_orders_recover 为本次导入的标签。

DATA INFILE("hdfs://hdfs_ip:hdfs_port/your_path/") ：*hdfs://hdfs_ip:hdfs_port/your_path/* 为导入文件路径。

* 代表导入该路径下的所有文件都进行导入，也可以指定唯一文件名称。

常见问题

导入报错 Scan byte per file scanner exceed limit：xxxxx

发现：show load 命令可以查看导入任务运行状态，ErrorMsg 展示报错信息。

```
MySQL [test_db]> show load\G;
*************************** 1. row ***************************
         JobId: 10164
         Label: exmpale_label_1
         State: CANCELLED
      Progress: ETL:N/A; LOAD:N/A
          Type: BROKER
       EtlInfo: NULL
      TaskInfo: cluster:N/A; timeout(s):3600; max_filter_ratio:0.0
      ErrorMsg: type:ETL_RUN_FAIL; msg:errCode = 2, detailMessage = No source file in this table(order_fo
    CreateTime: 2023-07-27 19:48:55
  EtlStartTime: NULL
 EtlFinishTime: NULL
 LoadStartTime: NULL
LoadFinishTime: 2023-07-27 19:48:59
           URL: NULL
    JobDetails: {"Unfinished backends":{},"ScannedRows":0,"TaskNumber":0,"LoadBytes":0,"All backends":{},
 TransactionId: 7
  ErrorTablets: {}
*************************** 2. row ***************************
         JobId: 10167
         Label: exmpale_label_2
         State: CANCELLED
      Progress: ETL:N/A; LOAD:N/A
          Type: BROKER
       EtlInfo: NULL
      TaskInfo: cluster:N/A; timeout(s):3600; max_filter_ratio:0.0
      ErrorMsg: type:ETL_RUN_FAIL; msg:errCode = 2, detailMessage = Scan bytes per file scanner exceed li
    CreateTime: 2023-07-27 19:49:31
  EtlStartTime: NULL
 EtlFinishTime: NULL
 LoadStartTime: NULL
LoadFinishTime: 2023-07-27 19:49:32
           URL: NULL
    JobDetails: {"Unfinished backends":{},"ScannedRows":0,"TaskNumber":0,"LoadBytes":0,"All backends":{},
 TransactionId: 9
  ErrorTablets: {}
2 rows in set (0.00 sec)

ERROR: No query specified

MySQL [test_db]>
```

原因：导入文件大小超过集群设置的最大值。

解决：修改 fe.conf 中配置max_broker_concurrency = BE 个数当前导入任务单个 BE 处理的数据量 = 原始文件大小/max_broker_concurrencymax_bytes_per_broker_scanner >= 当前导入任务单个 BE 处理的数据量

设置方式：

```
ADMIN SET FRONTEND CONFIG ("max_bytes_per_broker_scanner" = "52949672960");
ADMIN SET FRONTEND CONFIG ("max_broker_concurrency" = "3");
```

注意事项

不建议一次性导出大量数据。一个 Export 作业建议的导出数据量最大在几十 GB。过大的导出会导致更多的垃圾文件和更高的重试成本。

如果表数据量过大，建议按照分区导出。
在 Export 作业运行过程中，如果 FE 发生重启或切主，则 Export 作业会失败，需要用户重新提交。
如果 Export 作业运行失败，在远端存储中产生的 `__doris_export_tmp_xxx` 临时目录，以及已经生成的文件不会被删除，需要用户手动删除。
如果 Export 作业运行成功，在远端存储中产生的 `__doris_export_tmp_xxx` 目录，根据远端存储的文件系统语义，可能会保留，也可能会被清除。例如在百度对象存储（BOS）中，通过 rename 操作将一个目录中的最后一个文件移走后，该目录也会被删除。如果该目录没有被清除，用户可以手动清除。
当 Export 运行完成后（成功或失败），FE 发生重启或切主，则 `SHOW EXPORT` 展示的作业的部分信息会丢失，无法查看。
Export 作业只会导出 Base 表的数据，不会导出 Rollup Index 的数据。
Export 作业会扫描数据，占用 IO 资源，可能会影响系统的查询延迟。

相关配置

FE

`export_checker_interval_second` ：Export 作业调度器的调度间隔，默认为 5 秒。设置该参数需重启 FE。
`export_running_job_num_limit` ：正在运行的 Export 作业数量限制。如果超过，则作业将等待并处于 PENDING 状态。默认为 5，可以运行时调整。
`export_task_default_timeout_second` ：Export 作业默认超时时间。默认为 2 小时。可以运行时调整。
`export_tablet_num_per_task` ：一个查询计划负责的最大分片数。默认为 5。

使用 mysqldump工具导出表结构或者数据

最近更新时间：2024-06-27 11:07:02

Doris 在 0.15 之后的版本已经支持通过 `mysqldump` 工具导出数据或者表结构。

使用示例

导出

1. 导出 test 数据库中的 table1 表：`mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces --databases test --tables table1` 。
2. 导出 test 数据库中的 table1 表结构：`mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces --databases test --tables table1 --no-data` 。
3. 导出 test1，test2 数据库中所有表：`mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces --databases test1 test2` 。
4. 导出所有数据库和表 `mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces --all-databases` 。

更多的使用参数可以参考 `mysqldump` 的使用手册。

导入

`mysqldump` 导出的结果可以重定向到文件中，之后可以通过 source 命令导入到 Doris 中 `source filename.sql` 。

注意

1. 由于 Doris 中没有 mysql 里的 tablespace 概念，因此在使用 mysqldump 时要加上 `--no-tablespaces` 参数。
2. 使用 mysqldump 导出数据和表结构仅用于开发测试或者数据量很小的情况，请勿用于大数据量的生产环境。

导出查询结果集

最近更新时间：2024-06-27 11:07:17

本文档介绍如何使用 SELECT INTO OUTFILE 命令进行查询结果的导出操作。

语法

`SELECT INTO OUTFILE` 语句可以将查询结果导出到文件中。目前支持通过 Broker 进程, 通过 S3 协议, 或直接通过 HDFS 协议，导出到远端存储，如 HDFS，S3，BOS，COS（腾讯云）上，语法如下：

```
query_stmt
INTO OUTFILE "file_path"
[format_as]
[properties]
```

`file_path``file_path` 指向文件存储的路径以及文件前缀。如 `hdfs://path/to/my_file_` 。最终的文件名将由 `my_file_` ，文件序号以及文件格式后缀组成。其中文件序号由0开始，数量为文件被分割的数量。如：

```
my_file_abcdefg_0.csv
my_file_abcdefg_1.csv
my_file_abcdegf_2.csv
```

`[format_as]`

```
FORMAT AS CSV
```

指定导出格式。默认为 CSV。

`[properties]`

指定相关属性。目前支持通过 Broker 进程, 或通过 S3 协议进行导出。

Broker 相关属性需加前缀 `broker.` 。

HDFS 相关属性需加前缀 `hdfs.` 。

S3 协议则直接执行 S3 协议配置即可。

```
("broker.prop_key" = "broker.prop_val", ...)
or
("hdfs.fs.defaultFS" = "xxx", "hdfs.hdfs_user" = "xxx")
or
("AWS_ENDPOINT" = "xxx", ...)
```

其他属性：

```
("key1" = "val1", "key2" = "val2", ...)
```

目前支持以下属性：

`column_separator` ：列分隔符，仅对 CSV 格式适用。默认为 `\\t` 。

`line_delimiter` ：行分隔符，仅对 CSV 格式适用。默认为 `\\n` 。

`max_file_size` ：单个文件的最大大小。默认为 1GB。取值范围在 5MB 到 2GB 之间。超过这个大小的文件将会被切分。

`schema` ：PARQUET 文件schema信息。仅对 PARQUET 格式适用。导出文件格式为PARQUET时，必须指定 `schema` 。

并发导出

默认情况下，查询结果集的导出是非并发的，也就是单点导出。如果用户希望查询结果集可以并发导出，需要满足以下条件：

1. session variable 'enable_parallel_outfile' 开启并发导出: `set enable_parallel_outfile = true;` 。
2. 导出方式为 S3 , 或者 HDFS， 而不是使用 broker。
3. 查询可以满足并发导出的需求，例如顶层不包含 sort 等单点节点。（后面会举例说明，哪种属于不可并发导出结果集的查询）。

满足以上三个条件，就能触发并发导出查询结果集了。并发度 = `be_instance_num * parallel_fragment_exec_instance_num` 。

如何验证结果集被并发导出

用户通过 session 变量设置开启并发导出后，如果想验证当前查询是否能进行并发导出，则可以通过下面这个方法。

```
explain select xxx from xxx where xxx  into outfile "s3://xxx" format as csv proper
```

对查询进行 explain 后，Doris 会返回该查询的规划，如果您发现 `RESULT FILE SINK` 出现在 `PLAN FRAGMENT 1` 中，就说明导出并发开启成功了。

如果 `RESULT FILE SINK` 出现在 `PLAN FRAGMENT 0` 中，则说明当前查询不能进行并发导出 (当前查询不同时满足并发导出的三个条件)。

并发导出的规划示例：

```
+-----------------------------------------------------------------------------+
| Explain String                                                              |
+-----------------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                             |
|  OUTPUT EXPRS:<slot 2> | <slot 3> | <slot 4> | <slot 5>                    |
|   PARTITION: UNPARTITIONED                                                  |
|                                                                             |
|   RESULT SINK                                                               |
```

```
|                                                                                |
|   1:EXCHANGE                                                                   |
|                                                                                |
| PLAN FRAGMENT 1                                                                |
|  OUTPUT EXPRS:`k1` + `k2`                                                      |
|   PARTITION: HASH_PARTITIONED: `default_cluster:test`.`multi_tablet`.`k1`      |
|                                                                                |
|   RESULT FILE SINK                                                             |
|   FILE PATH: s3://ml-bd-repo/bpit_test/outfile_1951_                           |
|   STORAGE TYPE: S3                                                             |
|                                                                                |
|   0:OlapScanNode                                                               |
|      TABLE: multi_tablet                                                       |
+--------------------------------------------------------------------------------+
```

使用示例

1. 示例1

使用 broker 方式导出，将简单查询结果导出到文件 `hdfs://path/to/result.txt` 。指定导出格式为 CSV。使用 `Broker_Doris` 并设置 kerberos 认证信息。指定列分隔符为 `,` ，行分隔符为 `\\n` 。

```
SELECT * FROM tbl
INTO OUTFILE "hdfs://path/to/result_"
FORMAT AS CSV
PROPERTIES
(
      "broker.name" = "Broker_Doris",
      "broker.hadoop.security.authentication" = "kerberos",
      "broker.kerberos_principal" = "doris@YOUR.COM",
      "broker.kerberos_keytab" = "/home/doris/my.keytab",
      "column_separator" = ",",
      "line_delimiter" = "\\n",
      "max_file_size" = "100MB"
);
```

最终生成文件如果不大于 100MB，则为： `result_0.csv` 。如果大于 100MB，则可能为 `result_0.csv, result_1.csv, ...` 。

2. 示例2

将简单查询结果导出到文件 `hdfs://path/to/result.parquet` 。指定导出格式为 PARQUET。使用 `Broker_Doris` 并设置 kerberos 认证信息。

```
SELECT c1, c2, c3 FROM tbl
INTO OUTFILE "hdfs://path/to/result_"
```

```
FORMAT AS PARQUET
PROPERTIES
(
      "broker.name" = "Broker_Doris",
      "broker.hadoop.security.authentication" = "kerberos",
      "broker.kerberos_principal" = "doris@YOUR.COM",
      "broker.kerberos_keytab" = "/home/doris/my.keytab",
      "schema"="required,int32,c1;required,byte_array,c2;required,byte_array,c2"
);
```

查询结果导出到parquet文件需要明确指定 `schema` 。

3. 示例3

将 CTE 语句的查询结果导出到文件 `hdfs://path/to/result.txt` 。默认导出格式为 CSV。使用 `Broker_Doris` 并设置 hdfs 高可用信息。使用默认的行列分隔符。

```
WITH
x1 AS
(SELECT k1, k2 FROM tbl1),
x2 AS
(SELECT k3 FROM tbl2)
SELEC k1 FROM x1 UNION SELECT k3 FROM x2
INTO OUTFILE "hdfs://path/to/result_"
PROPERTIES
(
      "broker.name" = "Broker_Doris",
      "broker.username"="user",
      "broker.password"="passwd",
      "broker.dfs.nameservices" = "my_ha",
      "broker.dfs.ha.namenodes.my_ha" = "my_namenode1, my_namenode2",
      "broker.dfs.namenode.rpc-address.my_ha.my_namenode1" = "nn1_host:rpc_port",
      "broker.dfs.namenode.rpc-address.my_ha.my_namenode2" = "nn2_host:rpc_port",
      "broker.dfs.client.failover.proxy.provider" = "org.apache.hadoop.hdfs.server.n
);
```

最终生成文件如果不大于 1GB，则为： `result_0.csv` 。如果大于 1GB，则可能为 `result_0.csv, result_1.csv, ...` 。

4. 示例4

将 UNION 语句的查询结果导出到文件 `bos://bucket/result.txt` 。指定导出格式为 PARQUET。使用 `Broker_Doris` 并设置 hdfs 高可用信息。PARQUET 格式无需指定列分割符。导出完成后，生成一个标识文件。

```
SELECT k1 FROM tbl1 UNION SELECT k2 FROM tbl1
INTO OUTFILE "bos://bucket/result_"
FORMAT AS PARQUET
PROPERTIES
```

```
(
      "broker.name" = "Broker_Doris",
      "broker.bos_endpoint" = "http://bj.bcebos.com",
      "broker.bos_accesskey" = "xxxxxxxxxxxxxxxxxxxxxxxxxxx",
      "broker.bos_secret_accesskey" = "yyyyyyyyyyyyyyyyyyyyyyyyyyy",
      "schema"="required,int32,k1;required,byte_array,k2"
);
```

5. 示例5

将 select 语句的查询结果导出到文件 `cos://${bucket_name}/path/result.txt` 。指定导出格式为 csv。导出完成后，生成一个标识文件。

```
select k1,k2,v1 from tbl1 limit 100000
into outfile "cosn://my_bucket/export/my_file_"
FORMAT AS CSV
PROPERTIES
(
      "broker.name" = "Broker_Doris",
      "broker.fs.cosn.userinfo.secretId" = "xx",
      "broker.fs.cosn.userinfo.secretKey" = "xx",
      "broker.fs.cosn.bucket.endpoint_suffix" = "cos.<REGION>.myqcloud.com",
      "column_separator" = ",",
      "line_delimiter" = "\\n",
      "max_file_size" = "1024MB",
      "success_file_name" = "SUCCESS"
)
```

最终生成文件如果不大于 1GB，则为： `my_file_0.csv` 。如果大于 1GB，则可能为 `my_file_0.csv, result_1.csv, ...` 。

在 cos 上验证

不存在的path会自动创建。

access.key/secret.key/endpoint 需要和 cos 的同学确认。尤其是 endpoint 的值，不需要填写 bucket_name。

6. 示例6

使用 s3 协议导出到 bos，并且并发导出开启。

```
set enable_parallel_outfile = true;
select k1 from tb1 limit 1000
into outfile "s3://my_bucket/export/my_file_"
format as csv
properties
(
      "AWS_ENDPOINT" = "http://cos.<REGION>.mycloud.com",
      "AWS_ACCESS_KEY" = "xxxx",
      "AWS_SECRET_KEY" = "xxx",
      "AWS_REGION" = "<REGION>"
```

```
)
```

最终生成的文件前缀为 `my_file_{fragment_instance_id}_` 。

7. 示例7

使用 s3 协议导出到 bos，并且并发导出 session 变量开启。

```
set enable_parallel_outfile = true;
select k1 from tb1 order by k1 limit 1000
into outfile "s3://my_bucket/export/my_file_"
format as csv
properties
(
      "AWS_ENDPOINT" = "http://cos.<REGION>.mycloud.com",
      "AWS_ACCESS_KEY" = "xxxx",
      "AWS_SECRET_KEY" = "xxx",
      "AWS_REGION" = "<REGION>"
)
```

但由于查询语句带了一个顶层的排序节点，所以这个查询即使开启并发导出的 session 变量，也是无法并发导出的。

8. 示例8

使用 hdfs 方式导出，将简单查询结果导出到文件 `hdfs://path/to/result.txt` 。指定导出格式为 CSV。使用并设置 kerberos 认证信息。

```
SELECT * FROM tbl
INTO OUTFILE "hdfs://path/to/result_"
FORMAT AS CSV
PROPERTIES
(
      "hdfs.fs.defaultFS" = "hdfs://namenode:port",
      "hdfs.hadoop.security.authentication" = "kerberos",
      "hdfs.kerberos_principal" = "doris@YOUR.COM",
      "hdfs.kerberos_keytab" = "/home/doris/my.keytab"
);
```

返回结果

导出命令为同步命令。命令返回，即表示操作结束。同时会返回一行结果来展示导出的执行结果。

如果正常导出并返回，则结果如下：

```
mysql> select * from tbl1 limit 10 into outfile "file:///home/work/path/result_";
+------------+-----------+----------+--------------------------------------------
| FileNumber | TotalRows | FileSize | URL
```

```
+------------+-----------+----------+--------------------------------------------------+
|          1 |         2 |        8 | file:///192.168.1.10/home/work/path/result_{f
+------------+-----------+----------+--------------------------------------------------+
1 row in set (0.05 sec)
```

FileNumber：最终生成的文件个数。

TotalRows：结果集行数。

FileSize：导出文件总大小。单位字节。

URL：如果是导出到本地磁盘，则这里显示具体导出到哪个 Compute Node。

如果进行了并发导出，则会返回多行数据。

```
+------------+-----------+----------+--------------------------------------------------+
| FileNumber | TotalRows | FileSize | URL
+------------+-----------+----------+--------------------------------------------------+
|          1 |         3 |        7 | file:///192.168.1.10/home/work/path/result_{f
|          1 |         2 |        4 | file:///192.168.1.11/home/work/path/result_{f
+------------+-----------+----------+--------------------------------------------------+
2 rows in set (2.218 sec)
```

如果执行错误，则会返回错误信息，如：

```
mysql> SELECT * FROM tbl INTO OUTFILE ...
ERROR 1064 (HY000): errCode = 2, detailMessage = Open broker writer failed ...
```

注意事项

如果不开启并发导出，查询结果是由单个 BE 节点，单线程导出的。因此导出时间和导出结果集大小正相关。开启并发导出可以降低导出的时间。

导出命令不会检查文件及文件路径是否存在。是否会自动创建路径、或是否会覆盖已存在文件，完全由远端存储系统的语义决定。

如果在导出过程中出现错误，可能会有导出文件残留在远端存储系统上。Doris 不会清理这些文件。需要用户手动清理。

导出命令的超时时间同查询的超时时间。可以通过 `SET query_timeout=xxx` 进行设置。

对于结果集为空的查询，依然会产生一个大小为0的文件。

文件切分会保证一行数据完整的存储在单一文件中。因此文件的大小并不严格等于 `max_file_size` 。

对于部分输出为非可见字符的函数，如 BITMAP、HLL 类型，输出为 `\\N` ，即 NULL。

目前部分地理信息函数，如 `ST_Point` 的输出类型为 VARCHAR，但实际输出值为经过编码的二进制字符。当前这些函数会输出乱码。对于地理函数，请使用 `ST_AsText` 进行输出。

基础功能
变量（variables）

最近更新时间：2024-06-27 11:07:37

本文档主要介绍当前支持的变量（variables）。Doris 中的变量参考 MySQL 中的变量设置。但部分变量仅用于兼容一些 MySQL 客户端协议，并不产生其在 MySQL 数据库中的实际意义。

变量设置与查看

查看

可以通过 `SHOW VARIABLES [LIKE 'xxx'];` 查看所有或指定的变量。如：

```
SHOW VARIABLES;
SHOW VARIABLES LIKE '%time_zone%';
```

设置

部分变量可以设置全局生效或仅当前会话生效。设置全局生效后，后续新的会话连接中会沿用设置值。而设置仅当前会话生效，则变量仅对当前会话产生作用。

仅当前会话生效，通过 `SET var_name=xxx;` 语句来设置。如：

```
SET exec_mem_limit = 137438953472;
SET forward_to_master = true;
SET time_zone = "Asia/Shanghai";
```

全局生效，通过 `SET GLOBAL var_name=xxx;` 设置。如：

```
SET GLOBAL exec_mem_limit = 137438953472
```

说明

只有 ADMIN 用户可以设置变量的全局生效。

全局生效的变量不影响当前会话的变量值，仅影响新的会话中的变量。

既支持当前会话生效又支持全局生效的变量包括：

`time_zone`

`wait_timeout`

`sql_mode`

`enable_profile`

`query_timeout`

`exec_mem_limit`

`batch_size`

`allow_partition_column_nullable`

`insert_visible_timeout_ms`

`enable_fold_constant_by_be`

只支持全局生效的变量包括：

`default_rowset_type`

同时，变量设置也支持常量表达式。如：

```
SET exec_mem_limit = 10 * 1024 * 1024 * 1024;
SET forward_to_master = concat('tr', 'u', 'e');
```

在查询语句中设置变量

在一些场景中，我们可能需要对某些查询有针对性的设置变量。

通过使用SET_VAR提示可以在查询中设置会话变量（在单个语句内生效）。如：

```
SELECT /*+ SET_VAR(exec_mem_limit = 8589934592) */ name FROM people ORDER BY name;
SELECT /*+ SET_VAR(query_timeout = 1, enable_partition_cache=true) */ sleep(3);
```

注意

注释必须以/*+ 开头，并且只能跟随在 SELECT 之后。

支持的变量

`SQL_AUTO_IS_NULL`

用于兼容 JDBC 连接池 C3P0。 无实际作用。

`auto_increment_increment`

用于兼容 MySQL 客户端。无实际作用。

`autocommit`

用于兼容 MySQL 客户端。无实际作用。

`batch_size`

用于指定在查询执行过程中，各个节点传输的单个数据包的行数。默认一个数据包的行数为 1024 行，即源端节点每产生 1024 行数据后，打包发给目的节点。

较大的行数，会在扫描大数据量场景下提升查询的吞吐，但可能会在小查询场景下增加查询延迟。同时，也会增加查询的内存开销。建议设置范围1024至4096。

`character_set_client`

用于兼容 MySQL 客户端。无实际作用。

`character_set_connection`

用于兼容 MySQL 客户端。无实际作用。

`character_set_results`

用于兼容 MySQL 客户端。无实际作用。

`character_set_server`

用于兼容 MySQL 客户端。无实际作用。

`codegen_level`

用于设置 LLVM codegen 的等级。（当前未生效）。

`collation_connection`

用于兼容 MySQL 客户端。无实际作用。

`collation_database`

用于兼容 MySQL 客户端。无实际作用。

`collation_server`

用于兼容 MySQL 客户端。无实际作用。

`delete_without_partition`

设置为 true 时。当使用 delete 命令删除分区表数据时，可以不指定分区。delete 操作将会自动应用到所有分区。

但注意，自动应用到所有分区可能会导致 delete 命令耗时触发大量子任务导致耗时较长。如无必要，不建议开启。

`disable_colocate_join`

控制是否启用 Colocation Join 功能。默认为 false，表示启用该功能。true 表示禁用该功能。当该功能被禁用后，查询规划将不会尝试执行 Colocation Join。

`enable_bucket_shuffle_join`

控制是否启用 Bucket Shuffle Join 功能。默认为 true，表示启用该功能。false 表示禁用该功能。当该功能被禁用后，查询规划将不会尝试执行 Bucket Shuffle Join。

`disable_streaming_preaggregations`

控制是否开启流式预聚合。默认为 false，即开启。当前不可设置，且默认开启。

`enable_insert_strict`

用于设置通过 INSERT 语句进行数据导入时，是否开启 `strict` 模式。默认为 false，即不开启 `strict` 模式。

`enable_spilling`

用于设置是否开启大数据量落盘排序。默认为 false，即关闭该功能。当用户未指定 ORDER BY 子句的 LIMIT 条件，同时设置 `enable_spilling` 为 true 时，才会开启落盘排序。该功能启用后，会使用 BE 数据目录下 `doris-scratch/` 目录存放临时的落盘数据，并在查询结束后，清空临时数据。

该功能主要用于使用有限的内存进行大数据量的排序操作。

注意

该功能为实验性质，不保证稳定性，请谨慎开启。

`exec_mem_limit`

用于设置单个查询的内存限制。默认为 2GB，单位为B/K/KB/M/MB/G/GB/T/TB/P/PB, 默认为B。

该参数用于限制一个查询计划中，单个查询计划的实例所能使用的内存。一个查询计划可能有多个实例，一个 BE 节点可能执行一个或多个实例。所以该参数并不能准确限制一个查询在整个集群的内存使用，也不能准确限制一个查询在单一 BE 节点上的内存使用。具体需要根据生成的查询计划判断。

通常只有在一些阻塞节点（如排序节点、聚合节点、Join 节点）上才会消耗较多的内存，而其他节点（如扫描节点）中，数据为流式通过，并不会占用较多的内存。

当出现 `Memory Exceed Limit` 错误时，可以尝试指数级增加该参数，如 4G、8G、16G 等。

`forward_to_master`

用户设置是否将一些show 类命令转发到 Master FE 节点执行。默认为 `true`，即转发。Doris 中存在多个 FE 节点，其中一个为 Master 节点。通常用户可以连接任意 FE 节点进行全功能操作。但部分信息查看指令，只有从 Master FE 节点才能获取详细信息。

如 `SHOW BACKENDS;` 命令，如果不转发到 Master FE 节点，则仅能看到节点是否存活等一些基本信息，而转发到 Master FE 则可以获取包括节点启动时间、最后一次心跳时间等更详细的信息。

当前受该参数影响的命令如下：

1.1 `SHOW FRONTENDS;`

转发到 Master 可以查看最后一次心跳信息。

1.2 `SHOW BACKENDS;`

转发到 Master 可以查看启动时间、最后一次心跳信息、磁盘容量信息。

1.3 `SHOW BROKER;`

转发到 Master 可以查看启动时间、最后一次心跳信息。

1.4 `SHOW TABLET;` / `ADMIN SHOW REPLICA DISTRIBUTION;` / `ADMIN SHOW REPLICA STATUS;`

转发到 Master 可以查看 Master FE 元数据中存储的 tablet 信息。正常情况下，不同 FE 元数据中 tablet 信息应该是一致的。当出现问题时，可以通过这个方法比较当前 FE 和 Master FE 元数据的差异。

1.5 `SHOW PROC;`

转发到 Master 可以查看 Master FE 元数据中存储的相关 PROC 的信息。主要用于元数据比对。

`init_connect`

用于兼容 MySQL 客户端。无实际作用。

`interactive_timeout`

用于兼容 MySQL 客户端。无实际作用。

`enable_profile`

用于设置是否需要查看查询的 profile。默认为 false，即不需要 profile。

默认情况下，只有在查询发生错误时，BE 才会发送 profile 给 FE，用于查看错误。正常结束的查询不会发送 profile。发送 profile 会产生一定的网络开销，对高并发查询场景不利。

当用户希望对一个查询的 profile 进行分析时，可以将这个变量设为 true 后，发送查询。查询结束后，可以通过在当前连接的 FE 的 web 页面查看到 profile： `fe_host:fe_http_port/query`，

其中会显示最近100条，开启 `enable_profile` 的查询的 profile。

`language`

用于兼容 MySQL 客户端。无实际作用。

`license`

显示 Doris 的 License。无其他作用。

`load_mem_limit`

用于指定导入操作的内存限制。默认为 0，即表示不使用该变量，而采用 `exec_mem_limit` 作为导入操作的内存限制。

这个变量仅用于 INSERT 操作。因为 INSERT 操作设计查询和导入两个部分，如果用户不设置此变量，则查询和导入操作各自的内存限制均为 `exec_mem_limit` 。否则，INSERT 的查询部分内存限制为 `exec_mem_limit` ，而导入部分限制为 `load_mem_limit` 。

其他导入方式，如 BROKER LOAD，STREAM LOAD 的内存限制依然使用 `exec_mem_limit` 。

`lower_case_table_names`

用于控制用户表表名大小写是否敏感。

值为 0 时，表名大小写敏感。默认为0。

值为 1 时，表名大小写不敏感，doris 在存储和查询时会将表名转换为小写。优点：在一条语句中可以使用表名的任意大小写形式，下面的sql是正确的：

```
mysql> show tables;
+------------------+
| Tables_in_testdb |
+------------------+
| cost             |
+------------------+

mysql> select * from COST where COst.id < 100 order by cost.id;
```

缺点：建表后无法获得建表语句中指定的表名， `show tables` 查看的表名为指定表名的小写。

值为 2 时，表名大小写不敏感，doris存储建表语句中指定的表名，查询时转换为小写进行比较。

优点： `show tables` 查看的表名为建表语句中指定的表名；缺点：同一语句中只能使用表名的一种大小写形式，例如对 `cost` 表使用表名 `COST` 进行查询：

```
mysql> select * from COST where COST.id < 100 order by COST.id;
```

该变量兼容 MySQL。需在集群初始化时通过 fe.conf 指定 `lower_case_table_names=` 进行配置，集群初始化完成后无法通过 `set` 语句修改该变量，也无法通过重启、升级集群修改该变量。

information_schema 中的系统视图表名不区分大小写，当 `lower_case_table_names` 值为 0 时，表现为 2。

`max_allowed_packet`

用于兼容 JDBC 连接池 C3P0。 无实际作用。

`max_pushdown_conditions_per_column`

该变量默认置为 -1，表示使用 `be.conf` 中的配置值。如果设置大于 0，则当前会话中的查询会使用该变量值，而忽略 `be.conf` 中的配置值。

`max_scan_key_num`

该变量默认置为 -1，表示使用 `be.conf` 中的配置值。如果设置大于 0，则当前会话中的查询会使用该变量值，而忽略 `be.conf` 中的配置值。

`net_buffer_length`

用于兼容 MySQL 客户端。无实际作用。

`net_read_timeout`

用于兼容 MySQL 客户端。无实际作用。

`net_write_timeout`

用于兼容 MySQL 客户端。无实际作用。

`parallel_exchange_instance_num`

用于设置执行计划中，一个上层节点接收下层节点数据所使用的 exchange node 数量。默认为 -1，即表示 exchange node 数量等于下层节点执行实例的个数（默认行为）。当设置大于0，并且小于下层节点执行实例的个数，则 exchange node 数量等于设置值。

在一个分布式的查询执行计划中，上层节点通常有一个或多个 exchange node 用于接收来自下层节点在不同 BE 上的执行实例的数据。通常 exchange node 数量等于下层节点执行实例数量。

在一些聚合查询场景下，如果底层需要扫描的数据量较大，但聚合之后的数据量很小，则可以尝试修改此变量为一个较小的值，可以降低此类查询的资源开销。如在 DUPLICATE KEY 明细模型上进行聚合查询的场景。

`parallel_fragment_exec_instance_num`

针对扫描节点，设置其在每个 BE 节点上，执行实例的个数。默认为 1。

一个查询计划通常会产生一组 scan range，即需要扫描的数据范围。这些数据分布在多个 BE 节点上。一个 BE 节点会有一个或多个 scan range。默认情况下，每个 BE 节点的一组 scan range 只由一个执行实例处理。当机器资源比较充裕时，可以将增加该变量，让更多的执行实例同时处理一组 scan range，从而提升查询效率。

而 scan 实例的数量决定了上层其他执行节点，如聚合节点，join 节点的数量。因此相当于增加了整个查询计划执行的并发度。修改该参数会对大查询效率提升有帮助，但较大数值会消耗更多的机器资源，如 -CPU、内存、磁盘 IO。

`query_cache_size`

用于兼容 MySQL 客户端。无实际作用。

`query_cache_type`

用于兼容 JDBC 连接池 C3P0。 无实际作用。

- `query_timeout`

用于设置查询超时。该变量会作用于当前连接中所有的查询语句，以及 INSERT 语句。默认为 5 分钟，单位为秒。

`resource_group`

暂不使用。

`send_batch_parallelism` 用于设置执行 InsertStmt 操作时发送批处理数据的默认并行度，如果并行度的值超过 BE 配置中的 `max_send_batch_parallelism_per_job` ，那么作为协调点的 BE 将使用 `max_send_batch_parallelism_per_job` 的值。

`sql_mode`

用于指定 SQL 模式，以适应某些 SQL 方言。

`sql_safe_updates`

用于兼容 MySQL 客户端。无实际作用。

`sql_select_limit`

用于兼容 MySQL 客户端。无实际作用。

`system_time_zone`

显示当前系统时区。不可更改。

`time_zone`

用于设置当前会话的时区。时区会对某些时间函数的结果产生影响。

`tx_isolation`

用于兼容 MySQL 客户端。无实际作用。

`version`

用于兼容 MySQL 客户端。无实际作用。

`performance_schema`

用于兼容 8.0.16及以上版本的MySQL JDBC。无实际作用。

`version_comment`

用于显示 Doris 的版本。不可更改。

`wait_timeout`

用于设置空闲连接的连接时长。当一个空闲连接在该时长内与 Doris 没有任何交互，则 Doris 会主动断开这个链接。默认为 8 小时，单位为秒。

`default_rowset_type`

用于设置计算节点存储引擎默认的存储格式。当前支持的存储格式包括：alpha/beta。

`use_v2_rollup`

用于控制查询使用segment v2存储格式的rollup索引获取数据。该变量用于上线segment v2的时候，进行验证使用；其他情况，不建议使用。

`rewrite_count_distinct_to_bitmap_hll`

是否将 bitmap 和 hll 类型的 count distinct 查询重写为 bitmap_union_count 和 hll_union_agg 。

`prefer_join_method`

在选择 join 的具体实现方式是 broadcast join 还是 shuffle join 时，如果 broadcast join cost 和 shuffle join cost 相等时，优先选择哪种 join 方式。

目前该变量的可选值为"broadcast" 或者 "shuffle"。

`allow_partition_column_nullable`

建表时是否允许分区列为 NULL。默认为 true，表示允许为 NULL。false 表示分区列必须被定义为 NOT NULL。

`insert_visible_timeout_ms`

在执行 insert 语句时，导入动作(查询和插入)完成后，还需要等待事务提交，使数据可见。此参数控制等待数据可见的超时时间，默认为10000，最小为1000。

`enable_exchange_node_parallel_merge`

在一个排序的查询之中，一个上层节点接收下层节点有序数据时，会在 exchange node 上进行对应的排序来保证最终的数据是有序的。但是单线程进行多路数据归并时，如果数据量过大，会导致 exchange node 的单点的归并瓶颈。

Doris 在这部分进行了优化处理，如果下层的数据节点过多。exchange node 会启动多线程进行并行归并来加速排序过程。该参数默认为 False，即表示 exchange node 不采取并行的归并排序，来减少额外的 CPU 和内存消耗。

`extract_wide_range_expr`

用于控制是否开启 「宽泛公因式提取」的优化。取值有两种：true 和 false 。默认情况下开启。

`enable_fold_constant_by_be`

用于控制常量折叠的计算方式。默认是 `false` ，即在 `FE` 进行计算；若设置为 `true` ，则通过 `RPC` 请求经 `BE` 计算。

`cpu_resource_limit`

用于限制一个查询的资源开销。这是一个实验性质的功能。目前的实现是限制一个查询在单个节点上的 scan 线程数量。限制了 scan 线程数，从底层返回的数据速度变慢，从而限制了查询整体的计算资源开销。假设设置为 2，则一个查询在单节点上最多使用2个 scan 线程。

该参数会覆盖 `parallel_fragment_exec_instance_num` 的效果。即假设 `parallel_fragment_exec_instance_num` 设置为4，而该参数设置为2。则单个节点上的4个执行实例会共享最多2个扫描线程。

该参数会被 user property 中的 `cpu_resource_limit` 配置覆盖。

默认 -1，即不限制。

`disable_join_reorder`

用于关闭所有系统自动的 join reorder 算法。取值有两种：true 和 false。默认情况下关闭，也就是采用系统自动的 join reorder 算法。设置为 true 后，系统会关闭所有自动排序的算法，采用 SQL 原始的表顺序，执行 join。

`return_object_data_as_binary`

用于标识是否在 select 结果中返回 bitmap/hll 结果。在 select into outfile 语句中，如果导出文件格式为csv 则会将 bimap/hll 数据进行 base64编码，如果是 parquet 文件格式将会把数据作为 byte array 存储。

`block_encryption_mode`

可以通过 block_encryption_mode 参数，控制块加密模式，默认值为：空。当使用AES算法加密时相当于 `AES_128_ECB` ，当时用SM3算法加密时相当于 `SM3_128_ECB` 可选值：

```
AES_128_ECB,
AES_192_ECB,
AES_256_ECB,
AES_128_CBC,
AES_192_CBC,
AES_256_CBC,
AES_128_CFB,
AES_192_CFB,
AES_256_CFB,
```

```
AES_128_CFB1,
AES_192_CFB1,
AES_256_CFB1,
AES_128_CFB8,
AES_192_CFB8,
AES_256_CFB8,
AES_128_CFB128,
AES_192_CFB128,
AES_256_CFB128,
AES_128_CTR,
AES_192_CTR,
AES_256_CTR,
AES_128_OFB,
AES_192_OFB,
AES_256_OFB,
SM4_128_ECB,
SM4_128_CBC,
SM4_128_CFB128,
SM4_128_OFB,
SM4_128_CTR,
```

`enable_infer_predicate`

用于控制是否进行谓词推导。取值有两种：true 和 false。默认情况下关闭，系统不在进行谓词推导，采用原始的谓词进行相关操作。设置为 true 后，进行谓词扩展。

`trim_tailing_spaces_for_external_table_query`

用于控制查询 Hive 外表时是否过滤掉字段末尾的空格。默认为 false。

数据删除

最近更新时间：2024-06-27 11:07:54

批量删除

目前 Doris 支持 Broker Load（HDFS 数据），Routine Load（Kafka 数据），Stream load（本地文件）等多种导入方式，对于数据的删除目前只能通过 delete 语句进行删除，使用 delete 语句的方式删除时，每执行一次 delete 都会生成一个新的数据版本，如果频繁删除会严重影响查询性能，并且在使用 delete 方式删除时，是通过生成一个空的 rowset 来记录删除条件实现，每次读取都要对删除条件进行过滤，同样在条件较多时会对性能造成影响。
对比其他的系统，greenplum 的实现方式更像是传统数据库产品，snowflake 通过 merge 语法实现。
对于类似于 cdc 数据导入的场景，数据中 insert 和 delete 一般是穿插出现的，面对这种场景我们目前的导入方式也无法满足，即使我们能够分离出 insert 和 delete 虽然可以解决导入的问题，但是仍然解决不了删除的问题。使用批量删除功能可以解决这些个别场景的需求。数据导入有三种合并方式：
1. APPEND：数据全部追加到现有数据中。
2. DELETE：删除所有与导入数据 key 列值相同的行。
3. MERGE：根据 DELETE ON 的决定 APPEND 还是 DELETE。

基本原理

通过增加一个隐藏列 `__DORIS_DELETE_SIGN__` 实现，因为我们只是在 unique 模型上做批量删除，因此只需要增加一个类型为 bool 聚合函数为 replace 的隐藏列即可。在 BE 各种聚合写入流程都和正常列一样，读取方案有两个：
在 FE 遇到 select * 等扩展时去掉 `__DORIS_DELETE_SIGN__` 列，并且默认加上 `__DORIS_DELETE_SIGN__ != true` 的条件， BE 读取时都会加上一列进行判断，通过条件确定是否删除。

导入

导入时在 FE 解析时将隐藏列的值设置成 `DELETE ON` 表达式的值，其他的聚合行为和 replace 的聚合列相同。

读取

读取时在所有存在隐藏列的 olapScanNode 上增加 `__DORIS_DELETE_SIGN__ != true` 的条件，BE 不感知这一过程，正常执行。

Cumulative Compaction

Cumulative Compaction 时将隐藏列看作正常的列处理，Compaction 逻辑没有变化。

Base Compaction

Base Compaction 时要删掉被标记为删除的行，以减少数据占用的空间。

启用批量删除支持

启用批量删除支持有以下两种形式：

1. 通过在 FE 配置文件中增加 `enable_batch_delete_by_default=true` 重启 FE 后新建表的都支持批量删除，此选项默认为 false。

2. 对于没有更改上述 FE 配置或对于以存在的不支持批量删除功能的表，可以使用如下语句： `ALTER TABLE tablename ENABLE FEATURE "BATCH_DELETE"` 来启用批量删除。本操作本质上是一个 schema change 操作，操作立即返回，可以通过 `show alter table column` 来确认操作是否完成。

那么如何确定一个表是否支持批量删除，可以通过 设置一个 session variable 来显示隐藏列 `SET show_hidden_columns=true` ，之后使用 `desc tablename` ，如果输出中有 `__DORIS_DELETE_SIGN__` 列则支持，如果没有则不支持。

语法说明

导入的语法设计方面主要是增加一个指定删除标记列的字段的 colum 映射，并且需要在导入的数据中增加一列，各种导入方式设置的语法如下。

Stream Load

`Stream Load` 的写法在 header 中的 columns 字段增加一个设置删除标记列的字段， 示例 `-H "columns: k1, k2, label_c3" -H "merge_type: [MERGE|APPEND|DELETE]" -H "delete: label_c3=1"` 。

Broker Load

`Broker Load` 的写法在 `PROPERTIES` 处设置删除标记列的字段，语法如下：

```
LOAD LABEL db1.label1
(
    [MERGE|APPEND|DELETE] DATA INFILE("hdfs://abc.com:8888/user/palo/test/ml/file1"
    INTO TABLE tbl1
    COLUMNS TERMINATED BY ","
    (tmp_c1,tmp_c2, label_c3)
    SET
    (
        id=tmp_c2,
        name=tmp_c1,
    )
    [DELETE ON label_c3=true]
)
WITH BROKER 'broker'
(
    "username"="user",
    "password"="pass"
)
PROPERTIES
(
```

```
    "timeout" = "3600"
);
```

Routine Load

`Routine Load` 的写法在 `columns` 字段增加映射，映射方式同上，语法如下：

```
CREATE ROUTINE LOAD example_db.test1 ON example_tbl
 [WITH MERGE|APPEND|DELETE]
 COLUMNS(k1, k2, k3, v1, v2, label),
 WHERE k1 > 100 and k2 like "%doris%"
 [DELETE ON label=true]
 PROPERTIES
 (
     "desired_concurrent_number"="3",
     "max_batch_interval" = "20",
     "max_batch_rows" = "300000",
     "max_batch_size" = "209715200",
     "strict_mode" = "false"
 )
 FROM KAFKA
 (
     "kafka_broker_list" = "broker1:9092,broker2:9092,broker3:9092",
     "kafka_topic" = "my_topic",
     "kafka_partitions" = "0,1,2,3",
     "kafka_offsets" = "101,0,0,200"
 );
```

注意事项

1. 由于除 `Stream Load` 外的导入操作在 Doris 内部有可能乱序执行，因此在使用 `MERGE` 方式导入时如果不是 `Stream Load`，需要与 load sequence 一起使用，具体的语法可以参照 Sequence 列 相关的文档。
2. `DELETE ON` 条件只能与 MERGE 一起使用。

使用示例

查看是否启用批量删除支持

```
mysql> SET show_hidden_columns=true;
Query OK, 0 rows affected (0.00 sec)

mysql> DESC test;
+-----------------------+--------------+------+-------+---------+---------+
| Field                 | Type         | Null | Key   | Default | Extra   |
+-----------------------+--------------+------+-------+---------+---------+
| name                  | VARCHAR(100) | No   | true  | NULL    |         |
```

```
| gender                | VARCHAR(10)   | Yes   | false | NULL     | REPLACE |
| age                   | INT           | Yes   | false | NULL     | REPLACE |
| __DORIS_DELETE_SIGN__ | TINYINT       | No    | false | 0        | REPLACE |
+-----------------------+---------------+-------+-------+----------+---------+
4 rows in set (0.00 sec)
```

Stream Load 使用示例

1. 正常导入数据：

```
curl --location-trusted -u root: -H "column_separator:," -H "columns: siteid, cityc
```

其中的 APPEND 条件可以省略，与下面的语句效果相同：

```
curl --location-trusted -u root: -H "column_separator:," -H "columns: siteid, cityc
```

2. 将与导入数据 key 相同的数据全部删除：

```
curl --location-trusted -u root: -H "column_separator:," -H "columns: siteid, cityc
```

假设导入表中原有数据为:

```
+--------+----------+----------+------+
| siteid | citycode | username | pv   |
+--------+----------+----------+------+
|      3 |        2 | tom      |    2 |
|      4 |        3 | bush     |    3 |
|      5 |        3 | helen    |    3 |
+--------+----------+----------+------+
```

导入数据为：

```
3,2,tom,0
```

导入后数据变成：

```
+--------+----------+----------+------+
| siteid | citycode | username | pv   |
+--------+----------+----------+------+
|      4 |        3 | bush     |    3 |
|      5 |        3 | helen    |    3 |
+--------+----------+----------+------+
```

3. 将导入数据中与 `site_id=1` 的行的 key 列相同的行。

```
curl --location-trusted -u root: -H "column_separator:," -H "columns: siteid, cityc
```

假设导入前数据为：

```
+--------+----------+----------+------+
| siteid | citycode | username | pv   |
+--------+----------+----------+------+
|      4 |        3 | bush     |    3 |
|      5 |        3 | helen    |    3 |
|      1 |        1 | jim      |    2 |
+--------+----------+----------+------+
```

导入数据为：

```
2,1,grace,2
3,2,tom,2
1,1,jim,2
```

导入后为：

```
+--------+----------+----------+------+
| siteid | citycode | username | pv   |
+--------+----------+----------+------+
|      4 |        3 | bush     |    3 |
|      2 |        1 | grace    |    2 |
|      3 |        2 | tom      |    2 |
|      5 |        3 | helen    |    3 |
+--------+----------+----------+------+
```

Sql Delete 操作

Delete 不同于其他导入方式，它是一个同步过程，与 Insert into 相似，所有的 Delete 操作在 Doris 中是一个独立的导入作业，一般 Delete 语句需要指定表和分区以及删除的条件来筛选要删除的数据，并将会同时删除 base 表和 rollup 表的数据。Delete 操作的语法详见 DELETE 语法。

返回结果

Delete 命令是一个 SQL 命令，返回结果是同步的，分为以下几种：

1. 执行成功

如果 Delete 顺利执行完成并可见，将返回下列结果， `Query OK` 表示成功。

```
mysql> delete from test_tbl PARTITION p1 where k1 = 1;
Query OK, 0 rows affected (0.04 sec)
{'label':'delete_e7830c72-eb14-4cb9-bbb6-eebd4511d251', 'status':'VISIBLE', 'txnId'
```

2. 提交成功，但未可见。

Doris 的事务提交分为两步：提交和发布版本，只有完成了发布版本步骤，结果才对用户是可见的。若已经提交成功

了，那么就可以认为最终一定会发布成功，Doris 会尝试在提交完后等待发布一段时间，如果超时后即使发布版本还未完成也会优先返回给用户，提示用户提交已经完成。若如果 Delete 已经提交并执行，但是仍未发布版本和可见，将返回下列结果：

```
mysql> delete from test_tbl PARTITION p1 where k1 = 1;
Query OK, 0 rows affected (0.04 sec)
{'label':'delete_e7830c72-eb14-4cb9-bbb6-eebd4511d251', 'status':'COMMITTED', 'txn
```

结果会同时返回一个 json 字符串：

`affected rows` ：表示此次删除影响的行，由于 Doris 的删除目前是逻辑删除，因此对于这个值是恒为0。

`label` ：自动生成的 label，是该导入作业的标识。每个导入作业，都有一个在单 database 内部唯一的 Label。

`status` ：表示数据删除是否可见，如果可见则显示 `VISIBLE` ，如果不可见则显示 `COMMITTED` 。

`txnId` ：这个 Delete job 对应的事务 ID。

`err` ：字段会显示一些本次删除的详细信息。

3. 提交失败，事务取消。

如果 Delete 语句没有提交成功，将会被 Doris 自动中止，返回下列结果：

```
mysql> delete from test_tbl partition p1 where k1 > 80;
ERROR 1064 (HY000): errCode = 2, detailMessage = {错误原因}
```

示例：

例如说一个超时的删除，将会返回 timeout 时间和未完成的 `(tablet=replica)` 。

```
mysql> delete from test_tbl partition p1 where k1 > 80;
ERROR 1064 (HY000): errCode = 2, detailMessage = failed to delete replicas from job
```

综上，对于 Delete 操作返回结果的正确处理逻辑为：

1. 如果返回结果为 `ERROR 1064 (HY000)` ，则表示删除失败。

2. 如果返回结果为 `Query OK` ，则表示删除执行成功。

如果 `status` 为 `COMMITTED` ，表示数据仍不可见，用户可以稍等一段时间再用 `show delete` 命令查看结果。

如果 `status` 为 `VISIBLE` ，表示数据删除成功。

Delete 操作相关 FE 配置

TIMEOUT 配置

总体来说，Doris 的删除作业的超时时间限制在30秒到5分钟时间内，具体时间可通过下面配置项调整：

`tablet_delete_timeout_second`

delete 自身的超时时间是可受指定分区下 tablet 的数量弹性改变的，此项配置为平均一个 tablet 所贡献的 timeout 时间，默认值为2。

假设此次删除所指定分区下有5个 tablet，那么可提供给 delete 的 timeout 时间为10秒，由于低于最低超时时间30秒，因此最终超时时间为30秒。

`load_straggler_wait_second`

如果用户预估的数据量确实比较大，使得5分钟的上限不足时，用户可以通过此项调整 timeout 上限，默认值为300。

TIMEOUT 的具体计算规则为(秒)

```
TIMEOUT = MIN(load_straggler_wait_second, MAX(30, tablet_delete_timeout_second * ta
```

`query_timeout`

因为 delete 本身是一个 SQL 命令，因此删除语句也会受 session 限制，timeout 还受 Session 中的 `query_timeout` 值影响，可以通过 `SET query_timeout = xxx` 来增加超时时间，单位是秒。

IN 谓词配置

`max_allowed_in_element_num_of_delete`

如果用户在使用 in 谓词时需要占用的元素比较多，用户可以通过此项调整允许携带的元素上限，默认值为1024。

查看历史记录

用户可以通过 show delete 语句查看历史上已执行完成的删除记录。

语法如下：

```
SHOW DELETE [FROM db_name]
```

使用示例：

```
mysql> show delete from test_db;
+-----------+---------------+---------------------+-----------------+----------+
| TableName | PartitionName | CreateTime          | DeleteCondition | State    |
+-----------+---------------+---------------------+-----------------+----------+
| empty_tbl | p3            | 2020-04-15 23:09:35 | k1 EQ "1"       | FINISHED |
| test_tbl  | p4            | 2020-04-15 23:09:53 | k1 GT "80"      | FINISHED |
+-----------+---------------+---------------------+-----------------+----------+
2 rows in set (0.00 sec)
```

注意事项

不同于 Insert into 命令，delete 不能手动指定 `label` ，有关概念可以查看 INSERT INTO 文档。

更多帮助

更多详细语法请参阅 Delete 命令手册，也可以在 Mysql 客户端命令行下输入 `HELP DELETE` 获取更多帮助信息。

数据更新

最近更新时间：2024-06-27 11:08:07

如果需要修改或更新 Doris 中的数据，可以使用 UPDATE 命令来操作。

适用场景

对满足某些条件的行，修改其取值。
点更新、小范围更新，待更新的行最好是整个表的非常小一部分。
update 命令只能在 Unique 数据模型的表中操作。

名词解释

Unique 模型：Doris 系统中的一种数据模型。将列分为两类，Key 和 Value。当用户导入相同 Key 的行时，后者的 Value 会覆盖已有的 Value。与 Mysql 中的 Unique 含义一致。

基本原理

利用查询引擎自身的 where 过滤逻辑，从待更新表中筛选出需要被更新的行。再利用 Unique 模型自带的 Value 列新数据替换旧数据的逻辑，将待更新的行变更后，再重新插入到表中。从而实现行级别更新。
举例说明：
假设 Doris 中存在一张订单表，其中 订单 ID 是 Key 列，订单状态，订单金额是 Value 列。数据状态如下：

订单 ID	订单金额	订单状态
1	100	待付款

这时，用户点击付款后，Doris 系统需要将订单 id 为 '1' 的订单状态变更为 '待发货', 就需要用到 Update 功能。

```
UPDATE order SET 订单状态='待发货' WHERE 订单id=1;
```

用户执行 UPDATE 命令后，系统会进行如下三步：
第一步：读取满足 WHERE 订单 ID=1 的行：（1，100，'待付款'）
第二步：变更该行的订单状态，从'待付款'改为'待发货'：（1，100，'待发货'）
第三步：将更新后的行再插入回表中，从而达到更新的效果。

订单 ID	订单金额	订单状态

1	100	待付款
1	100	待发货

由于表 order 是 UNIQUE 模型，所以相同 Key 的行，之后后者才会生效，所以最终效果如下：

订单 ID	订单金额	订单状态
1	100	待发货

基本操作

UPDATE 语法

```
UPDATE table_name SET value=xxx WHERE condition;
```

`table_name` ：待更新的表，必须是 UNIQUE 模型的表才能进行更新。

`value=xxx` ：待更新的列，等式左边必须是表的 value 列。等式右边可以是常量，也可以是某个表中某列的表达式变换。

例如：value = 1, 则待更新的列值会变为1。value = value +1， 则待更新的列值会自增1。

`condition` ：只有满足 condition 的行才会被更新。condition 必须是一个结果为 Boolean 类型的表达式。

例如： k1 = 1，则只有当 k1 列值为1的行才会被更新。k1 = k2，则只有 k1 列值和 k2 列一样的行才会被更新。

不支持不填写 condition，即不支持全表更新。

同步

Update 语法在 Doris 中是一个同步语法，即 Update 语句成功就代表更新成功、数据可见。

性能

Update 语句的性能和待更新的行数，以及 condition 的检索效率密切相关。

待更新的行数：待更新的行数越多，Update 语句的速度就会越慢。这和导入的原理是一致的。Doris 的更新比较合适偶发更新的场景，例如修改个别行的值，并不适合大批量的修改数据。大批量修改会使得 Update 语句运行时间很久。

condition 的检索效率：Doris 的 Update 实现原理是先将满足 condition 的行读取处理，所以如果 condition 的检索效率高，则 Update 的速度也会快。

condition 列最好能命中索引或者分区分桶裁剪，这样 Doris 就不需要扫全表，可以快速定位到需要更新的行，从而提升更新效率。

注意

强烈不推荐 update 语句中的 condition 列中包含 UNIQUE 模型的 Value 列，避免更新时进行全表扫描，降低更新效率。

并发控制

默认情况下，并不允许同一时间对同一张表并发进行多个 Update 操作。主要原因是，Doris 目前支持的是行更新，这意味着，即使用户声明的是 `SET v2 = 1` 。实际上，其他所有的 Value 列也会被覆盖一遍（尽管值没有变化）。

这就会存在一个问题，如果同时有两个 Update 操作对同一行进行更新，那么其行为可能是不确定的。也就是可能存在脏数据。

但在实际应用中，如果用户自己可以保证即使并发更新，也不会同时对同一行进行操作的话，就可以手动打开并发限制。通过修改 FE 配置 `enable_concurrent_update` 。配置值默认为false，当配置值为 true 时，则对更新并发无限制。用户需要**谨慎评估**后再决定是否修改该配置。

使用风险

由于 Doris 目前支持的是行更新，并且采用的是读取后再写入的两步操作，则如果 Update 语句和其他导入或 Delete 语句刚好修改的是同一行时，存在不确定的数据结果。所以用户在使用的时候，一定要注意**用户侧自行控制** Update 语句和其他 DML 语句的并发。

Sequence 列

最近更新时间：2024-06-27 11:08:28

Unique 模型主要针对需要唯一主键的场景，可以保证主键唯一性约束，但由于使用 REPLACE 聚合方式，在同一批次中导入的数据，替换顺序不做保证。替换顺序无法保证则无法确定最终导入到表中的具体数据，存在了不确定性。
为了解决这个问题，Doris 支持了 sequence 列，通过用户在导入时指定 sequence 列，相同 key 列下，REPLACE 聚合类型的列将按照 sequence 列的值进行替换，较大值可以替换较小值，反之则无法替换。该方法将顺序的确定交给了用户，由用户控制替换顺序。

适用场景

Sequence 列只能在 Unique 数据模型下使用。

基本原理

通过增加一个隐藏列 `__DORIS_SEQUENCE_COL__` 实现，该列的类型由用户在建表时指定，在导入时确定该列具体值，并依据该值对 REPLACE 列进行替换。

建表

创建 Unique 表时，将按照用户指定类型自动添加一个隐藏列 `__DORIS_SEQUENCE_COL__` 。

导入

导入时，FE 在解析的过程中将隐藏列的值设置成 `order by` 表达式的值(Broker load 和 Routine load)，或者 `function_column.sequence_col` 表达式的值(stream load), Value 列将按照该值进行替换。隐藏列 `__DORIS_SEQUENCE_COL__` 的值既可以设置为数据源中一列，也可以是表结构中的一列。

读取

请求包含 Value 列时需要额外读取 `__DORIS_SEQUENCE_COL__` 列，该列用于在相同 Key 列下，REPLACE 聚合函数替换顺序的依据，较大值可以替换较小值，反之则不能替换。

Cumulative Compaction

Cumulative Compaction 时和读取过程原理相同。

Base Compaction

Base Compaction 时读取过程原理相同。

使用语法

建表时语法方面在 Property 中增加了一个属性，用来标识 `__DORIS_SEQUENCE_COL__` 的类型

导入的语法设计方面主要是增加一个从 sequence 列到其他 column 的映射，各个导入方式设置的将在下面介绍。

建表

创建 Unique 表时，可以指定 sequence 列类型。

```
PROPERTIES (
    "function_column.sequence_type" = 'Date',
);
```

sequence_type 用来指定 sequence 列的类型，可以为整型和时间类型。

Stream load

Stream load 的写法是在 header 中的 `function_column.sequence_col` 字段添加隐藏列对应的 source_sequence 的映射， 示例：

```
curl --location-trusted -u root -H "columns: k1,k2,source_sequence,v1,v2" -H "funct
```

Broker load

在 `ORDER BY` 处设置隐藏列映射的 source_sequence 字段。

```
LOAD LABEL db1.label1
(
    DATA INFILE("hdfs://host:port/user/data/*/test.txt")
    INTO TABLE `tbl1`
    COLUMNS TERMINATED BY ","
    (k1,k2,source_sequence,v1,v2)
    ORDER BY source_sequence
)
WITH BROKER 'broker'
(
    "username"="user",
    "password"="pass"
)
PROPERTIES
(
    "timeout" = "3600"
);
```

Routine load

映射方式同上，示例如下：

```
    CREATE ROUTINE LOAD example_db.test1 ON example_tbl
     [WITH MERGE|APPEND|DELETE]
     COLUMNS(k1, k2, source_sequence, v1, v2),
     WHERE k1 > 100 and k2 like "%doris%"
     [ORDER BY source_sequence]
     PROPERTIES
     (
         "desired_concurrent_number"="3",
         "max_batch_interval" = "20",
         "max_batch_rows" = "300000",
         "max_batch_size" = "209715200",
         "strict_mode" = "false"
     )
     FROM KAFKA
     (
         "kafka_broker_lsequence_typeist" = "broker1:9092,broker2:9092,broker3:9092"
         "kafka_topic" = "my_topic",
         "kafka_partitions" = "0,1,2,3",
         "kafka_offsets" = "101,0,0,200"
     );
```

启用 sequence column 支持

在新建表时如果设置了 `function_column.sequence_type` ，则新建表将支持 sequence column。

对于一个不支持 sequence column 的表，如果想要使用该功能，可以使用如下语句来启用。

```
  ALTER TABLE example_db.my_table ENABLE FEATURE "SEQUENCE_LOAD" WITH PROPERTIES ("fu
```

如果确定一个表是否支持 sequence column，可以通过设置一个 session variable来显示隐藏列 `SET show_hidden_columns=true` ，之后使用 `desc tablename` ，如果输出中有 `__DORIS_SEQUENCE_COL__` 列则支持，如果没有则不支持。

使用示例

下面以 Stream load 为例 展示下使用方式

1. 创建支持 sequence column 的表。

表结构如下：

```
  MySQL > desc test_table;
```

```
+-------------+--------------+------+-------+---------+---------+
| Field       | Type         | Null | Key   | Default | Extra   |
+-------------+--------------+------+-------+---------+---------+
| user_id     | BIGINT       | No   | true  | NULL    |         |
| date        | DATE         | No   | true  | NULL    |         |
| group_id    | BIGINT       | No   | true  | NULL    |         |
| modify_date | DATE         | No   | false | NULL    | REPLACE |
| keyword     | VARCHAR(128) | No   | false | NULL    | REPLACE |
+-------------+--------------+------+-------+---------+---------+
```

2. 正常导入数据。

导入如下数据：

```
1       2020-02-22      1       2020-02-22      a
1       2020-02-22      1       2020-02-22      b
1       2020-02-22      1       2020-03-05      c
1       2020-02-22      1       2020-02-26      d
1       2020-02-22      1       2020-02-22      e
1       2020-02-22      1       2020-02-22      b
```

此处以 Stream load 为例， 将 sequence column 映射为 modify_date 列。

```
curl --location-trusted -u root: -H "function_column.sequence_col: modify_date" -T
```

结果为：

```
MySQL > select * from test_table;
+---------+------------+----------+-------------+---------+
| user_id | date       | group_id | modify_date | keyword |
+---------+------------+----------+-------------+---------+
|       1 | 2020-02-22 |        1 | 2020-03-05  | c       |
+---------+------------+----------+-------------+---------+
```

在这次导入中，因 sequence column 的值（也就是 modify_date 中的值）中'2020-03-05'为最大值，所以 keyword 列中最终保留了 c。

3. 替换顺序的保证。

上述步骤完成后，接着导入如下数据：

```
1       2020-02-22      1       2020-02-22      a
1       2020-02-22      1       2020-02-23      b
```

查询数据：

```
MySQL [test]> select * from test_table;
+---------+------------+----------+-------------+---------+
| user_id | date       | group_id | modify_date | keyword |
+---------+------------+----------+-------------+---------+
```

```
|       1 | 2020-02-22 |        1 | 2020-03-05  | c        |
+---------+------------+----------+-------------+----------+
```

由于新导入的数据的 sequence column 都小于表中已有的值，无法替换。

再尝试导入如下数据：

```
1       2020-02-22      1       2020-02-22      a
1       2020-02-22      1       2020-03-23      w
```

查询数据：

```
MySQL [test]> select * from test_table;
+---------+------------+----------+-------------+----------+
| user_id | date       | group_id | modify_date | keyword |
+---------+------------+----------+-------------+----------+
|       1 | 2020-02-22 |        1 | 2020-03-23  | w        |
+---------+------------+----------+-------------+----------+
```

此时就可以替换表中原有的数据。

常见问题

对于指定了 function_column.sequence_type 的表，导入时没有指定 sequence_col 会怎么样?

导入任务将不会执行 sequence 列相关的判断逻辑，造成导入结果与预期不符。因此**强烈建议**使用设置 function_column.sequence_col 的方式使用 sequence列功能，避免因为忘记指定 sequence_col 列造成脏数据。

表结构变更

最近更新时间：2024-06-27 11:08:42

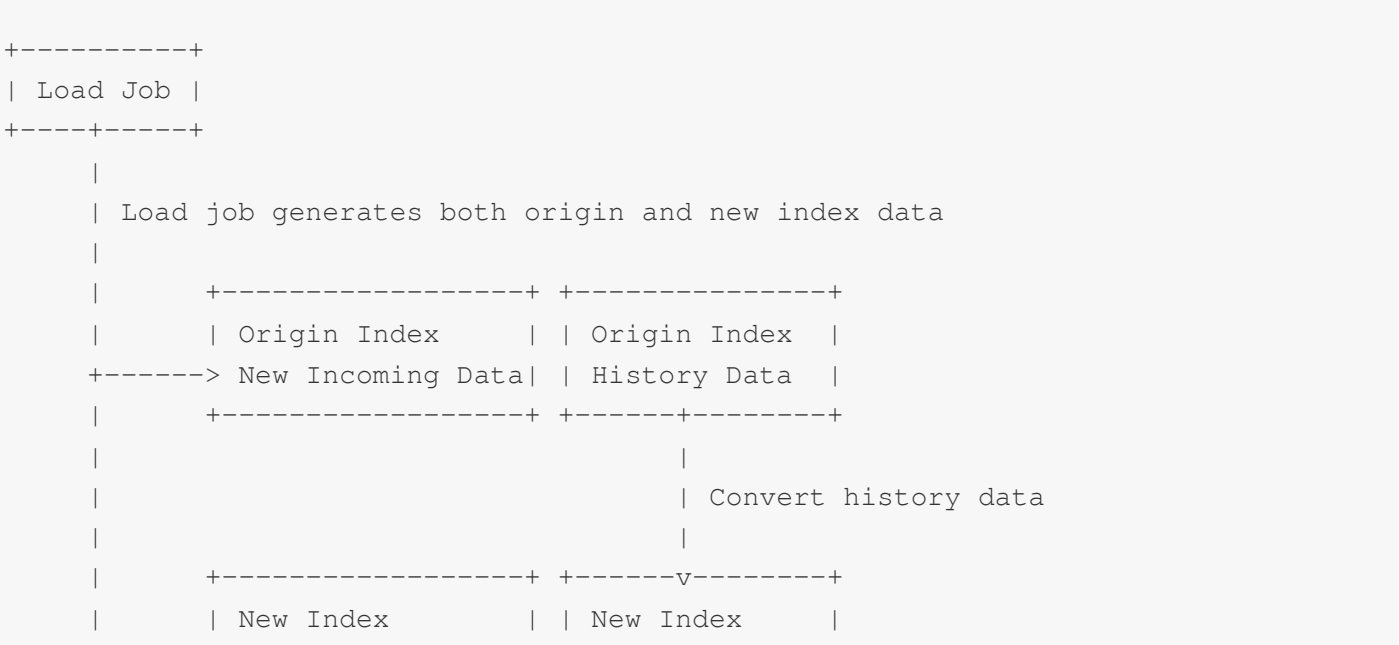
用户可以通过 Schema Change 操作来修改已存在表的 Schema。目前 Doris 支持以下几种修改:
增加、删除列
修改列类型
调整列顺序
增加、修改 Bloom Filter
增加、删除 bitmap index
本文档主要介绍如何创建 Schema Change 作业，以及进行 Schema Change 的一些注意事项和常见问题。

名词解释

Base Table：基表。每一个表被创建时，都对应一个基表。
Rollup：基于基表或者其他 Rollup 创建出来的上卷表。
Index：物化索引。Rollup 或 Base Table 都被称为物化索引。
Transaction：事务。每一个导入任务都是一个事务，每个事务有一个唯一递增的 Transaction ID。

原理介绍

执行 Schema Change 的基本过程，是通过原 Index 的数据，生成一份新 Schema 的 Index 的数据。其中主要需要进行两部分数据转换，一是已存在的历史数据的转换，二是在 Schema Change 执行过程中，新到达的导入数据的转换。



```
      +------> New Incoming Data|  | History Data   |
             +-------------------+ +----------------+
```

在开始转换历史数据之前，Doris 会获取一个最新的 Transaction ID。并等待这个 Transaction ID 之前的所有导入事务完成。这个 Transaction ID 成为分水岭。意思是，Doris 保证在分水岭之后的所有导入任务，都会同时为原 Index 和新 Index 生成数据。这样当历史数据转换完成后，可以保证新的 Index 中的数据是完整的。

创建作业

创建 Schema Change 的具体语法可以查看帮助 `HELP ALTER TABLE` 中 Schema Change 部分的说明。Schema Change 的创建是一个异步过程，作业提交成功后，用户需要通过 `SHOW ALTER TABLE COLUMN` 命令来查看作业进度。

查看作业

`SHOW ALTER TABLE COLUMN` 可以查看当前正在执行或已经完成的 Schema Change 作业。当一次 Schema Change 作业涉及到多个 Index 时，该命令会显示多行，每行对应一个 Index。举例如下：

```
         JobId: 20021
     TableName: tbl1
    CreateTime: 2019-08-05 23:03:13
    FinishTime: 2019-08-05 23:03:42
     IndexName: tbl1
       IndexId: 20022
OriginIndexId: 20017
SchemaVersion: 2:792557838
TransactionId: 10023
         State: FINISHED
           Msg:
      Progress: N/A
       Timeout: 86400
```

JobId：每个 Schema Change 作业的唯一 ID。

TableName：Schema Change 对应的基表的表名。

CreateTime：作业创建时间。

FinishedTime：作业结束时间。如未结束，则显示 "N/A"。

IndexName： 本次修改所涉及的某一个 Index 的名称。

IndexId：新的 Index 的唯一 ID。

OriginIndexId：旧的 Index 的唯一 ID。

SchemaVersion：以 M:N 的格式展示。其中 M 表示本次 Schema Change 变更的版本，N 表示对应的 Hash 值。每次 Schema Change，版本都会递增。

TransactionId：转换历史数据的分水岭 transaction ID。

State：作业所在阶段。

PENDING：作业在队列中等待被调度。

WAITING_TXN：等待分水岭 transaction ID 之前的导入任务完成。

RUNNING：历史数据转换中。

FINISHED：作业成功。

CANCELLED：作业失败。

Msg：如果作业失败，这里会显示失败信息。

Progress：作业进度。只有在 RUNNING 状态才会显示进度。进度是以 M/N 的形式显示。其中 N 为 Schema Change 涉及的总副本数。M 为已完成历史数据转换的副本数。

Timeout：作业超时时间。单位秒。

取消作业

在作业状态不为 FINISHED 或 CANCELLED 的情况下，可以通过以下命令取消 Schema Change 作业：

```
CANCEL ALTER TABLE COLUMN FROM tbl_name;
```

最佳实践

Schema Change 可以在一个作业中，对多个 Index 进行不同的修改。举例如下：

源 Schema：

```
+-----------+-------+------+------+------+---------+-------+
| IndexName | Field | Type | Null | Key  | Default | Extra |
+-----------+-------+------+------+------+---------+-------+
| tbl1      | k1    | INT  | No   | true | N/A     |       |
|           | k2    | INT  | No   | true | N/A     |       |
|           | k3    | INT  | No   | true | N/A     |       |
|           |       |      |      |      |         |       |
| rollup2   | k2    | INT  | No   | true | N/A     |       |
|           |       |      |      |      |         |       |
| rollup1   | k1    | INT  | No   | true | N/A     |       |
|           | k2    | INT  | No   | true | N/A     |       |
+-----------+-------+------+------+------+---------+-------+
```

可以通过以下命令给 rollup1 和 rollup2 都加入一列 k4，并且再给 rollup2 加入一列 k5：

```
ALTER TABLE tbl1
```

```
ADD COLUMN k4 INT default "1" to rollup1,
ADD COLUMN k4 INT default "1" to rollup2,
ADD COLUMN k5 INT default "1" to rollup2;
```

完成后，Schema 变为：

```
+-----------+-------+------+------+------+---------+-------+
| IndexName | Field | Type | Null | Key  | Default | Extra |
+-----------+-------+------+------+------+---------+-------+
| tbl1      | k1    | INT  | No   | true | N/A     |       |
|           | k2    | INT  | No   | true | N/A     |       |
|           | k3    | INT  | No   | true | N/A     |       |
|           | k4    | INT  | No   | true | 1       |       |
|           | k5    | INT  | No   | true | 1       |       |
|           |       |      |      |      |         |       |
| rollup2   | k2    | INT  | No   | true | N/A     |       |
|           | k4    | INT  | No   | true | 1       |       |
|           | k5    | INT  | No   | true | 1       |       |
|           |       |      |      |      |         |       |
| rollup1   | k1    | INT  | No   | true | N/A     |       |
|           | k2    | INT  | No   | true | N/A     |       |
|           | k4    | INT  | No   | true | 1       |       |
+-----------+-------+------+------+------+---------+-------+
```

可以看到，Base 表 tbl1 也自动加入了 k4, k5 列。即给任意 rollup 增加的列，都会自动加入到 Base 表中。

同时，不允许向 Rollup 中加入 Base 表已经存在的列。如果用户需要这样做，可以重新建立一个包含新增列的 Rollup，之后再删除原 Rollup。

注意事项

一张表在同一时间只能有一个 Schema Change 作业在运行。

Schema Change 操作不阻塞导入和查询操作。

分区列和分桶列不能修改。

如果 Schema 中有 REPLACE 方式聚合的 value 列，则不允许删除 Key 列。

如果删除 Key 列，Doris 无法决定 REPLACE 列的取值。

Unique 数据模型表的所有非 Key 列都是 REPLACE 聚合方式。

在新增聚合类型为 SUM 或者 REPLACE 的 value 列时，该列的默认值对历史数据没有含义。

因为历史数据已经失去明细信息，所以默认值的取值并不能实际反映聚合后的取值。

当修改列类型时，除 Type 以外的字段都需要按原列上的信息补全。

如修改列 `k1 INT SUM NULL DEFAULT "1"` 类型为 BIGINT，则需执行命令如下：

`ALTER TABLE tbl1 MODIFY COLUMN k1 BIGINT SUM NULL DEFAULT "1";`

注意

除新的列类型外，如聚合方式，Nullable 属性，以及默认值都要按照原信息补全。
不支持修改列名称、聚合类型、Nullable 属性、默认值以及列注释。

常见问题

Schema Change 的执行速度。
目前 Schema Change 执行速度按照最差效率估计约为 10MB/s。保守起见，用户可以根据这个速率来设置作业的超时时间。
提交作业报错 `Table xxx is not stable. ...` 。
Schema Change 只有在表数据完整且非均衡状态下才可以开始。如果表的某些数据分片副本不完整，或者某些副本正在进行均衡操作，则提交会被拒绝。
数据分片副本是否完整，可以通过以下命令查看：`ADMIN SHOW REPLICA STATUS FROM tbl WHERE STATUS != "OK";` 。
如果有返回结果，则说明有副本有问题。通常系统会自动修复这些问题，用户也可以通过以下命令优先修复这个表：`ADMIN REPAIR TABLE tbl1;` 。
用户可以通过以下命令查看是否有正在运行的均衡任务：`SHOW PROC "/cluster_balance/pending_tablets";` 。
可以等待均衡任务完成，或者通过以下命令临时禁止均衡操作：`ADMIN SET FRONTEND CONFIG ("disable_balance" = "true");` 。

相关配置

FE 配置

`alter_table_timeout_second` ：作业默认超时时间，86400 秒。

BE 配置

`alter_tablet_worker_count` ：在 BE 端用于执行历史数据转换的线程数。默认为 3。如果希望加快 Schema Change 作业的速度，可以适当调大这个参数后重启 BE。但过多的转换线程可能会导致 IO 压力增加，影响其他操作。该线程和 Rollup 作业共用。

原子交换（Swap）两个表

最近更新时间：2024-06-27 11:08:55

自0.14 版本起，Doris 支持对两个表进行原子的替换操作。 该操作仅适用于 OLAP 表。分区级别的替换操作，请参见 临时分区 文档。

语法说明

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

将表 tbl1 替换为表 tbl2。

如果 `swap` 参数为 `true`，则替换后，名称为 `tbl1` 表中的数据为原 `tbl2` 表中的数据。而名称为 `tbl2` 表中的数据为原 `tbl1` 表中的数据。即两张表数据发生了互换。

如果 `swap` 参数为 `false`，则替换后，名称为 `tbl1` 表中的数据为原 `tbl2` 表中的数据。而名称为 `tbl2` 表被删除。

原理

替换表功能，实际上是将以下操作集合变成一个原子操作。

假设要将表 A 替换为表 B，且 `swap` 为 `true`，则操作如下：

1. 将表 B 重名为表 A。
2. 将表 A 重名为表 B。

如果 `swap` 为 `false`，则操作如下：

1. 删除表 A。
2. 将表 B 重名为表 A。

注意事项

1. `swap` 参数默认为 `true`。即替换表操作相当于将两张表数据进行交换。
2. 如果设置 `swap` 参数为 `false`，则被替换的表（表A）将被删除，且无法恢复。
3. 替换操作仅能发生在两张 OLAP 表之间，且不会检查两张表的表结构是否一致。
4. 替换操作不会改变原有的权限设置。因为权限检查以表名称为准。

最佳实践

1. 原子的覆盖写操作。

某些情况下，用户希望能够重写某张表的数据，但如果采用先删除再导入的方式进行，在中间会有一段时间无法查看数据。这时，用户可以先使用 `CREATE TABLE LIKE` 语句创建一个相同结构的新表，将新的数据导入到新表后，通过替换操作，原子的替换旧表，以达到目的。分区级别的原子覆盖写操作，请参见 临时分区 文档。

动态分区

最近更新时间：2024-06-27 11:09:18

动态分区旨在对表级别的分区实现生命周期管理(TTL)，减少用户的使用负担。目前实现了动态添加分区及动态删除分区的功能。
动态分区只支持 Range 分区。

原理

在某些使用场景下，用户会将表按照天进行分区划分，每天定时执行例行任务，这时需要使用方手动管理分区，否则可能由于使用方没有创建分区导致数据导入失败，这给使用方带来了额外的维护成本。
通过动态分区功能，用户可以在建表时设定动态分区的规则。FE 会启动一个后台线程，根据用户指定的规则创建或删除分区。用户也可以在运行时对现有规则进行变更。

使用方式

动态分区的规则可以在建表时指定，或者在运行时修改。当前仅支持对单分区列的分区表设定动态分区规则。
建表时指定：

```
CREATE TABLE tbl1
(...)
PROPERTIES
(
      "dynamic_partition.prop1" = "value1",
      "dynamic_partition.prop2" = "value2",
      ...
)
```

运行时修改：

```
ALTER TABLE tbl1 SET
(
      "dynamic_partition.prop1" = "value1",
      "dynamic_partition.prop2" = "value2",
      ...
)
```

动态分区规则参数

动态分区的规则参数都以 `dynamic_partition.` 为前缀：

主要参数

`dynamic_partition.enable`
是否开启动态分区特性。可指定为 `TRUE` 或 `FALSE` 。如果不填写，默认为 `TRUE` 。如果为 `FALSE` ，则 Doris 会忽略该表的动态分区规则。

`dynamic_partition.time_unit`
动态分区调度的单位。可指定为 `HOUR` 、 `DAY` 、 `WEEK` 、 `MONTH` 。分别表示按天、按星期、按月进行分区创建或删除。
当指定为 `HOUR` 时，动态创建的分区名后缀格式为 `yyyyMMddHH` ，例如 `2020032501` 。小时为单位的分区列数据类型不能为 DATE。
当指定为 `DAY` 时，动态创建的分区名后缀格式为 `yyyyMMdd` ，例如 `20200325` 。
当指定为 `WEEK` 时，动态创建的分区名后缀格式为 `yyyy_ww` 。即当前日期属于这一年的第几周，例如 `2020-03-25` 创建的分区名后缀为 `2020_13` , 表明目前为2020年第13周。
当指定为 `MONTH` 时，动态创建的分区名后缀格式为 `yyyyMM` ，例如 `202003` 。

`dynamic_partition.start`
动态分区的起始偏移，为负数。根据 `time_unit` 属性的不同，以当天（星期/月）为基准，分区范围在此偏移之前的分区将会被删除。如果不填写，则默认为 `-2147483648` ，即不删除历史分区。

`dynamic_partition.end`
动态分区的结束偏移，为正数。根据 `time_unit` 属性的不同，以当天（星期/月）为基准，提前创建对应范围的分区。

`dynamic_partition.buckets`
动态创建的分区所对应的分桶数量。

`dynamic_partition.replication_num`
动态创建的分区所对应的副本数量，如果不填写，则默认为该表创建时指定的副本数量。

其他参数

`dynamic_partition.prefix`
动态创建的分区名前缀。

`dynamic_partition.time_zone`
动态分区的时区，如果不填写，则默认为当前机器的系统的时区，例如 `Asia/Shanghai` ，如果想获取当前支持的时区设置，可以参考 `https://en.wikipedia.org/wiki/List_of_tz_database_time_zones` 。

`dynamic_partition.start_day_of_week`
当 `time_unit` 为 `WEEK` 时，该参数用于指定每周的起始点。取值为 1 到 7。其中 1 表示周一，7 表示周日。默认为 1，即表示每周以周一为起始点。

`dynamic_partition.start_day_of_month`
当 `time_unit` 为 `MONTH` 时，该参数用于指定每月的起始日期。取值为 1 到 28。其中 1 表示每月1号，28 表示每月28号。默认为 1，即表示每月以1号位起始点。暂不支持以29、30、31号为起始日，以避免因闰年或闰月带来的歧义。

`dynamic_partition.create_history_partition`

默认为 false。当置为 true 时，Doris 会自动创建所有分区，具体创建规则见下文。同时，FE 的参数 `max_dynamic_partition_num` 会限制总分区数量，以避免一次性创建过多分区。当期望创建的分区个数大于 `max_dynamic_partition_num` 值时，操作将被禁止。

当不指定 `start` 属性时，该参数不生效。

`dynamic_partition.history_partition_num`

当 `create_history_partition` 为 `true` 时，该参数用于指定创建历史分区数量。默认值为 -1， 即未设置。

`dynamic_partition.hot_partition_num`

指定最新的多少个分区为热分区。对于热分区，系统会自动设置其 `storage_medium` 参数为SSD，并且设置 `storage_cooldown_time` 。

我们举例说明。假设今天是 2021-05-20，按天分区，动态分区的属性设置为：hot_partition_num=2, end=3, start=-3。则系统会自动创建以下分区，并且设置 `storage_medium` 和 `storage_cooldown_time` 参数：

```
p20210517：["2021-05-17", "2021-05-18") storage_medium=HDD storage_cooldown_time=99
p20210518：["2021-05-18", "2021-05-19") storage_medium=HDD storage_cooldown_time=99
p20210519：["2021-05-19", "2021-05-20") storage_medium=SSD storage_cooldown_time=202
p20210520：["2021-05-20", "2021-05-21") storage_medium=SSD storage_cooldown_time=202
p20210521：["2021-05-21", "2021-05-22") storage_medium=SSD storage_cooldown_time=202
p20210522：["2021-05-22", "2021-05-23") storage_medium=SSD storage_cooldown_time=202
p20210523：["2021-05-23", "2021-05-24") storage_medium=SSD storage_cooldown_time=202
```

`dynamic_partition.reserved_history_periods`

需要保留的历史分区的时间范围。当 `dynamic_partition.time_unit` 设置为 "DAY/WEEK/MONTH" 时，需要以 `[yyyy-MM-dd,yyyy-MM-dd],[...,...]` 格式进行设置。当 `dynamic_partition.time_unit` 设置为 "HOUR" 时，需要以 `[yyyy-MM-dd HH:mm:ss,yyyy-MM-dd HH:mm:ss],[...,...]` 的格式来进行设置。如果不设置，默认为 `"NULL"` 。

我们举例说明。假设今天是 2021-09-06，按天分类，动态分区的属性设置为：

```
time_unit="DAY/WEEK/MONTH", end=3, start=-3, reserved_history_periods="[2020-06-01,
```

则系统会自动保留：

```
["2020-06-01","2020-06-20"],
["2020-10-31","2020-11-15"]
```

或者动态分区的属性设置为：

```
time_unit="HOUR", end=3, start=-3, reserved_history_periods="[2020-06-01 00:00:00,2
```

则系统会自动保留：

```
["2020-06-01 00:00:00","2020-06-01 03:00:00"]
```

这两个时间段的分区。其中，`reserved_history_periods` 的每一个 `[...,...]` 是一对设置项，两者需要同时被设置，且第一个时间不能大于第二个时间。

创建历史分区规则

当 `create_history_partition` 为 `true`，即开启创建历史分区功能时，Doris 会根据 `dynamic_partition.start` 和 `dynamic_partition.history_partition_num` 来决定创建历史分区的个数。

假设需要创建的历史分区数量为 `expect_create_partition_num`，根据不同的设置具体数量如下：

`create_history_partition` = `true`

`dynamic_partition.history_partition_num` 未设置，即 -1。

`expect_create_partition_num` = `end` - `start`；

`dynamic_partition.history_partition_num` 已设置。

`expect_create_partition_num` = `end` - max(`start`，`-history_partition_num`)。

`create_history_partition` = `false` 不会创建历史分区，`expect_create_partition_num` = `end` - 0;当 `expect_create_partition_num` 大于 `max_dynamic_partition_num`（默认500）时，禁止创建过多分区。

举例说明：

1. 假设今天是 2021-05-20，按天分区，动态分区的属性设置为：`create_history_partition=true, end=3, start=-3, history_partition_num=1`，则系统会自动创建以下分区：

```
p20210519
p20210520
p20210521
p20210522
p20210523
```

2. `history_partition_num=5`，其余属性与 1 中保持一致，则系统会自动创建以下分区：

```
p20210517
p20210518
p20210519
p20210520
p20210521
p20210522
p20210523
```

3. `history_partition_num=-1` 即不设置历史分区数量，其余属性与 1 中保持一致，则系统会自动创建以下分区：

```
p20210517
p20210518
p20210519
```

```
p20210520
p20210521
p20210522
p20210523
```

注意事项

动态分区使用过程中，如果因为一些意外情况导致 `dynamic_partition.start` 和 `dynamic_partition.end` 之间的某些分区丢失，那么当前时间与 `dynamic_partition.end` 之间的丢失分区会被重新创建， `dynamic_partition.start` 与当前时间之间的丢失分区不会重新创建。

示例

1. 表 tbl1 分区列 k1 类型为 DATE，创建一个动态分区规则。按天分区，只保留最近7天的分区，并且预先创建未来3天的分区。

```
CREATE TABLE tbl1
(
 k1 DATE,
 ...
)
PARTITION BY RANGE(k1) ()
DISTRIBUTED BY HASH(k1)
PROPERTIES
(
 "dynamic_partition.enable" = "true",
 "dynamic_partition.time_unit" = "DAY",
 "dynamic_partition.start" = "-7",
 "dynamic_partition.end" = "3",
 "dynamic_partition.prefix" = "p",
 "dynamic_partition.buckets" = "32"
);
```

假设当前日期为 2020-05-29。则根据以上规则，tbl1 会产生以下分区：

```
p20200529: ["2020-05-29", "2020-05-30")
p20200530: ["2020-05-30", "2020-05-31")
p20200531: ["2020-05-31", "2020-06-01")
p20200601: ["2020-06-01", "2020-06-02")
```

在第二天，即 2020-05-30，会创建新的分区 `p20200602: ["2020-06-02", "2020-06-03")`

在 2020-06-06 时，因为 `dynamic_partition.start` 设置为 7，则将删除7天前的分区，即删除分区 `p20200529` 。

2. 表 tbl1 分区列 k1 类型为 DATETIME，创建一个动态分区规则。按星期分区，只保留最近2个星期的分区，并且预先创建未来2个星期的分区。

```
CREATE TABLE tbl1
(
    k1 DATETIME,
    ...
)
PARTITION BY RANGE(k1) ()
DISTRIBUTED BY HASH(k1)
PROPERTIES
(
    "dynamic_partition.enable" = "true",
    "dynamic_partition.time_unit" = "WEEK",
    "dynamic_partition.start" = "-2",
    "dynamic_partition.end" = "2",
    "dynamic_partition.prefix" = "p",
    "dynamic_partition.buckets" = "8"
);
```

假设当前日期为 2020-05-29，是 2020 年的第 22 周。默认每周起始为星期一。则根于以上规则，tbl1 会产生以下分区：

```
p2020_22: ["2020-05-25 00:00:00", "2020-06-01 00:00:00")
p2020_23: ["2020-06-01 00:00:00", "2020-06-08 00:00:00")
p2020_24: ["2020-06-08 00:00:00", "2020-06-15 00:00:00")
```

其中每个分区的起始日期为当周的周一。同时，因为分区列 k1 的类型为 DATETIME，则分区值会补全时分秒部分，且皆为 0。

在 2020-06-15，即第25周时，会删除2周前的分区，即删除 `p2020_22` 。

在上面的例子中，假设用户指定了周起始日为 `"dynamic_partition.start_day_of_week" = "3"` ，即以每周三为起始日。则分区如下：

```
p2020_22: ["2020-05-27 00:00:00", "2020-06-03 00:00:00")
p2020_23: ["2020-06-03 00:00:00", "2020-06-10 00:00:00")
p2020_24: ["2020-06-10 00:00:00", "2020-06-17 00:00:00")
```

即分区范围为当周的周三到下周的周二。

注意

2019-12-31 和 2020-01-01 在同一周内，如果分区的起始日期为 2019-12-31，则分区名为 `p2019_53` ，如果分区的起始日期为 2020-01-01，则分区名为 `p2020_01` 。

3. 表 tbl1 分区列 k1 类型为 DATE，创建一个动态分区规则。按月分区，不删除历史分区，并且预先创建未来2个月的分区。同时设定以每月3号为起始日。

```
CREATE TABLE tbl1
```

```
(
     k1 DATE,
     ...
)
PARTITION BY RANGE(k1) ()
DISTRIBUTED BY HASH(k1)
PROPERTIES
(
     "dynamic_partition.enable" = "true",
     "dynamic_partition.time_unit" = "MONTH",
     "dynamic_partition.end" = "2",
     "dynamic_partition.prefix" = "p",
     "dynamic_partition.buckets" = "8",
     "dynamic_partition.start_day_of_month" = "3"
);
```

假设当前日期为 2020-05-29。则根于以上规则，tbl1 会产生以下分区：

```
p202005: ["2020-05-03", "2020-06-03")
p202006: ["2020-06-03", "2020-07-03")
p202007: ["2020-07-03", "2020-08-03")
```

因为没有设置 `dynamic_partition.start`，则不会删除历史分区。

假设今天为 2020-05-20，并设置以每月28号为起始日，则分区范围为：

```
p202004: ["2020-04-28", "2020-05-28")
p202005: ["2020-05-28", "2020-06-28")
p202006: ["2020-06-28", "2020-07-28")
```

修改动态分区属性

通过如下命令可以修改动态分区的属性：

```
ALTER TABLE tbl1 SET
(
    "dynamic_partition.prop1" = "value1",
    ...
);
```

某些属性的修改可能会产生冲突。假设之前分区粒度为 DAY，并且已经创建了如下分区：

```
p20200519: ["2020-05-19", "2020-05-20")
p20200520: ["2020-05-20", "2020-05-21")
p20200521: ["2020-05-21", "2020-05-22")
```

如果此时将分区粒度改为 MONTH，则系统会尝试创建范围为 `["2020-05-01", "2020-06-01")` 的分区，而该分区的分区范围和已有分区冲突，所以无法创建。而范围为 `["2020-06-01", "2020-07-01")` 的分区可以正常创建。因此，2020-05-22 到 2020-05-30 时间段的分区，需要自行填补。

查看动态分区表调度情况

通过以下命令可以进一步查看当前数据库下，所有动态分区表的调度情况：

```
mysql> SHOW DYNAMIC PARTITION TABLES;
+-----------+--------+----------+-------------+------+--------+---------+----------
| TableName | Enable | TimeUnit | Start       | End  | Prefix | Buckets | StartOf
+-----------+--------+----------+-------------+------+--------+---------+----------
| d3        | true   | WEEK     | -3          | 3    | p      | 1       | MONDAY
| d5        | true   | DAY      | -7          | 3    | p      | 32      | N/A
| d4        | true   | WEEK     | -3          | 3    | p      | 1       | WEDNESDAY
| d6        | true   | MONTH    | -2147483648 | 2    | p      | 8       | 3rd
| d2        | true   | DAY      | -3          | 3    | p      | 32      | N/A
| d7        | true   | MONTH    | -2147483648 | 5    | p      | 8       | 24th
+-----------+--------+----------+-------------+------+--------+---------+----------
7 rows in set (0.02 sec)
```

LastUpdateTime: 最后一次修改动态分区属性的时间。

LastSchedulerTime: 最后一次执行动态分区调度的时间。

State: 最后一次执行动态分区调度的状态。

LastCreatePartitionMsg: 最后一次执行动态添加分区调度的错误信息。

LastDropPartitionMsg: 最后一次执行动态删除分区调度的错误信息。

高级操作

FE 配置项

dynamic_partition_enable

是否开启 Doris 的动态分区功能。默认为 false，即关闭。该参数只影响动态分区表的分区操作，不影响普通表。可以通过修改 fe.conf 中的参数并重启 FE 生效。也可以在运行时执行以下命令生效：

MySQL 协议：

`ADMIN SET FRONTEND CONFIG ("dynamic_partition_enable" = "true")`

HTTP 协议：

`curl --location-trusted -u username:password -XGET`

`http://fe_host:fe_http_port/api/_set_config?dynamic_partition_enable=true`

若要全局关闭动态分区，则设置此参数为 false 即可。

dynamic_partition_check_interval_seconds

动态分区线程的执行频率，默认为600(10分钟)，即每10分钟进行一次调度。可以通过修改 fe.conf 中的参数并重启

FE 生效。也可以在运行时执行以下命令修改：

MySQL 协议：

```
ADMIN SET FRONTEND CONFIG ("dynamic_partition_check_interval_seconds" = "7200")
```

HTTP 协议：

```
curl --location-trusted -u username:password -XGET
http://fe_host:fe_http_port/api/_set_config?
dynamic_partition_check_interval_seconds=432000
```

动态分区表与手动分区表相互转换

对于一个表来说，动态分区和手动分区可以自由转换，但二者不能同时存在，有且只有一种状态。

手动分区转换为动态分区

如果一个表在创建时未指定动态分区，可以通过 `ALTER TABLE` 在运行时修改动态分区相关属性来转化为动态分区，具体示例可以通过 `HELP ALTER TABLE` 查看。

开启动态分区功能后，Doris 将不再允许用户手动管理分区，会根据动态分区属性来自动管理分区。

注意

如果已设定 `dynamic_partition.start` ，分区范围在动态分区起始偏移之前的历史分区将会被删除。

动态分区转换为手动分区

通过执行 `ALTER TABLE tbl_name SET ("dynamic_partition.enable" = "false")` 即可关闭动态分区功能，将其转换为手动分区表。

关闭动态分区功能后，Doris 将不再自动管理分区，需要用户手动通过 `ALTER TABLE` 的方式创建或删除分区。

常见问题

1. 创建动态分区表后提示 `Could not create table with dynamic partition when fe config dynamic_partition_enable is false` 。

由于动态分区的总开关，也就是 FE 的配置 `dynamic_partition_enable` 为 false，导致无法创建动态分区表。

这时候请修改 FE 的配置文件，增加一行 `dynamic_partition_enable=true` ，并重启 FE。或者执行命令 ADMIN SET FRONTEND CONFIG ("dynamic_partition_enable" = "true") 将动态分区开关打开即可。

2. 动态分区的分桶和副本参数和表的对应参数的关系

2.1 动态分区的分桶和副本参数独立于表的分桶和副本参数，相互之间不影响，需要独立设置。动态创建分区使用动态分区的分桶和副本参数，手动创建分区使用表的分桶和副本参数。

2.2 只有在开启动态分区没有设置分桶和副本参数时，会将这两个参数设置为表的分桶和副本参数。设置完成后，两套参数相互独立。

3. 开启动态分区后能否手动编辑分区

不能，需要关闭动态分区功能后才能手动编辑分区。

临时分区

最近更新时间：2024-06-27 11:09:33

在 0.12 版本中，Doris 支持了临时分区功能。临时分区是归属于某一分区表的。只有分区表可以创建临时分区。

规则

临时分区的分区列和正式分区相同，且不可修改。
一张表所有临时分区之间的分区范围不可重叠，但临时分区的范围和正式分区范围可以重叠。
临时分区的分区名称不能和正式分区以及其他临时分区重复。

支持的操作

临时分区支持添加、删除、替换操作。

添加临时分区

可以通过 `ALTER TABLE ADD TEMPORARY PARTITION` 语句对一个表添加临时分区：

```
ALTER TABLE tbl1 ADD TEMPORARY PARTITION tp1 VALUES LESS THAN("2020-02-01");

ALTER TABLE tbl2 ADD TEMPORARY PARTITION tp1 VALUES [("2020-01-01"), ("2020-02-01")

ALTER TABLE tbl1 ADD TEMPORARY PARTITION tp1 VALUES LESS THAN("2020-02-01")
("in_memory" = "true", "replication_num" = "1")
DISTRIBUTED BY HASH(k1) BUCKETS 5;

ALTER TABLE tbl3 ADD TEMPORARY PARTITION tp1 VALUES IN ("Beijing", "Shanghai");

ALTER TABLE tbl4 ADD TEMPORARY PARTITION tp1 VALUES IN ((1, "Beijing"), (1, "Shangh

ALTER TABLE tbl3 ADD TEMPORARY PARTITION tp1 VALUES IN ("Beijing", "Shanghai")
("in_memory" = "true", "replication_num" = "1")
DISTRIBUTED BY HASH(k1) BUCKETS 5;
```

通过 `HELP ALTER TABLE;` 查看更多帮助和示例。
添加操作的一些说明：
临时分区的添加和正式分区的添加操作相似。临时分区的分区范围独立于正式分区。
临时分区可以独立指定一些属性。包括分桶数、副本数、是否是内存表、存储介质等信息。

删除临时分区

可以通过 `ALTER TABLE DROP TEMPORARY PARTITION` 语句删除一个表的临时分区：

```
ALTER TABLE tbl1 DROP TEMPORARY PARTITION tp1;
```

通过 `HELP ALTER TABLE;` 查看更多帮助和示例。

说明

删除临时分区，不影响正式分区的数据。

替换分区

可以通过 `ALTER TABLE REPLACE PARTITION` 语句将一个表的正式分区替换为临时分区。

```
ALTER TABLE tbl1 REPLACE PARTITION (p1) WITH TEMPORARY PARTITION (tp1);

ALTER TABLE tbl1 REPLACE PARTITION (p1, p2) WITH TEMPORARY PARTITION (tp1, tp2, tp3

ALTER TABLE tbl1 REPLACE PARTITION (p1, p2) WITH TEMPORARY PARTITION (tp1, tp2)
PROPERTIES (
    "strict_range" = "false",
    "use_temp_partition_name" = "true"
);
```

通过 `HELP ALTER TABLE;` 查看更多帮助和示例。

替换操作有两个特殊的可选参数：

1. `strict_range`

默认为 true。对于 Range 分区，当该参数为 true 时，表示要被替换的所有正式分区的范围并集需要和替换的临时分区的范围并集完全相同。当置为 false 时，只需要保证替换后，新的正式分区间的范围不重叠即可。

对于 List 分区，该参数恒为 true。要被替换的所有正式分区的枚举值必须和替换的临时分区枚举值完全相同。下面举例说明：

示例1

待替换的分区 p1, p2, p3 的范围 (=> 并集)：

```
[10, 20), [20, 30), [40, 50) => [10, 30), [40, 50)
```

替换分区 tp1, tp2 的范围(=> 并集)：

```
[10, 30), [40, 45), [45, 50) => [10, 30), [40, 50)
```

范围并集相同，则可以使用 tp1 和 tp2 替换 p1, p2, p3。

示例2

待替换的分区 p1 的范围 (=> 并集)：

```
[10, 50) => [10, 50)
```

替换分区 tp1, tp2 的范围(=> 并集)：

```
[10, 30), [40, 50) => [10, 30), [40, 50)
```

范围并集不相同，如果 `strict_range` 为 true，则不可以使用 tp1 和 tp2 替换 p1。如果为 false，且替换后的两个分区范围 `[10, 30), [40, 50)` 和其他正式分区不重叠，则可以替换。

示例3

待替换的分区 p1, p2 的枚举值(=> 并集)：

```
(1, 2, 3), (4, 5, 6) => (1, 2, 3, 4, 5, 6)
```

替换分区 tp1, tp2, tp3 的枚举值(=> 并集)：

```
(1, 2, 3), (4), (5, 6) => (1, 2, 3, 4, 5, 6)
```

枚举值并集相同，可以使用 tp1，tp2，tp3 替换 p1，p2。

示例4

待替换的分区 p1, p2，p3 的枚举值(=> 并集)：

```
(("1","beijing"), ("1", "shanghai")), (("2","beijing"), ("2", "shanghai")), (("3","
```

替换分区 tp1, tp2 的枚举值(=> 并集)：

```
(("1","beijing"), ("1", "shanghai")), (("2","beijing"), ("2", "shanghai"), ("3","be
```

枚举值并集相同，可以使用 tp1，tp2 替换 p1，p2，p3。

2. `use_temp_partition_name`

默认为 false。当该参数为 false，并且待替换的分区和替换分区的个数相同时，则替换后的正式分区名称维持不变。

如果为 true，则替换后，正式分区的名称为替换分区的名称。下面举例说明：

示例1

```
ALTER TABLE tbl1 REPLACE PARTITION (p1) WITH TEMPORARY PARTITION (tp1);
```

`use_temp_partition_name` 默认为 false，则在替换后，分区的名称依然为 p1，但是相关的数据和属性都替换为 tp1 的。

如果 `use_temp_partition_name` 默认为 true，则在替换后，分区的名称为 tp1。p1 分区不再存在。

示例2

```
ALTER TABLE tbl1 REPLACE PARTITION (p1, p2) WITH TEMPORARY PARTITION (tp1);
```

`use_temp_partition_name` 默认为 false，但因为待替换分区的个数和替换分区的个数不同，则该参数无效。

替换后，分区名称为 tp1，p1 和 p2 不再存在。

说明

分区替换成功后，被替换的分区将被删除且不可恢复。

临时分区的导入和查询

用户可以将数据导入到临时分区，也可以指定临时分区进行查询。

1. 导入临时分区。

根据导入方式的不同，指定导入临时分区的语法稍有差别。这里通过示例进行简单说明。

```
INSERT INTO tbl TEMPORARY PARTITION(tp1, tp2, ...) SELECT ....
```

```
curl --location-trusted -u root: -H "label:123" -H "temporary_partitions: tp1, tp2,
```

```
LOAD LABEL example_db.label1
(
DATA INFILE("hdfs://hdfs_host:hdfs_port/user/palo/data/input/file")
INTO TABLE `my_table`
TEMPORARY PARTITION (tp1, tp2, ...)
...
)
WITH BROKER hdfs ("username"="hdfs_user", "password"="hdfs_password");
```

```
CREATE ROUTINE LOAD example_db.test1 ON example_tbl
COLUMNS(k1, k2, k3, v1, v2, v3 = k1 * 100),
TEMPORARY PARTITIONS(tp1, tp2, ...),
WHERE k1 > 100
PROPERTIES
(...)
FROM KAFKA
(...);
```

2. 查询临时分区。

```
SELECT ... FROM
tbl1 TEMPORARY PARTITION(tp1, tp2, ...)
JOIN
tbl2 TEMPORARY PARTITION(tp1, tp2, ...)
ON ...
WHERE ...;
```

和其他操作的关系

DROP

使用 Drop 操作直接删除数据库或表后，可以通过 Recover 命令恢复数据库或表（限定时间内），但临时分区不会被恢复。
使用 Alter 命令删除正式分区后，可以通过 Recover 命令恢复分区（限定时间内）。操作正式分区和临时分区无关。
使用 Alter 命令删除临时分区后，无法通过 Recover 命令恢复临时分区。

TRUNCATE

使用 Truncate 命令清空表，表的临时分区会被删除，且不可恢复。
使用 Truncate 命令清空正式分区时，不影响临时分区。
不可使用 Truncate 命令清空临时分区。

ALTER

当表存在临时分区时，无法使用 Alter 命令对表进行 Schema Change、Rollup 等变更操作。
当表在进行变更操作时，无法对表添加临时分区。

最佳实践

1. 原子的覆盖写操作。
某些情况下，用户希望能够重写某一分区的数据，但如果采用先删除再导入的方式进行，在中间会有一段时间无法查看数据。这时，用户可以先创建一个对应的临时分区，将新的数据导入到临时分区后，通过替换操作，原子的替换原有分区，以达到目的。对于非分区表的原子覆盖写操作。
2. 修改分桶数。
某些情况下，用户在创建分区时使用了不合适的分桶数。则用户可以先创建一个对应分区范围的临时分区，并指定新的分桶数。然后通过 `INSERT INTO` 命令将正式分区的数据导入到临时分区中，通过替换操作，原子的替换原有分区，以达到目的。
3. 合并或分割分区。
某些情况下，用户希望对分区的范围进行修改，例如合并两个分区，或将一个大分区分割成多个小分区。则用户可以先建立对应合并或分割后范围的临时分区，然后通过 `INSERT INTO` 命令将正式分区的数据导入到临时分区中，通过替换操作，原子的替换原有分区，以达到目的。

自动分桶

最近更新时间：2024-06-27 11:09:45

说明

该功能适用于 Doris 1.2.2及后续版本。

用户经常设置不合适的 Bucket，导致各种问题，这里提供一种方式，来自动设置分桶数。

实现

以往创建分桶时需要手动设定分桶数，而自动分桶推算功能使 Doris 可以动态地推算分桶个数，使得分桶数始终保持在一个合适范围内，让用户不再操心桶数的细枝末节。 为了方便阐述该功能，该部分会将桶拆分为两个时期的桶，即初始分桶以及后续分桶；这里的初始和后续只是本文为了描述清楚该功能而采用的术语，Doris 分桶本身没有初始和后续之分。 BUCKET_DESC 非常简单，但是需要指定分桶个数。在自动分桶推算功能上，BUCKET_DESC 的语法直接将分桶数改成"Auto"，并新增一个 Properties 配置即可：

```
-- 旧版本指定分桶个数的创建语法
DISTRIBUTED BY HASH(site) BUCKETS 20

-- 新版本使用自动分桶推算的创建语法
DISTRIBUTED BY HASH(site) BUCKETS AUTO properties("estimate_partition_size" = "100G
```

新增的配置参数 estimate_partition_size 表示一个单分区的数据量。该参数是可选的，如果没有给出则 Doris 会将 estimate_partition_size 的默认值取为 10GB。 从上文中已经得知，一个分桶在物理层面就是一个Tablet，为了获得最好的性能，建议 Tablet 的大小在 1GB - 10GB 的范围内。那么自动分桶推算是如何保证 Tablet 大小处于这个范围内的呢？主要原则如下：

若是整体数据量较小，则分桶数不要设置过多。

若是整体数据量较大，则应使桶数跟总的磁盘块数相关，充分利用每台 BE 机器和每块磁盘的能力。

初始分桶推算

从原则出发，理解自动分桶推算功能的详细逻辑就变得简单了，首先来看初始分桶：

1. 先根据数据量得出一个桶数 N。首先使用 estimate_partition_size 的值除以 5（按文本格式存入 Doris 中有 5 比 1 的数据压缩比计算），得到的结果为：

(, 100MB)，则取 N=1。

[100MB, 1GB)，则取 N=2。

[1GB,)，则每GB一个分桶。

2. 根据 BE 节点数以及每个 BE 节点的磁盘容量，计算出桶数 M。其中每个 BE 节点算 1，每 50G 的磁盘容量算 1，那么 M 的计算规则为： M = BE 节点数 (一块磁盘块大小 / 50GB) 磁盘块数 例如有 3 台 BE，每台 BE 都有 4 块 500GB 的磁盘，那么 M = 3 (500GB / 50GB) 4 = 120。

3. 得到最终的分桶个数计算逻辑： 先计算一个中间值 x = min(M, N, 128)， 如果 x < N并且x < BE节点个数，则最终分桶为 y 即 BE 节点个数；否则最终分桶数为 x。

上述过程伪代码表现形式为：

```
int N = 计算N值;
int M = 计算M值;

int y = BE节点个数;
int x = min(M, N, 128);

if (x < N && x < y) {
    return y;
}
return x;
```

有了上述算法，再引入一些例子来更好地理解这部分逻辑：

```
case1:
数据量 100 MB, 10 台 BE 机器, 2TB *3 块盘
数据量 N = 1
BE 磁盘 M = 10* (2TB/50GB) * 3 = 1230
x = min(M, N, 128) =  1
最终: 1

case2:
数据量 1GB, 3 台 BE 机器, 500GB *2块盘
数据量 N = 2
BE 磁盘 M = 3* (500GB/50GB) * 2 = 60
x = min(M, N, 128) =  2
最终: 2

case3:
数据量100GB, 3台BE机器, 500GB *2块盘
数据量N = 20
BE磁盘M = 3* (500GB/50GB) * 2 = 60
x = min(M, N, 128) =  20
最终: 20

case4:
数据量500GB, 3台BE机器, 1TB *1块盘
数据量N = 100
BE磁盘M = 3* (1TB /50GB) * 1 = 60
x = min(M, N, 128) =  63
最终: 63

case5:
数据量500GB, 10台BE机器, 2TB *3块盘
```

```
数据量 N =  100
BE磁盘 M = 10* (2TB / 50GB) * 3 = 1230
x = min(M, N, 128) =  100
最终: 100

case 6:
数据量1TB, 10台BE机器, 2TB *3块盘
数据量 N =  205
BE磁盘M = 10* (2TB / 50GB) * 3 = 1230
x = min(M, N, 128) =  128
最终: 128

case 7:
数据量500GB, 1台BE机器, 100TB *1块盘
数据量 N = 100
BE磁盘M =  1* (100TB / 50GB) * 1 = 2048
x = min(M, N, 128) =  100
最终: 100

case 8:
数据量1TB, 200台BE机器, 4TB *7块盘
数据量 N = 205
BE磁盘M = 200* (4TB / 50GB) * 7 = 114800
x = min(M, N, 128) =  128
最终: 200
```

可以看到，详细逻辑与原则是匹配的。

后续分桶推算

上述是关于初始分桶的计算逻辑，后续分桶数因为已经有了一定的分区数据，可以根据已有的分区数据量来进行评估。后续分桶数会根据最多前 7 个分区数据量的 EMA（短期指数移动平均线）值，作为 estimate_partition_size 进行评估。此时计算分桶有两种计算方式，假设以天来分区，往前数第一天分区大小为 S7，往前数第二天分区大小为 S6，依次类推到 S1。

1. 如果 7 天内的分区数据每日严格递增，则此时会取趋势值。有6个 delta 值，分别是：

```
S7 - S6 = delta1,
S6 - S5 = delta2,
...
S2 - S1 = delta6
```

由此得到 ema(delta) 值： 那么，今天的 estimate_partition_size = S7 + ema(delta)

2. 非第一种的情况，此时直接取前几天的 EMA 平均值：

```
今天的estimate_partition_size = EMA(S1, ..., S7)
```

根据上述算法，初始分桶个数以及后续分桶个数都能被计算出来。跟之前只能指定固定分桶数不同，由于业务数据的变化，可能前面分区的分桶数和后面分区的分桶数不一样，这对用户是透明的，用户无需关心每一分区具体的分桶数是多少，而这一自动推算的功能会让分桶数更加合理。

最佳实践

创建使用自动分桶表

```
CREATE TABLE IF NOT EXISTS example_auto_bucket_tbl
(
    user_id VARCHAR(128) NOT NULL COMMENT "用户id",
    date DATE NOT NULL COMMENT "数据灌入日期时间",
    data varchar(512) NOT NULL
)
ENGINE=OLAP
DUPLICATE KEY(user_id)
PARTITION BY RANGE(date)()
DISTRIBUTED BY HASH(user_id) BUCKETS AUTO
PROPERTIES
(
    "replication_num" = "1",
    # 开启动态分区
    "dynamic_partition.enable" = "true",
    # 按月进行动态分区
    "dynamic_partition.time_unit" = "MONTH",
    # 保留5个历史分区
    "dynamic_partition.start" = "-5",
    # 提前创建10个未来分区
    "dynamic_partition.end" = "3",
    # 分区命名前缀
    "dynamic_partition.prefix" = "p_",
    # 开启创建历史分区
    "dynamic_partition.create_history_partition" = "true",
    # 创建3个历史分区
    "dynamic_partition.history_partition_num" = "3",
    # 预计分区的数据量
    "estimate_partition_size" = "1G"
);
```

可以看到，在创建表后，Doris系统自动为每个分区设置的分桶数量为2。

```
| example_auto_bucket_tbl | CREATE TABLE `example_auto_bucket_tbl` (
  `user_id` varchar(128) NOT NULL COMMENT '用户id',
  `date` date NOT NULL COMMENT '数据灌入日期时间',
  `data` varchar(512) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(`user_id`)
COMMENT 'OLAP'
PARTITION BY RANGE(`date`)
(PARTITION p_202304 VALUES [('2023-04-01'), ('2023-05-01')),
PARTITION p_202305 VALUES [('2023-05-01'), ('2023-06-01')),
PARTITION p_202306 VALUES [('2023-06-01'), ('2023-07-01')),
PARTITION p_202307 VALUES [('2023-07-01'), ('2023-08-01')),
PARTITION p_202308 VALUES [('2023-08-01'), ('2023-09-01')),
PARTITION p_202309 VALUES [('2023-09-01'), ('2023-10-01')),
PARTITION p_202310 VALUES [('2023-10-01'), ('2023-11-01')))
DISTRIBUTED BY HASH(`user_id`) BUCKETS AUTO
PROPERTIES (
"replication_allocation" = "tag.location.default: 1",
"dynamic_partition.enable" = "true",
"dynamic_partition.time_unit" = "MONTH",
"dynamic_partition.time_zone" = "Asia/Shanghai",
"dynamic_partition.start" = "-5",
"dynamic_partition.end" = "3",
"dynamic_partition.prefix" = "p_",
"dynamic_partition.replication_allocation" = "tag.location.default: 1",
"dynamic_partition.buckets" = "2",
"dynamic_partition.create_history_partition" = "true",
"dynamic_partition.history_partition_num" = "3",
"dynamic_partition.hot_partition_num" = "0",
"dynamic_partition.reserved_history_periods" = "NULL",
"dynamic_partition.storage_policy" = "",
"dynamic_partition.storage_medium" = "HDD",
"dynamic_partition.start_day_of_month" = "1",
"in_memory" = "false",
"storage_format" = "V2",
"estimate_partition_size" = "1G",
"disable_auto_compaction" = "false"
); |
```

```
MySQL [example]> show partitions from example_auto_bucket_tbl;
| PartitionId | PartitionName | VisibleVersion | VisibleVersionTime  | State  | PartitionKey | Range                                                                              | DistributionKey
 RemoteStoragePolicy | LastConsistencyCheckTime | DataSize | IsInMemory | ReplicaAllocation       |
| 29092       | p_202304      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-04-01]; ..types: [DATE]; keys: [2023-05-01]; ) | user_id
                     | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29097       | p_202305      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-05-01]; ..types: [DATE]; keys: [2023-06-01]; ) | user_id
                     | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29102       | p_202306      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-06-01]; ..types: [DATE]; keys: [2023-07-01]; ) | user_id
                     | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29107       | p_202307      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-07-01]; ..types: [DATE]; keys: [2023-08-01]; ) | user_id
                     | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29112       | p_202308      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-08-01]; ..types: [DATE]; keys: [2023-09-01]; ) | user_id
                     | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29117       | p_202309      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-09-01]; ..types: [DATE]; keys: [2023-10-01]; ) | user_id
                     | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29122       | p_202310      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-10-01]; ..types: [DATE]; keys: [2023-11-01]; ) | user_id
                     | NULL                     | 0.000    | false      | tag.location.default: 1 |
7 rows in set (0.00 sec)
```

接下来我们向表内导入一定数量的数据，之后调整 dynamic_partition.end 参数，让 Doris 自动创建新的分区。查看新分区的分桶情况，可以看到新创建的分区的分桶数量变为7。

```
MySQL [example]> show partitions from example_auto_bucket_tbl;
| PartitionId | PartitionName | VisibleVersion | VisibleVersionTime  | State  | PartitionKey | Range                                                                                  | DistributionKey
 RemoteStoragePolicy | LastConsistencyCheckTime | DataSize | IsInMemory | ReplicaAllocation        |
| 29092       | p_202304      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-04-01]; ..types: [DATE]; keys: [2023-05-01]; ) | user_id
                    | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29097       | p_202305      | 1              | 2023-07-14 17:21:53 | NORMAL | date         | [types: [DATE]; keys: [2023-05-01]; ..types: [DATE]; keys: [2023-06-01]; ) | user_id
                    | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29102       | p_202306      | 4              | 2023-07-14 17:34:34 | NORMAL | date         | [types: [DATE]; keys: [2023-06-01]; ..types: [DATE]; keys: [2023-07-01]; ) | user_id
                    | NULL                     | 3.637 GB | false      | tag.location.default: 1 |
| 29107       | p_202307      | 4              | 2023-07-14 17:34:34 | NORMAL | date         | [types: [DATE]; keys: [2023-07-01]; ..types: [DATE]; keys: [2023-08-01]; ) | user_id
                    | NULL                     | 7.274 GB | false      | tag.location.default: 1 |
| 29112       | p_202308      | 4              | 2023-07-14 17:34:34 | NORMAL | date         | [types: [DATE]; keys: [2023-08-01]; ..types: [DATE]; keys: [2023-09-01]; ) | user_id
                    | NULL                     | 7.274 GB | false      | tag.location.default: 1 |
| 29117       | p_202309      | 4              | 2023-07-14 17:34:34 | NORMAL | date         | [types: [DATE]; keys: [2023-09-01]; ..types: [DATE]; keys: [2023-10-01]; ) | user_id
                    | NULL                     | 7.274 GB | false      | tag.location.default: 1 |
| 29122       | p_202310      | 4              | 2023-07-14 17:34:34 | NORMAL | date         | [types: [DATE]; keys: [2023-10-01]; ..types: [DATE]; keys: [2023-11-01]; ) | user_id
                    | NULL                     | 4.635 GB | false      | tag.location.default: 1 |
| 29143       | p_202311      | 1              | 2023-07-14 17:34:52 | NORMAL | date         | [types: [DATE]; keys: [2023-11-01]; ..types: [DATE]; keys: [2023-12-01]; ) | user_id
                    | NULL                     | 0.000    | false      | tag.location.default: 1 |
| 29158       | p_202312      | 1              | 2023-07-14 17:34:52 | NORMAL | date         | [types: [DATE]; keys: [2023-12-01]; ..types: [DATE]; keys: [2024-01-01]; ) | user_id
                    | NULL                     | 0.000    | false      | tag.location.default: 1 |
9 rows in set (0.00 sec)
```

关闭自动分桶功能

临时关闭：关闭动态分区

```
ALTER TABLE example_auto_bucket_tbl SET
(
  # 关闭动态分区
    "dynamic_partition.enable" = "false"
);
```

永久关闭：关闭动态分区后，修改表的分桶数量关闭

```
# 关闭动态分区后，修改表的分桶数量
alter table example_auto_bucket_tbl modify DISTRIBUTION DISTRIBUTED BY HASH(`user_i
```

需要注意两种操作的效果：

临时关闭后，如果重新打开动态分区功能，动态分区新创建的分区分桶数量为 Doris 根据最近几个分区的数据量计算出的结果。

关闭动态分区后，若修改表的分桶数量，后续将无法再开启自动分桶功能。

说明

只对动态分区创建的分区生效。

开启 autobucket 之后，在 show create table 的时候看到的 schema 也是 BUCKETS AUTO。如果想要查看确切的 bucket 数，可以通过 show partitions from ${table}; 来查看。

不可以在已有表上开启自动分桶功能，自动分桶功能只能在创建表时开启。

自动分桶不会对已有分区产生影响，自动分桶只是在动态分区创建新分区时，根据历史分区数据，磁盘，doris 的 BE 节点数量等信息，通过一定的算法得出新分区的分桶数量，不会对已有的分区造成影响。

查询优化
Rollup 索引

最近更新时间：2024-06-27 11:10:05

ROLLUP 在多维分析中是“上卷”的意思，即将数据按某种指定的粒度进行进一步聚合。

基本概念

在 Doris 中，我们将用户通过建表语句创建出来的表称为 Base 表（Base Table）。Base 表中保存着按用户建表语句指定的方式存储的基础数据。

在 Base 表之上，我们可以创建任意多个 ROLLUP 表。这些 ROLLUP 的数据是基于 Base 表产生的，并且在物理上是**独立存储**的。

ROLLUP 表的基本作用，在于在 Base 表的基础上，获得更粗粒度的聚合数据。

下面我们用示例详细说明在不同数据模型中的 ROLLUP 表及其作用。

Aggregate 和 Unique 模型中的 ROLLUP

因为 Unique 只是 Aggregate 模型的一个特例，所以这里我们不加以区别。

示例1：获得每个用户的总消费

接 数据表和数据模型 的**示例2**，Base 表结构如下：

ColumnName	Type	AggregationType	Comment
user_id	LARGEINT	-	用户 ID
date	DATE	-	数据导入日期
timestamp	DATETIME	-	数据导入时间，精确到秒
city	VARCHAR(20)	-	用户所在城市
age	SMALLINT	-	用户年龄
sex	TINYINT	-	用户性别
last_visit_date	DATETIME	REPLACE	用户最后一次访问时间
cost	BIGINT	SUM	用户总消费
max_dwell_time	INT	MAX	用户最大停留时间

min_dwell_time	INT	MIN	用户最小停留时间

存储的数据如下：

user_id	date	timestamp	city	age	sex	last_visit_date	cost	max_dwell_time	min_
10000	2017-10-01	2017-10-01 08:00:05	北京	20	0	2017-10-01 06:00:00	20	10	10
10000	2017-10-01	2017-10-01 09:00:05	北京	20	0	2017-10-01 07:00:00	15	2	2
10001	2017-10-01	2017-10-01 18:12:10	北京	30	1	2017-10-01 17:05:45	2	22	22
10002	2017-10-02	2017-10-02 13:10:00	上海	20	1	2017-10-02 12:59:12	200	5	5
10003	2017-10-02	2017-10-02 13:15:00	广州	32	0	2017-10-02 11:20:00	30	11	11
10004	2017-10-01	2017-10-01 12:12:48	深圳	35	0	2017-10-01 10:00:15	100	3	3
10004	2017-10-03	2017-10-03 12:38:20	深圳	35	0	2017-10-03 10:20:22	11	6	6

在此基础上，我们创建一个 ROLLUP，该 ROLLUP 只包含两列：user_id 和 cost。则创建完成后，该 ROLLUP 中存储的数据如下：

user_id	cost
10000	35
10001	2
10002	200
10003	30

10004	111

可以看到，ROLLUP 中仅保留了每个 user_id，在 cost 列上的 SUM 的结果。那么当我们进行如下查询时：

```
SELECT user_id, sum(cost) FROM table GROUP BY user_id;
```

Doris 会自动命中这个 ROLLUP 表，从而只需扫描极少的数据量，即可完成这次聚合查询。

示例2：获得不同城市，不同年龄段用户的总消费、最长和最短页面驻留时间

在 Base 表基础之上，再创建一个 ROLLUP：

ColumnName	Type	AggregationType	Comment
city	VARCHAR(20)	-	用户所在城市
age	SMALLINT	-	用户年龄
cost	BIGINT	SUM	用户总消费
max_dwell_time	INT	MAX	用户最大停留时间
min_dwell_time	INT	MIN	用户最小停留时间

则创建完成后，该 ROLLUP 中存储的数据如下：

city	age	cost	max_dwell_time	min_dwell_time
北京	20	35	10	2
北京	30	2	22	22
上海	20	200	5	5
广州	32	30	11	11
深圳	35	111	6	3

当我们进行如下这些查询时：

```
mysql> SELECT city, age, sum(cost), max(max_dwell_time), min(min_dwell_time) FROM t
mysql> SELECT city, sum(cost), max(max_dwell_time), min(min_dwell_time) FROM table
mysql> SELECT city, age, sum(cost), min(min_dwell_time) FROM table GROUP BY city, a
```

Doris 执行这些 sql 时会自动命中这个 ROLLUP 表。

Duplicate 模型中的 ROLLUP

因为 Duplicate 模型没有聚合的语意，无法对 Duplicate 表 创建包含聚合函数的 rollup。所以该模型中的 ROLLUP，已经失去了“上卷”这一层含义。而仅作为调整列顺序，以命中前缀索引的作用。我们将在 索引、排序列和前缀索引 详细介绍前缀索引，以及如何使用 ROLLUP 改变前缀索引以获得更好的查询效率。

ROLLUP 调整前缀索引

因为建表时已经指定了列顺序，所以一个表只有一种前缀索引。这对于使用其他不能命中前缀索引的列作为条件进行的查询来说，效率上可能无法满足需求。因此，我们可以通过创建 ROLLUP 来人为的调整列顺序。举例说明：
Base 表结构如下：

ColumnName	Type
user_id	BIGINT
age	INT
message	VARCHAR(100)
max_dwell_time	DATETIME
min_dwell_time	DATETIME

我们可以在此基础上创建一个 ROLLUP 表：

ColumnName	Type
age	INT
user_id	BIGINT
message	VARCHAR(100)
max_dwell_time	DATETIME
min_dwell_time	DATETIME

可以看到，ROLLUP 和 Base 表的列完全一样，只是将 user_id 和 age 的顺序调换了。那么当我们进行如下查询时：

```
mysql> SELECT * FROM table where age=20 and message LIKE "%error%";
```

会优先选择 ROLLUP 表，因为 ROLLUP 的前缀索引匹配度更高。

ROLLUP 使用说明

ROLLUP 最根本的作用是提高某些查询的查询效率（无论是通过聚合来减少数据量，还是修改列顺序以匹配前缀索引）。因此 ROLLUP 的含义已经超出了 “上卷” 的范围。这也是我们在源代码中，将其命名为 Materialized Index（物化索引）的原因。

ROLLUP 是附属于 Base 表的，可以看做是 Base 表的一种辅助数据结构。用户可以在 Base 表的基础上，创建或删除 ROLLUP，但是不能在查询中显式的指定查询某 ROLLUP。是否命中 ROLLUP 完全由 Doris 系统自动决定。

ROLLUP 的数据是独立物理存储的。因此，创建的 ROLLUP 越多，占用的磁盘空间也就越大。同时对导入速度也会有影响（导入的ETL阶段会自动产生所有 ROLLUP 的数据），但是不会降低查询效率（只会更好）。

ROLLUP 的数据更新与 Base 表是完全同步的。用户无需关心这个问题。

ROLLUP 中列的聚合方式，与 Base 表完全相同。在创建 ROLLUP 无需指定，也不能修改。

查询能否命中 ROLLUP 的一个必要条件（非充分条件）是，查询所涉及的**所有列**（包括 select list 和 where 中的查询条件列等）都存在于该 ROLLUP 的列中。否则，查询只能命中 Base 表。

某些类型的查询（如 count(*)）在任何条件下，都无法命中 ROLLUP。具体参见接下来的 **聚合模型的局限性** 一节。

可以通过 `EXPLAIN your_sql;` 命令获得查询执行计划，在执行计划中，查看是否命中 ROLLUP。

可以通过 `DESC tbl_name ALL;` 语句显示 Base 表和所有已创建完成的 ROLLUP。

ROLLUP 使用限制

在 Aggregate 表和 Unique 表上，创建 Rollup 时，无法指定聚合函数，会自动根据 Base 表对应字段的聚合函数进行聚合。

Rollup 只能对单表操作，不支持跨表。

对 Unique 表进行 Rollup 时，Rollup 必须包含所有 Unique key,否则会创建失败。可以对 Base 表的 Unique key 进行顺序调整，形成不同的前缀索引达到查询加速的目的。

创建物化视图时，不支持对同一个 key 进行多维度聚合，即一个 key 不能出现多次。

```
MySQL [tpch_my]> desc lineitem;
+-----------------+-----------------+------+-------+---------+---------+
| Field           | Type            | Null | Key   | Default | Extra   |
+-----------------+-----------------+------+-------+---------+---------+
| l_shipdate      | DATE            | No   | true  | NULL    |         |
| l_orderkey      | BIGINT          | No   | true  | NULL    |         |
| l_linenumber    | INT             | No   | true  | NULL    |         |
| l_partkey       | INT             | No   | true  | NULL    |         |
| l_suppkey       | INT             | No   | true  | NULL    |         |
| l_quantity      | DECIMALV3(15,2) | No   | false | NULL    | REPLACE |
| l_extendedprice | DECIMALV3(15,2) | No   | false | NULL    | REPLACE |
| l_discount      | DECIMALV3(15,2) | No   | false | NULL    | REPLACE |
| l_tax           | DECIMALV3(15,2) | No   | false | NULL    | REPLACE |
| l_returnflag    | VARCHAR(1)      | No   | false | NULL    | REPLACE |
| l_linestatus    | VARCHAR(1)      | No   | false | NULL    | REPLACE |
| l_commitdate    | DATE            | No   | false | NULL    | REPLACE |
| l_receiptdate   | DATE            | No   | false | NULL    | REPLACE |
| l_shipinstruct  | VARCHAR(25)     | No   | false | NULL    | REPLACE |
| l_shipmode      | VARCHAR(10)     | No   | false | NULL    | REPLACE |
| l_comment       | VARCHAR(44)     | No   | false | NULL    | REPLACE |
+-----------------+-----------------+------+-------+---------+---------+
16 rows in set (0.00 sec)

MySQL [tpch_my]> create materialized view supkey_count as select l_suppkey,count(l_suppkey) as count from lineitem g
ERROR 1060 (42S21): errCode = 2, detailMessage = Duplicate column name 'l_suppkey'
MySQL [tpch_my]> create materialized view supkey_sum as select l_suppkey,sum(l_suppkey) as count from lineitem group
ERROR 1060 (42S21): errCode = 2, detailMessage = Duplicate column name 'l_suppkey'
MySQL [tpch_my]> create materialized view supkey_sum_max as select l_suppkey,sum(l_discount),max(l_discount) as coun
ERROR 1060 (42S21): errCode = 2, detailMessage = Duplicate column name 'l_discount'
MySQL [tpch_my]>
```

查询

在 Doris 里 Rollup 作为一份聚合物化视图，其在查询中可以起到两个作用：

索引

聚合数据（仅用于聚合模型，即 aggregate key）

但是为了命中 Rollup 需要满足一定的条件，并且可以通过执行计划中 ScanNode 节点的 PreAggregation 的值来判断是否可以命中 Rollup，以及 Rollup 字段来判断命中的是哪一张 Rollup 表。

索引

前面的查询实践中已经介绍过 Doris 的前缀索引，即 Doris 会把 Base/Rollup 表中的前 36 个字节（有 varchar 类型则可能导致前缀索引不满 36 个字节，varchar 会截断前缀索引，并且最多使用 varchar 的 20 个字节）在底层存储引擎单独生成一份排序的稀疏索引数据(数据也是排序的，用索引定位，然后在数据中做二分查找)，然后在查询的时候会根据查询中的条件来匹配每个 Base/Rollup 的前缀索引，并且选择出匹配前缀索引最长的一个 Base/Rollup。

```
        -----> 从左到右匹配
+----+----+----+----+----+----+
| c1 | c2 | c3 | c4 | c5 |... |
```

取查询中 where 以及 on 上下推到 ScanNode 的条件，从前缀索引的第一列开始匹配，检查条件中是否有这些列，有则累计匹配的长度，直到匹配不上或者36字节结束（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），然后选择出匹配长度最长的一个 Base/Rollup。

下面举例说明，创建了一张 Base 表以及四张 rollup：

```
+---------------+-------+--------------+------+-------+---------+-------+
| IndexName     | Field | Type         | Null | Key   | Default | Extra |
+---------------+-------+--------------+------+-------+---------+-------+
| test          | k1    | TINYINT      | Yes  | true  | N/A     |       |
|               | k2    | SMALLINT     | Yes  | true  | N/A     |       |
|               | k3    | INT          | Yes  | true  | N/A     |       |
|               | k4    | BIGINT       | Yes  | true  | N/A     |       |
|               | k5    | DECIMAL(9,3) | Yes  | true  | N/A     |       |
|               | k6    | CHAR(5)      | Yes  | true  | N/A     |       |
|               | k7    | DATE         | Yes  | true  | N/A     |       |
|               | k8    | DATETIME     | Yes  | true  | N/A     |       |
|               | k9    | VARCHAR(20)  | Yes  | true  | N/A     |       |
|               | k10   | DOUBLE       | Yes  | false | N/A     | MAX   |
|               | k11   | FLOAT        | Yes  | false | N/A     | SUM   |
|               |       |              |      |       |         |       |
| rollup_index1 | k9    | VARCHAR(20)  | Yes  | true  | N/A     |       |
|               | k1    | TINYINT      | Yes  | true  | N/A     |       |
|               | k2    | SMALLINT     | Yes  | true  | N/A     |       |
|               | k3    | INT          | Yes  | true  | N/A     |       |
|               | k4    | BIGINT       | Yes  | true  | N/A     |       |
|               | k5    | DECIMAL(9,3) | Yes  | true  | N/A     |       |
|               | k6    | CHAR(5)      | Yes  | true  | N/A     |       |
|               | k7    | DATE         | Yes  | true  | N/A     |       |
|               | k8    | DATETIME     | Yes  | true  | N/A     |       |
|               | k10   | DOUBLE       | Yes  | false | N/A     | MAX   |
|               | k11   | FLOAT        | Yes  | false | N/A     | SUM   |
|               |       |              |      |       |         |       |
| rollup_index2 | k9    | VARCHAR(20)  | Yes  | true  | N/A     |       |
|               | k2    | SMALLINT     | Yes  | true  | N/A     |       |
|               | k1    | TINYINT      | Yes  | true  | N/A     |       |
|               | k3    | INT          | Yes  | true  | N/A     |       |
|               | k4    | BIGINT       | Yes  | true  | N/A     |       |
|               | k5    | DECIMAL(9,3) | Yes  | true  | N/A     |       |
|               | k6    | CHAR(5)      | Yes  | true  | N/A     |       |
|               | k7    | DATE         | Yes  | true  | N/A     |       |
|               | k8    | DATETIME     | Yes  | true  | N/A     |       |
|               | k10   | DOUBLE       | Yes  | false | N/A     | MAX   |
|               | k11   | FLOAT        | Yes  | false | N/A     | SUM   |
|               |       |              |      |       |         |       |
| rollup_index3 | k4    | BIGINT       | Yes  | true  | N/A     |       |
|               | k5    | DECIMAL(9,3) | Yes  | true  | N/A     |       |
|               | k6    | CHAR(5)      | Yes  | true  | N/A     |       |
|               | k1    | TINYINT      | Yes  | true  | N/A     |       |
|               | k2    | SMALLINT     | Yes  | true  | N/A     |       |
```

```
|               | k3    | INT           | Yes  | true  | N/A     |       |
|               | k7    | DATE          | Yes  | true  | N/A     |       |
|               | k8    | DATETIME      | Yes  | true  | N/A     |       |
|               | k9    | VARCHAR(20)   | Yes  | true  | N/A     |       |
|               | k10   | DOUBLE        | Yes  | false | N/A     | MAX   |
|               | k11   | FLOAT         | Yes  | false | N/A     | SUM   |
|               |       |               |      |       |         |       |
| rollup_index4 | k4    | BIGINT        | Yes  | true  | N/A     |       |
|               | k6    | CHAR(5)       | Yes  | true  | N/A     |       |
|               | k5    | DECIMAL(9,3)  | Yes  | true  | N/A     |       |
|               | k1    | TINYINT       | Yes  | true  | N/A     |       |
|               | k2    | SMALLINT      | Yes  | true  | N/A     |       |
|               | k3    | INT           | Yes  | true  | N/A     |       |
|               | k7    | DATE          | Yes  | true  | N/A     |       |
|               | k8    | DATETIME      | Yes  | true  | N/A     |       |
|               | k9    | VARCHAR(20)   | Yes  | true  | N/A     |       |
|               | k10   | DOUBLE        | Yes  | false | N/A     | MAX   |
|               | k11   | FLOAT         | Yes  | false | N/A     | SUM   |
+---------------+-------+---------------+------+-------+---------+-------+
```

这五张表的前缀索引分别为：

```
Base(k1 ,k2, k3, k4, k5, k6, k7)

rollup_index1(k9)

rollup_index2(k9)

rollup_index3(k4, k5, k6, k1, k2, k3, k7)

rollup_index4(k4, k6, k5, k1, k2, k3, k7)
```

能使用前缀索引的列上的条件需要是 `=<><=>=inbetween` ，并列的且关系使用 `and` 连接，对于 `or` 、 `!=` 等这些不能命中，然后看以下查询：

`SELECT * FROM test WHERE k1 = 1 AND k2 > 3;` 。

有 k1 以及 k2 上的条件，检查只有 Base 的第一列含有条件里的 k1，所以匹配最长的前缀索引即 test：

```
|   0:OlapScanNode
|      TABLE: test
|      PREAGGREGATION: OFF. Reason: No AggregateInfo
|      PREDICATES: `k1` = 1, `k2` > 3
|      partitions=1/1
|      rollup: test
|      buckets=1/10
|      cardinality=-1
|      avgRowSize=0.0
```

```
|      numNodes=0
|      tuple ids: 0
```

再看以下查询：

```
SELECT * FROM test WHERE k4 = 1 AND k5 > 3;
```

有 k4 以及 k5 的条件，检查 rollup_index3、rollup_index4 的第一列含有 k4，但是 rollup_index3 的第二列含有 k5，所以匹配的前缀索引最长。

```
|   0:OlapScanNode
|      TABLE: test
|      PREAGGREGATION: OFF. Reason: No AggregateInfo
|      PREDICATES: `k4` = 1, `k5` > 3
|      partitions=1/1
|      rollup: rollup_index3
|      buckets=10/10
|      cardinality=-1
|      avgRowSize=0.0
|      numNodes=0
|      tuple ids: 0
```

现在我们尝试匹配含有 varchar 列上的条件，如下：`SELECT * FROM test WHERE k9 IN ("xxx", "yyyy") AND k1 = 10;` 。

有 k9 以及 k1 两个条件，rollup_index1 以及 rollup_index2 的第一列都含有 k9，按理说这里选择这两个 rollup 都可以命中前缀索引并且效果是一样的随机选择一个即可（因为这里 varchar 刚好20个字节，前缀索引不足36个字节被截断），但是当前策略这里还会继续匹配 k1，因为 rollup_index1 的第二列为 k1，所以选择了 rollup_index1，其实后面的 k1 条件并不会起到加速的作用。(如果对于前缀索引外的条件需要其可以起到加速查询的目的，可以通过建立 Bloom Filter 过滤器加速。一般对于字符串类型建立即可，因为 Doris 针对列存在 Block 级别对于整形、日期已经有 Min/Max 索引) 以下是 explain 的结果。

```
|   0:OlapScanNode
|      TABLE: test
|      PREAGGREGATION: OFF. Reason: No AggregateInfo
|      PREDICATES: `k9` IN ('xxx', 'yyyy'), `k1` = 10
|      partitions=1/1
|      rollup: rollup_index1
|      buckets=1/10
|      cardinality=-1
|      avgRowSize=0.0
|      numNodes=0
|      tuple ids: 0
```

最后看一个多张Rollup都可以命中的查询：`SELECT * FROM test WHERE k4 < 1000 AND k5 = 80 AND k6 >= 10000;` 。

有 k4,k5,k6 三个条件，rollup_index3 以及 rollup_index4 的前3列分别含有这三列，所以两者匹配的前缀索引长度一致，选取两者都可以，当前默认的策略为选取了比较早创建的一张 rollup，这里为 rollup_index3。

```
|   0:OlapScanNode
|      TABLE: test
|      PREAGGREGATION: OFF. Reason: No AggregateInfo
|      PREDICATES: `k4` < 1000, `k5` = 80, `k6` >= 10000.0
|      partitions=1/1
|      rollup: rollup_index3
|      buckets=10/10
|      cardinality=-1
|      avgRowSize=0.0
|      numNodes=0
|      tuple ids: 0
```

如果稍微修改上面的查询为：`SELECT * FROM test WHERE k4 < 1000 AND k5 = 80 OR k6 >= 10000;` 。

则这里的查询不能命中前缀索引。（甚至 Doris 存储引擎内的任何 Min/Max，BloomFilter 索引都不能起作用）。

聚合数据

当然一般的聚合物化视图其聚合数据的功能是必不可少的，这类物化视图对于聚合类查询或报表类查询都有非常大的帮助，要命中聚合物化视图需要下面一些前提：

1. 查询或者子查询中涉及的所有列都存在一张独立的 Rollup 中。
2. 如果查询或者子查询中有 Join，则 Join 的类型需要是 Inner join。

以下是可以命中 Rollup 的一些聚合查询的种类。

列类型 查询类型	Sum	Distinct/Count Distinct	Min	Max	APPROX_COUNT_DISTINCT
Key	false	true	true	true	true
Value(Sum)	true	false	false	false	false
Value(Replace)	false	false	false	false	false
Value(Min)	false	false	true	false	false
Value(Max)	false	false	false	true	false

如果符合上述条件，则针对聚合模型在判断命中 Rollup 的时候会有两个阶段：

1. 首先通过条件匹配出命中前缀索引最长的 Rollup 表，参见上述**索引策略**。
2. 然后比较 Rollup 的行数，选择最小的一张 Rollup。

如下 Base 表以及 Rollup：

```
+-------------+-------+--------------+------+-------+---------+-------+
| IndexName   | Field | Type         | Null | Key   | Default | Extra |
+-------------+-------+--------------+------+-------+---------+-------+
| test_rollup | k1    | TINYINT      | Yes  | true  | N/A     |       |
|             | k2    | SMALLINT     | Yes  | true  | N/A     |       |
|             | k3    | INT          | Yes  | true  | N/A     |       |
|             | k4    | BIGINT       | Yes  | true  | N/A     |       |
|             | k5    | DECIMAL(9,3) | Yes  | true  | N/A     |       |
|             | k6    | CHAR(5)      | Yes  | true  | N/A     |       |
|             | k7    | DATE         | Yes  | true  | N/A     |       |
|             | k8    | DATETIME     | Yes  | true  | N/A     |       |
|             | k9    | VARCHAR(20)  | Yes  | true  | N/A     |       |
|             | k10   | DOUBLE       | Yes  | false | N/A     | MAX   |
|             | k11   | FLOAT        | Yes  | false | N/A     | SUM   |
|             |       |              |      |       |         |       |
| rollup2     | k1    | TINYINT      | Yes  | true  | N/A     |       |
|             | k2    | SMALLINT     | Yes  | true  | N/A     |       |
|             | k3    | INT          | Yes  | true  | N/A     |       |
|             | k10   | DOUBLE       | Yes  | false | N/A     | MAX   |
|             | k11   | FLOAT        | Yes  | false | N/A     | SUM   |
|             |       |              |      |       |         |       |
| rollup1     | k1    | TINYINT      | Yes  | true  | N/A     |       |
|             | k2    | SMALLINT     | Yes  | true  | N/A     |       |
|             | k3    | INT          | Yes  | true  | N/A     |       |
|             | k4    | BIGINT       | Yes  | true  | N/A     |       |
|             | k5    | DECIMAL(9,3) | Yes  | true  | N/A     |       |
|             | k10   | DOUBLE       | Yes  | false | N/A     | MAX   |
|             | k11   | FLOAT        | Yes  | false | N/A     | SUM   |
+-------------+-------+--------------+------+-------+---------+-------+
```

看以下查询：`SELECT SUM(k11) FROM test_rollup WHERE k1 = 10 AND k2 > 200 AND k3 in (1,2,3);` 。

首先判断查询是否可以命中聚合的 Rollup表，经过查上面的图是可以的，然后条件中含有 k1，k2，k3 三个条件，这三个条件 test_rollup、rollup1、rollup2 的前三列都含有，所以前缀索引长度一致，然后比较行数显然 rollup2 的聚合程度最高行数最少所以选取 rollup2。

```
|   0:OlapScanNode                                      |
|      TABLE: test_rollup                               |
|      PREAGGREGATION: ON                               |
|      PREDICATES: `k1` = 10, `k2` > 200, `k3` IN (1, 2, 3) |
|      partitions=1/1                                   |
|      rollup: rollup2                                  |
|      buckets=1/10                                     |
|      cardinality=-1                                   |
|      avgRowSize=0.0                                   |
```

```
|      numNodes=0                                                    |
|      tuple ids: 0                                                  |
```

ROLLUP 最佳实践

在使用 rollup 时，预期进行查询加速，但是在实际使用过程中，如果使用不正确容易出现优化不生效，或者给系统带来负面影响的情况。

建议在使用 rollup 时，遵循以下步骤：

1. 根据业务场景规划 rollup 结构。
2. 创建 rollup。
3. 确认 rollup 构建完成。
4. 确认业务查询语句命中 rollup。

这里基于 TPC-H 数据测试积的 lineitem 表作为 case 表，展示 rollup 的使用步骤。以下是 lineitem 这张表的基础信息，导入了62G数据，数据总量为6000W行。

```
MySQL [tpch_100g]> desc lineitem all;
+-----------+---------------+-----------------+---------------+---------------+------+-------+---------+-------+-------+
| IndexName | IndexKeysType | Field           | Type          | InternalType  | Null | Key   | Default | Extra | Visib |
+-----------+---------------+-----------------+---------------+---------------+------+-------+---------+-------+-------+
| lineitem  | DUP_KEYS      | l_shipdate      | DATE          | DATEV2        | No   | true  | NULL    |       | true  |
|           |               | l_orderkey      | BIGINT        | BIGINT        | No   | true  | NULL    |       | true  |
|           |               | l_linenumber    | INT           | INT           | No   | false | NULL    | NONE  | true  |
|           |               | l_partkey       | INT           | INT           | No   | false | NULL    | NONE  | true  |
|           |               | l_suppkey       | INT           | INT           | No   | false | NULL    | NONE  | true  |
|           |               | l_quantity      | DECIMAL(15,2) | DECIMAL(15,2) | No   | false | NULL    | NONE  | true  |
|           |               | l_extendedprice | DECIMAL(15,2) | DECIMAL(15,2) | No   | false | NULL    | NONE  | true  |
|           |               | l_discount      | DECIMAL(15,2) | DECIMAL(15,2) | No   | false | NULL    | NONE  | true  |
|           |               | l_tax           | DECIMAL(15,2) | DECIMAL(15,2) | No   | false | NULL    | NONE  | true  |
|           |               | l_returnflag    | VARCHAR(1)    | VARCHAR(1)    | No   | false | NULL    | NONE  | true  |
|           |               | l_linestatus    | VARCHAR(1)    | VARCHAR(1)    | No   | false | NULL    | NONE  | true  |
|           |               | l_commitdate    | DATE          | DATEV2        | No   | false | NULL    | NONE  | true  |
|           |               | l_receiptdate   | DATE          | DATEV2        | No   | false | NULL    | NONE  | true  |
|           |               | l_shipinstruct  | VARCHAR(25)   | VARCHAR(25)   | No   | false | NULL    | NONE  | true  |
|           |               | l_shipmode      | VARCHAR(10)   | VARCHAR(10)   | No   | false | NULL    | NONE  | true  |
|           |               | l_comment       | VARCHAR(44)   | VARCHAR(44)   | No   | false | NULL    | NONE  | true  |
+-----------+---------------+-----------------+---------------+---------------+------+-------+---------+-------+-------+
16 rows in set (0.00 sec)

MySQL [tpch_100g]> select count(*) from lineitem;
+------------+
| count(*)   |
+------------+
| 600037902 |
+------------+
1 row in set (0.03 sec)

MySQL [tpch_100g]> show data;
+-----------+-------------+--------------+
| TableName | Size        | ReplicaCount |
+-----------+-------------+--------------+
| customer  | 3.951 GB    | 72           |
| lineitem  | 62.679 GB   | 288          |
| nation    | 7.714 KB    | 3            |
| orders    | 18.910 GB   | 288          |
| part      | 2.204 GB    | 72           |
| partsupp  | 13.125 GB   | 72           |
| region    | 3.270 KB    | 3            |
| supplier  | 335.200 MB  | 108          |
| Total     | 101.197 GB  | 906          |
| Quota     | 1024.000 TB | 1073741824   |
| Left      | 1023.901 TB | 1073740918   |
+-----------+-------------+--------------+
11 rows in set (0.00 sec)
```

基于 lineitem 表，业务有以下查询语句：

```
select l_partkey,l_suppkey,l_quantity from lineitem where l_partkey=xxxx;
```

针对这个 sql，预期可以通过 rollup 进行查询加速。

因为目前并不命中前缀索引，可以通过 rollup，进行 key 调整，使 where 条件命中前缀索引。通过以下语句创建 lineitem 表的 Rollup：

```
alter table lineitem add rollup rutest(l_partkey,l_suppkey,l_quantity);
```

执行 rollup，通过 show alter table rollup; 命令查看物化视图构建进度,确认构建任务完成。

```
MySQL [tpch_100g]> show alter table rollup;
+-------+-----------+---------------------+---------------------+--------------+---------------------------+----------+---------
| JobId | TableName | CreateTime          | FinishTime          | BaseIndexName | RollupIndexName           | RollupId | Transact
+-------+-----------+---------------------+---------------------+--------------+---------------------------+----------+---------
| 15653 | supplier  | 2023-07-11 19:46:58 | 2023-07-11 19:47:28 | supplier      | aggregate_func_rollup_test | 15654    | 2123
| 15703 | supplier  | 2023-07-11 19:58:46 | 2023-07-11 19:59:08 | supplier      | aggregate_func_mv_test     | 15704    | 2125
| 16535 | lineitem  | 2023-07-12 12:31:25 | 2023-07-12 12:37:08 | lineitem      | suppkey_count              | 16536    | 2172
| 16921 | lineitem  | 2023-07-12 16:57:19 | 2023-07-12 17:05:59 | lineitem      | mainkey_aggregate          | 16922    | 2199
| 17700 | lineitem  | 2023-07-13 10:37:48 | 2023-07-13 10:44:39 | lineitem      | mv_02                      | 17701    | 2232
| 18086 | lineitem  | 2023-07-13 10:46:59 | 2023-07-13 10:54:09 | lineitem      | mv_03                      | 18087    | 2237
| 18472 | lineitem  | 2023-07-13 11:22:55 | 2023-07-13 11:27:29 | lineitem      | mv_04                      | 18473    | 2249
| 18858 | lineitem  | 2023-07-13 14:17:04 | 2023-07-13 14:26:10 | lineitem      | mv_05                      | 18859    | 2260
| 19244 | lineitem  | 2023-07-13 14:40:02 | 2023-07-13 14:50:40 | lineitem      | mv_07                      | 19245    | 2273
| 19630 | lineitem  | 2023-07-13 17:28:29 | 2023-07-13 17:30:40 | lineitem      | mv_08                      | 19631    | 2334
| 20016 | lineitem  | 2023-07-13 18:48:36 | 2023-07-13 18:51:00 | lineitem      | mv_01                      | 20017    | 2343
| 20407 | lineitem  | 2023-07-14 15:24:45 | 2023-07-14 15:29:11 | lineitem      | ru_test                    | 20408    | 2375
| 20793 | lineitem  | 2023-07-14 15:38:51 | 2023-07-14 15:44:41 | lineitem      | rutest                     | 20794    | 2385
+-------+-----------+---------------------+---------------------+--------------+---------------------------+----------+---------
13 rows in set (0.00 sec)
```

确认业务查询语句命中 Rollup：

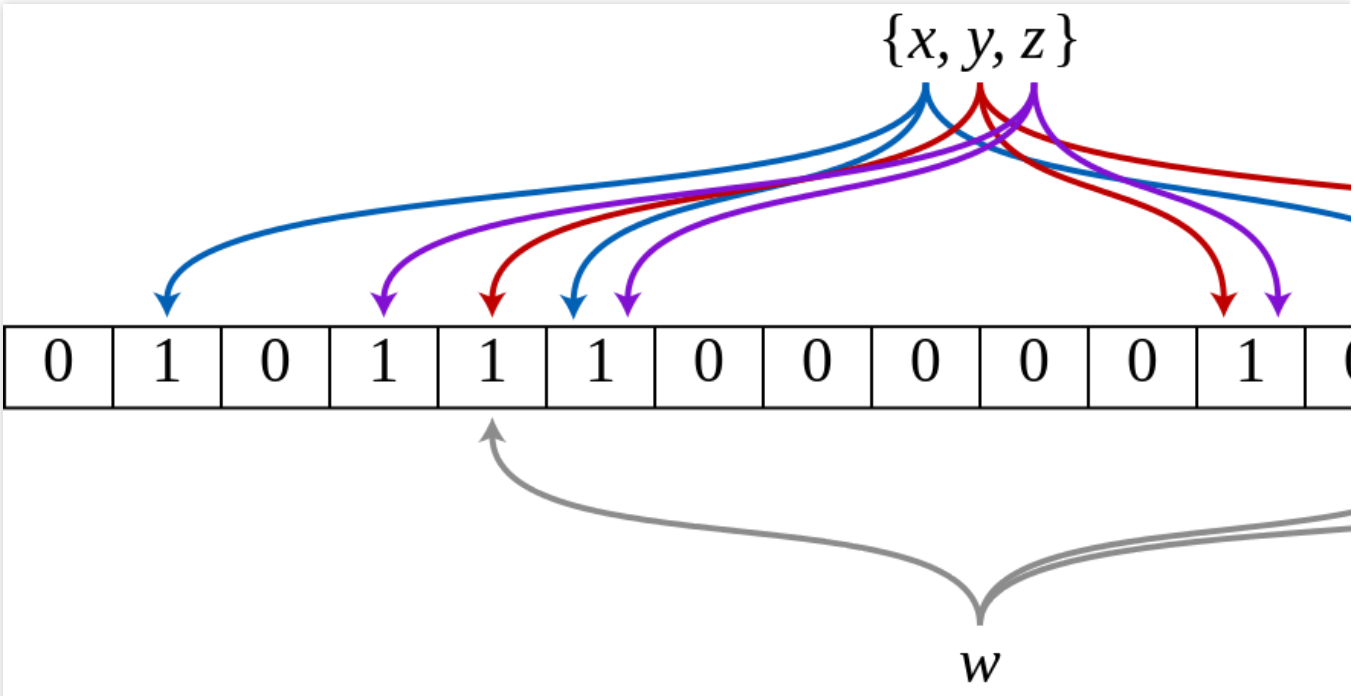
```
explain select l_partkey,l_suppkey,l_quantity from   lineitem where l_partkey= 1018
```

```
MySQL [tpch_100g]> explain select l_partkey,l_suppkey,l_quantity from   lineitem where l_partkey= 10187238;
+-----------------------------------------------------------------------------------+
| Explain String                                                                    |
+-----------------------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                                   |
|   OUTPUT EXPRS:                                                                   |
|     `l_partkey`                                                                   |
|     `l_suppkey`                                                                   |
|     `l_quantity`                                                                  |
|   PARTITION: UNPARTITIONED                                                        |
|                                                                                   |
|   VRESULT SINK                                                                    |
|                                                                                   |
|   1:VEXCHANGE                                                                     |
|      offset: 0                                                                    |
|                                                                                   |
| PLAN FRAGMENT 1                                                                   |
|                                                                                   |
|   PARTITION: HASH_PARTITIONED: `default_cluster:tpch_100g`.`lineitem`.`l_orderkey` |
|                                                                                   |
|   STREAM DATA SINK                                                                |
|     EXCHANGE ID: 01                                                               |
|     UNPARTITIONED                                                                 |
|                                                                                   |
|   0:VOlapScanNode                                                                 |
|      TABLE: default_cluster:tpch_100g.lineitem(rutest), PREAGGREGATION: ON        |
|      PREDICATES: `l_partkey` = 10187238                                           |
|      partitions=1/1, tablets=96/96, tabletList=20795,20799,20803 ...              |
|      cardinality=600037902, avgRowSize=20.292025, numNodes=3                      |
+-----------------------------------------------------------------------------------+
```

BloomFilter 索引

最近更新时间：2024-06-27 11:10:19

BloomFilter 是由 Bloom 在1970年提出的一种多哈希函数映射的快速查找算法。通常应用在一些需要快速判断某个元素是否属于集合，但是并不严格要求100%正确的场合，BloomFilter 有以下特点：
空间效率高的概率型数据结构，用来检查一个元素是否在一个集合中。
对于一个元素检测是否存在的调用，BloomFilter 会告诉调用者两个结果之一：可能存在或者一定不存在。
缺点是存在误判，告诉您可能存在，不一定真实存在。
布隆过滤器实际上是由一个超长的二进制位数组和一系列的哈希函数组成。二进制位数组初始全部为0，当给定一个待查询的元素时，这个元素会被一系列哈希函数计算映射出一系列的值，所有的值在位数组的偏移量处值置为1。
下图所示出一个 m=18，k=3 （m 是该 Bit 数组的大小，k 是 Hash 函数的个数）的 Bloom Filter 示例。集合中的 x、y、z 三个元素通过 3 个不同的哈希函数散列到位数组中。当查询元素 w 时，通过 Hash 函数计算之后因为有一个比特为0，因此 w 不在该集合中。



那么怎么判断某个元素是否在集合中呢？同样是这个元素经过哈希函数计算后得到所有的偏移位置，若这些位置全都为1，则判断这个元素在这个集合中，若有一个不为1，则判断这个元素不在这个集合中。就是这么简单。

Doris BloomFilter 索引及使用场景

举个例子：在 Hbase 中，如果要查找一个占用100字节存储空间大小的短行，一个64KB的 HFile 数据块应该包含(64 * 1024)/100 = 655.53 = ~700行，如果仅能在整个数据块的起始行键上建立索引，那么它是无法给您提供细粒度的索引信息的。因为要查找的行数据可能会落在该数据块的行区间上，也可能行数据没在该数据块上，也可能是表中根

本就不存在该行数据，也或者是行数据在另一个 HFile 里，甚至在 MemStore 里。以上这几种情况，都会导致从磁盘读取数据块时带来额外的 IO 开销，也会滥用数据块的缓存，当面对一个巨大的数据集且处于高并发读时，会严重影响性能。

因此，HBase 提供了布隆过滤器，它允许您对存储在每个数据块的数据做一个反向测试。

当某行被请求时，通过布隆过滤器先检查该行是否不在这个数据块，布隆过滤器会回答“该行不在”/“不知道”。这就是为什么我们称它是反向测试。布隆过滤器同样也可以应用到行里的单元上，当访问某列时可以先使用同样的反向测试。

但布隆过滤器也不是没有代价。存储这个索引会占用额外的空间。布隆过滤器随着它们的索引对象数据增长而增长。当空间不是问题时，它们可以帮助您榨干系统的性能潜力。

Doris 的 BloomFilter 索引可以在建表的时候指定，或者通过表的 ALTER 操作来完成。Bloom Filter 本质上是一种位图结构，用于快速的判断一个给定的值是否在一个集合中。这种判断会产生小概率的误判。即如果返回 false，则一定不在这个集合内。而如果范围true，则有可能不在这个集合内。

Doris 的 BloomFilter 索引也是以 Block 为粒度创建的。每个 Block 中，指定列的值作为一个集合生成一个 BloomFilter 索引条目，用于在查询时快速过滤不满足条件的数据。

创建 BloomFilter 索引

Doris BloomFilter 索引的创建是通过在建表语句的 PROPERTIES 里加上"bloom_filter_columns"="k1,k2,k3"，这个属性，k1，k2，k3是您要创建的 BloomFilter 索引的 Key 列名称，例如下面我们对表里的 saler_id，category_id 创建了 BloomFilter 索引。

```
CREATE TABLE IF NOT EXISTS sale_detail_bloom  (
    sale_date date NOT NULL COMMENT "销售时间",
    customer_id int NOT NULL COMMENT "客户编号",
    saler_id int NOT NULL COMMENT "销售员",
    sku_id int NOT NULL COMMENT "商品编号",
    category_id int NOT NULL COMMENT "商品分类",
    sale_count int NOT NULL COMMENT "销售数量",
    sale_price DECIMAL(12,2) NOT NULL COMMENT "单价",
    sale_amt DECIMAL(20,2)  COMMENT "销售总金额"
)
Duplicate  KEY(sale_date, customer_id,saler_id,sku_id,category_id)
PARTITION BY RANGE(sale_date)
(
PARTITION P_202111 VALUES [('2021-11-01'), ('2021-12-01'))
)
DISTRIBUTED BY HASH(saler_id) BUCKETS 10
PROPERTIES (
"replication_num" = "3",
"bloom_filter_columns"="saler_id,category_id",
"dynamic_partition.enable" = "true",
```

```
"dynamic_partition.time_unit" = "MONTH",
"dynamic_partition.time_zone" = "Asia/Shanghai",
"dynamic_partition.start" = "-2147483648",
"dynamic_partition.end" = "2",
"dynamic_partition.prefix" = "P_",
"dynamic_partition.replication_num" = "3",
"dynamic_partition.buckets" = "3"
);
```

也可以对已经创建的表进行修改，指定列构建 Bloomfilter 索引。

```
alter table sale_detail_bloom set ("bloom_filter_columns" = "saler_id,category_id")
```

查看 BloomFilter 索引

查看我们在表上建立的 BloomFilter 索引是使用：

```
SHOW CREATE TABLE <table_name>;
```

删除 BloomFilter 索引

删除索引即为将索引列从 bloom_filter_columns 属性中移除：

```
ALTER TABLE <db.table_name> SET ("bloom_filter_columns" = "");
```

修改 BloomFilter 索引

修改索引即为修改表的 bloom_filter_columns 属性：

```
ALTER TABLE <db.table_name> SET ("bloom_filter_columns" = "k1,k3");
```

Doris BloomFilter 使用场景

满足以下几个条件时可以考虑对某列建立 BloomFilter 索引：

1. 首先 BloomFilter 适用于非前缀过滤。
2. 查询会根据该列高频过滤，而且查询条件大多是 in 和 = 过滤。

3. 不同于 Bitmap，BloomFilter 适用于高基数列。例如 UserID。因为如果创建在低基数的列上，例如 “性别” 列，则每个 Block 几乎都会包含所有取值，导致 BloomFilter 索引失去意义。

Doris BloomFilter 使用注意事项

1. 不支持对 Tinyint、Float、Double 类型的列建 BloomFilter 索引。
2. BloomFilter 索引只对 in 和 = 过滤查询有加速效果。
3. 如果要查看某个查询是否命中了 BloomFilter 索引，可以通过查询的 Profile 信息查看。
4. 支持对一个表的多个列分别构建 bloomfilter index，但是某一列只能支持bitmap index/bloomfilter index 中的一个。

使用条件

支持的列类型：

```
SMALLINT
INT
UNSIGNEDINT
BIGINT
LARGEINT
CHAR
VARCHAR
DATE
DATETIME
DECIMAL
其他类型暂不支持
```

查询生效的条件：

"=" ,例如：field = value

"is" ,例如：field IS NULL

"in" ,例如：field IN {value1, value2, ...}

最佳实践

以下通过对 lineitem2 表的查询优化，展示 Bloomfilter 索引的最佳实践。lineitem2 表有6亿数据。

```
MySQL [tpch_100g]> desc lineitem2 all;
+-----------+---------------+-----------------+---------------+---------------+------+-------+
| IndexName | IndexKeysType | Field           | Type          | InternalType  | Null | Key   |
+-----------+---------------+-----------------+---------------+---------------+------+-------+
| lineitem2 | DUP_KEYS      | l_shipdate      | DATE          | DATE          | No   | true  |
|           |               | l_orderkey      | BIGINT        | BIGINT        | No   | true  |
|           |               | l_linenumber    | INT           | INT           | No   | false |
|           |               | l_partkey       | INT           | INT           | No   | false |
|           |               | l_suppkey       | INT           | INT           | No   | false |
|           |               | l_quantity      | DECIMAL(15,2) | DECIMAL(15,2) | No   | false |
|           |               | l_extendedprice | DECIMAL(15,2) | DECIMAL(15,2) | No   | false |
|           |               | l_discount      | DECIMAL(15,2) | DECIMAL(15,2) | No   | false |
|           |               | l_tax           | DECIMAL(15,2) | DECIMAL(15,2) | No   | false |
|           |               | l_returnflag    | VARCHAR(1)    | VARCHAR(1)    | No   | false |
|           |               | l_linestatus    | VARCHAR(1)    | VARCHAR(1)    | No   | false |
|           |               | l_commitdate    | DATE          | DATE          | No   | false |
|           |               | l_receiptdate   | DATE          | DATE          | No   | false |
|           |               | l_shipinstruct  | VARCHAR(25)   | VARCHAR(25)   | No   | false |
|           |               | l_shipmode      | VARCHAR(10)   | VARCHAR(10)   | No   | false |
|           |               | l_comment       | VARCHAR(44)   | VARCHAR(44)   | No   | false |
+-----------+---------------+-----------------+---------------+---------------+------+-------+
16 rows in set (0.04 sec)

MySQL [tpch_100g]> select count(*) from lineitem2;
+-----------+
| count(*)  |
+-----------+
| 600037902 |
+-----------+
1 row in set (0.07 sec)
```

假设有以下查询 sql 需要进行优化：

```
select l_partkey,l_orderkey from lineitem2 where l_partkey =xxx;
```

分析查询 sql 是否可以使用 Bloomfilter index 进行优化

sql 的查询条件 l_partkey 对应的列取值为 2000W，识别度较高，字段类型为 INT，查询时使用等值条件，可以使用 Bloomfilter 进行优化。

创建和等待 Bloomfilter index 构建完成

创建 Bloomfilter index：

```
alter table lineitem2 set ("bloom_filter_columns" = "l_partkey");
```

通过 show alter table column 查看索引构建是否完成，注意如果未构建完成，查询过程时不会使用 Bloomfilter index 的。

```
MySQL [tpch_100g]> show alter table column;
+-------+-----------+-------------------------+-------------------------+-----------+---------+---------------+----------------+
| JobId | TableName | CreateTime              | FinishTime              | IndexName | IndexId | OriginIndexId | SchemaVersion  |
+-------+-----------+-------------------------+-------------------------+-----------+---------+---------------+----------------+
| 21216 | lineitem  | 2023-07-17 11:20:04.000 | 2023-07-17 11:24:00.186 | lineitem  | 21217   | 12624         | 1:345227157    |
| 21216 | lineitem  | 2023-07-17 11:20:04.000 | 2023-07-17 11:24:00.186 | mv_01     | 21602   | 20017         | 1:1027292799   |
| 21216 | lineitem  | 2023-07-17 11:20:04.000 | 2023-07-17 11:24:00.186 | rutest    | 21987   | 20794         | 1:1046778555   |
| 22760 | lineitem  | 2023-07-17 14:05:49.210 | 2023-07-17 14:09:33.765 | lineitem  | 22761   | 21217         | 2:496593265    |
| 22760 | lineitem  | 2023-07-17 14:05:49.210 | 2023-07-17 14:09:33.765 | mv_01     | 23146   | 21602         | 2:1299108797   |
| 22760 | lineitem  | 2023-07-17 14:05:49.210 | 2023-07-17 14:09:33.765 | rutest    | 23531   | 21987         | 2:1363412795   |
| 23917 | lineitem  | 2023-07-17 14:26:10.504 | 2023-07-17 14:30:37.477 | lineitem  | 23918   | 22761         | 3:761594879    |
| 24304 | lineitem  | 2023-07-17 14:51:51.935 | 2023-07-17 15:12:13.407 | lineitem  | 24305   | 23918         | 4:810741075    |
| 24723 | lineitem  | 2023-07-17 16:08:18.880 | 2023-07-17 16:28:03.666 | lineitem  | 24724   | 24305         | 5:125846595    |
| 25110 | lineitem  | 2023-07-17 16:58:02.647 | 2023-07-17 17:44:15.965 | lineitem  | 25111   | 24724         | 6:350391292    |
| 25501 | lineitem2 | 2023-07-18 10:26:01.111 | 2023-07-18 10:31:46.995 | lineitem2 | 25502   | 17314         | 1:961505829    |
+-------+-----------+-------------------------+-------------------------+-----------+---------+---------------+----------------+
11 rows in set (0.00 sec)
```

创建成功后，通过 show create table 语句可以看到。

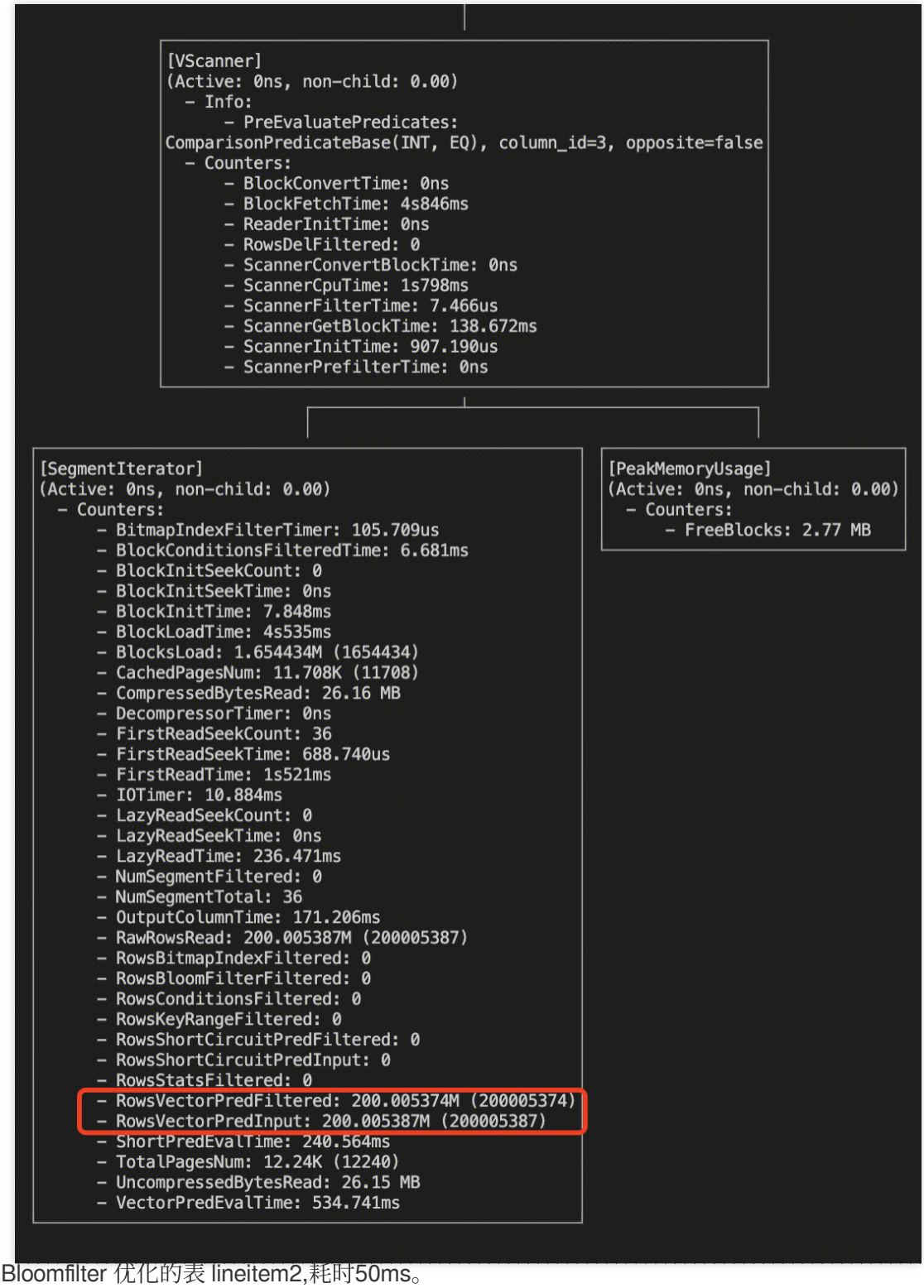
```
| lineitem2 | CREATE TABLE `lineitem2` (
  `l_shipdate` date NOT NULL,
  `l_orderkey` bigint(20) NOT NULL,
  `l_linenumber` int(11) NOT NULL,
  `l_partkey` int(11) NOT NULL,
  `l_suppkey` int(11) NOT NULL,
  `l_quantity` decimal(15, 2) NOT NULL,
  `l_extendedprice` decimal(15, 2) NOT NULL,
  `l_discount` decimal(15, 2) NOT NULL,
  `l_tax` decimal(15, 2) NOT NULL,
  `l_returnflag` varchar(1) NOT NULL,
  `l_linestatus` varchar(1) NOT NULL,
  `l_commitdate` date NOT NULL,
  `l_receiptdate` date NOT NULL,
  `l_shipinstruct` varchar(25) NOT NULL,
  `l_shipmode` varchar(10) NOT NULL,
  `l_comment` varchar(44) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(`l_shipdate`, `l_orderkey`)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(`l_orderkey`) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"bloom_filter_columns" = "l_partkey",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
); |
```

验证优化生效

对没有优化的表 lineitem，耗时490ms。

```
MySQL [tpch_100g]> select l_partkey ,l_orderkey from lineitem where l_partkey =18851800;
+------------+-------------+
| l_partkey | l_orderkey |
+------------+-------------+
|  18851800 |  224765380 |
|  18851800 |  321002755 |
|  18851800 |  353048935 |
|  18851800 |  540573248 |
|  18851800 |   76608484 |
|  18851800 |  205390308 |
|  18851800 |  377718118 |
|  18851800 |  474167366 |
|  18851800 |  352124418 |
|  18851800 |  421346599 |
|  18851800 |  305765702 |
|  18851800 |  210268359 |
|  18851800 |  219414821 |
|  18851800 |  440029478 |
|  18851800 |  122797607 |
|  18851800 |  359076964 |
|  18851800 |  280181734 |
|  18851800 |   32528448 |
|  18851800 |  542927936 |
|  18851800 |  289145253 |
|  18851800 |  350422887 |
|  18851800 |  200909666 |
|  18851800 |   91146402 |
|  18851800 |  226586309 |
|  18851800 |   12835713 |
|  18851800 |   41715395 |
|  18851800 |  304229987 |
|  18851800 |  257264258 |
|  18851800 |  188688642 |
|  18851800 |   23222656 |
|  18851800 |  403302980 |
|  18851800 |  371814022 |
|  18851800 |  427804998 |
|  18851800 |  226532740 |
|  18851800 |  334523904 |
+------------+-------------+
35 rows in set (0.49 sec)
```

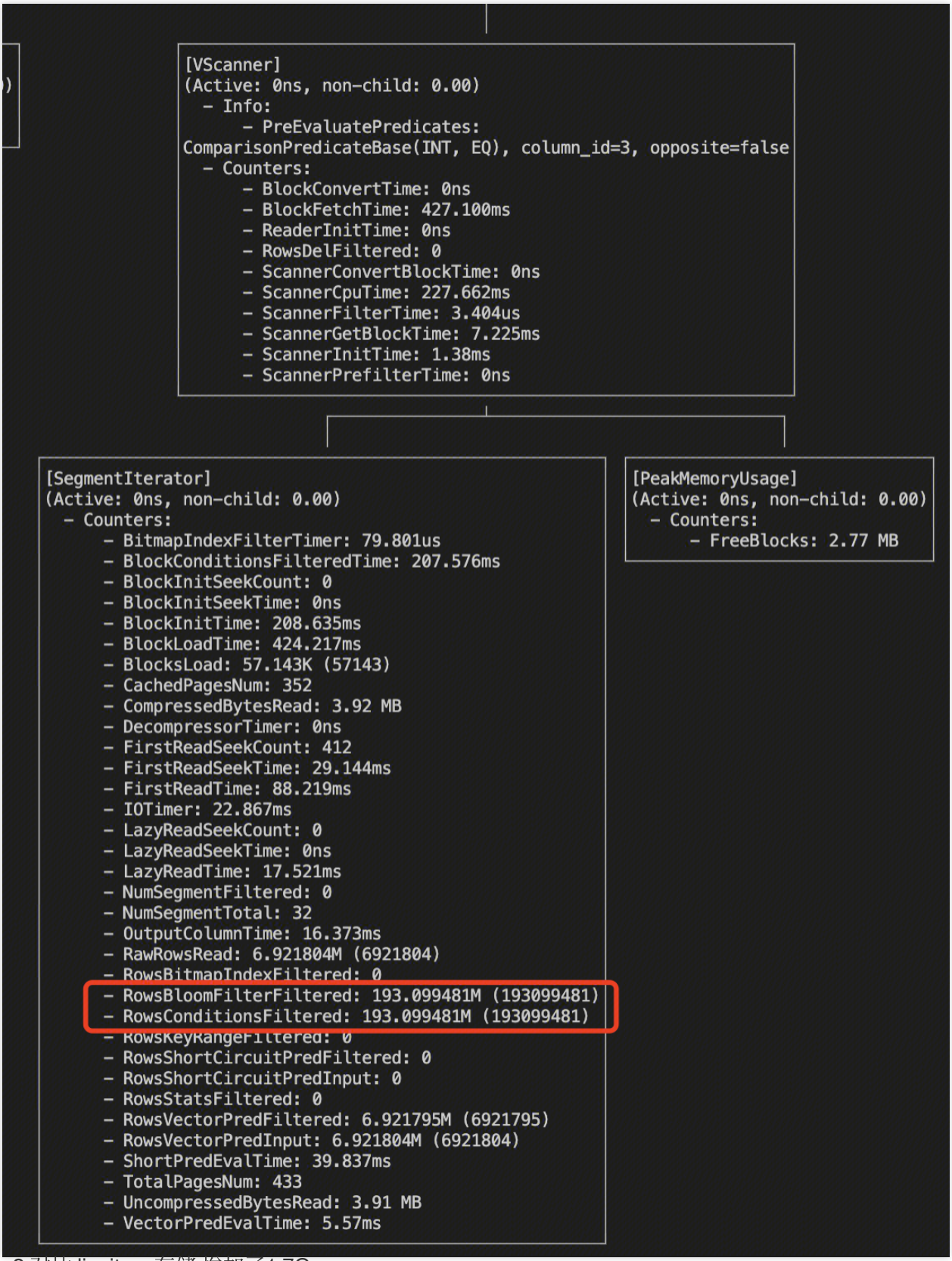
profile 看直接使用向量化能力过滤。



使用 Bloomfilter 优化的表 lineitem2,耗时50ms。

```
MySQL [tpch_100g]> select l_partkey ,l_orderkey from lineitem2 where l_partkey =18851800;
+-----------+------------+
| l_partkey | l_orderkey |
+-----------+------------+
|  18851800 |   23222656 |
|  18851800 |  122797607 |
|  18851800 |  427804998 |
|  18851800 |  226532740 |
|  18851800 |  353048935 |
|  18851800 |  257264258 |
|  18851800 |  540573248 |
|  18851800 |  352124418 |
|  18851800 |  224765380 |
|  18851800 |   41715395 |
|  18851800 |  304229987 |
|  18851800 |   91146402 |
|  18851800 |  226586309 |
|  18851800 |   12835713 |
|  18851800 |  188688642 |
|  18851800 |  321002755 |
|  18851800 |   76608484 |
|  18851800 |  210268359 |
|  18851800 |  371814022 |
|  18851800 |  205390308 |
|  18851800 |  305765702 |
|  18851800 |  289145253 |
|  18851800 |  474167366 |
|  18851800 |  403302980 |
|  18851800 |  421346599 |
|  18851800 |  359076964 |
|  18851800 |   32528448 |
|  18851800 |  542927936 |
|  18851800 |  280181734 |
|  18851800 |  200909666 |
|  18851800 |  219414821 |
|  18851800 |  440029478 |
|  18851800 |  350422887 |
|  18851800 |  334523904 |
|  18851800 |  377718118 |
+-----------+------------+
35 rows in set (0.05 sec)
```

profile 看，使用了 Bloomfilter index 过滤 + 少量向量化能力过滤。



lineitem2 对比 lineitem 存储 增加了1.7G。

```
MySQL [tpch_100g]> show data;
+---------------------------+-------------+---------------+
| TableName                 | Size        | ReplicaCount  |
+---------------------------+-------------+---------------+
| create_table_with_bitmap  | 0.000       | 288           |
| customer                  | 3.951 GB    | 72            |
| lineitem                  | 62.679 GB   | 288           |
| lineitem2                 | 64.316 GB   | 288           |
| nation                    | 7.714 KB    | 3             |
| orders                    | 18.910 GB   | 288           |
| part                      | 2.204 GB    | 72            |
| partsupp                  | 13.125 GB   | 72            |
| region                    | 3.270 KB    | 3             |
| supplier                  | 335.200 MB  | 108           |
| Total                     | 165.513 GB  | 1482          |
| Quota                     | 1024.000 TB | 1073741824    |
| Left                      | 1023.838 TB | 1073740342    |
+---------------------------+-------------+---------------+
13 rows in set (0.00 sec)
```

Bitmap 索引

最近更新时间：2024-06-27 11:10:40

用户可以通过创建 bitmap index 加速查询。
本文档主要介绍如何创建 bitmap 索引，以及创建 bitmap 索引的一些注意事项和常见问题。

名词解释

bitmap 索引：用位图表示的索引，对索引列的每个键值建立一个位图。是一种快速数据结构，能够加快查询速度。
索引原理示意：

ID	Name	Gender	Bitmap Index1	
			M	F
1	Mike	M	1	0
2	Susan	F	0	1
3	Lily	F	0	1
4	Alex	M	1	0
5	Tony	M	1	0
在Gender列上建bitmap索引				

原理介绍

主要针对大量相同值的列而创建（例如：类别，操作员，部门 ID，库房 ID 等），索引块的一个索引行中存储键值和起止 Rowid，以及这些键值的位置编码，位置编码中的每一位表示键值对应的数据行的有无。一个块可能指向的是几十甚至成百上千行数据的位置。
当根据键值查询时，可以根据起始 Rowid 和位图状态，快速定位数据。
当根据键值做 and，or 或 in(x,y,..)查询时，直接用索引的位图进行位运算，快速得出结果行数据。
当执行 select count(xx) 时，可以直接访问索引就快速得出统计数据。

语法

创建和删除本质上是一个 schema change 的作业，具体细节可以参照 Schema Change。

创建索引

支持在创建表时就进行 Bitmap 索引指定。

```
CREATE TABLE create_table_with_bitmap (
  l_shipdate date NOT NULL,
  l_comment varchar(44) NOT NULL,
  INDEX shipdate_bm_index (l_shipdate) USING BITMAP COMMENT 'shipdate bitmap index'
) ENGINE=OLAP
DUPLICATE KEY(l_shipdate)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(l_shipdate) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
);
```

也支持对现有的表进行修改增加某一列的 Bitmap 索引，例如在 table1 上为 siteid 创建 Bitmap 索引：

```
CREATE INDEX [IF NOT EXISTS] index_name ON table1 (siteid) USING BITMAP COMMENT 'ba
```

查看索引

展示指定 table_name 的下索引：

```
SHOW INDEX FROM example_db.table_name;
```

删除索引

删除指定 table_name 的下索引：

```
DROP INDEX [IF EXISTS] index_name ON [db_name.]table_name;
```

```
MySQL [example_db]> CREATE INDEX index_name ON sale_detail_bloom (saler_id) USING BITMAP COMMENT 'balabala';
Query OK, 0 rows affected (0.00 sec)

MySQL [example_db]> show index from sale_detail_bloom;
+-----------------------------------------+------------+------------+--------------+-------------+-------
---------+----------+
| Table                                   | Non_unique | Key_name   | Seq_in_index | Column_name | Collat
ndex_type | Comment  |
+-----------------------------------------+------------+------------+--------------+-------------+-------
---------+----------+
| default_cluster:example_db.sale_detail_bloom |          | index_name |              | saler_id    |
ITMAP     | balabala |
+-----------------------------------------+------------+------------+--------------+-------------+-------
---------+----------+
1 row in set (0.00 sec)

MySQL [example_db]> drop INDEX index_name ON sale_detail_bloom;
Query OK, 0 rows affected (0.01 sec)

MySQL [example_db]> show index from sale_detail_bloom;
Empty set (0.00 sec)
```

基本原理

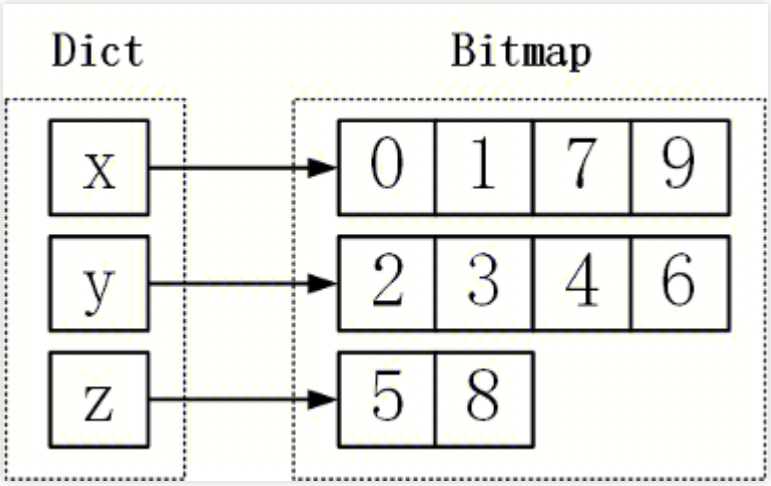
腾讯云数据仓库 TChouse-D 使用的是列存储, 在没有索引的情况下，要判断某列F是否与某个值 V 相等，只能遍历一整列所有的数据，才能得到与 V 相等的行号。Bitmap 索引为了避免检索时对所有数据进行遍历，在原来列存数据的基础上，顺序保存列所有的值（有序字典），以及每个值对应的行号。这样在检索时，就可以通过值，快速查找到等于这个值的所有行号。

例如一列数据为[x, x, y, y, y, z, y, x, z, x]，一共包含10行，则该列数据的 Bitmap 索引的有序字典为{x, y, z}, x、y、z对应的位图分别为：

x的位图: [0, 1, 7, 9]

y的位图: [2, 3, 4, 6]

z的位图: [5, 8]



注意事项

目前索引命令（[show | create | drop] index）仅支持 bitmap 类型的索引。
bitmap 索引仅在单列上创建。
bitmap 索引能够应用在 `Duplicate` 、 `Uniq` 数据模型的所有列和 `Aggregate` 模型的 key 列上。
bitmap 索引支持的数据类型如下：

`TINYINT`
`SMALLINT`
`INT`
`BIGINT`
`CHAR`
`VARCHAR`
`DATE`
`DATETIME`
`LARGEINT`
`DECIMAL`
`BOOL`

bitmap 索引仅在 Segment V2 下生效。当创建 index 时，表的存储格式将默认转换为 V2 格式（查看表的存储格式可通过 show create table table_name 命令）。

bitmap 索引适用场景

适用于低基数的列上，建议在100到100,000之间，如：职业、地市等。重复度过高则对比其他类型索引没有明显优势；重复度过低，则空间效率和性能会大大降低。
特定类型的查询例如 count、or、and 等逻辑操作因为只需要进行位运算。如：通过多个条件组合查询， `select count(*) from table where job = '医生' and phonetype = 'iphone' and gender ='男';` 类似这种场景，如果在每个查询条件列上都建立了 bitmap 索引，则数据库可以进行高效的位运算，精确定位到需要的数据，减少磁盘 IO。并且筛选出的结果集越小，bitmap 索引的优势越明显。
适用于即席查询、多维分析等 OLAP 场景。如果有一张表有100列，用户会使用其中的20个列作为查询条件（任意使用这20个列上的N的列），几乎没有办法创建合适的 b-tree 索引。但是在这些列上创建20个 bitmap 索引，那么所有的查询都可以应用到索引。
Bitmap索引支持等值查询和范围查找,支持= , < , <= ,> ,>= 表达式。

bitmap 索引不适用场景

值重复度低的列，如：身份证号、手机号码等。
重复度过高的列，如：性别，可以建立 bitmap 索引，但不建议单独作为查询条件使用，建议与其他条件共同过滤。
经常需要更新修改的列。

最佳实践

以下基于 TPCH 测试集的 lineitem 表(6亿条记录), 构建 Bitmap 索引和使用。

```
-------------------------------------+
| lineitem | CREATE TABLE `lineitem` (
  `l_shipdate` date NOT NULL,
  `l_orderkey` bigint(20) NOT NULL,
  `l_linenumber` int(11) NOT NULL,
  `l_partkey` int(11) NOT NULL,
  `l_suppkey` int(11) NOT NULL,
  `l_quantity` decimal(15, 2) NOT NULL,
  `l_extendedprice` decimal(15, 2) NOT NULL,
  `l_discount` decimal(15, 2) NOT NULL,
  `l_tax` decimal(15, 2) NOT NULL,
  `l_returnflag` varchar(1) NOT NULL,
  `l_linestatus` varchar(1) NOT NULL,
  `l_commitdate` date NOT NULL,
  `l_receiptdate` date NOT NULL,
  `l_shipinstruct` varchar(25) NOT NULL,
  `l_shipmode` varchar(10) NOT NULL,
  `l_comment` varchar(44) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(`l_shipdate`, `l_orderkey`)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(`l_orderkey`) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
); |
+----------+-----------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-----------------------------------------------------------------
-------------------------------------+
1 row in set (0.00 sec)

MySQL [tpch_100g]> select count(*) from lineitem;
+-----------+
| count(*)  |
+-----------+
| 600037902 |
+-----------+
1 row in set (0.06 sec)

MySQL [tpch_100g]> show data;
+--------------------------+-------------+--------------+
| TableName                | Size        | ReplicaCount |
+--------------------------+-------------+--------------+
| create_table_with_bitmap | 0.000       | 288          |
| customer                 | 3.951 GB    | 72           |
| lineitem                 | 69.525 GB   | 864          |
| lineitem2                | 62.638 GB   | 288          |
```

假设对于 lineitem 表，预期优化以下 sql，优化前耗时约为900ms。

```
select l_quantity,l_returnflag,l_commitdate from lineitem where l_suppkey = 'xxx';
```

分析是否适用 Bitmap 索引

查询条件 l_suppkey，此时并不命中前缀索引也没有其他索引。l_suppkey 字段类型为INT ,为可以使用 Bitmap 索引的类型。

查看 l_suppkey 这一列的取值基数，在6亿数据表上，l_suppkey 的取值为100W，取值基数并没有特别高，这里尝试使用 Bitmap 索引进行优化。

```
MySQL [tpch_100g]> select count(*) from lineitem;
+-----------+
| count(*)  |
+-----------+
| 600037902 |
+-----------+
1 row in set (0.10 sec)

MySQL [tpch_100g]> select count(distinct l_suppkey) from lineitem;
+---------------------------------+
| count(DISTINCT `l_suppkey`) |
+---------------------------------+
|                         1000000 |
+---------------------------------+
1 row in set (38.32 sec)
```

创建 Bitmap 和等待索引构建

创建 Bitmap 索引：

```
create index suppkey_bitmap_index on lineitem(l_suppkey) using bitmap comment 'test
```

通过 show alter table column 命令查看索引构建进度：

```
MySQL [tpch_100g]> show alter table column;
+-------+-----------+-------------------------+-------------------------+-----------+---------+---------------+---------------+-----
| JobId | TableName | CreateTime              | FinishTime              | IndexName | IndexId | OriginIndexId | SchemaVersion | Tra
+-------+-----------+-------------------------+-------------------------+-----------+---------+---------------+---------------+-----
| 21216 | lineitem  | 2023-07-17 11:20:04.000 | 2023-07-17 11:24:00.186 | lineitem  | 21217   | 12624         | 1:345227157   | 241
| 21216 | lineitem  | 2023-07-17 11:20:04.000 | 2023-07-17 11:24:00.186 | mv_01     | 21602   | 20017         | 1:1027292799  | 241
| 21216 | lineitem  | 2023-07-17 11:20:04.000 | 2023-07-17 11:24:00.186 | rutest    | 21987   | 20794         | 1:1046778555  | 241
| 22760 | lineitem  | 2023-07-17 14:05:49.210 | 2023-07-17 14:09:33.765 | lineitem  | 22761   | 21217         | 2:496593265   | 241
| 22760 | lineitem  | 2023-07-17 14:05:49.210 | 2023-07-17 14:09:33.765 | mv_01     | 23146   | 21602         | 2:1299108797  | 241
| 22760 | lineitem  | 2023-07-17 14:05:49.210 | 2023-07-17 14:09:33.765 | rutest    | 23531   | 21987         | 2:1363412795  | 241
| 23917 | lineitem  | 2023-07-17 14:26:10.504 | 2023-07-17 14:30:37.477 | lineitem  | 23918   | 22761         | 3:761594879   | 242
| 24304 | lineitem  | 2023-07-17 14:51:51.935 | 2023-07-17 15:12:13.407 | lineitem  | 24305   | 23918         | 4:810741075   | 244
| 24723 | lineitem  | 2023-07-17 16:08:18.880 | 2023-07-17 16:28:03.666 | lineitem  | 24724   | 24305         | 5:125846595   | 250
+-------+-----------+-------------------------+-------------------------+-----------+---------+---------------+---------------+-----
9 rows in set (0.00 sec)
```

通过 show create table lineitem 命令查看 lineitem 表存在的索引：

```
| lineitem | CREATE TABLE `lineitem` (
  `l_shipdate` date NOT NULL,
  `l_orderkey` bigint(20) NOT NULL,
  `l_linenumber` int(11) NOT NULL,
  `l_partkey` int(11) NOT NULL,
  `l_suppkey` int(11) NOT NULL,
  `l_quantity` decimal(15, 2) NOT NULL,
  `l_extendedprice` decimal(15, 2) NOT NULL,
  `l_discount` decimal(15, 2) NOT NULL,
  `l_tax` decimal(15, 2) NOT NULL,
  `l_returnflag` varchar(1) NOT NULL,
  `l_linestatus` varchar(1) NOT NULL,
  `l_commitdate` date NOT NULL,
  `l_receiptdate` date NOT NULL,
  `l_shipinstruct` varchar(25) NOT NULL,
  `l_shipmode` varchar(10) NOT NULL,
  `l_comment` varchar(44) NOT NULL,
  INDEX suppkey_bitmap_index (`l_suppkey`) USING BITMAP COMMENT 'test'
) ENGINE=OLAP
DUPLICATE KEY(`l_shipdate`, `l_orderkey`)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(`l_orderkey`) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
); |
```

对比优化前后效果

优化后，耗时40ms。

```
+------------+-------------+--------------+
608 rows in set (0.04 sec)

MySQL [tpch_100g]> select l_quantity,l_returnflag,l_commitdate from lineitem where l_suppkey =
```

优化前，耗时240ms（这里使用的 lineitem2 表是 lineitem 表的备份，数据量和表结构一致）。

```
|       28.00 | N            | 1997-08-13   |
+-------------+--------------+--------------+
608 rows in set (0.24 sec)

MySQL [tpch_100g]> select l_quantity,l_returnflag,l_commitdate from lineitem2 where l_suppkey =
```

常见问题

select count(*) 对 Bitmap 索引列生效吗?

生效，通过查看对 `select count(*) from lineitem where l_suppkey = '388041';` 语句的profile，可以看到 RowsBitmapIndexFiltered 字段帮助过滤了大量数据。

RowsBitmapIndexFiltered: 200.01525M (200015250)

列的取值基数比较大，为什么不适合使用 bitmap 索引？

实测对6亿记录的 lineitem 的 l_partkey 列（2000W取值）进行 bitmap 索引构建。优化前点查耗时600ms,优化后点查耗时为120ms。效率提升了5倍。可见对基数高的列建立索引，查询也能带来效率提升。
但是存储消耗比较大。lineitem 表6亿数据3副本占用约62G存储，对 l_partkey 列构建完 bitmap 索引后，lineitem 表存储涨到95G。理论上，bitmap 索引构建增长的存储，在未压缩情况，为 2000W（取值）* 60000W（行数）bit。
可见对高基数列构建 bitmap 索引，性价比并不高，这个时候更推荐 bloomfilter 索引。

物化视图

最近更新时间：2024-06-27 11:11:03

物化视图是将预先计算（根据定义好的 SELECT 语句）好的数据集，存储在 Doris 中的一个特殊的表。
物化视图的出现主要是为了满足用户，既能对原始明细数据的任意维度分析，也能快速的对固定维度进行分析查询。

适用场景

分析需求覆盖明细数据查询以及固定维度查询两方面。
查询仅涉及表中的很小一部分列或行。
查询包含一些耗时处理操作，例如：时间很久的聚合操作等。
查询需要匹配不同前缀索引。

优势

对于那些经常重复的使用相同的子查询结果的查询性能大幅提升。
Doris 自动维护物化视图的数据，无论是新的导入，还是删除操作都能保证 base 表和物化视图表的数据一致性。无需任何额外的人工维护成本。
查询时，会自动匹配到最优物化视图，并直接从物化视图中读取数据。
说明
自动维护物化视图的数据会造成一些维护开销，会在后面的物化视图的局限性中展开说明。

物化视图 VS Rollup

在没有物化视图功能之前，用户一般都是使用 Rollup 功能通过预聚合方式提升查询效率的。但是 Rollup 具有一定的局限性，他不能基于明细模型做预聚合。
物化视图则在覆盖了 Rollup 的功能的同时，还能支持更丰富的聚合函数。所以物化视图其实是 Rollup 的一个超集。
也就是说，之前 ALTER TABLE ADD ROLLUP 语法支持的功能现在均可以通过 CREATE MATERIALIZED VIEW 实现。
物化视图和 Rollup 支持的聚合函数不同，物化视图支持更多的聚合函数（ replace 聚合方式因为不会在查询上使用，所以没有列出来）

聚合函数	SUM	MAX	SUM	COUNT	BITMAP_UNION	HLL_UNION
Rollup	true	true	true	false	false	false

物化视图	true	true	true	true	true	true

使用物化视图

Doris 系统提供了一整套对物化视图的 DDL 语法，包括创建，查看，删除。DDL 的语法和 PostgreSQL，Oracle 都是一致的。
查询或者子查询中的所有列都要在物化视图里面，即查询所涉及的所有列（包括 select list 和 where 中的查询条件列等）都存在于该物化视图的列中。否则，查询只能命中 Base 表。
能用的上前缀索引的列上的条件需要是 **= < > <= >= in between** 这些并且这些条件是并列的且关系使用 **and** 连接，对于 **or** 、 **!=** 等这些不能命中。

创建物化视图

这里首先您要根据您的查询语句的特点来决定创建一个什么样的物化视图。这里并不是说您的物化视图定义和您的某个查询语句一模一样就最好。这里有两个原则：
1. 从查询语句中**抽象**出，多个查询共有的分组和聚合方式作为物化视图的定义。
2. 不需要给所有维度组合都创建物化视图。
首先第一个点，一个物化视图如果抽象出来，并且多个查询都可以匹配到这张物化视图。这种物化视图效果最好。因为物化视图的维护本身也需要消耗资源。
如果物化视图只和某个特殊的查询很贴合，而其他查询均用不到这个物化视图。则会导致这张物化视图的性价比不高，既占用了集群的存储资源，还不能为更多的查询服务。
所以用户需要结合自己的查询语句，以及数据维度信息去抽象出一些物化视图的定义。
第二点就是，在实际的分析查询中，并不会覆盖到所有的维度分析。所以给常用的维度组合创建物化视图即可，从而到达一个空间和时间上的平衡。
通过下面命令就可以创建物化视图了。创建物化视图是一个异步的操作，也就是说用户成功提交创建任务后，Doris 会在后台对存量的数据进行计算，直到创建成功。
具体的语法可以通过下面命令查看：

```
HELP CREATE MATERIALIZED VIEW
```

支持聚合函数

目前物化视图创建语句支持的聚合函数有：
SUM，MIN，MAX (Version 0.12)。
COUNT，BITMAP_UNION，HLL_UNION (Version 0.13)。
BITMAP_UNION 的形式必须为：`BITMAP_UNION(TO_BITMAP(COLUMN))` column 列的类型只能是整数（largeint 也不支持），或者 `BITMAP_UNION(COLUMN)` 且 base 表为 AGG 模型。

HLL_UNION 的形式必须为： `HLL_UNION(HLL_HASH(COLUMN))` column 列的类型不能是 DECIMAL ，或者 `HLL_UNION(COLUMN)` 且 base 表为 AGG 模型。

更新策略

为保证物化视图表和 Base 表的数据一致性，Doris 会将导入，删除等对 base 表的操作都同步到物化视图表中。并且通过增量更新的方式来提升更新效率。通过事务方式来保证原子性。
如果用户通过 INSERT 命令插入数据到 base 表中，则这条数据会同步插入到物化视图中。当 base 表和物化视图表均写入成功后，INSERT 命令才会成功返回。

查询自动匹配

物化视图创建成功后，用户的查询不需要发生任何改变，也就是还是查询的 base 表。Doris 会根据当前查询的语句去自动选择一个最优的物化视图，从物化视图中读取数据并计算。
用户可以通过 EXPLAIN 命令来检查当前查询是否使用了物化视图。
物化视图中的聚合和查询中聚合的匹配关系：

物化视图聚合	查询中聚合
sum	sum
min	min
max	max
count	count
bitmap_union	bitmap_union,bitmap_union_count, count(distinct)
hll_union	hll_raw_agg, hll_union_agg, ndv, approx_count_distinct

其中 bitmap 和 hll 的聚合函数在查询匹配到物化视图后，查询的聚合算子会根据物化视图的表结构进行一个改写。详情请参见最佳实践2。

查询物化视图

查看当前表都有哪些物化视图，以及他们的表结构都是什么样的。通过下面命令：

```
MySQL [test]> desc mv_test all;
+-----------+---------------+-----------------+----------+------+-------+---------+
| IndexName | IndexKeysType | Field           | Type     | Null | Key   | Default |
+-----------+---------------+-----------------+----------+------+-------+---------+
| mv_test   | DUP_KEYS      | k1              | INT      | Yes  | true  | NULL    |
|           |               | k2              | BIGINT   | Yes  | true  | NULL    |
|           |               | k3              | LARGEINT | Yes  | true  | NULL    |
|           |               | k4              | SMALLINT | Yes  | false | NULL    |
```

```
|           |               |                  |           |      |       |          |
| mv_2      | AGG_KEYS      | k2               | BIGINT    | Yes  | true  | NULL     |
|           |               | k4               | SMALLINT  | Yes  | false | NULL     |
|           |               | k1               | INT       | Yes  | false | NULL     |
|           |               |                  |           |      |       |          |
| mv_3      | AGG_KEYS      | k1               | INT       | Yes  | true  | NULL     |
|           |               | to_bitmap(`k2`)  | BITMAP    | No   | false |          |
|           |               |                  |           |      |       |          |
| mv_1      | AGG_KEYS      | k4               | SMALLINT  | Yes  | true  | NULL     |
|           |               | k1               | BIGINT    | Yes  | false | NULL     |
|           |               | k3               | LARGEINT  | Yes  | false | NULL     |
|           |               | k2               | BIGINT    | Yes  | false | NULL     |
+-----------+---------------+------------------+-----------+------+-------+----------+
```

可以看到当前 `mv_test` 表一共有三张物化视图：mv_1，mv_2 和 mv_3，以及他们的表结构。

删除物化视图

如果用户不再需要物化视图，则可以通过下面命令删除物化视图：

```
DROP MATERIALIZED VIEW
```

具体的语法可以通过下面命令查看：

```
HELP DROP MATERIALIZED VIEW
```

查看已创建的物化视图

用户可以通过命令查看已创建的物化视图的

具体的语法可查看：

```
SHOW CREATE MATERIALIZED VIEW
```

物化视图最佳实践

最佳实践1

使用物化视图一般分为以下几个步骤：

1. 创建物化视图。

2. 异步检查物化视图是否构建完成。

3. 查询并自动匹配物化视图。

第一步：创建物化视图。

假设用户有一张销售记录明细表，存储了每个交易的交易id，销售员，售卖门店，销售时间，以及金额。建表语句

为：

```
create table sales_records(record_id int, seller_id int, store_id int, sale_date da
```

这张 `sales_records` 的表结构如下：

```
MySQL [test]> desc sales_records;
+-----------+--------+------+-------+---------+-------+
| Field     | Type   | Null | Key   | Default | Extra |
+-----------+--------+------+-------+---------+-------+
| record_id | INT    | Yes  | true  | NULL    |       |
| seller_id | INT    | Yes  | true  | NULL    |       |
| store_id  | INT    | Yes  | true  | NULL    |       |
| sale_date | DATE   | Yes  | false | NULL    | NONE  |
| sale_amt  | BIGINT | Yes  | false | NULL    | NONE  |
+-----------+--------+------+-------+---------+-------+
```

这时候如果用户经常对不同门店的销售量进行一个分析查询，则可以给这个 `sales_records` 表创建一张以售卖门店分组，对相同售卖门店的销售额求和的一个物化视图。创建语句如下：

```
MySQL [test]> create materialized view store_amt as select store_id, sum(sale_amt)
```

后端返回下图，则说明创建物化视图任务提交成功。

```
Query OK, 0 rows affected (0.012 sec)
```

第二步：检查物化视图是否构建完成。

由于创建物化视图是一个异步的操作，用户在提交完创建物化视图任务后，需要异步的通过命令检查物化视图是否构建完成。命令如下：

```
SHOW ALTER TABLE ROLLUP FROM db_name; (Version 0.12)
SHOW ALTER TABLE MATERIALIZED VIEW FROM db_name; (Version 0.13)
```

这个命令中 `db_name` 是一个参数，您需要替换成自己真实的 db 名称。命令的结果是显示这个 db 的所有创建物化视图的任务。结果如下：

```
+-------+---------------+---------------------+---------------------+--------------
| JobId | TableName     | CreateTime          | FinishedTime        | BaseIndexName
+-------+---------------+---------------------+---------------------+--------------
| 22036 | sales_records | 2020-07-30 20:04:28 | 2020-07-30 20:04:57 | sales_records
+-------+---------------+---------------------+---------------------+--------------
```

其中 TableName 指的是物化视图的数据来自于哪个表，RollupIndexName 指的是物化视图的名称叫什么。其中比较重要的指标是 State。

当创建物化视图任务的 State 已经变成 FINISHED 后，就说明这个物化视图已经创建成功了。这就意味着，查询的时候有可能自动匹配到这张物化视图了。

第三步：查询。

当创建完成物化视图后，用户再查询不同门店的销售量时，就会直接从刚才创建的物化视图 `store_amt` 中读取聚合好的数据。达到提升查询效率的效果。

用户的查询依旧指定查询 `sales_records` 表，例如：

```
SELECT store_id, sum(sale_amt) FROM sales_records GROUP BY store_id;
```

上面查询就能自动匹配到 `store_amt` 。用户可以通过下面命令，检验当前查询是否匹配到了合适的物化视图。

```
EXPLAIN SELECT store_id, sum(sale_amt) FROM sales_records GROUP BY store_id;
+------------------------------------------------------------------------------+
| Explain String                                                               |
+------------------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                              |
|  OUTPUT EXPRS:<slot 2> `store_id` | <slot 3> sum(`sale_amt`)                 |
|   PARTITION: UNPARTITIONED                                                   |
|                                                                              |
|   RESULT SINK                                                                |
|                                                                              |
|   4:EXCHANGE                                                                 |
|                                                                              |
| PLAN FRAGMENT 1                                                              |
|  OUTPUT EXPRS:                                                               |
|   PARTITION: HASH_PARTITIONED: <slot 2> `store_id`                           |
|                                                                              |
|   STREAM DATA SINK                                                           |
|     EXCHANGE ID: 04                                                          |
|     UNPARTITIONED                                                            |
|                                                                              |
|   3:AGGREGATE (merge finalize)                                               |
|   |  output: sum(<slot 3> sum(`sale_amt`))                                   |
|   |  group by: <slot 2> `store_id`                                           |
|   |                                                                          |
|   2:EXCHANGE                                                                 |
|                                                                              |
| PLAN FRAGMENT 2                                                              |
|  OUTPUT EXPRS:                                                               |
|   PARTITION: RANDOM                                                          |
|                                                                              |
|   STREAM DATA SINK                                                           |
|     EXCHANGE ID: 02                                                          |
|     HASH_PARTITIONED: <slot 2> `store_id`                                    |
|                                                                              |
|   1:AGGREGATE (update serialize)                                             |
|   |  STREAMING                                                               |
|   |  output: sum(`sale_amt`)                                                 |
```

```
|   |  group by: `store_id`                                                                |
|   |                                                                                      |
|   0:OlapScanNode                                                                       |
|      TABLE: sales_records                                                              |
|      PREAGGREGATION: ON                                                                |
|      partitions=1/1                                                                    |
|      rollup: store_amt                                                                 |
|      tabletRatio=10/10                                                                 |
|      tabletList=22038,22040,22042,22044,22046,22048,22050,22052,22054,22056 |
|      cardinality=0                                                                     |
|      avgRowSize=0.0                                                                    |
|      numNodes=1                                                                        |
+----------------------------------------------------------------------------------------+
45 rows in set (0.006 sec)
```

其中最重要的就是 OlapScanNode 中的 rollup 属性。可以看到当前查询的 rollup 显示的是 `store_amt` 。也就是说查询已经正确匹配到物化视图 `store_amt` ，并直接从物化视图中读取数据了。

最佳实践2 PV，UV

业务场景：计算广告的 UV，PV 。

假设用户的原始广告点击数据存储在 Doris，那么针对广告 PV，UV 查询就可以通过创建 `bitmap_union` 的物化视图来提升查询速度。

通过下面语句首先创建一个存储广告点击数据明细的表，包含每条点击的点击事件，点击的是什么广告，通过什么渠道点击，以及点击的用户是谁。

```
MySQL [test]> create table advertiser_view_record(time date, advertiser varchar(10)
Query O
K, 0 rows affected (0.014 sec)
```

原始的广告点击数据表结构为：

```
MySQL [test]> desc advertiser_view_record;
+------------+-------------+------+-------+---------+-------+
| Field      | Type        | Null | Key   | Default | Extra |
+------------+-------------+------+-------+---------+-------+
| time       | DATE        | Yes  | true  | NULL    |       |
| advertiser | VARCHAR(10) | Yes  | true  | NULL    |       |
| channel    | VARCHAR(10) | Yes  | false | NULL    | NONE  |
| user_id    | INT         | Yes  | false | NULL    | NONE  |
+------------+-------------+------+-------+---------+-------+
4 rows in set (0.001 sec)
```

1. 创建物化视图。

由于用户想要查询的是广告的 UV 值，也就是需要对相同广告的用户进行一个精确去重，则查询一般为：

```
SELECT advertiser, channel, count(distinct user_id) FROM advertiser_view_record GRO
```

针对这种求 UV 的场景，我们就可以创建一个带 `bitmap_union` 的物化视图从而达到一个预先精确去重的效果。
在 Doris 中， `count(distinct)` 聚合的结果和 `bitmap_union_count` 聚合的结果是完全一致的。
而 `bitmap_union_count` 等于 `bitmap_union` 的结果求 count，所以如果查询中**涉及到 `count(distinct)` 则通过创建带 `bitmap_union` 聚合的物化视图方可加快查询**。
针对这个 case，则可以创建一个根据广告和渠道分组，对 `user_id` 进行精确去重的物化视图。

```
MySQL [test]> create materialized view advertiser_uv as select advertiser, channel,
Query OK, 0 rows affected (0.012 sec)
```

注意

因为本身 user_id 是一个 INT 类型，所以在 Doris 中需要先将字段通过函数 `to_bitmap` 转换为 bitmap 类型然后才可以进行 `bitmap_union` 聚合。
创建完成后，广告点击明细表和物化视图表的表结构如下：

```
MySQL [test]> desc advertiser_view_record all;
+------------------------+---------------+----------------------+-------------+----
| IndexName              | IndexKeysType | Field                | Type        | Nul
+------------------------+---------------+----------------------+-------------+----
| advertiser_view_record | DUP_KEYS      | time                 | DATE        | Yes
|                        |               | advertiser           | VARCHAR(10) | Yes
|                        |               | channel              | VARCHAR(10) | Yes
|                        |               | user_id              | INT         | Yes
|                        |               |                      |             |
| advertiser_uv          | AGG_KEYS      | advertiser           | VARCHAR(10) | Yes
|                        |               | channel              | VARCHAR(10) | Yes
|                        |               | to_bitmap(`user_id`) | BITMAP      | No
+------------------------+---------------+----------------------+-------------+----
```

2. 查询自动匹配。

当物化视图表创建完成后，查询广告 UV 时，Doris就会自动从刚才创建好的物化视图 `advertiser_uv` 中查询数据。例如原始的查询语句如下：

```
SELECT advertiser, channel, count(distinct user_id) FROM advertiser_view_record GRO
```

在选中物化视图后，实际的查询会转化为：

```
SELECT advertiser, channel, bitmap_union_count(to_bitmap(user_id)) FROM advertiser_
```

通过 EXPLAIN 命令可以检验到 Doris 是否匹配到了物化视图：

```
MySQL [test]> explain SELECT advertiser, channel, count(distinct user_id) FROM  adv
+-----------------------------------------------------------------------------------
```

```
| Explain String
+----------------------------------------------------------------------------------
| PLAN FRAGMENT 0
|  OUTPUT EXPRS:<slot 7> `advertiser` | <slot 8> `channel` | <slot 9> bitmap_union_
|   PARTITION: UNPARTITIONED
|
|   RESULT SINK
|
|   4:EXCHANGE
|
| PLAN FRAGMENT 1
|  OUTPUT EXPRS:
|   PARTITION: HASH_PARTITIONED: <slot 4> `advertiser`, <slot 5> `channel`
|
|   STREAM DATA SINK
|     EXCHANGE ID: 04
|     UNPARTITIONED
|
|   3:AGGREGATE (merge finalize)
|   |  output: bitmap_union_count(<slot 6> bitmap_union_count(`default_cluster:test
|   |  group by: <slot 4> `advertiser`, <slot 5> `channel`
|   |
|   2:EXCHANGE
|
| PLAN FRAGMENT 2
|  OUTPUT EXPRS:
|   PARTITION: RANDOM
|
|   STREAM DATA SINK
|     EXCHANGE ID: 02
|     HASH_PARTITIONED: <slot 4> `advertiser`, <slot 5> `channel`
|
|   1:AGGREGATE (update serialize)
|   |  STREAMING
|   |  output: bitmap_union_count(`default_cluster:test`.`advertiser_view_record`.`
|   |  group by: `advertiser`, `channel`
|   |
|   0:OlapScanNode
|      TABLE: advertiser_view_record
|      PREAGGREGATION: ON
|      partitions=1/1
|      rollup: advertiser_uv
|      tabletRatio=10/10
|      tabletList=22084,22086,22088,22090,22092,22094,22096,22098,22100,22102
|      cardinality=0
|      avgRowSize=0.0
|      numNodes=1
```

```
+-------------------------------------------------------------------------------------
45 rows in set (0.030 sec)
```

在 EXPLAIN 的结果中，首先可以看到 OlapScanNode 的 rollup 属性值为 advertiser_uv。也就是说，查询会直接扫描物化视图的数据。说明匹配成功。

其次对于 `user_id` 字段求 `count(distinct)` 被改写为求 `bitmap_union_count(to_bitmap)` 。也就是通过 bitmap 的方式来达到精确去重的效果。

最佳实践3

业务场景：匹配更丰富的前缀索引。

用户的原始表有 （k1, k2, k3） 三列。其中 k1, k2 为前缀索引列。这时候如果用户查询条件中包含 `where k1=1 and k2=2` 就能通过索引加速查询。

但是有些情况下，用户的过滤条件无法匹配到前缀索引，例如 `where k3=3` 。则无法通过索引提升查询速度。创建以 k3 作为第一列的物化视图就可以解决这个问题。

1. 创建物化视图。

```
CREATE MATERIALIZED VIEW mv_1 as SELECT k3, k2, k1 FROM tableA ORDER BY k3;
```

通过上面语法创建完成后，物化视图中既保留了完整的明细数据，且物化视图的前缀索引为 k3 列。表结构如下：

```
MySQL [test]> desc tableA all;
+-----------+---------------+-------+------+------+-------+---------+-------+
| IndexName | IndexKeysType | Field | Type | Null | Key   | Default | Extra |
+-----------+---------------+-------+------+------+-------+---------+-------+
| tableA    | DUP_KEYS      | k1    | INT  | Yes  | true  | NULL    |       |
|           |               | k2    | INT  | Yes  | true  | NULL    |       |
|           |               | k3    | INT  | Yes  | true  | NULL    |       |
|           |               |       |      |      |       |         |       |
| mv_1      | DUP_KEYS      | k3    | INT  | Yes  | true  | NULL    |       |
|           |               | k2    | INT  | Yes  | false | NULL    | NONE  |
|           |               | k1    | INT  | Yes  | false | NULL    | NONE  |
+-----------+---------------+-------+------+------+-------+---------+-------+
```

2. 查询匹配。

这时候如果用户的查询存在 k3 列的过滤条件时，例如：

```
select k1, k2, k3 from table A where k3=3;
```

这时候查询就会直接从刚才创建的 mv_1 物化视图中读取数据。物化视图对 k3 是存在前缀索引的，查询效率也会提升。

物化视图的局限性

1. 物化视图的聚合函数的参数不支持表达式仅支持单列，例如： sum(a+b)不支持。
2. 如果删除语句的条件列，在物化视图中不存在，则不能进行删除操作。如果一定要删除数据，则需要先将物化视图删除，然后方可删除数据。
3. 单表上过多的物化视图会影响导入的效率：导入数据时，物化视图和 base 表数据是同步更新的，如果一张表的物化视图表超过10张，则有可能导致导入速度很慢。这就像单次导入需要同时导入10张表数据是一样的。
4. 相同列，不同聚合函数，不能同时出现在一张物化视图中，例如：select sum(a)，min(a) from table 不支持。
5. 物化视图针对 Unique Key 数据模型，只能改变列顺序，不能起到聚合的作用，所以在 Unique Key 模型上不能通过创建物化视图的方式对数据进行粗粒度聚合操作。

异常错误

1. DATA_QUALITY_ERR: "The data quality does not satisfy, please check your data"
由于数据质量问题导致物化视图创建失败。

注意

bitmap 类型仅支持正整型，如果原始数据中存在负数，会导致物化视图创建失败。

缓存表或分区到内存

最近更新时间：2024-06-27 11:11:17

Doris 支持把一张表的所有数据或指定分区的数据缓存在内存中，内存表可以提高查询计算性能。但内存容量毕竟有限，因此最好仅使用数据总量较小的表。

用法

缓存整张表

对于新增表，建表时在 `PROPERTIES` 中加上配置 `"in_memory"="true"` 即可，如：

```
CREATE TABLE IF NOT EXISTS example_db.expamle_tbl
(
    `user_id` LARGEINT NOT NULL COMMENT "用户id",
    `date` DATE NOT NULL COMMENT "数据灌入日期时间",
    `city` VARCHAR(20) COMMENT "用户所在城市",
    `age` SMALLINT COMMENT "用户年龄",
    `sex` TINYINT COMMENT "用户性别",
    `last_visit_date` DATETIME REPLACE DEFAULT "1970-01-01 00:00:00" COMMENT "用户最
    `cost` BIGINT SUM DEFAULT "0" COMMENT "用户总消费",
    `max_dwell_time` INT MAX DEFAULT "0" COMMENT "用户最大停留时间",
    `min_dwell_time` INT MIN DEFAULT "99999" COMMENT "用户最小停留时间"
)
AGGREGATE KEY(`user_id`, `date`, `city`, `age`, `sex`)
DISTRIBUTED BY HASH(`user_id`) BUCKETS 1
PROPERTIES (
    "replication_allocation" = "tag.location.default: 1",
    "in_memory"="true"
);
```

对于存量表，修改表配置设置 `"in_memory"="true"` 即可：

```
ALTER TABLE example_db.my_table set ("in_memory" = "true");
```

缓存指定分区

正式分区和临时分区都可缓存。

对于新增的分区，添加分区时设置 `"in_memory"="true"` 即可：

```
ALTER TABLE example_db.my_table ADD [TEMPORARY] PARTITION p1 VALUES LESS THAN ("202
```

对于已存在的分区，修改分区配置设置 `"in_memory"="true"` 即可：

```
ALTER TABLE example_db.my_table MODIFY PARTITION (p1, p2, p4) SET("in_memory"="true
```

去除数据缓存

当不需要缓存数据用于加速计算时需要及时把缓存数据从内存释放掉，减少不必要的数据释放。去除缓存只需修改相应表或分区的配置改为 `"in_memory"="false"` 即可。

```
ALTER TABLE example_db.my_table set ("in_memory" = "false");
```

```
ALTER TABLE example_db.my_table MODIFY PARTITION (p1, p2, p4) SET("in_memory"="fals
```

注意事项

in_memory 属性

当建表时指定了 `"in_memory" = "true"` 属性。则 Doris 会尽量将该表的数据块缓存在存储引擎的 PageCache 中，以减少磁盘 IO。但这个属性不会保证数据块常驻在内存中，仅作为一种尽力而为的标识。

相关配置参数

内存表 缓存内存的方式是使用 Doris 的 page cache，需要确保 page cache 功能开启和分配了合适的内存。相关的参数通过 be.conf 进行设置。

disable_storage_page_cache

类型：bool

描述：是否使用 page cache 进行 index 的缓存，该配置仅在 BETA 存储格式时生效。

默认值：false

chunk_reserved_bytes_limit

描述：Chunk Allocator的reserved bytes 限制，通常被设置为 mem_limit 的百分比。默认单位字节，值必须是2的倍数，且必须大于0，如果大于物理内存，将被设置为物理内存大小。增加这个变量可以提高性能，但是会获得更多其他模块无法使用的空闲内存。

默认值：10%

最佳实践

尝试比较开启内存表前后的性能差距。

环境准备

测试版本 ：1.2.6

FE节点配置：4核16G 1个节点 硬盘400GB

BE节点配置：4核16G 3个节点 硬盘300GB

配置：chunk_reserved_bytes_limit=30%

开展测试

使用 TPC-H 测试集的 supplier 表测试，其中 supplier 表为内存表，supplier1 表为非内存表。

```
| supplier | CREATE TABLE `supplier` (
  `s_suppkey` int(11) NOT NULL,
  `s_name` varchar(25) NOT NULL,
  `s_address` varchar(40) NOT NULL,
  `s_nationkey` int(11) NOT NULL,
  `s_phone` varchar(15) NOT NULL,
  `s_acctbal` decimalv3(15, 2) NOT NULL,
  `s_comment` varchar(101) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(`s_suppkey`)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(`s_suppkey`) BUCKETS 12
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "true",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
); |
1 row in set (0.01 sec)

MySQL [tpch_100_d]> show create table supplier1;
| Table     | Create Table
                                          |
| supplier1 | CREATE TABLE `supplier1` (
  `s_suppkey` int(11) NOT NULL,
  `s_name` varchar(25) NOT NULL,
  `s_address` varchar(40) NOT NULL,
  `s_nationkey` int(11) NOT NULL,
  `s_phone` varchar(15) NOT NULL,
  `s_acctbal` decimalv3(15, 2) NOT NULL,
  `s_comment` varchar(101) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(`s_suppkey`)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(`s_suppkey`) BUCKETS 12
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
); |
1 row in set (0.00 sec)
```

内存表

非内存表

大小均为250M，共100W条记录。

```
MySQL [tpch_100_d]> show data;
+-----------+-------------+--------------+
| TableName | Size        | ReplicaCount |
+-----------+-------------+--------------+
| customer  | 3.860 GB    | 72           |
| lineitem  | 55.385 GB   | 288          |
| nation    | 7.714 KB    | 3            |
| orders    | 17.999 GB   | 288          |
| part      | 2.092 GB    | 72           |
| partsupp  | 12.665 GB   | 72           |
| region    | 3.270 KB    | 3            |
| supplier  | 250.393 MB  | 36           |
| supplier1 | 250.393 MB  | 36           |
| Total     | 92.490 GB   | 870          |
| Quota     | 1024.000 TB | 1073741824   |
| Left      | 1023.910 TB | 1073740954   |
+-----------+-------------+--------------+
```

使用 mysqlslap 进行压测，压测 SQL 为针对 supplier 的全表遍历。

```
[root@9 data]# tail test.sql
select * from supplier where s_suppkey=1;
...
select * from supplier where s_suppkey=999991;
select * from supplier where s_suppkey=999992;
select * from supplier where s_suppkey=999993;
select * from supplier where s_suppkey=999994;
select * from supplier where s_suppkey=999995;
select * from supplier where s_suppkey=999996;
select * from supplier where s_suppkey=999997;
select * from supplier where s_suppkey=999998;
select * from supplier where s_suppkey=999999;
[root@9 data]# tail test2.sql
select * from supplier1 where s_suppkey=1;
...
select * from supplier1 where s_suppkey=999991;
select * from supplier1 where s_suppkey=999992;
select * from supplier1 where s_suppkey=999993;
select * from supplier1 where s_suppkey=999994;
select * from supplier1 where s_suppkey=999995;
select * from supplier1 where s_suppkey=999996;
select * from supplier1 where s_suppkey=999997;
select * from supplier1 where s_suppkey=999998;
select * from supplier1 where s_suppkey=999999;
```

压测 SQL：

```
mysqlslap -h127.0.0.1 -uadmin -P9030 -pxxxxx --iterations=1 --concurrency=500 --num
```

测试结果

内存表 平均耗时160s

```
^C
[root@9 data]# mysqslap -h127.0.0.1 -uadmin -P9030 -pAbc123456 --iterations=1 --concurrency=500 --number-of-queries=1000000 --create-sch
Benchmark
        Average number of seconds to run all queries: 177.389 seconds
        Minimum number of seconds to run all queries: 177.389 seconds
        Maximum number of seconds to run all queries: 177.389 seconds
        Number of clients running queries: 500
        Average number of queries per client: 2000

[root@9 data]# mysqslap -h127.0.0.1 -uadmin -P9030 -pAbc123456 --iterations=1 --concurrency=500 --number-of-queries=1000000 --create-sch
Benchmark
        Average number of seconds to run all queries: 162.130 seconds
        Minimum number of seconds to run all queries: 162.130 seconds
        Maximum number of seconds to run all queries: 162.130 seconds
        Number of clients running queries: 500
        Average number of queries per client: 2000

[root@9 data]# mysqslap -h127.0.0.1 -uadmin -P9030 -pAbc123456 --iterations=1 --concurrency=500 --number-of-queries=1000000 --create-sch
Benchmark
        Average number of seconds to run all queries: 145.209 seconds
        Minimum number of seconds to run all queries: 145.209 seconds
        Maximum number of seconds to run all queries: 145.209 seconds
        Number of clients running queries: 500
        Average number of queries per client: 2000

[root@9 data]# mysqslap -h127.0.0.1 -uadmin -P9030 -pAbc123456 --iterations=1 --concurrency=500 --number-of-queries=1000000 --create-sch
Benchmark
        Average number of seconds to run all queries: 161.736 seconds
        Minimum number of seconds to run all queries: 161.736 seconds
        Maximum number of seconds to run all queries: 161.736 seconds
        Number of clients running queries: 500
        Average number of queries per client: 2000
```

非内存表 平均耗时260s

```
[root@9 data]# mysqslap -h127.0.0.1 -uadmin -P9030 -pAbc123456 --iterations=1 --concurrency=500 --number-of-queries=1000000 --create-sch
Benchmark
        Average number of seconds to run all queries: 260.685 seconds
        Minimum number of seconds to run all queries: 260.685 seconds
        Maximum number of seconds to run all queries: 260.685 seconds
        Number of clients running queries: 500
        Average number of queries per client: 2000

[root@9 data]# mysqslap -h127.0.0.1 -uadmin -P9030 -pAbc123456 --iterations=1 --concurrency=500 --number-of-queries=1000000 --create-sch
Benchmark
        Average number of seconds to run all queries: 262.500 seconds
        Minimum number of seconds to run all queries: 262.500 seconds
        Maximum number of seconds to run all queries: 262.500 seconds
        Number of clients running queries: 500
        Average number of queries per client: 2000
```

内存表相对非内存表提升38%。

使用建议

配置内存表的数据不宜过大，建议在300M以下。

主要考虑到 page cache 是全局共用的缓存，单表数据过大会挤占其他 sql page cache 的空间。同时内存表也并不保证表的所有 page 都被加载进内存，只是告诉系统尽力而为的一个标志， 也容易跟其他表数据 page cache 相互挤占，导致 page cache 的内存停驻时间变少，而降低了page cache 的命中率。

Colocation Join

最近更新时间：2024-06-27 11:11:33

Colocation Join 旨在为某些 Join 查询提供本地性优化，来减少数据在节点间的传输耗时，加速查询。最初的设计、实现和效果可以参阅 ISSUE 245。
Colocation Join 功能经过一次改版，设计和使用方式和最初设计稍有不同。本文档主要介绍 Colocation Join 的原理、实现、使用方式和注意事项。

名词解释

FE：Frontend，Doris 的前端节点。负责元数据管理和请求接入。
BE：Backend，Doris 的后端节点。负责查询执行和数据存储。
Colocation Group（CG）：一个 CG 中会包含一张及以上的 Table。在同一个 Group 内的 Table 有着相同的 Colocation Group Schema，并且有着相同的数据分片分布。
Colocation Group Schema（CGS）：用于描述一个 CG 中的 Table，和 Colocation 相关的通用 Schema 信息。包括分桶列类型，分桶数以及副本数等。

原理

Colocation Join 功能，是将一组拥有相同 CGS 的 Table 组成一个 CG。并保证这些 Table 对应的数据分片会落在同一个 BE 节点上。使得当 CG 内的表进行分桶列上的 Join 操作时，可以通过直接进行本地数据 Join，减少数据在节点间的传输耗时。
一个表的数据，最终会根据分桶列值 Hash、对桶数取模后落在某一个分桶内。假设一个 Table 的分桶数为 8，则共有 `[0, 1, 2, 3, 4, 5, 6, 7]` 8 个分桶（Bucket），我们称这样一个序列为一个 `BucketsSequence`。每个 Bucket 内会有一个或多个数据分片（Tablet）。当表为单分区表时，一个 Bucket 内仅有一个 Tablet。如果是多分区表，则会有多个。
为了使得 Table 能够有相同的数据分布，同一 CG 内的 Table 必须保证以下属性相同：
1. 分桶列和分桶数。
分桶列，即在建表语句中 `DISTRIBUTED BY HASH(col1, col2, ...)` 中指定的列。分桶列决定了一张表的数据通过哪些列的值进行 Hash 划分到不同的 Tablet 中。同一 CG 内的 Table 必须保证分桶列的类型和数量完全一致，并且桶数一致，才能保证多张表的数据分片能够一一对应的进行分布控制。
2. 副本数。
同一个 CG 内所有表的所有分区（Partition）的副本数必须一致。如果不一致，可能出现某一个 Tablet 的某一个副本，在同一个 BE 上没有其他的表分片的副本对应。
同一个 CG 内的表，分区的个数、范围以及分区列的类型不要求一致。

在固定了分桶列和分桶数后，同一个 CG 内的表会拥有相同的 BucketsSequence。而副本数决定了每个分桶内的 Tablet 的多个副本，存放在哪些 BE 上。假设 BucketsSequence 为 `[0, 1, 2, 3, 4, 5, 6, 7]` ，BE 节点有 `[A, B, C, D]` 4个。则一个可能的数据分布如下：

```
+---+ +---+ +---+ +---+ +---+ +---+ +---+ +---+
| 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 |
+---+ +---+ +---+ +---+ +---+ +---+ +---+ +---+
| A | | B | | C | | D | | A | | B | | C | | D |
|   | |   | |   | |   | |   | |   | |   | |   |
| B | | C | | D | | A | | B | | C | | D | | A |
|   | |   | |   | |   | |   | |   | |   | |   |
| C | | D | | A | | B | | C | | D | | A | | B |
+---+ +---+ +---+ +---+ +---+ +---+ +---+ +---+
```

CG 内所有表的数据都会按照上面的规则进行统一分布，这样就保证了，分桶列值相同的数据都在同一个 BE 节点上，可以进行本地数据 Join。

使用方式

建表

建表时，可以在 `PROPERTIES` 中指定属性 `"colocate_with" = "group_name"` ，表示这个表是一个 Colocation Join 表，并且归属于一个指定的 Colocation Group。

示例：

```
CREATE TABLE tbl (k1 int, v1 int sum)
DISTRIBUTED BY HASH(k1)
BUCKETS 8
PROPERTIES(
    "colocate_with" = "group1"
);
```

如果指定的 Group 不存在，则 Doris 会自动创建一个只包含当前这张表的 Group。如果 Group 已存在，则 Doris 会检查当前表是否满足 Colocation Group Schema。如果满足，则会创建该表，并将该表加入 Group。同时，表会根据已存在的 Group 中的数据分布规则创建分片和副本。

Group 归属于一个 Database，Group 的名字在一个 Database 内唯一。在内部存储是 Group 的全名为 `dbId_groupName` ，但用户只感知 groupName。

删表

当 Group 中最后一张表彻底删除后（彻底删除是指从回收站中删除。通常，一张表通过 `DROP TABLE` 命令删除后，会在回收站默认停留一天的时间后，再删除），该 Group 也会被自动删除。

查看 Group

以下命令可以查看集群内已存在的 Group 信息。

```
SHOW PROC '/colocation_group';

+-------------+--------------+--------------+------------+----------------+--------
| GroupId     | GroupName    | TableIds     | BucketsNum | ReplicationNum | DistCol
+-------------+--------------+--------------+------------+----------------+--------
| 10005.10008 | 10005_group1 | 10007, 10040 | 10         | 3              | int(11)
+-------------+--------------+--------------+------------+----------------+--------
```

GroupId： 一个 Group 的全集群唯一标识，前半部分为 db id，后半部分为 group id。

GroupName： Group 的全名。

TabletIds： 该 Group 包含的 Table 的 id 列表。

BucketsNum： 分桶数。

ReplicationNum： 副本数。

DistCols： Distribution columns，即分桶列类型。

IsStable： 该 Group 是否稳定（稳定的定义，见 **Colocation 副本均衡和修复**一节）。

通过以下命令可以进一步查看一个 Group 的数据分布情况：

```
SHOW PROC '/colocation_group/10005.10008';

+-------------+---------------------+
| BucketIndex | BackendIds          |
+-------------+---------------------+
| 0           | 10004, 10002, 10001 |
| 1           | 10003, 10002, 10004 |
| 2           | 10002, 10004, 10001 |
| 3           | 10003, 10002, 10004 |
| 4           | 10002, 10004, 10003 |
| 5           | 10003, 10002, 10001 |
| 6           | 10003, 10004, 10001 |
| 7           | 10003, 10004, 10002 |
+-------------+---------------------+
```

BucketIndex：分桶序列的下标。

BackendIds：分桶中数据分片所在的 BE 节点 id 列表。

注意

以上命令需要 ADMIN 权限。暂不支持普通用户查看。

使用限制

要使用colocation join，需要满足以下条件：

参与colocation join的表，要配置开启Colocation Group属性，并属于同一个Colocation Group。

同一 CG 内的 Table 必须保证以下属性相同：
分桶列和分桶数
副本数

修改表 Colocate Group 属性

可以对一个已经创建的表，修改其 Colocation Group 属性。示例：`ALTER TABLE tbl SET ("colocate_with" = "group2");` 。
如果该表之前没有指定过 Group，则该命令检查 Schema，并将该表加入到该 Group（Group 不存在则会创建）。
如果该表之前有指定其他 Group，则该命令会先将该表从原有 Group 中移除，并加入新 Group（Group 不存在则会创建）。
也可以通过以下命令，删除一个表的 Colocation 属性：`ALTER TABLE tbl SET ("colocate_with" = "");` 。

其他相关操作

当对一个具有 Colocation 属性的表进行增加分区（ADD PARTITION）、修改副本数时，Doris 会检查修改是否会违反 Colocation Group Schema，如果违反则会拒绝。

Colocation 副本均衡和修复

Colocation 表的副本分布需要遵循 Group 中指定的分布，所以在副本修复和均衡方面和普通分片有所区别。
Group 自身有一个 Stable 属性，当 Stable 为 true 时，表示当前 Group 内的表的所有分片没有正在进行变动，Colocation 特性可以正常使用。当 Stable 为 false 时（Unstable），表示当前 Group 内有部分表的分片正在做修复或迁移，此时，相关表的 Colocation Join 将退化为普通 Join。

副本修复

副本只能存储在指定的 BE 节点上。所以当某个 BE 不可用时（宕机、Decommission 等），需要寻找一个新的 BE 进行替换。Doris 会优先寻找负载最低的 BE 进行替换。替换后，该 Bucket 内的所有在旧 BE 上的数据分片都要做修复。迁移过程中，Group 被标记为 Unstable。

副本均衡

Doris 会尽力将 Colocation 表的分片均匀分布在所有 BE 节点上。对于普通表的副本均衡，是以单副本为粒度的，即单独为每一个副本寻找负载较低的 BE 节点即可。而 Colocation 表的均衡是 Bucket 级别的，即一个 Bucket 内的所有副本都会一起迁移。我们采用一个简单的均衡算法，即在不考虑副本实际大小，而只根据副本数量，将 BucketsSequence 均匀的分布在所有 BE 上。具体算法可以参阅 `ColocateTableBalancer.java` 中的代码注释。
注意

当前的 Colocation 副本均衡和修复算法，对于异构部署的 Doris 集群效果可能不佳。所谓异构部署，即 BE 节点的磁盘容量、数量、磁盘类型（SSD 和 HDD）不一致。在异构部署情况下，可能出现小容量的 BE 节点和大容量的 BE 节点存储了相同的副本数量。

当一个 Group 处于 Unstable 状态时，其中的表的 Join 将退化为普通 Join。此时可能会极大降低集群的查询性能。

如果不希望系统自动均衡，可以设置 FE 的配置项 `disable_colocate_balance` 来禁止自动均衡。然后在合适的时间打开即可。（具体参阅**高级操作**一节）。

查询

对 Colocation 表的查询方式和普通表一样，用户无需感知 Colocation 属性。如果 Colocation 表所在的 Group 处于 Unstable 状态，将自动退化为普通 Join。

举例说明：

表1：

```
CREATE TABLE `tbl1` (
    `k1` date NOT NULL COMMENT "",
    `k2` int(11) NOT NULL COMMENT "",
    `v1` int(11) SUM NOT NULL COMMENT ""
) ENGINE=OLAP
AGGREGATE KEY(`k1`, `k2`)
PARTITION BY RANGE(`k1`)
(
    PARTITION p1 VALUES LESS THAN ('2019-05-31'),
    PARTITION p2 VALUES LESS THAN ('2019-06-30')
)
DISTRIBUTED BY HASH(`k2`) BUCKETS 8
PROPERTIES (
    "colocate_with" = "group1"
);
```

表2：

```
CREATE TABLE `tbl2` (
    `k1` datetime NOT NULL COMMENT "",
    `k2` int(11) NOT NULL COMMENT "",
    `v1` double SUM NOT NULL COMMENT ""
) ENGINE=OLAP
AGGREGATE KEY(`k1`, `k2`)
DISTRIBUTED BY HASH(`k2`) BUCKETS 8
PROPERTIES (
    "colocate_with" = "group1"
);
```

查看查询计划：

```
DESC SELECT * FROM tbl1 INNER JOIN tbl2 ON (tbl1.k2 = tbl2.k2);

+----------------------------------------------------+
| Explain String                                     |
+----------------------------------------------------+
| PLAN FRAGMENT 0                                    |
|  OUTPUT EXPRS:`tbl1`.`k1` |                        |
|   PARTITION: RANDOM                                |
|                                                    |
|   RESULT SINK                                      |
|                                                    |
|   2:HASH JOIN                                      |
|   |  join op: INNER JOIN                           |
|   |  hash predicates:                              |
|   |  colocate: true                                |
|   |    `tbl1`.`k2` = `tbl2`.`k2`                   |
|   |  tuple ids: 0 1                                |
|   |                                                |
|   |----1:OlapScanNode                              |
|   |       TABLE: tbl2                              |
|   |       PREAGGREGATION: OFF. Reason: null        |
|   |       partitions=0/1                           |
|   |       rollup: null                             |
|   |       buckets=0/0                              |
|   |       cardinality=-1                           |
|   |       avgRowSize=0.0                           |
|   |       numNodes=0                               |
|   |       tuple ids: 1                             |
|   |                                                |
|   0:OlapScanNode                                   |
|      TABLE: tbl1                                   |
|      PREAGGREGATION: OFF. Reason: No AggregateInfo |
|      partitions=0/2                                |
|      rollup: null                                  |
|      buckets=0/0                                   |
|      cardinality=-1                                |
|      avgRowSize=0.0                                |
|      numNodes=0                                    |
|      tuple ids: 0                                  |
+----------------------------------------------------+
```

如果 Colocation Join 生效，则 Hash Join 节点会显示 `colocate: true` 。

如果没有生效，则查询计划如下：

```
+----------------------------------------------------+
```

```
| Explain String                                          |
+---------------------------------------------------------+
| PLAN FRAGMENT 0                                         |
|  OUTPUT EXPRS:`tbl1`.`k1` |                             |
|   PARTITION: RANDOM                                     |
|                                                         |
|   RESULT SINK                                           |
|                                                         |
|   2:HASH JOIN                                           |
|   |  join op: INNER JOIN (BROADCAST)                    |
|   |  hash predicates:                                   |
|   |  colocate: false, reason: group is not stable       |
|   |    `tbl1`.`k2` = `tbl2`.`k2`                        |
|   |  tuple ids: 0 1                                     |
|   |                                                     |
|   |----3:EXCHANGE                                       |
|   |       tuple ids: 1                                  |
|   |                                                     |
|   0:OlapScanNode                                        |
|      TABLE: tbl1                                        |
|      PREAGGREGATION: OFF. Reason: No AggregateInfo      |
|      partitions=0/2                                     |
|      rollup: null                                       |
|      buckets=0/0                                        |
|      cardinality=-1                                     |
|      avgRowSize=0.0                                     |
|      numNodes=0                                         |
|      tuple ids: 0                                       |
|                                                         |
| PLAN FRAGMENT 1                                         |
|  OUTPUT EXPRS:                                          |
|   PARTITION: RANDOM                                     |
|                                                         |
|   STREAM DATA SINK                                      |
|     EXCHANGE ID: 03                                     |
|     UNPARTITIONED                                       |
|                                                         |
|   1:OlapScanNode                                        |
|      TABLE: tbl2                                        |
|      PREAGGREGATION: OFF. Reason: null                  |
|      partitions=0/1                                     |
|      rollup: null                                       |
|      buckets=0/0                                        |
|      cardinality=-1                                     |
|      avgRowSize=0.0                                     |
|      numNodes=0                                         |
|      tuple ids: 1                                       |
```

```
+------------------------------------------------------+
```

HASH JOIN 节点会显示对应原因：`colocate: false, reason: group is not stable`。同时会有一个 EXCHANGE 节点生成。

高级操作

FE 配置项

disable_colocate_relocate

是否关闭 Doris 的自动 Colocation 副本修复。默认为 false，即不关闭。该参数只影响 Colocation 表的副本修复，不影响普通表。

disable_colocate_balance

是否关闭 Doris 的自动 Colocation 副本均衡。默认为 false，即不关闭。该参数只影响 Colocation 表的副本均衡，不影响普通表。

以上参数可以动态修改，设置方式请参阅 `HELP ADMIN SHOW CONFIG;` 和 `HELP ADMIN SET CONFIG;`。

disable_colocate_join

是否关闭 Colocation Join 功能。在 0.10 及之前的版本，默认为 true，即关闭。在之后的某个版本中将默认为 false，即开启。

use_new_tablet_scheduler

在 0.10 及之前的版本中，新的副本调度逻辑与 Colocation Join 功能不兼容，所以在 0.10 及之前版本，如果 `disable_colocate_join = false`，则需设置 `use_new_tablet_scheduler = false`，即关闭新的副本调度器。之后的版本中，`use_new_tablet_scheduler` 将均衡为 true。

HTTP Restful API

Doris 提供了几个和 Colocation Join 有关的 HTTP Restful API，用于查看和修改 Colocation Group。

该 API 实现在 FE 端，使用 `fe_host:fe_http_port` 进行访问。需要 ADMIN 权限。

1. 查看集群的全部 Colocation 信息。

```
GET /api/colocate

返回以 Json 格式表示内部 Colocation 信息。

{
     "msg": "success",
 "code": 0,
 "data": {
     "infos": [
         ["10003.12002", "10003_group1", "10037, 10043", "1", "1", "int(11)", "true
     ],
     "unstableGroupIds": [],
```

```
      "allGroupIds": [{
          "dbId": 10003,
          "grpId": 12002
      }]
  },
  "count": 0
}
```

2. 将 Group 标记为 Stable 或 Unstable。

标记为 Stable：

```
POST /api/colocate/group_stable?db_id=10005&group_id=10008

返回：200
```

标记为 Unstable：

```
DELETE /api/colocate/group_stable?db_id=10005&group_id=10008

返回：200
```

3. 设置 Group 的数据分布。

该接口可以强制设置某一 Group 的数分布。

```
POST /api/colocate/bucketseq?db_id=10005&group_id=10008

Body:
[[10004,10002],[10003,10002],[10002,10004],[10003,10002],[10002,10004],[10003,10002

返回 200
```

其中 Body 是以嵌套数组表示的 BucketsSequence 以及每个 Bucket 中分片分布所在 BE 的 id。

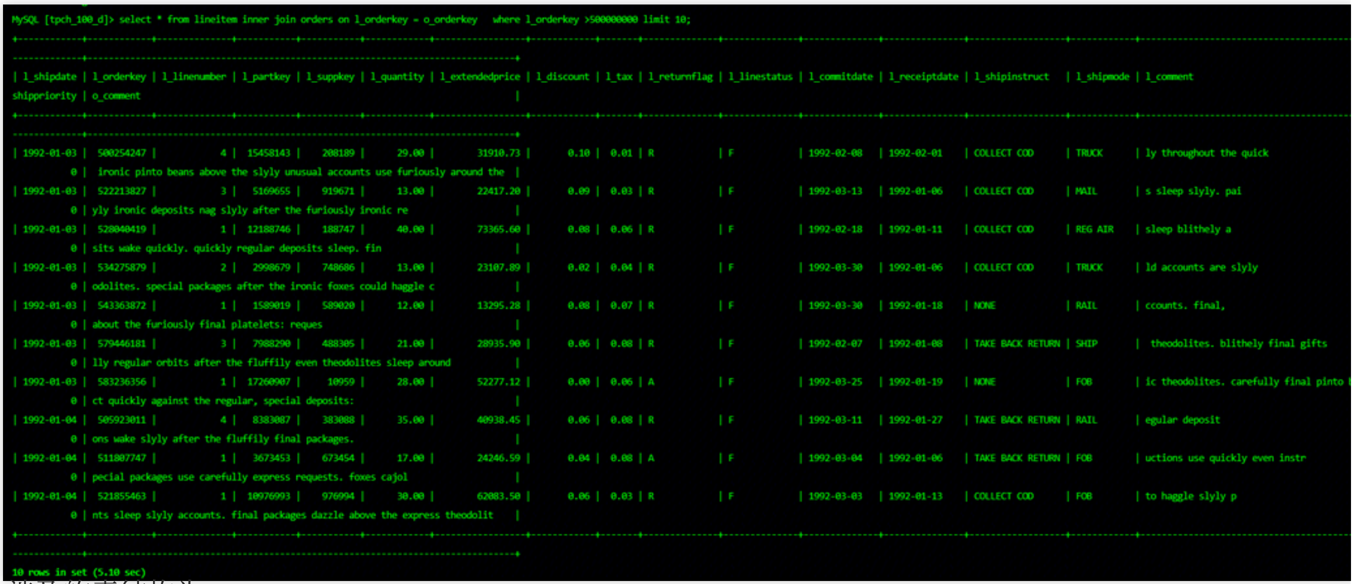
注意

使用该命令，可能需要将 FE 的配置 `disable_colocate_relocate` 和 `disable_colocate_balance` 设为 true。即关闭系统自动的 Colocation 副本修复和均衡。否则可能在修改后，会被系统自动重置。

最佳实践

这里基于 TPC-H 100G 数据集作为 case，进行业务模拟。有以下 SQL，耗时5s，预期进行 join 优化。

```
select * from lineitem inner join orders on l_orderkey = o_orderkey   where l_order
```



```
MySQL [tpch_100_d]> select * from lineitem inner join orders on l_orderkey = o_orderkey  where l_orderkey >500000000 limit 10;
| l_shipdate | l_orderkey | l_linenumber | l_partkey | l_suppkey | l_quantity | l_extendedprice | l_discount | l_tax | l_returnflag | l_linestatus | l_commitdate | l_receiptdate | l_shipinstruct | l_shipmode | l_comment
shippriority | o_comment |
| 1992-01-03 | 500254247 | 4 | 15458143 | 208189 | 29.00 | 31910.73 | 0.10 | 0.01 | R | F | 1992-02-08 | 1992-02-01 | COLLECT COD | TRUCK | ly throughout the quick
0 | ironic pinto beans above the slyly unusual accounts use furiously around the |
| 1992-01-03 | 522213827 | 3 | 5169655 | 919671 | 13.00 | 22417.20 | 0.09 | 0.03 | R | F | 1992-03-13 | 1992-01-06 | COLLECT COD | MAIL | s sleep slyly. pai
0 | yly ironic deposits nag slyly after the furiously ironic re |
| 1992-01-03 | 528040419 | 1 | 12188746 | 188747 | 40.00 | 73365.60 | 0.08 | 0.06 | R | F | 1992-02-18 | 1992-01-11 | COLLECT COD | REG AIR | sleep blithely a
0 | sits wake quickly. quickly regular deposits sleep. fin |
| 1992-01-03 | 534275879 | 2 | 2998679 | 748686 | 13.00 | 23107.89 | 0.02 | 0.04 | R | F | 1992-03-30 | 1992-01-06 | COLLECT COD | TRUCK | ld accounts are slyly
0 | odolites. special packages after the ironic foxes could haggle c |
| 1992-01-03 | 543363872 | 1 | 1589019 | 589020 | 12.00 | 13295.28 | 0.08 | 0.07 | R | F | 1992-03-30 | 1992-01-18 | NONE | RAIL | ccounts. final,
0 | about the furiously final platelets: reques |
| 1992-01-03 | 579446181 | 3 | 7988290 | 488305 | 21.00 | 28935.90 | 0.06 | 0.08 | R | F | 1992-02-07 | 1992-01-08 | TAKE BACK RETURN | SHIP |  theodolites. blithely final gifts
0 | lly regular orbits after the fluffily even theodolites sleep around |
| 1992-01-03 | 583236356 | 1 | 17260907 | 10959 | 28.00 | 52277.12 | 0.00 | 0.06 | A | F | 1992-03-25 | 1992-01-19 | NONE | FOB | ic theodolites. carefully final pinto
0 | ct quickly against the regular, special deposits: |
| 1992-01-04 | 505923011 | 4 | 8383087 | 383088 | 35.00 | 40938.45 | 0.06 | 0.08 | R | F | 1992-03-11 | 1992-01-27 | TAKE BACK RETURN | RAIL | egular deposit
0 | ons wake slyly after the fluffily final packages. |
| 1992-01-04 | 511807747 | 1 | 3673453 | 673454 | 17.00 | 24246.59 | 0.04 | 0.08 | A | F | 1992-03-04 | 1992-01-06 | TAKE BACK RETURN | FOB | uctions use quickly even instr
0 | pecial packages use carefully express requests. foxes cajol |
| 1992-01-04 | 521855463 | 1 | 10976993 | 976994 | 30.00 | 62083.50 | 0.06 | 0.03 | R | F | 1992-03-03 | 1992-01-13 | COLLECT COD | FOB | to haggle slyly p
0 | nts sleep slyly accounts. final packages dazzle above the express theodolit |
10 rows in set (5.10 sec)
```

其中，涉及的表结构为：

lineitem 表：

```
CREATE TABLE lineitem (
  l_shipdate date NOT NULL,
  l_orderkey bigint(20) NOT NULL,
  l_linenumber int(11) NOT NULL,
  l_partkey int(11) NOT NULL,
  l_suppkey int(11) NOT NULL,
  l_quantity decimalv3(15, 2) NOT NULL,
  l_extendedprice decimalv3(15, 2) NOT NULL,
  l_discount decimalv3(15, 2) NOT NULL,
  l_tax decimalv3(15, 2) NOT NULL,
  l_returnflag varchar(1) NOT NULL,
  l_linestatus varchar(1) NOT NULL,
  l_commitdate date NOT NULL,
  l_receiptdate date NOT NULL,
  l_shipinstruct varchar(25) NOT NULL,
  l_shipmode varchar(10) NOT NULL,
  l_comment varchar(44) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(l_shipdate, l_orderkey)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(l_orderkey) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
);
```

orders 表：

```
 CREATE TABLE orders (
  o_orderkey bigint(20) NOT NULL,
  o_orderdate date NOT NULL,
  o_custkey int(11) NOT NULL,
  o_orderstatus varchar(1) NOT NULL,
  o_totalprice decimalv3(15, 2) NOT NULL,
  o_orderpriority varchar(15) NOT NULL,
  o_clerk varchar(15) NOT NULL,
  o_shippriority int(11) NOT NULL,
  o_comment varchar(79) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(o_orderkey, o_orderdate)
COMMENT 'OLAP'
DISTRIBUTED BY HASH(o_orderkey) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false",
"storage_format" = "V2",
"disable_auto_compaction" = "false"
);
```

通过explain 查看执行计划，分析现在使用的 join 方式

```
explain select * from lineitem inner join orders on l_orderkey = o_orderkey   where
```

```
|                                                                                                   |
| PLAN FRAGMENT 1                                                                                   |
|                                                                                                   |
|   PARTITION: HASH_PARTITIONED: `default_cluster:tpch_100_d`.`lineitem`.`l_orderkey`               |
|                                                                                                   |
|   STREAM DATA SINK                                                                                |
|     EXCHANGE ID: 04                                                                               |
|     UNPARTITIONED                                                                                 |
|                                                                                                   |
|   2:VHASH JOIN                                                                                    |
|   |  join op: INNER JOIN(BUCKET_SHUFFLE)[Tables are not in the same group]                        |
|   |  is mark: false                                                                               |
|   |  equal join conjunct: `l_orderkey` = `o_orderkey`                                             |
|   |  runtime filters: RF000[in_or_bloom] <- `o_orderkey`                                          |
|   |  cardinality=600,037,914                                                                      |
|   |  vec output tuple id: 2                                                                       |
|   |  vIntermediate tuple ids: 3 4                                                                 |
|   |  output slot ids: 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49  |
|   |  hash output slot ids: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24       |
|   |  limit: 10                                                                                    |
|   |                                                                                               |
|   |----3:VEXCHANGE                                                                                |
|   |       offset: 0                                                                               |
|   |                                                                                               |
|   0:VOlapScanNode                                                                                 |
|      TABLE: default_cluster:tpch_100_d.lineitem(lineitem), PREAGGREGATION: ON                     |
|      PREDICATES: `l_orderkey` > 500000000                                                         |
|      runtime filters: RF000[in_or_bloom] -> `l_orderkey`                                          |
|      partitions=1/1, tablets=96/96, tabletList=16332,16336,16340 ...                              |
|      cardinality=600037914, avgRowSize=165.18039, numNodes=3                                      |
|                                                                                                   |
| PLAN FRAGMENT 2                                                                                   |
|                                                                                                   |
|   PARTITION: HASH_PARTITIONED: `default_cluster:tpch_100_d`.`orders`.`o_orderkey`                 |
|                                                                                                   |
|   STREAM DATA SINK                                                                                |
|     EXCHANGE ID: 03                                                                               |
|     BUCKET_SHFFULE_HASH_PARTITIONED: `o_orderkey`                                                 |
|                                                                                                   |
|   1:VOlapScanNode                                                                                 |
|      TABLE: default_cluster:tpch_100_d.orders(orders), PREAGGREGATION: ON                         |
|      PREDICATES: `o_orderkey` > 500000000                                                         |
|      partitions=1/1, tablets=96/96, tabletList=16975,16979,16983 ...                              |
|      cardinality=150000000, avgRowSize=214.74033, numNodes=3                                      |
+---------------------------------------------------------------------------------------------------+
78 rows in set (0.00 sec)

MySQL [tpch_100_d]> explain select * from lineitem inner join orders on l_orderkey = o_orderkey   where l_orderkey >500000000 lim
```

在 hash join node 可以看到使用的 join 是 bucket shuffle join。

分析左表右表是否满足 colocation join 条件

可以看到2个表的相关条件：

条件	lineitem 表	order 表	是否满足 colocation join 条件
分桶列	l_orderkey(类型 bigint20)	o_orderkey(类型 bigint 20)	满足
分桶数	96	96	满足
副本数	3	3	满足

如果业务的表不满足，可以通过重新建表，进行适配，然后使用 select into select 的方式进行数据迁移。

修改表属性满足同一个 colocation group 条件

```
ALTER TABLE lineitem SET ("colocate_with" = "tpch_group");ALTER TABLE orders SET ("
```

通过 explain 分析 colocation join 是否生效

```
|    <slot 46>                                                                          |
|    <slot 47>                                                                          |
|    <slot 48>                                                                          |
|    <slot 49>                                                                          |
|   PARTITION: UNPARTITIONED                                                            |
|                                                                                       |
|   VRESULT SINK                                                                        |
|                                                                                       |
|   3:VEXCHANGE                                                                         |
|      offset: 0                                                                        |
|      limit: 10                                                                        |
|                                                                                       |
| PLAN FRAGMENT 1                                                                       |
|                                                                                       |
|   PARTITION: HASH_PARTITIONED: `default_cluster:tpch_100_d`.`lineitem`.`l_orderkey`   |
|                                                                                       |
|   STREAM DATA SINK                                                                    |
|     EXCHANGE ID: 03                                                                   |
|     UNPARTITIONED                                                                     |
|                                                                                       |
|   2:VHASH JOIN                                                                        |
|   |  join op: INNER JOIN(COLOCATE[])[]                                                |
|   |  is mark: false                                                                   |
|   |  equal join conjunct: `l_orderkey` = `o_orderkey`                                 |
|   |  runtime filters: RF000[in_or_bloom] <- `o_orderkey`                              |
|   |  cardinality=600,037,914                                                          |
|   |  vec output tuple id: 2                                                           |
|   |  vIntermediate tuple ids: 3 4                                                     |
|   |  output slot ids: 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 |
|   |  hash output slot ids: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 |
|   |  limit: 10                                                                        |
|   |                                                                                   |
|   |----1:VOlapScanNode                                                                |
|   |       TABLE: default_cluster:tpch_100_d.orders(orders), PREAGGREGATION: ON        |
|   |       PREDICATES: `o_orderkey` > 500000000                                        |
|   |       partitions=1/1, tablets=96/96, tabletList=16975,16979,16983 ...             |
|   |       cardinality=150000000, avgRowSize=214.74033, numNodes=3                     |
|   |                                                                                   |
|   0:VOlapScanNode                                                                     |
|      TABLE: default_cluster:tpch_100_d.lineitem(lineitem), PREAGGREGATION: ON         |
|      PREDICATES: `l_orderkey` > 500000000                                             |
|      runtime filters: RF000[in_or_bloom] -> `l_orderkey`                              |
|      partitions=1/1, tablets=96/96, tabletList=16332,16336,16340 ...                  |
|      cardinality=600037914, avgRowSize=165.18039, numNodes=3                          |
+---------------------------------------------------------------------------------------+
67 rows in set (0.00 sec)

MySQL [tpch_100_d]> explain select * from lineitem inner join orders on l_orderkey = o_orderkey  where l_orderkey >500000000 limit 10;
```

实际执行 SQL 查看效率是否有所提升

对比 colocation join 优化前的耗时（5s+），colocation join 优化后，耗时变为2s。效率提升了一倍多。在大表 join 大表场景，效率提升会更加明显。

```
MySQL [tpch_100_d]> select * from lineitem inner join orders on l_orderkey = o_orderkey  where l_orderkey >500000000 limit 10;
+------------+-----------+--------------+-----------+-----------+------------+-----------------+------------+-------+--------------+--------------+--------------+---------------+-------------------+------------+------------------------------------
------------+-------------------------------------------------------------------------+
| l_shipdate | l_orderkey | l_linenumber | l_partkey | l_suppkey | l_quantity | l_extendedprice | l_discount | l_tax | l_returnflag | l_linestatus | l_commitdate | l_receiptdate | l_shipinstruct    | l_shipmode | l_comment
shippriority | o_comment                                                               |
+------------+-----------+--------------+-----------+-----------+------------+-----------------+------------+-------+--------------+--------------+--------------+---------------+-------------------+------------+------------------------------------
------------+-------------------------------------------------------------------------+
| 1992-01-03 |  500254247 |            4 |  15458143 |    208189 |      29.00 |        31910.73 |       0.10 |  0.01 | R            | F            | 1992-02-08   | 1992-02-01    | COLLECT COD       | TRUCK      | ly throughout the quick
           0 |  ironic pinto beans above the slyly unusual accounts use furiously around the  |
| 1992-01-03 |  522213827 |            3 |   5169655 |    919671 |      13.00 |        22417.20 |       0.09 |  0.03 | R            | F            | 1992-03-13   | 1992-01-06    | COLLECT COD       | MAIL       | s sleep slyly. pai
           0 | yly ironic deposits nag slyly after the furiously ironic re                    |
| 1992-01-03 |  528040419 |            1 |  12188746 |    188747 |      40.00 |        73365.60 |       0.08 |  0.06 | R            | F            | 1992-02-18   | 1992-01-11    | COLLECT COD       | REG AIR    | sleep blithely a
           0 | sits wake quickly. quickly regular deposits sleep. fin                         |
| 1992-01-03 |  534275879 |            2 |   2998679 |    748686 |      13.00 |        23107.89 |       0.02 |  0.04 | R            | F            | 1992-03-30   | 1992-01-06    | COLLECT COD       | TRUCK      | ld accounts are slyly
           0 | odolites. special packages after the ironic foxes could haggle c               |
| 1992-01-03 |  543363872 |            1 |   1589019 |    589020 |      12.00 |        13295.28 |       0.08 |  0.07 | R            | F            | 1992-03-30   | 1992-01-18    | NONE              | RAIL       | ccounts. final,
           0 | about the furiously final platelets: reques                                    |
| 1992-01-03 |  579446181 |            3 |   7988290 |    488305 |      21.00 |        28935.90 |       0.06 |  0.08 | R            | F            | 1992-02-07   | 1992-01-08    | TAKE BACK RETURN  | SHIP       |  theodolites. blithely final gifts
           0 | lly regular orbits after the fluffily even theodolites sleep around            |
| 1992-01-03 |  583236356 |            1 |  17260907 |     10959 |      28.00 |        52277.12 |       0.00 |  0.06 | A            | F            | 1992-03-25   | 1992-01-19    | NONE              | FOB        | ic theodolites. carefully final pinto 
           0 | ct quickly against the regular, special deposits:                              |
| 1992-01-04 |  505923011 |            4 |   8383087 |    383088 |      35.00 |        40938.45 |       0.06 |  0.08 | R            | F            | 1992-03-11   | 1992-01-27    | TAKE BACK RETURN  | RAIL       | egular deposit
           0 | ons wake slyly after the fluffily final packages.                              |
| 1992-01-04 |  511807747 |            1 |   3673453 |    673454 |      17.00 |        24246.59 |       0.04 |  0.08 | A            | F            | 1992-03-04   | 1992-01-06    | TAKE BACK RETURN  | FOB        | uctions use quickly even instr
           0 | pecial packages use carefully express requests. foxes cajol                    |
| 1992-01-04 |  521855463 |            1 |  10976993 |    976994 |      30.00 |        62083.50 |       0.06 |  0.03 | R            | F            | 1992-03-03   | 1992-01-13    | COLLECT COD       | FOB        | to haggle slyly p
           0 | nts sleep slyly accounts. final packages dazzle above the express theodolit   |
+------------+-----------+--------------+-----------+-----------+------------+-----------------+------------+-------+--------------+--------------+--------------+---------------+-------------------+------------+------------------------------------
------------+-------------------------------------------------------------------------+
10 rows in set (1.99 sec)
```

Bucket Shuffle Join

最近更新时间：2024-06-27 11:11:46

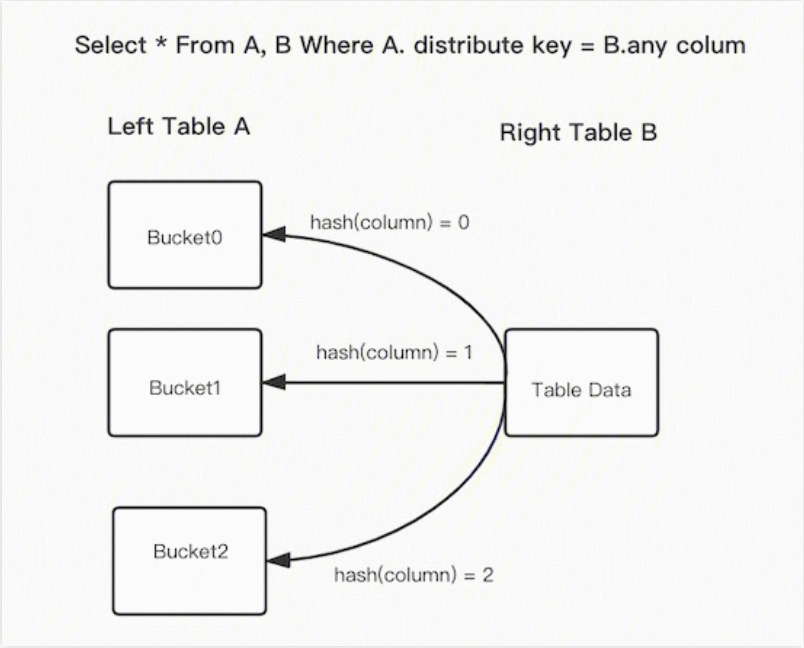
Bucket Shuffle Join 旨在为某些 Join 查询提供本地性优化，来减少数据在节点间的传输耗时，来加速查询。它的设计、实现和效果可以参阅 ISSUE 4394。

名词解释

FE：Frontend，Doris 的前端节点。负责元数据管理和请求接入。
BE：Backend，Doris 的后端节点。负责查询执行和数据存储。
左表：Join 查询时，左边的表。进行 Probe 操作。可被 Join Reorder 调整顺序。
右表：Join 查询时，右边的表。进行 Build 操作。可被 Join Reorder 调整顺序。

原理

Doris 支持的常规分布式 Join 方式包括了 shuffle join 和 broadcast join。这两种 join 都会导致不小的网络开销：
举个例子，当前存在A表与B表的 Join 查询，它的 Join 方式为 HashJoin，不同 Join 类型的开销如下：
Broadcast Join：如果根据数据分布，查询规划出A表有3个执行的 HashJoinNode，那么需要将B表全量的发送到3个 HashJoinNode，那么它的网络开销是 `3B`，它的内存开销也是 `3B`。
Shuffle Join：Shuffle Join会将A，B两张表的数据根据哈希计算分散到集群的节点之中，所以它的网络开销为 `A + B`，内存开销为 `B`。
在 FE 之中保存了 Doris 每个表的数据分布信息，如果 join 语句命中了表的数据分布列，我们应该使用数据分布信息来减少 join 语句的网络与内存开销，这就是 Bucket Shuffle Join 的思路来源。



上面的图片展示了 Bucket Shuffle Join 的工作原理。SQL 语句为 A 表 join B 表，并且 join 的等值表达式命中了 A 的数据分布列。而 Bucket Shuffle Join 会根据 A 表的数据分布信息，将 B 表的数据发送到对应的 A 表的数据存储计算节点。Bucket Shuffle Join 开销如下：

网络开销： `B < min(3B, A + B)`

内存开销： `B <= min(3B, B)`

可见，相比于 Broadcast Join 与 Shuffle Join， Bucket Shuffle Join 有着较为明显的性能优势。减少数据在节点间的传输耗时和 Join 时的内存开销。相对于 Doris 原有的 Join 方式，它有着下面的优点：

Bucket-Shuffle-Join 降低了网络与内存开销，使一些 Join 查询具有了更好的性能。尤其是当 FE 能够执行左表的分区裁剪与桶裁剪时。

同时与 Colocate Join 不同，它对于表的数据分布方式并没有侵入性，这对于用户来说是透明的。对于表的数据分布没有强制性的要求，不容易导致数据倾斜的问题。

它可以为 Join Reorder 提供更多可能的优化空间。

使用方式

设置 Session 变量

将 session 变量 `enable_bucket_shuffle_join` 设置为 `true` ，则FE在进行查询规划时就会默认将能够转换为 Bucket Shuffle Join 的查询自动规划为 Bucket Shuffle Join。

```
set enable_bucket_shuffle_join = true;
```

在 FE 进行分布式查询规划时，优先选择的顺序为 Colocate Join > Bucket Shuffle Join > Broadcast Join > Shuffle Join。但是如果用户显式 hint 了 Join 的类型，如：

```
select * from test join [shuffle] baseall on test.k1 = baseall.k1;
```

则上述的选择优先顺序则不生效。

该 session 变量在0.14版本默认为 `true` , 而0.13版本需要手动设置为 `true` 。

查看 Join 的类型

可以通过 `explain` 命令来查看Join是否为Bucket Shuffle Join：

```
|   2:HASH JOIN
|   |  join op: INNER JOIN (BUCKET_SHUFFLE)
|   |  hash predicates:
|   |  colocate: false, reason: table not in the same group
|   |  equal join conjunct: `test`.`k1` = `baseall`.`k1`
```

在 Join 类型之中会指明使用的 Join 方式为： `BUCKET_SHUFFLE` 。

Bucket Shuffle Join 的规划规则

在绝大多数场景之中，用户只需要默认打开 seesion 变量的开关就可以透明的使用这种 Join 方式带来的性能提升，但是如果了解 Bucket Shuffle Join 的规划规则，可以帮助我们利用它写出更加高效的 SQL。

Bucket Shuffle Join 只生效于 Join 条件为等值的场景，原因与 Colocate Join 类似，它们都依赖 hash 来计算确定的数据分布。

在等值 Join 条件之中包含两张表的分桶列，当左表的分桶列为等值的 Join 条件时，它有很大概率会被规划为 Bucket Shuffle Join。

由于不同的数据类型的 hash 值计算结果不同，所以 Bucket Shuffle Join 要求左表的分桶列的类型与右表等值 join 列的类型需要保持一致，否则无法进行对应的规划。

Bucket Shuffle Join 只作用于 Doris 原生的 OLAP 表，对于 ODBC，MySQL，ES 等外表，当其作为左表时是无法规划生效的。

对于分区表，由于每一个分区的数据分布规则可能不同，所以 Bucket Shuffle Join 只能保证左表为单分区时生效。所以在 SQL 执行之中，需要尽量使用 `where` 条件使分区裁剪的策略能够生效。

假如左表为 Colocate 的表，那么它每个分区的数据分布规则是确定的，Bucket Shuffle Join 能在 Colocate 表上表现更好。

最佳实践

基于现有 SQL 判断是否满足 Bucket Shuffle Join 条件。

这里以 TPC-H 测试集的 sql3 为例子。

```
    select     l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,    o_shippriorityfrom    (
      select l_orderkey, l_extendedprice, l_discount, o_orderdate, o_shippriority,
      from       lineitem
      join orders
      where l_orderkey = o_orderkey
      and o_orderdate < date '1995-03-15'
      and l_shipdate > date '1995-03-15'    ) t1
    join customer c    on c.c_custkey = t1.o_custkey
    where c_mktsegment = 'BUILDING'
    group by    l_orderkey,    o_orderdate,    o_shippriority
    order by    revenue desc,    o_orderdate
    limit 10;
```

从 sql3 可以看到子查询使用的是 join条件是等值，满足其中一个条件，其他条件，我们可以在建表阶段满足。

```
select l_orderkey, l_extendedprice, l_discount, o_orderdate, o_shippriority, o_cust
from lineitem
join orderswhere l_orderkey = o_orderkeyand o_orderdate < date '1995-03-15'
and l_shipdate > date '1995-03-15'
```

检查环境变量开启

```
MySQL [tpch_100_d]> show variables like "%enable_bucket_shuffle_join%";
+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| enable_bucket_shuffle_join | true  |
+---------------------------+-------+
1 row in set (0.00 sec)
```

创建对应的表，保持两张表分桶列和其对应的类型一致

```
CREATE TABLE `lineitem` (
`l_shipdate` date NOT NULL,
`l_orderkey` bigint(20) NOT NULL,
`l_linenumber` int(11) NOT NULL,
`l_partkey` int(11) NOT NULL,
`l_suppkey` int(11) NOT NULL,
`l_quantity` decimalv3(15, 2) NOT NULL,
`l_extendedprice` decimalv3(15, 2) NOT NULL,
`l_discount` decimalv3(15, 2) NOT NULL,
```

```
`l_tax` decimalv3(15, 2) NOT NULL,
`l_returnflag` varchar(1) NOT NULL,
`l_linestatus` varchar(1) NOT NULL,
`l_commitdate` date NOT NULL,
`l_receiptdate` date NOT NULL,
`l_shipinstruct` varchar(25) NOT NULL,
`l_shipmode` varchar(10) NOT NULL,
`l_comment` varchar(44) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(`l_shipdate`, `l_orderkey`)COMMENT 'OLAP'
DISTRIBUTED BY HASH(`l_orderkey`) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false","storage_format" = "V2",
"disable_auto_compaction" = "false"
);

CREATE TABLE `orders` (
`o_orderkey` bigint(20) NOT NULL,
`o_orderdate` date NOT NULL,
`o_custkey` int(11) NOT NULL,
`o_orderstatus` varchar(1) NOT NULL,
`o_totalprice` decimalv3(15, 2) NOT NULL,
`o_orderpriority` varchar(15) NOT NULL,
`o_clerk` varchar(15) NOT NULL,
`o_shippriority` int(11) NOT NULL,
`o_comment` varchar(79) NOT NULL
) ENGINE=OLAP
DUPLICATE KEY(`o_orderkey`, `o_orderdate`) COMMENT 'OLAP'
DISTRIBUTED BY HASH(`o_orderkey`) BUCKETS 96
PROPERTIES (
"replication_allocation" = "tag.location.default: 3",
"in_memory" = "false","storage_format" = "V2",
"disable_auto_compaction" = "false"
);
```

通过 explain 分析业务 sql，确保 Bucket Shuffle Join 可以生效

```
     offset: 0

 PLAN FRAGMENT 3

   PARTITION: HASH_PARTITIONED: `default_cluster:tpch_100_d`.`customer`.`c_custkey`

   STREAM DATA SINK
     EXCHANGE ID: 09
     HASH_PARTITIONED: `c`.`c_custkey`

   3:VOlapScanNode
      TABLE: default_cluster:tpch_100_d.customer(customer), PREAGGREGATION: ON
      PREDICATES: `c`.`c_mktsegment` = 'BUILDING'
      partitions=1/1, tablets=24/24, tabletList=17369,17373,17377 ...
      cardinality=15000000, avgRowSize=460.55078, numNodes=3

 PLAN FRAGMENT 4

   PARTITION: HASH_PARTITIONED: `default_cluster:tpch_100_d`.`lineitem`.`l_orderkey`

   STREAM DATA SINK
     EXCHANGE ID: 08
     HASH_PARTITIONED: <slot 30>

   2:VHASH JOIN
   |  join op: INNER JOIN(BUCKET_SHUFFLE)[Tables are not in the same group]
   |  is mark: false
   |  equal join conjunct: `l_orderkey` = `o_orderkey`
   |  runtime filters: RF001[in_or_bloom] <- `o_orderkey`
   |  cardinality=600,037,902
   |  vec output tuple id: 6
   |  vIntermediate tuple ids: 8 9
   |  output slot ids: 24 25 26 28 29 30
   |  hash output slot ids: 0 1 2 3 4 5
   |
   |----7:VEXCHANGE
   |       offset: 0
   |
   0:VOlapScanNode
      TABLE: default_cluster:tpch_100_d.lineitem(lineitem), PREAGGREGATION: ON
      PREDICATES: `l_shipdate` > '1995-03-15 00:00:00'
      runtime filters: RF001[in_or_bloom] -> `l_orderkey`
      partitions=1/1, tablets=96/96, tabletList=16332,16336,16340 ...
      cardinality=600037902, avgRowSize=165.18037, numNodes=3

 PLAN FRAGMENT 5

   PARTITION: HASH_PARTITIONED: `default_cluster:tpch_100_d`.`orders`.`o_orderkey`

   STREAM DATA SINK
     EXCHANGE ID: 07
     BUCKET_SHFFULE_HASH_PARTITIONED: `o_orderkey`

   1:VOlapScanNode
      TABLE: default_cluster:tpch_100_d.orders(orders), PREAGGREGATION: ON
      PREDICATES: `o_orderdate` < '1995-03-15 00:00:00'
      runtime filters: RF000[in_or_bloom] -> <slot 5>
      partitions=1/1, tablets=96/96, tabletList=16975,16979,16983 ...
      cardinality=150000000, avgRowSize=214.74033, numNodes=3
+----------------------------------------------------------------------------------------------------------------------+
123 rows in set (0.00 sec)

MySQL [tpch_100_d]> explain select      l_orderkey,     sum(l_extendedprice * (1 - l_discount)) as revenue,     o_orderdate,     o_shippriority from     (         select l_o
 from         lineitem join orders         where l_orderkey = o_orderkey         and o_orderdate < date '1995-03-15'         and l_shipdate > date '1995-03-15'     ) t1 join
DING' group by     l_orderkey,     o_orderdate,     o_shippriority order by     revenue desc,     o_orderdate limit 10;
```

Runtime Filter

最近更新时间：2024-06-27 11:12:22

Runtime Filter 是在 Doris 0.15 版本中正式加入的新功能。旨在为某些 Join 查询在运行时动态生成过滤条件，来减少扫描的数据量，避免不必要的 I/O 和网络传输，从而加速查询。它的设计、实现和效果可以参阅 ISSUE 6116。

名词解释

FE：Frontend，Doris 的前端节点。负责元数据管理和请求接入。
BE：Backend，Doris 的后端节点。负责查询执行和数据存储。
左表：Join 查询时，左边的表。进行 Probe 操作。可被 Join Reorder 调整顺序。
右表：Join 查询时，右边的表。进行 Build 操作。可被 Join Reorder 调整顺序。
Fragment：FE 会将具体的 SQL 语句的执行转化为对应的 Fragment 并下发到 BE 进行执行。BE 上执行对应 Fragment，并将结果汇聚返回给 FE。
Join on clause: `A join B on A.a=B.b` 中的 `A.a=B.b`，在查询规划时基于此生成 join conjuncts，包含 join Build 和 Probe 使用的 expr，其中 Build expr 在 Runtime Filter 中称为 src expr，Probe expr 在 Runtime Filter 中称为 target expr。

原理

Runtime Filter 在查询规划时生成，在 HashJoinNode 中构建，在 ScanNode 中应用。
举个例子，当前存在 T1表与 T2表的 Join 查询，它的 Join 方式为 HashJoin，T1是一张事实表，数据行数为 100000，T2是一张维度表，数据行数为2000，Doris join 的实际情况是:

```
|           >       HashJoinNode      <
|           |                           |
|           | 100000                    | 2000
|           |                           |
|    OlapScanNode                OlapScanNode
|           ^                           ^
|           | 100000                    | 2000
|         T1                          T2
|
```

显而易见对 T2扫描数据要远远快于 T1，如果我们主动等待一段时间再扫描 T1，等 T2将扫描的数据记录交给 HashJoinNode 后，HashJoinNode 根据 T2的数据计算出一个过滤条件，例如 T2数据的最大和最小值，或者构建一个 Bloom Filter，接着将这个过滤条件发给等待扫描 T1的 ScanNode，后者应用这个过滤条件，将过滤后的数据交给 HashJoinNode，从而减少 probe hash table 的次数和网络开销，这个过滤条件就是 Runtime Filter，效果如下:

```
|           >       HashJoinNode      <
|           |                           |
|           | 6000                      | 2000
|           |                           |
|    OlapScanNode                 OlapScanNode
|           ^                           ^
|           | 100000                    | 2000
|         T1                          T2
|
```

如果能将过滤条件（Runtime Filter）下推到存储引擎，则某些情况下可以利用索引来直接减少扫描的数据量，从而大大减少扫描耗时，效果如下:

```
|           >       HashJoinNode      <
|           |                           |
|           | 6000                      | 2000
|           |                           |
|    OlapScanNode                 OlapScanNode
|           ^                           ^
|           | 6000                      | 2000
|         T1                          T2
|
```

可见，和谓词下推、分区裁剪不同，Runtime Filter 是在运行时动态生成的过滤条件，即在查询运行时解析 join on clause 确定过滤表达式，并将表达式广播给正在读取左表的 ScanNode，从而减少扫描的数据量，进而减少 probe hash table 的次数，避免不必要的 I/O 和网络传输。
Runtime Filter 主要用于优化针对大表的 join，如果左表的数据量太小，或者右表的数据量太大，则 Runtime Filter 可能不会取得预期效果。

使用方式

Runtime Filter查询选项

与 Runtime Filter 相关的查询选项信息，请参阅以下部分:
第一个查询选项是调整使用的Runtime Filter类型，大多数情况下，您只需要调整这一个选项，其他选项保持默认即可。

`runtime_filter_type`：包括Bloom Filter、MinMax Filter、IN predicate，默认会保守的只使用IN predicate，部分情况下同时使用Bloom Filter、MinMax Filter、IN predicate时性能更高。
其他查询选项通常仅在某些特定场景下，才需进一步调整以达到最优效果。通常只在性能测试后，针对资源密集型、运行耗时足够长且频率足够高的查询进行优化。

`runtime_filter_mode`：用于调整Runtime Filter的下推策略，包括OFF、LOCAL、GLOBAL三种策略，默认设置为GLOBAL策略

`runtime_filter_wait_time_ms` : 左表的ScanNode等待每个Runtime Filter的时间，默认1000ms。
`runtime_filters_max_num` : 每个查询可应用的Runtime Filter中Bloom Filter的最大数量，默认10。
`runtime_bloom_filter_min_size` : Runtime Filter中Bloom Filter的最小长度，默认1048576（1M）。
`runtime_bloom_filter_max_size` : Runtime Filter中Bloom Filter的最大长度，默认16777216（16M）。
`runtime_bloom_filter_size` : Runtime Filter中Bloom Filter的默认长度，默认2097152（2M）。
`runtime_filter_max_in_num` : 如果join右表数据行数大于这个值，我们将不生成IN predicate，默认1024。

1. runtime_filter_type

使用的 Runtime Filter 类型。
类型：数字(1, 2, 4)或者相对应的助记符字符串(IN, BLOOM_FILTER, MIN_MAX)，默认1(IN predicate)，使用多个时用逗号分隔，注意需要加引号，或者将任意多个类型的数字相加，例如:

```
set runtime_filter_type="BLOOM_FILTER,IN,MIN_MAX";
```

等价于:

```
set runtime_filter_type=7;
```

使用注意事项
Bloom Filter: 有一定的误判率，导致过滤的数据比预期少一点，但不会导致最终结果不准确，在大部分情况下 Bloom Filter 都可以提升性能或对性能没有显著影响，但在部分情况下会导致性能降低。
Bloom Filter 构建和应用的开销较高，所以当过滤率较低时，或者左表数据量较少时，Bloom Filter 可能会导致性能降低。
目前只有左表的 Key 列应用 Bloom Filter 才能下推到存储引擎，而测试结果显示 Bloom Filter 不下推到存储引擎时往往会导致性能降低。
目前 Bloom Filter 仅在 ScanNode 上使用表达式过滤时有短路(short-circuit)逻辑，即当假阳性率过高时，不继续使用 Bloom Filter，但当 Bloom Filter 下推到存储引擎后没有短路逻辑，所以当过滤率较低时可能导致性能降低。
MinMax Filter: 包含最大值和最小值，从而过滤小于最小值和大于最大值的数据，MinMax Filter 的过滤效果与 join on clause 中 Key 列的类型和左右表数据分布有关。
当 join on clause 中 Key 列的类型为 int/bigint/double 等时，极端情况下，如果左右表的最大最小值相同则没有效果，反之右表最大值小于左表最小值，或右表最小值大于左表最大值，则效果最好。
当 join on clause 中 Key 列的类型为 varchar 等时，应用 MinMax Filter 往往会导致性能降低。
IN predicate: 根据 join on clause 中 Key 列在右表上的所有值构建 IN predicate，使用构建的 IN predicate 在左表上过滤，相比 Bloom Filter 构建和应用的开销更低，在右表数据量较少时往往性能更高。
默认只有右表数据行数少于1024才会下推（可通过 session 变量中的 `runtime_filter_max_in_num` 调整）。
目前 IN predicate 没有实现合并方法，即无法跨 Fragment 下推，所以目前当需要下推给 shuffle join 左表的 ScanNode 时，如果没有生成 Bloom Filter，那么我们会将 IN predicate 转为 Bloom Filter，用于处理跨 Fragment 下推，所以即使类型只选择了 IN predicate，实际也可能应用了 Bloom Filter；

2. runtime_filter_mode

用于控制 Runtime Filter 在 instance 之间传输的范围。

类型：数字(0, 1, 2)或者相对应的助记符字符串(OFF, LOCAL, GLOBAL)，默认2(GLOBAL)。

使用注意事项

LOCAL：相对保守，构建的 Runtime Filter 只能在同一个 instance（查询执行的最小单元）上同一个 Fragment 中使用，即 Runtime Filter 生产者（构建 Filter 的 HashJoinNode）和消费者（使用 RuntimeFilter 的 ScanNode）在同一个 Fragment，例如 broadcast join 的一般场景。

GLOBAL：相对激进，除满足 LOCAL 策略的场景外，还可以将 Runtime Filter 合并后通过网络传输到不同 instance 上的不同 Fragment 中使用，例如 Runtime Filter 生产者和消费者在不同 Fragment，例如 shuffle join。

大多数情况下 GLOBAL 策略可以在更广泛的场景对查询进行优化，但在有些 shuffle join 中生成和合并 Runtime Filter 的开销超过给查询带来的性能优势，可以考虑更改为 LOCAL 策略。

如果集群中涉及的 join 查询不会因为 Runtime Filter 而提高性能，您可以将设置更改为 OFF，从而完全关闭该功能。

在不同 Fragment 上构建和应用 Runtime Filter 时，需要合并 Runtime Filter 的原因和策略可参阅 ISSUE 6116。

3.runtime_filter_wait_time_ms

Runtime Filter 的等待耗时。

类型: 整数，默认1000，单位ms。

使用注意事项

在开启 Runtime Filter 后，左表的 ScanNode 会为每一个分配给自己的 Runtime Filter 等待一段时间再扫描数据，即如果 ScanNode 被分配了3个 Runtime Filter，那么它最多会等待3000ms。

因为 Runtime Filter 的构建和合并均需要时间，ScanNode 会尝试将等待时间内到达的 Runtime Filter 下推到存储引擎，如果超过等待时间后，ScanNode 会使用已经到达的 Runtime Filter 直接开始扫描数据。

如果 Runtime Filter 在 ScanNode 开始扫描之后到达，则 ScanNode 不会将该 Runtime Filter 下推到存储引擎，而是对已经从存储引擎扫描上来的数据，在 ScanNode 上基于该 Runtime Filter 使用表达式过滤，之前已经扫描的数据则不会应用该 Runtime Filter，这样得到的中间数据规模会大于最优解，但可以避免严重的裂化。

如果集群比较繁忙，并且集群上有许多资源密集型或长耗时的查询，可以考虑增加等待时间，以避免复杂查询错过优化机会。如果集群负载较轻，并且集群上有许多只需要几秒的小查询，可以考虑减少等待时间，以避免每个查询增加1s的延迟。

4.runtime_filters_max_num

每个查询生成的 Runtime Filter 中 Bloom Filter 数量的上限。

类型: 整数，默认10。

使用注意事项

目前仅对 Bloom Filter 的数量进行限制，因为相比 MinMax Filter 和 IN predicate，Bloom Filter 构建和应用的代价更高。

如果生成的 Bloom Filter 超过允许的最大数量，则保留选择性大的 Bloom Filter，选择性大意味着预期可以过滤更多的行。这个设置可以防止 Bloom Filter 耗费过多的内存开销而导致潜在的问题。

```
选择性=(HashJoinNode Cardinality / HashJoinNode left child Cardinality)
-- 因为目前FE拿到Cardinality不准，所以这里Bloom Filter计算的选择性与实际不准，因此最终可能只是
```

仅在对涉及大表间 join 的某些长耗时查询进行调优时，才需要调整此查询选项。

5.Bloom Filter长度相关参数

包括 `runtime_bloom_filter_min_size` 、 `runtime_bloom_filter_max_size` 、 `runtime_bloom_filter_size` ，用于确定 Runtime Filter 使用的 Bloom Filter 数据结构的大小（以字节为单位）。

类型: 整数。

使用注意事项

因为需要保证每个 HashJoinNode 构建的 Bloom Filter 长度相同才能合并，所以目前在FE查询规划时计算 Bloom Filter 的长度。

如果能拿到 join 右表统计信息中的数据行数(Cardinality)，会尝试根据 Cardinality 估计 Bloom Filter 的最佳大小，并四舍五入到最接近的2的幂(以2为底的log值)。如果无法拿到右表的 Cardinality，则会使用默认的Bloom Filter 长度 `runtime_bloom_filter_size` 。 `runtime_bloom_filter_min_size` 和 `runtime_bloom_filter_max_size` 用于限制最终使用的 Bloom Filter 长度最小和最大值。

更大的 Bloom Filter 在处理高基数的输入集时更有效，但需要消耗更多的内存。假如查询中需要过滤高基数列（例如含有数百万个不同的取值），可以考虑增加 `runtime_bloom_filter_size` 的值进行一些基准测试，这有助于使 Bloom Filter 过滤的更加精准，从而获得预期的性能提升。

Bloom Filter 的有效性取决于查询的数据分布，因此通常仅对一些特定查询额外调整其 Bloom Filter 长度，而不是全局修改，一般仅在对涉及大表间 join 的某些长耗时查询进行调优时，才需要调整此查询选项。

查看query生成的Runtime Filter

`explain` 命令可以显示的查询计划中包括每个 Fragment 使用的 join on clause 信息，以及 Fragment 生成和使用 Runtime Filter 的注释，从而确认是否将 Runtime Filter 应用到了期望的 join on clause 上。

生成 Runtime Filter 的 Fragment 包含的注释，例如 `runtime filters: filter_id[type] <- table.column` 。

使用 Runtime Filter 的 Fragment 包含的注释，例如 `runtime filters: filter_id[type] -> table.column` 。

下面例子中的查询使用了一个 ID 为RF000的 Runtime Filter。

```
CREATE TABLE test (t1 INT) DISTRIBUTED BY HASH (t1) BUCKETS 2 PROPERTIES("replicati
INSERT INTO test VALUES (1), (2), (3), (4);

CREATE TABLE test2 (t2 INT) DISTRIBUTED BY HASH (t2) BUCKETS 2 PROPERTIES("replicat
INSERT INTO test2 VALUES (3), (4), (5);

EXPLAIN SELECT t1 FROM test JOIN test2 where test.t1 = test2.t2;
+------------------------------------------------------------------------+
| Explain String                                                         |
+------------------------------------------------------------------------+
| PLAN FRAGMENT 0                                                        |
```

```
|  OUTPUT EXPRS:`t1`                                                    |
|                                                                       |
|   4:EXCHANGE                                                          |
|                                                                       |
| PLAN FRAGMENT 1                                                       |
|  OUTPUT EXPRS:                                                        |
|   PARTITION: HASH_PARTITIONED: `default_cluster:ssb`.`test`.`t1`      |
|                                                                       |
|   2:HASH JOIN                                                         |
|   |  join op: INNER JOIN (BUCKET_SHUFFLE)                             |
|   |  equal join conjunct: `test`.`t1` = `test2`.`t2`                  |
|   |  runtime filters: RF000[in] <- `test2`.`t2`                       |
|   |                                                                   |
|   |----3:EXCHANGE                                                     |
|   |                                                                   |
|   0:OlapScanNode                                                      |
|      TABLE: test                                                      |
|      runtime filters: RF000[in] -> `test`.`t1`                        |
|                                                                       |
| PLAN FRAGMENT 2                                                       |
|  OUTPUT EXPRS:                                                        |
|   PARTITION: HASH_PARTITIONED: `default_cluster:ssb`.`test2`.`t2`     |
|                                                                       |
|   1:OlapScanNode                                                      |
|      TABLE: test2                                                     |
+-----------------------------------------------------------------------+
-- 上面`runtime filters`的行显示了`PLAN FRAGMENT 1`的`2:HASH JOIN`生成了ID为RF000的IN p
-- 其中`test2`.`t2`的key values仅在运行时可知,
-- 在`0:OlapScanNode`使用了该IN predicate用于在读取`test`.`t1`时过滤不必要的数据。

SELECT t1 FROM test JOIN test2 where test.t1 = test2.t2;
-- 返回2行结果[3, 4];

-- 通过query的profile（set enable_profile=true;）可以查看查询内部工作的详细信息,
-- 包括每个Runtime Filter是否下推、等待耗时、以及OLAP_SCAN_NODE从prepare到接收到Runtime Fi
RuntimeFilter:in:
     -  HasPushDownToEngine:  true
     -  AWaitTimeCost:  0ns
     -  EffectTimeCost:  2.76ms

-- 此外，在profile的OLAP_SCAN_NODE中还可以查看Runtime Filter下推后的过滤效果和耗时。
     -  RowsVectorPredFiltered:  9.320008M  (9320008)
     -  VectorPredEvalTime:  364.39ms
```

Runtime Filter的规划规则

1. 只支持对 join on clause 中的等值条件生成 Runtime Filter，不包括 Null-safe 条件，因为其可能会过滤掉 join 左表的 null 值。
2. 不支持将 Runtime Filter 下推到 left outer、full outer、anti join 的左表。
3. 不支持 src expr 或 target expr 是常量。
4. 不支持 src expr 和 target expr 相等。
5. 不支持 src expr 的类型等于 `HLL` 或者 `BITMAP` 。
6. 目前仅支持将 Runtime Filter 下推给 OlapScanNode。
7. 不支持 target expr 包含 NULL-checking 表达式，例如 `COALESCE/IFNULL/CASE` ，因为当 outer join 上层其他 join 的 join on clause 包含 NULL-checking 表达式并生成 Runtime Filter 时，将这个 Runtime Filter 下推到 outer join 的左表时可能导致结果不正确。
8. 不支持 target expr 中的列（slot）无法在原始表中找到某个等价列。
9. 不支持列传导，这包含两种情况：
一是例如 join on clause 包含 A.k = B.k and B.k = C.k 时，目前 C.k 只可以下推给 B.k，而不可以下推给 A.k。
二是例如 join on clause 包含 A.a + B.b = C.c，如果 A.a 可以列传导到 B.a，即 A.a 和 B.a 是等价的列，那么可以用 B.a 替换 A.a，然后可以尝试将 Runtime Filter 下推给 B（如果 A.a 和 B.a 不是等价列，则不能下推给 B，因为 target expr 必须与唯一一个 join 左表绑定）。
10. Target expr 和 src expr 的类型必须相等，因为 Bloom Filter 基于 hash，若类型不等则会尝试将 target expr 的类型转换为 src expr 的类型。
11. 不支持 `PlanNode.Conjuncts` 生成的 Runtime Filter 下推，与 HashJoinNode 的 `eqJoinConjuncts` 和 `otherJoinConjuncts` 不同， `PlanNode.Conjuncts` 生成的 Runtime Filter 在测试中发现可能会导致错误的结果，例如 `IN` 子查询转换为 join 时，自动生成的 join on clause 将保存在 `PlanNode.Conjuncts` 中，此时应用 Runtime Filter 可能会导致结果缺少一些行。

查询剖析（Profile）和调优

最近更新时间：2024-06-27 11:12:36

本文档主要介绍 Doris 在查询执行的统计结果。利用这些统计的信息，可以更好的帮助我们了解 Doris 的执行情况，并有针对性的进行相应 **Debug 与调优工作**。

名词解释

FE：Frontend，Doris 的前端节点。负责元数据管理和请求接入。
BE：Backend，Doris 的后端节点。负责查询执行和数据存储。
Fragment：FE 会将具体的 SQL 语句的执行转化为对应的 Fragment 并下发到 BE 进行执行。BE 上执行对应 Fragment，并将结果汇聚返回给 FE。

基本原理

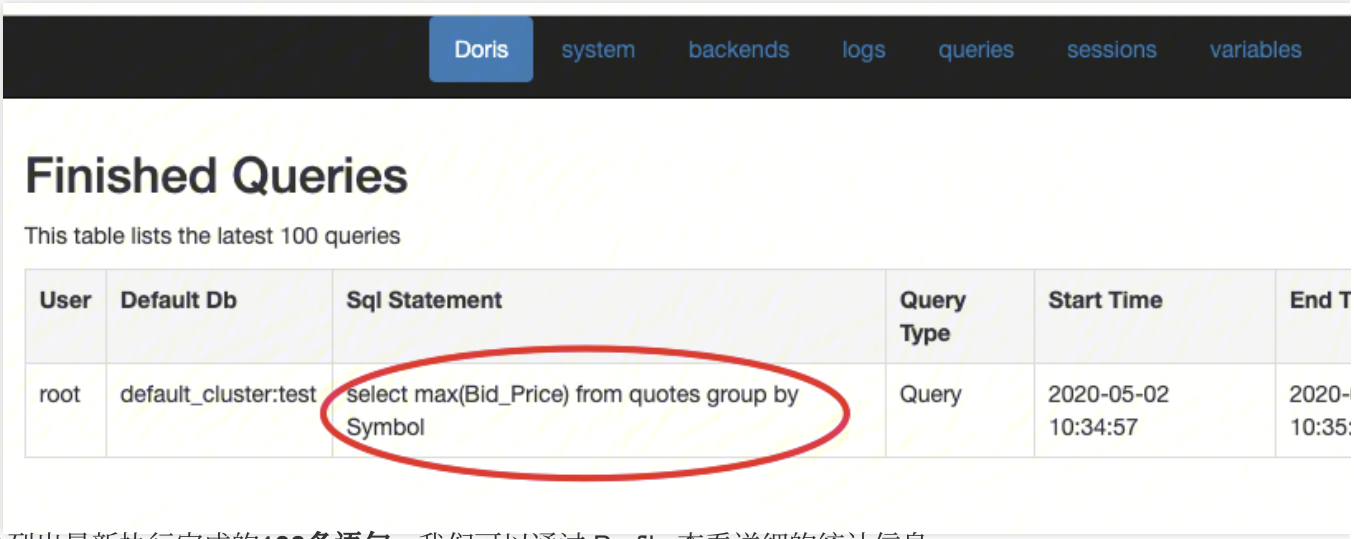
FE 将查询计划拆分成为 Fragment 下发到 BE 进行任务执行。BE 在执行 Fragment 时记录了**运行状态时的统计值**，并将 Fragment 执行的统计信息输出到日志之中。 FE 也可以通过开关将各个 Fragment 记录的这些统计值进行搜集，并在 FE 的 Web 页面上打印结果。

操作流程

通过 Mysql 命令，将 FE 上的 Report 的开关打开。

```
mysql> set enable_profile=true;
```

之后执行对应的 SQL 语句之后（旧版本为 `is_report_success` ），在 FE 的 Web 页面就可以看到对应 SQL 语句执行的 Report 信息：



这里会列出最新执行完成的**100条语句**，我们可以通过 Profile 查看详细的统计信息。

```
Query:
  Summary:
    Query ID: 9664061c57e84404-85ae111b8ba7e83a
    Start Time: 2020-05-02 10:34:57
    End Time: 2020-05-02 10:35:08
    Total: 10s323ms
    Query Type: Query
    Query State: EOF
    Doris Version: trunk
    User: root
    Default Db: default_cluster:test
    Sql Statement: select max(Bid_Price) from quotes group by Symbol
```

这里详尽的列出了**查询的 ID，执行时间，执行语句**等等的总结信息。接下来内容是打印从 BE 收集到的各个 Fragment 的详细信息。

```
    Fragment 0:
      Instance 9664061c57e84404-85ae111b8ba7e83d (host=TNetworkAddress(hostname:192
         - MemoryLimit: 2.00 GB
         - BytesReceived: 168.08 KB
         - PeakUsedReservation: 0.00
         - SendersBlockedTimer: 0ns
         - DeserializeRowBatchTimer: 501.975us
         - PeakMemoryUsage: 577.04 KB
         - RowsProduced: 8.322K (8322)
        EXCHANGE_NODE (id=4):(Active: 10s256ms, % non-child: 99.35%)
           - ConvertRowBatchTime: 180.171us
           - PeakMemoryUsage: 0.00
           - RowsReturned: 8.322K (8322)
           - MemoryUsed: 0.00
           - RowsReturnedRate: 811
```

这里列出了 Fragment 的 ID。

`hostname` 指的是执行 Fragment 的 BE 节点。

`Active：10s270ms` 表示该节点的执行总时间。

`non-child: 0.14%` 表示执行节点自身的执行时间（不包含子节点的执行时间）占总时间的百分比。

`PeakMemoryUsage` 表示 `EXCHANGE_NODE` 内存使用的峰值。

`RowsReturned` 表示 `EXCHANGE_NODE` 结果返回的行数。

`RowsReturnedRate` = `RowsReturned` / `ActiveTime` 。

这三个统计信息在其他 `NODE` 中的含义相同。

后续依次打印子节点的统计信息，**这里可以通过缩进区分节点之间的父子关系**。

Profile参数解析

BE 端收集的统计信息较多，下面列出了各个参数的对应含义：

`Fragment`

AverageThreadTokens: 执行 Fragment 使用线程数目，不包含线程池的使用情况。

Buffer Pool PeakReservation: Buffer Pool 使用的内存的峰值。

MemoryLimit: 查询时的内存限制。

PeakMemoryUsage: 整个 Instance 在查询时内存使用的峰值。

RowsProduced: 处理列的行数。

`BlockMgr`

BlocksCreated: BlockMgr 创建的 Blocks 数目。

BlocksRecycled: 重用的 Blocks 数目。

BytesWritten: 总的落盘写数据量。

MaxBlockSize: 单个 Block 的大小。

TotalReadBlockTime: 读 Block 的总耗时。

`DataStreamSender`

BytesSent: 发送的总数据量 = 接受者 * 发送数据量。

IgnoreRows: 过滤的行数。

LocalBytesSent: 数据在 Exchange 过程中，记录本机节点的自发自收数据量。

OverallThroughput: 总的吞吐量 = BytesSent/时间。

SerializeBatchTime: 发送数据序列化消耗的时间。

UncompressedRowBatchSize: 发送数据压缩前的 RowBatch 的大小。

`ODBC_TABLE_SINK`

NumSentRows: 写入外表的总行数。
TupleConvertTime: 发送数据序列化为 Insert 语句的耗时。
ResultSendTime: 通过 ODBC Driver 写入的耗时。

EXCHANGE_NODE

BytesReceived: 通过网络接收的数据量大小。
MergeGetNext: 当下层节点存在排序时，会在 EXCHANGE NODE 进行统一的归并排序，输出有序结果。该指标记录了 Merge 排序的总耗时，包含了 MergeGetNextBatch 耗时。
MergeGetNextBatch：Merge 节点取数据的耗时，如果为单层 Merge 排序，则取数据的对象为网络队列。若为多层 Merge 排序取数据对象为 Child Merger。
ChildMergeGetNext: 当下层的发送数据的 Sender 过多时，单线程的 Merge 会成为性能瓶颈，Doris 会启动多个 Child Merge 线程并行归并排序。记录了 Child Merge 的排序耗时 该数值是多个线程的累加值。
ChildMergeGetNextBatch: Child Merge 节点从取数据的耗时，如果耗时过大，可能的瓶颈为下层的数据发送节点。
DataArrivalWaitTime: 等待 Sender 发送数据的总时间。
FirstBatchArrivalWaitTime: 等待第一个 batch 从 Sender 获取的时间。
DeserializeRowBatchTimer: 反序列化网络数据的耗时。
SendersBlockedTotalTimer(*): DataStreamRecv 的队列的内存被打满，Sender 端等待的耗时。
ConvertRowBatchTime: 接收数据转为 RowBatch 的耗时。
RowsReturned: 接收行的数目。
RowsReturnedRate: 接收行的速率。

SORT_NODE

InMemorySortTime: 内存之中的排序耗时。
InitialRunsCreated: 初始化排序的趟数（如果内存排序的话，该数为1）。
SortDataSize: 总的排序数据量。
MergeGetNext: MergeSort 从多个 sort_run 获取下一个 batch 的耗时 (仅在落盘时计时）。
MergeGetNextBatch: MergeSort 提取下一个 sort_run的batch 的耗时 (仅在落盘时计时）。
TotalMergesPerformed: 进行外排 merge 的次数。

AGGREGATION_NODE

PartitionsCreated: 聚合查询拆分成 Partition 的个数。
GetResultsTime: 从各个 partition 之中获取聚合结果的时间。
HTResizeTime: HashTable 进行 resize 消耗的时间。
HTResize: HashTable 进行 resize 的次数。
HashBuckets: HashTable 中 Buckets 的个数。
HashBucketsWithDuplicate: HashTable 有 DuplicateNode 的 Buckets 的个数。
HashCollisions: HashTable 产生哈希冲突的次数。
HashDuplicateNodes: HashTable 出现 Buckets 相同 DuplicateNode 的个数。

HashFailedProbe: HashTable Probe 操作失败的次数。
HashFilledBuckets: HashTable 填入数据的 Buckets 数目。
HashProbe: HashTable 查询的次数。
HashTravelLength: HashTable 查询时移动的步数。

HASH_JOIN_NODE

ExecOption: 对右孩子构造HashTable的方式（同步or异步），Join中右孩子可能是表或子查询，左孩子同理。
BuildBuckets: HashTable 中 Buckets 的个数。
BuildRows: HashTable 的行数。
BuildTime: 构造 HashTable 的耗时。
LoadFactor: HashTable 的负载因子（即非空 Buckets 的数量）。
ProbeRows: 遍历左孩子进行 Hash Probe 的行数。
ProbeTime: 遍历左孩子进行 Hash Probe 的耗时，不包括对左孩子 RowBatch 调用 GetNext 的耗时
PushDownComputeTime: 谓词下推条件计算耗时。
PushDownTime: 谓词下推的总耗时，Join 时对满足要求的右孩子，转为左孩子的 in 查询。

CROSS_JOIN_NODE

ExecOption: 对右孩子构造 RowBatchList 的方式（同步 or 异步）。
BuildRows: RowBatchList 的行数（即右孩子的行数）。
BuildTime: 构造 RowBatchList 的耗时。
LeftChildRows: 左孩子的行数。
LeftChildTime: 遍历左孩子，和右孩子求笛卡尔积的耗时，不包括对左孩子 RowBatch 调用 GetNext 的耗时。

UNION_NODE

MaterializeExprsEvaluateTime: Union 两端字段类型不一致时，类型转换表达式计算及物化结果的耗时。

ANALYTIC_EVAL_NODE

EvaluationTime: 分析函数（窗口函数）计算总耗时。
GetNewBlockTime: 初始化时申请一个新的 Block 的耗时，Block 用来缓存 Rows 窗口或整个分区，用于分析函数计算。
PinTime: 后续申请新的 Block 或将写入磁盘的 Block 重新读取回内存的耗时。
UnpinTime: 对暂不需要使用的 Block 或当前操作符内存压力大时，将 Block 的数据刷入磁盘的耗时。

OLAP_SCAN_NODE

`OLAP_SCAN_NODE` 节点负责具体的数据扫描任务。一个 `OLAP_SCAN_NODE` 会生成一个或多个 `OlapScanner` 。每个 Scanner 线程负责扫描部分数据。
查询中的部分或全部谓词条件会推送给 `OLAP_SCAN_NODE` 。这些谓词条件中一部分会继续下推给存储引擎，以便利用存储引擎的索引进行数据过滤。另一部分会保留在 `OLAP_SCAN_NODE` 中，用于过滤从存储引擎中返回的数

据。

`OLAP_SCAN_NODE` 节点的 Profile 通常用于分析数据扫描的效率，依据调用关系分为 `OLAP_SCAN_NODE` 、 `OlapScanner` 、 `SegmentIterator` 三层。

一个典型的 `OLAP_SCAN_NODE` 节点的 Profile 如下。部分指标会因存储格式的不同（V1 或 V2）而有不同含义。

```
OLAP_SCAN_NODE (id=0):(Active: 1.2ms, % non-child: 0.00%)
  - BytesRead: 265.00 B                  # 从数据文件中读取到的数据量。假设读取到了是10个32
  - NumDiskAccess: 1                     # 该 ScanNode 节点涉及到的磁盘数量。
  - NumScanners: 20                      # 该 ScanNode 生成的 Scanner 数量。
  - PeakMemoryUsage: 0.00                # 查询时内存使用的峰值，暂未使用
  - RowsRead: 7                          # 从存储引擎返回到 Scanner 的行数，不包括经 Scan
  - RowsReturned: 7                      # 从 ScanNode 返回给上层节点的行数。
  - RowsReturnedRate: 6.979K /sec        # RowsReturned/ActiveTime
  - TabletCount : 20                     # 该 ScanNode 涉及的 Tablet 数量。
  - TotalReadThroughput: 74.70 KB/sec    # BytesRead除以该节点运行的总时间（从Open到Close
  - ScannerBatchWaitTime: 426.886us      # 用于统计transfer 线程等待scaner 线程返回rowba
  - ScannerWorkerWaitTime: 17.745us      # 用于统计scanner thread 等待线程池中可用工作线程
  OlapScanner:
    - BlockConvertTime: 8.941us          # 将向量化Block转换为行结构的 RowBlock 的耗时。向
    - BlockFetchTime: 468.974us          # Rowset Reader 获取 Block 的时间。
    - ReaderInitTime: 5.475ms            # OlapScanner 初始化 Reader 的时间。V1 中包括组
    - RowsDelFiltered: 0                 # 包括根据 Tablet 中存在的 Delete 信息过滤掉的行
    - RowsPushedCondFiltered: 0          # 根据传递下推的谓词过滤掉的条件，例如 Join 计算中
    - ScanTime: 39.24us                  # 从 ScanNode 返回给上层节点的行数。
    - ShowHintsTime_V1: 0ns              # V2 中无意义。V1 中读取部分数据来进行 ScanRange
    SegmentIterator:
      - BitmapIndexFilterTimer: 779ns    # 利用 bitmap 索引过滤数据的耗时。
      - BlockLoadTime: 415.925us         # SegmentReader(V1) 或 SegmentIterator(V2)
      - BlockSeekCount: 12               # 读取 Segment 时进行 block seek 的次数。
      - BlockSeekTime: 222.556us         # 读取 Segment 时进行 block seek 的耗时。
      - BlocksLoad: 6                    # 读取 Block 的数量
      - CachedPagesNum: 30               # 仅 V2 中，当开启 PageCache 后，命中 Cache 的
      - CompressedBytesRead: 0.00        # V1 中，从文件中读取的解压前的数据大小。V2 中，读
      - DecompressorTimer: 0ns           # 数据解压耗时。
      - IOTimer: 0ns                     # 实际从操作系统读取数据的 IO 时间。
      - IndexLoadTime_V1: 0ns            # 仅 V1 中，读取 Index Stream 的耗时。
      - NumSegmentFiltered: 0            # 在生成 Segment Iterator 时，通过列统计信息和查
      - NumSegmentTotal: 6               # 查询涉及的所有 Segment 数量。
      - RawRowsRead: 7                   # 存储引擎中读取的原始行数。详情见下文。
      - RowsBitmapIndexFiltered: 0       # 仅 V2 中，通过 Bitmap 索引过滤掉的行数。
      - RowsBloomFilterFiltered: 0       # 仅 V2 中，通过 BloomFilter 索引过滤掉的行数。
      - RowsKeyRangeFiltered: 0          # 仅 V2 中，通过 SortkeyIndex 索引过滤掉的行数。
      - RowsStatsFiltered: 0             # V2 中，通过 ZoneMap 索引过滤掉的行数，包含删除
      - RowsConditionsFiltered: 0        # 仅 V2 中，通过各种列索引过滤掉的行数。
      - RowsVectorPredFiltered: 0        # 通过向量化条件过滤操作过滤掉的行数。
      - TotalPagesNum: 30                # 仅 V2 中，读取的总 Page 数量。
```

```
        - UncompressedBytesRead: 0.00      # V1 中为读取的数据文件解压后的大小（如果文件无需解
        - VectorPredEvalTime: 0ns          # 向量化条件过滤操作的耗时。
```

通过 Profile 中数据行数相关指标可以推断谓词条件下推和索引使用情况。以下仅针对 Segment V2 格式数据读取流程中的 Profile 进行说明。Segment V1 格式中，这些指标的含义略有不同。

当读取一个 V2 格式的 Segment 时，若查询存在 key_ranges（前缀 key 组成的查询范围），首先通过 SortkeyIndex 索引过滤数据，过滤的行数记录在 `RowsKeyRangeFiltered` 。

之后，对查询条件中含有 bitmap 索引的列，使用 Bitmap 索引进行精确过滤，过滤的行数记录在 `RowsBitmapIndexFiltered` 。

之后，按查询条件中的等值（eq，in，is）条件，使用 BloomFilter 索引过滤数据，记录在 `RowsBloomFilterFiltered` 。 `RowsBloomFilterFiltered` 的值是 Segment 的总行数（而不是Bitmap索引过滤后的行数）和经过 BloomFilter 过滤后剩余行数的差值，因此 BloomFilter 过滤的数据可能会和 Bitmap 过滤的数据有重叠。

之后，按查询条件和删除条件，使用 ZoneMap 索引过滤数据，记录在 `RowsStatsFiltered` 。

`RowsConditionsFiltered` 是各种索引过滤的行数，包含了 `RowsBloomFilterFiltered` 和 `RowsStatsFiltered` 的值。

至此 Init 阶段完成，Next 阶段删除条件过滤的行数，记录在 `RowsDelFiltered` 。因此删除条件实际过滤的行数，分别记录在 `RowsStatsFiltered` 和 `RowsDelFiltered` 中。

`RawRowsRead` 是经过上述过滤后，最终需要读取的行数。

`RowsRead` 是最终返回给 Scanner 的行数。 `RowsRead` 通常小于 `RawRowsRead` ，是因为从存储引擎返回到 Scanner，可能会经过一次数据聚合。如果 `RawRowsRead` 和 `RowsRead` 差距较大，则说明大量的行被聚合，而聚合可能比较耗时。

`RowsReturned` 是 ScanNode 最终返回给上层节点的行数。 `RowsReturned` 通常也会小于 `RowsRead` 。因为在 Scanner 上会有一些没有下推给存储引擎的谓词条件，会进行一次过滤。如果 `RowsRead` 和 `RowsReturned` 差距较大，则说明很多行在 Scanner 中进行了过滤。这说明很多选择度高的谓词条件并没有推送给存储引擎。而在 Scanner 中的过滤效率会比在存储引擎中过滤效率差。

通过以上指标，可以大致分析出存储引擎处理的行数以及最终过滤后的结果行数大小。通过 `Rows***Filtered` 这组指标，也可以分析查询条件是否下推到了存储引擎，以及不同索引的过滤效果。此外还可以通过以下几个方面进行简单的分析。

`OlapScanner` 下的很多指标，如 `IOTimer` ， `BlockFetchTime` 等都是所有 Scanner 线程指标的累加，因此数值可能会比较大。并且因为 Scanner 线程是异步读取数据的，所以这些累加指标只能反映 Scanner 累加的工作时间，并不直接代表 ScanNode 的耗时。ScanNode 在整个查询计划中的耗时占比为 `Active` 字段记录的值。有时会出现例如 `IOTimer` 有几十秒，而 `Active` 实际只有几秒钟。这种情况通常因为：

`IOTimer` 为多个 Scanner 的累加时间，而 Scanner 数量较多。

上层节点比较耗时。例如上层节点耗时 100秒，而底层 ScanNode 只需 10秒。则反映在 `Active` 的字段可能只有几毫秒。因为在上层处理数据的同时，ScanNode 已经异步的进行了数据扫描并准备好了数据。当上层节点从 ScanNode 获取数据时，可以获取到已经准备好的数据，因此 Active 时间很短。

`NumScanners` 表示 Scanner 提交到线程池的Task个数，由 `RuntimeState` 中的线程池调度，`doris_scanner_thread_pool_thread_num` 和 `doris_scanner_thread_pool_queue_size` 两个参数分别控制线程池的大小和队列长度。线程数过多或过少都会影响查询效率。同时可以用一些汇总指标除以线程数来大致的估算每个线程的耗时。

`TabletCount` 表示需要扫描的 tablet 数量。数量过多可能意味着需要大量的随机读取和数据合并操作。

`UncompressedBytesRead` 间接反映了读取的数据量。如果该数值较大，说明可能有大量的 IO 操作。

`CachedPagesNum` 和 `TotalPagesNum` 可以查看命中 PageCache 的情况。命中率越高，说明 IO 和解压操作耗时越少。

`Buffer pool`

AllocTime: 内存分配耗时。

CumulativeAllocationBytes: 累计内存分配的量。

CumulativeAllocations: 累计的内存分配次数。

PeakReservation: Reservation 的峰值。

PeakUnpinnedBytes: unpin 的内存数据量。

PeakUsedReservation: Reservation 的内存使用量。

ReservationLimit: BufferPool的Reservation 的限制量。

查询缓存（cache）配置

最近更新时间：2024-06-27 11:12:49

需求场景

大部分数据分析场景是写少读多，数据写入一次，多次频繁读取，例如一张报表涉及的维度和指标，数据在凌晨一次性计算好，但每天有数百甚至数千次的页面访问，因此非常适合把结果集缓存起来。在数据分析或BI应用中，存在下面的业务场景：

高并发场景，Doris 可以较好的支持高并发，但单台服务器无法承载太高的 QPS。

复杂图表的看板，复杂的 Dashboard 或者大屏类应用，数据来自多张表，每个页面有数十个查询，虽然每个查询只有数十毫秒，但是总体查询时间会在数秒。

趋势分析，给定日期范围的查询，指标按日显示，例如查询最近7天内的用户数的趋势，这类查询数据量大，查询范围广，查询时间通常需要数十秒。

用户重复查询，如果产品没有防重刷机制，用户因手误或其他原因重复刷新页面，导致提交大量的重复的 SQL。

以上四种场景，在应用层的解决方案，把查询结果放到 Redis 中，周期性的更新缓存或者用户手工刷新缓存，但是这个方案有如下问题：

数据不一致，无法感知数据的更新，导致用户经常看到旧的数据。

命中率低，缓存整个查询结果，如果数据实时写入，缓存频繁失效，命中率低且系统负载较重。

额外成本，引入外部缓存组件，会带来系统复杂度，增加额外成本。

解决方案

本分区缓存策略可以解决上面的问题，优先保证数据一致性，在此基础上细化缓存粒度，提升命中率，因此有如下特点：

用户无需担心数据一致性，通过版本来控制缓存失效，缓存的数据和从 BE 中查询的数据是一致的。

没有额外的组件和成本，缓存结果存储在 BE 的内存中，用户可以根据需要调整缓存内存大小。

实现了两种缓存策略，SQLCache 和 PartitionCache，后者缓存粒度更细。

用一致性哈希解决 BE 节点上下线的问题，BE 中的缓存算法是改进的 LRU。

SQLCache

SQLCache 按 SQL 的签名、查询的表的分区 ID、分区最新版本来存储和获取缓存。三者组合确定一个缓存数据集，任何一个变化了，如 SQL 有变化，如查询字段或条件不一样，或数据更新后版本变化了，会导致命中不了缓存。

如果多张表 Join，使用最近更新的分区 ID 和最新的版本号，如果其中一张表更新了，会导致分区 ID 或版本号不一

样，也一样命中不了缓存。
SQLCache，更适合 T+1更新的场景，凌晨数据更新，首次查询从 BE 中获取结果放入到缓存中，后续相同查询从缓存中获取。实时更新数据也可以使用，但是可能存在命中率低的问题，可以参考如下 PartitionCache。

PartitionCache

设计原理

1. SQL 可以并行拆分，Q = Q1 ∪ Q2 ... ∪ Qn，R= R1 ∪ R2 ... ∪ Rn，Q 为查询语句，R 为结果集。
2. 拆分为只读分区和可更新分区，只读分区缓存，更新分区不缓存。
如上，查询最近7天的每天用户数，如按日期分区，数据只写当天分区，当天之外的其他分区的数据，都是固定不变的，在相同的查询 SQL 下，查询某个不更新分区的指标都是固定的。如下，在2020-03-09当天查询前7天的用户数，2020-03-03至2020-03-07的数据来自缓存，2020-03-08第一次查询来自分区，后续的查询来自缓存，2020-03-09因为当天在不停写入，所以来自分区。
因此，查询 N 天的数据，数据更新最近的 D 天，每天只是日期范围不一样相似的查询，只需要查询 D 个分区即可，其他部分都来自缓存，可以有效降低集群负载，减少查询时间。

```
MySQL [(none)]> SELECT eventdate,count(userid) FROM testdb.appevent WHERE eventdate
+------------+-----------------+
| eventdate  | count(`userid`) |
+------------+-----------------+
| 2020-03-03 |              15 |
| 2020-03-04 |              20 |
| 2020-03-05 |              25 |
| 2020-03-06 |              30 |
| 2020-03-07 |              35 |
| 2020-03-08 |              40 | //第一次来自分区，后续来自缓存
| 2020-03-09 |              25 | //来自分区
+------------+-----------------+
7 rows in set (0.02 sec)
```

在 PartitionCache 中，缓存第一级 Key 是去掉了分区条件后的 SQL 的128位 MD5签名，下面是改写后的待签名的 SQL：

```
SELECT eventdate,count(userid) FROM testdb.appevent GROUP BY eventdate ORDER BY eve
```

缓存的第二级 Key 是查询结果集的分区字段的内容，例如上面查询结果的 eventdate 列的内容，二级 Key 的附属信息是分区的版本号和版本更新时间。
下面演示上面 SQL 在2020-03-09当天第一次执行的流程：
1. 从缓存中获取数据。

```
+------------+-----------------+
| 2020-03-03 |              15 |
```

```
| 2020-03-04 |               20 |
| 2020-03-05 |               25 |
| 2020-03-06 |               30 |
| 2020-03-07 |               35 |
+------------+------------------+
```

2. 从 BE 中获取数据的 SQL 和数据。

```
SELECT eventdate,count(userid) FROM testdb.appevent WHERE eventdate>="2020-03-08" A

+------------+------------------+
| 2020-03-08 |               40 |
+------------+------------------+
| 2020-03-09 |               25 |
+------------+------------------+
```

3. 最后发送给终端的数据。

```
+------------+------------------+
| eventdate  | count(`userid`) |
+------------+------------------+
| 2020-03-03 |               15 |
| 2020-03-04 |               20 |
| 2020-03-05 |               25 |
| 2020-03-06 |               30 |
| 2020-03-07 |               35 |
| 2020-03-08 |               40 |
| 2020-03-09 |               25 |
+------------+------------------+
```

4. 发送给缓存的数据。

```
+------------+------------------+
| 2020-03-08 |               40 |
+------------+------------------+
```

Partition 缓存，适合按日期分区，部分分区实时更新，查询 SQL 较为固定。分区字段也可以是其他字段，但是需要保证只有少量分区更新。

使用限制

只支持 OlapTable，其他存储如 MySQL 的表没有版本信息，无法感知数据是否更新。
只支持按分区字段分组，不支持按其他字段分组，按其他字段分组，该分组数据都有可能被更新，会导致缓存都失效。
只支持结果集的前半部分、后半部分以及全部命中缓存，不支持结果集被缓存数据分割成几个部分。

使用方式

开启 SQLCache

1. 确保 fe.conf 的 cache_enable_sql_mode=true（默认是 true）。

```
vim fe/conf/fe.conf
cache_enable_sql_mode=true
```

2. 在 MySQL 命令行中设置变量。

```
MySQL [(none)]> set [global] enable_sql_cache=true;
```

注意

global 是全局变量，不加指当前会话变量。

开启 PartitionCache

1. 确保 fe.conf 的 cache_enable_partition_mode=true(默认是 true)。

```
vim fe/conf/fe.conf
cache_enable_partition_mode=true
```

2. 在 MySQL 命令行中设置变量。

```
MySQL [(none)]> set [global] enable_partition_cache=true;
```

如果同时开启了两个缓存策略，下面的参数，需要注意一下:

```
cache_last_version_interval_second=900
```

如果分区的最新版本的时间离现在的间隔，大于 cache_last_version_interval_second，则会优先把整个查询结果缓存。如果小于这个间隔，如果符合 PartitionCache 的条件，则按 PartitionCache 数据。

监控

FE 的监控项：

```
query_table             //Query中有表的数量
query_olap_table        //Query中有Olap表的数量
cache_mode_sql          //识别缓存模式为sql的Query数量
cache_hit_sql           //模式为sql的Query命中Cache的数量
query_mode_partition    //识别缓存模式为Partition的Query数量
cache_hit_partition     //通过Partition命中的Query数量
partition_all           //Query中扫描的所有分区
partition_hit           //通过Cache命中的分区数量

Cache命中率      =  (cache_hit_sql + cache_hit_partition) / query_olap_table
```

```
Partition命中率 = partition_hit / partition_all
```

BE 的监控项：

```
query_cache_memory_total_byte       //Cache内存大小
query_query_cache_sql_total_count   //Cache的SQL的数量
query_cache_partition_total_count   //Cache分区数量

SQL平均数据大小       = cache_memory_total / cache_sql_total
Partition平均数据大小 = cache_memory_total / cache_partition_total
```

其他监控：

可以从 Grafana 中查看 BE 节点的 CPU 和内存指标，Query 统计中的 Query Percentile 等指标，配合 Cache 参数的调整来达成业务目标。

优化参数

FE 的配置项 cache_result_max_row_count，查询结果集放入缓存的最大行数，可以根据实际情况调整，但建议不要设置过大，避免过多占用内存，超过这个大小的结果集不会被缓存。

```
vim fe/conf/fe.conf
cache_result_max_row_count=3000
```

BE 最大分区数量 cache_max_partition_count，指每个 SQL 对应的最大分区数，如果是按日期分区，能缓存2年多的数据，假如想保留更长时间的缓存，请把这个参数设置得更大，同时修改 cache_result_max_row_count 的参数。

```
vim be/conf/be.conf
cache_max_partition_count=1024
```

BE 中缓存内存设置，有两个参数 query_cache_max_size 和 query_cache_elasticity_size 两部分组成（单位：MB），内存超过 query_cache_max_size + cache_elasticity_size 会开始清理，并把内存控制到 query_cache_max_size 以下。可以根据 BE 节点数量，节点内存大小，和缓存命中率来设置这两个参数。

```
query_cache_max_size_mb=256
query_cache_elasticity_size_mb=128
```

计算方法：

假如缓存10K 个 Query，每个 Query 缓存1000行，每行是128个字节，分布在10台 BE 上，则每个 BE 需要128M 内存（10K*1000*128/10）。

注意事项

T+1 的数据，目前不支持使用 Partition 缓存。

类似的 SQL，之前查询了2个指标，现在查询3个指标，目前不支持利用2个指标的缓存 。

按日期分区，但是需要按周维度汇总数据，目前不支持 PartitionCache。

生态扩展功能
外表
Hive 外表

最近更新时间：2024-07-04 15:35:17

说明：

本文档展示的内容仅适用于腾讯云数据仓库 TCHouse-D 1.1及以下版本，后续版本建议使用 Multi-Catalog 功能对接外部数据目录。

Hive External Table of Doris 提供了 Doris 直接访问 Hive 外部表的能力，外部表省去了繁琐的数据导入工作，并借助 Doris 本身的 OLAP 的能力来解决 Hive 表的数据分析问题：

1. 支持 Hive 数据源接入Doris。
2. 支持 Doris 与 Hive 数据源中的表联合查询，进行更加复杂的分析操作。
3. 支持 访问开启 Kerberos 的 Hive 数据源。

本文档主要介绍该功能的使用方式和注意事项等。

使用方法

Doris 中创建 Hive 的外表

```
-- 语法
CREATE [EXTERNAL] TABLE table_name (
  col_name col_type [NULL | NOT NULL] [COMMENT "comment"]
) ENGINE=HIVE
[COMMENT "comment"]
PROPERTIES (
  'property_name'='property_value',
  ...
);

-- 例子1：创建 Hive 集群中 hive_db 下的 hive_table 表
CREATE TABLE `t_hive` (
  `k1` int NOT NULL COMMENT "",
  `k2` char(10) NOT NULL COMMENT "",
  `k3` datetime NOT NULL COMMENT "",
  `k5` varchar(20) NOT NULL COMMENT "",
  `k6` double NOT NULL COMMENT ""
) ENGINE=HIVE
COMMENT "HIVE"
PROPERTIES (
```

```
'hive.metastore.uris' = 'thrift://192.168.0.1:9083',
'database' = 'hive_db',
'table' = 'hive_table'
);

-- 例子2：创建 Hive 集群中 hive_db 下的 hive_table 表,HDFS使用HA配置
CREATE TABLE `t_hive` (
  `k1` int NOT NULL COMMENT "",
  `k2` char(10) NOT NULL COMMENT "",
  `k3` datetime NOT NULL COMMENT "",
  `k5` varchar(20) NOT NULL COMMENT "",
  `k6` double NOT NULL COMMENT ""
) ENGINE=HIVE
COMMENT "HIVE"
PROPERTIES (
'hive.metastore.uris' = 'thrift://192.168.0.1:9083',
'database' = 'hive_db',
'table' = 'hive_table',
'dfs.nameservices'='hacluster',
'dfs.ha.namenodes.hacluster'='n1,n2',
'dfs.namenode.rpc-address.hacluster.n1'='192.168.0.1:8020',
'dfs.namenode.rpc-address.hacluster.n2'='192.168.0.2:8020',
'dfs.client.failover.proxy.provider.hacluster'='org.apache.hadoop.hdfs.server.namen
);

-- 例子3：创建 Hive 集群中 hive_db 下的 hive_table 表, HDFS使用HA配置并开启kerberos认证方式
CREATE TABLE `t_hive` (
  `k1` int NOT NULL COMMENT "",
  `k2` char(10) NOT NULL COMMENT "",
  `k3` datetime NOT NULL COMMENT "",
  `k5` varchar(20) NOT NULL COMMENT "",
  `k6` double NOT NULL COMMENT ""
) ENGINE=HIVE
COMMENT "HIVE"
PROPERTIES (
'hive.metastore.uris' = 'thrift://192.168.0.1:9083',
'database' = 'hive_db',
'table' = 'hive_table',
'dfs.nameservices'='hacluster',
'dfs.ha.namenodes.hacluster'='n1,n2',
'dfs.namenode.rpc-address.hacluster.n1'='192.168.0.1:8020',
'dfs.namenode.rpc-address.hacluster.n2'='192.168.0.2:8020',
'dfs.client.failover.proxy.provider.hacluster'='org.apache.hadoop.hdfs.server.namen
'dfs.namenode.kerberos.principal'='hadoop/_HOST@REALM.COM'
'hadoop.security.authentication'='kerberos',
'hadoop.kerberos.principal'='doris_test@REALM.COM',
'hadoop.kerberos.keytab'='/path/to/doris_test.keytab'
```

```
);

-- 例子4：创建 Hive 集群中 hive_db 下的 hive_table 表，Hive数据存储在S3上
CREATE TABLE `t_hive` (
  `k1` int NOT NULL COMMENT "",
  `k2` char(10) NOT NULL COMMENT "",
  `k3` datetime NOT NULL COMMENT "",
  `k5` varchar(20) NOT NULL COMMENT "",
  `k6` double NOT NULL COMMENT ""
) ENGINE=HIVE
COMMENT "HIVE"
PROPERTIES (
'hive.metastore.uris' = 'thrift://192.168.0.1:9083',
'database' = 'hive_db',
'table' = 'hive_table',
'AWS_ACCESS_KEY' = 'your_access_key',
'AWS_SECRET_KEY' = 'your_secret_key',
'AWS_ENDPOINT' = 's3.us-east-1.amazonaws.com',
'AWS_REGION' = 'us-east-1'
);
```

参数说明：

外表列：

列名要于 Hive 表一一对应。

列的顺序需要与 Hive 表一致。

必须包含 Hive 表中的全部列。

Hive 表分区列无需指定，与普通列一样定义即可。

ENGINE 需要指定为 HIVE。

PROPERTIES 属性：

`hive.metastore.uris` ：Hive Metastore 服务地址。

`database` ：挂载 Hive 对应的数据库名。

`table` ：挂载 Hive 对应的表名。

`dfs.nameservices` ：name service名称，与 hdfs-site.xml 保持一致。

`dfs.ha.namenodes.[nameservice ID]` ：namenode 的 ID 列表，与 hdfs-site.xml 保持一致。

`dfs.namenode.rpc-address.[nameservice ID].[name node ID]` ：namenode 的 rpc 地址，数量与 namenode 数量相同，与 hdfs-site.xml 保持一致。

`dfs.client.failover.proxy.provider.[nameservice ID]` ：HDFS 客户端连接活跃 namenode 的 Java 类，通常是 `org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider` 。

访问开启 Kerberos 的 Hive 数据源，需要为 Hive 外表额外配置如下 PROPERTIES 属性：

`hadoop.security.authentication` ：认证方式请设置为 Kerberos，默认为 simple。

`dfs.namenode.kerberos.principal` ：HDFS namenode 服务的 Kerberos 主体。

`hadoop.kerberos.principal` ：设置 Doris 连接 HDFS 时使用的 Kerberos 主体。

`hadoop.kerberos.keytab` ：设置 keytab 本地文件路径。

`yarn.resourcemanager.principal` ： Doris链接Hadoop集群的resource mananger的Kerberos主体。

如果数据存储在类S3系统时，例如腾讯云 COS 等，需要配置下列属性：

`AWS_ACCESS_KEY` ：腾讯云账户的 SecretId。

`AWS_SECRET_KEY` ：腾讯云账户的 SecretKey。

`AWS_ENDPOINT` ：COS 桶的 endpoint，例如：cos.ap-nanjing.myqcloud.com。

`AWS_REGION` ：COS 桶的区域，例如：ap-nanjing。

注意

若要使 Doris 访问开启 Kerberos 认证方式的 Hadoop 集群，需要在 Doris 集群所有运行节点上部署 Kerberos 客户端 kinit，并配置 krb5.conf，填写 KDC 服务信息等。

PROPERTIES 属性 `hadoop.kerberos.keytab` 的值需要指定 keytab 本地文件的绝对路径，并允许 Doris 进程访问该本地文件。

关于 HDFS 集群的配置可以写入 hdfs-site.xml 文件中，该配置文件在 fe 和 be 的 conf 目录下，用户创建 Hive 表时，不需要再填写 HDFS 集群配置的相关信息。

类型匹配

支持的 Hive 列类型与 Doris 对应关系如下表：

Hive	Doris	描述
BOOLEAN	BOOLEAN	
CHAR	CHAR	当前仅支持 UTF8编码
VARCHAR	VARCHAR	当前仅支持 UTF8编码
TINYINT	TINYINT	
SMALLINT	SMALLINT	
INT	INT	
BIGINT	BIGINT	
FLOAT	FLOAT	
DOUBLE	DOUBLE	
DECIMAL	DECIMAL	

DATE	DATE	
TIMESTAMP	DATETIME	Timestamp 转成 Datetime 会损失精度

注意

Hive 表 Schema 变更**不会自动同步**，需要在 Doris 中重建 Hive 外表。

当前 Hive 的存储格式仅支持 Text，Parquet 和 ORC 类型。

当前默认支持的 Hive 版本为 `2.3.7、3.1.2`，未在其他版本进行测试。后续后支持更多版本。

查询用法

完成在 Doris 中建立 Hive 外表后，除了无法使用 Doris 中的数据模型(rollup、预聚合、物化视图等)外，与普通的 Doris OLAP 表并无区别。

```
select * from t_hive where k1 > 1000 and k3 ='term' or k4 like '%doris';
```

ES 外表

最近更新时间：2024-07-04 15:35:48

说明：
本文档展示的内容仅适用于腾讯云数据仓库 TCHouse-D 1.1及以下版本，后续版本建议使用 Multi-Catalog 功能对接外部数据目录。
Doris-On-ES 将 Doris 的分布式查询规划能力和 ES（Elasticsearch）的全文检索能力相结合，提供更完善的 OLAP 分析场景解决方案：
1. ES 中的多 index 分布式 Join 查询。
2. Doris 和 ES 中的表联合查询，更复杂的全文检索过滤。
本文档主要介绍该功能的实现原理、使用方式等。

名词解释

Doris 相关

FE：Frontend，Doris 的前端节点，负责元数据管理和请求接入。
BE：Backend，Doris 的后端节点，负责查询执行和数据存储。

ES 相关

DataNode：ES 的数据存储与计算节点。
MasterNode：ES 的 Master 节点，管理元数据、节点、数据分布等。
scroll：ES内置的数据集游标特性，用来对数据进行流式扫描和过滤。
_source：导入时传入的原始 JSON 格式文档内容。
doc_values：ES/Lucene 中字段的列式存储定义。
keyword：字符串类型字段，ES/Lucene 不会对文本内容进行分词处理。
text：字符串类型字段，ES/Lucene 会对文本内容进行分词处理，分词器需要用户指定，默认为 standard 英文分词器。

使用方法

创建 ES 索引

```
PUT test
{
   "settings": {
      "index": {
```

```
            "number_of_shards": "1",
            "number_of_replicas": "0"
        }
    },
    "mappings": {
        "doc": { // ES 7.x版本之后创建索引时不需要指定type，会有一个默认且唯一的`_doc` type
            "properties": {
                "k1": {
                    "type": "long"
                },
                "k2": {
                    "type": "date"
                },
                "k3": {
                    "type": "keyword"
                },
                "k4": {
                    "type": "text",
                    "analyzer": "standard"
                },
                "k5": {
                    "type": "float"
                }
            }
        }
    }
}
```

ES 索引导入数据

```
OST /_bulk
{"index":{"_index":"test","_type":"doc"}}
{ "k1" : 100, "k2": "2020-01-01", "k3": "Trying out Elasticsearch", "k4": "Trying o
{"index":{"_index":"test","_type":"doc"}}
{ "k1" : 100, "k2": "2020-01-01", "k3": "Trying out Doris", "k4": "Trying out Doris
{"index":{"_index":"test","_type":"doc"}}
{ "k1" : 100, "k2": "2020-01-01", "k3": "Doris On ES", "k4": "Doris On ES", "k5": 1
{"index":{"_index":"test","_type":"doc"}}
{ "k1" : 100, "k2": "2020-01-01", "k3": "Doris", "k4": "Doris", "k5": 10.0}
{"index":{"_index":"test","_type":"doc"}}
{ "k1" : 100, "k2": "2020-01-01", "k3": "ES", "k4": "ES", "k5": 10.0}
```

Doris 中创建 ES 外表

具体建表语法参照 CREATE TABLE。

```
CREATE EXTERNAL TABLE `test` // 不指定schema，自动拉取es mapping进行建表
ENGINE=ELASTICSEARCH
PROPERTIES (
"hosts" = "http://192.168.0.1:8200,http://192.168.0.2:8200",
"index" = "test",
"type" = "doc",
"user" = "root",
"password" = "root"
);

CREATE EXTERNAL TABLE `test` (
  `k1` bigint(20) COMMENT "",
  `k2` datetime COMMENT "",
  `k3` varchar(20) COMMENT "",
  `k4` varchar(100) COMMENT "",
  `k5` float COMMENT ""
) ENGINE=ELASTICSEARCH // ENGINE必须是Elasticsearch
PROPERTIES (
"hosts" = "http://192.168.0.1:8200,http://192.168.0.2:8200",
"index" = "test",
"type" = "doc",
"user" = "root",
"password" = "root"
);
```

参数说明：

参数	说明
hosts	ES 集群地址，可以是一个或多个，也可以是 ES 前端的负载均衡地址
index	对应的 ES 的 index 名字，支持 alias，如果使用 doc_value，需要使用真实的名称
type	index 的 type，ES 7.x 及以后的版本不传此参数
user	ES 集群用户名
password	对应用户的密码信息

ES 7.x 之前的集群请注意在建表的时候选择正确的**索引类型 type**。

认证方式目前仅支持 HTTP Basic 认证，并且需要确保该用户有访问：/_cluster/state/、_nodes/http 等路径和 index 的读权限；集群未开启安全认证，用户名和密码不需要设置。

Doris 表中的列名需要和 ES 中的字段名完全匹配，字段类型应该保持一致。

ENGINE 必须是 **Elasticsearch**。

过滤条件下推

`Doris On ES` 一个重要的功能就是过滤条件的下推：过滤条件下推给 ES，这样只有真正满足条件的数据才会被返回，能够显著的提高查询性能和降低 Doris和 Elasticsearch 的 CPU、memory、IO 使用量。

`enable_new_es_dsl` 代表是否使用新版 DSL 生成逻辑，后续 bug 修复和迭代都在新版 DSL 开发, 默认为 `true` ，可在 `fe.conf` 中进行修改。

下面的操作符（Operators）会被优化成如下 ES Query：

SQL syntax	ES 5.x+ syntax
=	term query
in	terms query
> , < , >= , ⇐	range query
and	bool.filter
or	bool.should
not	bool.must_not
not in	bool.must_not + terms query
is_not_null	exists query
is_null	bool.must_not + exists query
esquery	ES 原生 json 形式的 QueryDSL

数据类型映射

Doris/ES	byte	short	integer	long	float	double	keyword	text	date
tinyint	✓	-	-	-	-	-	-	-	-
smallint	✓	✓	-	-	-	-	-	-	-
int	✓	✓	✓	-	-	-	-	-	-
bigint	✓	✓	✓	✓	-	-	-	-	-
float	-	-	-	-	✓	-	-	-	-
double	-	-	-	-	-	✓	-	-	-
char	-	-	-	-	-	-	✓	✓	-
varchar	-	-	-	-	-	-	✓	✓	-

date	-	-	-	-	-	-	-	-	✓
datetime	-	-	-	-	-	-	-	-	✓

启用列式扫描优化查询速度（enable_docvalue_scan=true）

```
CREATE EXTERNAL TABLE `test` (
  `k1` bigint(20) COMMENT "",
  `k2` datetime COMMENT "",
  `k3` varchar(20) COMMENT "",
  `k4` varchar(100) COMMENT "",
  `k5` float COMMENT ""
) ENGINE=ELASTICSEARCH
PROPERTIES (
"hosts" = "http://192.168.0.1:8200,http://192.168.0.2:8200",
"index" = "test",
"user" = "root",
"password" = "root",
"enable_docvalue_scan" = "true"
);
```

参数说明：

参数	说明
enable_docvalue_scan	是否开启通过 ES/Lucene 列式存储获取查询字段的值，默认为 false

开启后 Doris 从 ES 中获取数据会遵循以下两个原则：

尽力而为：自动探测要读取的字段是否开启列式存储（doc_value: true），如果获取的字段全部有列存，Doris 会从列式存储中获取所有字段的值。

自动降级：如果要获取的字段只要有一个字段没有列存，所有字段的值都会从行存 `_source` 中解析获取。

优势

默认情况下，Doris On ES 会从行存也就是 `_source` 中获取所需的所有列，`_source` 的存储采用的行式+ json 的形式存储，在批量读取性能上要劣于列式存储，尤其在只需要少数列的情况下尤为明显，只获取少数列的情况下，docvalue 的性能大约是 `_source` 性能的十几倍。

注意

1. `text` 类型的字段在 ES 中是没有列式存储，因此如果要获取的字段值有 `text` 类型字段会自动降级为从 `_source` 中获取。
2. 在获取的字段数量过多的情况下(`>= 25`)，从 `docvalue` 中获取字段值的性能会和从 `_source` 中获取字段值基本一样。

探测 keyword 类型字段（enable_keyword_sniff=true）

```
CREATE EXTERNAL TABLE `test` (
  `k1` bigint(20) COMMENT "",
  `k2` datetime COMMENT "",
  `k3` varchar(20) COMMENT "",
  `k4` varchar(100) COMMENT "",
  `k5` float COMMENT ""
) ENGINE=ELASTICSEARCH
PROPERTIES (
"hosts" = "http://192.168.0.1:8200,http://192.168.0.2:8200",
"index" = "test",
"user" = "root",
"password" = "root",
"enable_keyword_sniff" = "true"
);
```

参数说明：

参数	说明
enable_keyword_sniff	是否对 ES 中字符串类型分词类型（**text**） `fields` 进行探测，获取额外的未分词（**keyword**）字段名（multi-fields 机制）

在 ES 中可以不建立 index 直接进行数据导入，这时候 ES 会自动创建一个新的索引，针对字符串类型的字段 ES 会创建一个既有 `text` 类型的字段又有 `keyword` 类型的字段，这就是 ES 的 multi fields 特性，mapping 如下：

```
"k4": {
   "type": "text",
   "fields": {
      "keyword": {
         "type": "keyword",
         "ignore_above": 256
      }
   }
}
```

对 k4进行条件过滤时会将查询转换为 ES 的 TermQuery。

SQL 过滤条件：

```
k4 = "Doris On ES"
```

转换成 ES 的 query DSL 为：

```
"term" : {
    "k4": "Doris On ES"
```

```
}
```

因为 k4 的第一字段类型为 `text`，在数据导入的时候就会根据 k4 设置的分词器（如果没有设置，就是 standard 分词器）进行分词处理得到 doris、on、es 三个 Term，如下 ES analyze API 分析：

```
POST /_analyze
{
  "analyzer": "standard",
  "text": "Doris On ES"
}
```

分词的结果是：

```
{
   "tokens": [
      {
         "token": "doris",
         "start_offset": 0,
         "end_offset": 5,
         "type": "<ALPHANUM>",
         "position": 0
      },
      {
         "token": "on",
         "start_offset": 6,
         "end_offset": 8,
         "type": "<ALPHANUM>",
         "position": 1
      },
      {
         "token": "es",
         "start_offset": 9,
         "end_offset": 11,
         "type": "<ALPHANUM>",
         "position": 2
      }
   ]
}
```

查询时使用的是：

```
"term" : {
    "k4": "Doris On ES"
}
```

`Doris On ES` 这个 term 匹配不到词典中的任何 term，不会返回任何结果，而启用 `enable_keyword_sniff: true` 会自动将 `k4 = "Doris On ES"` 转换成 `k4.keyword = "Doris On ES"` 来完全匹配 SQL 语义，转换后的 ES query DSL 为：

```
"term" : {
    "k4.keyword": "Doris On ES"
}
```

`k4.keyword` 的类型是 `keyword`，数据写入 ES 中是一个完整的 term，所以可以匹配。

开启节点自动发现

默认为 true（nodes_discovery=true）。

```
CREATE EXTERNAL TABLE `test` (
  `k1` bigint(20) COMMENT "",
  `k2` datetime COMMENT "",
  `k3` varchar(20) COMMENT "",
  `k4` varchar(100) COMMENT "",
  `k5` float COMMENT ""
) ENGINE=ELASTICSEARCH
PROPERTIES (
"hosts" = "http://192.168.0.1:8200,http://192.168.0.2:8200",
"index" = "test",
"user" = "root",
"password" = "root",
"nodes_discovery" = "true"
);
```

参数说明：

参数	说明
nodes_discovery	是否开启 EST 节点发现，默认为 true

当配置为 true 时，Doris 将从 ES 找到所有可用的相关数据节点(在上面分配的分片)。如果 ES 数据节点的地址没有被 Doris BE 访问，则设置为 false。ES 群部署在与公共 Internet 隔离的内网，用户通过代理访问。

ES 集群是否开启 HTTPS 访问模式

如果开启应设置为 `true`，默认为 false(http_ssl_enabled=true) 。

```
CREATE EXTERNAL TABLE `test` (
  `k1` bigint(20) COMMENT "",
  `k2` datetime COMMENT "",
  `k3` varchar(20) COMMENT "",
  `k4` varchar(100) COMMENT "",
```

```
  `k5` float COMMENT ""
) ENGINE=ELASTICSEARCH
PROPERTIES (
"hosts" = "http://192.168.0.1:8200,http://192.168.0.2:8200",
"index" = "test",
"user" = "root",
"password" = "root",
"http_ssl_enabled" = "true"
);
```

参数说明：

参数	说明
http_ssl_enabled	ES 集群是否开启 HTTPS 访问模式

目前的 fe/be 实现方式为信任所有，这是临时解决方案，后续会使用真实的用户配置证书。

查询用法

完成在 Doris 中建立 ES 外表后，除了无法使用 Doris 中的数据模型（rollup、预聚合、物化视图等）外并无区别。

基本查询

```
select * from es_table where k1 > 1000 and k3 ='term' or k4 like 'fu*z_'
```

扩展的 esquery(field, QueryDSL)

通过 `esquery(field, QueryDSL)` 函数将一些无法用 SQL 表述的 query 如 match_phrase、geoshape 等下推给 ES 进行过滤处理， `esquery` 的第一个列名参数用于关联 `index` ，第二个参数是ES的基本 `Query DSL` 的 json 表述，使用花括号 `{}` 包含，json 的 `root key` 有且只能有一个，如 match_phrase、geo_shape、bool 等。

match_phrase 查询：

```
select * from es_table where esquery(k4, '{
        "match_phrase": {
           "k4": "doris on es"
        }
    }');
```

geo相关查询：

```
select * from es_table where esquery(k4, '{
      "geo_shape": {
         "location": {
            "shape": {
```

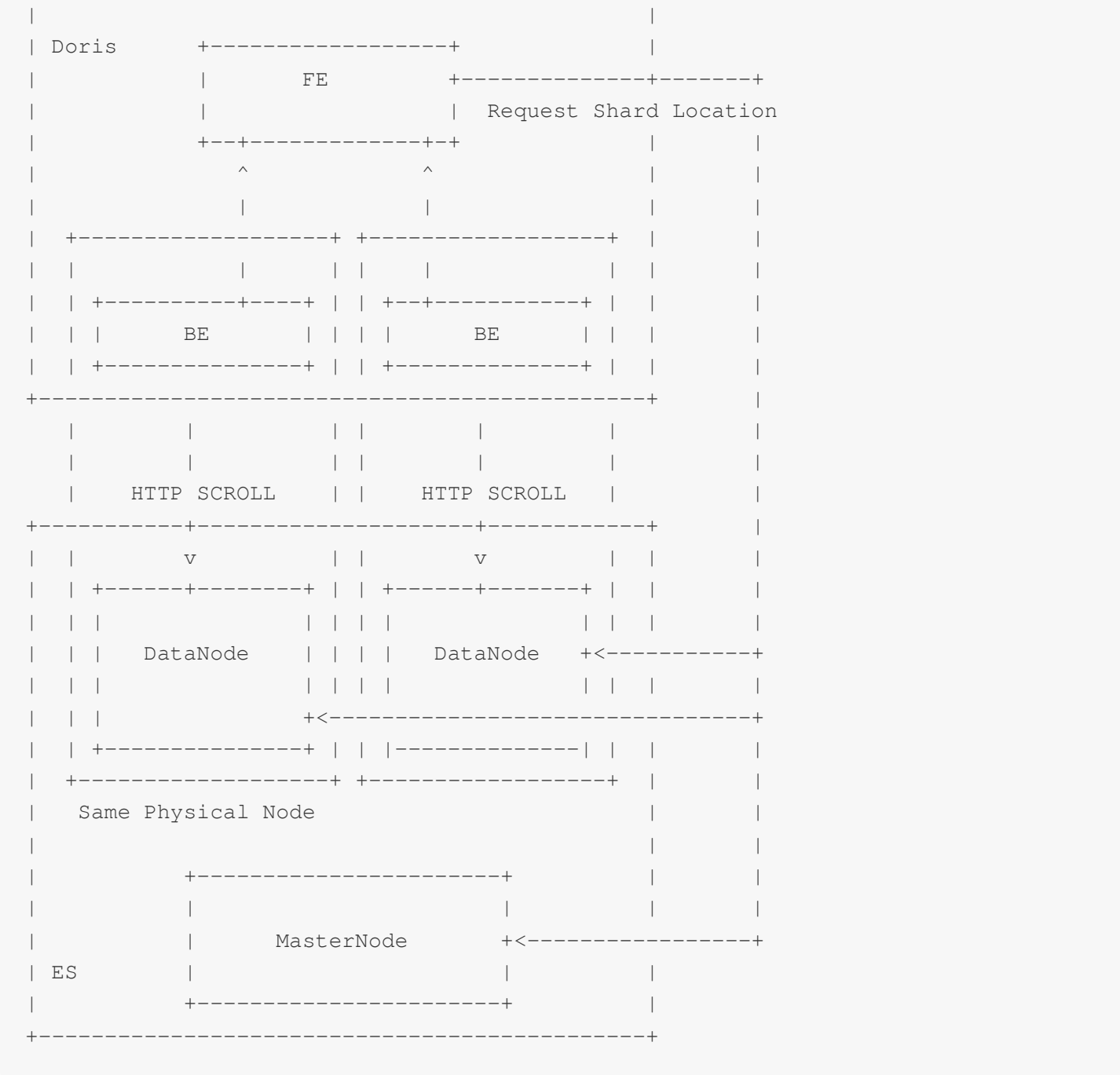
```
                "type": "envelope",
                "coordinates": [
                    [
                        13,
                        53
                    ],
                    [
                        14,
                        52
                    ]
                ]
            },
            "relation": "within"
        }
    }
}');
```

bool 查询：

```
select * from es_table where esquery(k4, ' {
        "bool": {
            "must": [
                {
                    "terms": {
                        "k1": [
                            11,
                            12
                        ]
                    }
                },
                {
                    "terms": {
                        "k2": [
                            100
                        ]
                    }
                }
            ]
        }
    }');
```

原理

```
+-----------------------------------------------+
```



1. 创建 ES 外表后，FE 会请求建表指定的主机，获取所有节点的 HTTP 端口信息以及 index 的 shard 分布信息等，如果请求失败会顺序遍历 host 列表直至成功或完全失败。
2. 查询时会根据 FE 得到的一些节点信息和 index 的元数据信息，生成查询计划并发给对应的 BE 节点。
3. BE 节点会根据**就近原则**即优先请求本地部署的 ES 节点，BE 通过 `HTTP Scroll` 方式流式的从 ES index 的每个分片中并发的从 `_source` 或 `docvalue` 中获取数据。
4. Doris 计算完结果后，返回给用户。

最佳实践

时间类型字段使用建议

在 ES 中，时间类型的字段使用十分灵活，但是在 Doris On ES 中如果对时间类型字段的类型设置不当，则会造成过滤条件无法下推。
创建索引时对时间类型格式的设置做最大程度的格式兼容：

```
"dt": {
    "type": "date",
    "format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch_millis"
}
```

在 Doris 中建立该字段时建议设置为 `date` 或 `datetime`，也可以设置为 `varchar` 类型，使用如下 SQL 语句都可以直接将过滤条件下推至 ES：

```
select * from doe where k2 > '2020-06-21';

select * from doe where k2 < '2020-06-21 12:00:00';

select * from doe where k2 < 1593497011;

select * from doe where k2 < now();

select * from doe where k2 < date_format(now(), '%Y-%m-%d');
```

注意在 ES 中如果不对时间类型的字段设置 `format`，默认的时间类型字段格式为：

```
strict_date_optional_time||epoch_millis
```

导入到 ES 的日期字段如果是时间戳需要转换成 `ms`，ES 内部处理时间戳都是按照 `ms` 进行处理的，否则 Doris On ES 会出现显示错误。

获取 ES 元数据字段 `_id`

导入文档在不指定 `_id` 的情况下 ES 会给每个文档分配一个全局唯一的 `_id` 即主键，用户也可以在导入时为文档指定一个含有特殊业务意义的 `_id`；如果需要在 Doris On ES 中获取该字段值，建表时可以增加类型为 `varchar` 的 `_id` 字段：

```
CREATE EXTERNAL TABLE `doe` (
  `_id` varchar COMMENT "",
  `city`  varchar COMMENT ""
) ENGINE=ELASTICSEARCH
PROPERTIES (
"hosts" = "http://127.0.0.1:8200",
"user" = "root",
"password" = "root",
"index" = "doe"
}
```

注意

`_id` 字段的过滤条件仅支持 `=` 和 `in` 两种。

`_id` 字段只能是 `varchar` 类型。

常见问题

1. Doris On ES 对 ES 的版本要求。

ES 主版本大于5，ES 在2.x 之前和5.x 之后数据的扫描方式不同，目前仅支持5.x 之后的。

2. 是否支持 X-Pack 认证的 ES 集群？

支持所有使用 HTTP Basic 认证方式的 ES 集群。

3. 一些查询比请求 ES 慢很多？

是，例如_count 相关的 query 等，ES 内部会直接读取满足条件的文档个数相关的元数据，不需要对真实的数据进行过滤。

4. 聚合操作是否可以下推？

目前 Doris On ES 不支持聚合操作如 sum，avg，min/max 等下推，计算方式是批量流式的从 ES 获取所有满足条件的文档，然后在 Doris 中进行计算。

Hudi 外表

最近更新时间：2024-07-04 15:37:29

说明：

本文档展示的内容仅适用于腾讯云数据仓库 TCHouse-D 1.1及以下版本，后续版本建议使用 Multi-Catalog 功能对接外部数据目录。

Hudi External Table of Doris 提供了腾讯云数据仓库 TCHouse-D 直接访问 Hudi 外部表的能力，外部表省去了繁琐的数据导入工作，并借助 Doris 本身的 OLAP 的能力来解决 Hudi 表的数据分析问题：

1. 支持 Hudi 数据源接入 Doris。
2. 支持 Doris 与 Hive 数据源 Hudi 中的表联合查询，进行更加复杂的分析操作。

本文档主要介绍该功能的使用方式和注意事项等。

使用方法

Doris 中创建 Hudi 的外表

可以通过以下两种方式在 Doris 中创建 Hudi 外表。建外表时无需声明表的列定义，Doris 可以在查询时从 HiveMetaStore 中获取列信息。

1. 创建一个单独的外表，用于挂载 Hudi 表。具体相关语法，可以通过 CREATE TABLE 查看。

```
-- 语法
CREATE [EXTERNAL] TABLE table_name
[(column_definition1[, column_definition2, ...])]
ENGINE = HUDI
[COMMENT "comment"]
PROPERTIES (
"hudi.database" = "hudi_db_in_hive_metastore",
"hudi.table" = "hudi_table_in_hive_metastore",
"hudi.hive.metastore.uris" = "thrift://127.0.0.1:9083"
);

-- 例子：挂载 HiveMetaStore 中 hudi_db_in_hive_metastore 下的 hudi_table_in_hive_metas
CREATE TABLE `t_hudi`
ENGINE = HUDI
PROPERTIES (
"hudi.database" = "hudi_db_in_hive_metastore",
"hudi.table" = "hudi_table_in_hive_metastore",
"hudi.hive.metastore.uris" = "thrift://127.0.0.1:9083"
);

-- 例子：挂载时指定schema
```

```
CREATE TABLE `t_hudi` (
     `id` int NOT NULL COMMENT "id number",
     `name` varchar(10) NOT NULL COMMENT "user name"
) ENGINE = HUDI
PROPERTIES (
"hudi.database" = "hudi_db_in_hive_metastore",
"hudi.table" = "hudi_table_in_hive_metastore",
"hudi.hive.metastore.uris" = "thrift://127.0.0.1:9083"
);
```

参数说明：

外表列：

可以不指定列名，这时查询时会从 HiveMetaStore 中获取列信息，推荐这种建表方式。

指定列名时指定的列名要在 Hudi 表中存在。

ENGINE 需要指定为 HUDI。

PROPERTIES 属性：

`hudi.hive.metastore.uris` ：Hive Metastore 服务地址。

`hudi.database` ：挂载 Hudi 对应的数据库名。

`hudi.table` ：挂载 Hudi 对应的表名。

展示表结构

展示表结构可以通过 `SHOW CREATE TABLE` 查看。

类型匹配

支持的 Hudi 列类型与 Doris 对应关系如下表：

Hudi	Doris	描述
BOOLEAN	BOOLEAN	-
INTEGER	INT	-
LONG	BIGINT	-
FLOAT	FLOAT	-
DOUBLE	DOUBLE	-
DATE	DATE	-
TIMESTAMP	DATETIME	Timestamp 转成 Datetime 会损失精度

STRING	STRING	-
UUID	VARCHAR	使用 VARCHAR 来代替
DECIMAL	DECIMAL	-
TIME	-	不支持
FIXED	-	不支持
BINARY	-	不支持
STRUCT	-	不支持
LIST	-	不支持
MAP	-	不支持

注意

当前默认支持的 Hudi 版本为 0.10.0，未在其他版本进行测试。

查询用法

完成在 Doris 中建立 Hudi 外表后，除了无法使用 Doris 中的数据模型(rollup、预聚合、物化视图等)外，与普通的 Doris OLAP 表并无区别。

```
select * from t_hudi where k1 > 1000 and k3 ='term' or k4 like '%doris';
```

Iceberg 外表

最近更新时间：2024-07-04 15:38:16

说明：

本文档展示的内容仅适用于腾讯云数据仓库 TCHouse-D 1.1及以下版本，后续版本建议使用 Multi-Catalog功能对接外部数据目录。

Iceberg External Table of Doris 提供了腾讯云数据仓库 TCHouse-D 直接访问 Iceberg 外部表的能力，外部表省去了繁琐的数据导入工作，并借助 Doris 本身的 OLAP 的能力来解决 Iceberg 表的数据分析问题：

1. 支持 Iceberg 数据源接入 Doris。
2. 支持 Doris 与 Iceberg 数据源中的表联合查询，进行更加复杂的分析操作。

本文档主要介绍该功能的使用方式和注意事项等。

使用方法

Doris 中创建 Iceberg 的外表

可以通过以下两种方式在 Doris 中创建 Iceberg 外表。建外表时无需声明表的列定义，Doris 可以根据 Iceberg 中表的列定义自动转换。

1. 创建一个单独的外表，用于挂载 Iceberg 表。具体相关语法，可以通过 CREATE TABLE 查看。

```
-- 语法
CREATE [EXTERNAL] TABLE table_name
ENGINE = ICEBERG
[COMMENT "comment"]
PROPERTIES (
"iceberg.database" = "iceberg_db_name",
"iceberg.table" = "icberg_table_name",
"iceberg.hive.metastore.uris"  =  "thrift://192.168.0.1:9083",
"iceberg.catalog.type"  =  "HIVE_CATALOG"
);


-- 例子1：挂载 Iceberg 中 iceberg_db 下的 iceberg_table
CREATE TABLE `t_iceberg`
ENGINE = ICEBERG
PROPERTIES (
"iceberg.database" = "iceberg_db",
"iceberg.table" = "iceberg_table",
"iceberg.hive.metastore.uris"  =  "thrift://192.168.0.1:9083",
"iceberg.catalog.type"  =  "HIVE_CATALOG"
);
```

```
-- 例子2：挂载 Iceberg 中 iceberg_db 下的 iceberg_table，HDFS开启HA
CREATE TABLE `t_iceberg`
ENGINE = ICEBERG
PROPERTIES (
"iceberg.database" = "iceberg_db",
"iceberg.table" = "iceberg_table",
"iceberg.hive.metastore.uris"  =  "thrift://192.168.0.1:9083",
"iceberg.catalog.type"  =  "HIVE_CATALOG",
"dfs.nameservices"="HDFS8000463",
"dfs.ha.namenodes.HDFS8000463"="nn2,nn1",
"dfs.namenode.rpc-address.HDFS8000463.nn2"="172.21.16.5:4007",
"dfs.namenode.rpc-address.HDFS8000463.nn1"="172.21.16.26:4007",
"dfs.client.failover.proxy.provider.HDFS8000463"="org.apache.hadoop.hdfs.server.nam
);
```

2. 创建一个 Iceberg 数据库，用于挂载远端对应 Iceberg 数据库，同时挂载该 database 下的所有 table。具体相关语法，可以通过 CREATE DATABASE 查看。

```
-- 语法
CREATE DATABASE db_name
[COMMENT "comment"]
PROPERTIES (
"iceberg.database" = "iceberg_db_name",
"iceberg.hive.metastore.uris" = "thrift://192.168.0.1:9083",
"iceberg.catalog.type" = "HIVE_CATALOG"
);

-- 例子：挂载 Iceberg 中的 iceberg_db，同时挂载该 db 下的所有 table
CREATE DATABASE `iceberg_test_db`
PROPERTIES (
"iceberg.database" = "iceberg_db",
"iceberg.hive.metastore.uris" = "thrift://192.168.0.1:9083",
"iceberg.catalog.type" = "HIVE_CATALOG"
);
```

`iceberg_test_db` 中的建表进度可以通过 `HELP SHOW TABLE CREATION` 查看。

也可以根据自己的需求明确指定列定义来创建 Iceberg 外表。

1. 创一个 Iceberg 外表

```
-- 语法
CREATE [EXTERNAL] TABLE table_name (
     col_name col_type [NULL | NOT NULL] [COMMENT "comment"]
) ENGINE = ICEBERG
[COMMENT "comment"]
```

```
PROPERTIES (
"iceberg.database" = "iceberg_db_name",
"iceberg.table" = "icberg_table_name",
"iceberg.hive.metastore.uris"  =  "thrift://192.168.0.1:9083",
"iceberg.catalog.type"  =  "HIVE_CATALOG"
);

-- 例子1：挂载 Iceberg 中 iceberg_db 下的 iceberg_table
CREATE TABLE `t_iceberg` (
     `id` int NOT NULL COMMENT "id number",
     `name` varchar(10) NOT NULL COMMENT "user name"
) ENGINE = ICEBERG
PROPERTIES (
"iceberg.database" = "iceberg_db",
"iceberg.table" = "iceberg_table",
"iceberg.hive.metastore.uris"  =  "thrift://192.168.0.1:9083",
"iceberg.catalog.type"  =  "HIVE_CATALOG"
);

-- 例子2：挂载 Iceberg 中 iceberg_db 下的 iceberg_table, HDFS开启HA
CREATE TABLE `t_iceberg` (
     `id` int NOT NULL COMMENT "id number",
     `name` varchar(10) NOT NULL COMMENT "user name"
) ENGINE = ICEBERG
PROPERTIES (
"iceberg.database" = "iceberg_db",
"iceberg.table" = "iceberg_table",
"iceberg.hive.metastore.uris"  =  "thrift://192.168.0.1:9083",
"iceberg.catalog.type"  =  "HIVE_CATALOG",
"dfs.nameservices"="HDFS8000463",
"dfs.ha.namenodes.HDFS8000463"="nn2,nn1",
"dfs.namenode.rpc-address.HDFS8000463.nn2"="172.21.16.5:4007",
"dfs.namenode.rpc-address.HDFS8000463.nn1"="172.21.16.26:4007",
"dfs.client.failover.proxy.provider.HDFS8000463"="org.apache.hadoop.hdfs.server.nam
);
```

参数说明：

外表列

列名要于 Iceberg 表一一对应。

列的顺序需要与 Iceberg 表一致。

ENGINE 需要指定为 ICEBERG。

PROPERTIES 属性：

`iceberg.hive.metastore.uris` ：Hive Metastore 服务地址。

`iceberg.database` ：挂载 Iceberg 对应的数据库名。

`iceberg.table` ：挂载 Iceberg 对应的表名，挂载 Iceberg database 时无需指定。
`iceberg.catalog.type` ：Iceberg 中使用的 catalog 方式，默认为 `HIVE_CATALOG` ，当前仅支持该方式，后续会支持更多的 Iceberg catalog 接入方式。

展示表结构

展示表结构可以通过 `SHOW CREATE TABLE` 查看。

同步挂载

当 Iceberg 表 Schema 发生变更时，可以通过 `REFRESH` 命令手动同步，该命令会将 Doris 中的 Iceberg 外表删除重建，具体帮助可以通过 `HELP REFRESH` 查看。

```
-- 同步 Iceberg 表
REFRESH TABLE t_iceberg;

-- 同步 Iceberg 数据库
REFRESH DATABASE iceberg_test_db;
```

类型匹配

支持的 Iceberg 列类型与 Doris 对应关系如下表：

Iceberg	Doris	描述
BOOLEAN	BOOLEAN	-
INTEGER	INT	-
LONG	BIGINT	-
FLOAT	FLOAT	-
DOUBLE	DOUBLE	-
DATE	DATE	-
TIMESTAMP	DATETIME	Timestamp 转成 Datetime 会损失精度
STRING	STRING	-
UUID	VARCHAR	使用 VARCHAR 来代替
DECIMAL	DECIMAL	-
TIME	-	不支持

FIXED	-	不支持
BINARY	-	不支持
STRUCT	-	不支持
LIST	-	不支持
MAP	-	不支持

注意

Iceberg 表 Schema 变更**不会自动同步**，需要在 Doris 中通过 `REFRESH` 命令同步 Iceberg 外表或数据库。

当前默认支持的 Iceberg 版本为 0.12.0、0.13.2，未在其他版本进行测试。后续会支持更多版本。

查询用法

完成在 Doris 中建立 Iceberg 外表后，除了无法使用 Doris 中的数据模型(rollup、预聚合、物化视图等)外，与普通的 Doris OLAP 表并无区别。

```
select * from t_iceberg where k1 > 1000 and k3 ='term' or k4 like '%doris';
```

相关系统配置

FE 配置

下面几个配置属于 Iceberg 外表系统级别的配置，可以通过修改 `fe.conf` 来配置，也可以通过 `ADMIN SET CONFIG` 来配置。

`iceberg_table_creation_strict_mode`

创建 Iceberg 表默认开启 strict mode。strict mode 是指对 Iceberg 表的列类型进行严格过滤，如果有 Doris 目前不支持的数据类型，则创建外表失败。

`iceberg_table_creation_interval_second`

自动创建 Iceberg 表的后台任务执行间隔，默认为 10s。

`max_iceberg_table_creation_record_size`

Iceberg 表创建记录保留的最大值，默认为 2000。仅针对创建 Iceberg 数据库记录。

ODBC 外表

最近更新时间：2024-07-04 15:39:03

说明：

本文档展示的内容仅适用于腾讯云数据仓库 TCHouse-D 1.1及以下版本，后续版本建议使用 Multi-Catalog 功能对接外部数据目录。

ODBC External Table 提供了腾讯云数据仓库 TCHouse-D 通过数据库访问的标准接口（ODBC）来访问外部表，外部表省去了繁琐的数据导入工作，让腾讯云数据仓库 TCHouse-D 可以具有了访问各式数据库的能力，并借助腾讯云数据仓库 TCHouse-D 本身的 OLAP 的能力来解决外部表的数据分析问题：

支持各种数据源接入腾讯云数据仓库 TCHouse-D。

支持腾讯云数据仓库 TCHouse-D 与各种数据源中的表联合查询，进行更加复杂的分析操作。

通过 insert into 将腾讯云数据仓库 TCHouse-D 执行的查询结果写入外部的数据源。

本文档主要介绍该功能的实现原理、使用方式等。

使用方法

腾讯云数据仓库 TCHouse-D 中创建 ODBC 的外表

具体建表语法参照：CREATE TABLE。

不使用 Resource 创建 ODBC 的外表

```
CREATE EXTERNAL TABLE `baseall_oracle` (
  `k1` decimal(9, 3) NOT NULL COMMENT "",
  `k2` char(10) NOT NULL COMMENT "",
  `k3` datetime NOT NULL COMMENT "",
  `k5` varchar(20) NOT NULL COMMENT "",
  `k6` double NOT NULL COMMENT ""
) ENGINE=ODBC
COMMENT "ODBC"
PROPERTIES (
"host" = "192.168.0.1",
"port" = "8086",
"user" = "test",
"password" = "test",
"database" = "test",
"table" = "baseall",
"driver" = "Oracle 19 ODBC driver",
"odbc_type" = "oracle"
);
```

通过 ODBC_Resource 来创建 ODBC 外表 (推荐使用的方式)

```
CREATE EXTERNAL RESOURCE `oracle_odbc`
PROPERTIES (
"type" = "odbc_catalog",
"host" = "192.168.0.1",
"port" = "8086",
"user" = "test",
"password" = "test",
"database" = "test",
"odbc_type" = "oracle",
"driver" = "Oracle 19 ODBC driver"
);

CREATE EXTERNAL TABLE `baseall_oracle` (
  `k1` decimal(9, 3) NOT NULL COMMENT "",
  `k2` char(10) NOT NULL COMMENT "",
  `k3` datetime NOT NULL COMMENT "",
  `k5` varchar(20) NOT NULL COMMENT "",
  `k6` double NOT NULL COMMENT ""
) ENGINE=ODBC
COMMENT "ODBC"
PROPERTIES (
"odbc_catalog_resource" = "oracle_odbc",
"database" = "test",
"table" = "baseall"
);
```

参数说明：

参数	说明
hosts	外表数据库的 IP 地址
driver	ODBC 外表的 Driver名，该名字需要和 be/conf/odbcinst.ini 中的 Driver 名一致。
odbc_type	外表数据库的类型，当前支持 Oracle，Mysql，Postgresql
user	外表数据库的用户名
password	对应用户的密码信息
charset	数据库连接使用的字符集

说明

`PROPERTIES` 中除了可以添加上述参数之外，还支持每个数据库的 ODBC Driver 实现的专用参数，例如 Mysql 的 `sslverify` 等。

ODBC Driver 的安装和配置

各大主流数据库都会提供 ODBC 的访问 Driver，用户可以参照各数据库官方推荐的方式安装对应的 ODBC Driver lib 库。

安装完成之后，查找对应的数据库的Driver Lib库的路径，并且修改 be/conf/odbcinst.ini 的配置：

```
[MySQL Driver]
Description     = ODBC for MySQL
Driver          = /usr/lib64/libmyodbc8w.so
FileUsage       = 1
```

上述配置 `[]` 里的对应的是 Driver 名，在建立外部表时需要保持外部表的 Driver 名和配置文件之中的一致。

`Driver=` 这个要根据实际 BE 安装 Driver 的路径来填写，本质上就是一个动态库的路径，这里需要保证该动态库的前置依赖都被满足。

注意

切记，这里要求所有的 BE 节点都安装上相同的 Driver，并且安装路径相同，同时有相同的 be/conf/odbcinst.ini 的配置。

查询用法

完成建立 ODBC 外表后，除了无法使用 Doris 中的数据模型（rollup、预聚合、物化视图等）外，与普通的 Doris 表并无区别。

```
select * from oracle_table where k1 > 1000 and k3 ='term' or k4 like '%doris';
```

数据写入

在 Doris 中建立 ODBC 外表后，可以通过 insert into 语句直接写入数据，也可以将 Doris 执行完查询之后的结果写入 ODBC 外表，或者是从一个 ODBC 外表将数据导入另一个 ODBC 外表。

```
insert into oracle_table values(1, "doris");
insert into oracle_table select * from postgre_table;
```

事务

Doris 的数据是由一组 batch 的方式写入外部表的，如果中途导入中断，之前写入数据可能需要回滚。所以 ODBC 外表支持数据写入时的事务，事务的支持需要通过 session variable： `enable_odbc_transcation` 设置。

```
set enable_odbc_transcation = true;
```

事务保证了 ODBC 外表数据写入的原子性，但是一定程度上会降低数据写入的性能，可以考虑酌情开启该功能。

数据库 ODBC 版本对应关系

Centos 操作系统

使用的 unixODBC 版本是：2.3.1，Doris 0.15，centos 7.9，全部使用 yum 方式安装。

Mysql

Mysql 版本	Mysql ODBC 版本
8.0.27	8.0.27, 8.026
5.7.36	5.3.11, 5.3.13
5.6.51	5.3.11, 5.3.13
5.5.62	5.3.11, 5.3.13

PostgreSQL

PostgreSQL 的 yum 源 rpm 包地址：

```
https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-x86_64/pgdg-redhat-repo
```

这里面包含 PostgreSQL 从9.x 到 14.x的全部版本，包括对应的 ODBC 版本，可以根据需要选择安装。

PostgreSQL 版本	PostgreSQL ODBC 版本
12.9	postgresql12-odbc-13.02.0000
13.5	postgresql13-odbc-13.02.0000
14.1	postgresql14-odbc-13.02.0000
9.6.24	postgresql96-odbc-13.02.0000
10.6	postgresql10-odbc-13.02.0000
11.6	postgresql11-odbc-13.02.0000

Oracle

Oracle 版本	Oracle ODBC 版本
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production	oracle-instantclient19.13-odbc-19.13.0.0.0
Oracle Database 12c Standard Edition Release 12.2.0.1.0 - 64bit	oracle-instantclient19.13-odbc-

Production	19.13.0.0.0
Oracle Database 18c Enterprise Edition Release 18.0.0.0.0 - Production	oracle-instantclient19.13-odbc-19.13.0.0.0
Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production	oracle-instantclient19.13-odbc-19.13.0.0.0
Oracle Database 21c Enterprise Edition Release 21.0.0.0.0 - Production	oracle-instantclient19.13-odbc-19.13.0.0.0

Oracle ODBC 驱动版本下载地址：

```
https://download.oracle.com/otn_software/linux/instantclient/1913000/oracle-instant
https://download.oracle.com/otn_software/linux/instantclient/1913000/oracle-instant
https://download.oracle.com/otn_software/linux/instantclient/1913000/oracle-instant
https://download.oracle.com/otn_software/linux/instantclient/1913000/oracle-instant
```

Ubuntu 操作系统

使用的 unixODBC 版本是：2.3.4，Doris 0.15，Ubuntu 20.04。

Mysql

Mysql 版本	Mysql ODBC 版本
8.0.27	8.0.11, 5.3.13

目前只测试了这一个版本其他版本测试后补充。

PostgreSQL

PostgreSQL版本	PostgreSQL ODBC版本
12.9	psqlodbc-12.02.0000

其他版本只要下载和数据库大版本相符合的 ODBC 驱动版本，这块后续会持续补充其他版本在 Ubuntu 系统下的测试结果。

Oracle

同上 Centos 操作系统的 Oracle 数据库及 ODBC 对应关系，在 ubuntu 下安装 rpm 软件包使用下面方式。
为了在 ubuntu 下可以进行安装 rpm 包，我们还需要安装一个 alien，这是一个可以将 rpm 包转换成 deb 安装包的工具。

```
sudo apt-get install alien
```

然后执行安装上面四个包：

```
sudo alien -i  oracle-instantclient19.13-basic-19.13.0.0.0-2.x86_64.rpm
sudo alien -i  oracle-instantclient19.13-devel-19.13.0.0.0-2.x86_64.rpm
sudo alien -i  oracle-instantclient19.13-odbc-19.13.0.0.0-2.x86_64.rpm
sudo alien -i  oracle-instantclient19.13-sqlplus-19.13.0.0.0-2.x86_64.rpm
```

类型匹配

各个数据库之间数据类型存在不同，这里列出了各个数据库中的类型和 Doris 之中数据类型匹配的情况。

MySQL

MySQL	Doris	替换方案
BOOLEAN	BOOLEAN	-
CHAR	CHAR	当前仅支持 UTF8编码
VARCHAR	VARCHAR	当前仅支持 UTF8编码
DATE	DATE	-
FLOAT	FLOAT	-
TINYINT	TINYINT	-
SMALLINT	SMALLINT	-
INT	INT	-
BIGINT	BIGINT	-
DOUBLE	DOUBLE	-
DATETIME	DATETIME	-
DECIMAL	DECIMAL	-

PostgreSQL

PostgreSQL	Doris	替换方案

BOOLEAN	BOOLEAN	-
CHAR	CHAR	当前仅支持 UTF8编码
VARCHAR	VARCHAR	当前仅支持 UTF8编码
DATE	DATE	-
REAL	FLOAT	-
SMALLINT	SMALLINT	-
INT	INT	-
BIGINT	BIGINT	-
DOUBLE	DOUBLE	-
TIMESTAMP	DATETIME	-
DECIMAL	DECIMAL	-

Oracle

Oracle	Doris	替换方案
不支持	BOOLEAN	Oracle 可用 number(1) 替换 boolean
CHAR	CHAR	-
VARCHAR	VARCHAR	-
DATE	DATE	-
FLOAT	FLOAT	-
无	TINYINT	Oracle 可由 NUMMBER 替换
SMALLINT	SMALLINT	-
INT	INT	-
无	BIGINT	Oracle 可由 NUMMBER 替换
无	DOUBLE	Oracle 可由 NUMMBER 替换
DATETIME	DATETIME	-
NUMBER	DECIMAL	-

SQLServer

SQLServer	Doris	替换方案
BOOLEAN	BOOLEAN	-
CHAR	CHAR	当前仅支持 UTF8编码
VARCHAR	VARCHAR	当前仅支持 UTF8编码
DATE	DATE	-
REAL	FLOAT	-
TINYINT	TINYINT	-
SMALLINT	SMALLINT	-
INT	INT	-
BIGINT	BIGINT	-
FLOAT	DOUBLE	-
DATETIME/DATETIME2	DATETIME	-
DECIMAL/NUMERIC	DECIMAL	-

最佳实践

适用于少数据量的同步。
例如 Mysql 中一张表有100万数据，想同步到 Doris，就可以采用 ODBC 的方式将数据映射过来，在使用 insert into 方式将数据同步到 Doris 中，如果想同步大批量数据，可以分批次使用 insert into 同步（不建议使用）。

常见问题

1. 与原先的 MySQL 外表的关系。
在接入 ODBC 外表之后，原先的访问 MySQL 外表的方式将被逐渐弃用。如果之前没有使用过 MySQL 外表，建议新接入的 MySQL 表直接使用 ODBC 的 MySQL 外表。
2. 除了MySQL，Oracle，PostgreSQL，SQLServer是否能够支持更多的数据库？
目前 Doris 只适配了 MySQL，Oracle，PostgreSQL，SQLServer，关于其他的数据库的适配工作正在规划之中，原

则上来说任何支持 ODBC 访问的数据库都能通过 ODBC 外表来访问。如果您有访问其他外表的需求，欢迎修改代码并贡献给 Doris。

3. 什么场合适合通过外表访问？

通常在外表数据量较小，少于100W条时，可以通过外部表的方式访问。由于外表无法发挥 Doris 在存储引擎部分的能力和会带来额外的网络开销，所以建议根据实际对查询的访问时延要求来确定是否通过外部表访问还是将数据导入 Doris 之中。

4. 通过 Oracle 访问出现乱码？

尝试在 BE 启动脚本之中添加如下参数：`export NLS_LANG=AMERICAN_AMERICA.AL32UTF8`，并重新启动所有 BE。

5. ANSI Driver or Unicode Driver？

当前 ODBC 支持 ANSI 与 Unicode 两种 Driver 形式，当前 Doris 只支持 Unicode Driver。如果强行使用 ANSI Driver 可能会导致查询结果出错。

6. 报错 `driver connect Err: 01000 [unixODBC][Driver Manager]Can't open lib 'Xxx' : file not found (0)` 。

没有在每一个BE上安装好对应数据的 Driver，或者是没有在 be/conf/odbcinst.ini 配置正确的路径，抑或是建表是 Driver 名与 be/conf/odbcinst.ini 不同。

7. 报错 `Fail to convert odbc value 'PALO ' TO INT on column:'A'` ？

ODBC 外表的 A 列类型转换出错，说明外表的实际列与 ODBC 的映射列的数据类型不同，需要修改列的类型映射。

8. 同时使用旧的 MySQL 表与 ODBC 外表的 Driver 时出现程序 Crash？

这个是 MySQL 数据库的 Driver 与现有 Doris 依赖 MySQL 外表的兼容问题。推荐解决的方式如下：

方式1：通过 ODBC 外表替换旧的 MySQL 外表，并重新编译 BE，关闭 WITH_MYSQL 的选项。

方式2：不使用最新8.X 的 MySQL 的 ODBC Driver，而是使用5.X 的 MySQL 的 ODBC Driver。

9. 过滤条件下推？

当前 ODBC 外表支持过滤条件下推，目前 MySQL 的外表是能够支持所有条件下推的。其他的数据库的函数与 Doris 不同会导致下推查询失败。目前除 MySQL 外表之外，其他的数据库不支持函数调用的条件下推。Doris 是否将所需过滤条件下推，可以通过 `explain` 查询语句进行确认。

10. 报错 `driver connect Err: xxx` ？

通常是连接数据库失败，Err 部分代表了不同的数据库连接失败的报错。这种情况通常是配置存在问题。可以检查是否错配了 IP 地址，端口或账号密码。

11. 读写 mysql 外表的 emoji 表情出现乱码？

Doris 进行 odbc 外表连接时，默认采用的编码为 utf8，由于 mysql 之中默认的 utf8编码为 utf8mb3，无法表示需要4字节编码的emoji表情。这里需要在建立 mysql 外表时设置 `charset = utf8mb4`，便可以正常读写 emoji 表情。

Multi-Catalog

概述

最近更新时间：2024-07-04 15:39:49

多源数据目录（Multi-Catalog）是腾讯云数据仓库 TCHouse-D 自1.2.0版本后支持的功能，旨在能够更方便对接外部数据目录，以增强数据湖分析和联邦数据查询能力。

在之前的内核版本中，用户数据只有两个层级：Database 和 Table。

当需要连接一个外部数据目录时，只能在 Database 或 Table 层级进行对接。例如通过 `create external table` 的方式创建一个外部数据目录中的表的映射，或通过 `create external database` 的方式映射一个外部数据目录中的 Database。 如果外部数据目录中的 Database 或 Table 非常多，则需要用户手动进行一一映射，使用体验不佳。

Multi-Catalog 功能在原有的元数据层级上新增一层 Catalog，构成 Catalog > Database > Table 的三层元数据层级。

其中，Catalog 可以直接对应到外部数据目录。目前支持的外部数据目录包括：Hive、Iceberg、Hudi、Elasticsearch，以及通过对接数据库访问的标准接口（JDBC）来访问各式数据库的数据。

说明：

该功能适用于腾讯云数据仓库 TCHouse-D 1.2及后续版本，相较于外表连接，我们更推荐用户使用 Multi-Catalog 的方式实现多数据目录联邦查询。

基础概念

Internal Catalog

Doris 原有的 Database 和 Table 都将归属于 Internal Catalog。Internal Catalog 是内置的默认 Catalog，用户不可修改或删除。

External Catalog

可以通过 CREATE CATALOG 命令创建一个 External Catalog。创建后，可以通过 SHOW CATALOGS 命令查看已创建的 Catalog。

切换 Catalog

用户登录 Doris 后，默认进入 Internal Catalog，因此默认使用和之前版本无差别，可直接使用 `SHOW DATABASES` 、 `USE DB` 等命令查看和切换数据库。用户可以通过 SWITCH 命令切换 Catalog。如：

```
SWITCH internal;
SWITCH hive_catalog;
```

切换后，可以直接通过 `SHOW DATABASES` ， `USE DB` 等命令查看和切换对应 Catalog 中的 Database。Doris 会自动通过 Catalog 中的 Database 和 Table。用户可以像使用 Internal Catalog 一样，对 External Catalog 中的数据进行查看和访问。当前，Doris 只支持对 External Catalog 中的数据进行只读访问。

删除 Catalog

External Catalog 中的 Database 和 Table 都是只读的。但是可以删除 Catalog（Internal Catalog无法删除）。可以通过 DROP CATALOG 命令删除一个 External Catalog。 该操作仅会删除 Doris 中该 Catalog 的映射信息，并不会修改或变更任何外部数据目录的内容。

Resource

Resource 是一组配置的集合。用户可以通过 CREATE RESOURCE 命令创建一个 Resource。之后可以在创建 Catalog 时使用这个 Resource。一个 Resource 可以被多个 Catalog 使用，以复用其中的配置。

连接示例

连接 Hive

本文通过连接一个 Hive 集群说明如何使用 Catalog 功能。

更多关于 Hive 的说明，请参阅：Hive Catalog。

1. 创建 Catalog。

```
CREATE CATALOG hive PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004'
);
```

2. 查看 Catalog。

创建后，可以通过 `SHOW CATALOGS` 命令查看 Catalog：

```
mysql> SHOW CATALOGS;
+-----------+-------------+----------+
| CatalogId | CatalogName | Type     |
+-----------+-------------+----------+
|     10024 | hive        | hms      |
|         0 | internal    | internal |
+-----------+-------------+----------+
```

3. 切换 Catalog。

通过 `SWITCH` 命令切换到 Hive Catalog，并查看其中的数据库：

```
mysql> SWITCH hive;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW DATABASES;
+-----------+
| Database  |
+-----------+
| default   |
```

```
| random    |
| ssb100    |
| tpch1     |
| tpch100   |
| tpch1_orc |
+-----------+
```

4. 使用 Catalog。

切换到 Catalog 后，则可以正常使用内部数据源的功能。如切换到 tpch100 数据库，并查看其中的表：

```
mysql> USE tpch100;
Database changed

mysql> SHOW TABLES;
+-------------------+
| Tables_in_tpch100 |
+-------------------+
| customer          |
| lineitem          |
| nation            |
| orders            |
| part              |
| partsupp          |
| region            |
| supplier          |
+-------------------+
```

查看 lineitem 表的 schema：

```
mysql> DESC lineitem;
+-----------------+---------------+------+------+---------+-------+
| Field           | Type          | Null | Key  | Default | Extra |
+-----------------+---------------+------+------+---------+-------+
| l_shipdate      | DATE          | Yes  | true | NULL    |       |
| l_orderkey      | BIGINT        | Yes  | true | NULL    |       |
| l_linenumber    | INT           | Yes  | true | NULL    |       |
| l_partkey       | INT           | Yes  | true | NULL    |       |
| l_suppkey       | INT           | Yes  | true | NULL    |       |
| l_quantity      | DECIMAL(15,2) | Yes  | true | NULL    |       |
| l_extendedprice | DECIMAL(15,2) | Yes  | true | NULL    |       |
| l_discount      | DECIMAL(15,2) | Yes  | true | NULL    |       |
| l_tax           | DECIMAL(15,2) | Yes  | true | NULL    |       |
| l_returnflag    | TEXT          | Yes  | true | NULL    |       |
| l_linestatus    | TEXT          | Yes  | true | NULL    |       |
| l_commitdate    | DATE          | Yes  | true | NULL    |       |
| l_receiptdate   | DATE          | Yes  | true | NULL    |       |
| l_shipinstruct  | TEXT          | Yes  | true | NULL    |       |
```

```
| l_shipmode      | TEXT           | Yes   | true | NULL    |        |
| l_comment       | TEXT           | Yes   | true | NULL    |        |
+-----------------+----------------+------+------+---------+-------+
```

查询示例：

```
mysql> SELECT l_shipdate, l_orderkey, l_partkey FROM lineitem limit 10;
+------------+------------+-----------+
| l_shipdate | l_orderkey | l_partkey |
+------------+------------+-----------+
| 1998-01-21 |   66374304 |    270146 |
| 1997-11-17 |   66374304 |    340557 |
| 1997-06-17 |   66374400 |   6839498 |
| 1997-08-21 |   66374400 |  11436870 |
| 1997-08-07 |   66374400 |  19473325 |
| 1997-06-16 |   66374400 |   8157699 |
| 1998-09-21 |   66374496 |  19892278 |
| 1998-08-07 |   66374496 |   9509408 |
| 1998-10-27 |   66374496 |   4608731 |
| 1998-07-14 |   66374592 |  13555929 |
+------------+------------+-----------+
```

也可以和其他数据目录中的表进行关联查询：

```
mysql> SELECT l.l_shipdate FROM hive.tpch100.lineitem l WHERE l.l_partkey IN (SELEC
+-------------+
| l_shipdate  |
+-------------+
| 1993-02-16  |
| 1995-06-26  |
| 1995-08-19  |
| 1992-07-23  |
| 1998-05-23  |
| 1997-07-12  |
| 1994-03-06  |
| 1996-02-07  |
| 1997-06-01  |
| 1996-08-23  |
+-------------+
```

这里我们通过 `catalog.database.table` 这种全限定的方式标识一张表，如： `internal.db1.part` 。

其中 `catalog` 和 `database` 可以省略，缺省使用当前 SWITCH 和 USE 后切换的 Catalog 和 Database。

可以通过 INSERT INTO 命令，将 Hive Catalog 中的表数据，插入到 Interal Catalog 中的内部表，从而达到**导入外部数据目录数据**的效果：

```
mysql> SWITCH internal;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> USE db1;
Database changed

mysql> INSERT INTO part SELECT * FROM hive.tpch100.part limit 1000;
Query OK, 1000 rows affected (0.28 sec)
{'label':'insert_212f67420c6444d5_9bfc184bf2e7edb8', 'status':'VISIBLE', 'txnId':'4
```

连接 Iceberg

详情请参见 Iceberg Catalog。

连接 Hudi

详情请参见 Hudi Catalog。

连接 Elasticsearch

详情请参见 ES Catalog。

连接 JDBC

详情请参见 JDBC Catalog。

列类型映射

用户创建 Catalog 后，Doris 会自动同步数据目录的数据库和表，针对不同的数据目录和数据表格式，Doris 会进行列映射。对于当前无法映射到 Doris 列类型的外表类型，如 `UNION`，`INTERVAL` 等。Doris 会将列类型映射为 UNSUPPORTED 类型。对于 UNSUPPORTED 类型的查询，示例如下：
假设同步后的表 schema 为：

```
k1 INT,
k2 INT,
k3 UNSUPPORTED,
k4 INT
```

```
select * from table;                // Error: Unsupported type 'UNSUPPORTED_TYPE' i
select * except(k3) from table;     // Query OK.
select k1, k3 from table;           // Error: Unsupported type 'UNSUPPORTED_TYPE' i
select k1, k4 from table;           // Query OK.
```

不同的数据源的列映射规则，详情请参阅不同数据源的文档。

权限管理

使用 Doris 对 External Catalog 中库表进行访问，并不受外部数据目录自身的权限控制，而是依赖 Doris 自身的权限访问管理功能。
Doris 的权限管理功能提供了对 Catalog 层级的扩展，具体可参阅 权限管理 文档。

指定需要同步的数据库

通过在 Catalog 配置中设置 `include_database_list` 和 `exclude_database_list` 可以指定需要同步的数据库。

`include_database_list` ：支持只同步指定的多个database，以','分隔。默认为''，同步所有database。db名称是大小写敏感的。

`exclude_database_list` ：支持指定不需要同步的多个database，以','分割。默认为''，即不做任何过滤，同步所有database。db名称是大小写敏感的。

说明：

当 `include_database_list` 和 `exclude_database_list` 有重合的database配置时，`exclude_database_list` 会优先生效。

连接 JDBC 时，上述 2 个配置需要和配置 `only_specified_database` 搭配使用，详情请参见 JDBC Catalog。

元数据更新

默认情况下，外部数据源的元数据变动，如创建、删除表，加减列等操作，不会同步给 Doris。

手动刷新

用户需要通过 REFRESH CATALOG 命令手动刷新元数据。

自动刷新

Hive Metastore

自动刷新目前仅支持 Hive Metastore 元数据服务。通过让 FE 节点定时读取 HMS 的 notification event 来感知 Hive 表元数据的变更情况，目前支持处理如下event：

事件	事件行为和对应的动作
CREATE DATABASE	在对应数据目录下创建数据库。
DROP DATABASE	在对应数据目录下删除数据库。

ALTER DATABASE	此事件的影响主要有更改数据库的属性信息，注释及默认存储位置等，这些改变不影响 doris 对外部数据目录的查询操作，因此目前会忽略此 event。
CREATE TABLE	在对应数据库下创建表。
DROP TABLE	在对应数据库下删除表，并失效表的缓存。
ALTER TABLE	如果是重命名，先删除旧名字的表，再用新名字创建表，否则失效该表的缓存。
ADD PARTITION	在对应表缓存的分区列表里添加分区。
DROP PARTITION	在对应表缓存的分区列表里删除分区，并失效该分区的缓存。
ALTER PARTITION	如果是重命名，先删除旧名字的分区，再用新名字创建分区，否则失效该分区的缓存。

说明：

当导入数据导致文件变更,分区表会走 ALTER PARTITION event 逻辑，不分区表会走 ALTER TABLE event 逻辑。

如果绕过 HMS 直接操作文件系统的话，HMS 不会生成对应事件，Doris 因此也无法感知。

该特性在 fe.conf 中有如下参数：

1. `enable_hms_events_incremental_sync` ：是否开启元数据自动增量同步功能，默认关闭。
2. `hms_events_polling_interval_ms` ：读取 event 的间隔时间，默认值为 10000，单位：毫秒。
3. `hms_events_batch_size_per_rpc` ：每次读取 event 的最大数量，默认值为 500。

如果想使用该特性，需要更改 HMS 的 hive-site.xml 并重启 HMS：

```
<property>
    <name>hive.metastore.event.db.notification.api.auth</name>
    <value>false</value>
</property>
<property>
    <name>hive.metastore.dml.events</name>
    <value>true</value>
</property>
<property>
    <name>hive.metastore.transactional.event.listeners</name>
    <value>org.apache.hive.hcatalog.listener.DbNotificationListener</value>
</property>
```

使用建议： 无论是之前已经创建好的 Catalog 现在想改为自动刷新，还是新创建的 Catalog，都只需要把 `enable_hms_events_incremental_sync` 设置为 true，重启 FE 节点，无需重启之前或之后再手动刷新元数据。

定时刷新

在创建 Catalog 时，在 properties 中指定刷新时间参数 `metadata_refresh_interval_sec` ，以秒为单位，若在创建 Catalog 时设置了该参数，FE 的master 节点会根据参数值定时刷新该 Catalog。目前支持三种类型：

HMS：Hive MetaStore

ES：Elasticsearch

JDBC：数据库访问的标准接口（JDBC）

示例

```
-- 设置catalog刷新间隔为20秒
CREATE CATALOG es PROPERTIES (
    "type"="es",
    "hosts"="http://127.0.0.1:9200",
    "metadata_refresh_interval_sec"="20"
);
```

Hive Catalog

最近更新时间：2024-12-12 10:43:03

说明：

该功能适用于腾讯云数据仓库 TCHouse-D 1.2及后续版本。

通过连接 Hive Metastore，或者兼容 Hive Metastore 的元数据服务，Doris 可以自动获取 Hive 的库表信息，并进行数据查询。

除了 Hive 外，很多其他系统也会使用 Hive Metastore 存储元数据。所以通过 Hive Catalog，我们不仅能访问 Hive，也能访问使用 Hive Metastore 作为元数据存储的系统。如 Iceberg、Hudi 等。

使用须知

1. 将 core-site.xml，hdfs-site.xml 和 hive-site.xml 放到 FE 和 BE 的 conf 目录下。优先读取 conf 目录下的 hadoop 配置文件，再读取环境变量 `HADOOP_CONF_DIR` 的相关配置文件。
2. Hive 支持 1/2/3 版本。
3. 支持 Managed Table 和 External Table。
4. 可以识别 Hive Metastore 中存储的 Hive、Iceberg、Hudi 元数据。
5. 支持数据存储在 Juicefs 上的 Hive 表，用法如下（需要把 juicefs-hadoop-x.x.x.jar放在 fe/lib/ 和 apache_hdfs_broker/lib/ 下）。
6. 支持数据存储在 CHDFS 上的 Hive 表。需配置环境：

6.1 把chdfs_hadoop_plugin_network-x.x.jar 放在 fe/lib/ 和 apache_hdfs_broker/lib/ 下。

6.2 将 Hive 所在 Hadoop 集群的 core-site.xml 和 hdfs-site.xml 复制到 fe/conf/ 和 apache_hdfs_broker/conf 目录下。

7. 支持数据存储在 GooseFS(GFS) 上的 Hive、Iceberg 表。需配置环境：

7.1 把 goosefs-x.x.x-client.jar 放在 fe/lib/ 和 apache_hdfs_broker/lib/ 下。

7.2 创建 catalog 时增加属性：'fs.AbstractFileSystem.gfs.impl' = 'com.qcloud.cos.goosefs.hadoop.GooseFileSystem'， 'fs.gfs.impl' = 'com.qcloud.cos.goosefs.hadoop.FileSystem'。

功能限制

1. 不支持查询Hive / Iceberg / Hudi 中的视图。
2. v1.2 版本不支持查询 show partitions from 语句，v2.0之后支持。
3. 对于文本格式的表，Doris只支持非压缩格式，不支持lzo, bzip2, deflate, lz4 和 zstd 等压缩格式。
4. Doris 会缓存 Hive / Iceberg / Hudi 表的元数据，如原表更新频繁且要从doris获得最准确的数据，需要查表之前执行refresh table。

创建 Catalog

```
CREATE CATALOG hive PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004',
    'hadoop.username' = 'hive',
    'dfs.nameservices'='your-nameservice',
    'dfs.ha.namenodes.your-nameservice'='nn1,nn2',
    'dfs.namenode.rpc-address.your-nameservice.nn1'='172.21.0.2:4007',
    'dfs.namenode.rpc-address.your-nameservice.nn2'='172.21.0.3:4007',
    'dfs.client.failover.proxy.provider.your-nameservice'='org.apache.hadoop.hdfs.s
);
```

除了 `type` 和 `hive.metastore.uris` 两个必须参数外，还可以通过更多参数来传递连接所需要的信息。

如提供 HDFS HA 信息，示例如下：

```
CREATE CATALOG hive PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004',
    'hadoop.username' = 'hive',
    'dfs.nameservices'='your-nameservice',
    'dfs.ha.namenodes.your-nameservice'='nn1,nn2',
    'dfs.namenode.rpc-address.your-nameservice.nn1'='172.21.0.2:4007',
    'dfs.namenode.rpc-address.your-nameservice.nn2'='172.21.0.3:4007',
    'dfs.client.failover.proxy.provider.your-nameservice'='org.apache.hadoop.hdfs.s
);
```

同时提供 HDFS HA 信息和 Kerberos 认证信息，示例如下：

```
CREATE CATALOG hive PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004',
    'hive.metastore.sasl.enabled' = 'true',
    'hive.metastore.kerberos.principal' = 'your-hms-principal',
    'dfs.nameservices'='your-nameservice',
    'dfs.namenode.rpc-address.your-nameservice.nn1'='172.21.0.2:4007',
    'dfs.namenode.rpc-address.your-nameservice.nn2'='172.21.0.3:4007',
    'dfs.client.failover.proxy.provider.your-nameservice'='org.apache.hadoop.hdfs.s
    'hadoop.security.authentication' = 'kerberos',
    'hadoop.kerberos.keytab' = '/your-keytab-filepath/your.keytab',
    'hadoop.kerberos.principal' = 'your-principal@YOUR.COM',
    'yarn.resourcemanager.principal' = 'your-rm-principal'
);
```

请在所有的 `BE` 、 `FE` 节点下放置 `krb5.conf` 文件和 `keytab` 认证文件， `keytab` 认证文件路径和配置保持一致， `krb5.conf` 文件默认放置在 `/etc/krb5.conf` 路径。

`hive.metastore.kerberos.principal` 的值需要和所连接的 hive metastore 的同名属性保持一致，可从 `hive-site.xml` 中获取。

提供 Hadoop KMS 加密传输信息，示例如下：

```
CREATE CATALOG hive PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004',
    'dfs.encryption.key.provider.uri' = 'kms://http@kms_host:kms_port/kms'
);
```

Hive数据存储在JuiceFS，示例如下：

```
CREATE CATALOG hive PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004',
    'hadoop.username' = 'root',
    'fs.jfs.impl' = 'io.juicefs.JuiceFileSystem',
    'fs.AbstractFileSystem.jfs.impl' = 'io.juicefs.JuiceFS',
    'juicefs.meta' = 'xxx'
);
```

在 1.2.1 版本之后，我们也可以将这些信息通过创建一个 Resource 统一存储，然后在创建 Catalog 时使用这个 Resource。示例如下：

```
# 1. 创建 Resource
CREATE RESOURCE hms_resource PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004',
    'hadoop.username' = 'hive',
    'dfs.nameservices'='your-nameservice',
    'dfs.ha.namenodes.your-nameservice'='nn1,nn2',
    'dfs.namenode.rpc-address.your-nameservice.nn1'='172.21.0.2:4007',
    'dfs.namenode.rpc-address.your-nameservice.nn2'='172.21.0.3:4007',
    'dfs.client.failover.proxy.provider.your-nameservice'='org.apache.hadoop.hdfs.s
);

# 2. 创建 Catalog 并使用 Resource，这里的 Key Value 信息会覆盖 Resource 中的信息。
CREATE CATALOG hive WITH RESOURCE hms_resource PROPERTIES(
    'key' = 'value'
);
```

创建 Catalog 时可以采用参数 `file.meta.cache.ttl-second` 来设置 File Cache 自动失效时间，也可以将该值设置为 0 来禁用 File Cache。时间单位为：秒。示例如下：

```
CREATE CATALOG hive PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004',
    'hadoop.username' = 'hive',
    'dfs.nameservices'='your-nameservice',
    'dfs.ha.namenodes.your-nameservice'='nn1,nn2',
    'dfs.namenode.rpc-address.your-nameservice.nn1'='172.21.0.2:4007',
    'dfs.namenode.rpc-address.your-nameservice.nn2'='172.21.0.3:4007',
    'dfs.client.failover.proxy.provider.your-nameservice'='org.apache.hadoop.hdfs.s
    'file.meta.cache.ttl-second' = '60'
);
```

我们也可以直接将 hive-site.xml 放到 FE 和 BE 的 conf 目录下，系统也会自动读取 hive-site.xml 中的信息。信息覆盖的规则如下：

Resource 中的信息覆盖 hive-site.xml 中的信息。

CREATE CATALOG PROPERTIES 中的信息覆盖 Resource 中的信息。

Hive 版本

Doris 可以正确访问不同 Hive 版本中的 Hive Metastore。在默认情况下，Doris 会以 Hive 2.3 版本的兼容接口访问 Hive Metastore。您也可以在创建 Catalog 时指定 hive 的版本。如访问 Hive 1.1.0 版本：

```
CREATE CATALOG hive PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.0.1:7004',
    'hive.version' = '1.1.0'
);
```

列类型映射

适用于 Hive/Iceberge/Hudi

HMS Type	Doris Type	Comment
boolean	boolean	-
tinyint	tinyint	-
smallint	smallint	-
int	int	-
bigint	bigint	-

date	date	-
timestamp	datetime	-
float	float	-
double	double	-
char	char	-
varchar	varchar	-
decimal	decimal	-
`array<type>`	`array<type>`	支持array嵌套，如 `array<array<int>>`
`map<KeyType, ValueType>`	`map<KeyType, ValueType>`	暂不支持嵌套，KeyType 和 ValueType 需要为基础类型
`struct<col1: Type1, col2: Type2, ...>`	`struct<col1: Type1, col2: Type2, ...>`	暂不支持嵌套，Type1，Type2，... 需要为基础类型
other	unsupported	-

使用Ranger进行权限校验

说明：

该功能适用于 Doris 2.0及后续版本。

Apache Ranger 是一个用来在 Hadoop 平台上进行监控，启用服务，以及全方位数据安全访问管理的安全框架。目前 Doris 支持 Ranger 的库、表、列权限，不支持加密、行权限等。

环境配置

连接开启 Ranger 权限校验的 Hive Metastore 需要增加配置和配置环境：

1. 创建 Catalog 时增加：

```
"access_controller.properties.ranger.service.name" = "hive",
"access_controller.class" = "org.apache.doris.catalog.authorizer.RangerHiveAccessCo
```

2. 配置所有 FE 环境：

2.1 将 HMS conf 目录下的配置文件 ranger-hive-audit.xml,ranger-hive-security.xml,ranger-policymgr-ssl.xml 复制到 <doris_home>/conf 目录下。

2.2 修改 ranger-hive-security.xml 的属性，参考配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    #The directory for caching permission data, needs to be writable
    <property>
        <name>ranger.plugin.hive.policy.cache.dir</name>
        <value>/mnt/datadisk0/zhangdong/rangerdata</value>
    </property>
    #The time interval for periodically pulling permission data
    <property>
        <name>ranger.plugin.hive.policy.pollIntervalMs</name>
        <value>30000</value>
    </property>

    <property>
        <name>ranger.plugin.hive.policy.rest.client.connection.timeoutMs</name>
        <value>60000</value>
    </property>

    <property>
        <name>ranger.plugin.hive.policy.rest.client.read.timeoutMs</name>
        <value>60000</value>
    </property>

    <property>
        <name>ranger.plugin.hive.policy.rest.ssl.config.file</name>
        <value></value>
    </property>

    <property>
        <name>ranger.plugin.hive.policy.rest.url</name>
        <value>http://172.21.0.32:6080</value>
    </property>

    <property>
        <name>ranger.plugin.hive.policy.source.impl</name>
        <value>org.apache.ranger.admin.client.RangerAdminRESTClient</value>
    </property>

    <property>
        <name>ranger.plugin.hive.service.name</name>
        <value>hive</value>
    </property>

    <property>
        <name>xasecure.hive.update.xapolicies.on.grant.revoke</name>
```

```
        <value>true</value>
    </property>

</configuration>
```

2.3 为获取到 Ranger 鉴权本身的日志，可在 <doris_home>/conf 目录下添加配置文件 log4j.properties。

2.4 重启 FE。

最佳实践

1. 在 Ranger 端创建用户 user1 并授权 db1.table1.col1 的查询权限。
2. 在 Ranger 端创建角色 role1 并授权 db1.table1.col2 的查询权限。
3. 在 Doris 创建同名用户 user1，user1 将直接拥有 db1.table1.col1 的查询权限。
4. 在 Doris 创建同名角色 role1，并将 role1 分配给 user1，user1 将同时拥有 db1.table1.col1 和 col2 的查询权限。

ES Catalog

最近更新时间：2024-07-04 15:42:27

Elasticsearch Catalog（ES）除了支持自动映射 ES 元数据外，也可以利用腾讯云数据仓库 TCHouse-D 的分布式查询规划能力和 ES 的全文检索能力相结合，提供更完善的 OLAP 分析场景解决方案：

1. ES 中的多 index 分布式 Join 查询。
2. 腾讯云数据仓库 TCHouse-D 和 ES 中的表联合查询，更复杂的全文检索过滤。

说明：

该功能适用于腾讯云数据仓库 TCHouse-D 1.2及后续版本，支持 Elasticsearch 5.x 及以上版本。

创建 Catalog

```
CREATE CATALOG es PROPERTIES (
    "type"="es",
    "hosts"="http://127.0.0.1:9200"
);
```

因为 Elasticsearch 没有 Database 的概念，所以连接 ES 后，会自动生成一个唯一的 Database：`default_db`。并且在通过 SWITCH 命令切换到 ES Catalog 后，会自动切换到 `default_db`。无需再执行 `USE default_db` 命令。

参数说明

参数	是否必须	默认值	说明
`hosts`	是	-	ES 地址，可以是一个或多个，也可以是 ES 的负载均衡地址
`user`	否	空	ES 用户名
`password`	否	空	对应用户的密码信息
`doc_value_scan`	否	true	是否开启通过 ES/Lucene 列式存储获取查询字段的值
`keyword_sniff`	否	true	是否对 ES 中字符串分词类型 text.fields 进行探测，通过 keyword 进行查询。设置为 false 会按照分词后的内容匹配
`nodes_discovery`	否	true	是否开启 ES 节点发现，默认为 true，在网络隔离环境下设置为 false，只连接指定节点
`ssl`	否	false	ES 是否开启 https 访问模式，目前在 fe/be 实现方式为信任所

			有
`mapping_es_id`	否	false	是否映射 ES 索引中的 `_id` 字段
`like_push_down`	否	true	是否将 like 转化为 wildcard 下推到 ES，会增加 ES cpu 消耗

注意：

认证方式目前仅支持 Http Basic 认证，并且需要确保该用户有访问: `/_cluster/state/`、`_nodes/http` 等路径和 index 的读权限; 集群未开启安全认证，用户名和密码不需要设置。

5.x 和 6.x 中一个 index 中的多个 type 默认取第一个。

列类型映射

ES Type	Doris Type	Comment
null	null	-
boolean	boolean	-
byte	tinyint	-
short	smallint	-
integer	int	-
long	bigint	-
unsigned_long	largeint	-
float	float	-
half_float	float	-
double	double	-
scaled_float	double	-
date	date	仅支持 default/yyyy-MM-dd HH:mm:ss/yyyy-MM-dd/epoch_millis 格式
keyword	string	-
text	string	-
ip	string	-

nested	string	-
object	string	-
other	unsupported	-

Array 类型

Elasticsearch 没有明确的数组类型，但是它的某个字段可以含有0个或多个值。为了表示一个字段是数组类型，可以在索引映射的_meta部分添加特定的 `doris` 结构注释。对于 Elasticsearch 6.x 及之前版本，请参考_meta。
举例说明，假设有一个索引 `doc` 包含以下的数据结构：

```
{
  "array_int_field": [1, 2, 3, 4],
  "array_string_field": ["doris", "is", "the", "best"],
  "id_field": "id-xxx-xxx",
  "timestamp_field": "2022-11-12T12:08:56Z",
  "array_object_field": [
    {
      "name": "xxx",
      "age": 18
    }
  ]
}
```

该结构的数组字段可以通过使用以下命令将字段属性定义添加到目标索引映射的 `_meta.doris` 属性来定义。

```
# ES 7.x and above
curl -X PUT "localhost:9200/doc/_mapping?pretty" -H 'Content-Type:application/json'
{
    "_meta": {
        "doris":{
            "array_fields":[
                "array_int_field",
                "array_string_field",
                "array_object_field"
            ]
        }
    }
}'

# ES 6.x and before
curl -X PUT "localhost:9200/doc/_mapping?pretty" -H 'Content-Type: application/json
{
    "_doc": {
        "_meta": {
```

```
            "doris":{
                "array_fields":[
                    "array_int_field",
                    "array_string_field",
                    "array_object_field"
                ]
            }
    }
    }
}
```

`array_fields` ：用来表示数组类型的字段。

最佳实践

过滤条件下推

ES Catalog 支持过滤条件的下推: 过滤条件下推给 ES，这样只有真正满足条件的数据才会被返回，能够显著的提高查询性能和降低 Doris 和 Elasticsearch 的CPU、memory、IO 使用量。下面的操作符（Operators）会被优化成如下 ES Query：

SQL syntax	ES 5.x+ syntax
=	term query
in	terms query
> , < , >= , ⇐	range query
and	bool.filter
or	bool.should
not	bool.must_not
not in	bool.must_not + terms query
is_not_null	exists query
is_null	bool.must_not + exists query
esquery	ES 原生 json 形式的 QueryDSL

启用列式扫描优化查询速度（enable_docvalue_scan=true）

设置 `"enable_docvalue_scan" = "true"`

开启后 Doris 从 ES 中获取数据会遵循以下两个原则：

尽力而为：自动探测要读取的字段是否开启列式存储(doc_value: true)，如果获取的字段全部有列存，Doris 会从列式存储中获取所有字段的值。

自动降级：如果要获取的字段只要有一个字段没有列存，所有字段的值都会从行存 `_source` 中解析获取。

优势

默认情况下，Doris On ES 会从行存也就是 `_source` 中获取所需的所有列， `_source` 的存储采用的行式+json的形式存储，在批量读取性能上要劣于列式存储，尤其在只需要少数列的情况下尤为明显，只获取少数列的情况下，docvalue 的性能大约是 `_source` 性能的十几倍。

注意：

`text` 类型的字段在 ES 中是没有列式存储，因此如果要获取的字段值有 `text` 类型字段会自动降级为从 `_source` 中获取。

在获取的字段数量过多的情况下(`>= 25`)，从 `docvalue` 中获取字段值的性能会和从 `_source` 中获取字段值基本一样。

探测 keyword 类型字段

设置 `"enable_keyword_sniff" = "true"`

在 ES 中可以不建立 index 直接进行数据导入，这时候 ES 会自动创建一个新的索引，针对字符串类型的字段ES会创建一个既有 `text` 类型的字段又有 `keyword` 类型的字段，这就是 ES 的 multi fields 特性，mapping 如下：

```
"k4": {
   "type": "text",
   "fields": {
      "keyword": {
         "type": "keyword",
         "ignore_above": 256
      }
   }
}
```

对 k4 进行条件过滤时比如=，Doris On ES会将查询转换为 ES 的 TermQuery。

SQL 过滤条件：

```
k4 = "Doris On ES"
```

转换成 ES 的 query DSL 为：

```
"term" : {
    "k4": "Doris On ES"
}
```

因为 k4 的第一字段类型为 `text` ，在数据导入的时候就会根据 k4 设置的分词器（如果没有设置，就是 standard 分词器）进行分词处理得到 doris、on、es三个 Term，如下 ES analyze API 分析：

```
POST /_analyze
{
  "analyzer": "standard",
  "text": "Doris On ES"
}
```

分词的结果是：

```
{
   "tokens": [
      {
         "token": "doris",
         "start_offset": 0,
         "end_offset": 5,
         "type": "<ALPHANUM>",
         "position": 0
      },
      {
         "token": "on",
         "start_offset": 6,
         "end_offset": 8,
         "type": "<ALPHANUM>",
         "position": 1
      },
      {
         "token": "es",
         "start_offset": 9,
         "end_offset": 11,
         "type": "<ALPHANUM>",
         "position": 2
      }
   ]
}
```

查询时使用的是：

```
"term" : {
    "k4": "Doris On ES"
}
```

`Doris On ES` 这个 term 匹配不到词典中的任何 term，不会返回任何结果，而启用 `enable_keyword_sniff: true` 会自动将 `k4 = "Doris On ES"` 转换成 `k4.keyword = "Doris On ES"` 来完全匹配 SQL 语义，转换后的 ES query DSL 为：

```
"term" : {
    "k4.keyword": "Doris On ES"
}
```

`k4.keyword` 的类型是 `keyword`，数据写入 ES 中是一个完整的 term，所以可以匹配。

开启节点自动发现，默认为 true(nodes_discovery=true)

设置 `"nodes_discovery" = "true"`
当配置为 true 时，Doris 将从 ES 找到所有可用的相关数据节点(在上面分配的分片)。如果 ES 数据节点的地址没有被 Doris BE 访问，则设置为 false。ES 集群部署在与公共 Internet 隔离的内网，用户通过代理访问。

ES 集群是否开启 https 访问模式

设置 `"ssl" = "true"`，目前 FE/BE 实现方式为信任所有，这是临时解决方案，后续会使用真实的用户配置证书。

查询用法

完成在 Doris 中建立 ES 外表后，除了无法使用 Doris 中的数据模型（Rollup、预聚合、物化视图等）外，和内表使用并无区别。

基本查询

```
select * from es_table where k1 > 1000 and k3 ='term' or k4 like 'fu*z_'
```

扩展的 esquery(field, QueryDSL)

通过 `esquery(field, QueryDSL)` 函数将一些无法用 sql 表述的 query 如 match_phrase、geoshape 等下推给 ES 进行过滤处理，`esquery` 的第一个列名参数用于关联 `index`，第二个参数是 ES 的基本 `Query DSL` 的 json 表述，使用花括号 `{}` 包含，json 的 `root key` 有且只能有一个，如 `match_phrase`、`geo_shape`、`bool` 等。
`match_phrase` 查询：

```
select * from es_table where esquery(k4, '{
        "match_phrase": {
           "k4": "doris on es"
        }
    }');
```

`geo` 相关查询：

```
select * from es_table where esquery(k4, '{
      "geo_shape": {
         "location": {
            "shape": {
               "type": "envelope",
               "coordinates": [
                  [
```

```
                    13,
                    53
                ],
                [
                    14,
                    52
                ]
            ]
        },
        "relation": "within"
    }
}
}');
```

`bool` 查询：

```
select * from es_table where esquery(k4, ' {
        "bool": {
            "must": [
                {
                    "terms": {
                        "k1": [
                            11,
                            12
                        ]
                    }
                },
                {
                    "terms": {
                        "k2": [
                            100
                        ]
                    }
                }
            ]
        }
    }');
```

时间类型字段使用建议

说明：

仅 ES 外表适用，ES Catalog 中自动映射日期类型为 Date 或 Datetime。

在 ES 中，时间类型的字段使用十分灵活，但是在 ES 外表中如果对时间类型字段的类型设置不当，则会造成过滤条件无法下推。

创建索引时对时间类型格式的设置做最大程度的格式兼容：

```
 "dt": {
     "type": "date",
     "format": "yyyy-MM-dd HH:mm:ss||yyyy-MM-dd||epoch_millis"
 }
```

在 Doris 中建立该字段时建议设置为 `date` 或 `datetime` ，也可以设置为 `varchar` 类型, 使用如下 SQL 语句都可以直接将过滤条件下推至 ES：

```
select * from doe where k2 > '2020-06-21';

select * from doe where k2 < '2020-06-21 12:00:00';

select * from doe where k2 < 1593497011;

select * from doe where k2 < now();

select * from doe where k2 < date_format(now(), '%Y-%m-%d');
```

注意：

在 ES 中如果不对时间类型的字段设置 `format` ，默认的时间类型字段格式为：

```
   strict_date_optional_time||epoch_millis
```

导入到 ES 的日期字段如果是时间戳需要转换成 `ms` ，ES内部处理时间戳都是按照 `ms` 进行处理的, 否则 ES 外表会出现显示错误。

获取 ES 元数据字段 `_id`

导入文档在不指定 `_id` 的情况下，ES 会给每个文档分配一个全局唯一的 `_id` 即主键, 用户也可以在导入时为文档指定一个含有特殊业务意义的 `_id` ；

如果需要在 ES 外表中获取该字段值，建表时可以增加类型为 `varchar` 的 `_id` 字段：

```
CREATE EXTERNAL TABLE `doe` (
  `_id` varchar COMMENT "",
  `city`  varchar COMMENT ""
) ENGINE=ELASTICSEARCH
PROPERTIES (
"hosts" = "http://127.0.0.1:8200",
"user" = "root",
"password" = "root",
"index" = "doe"
}
```

如果需要在 ES Catalog 中获取该字段值，请设置 `"mapping_es_id" = "true"` 。

注意：

`_id` 字段的过滤条件仅支持 `=` 和 `in` 两种。

`_id` 字段必须为 `varchar` 类型。

常见问题

是否支持 X-Pack 认证的 ES 集群?

支持所有使用 HTTP Basic 认证方式的 ES 集群。

一些查询比请求 ES 慢很多？

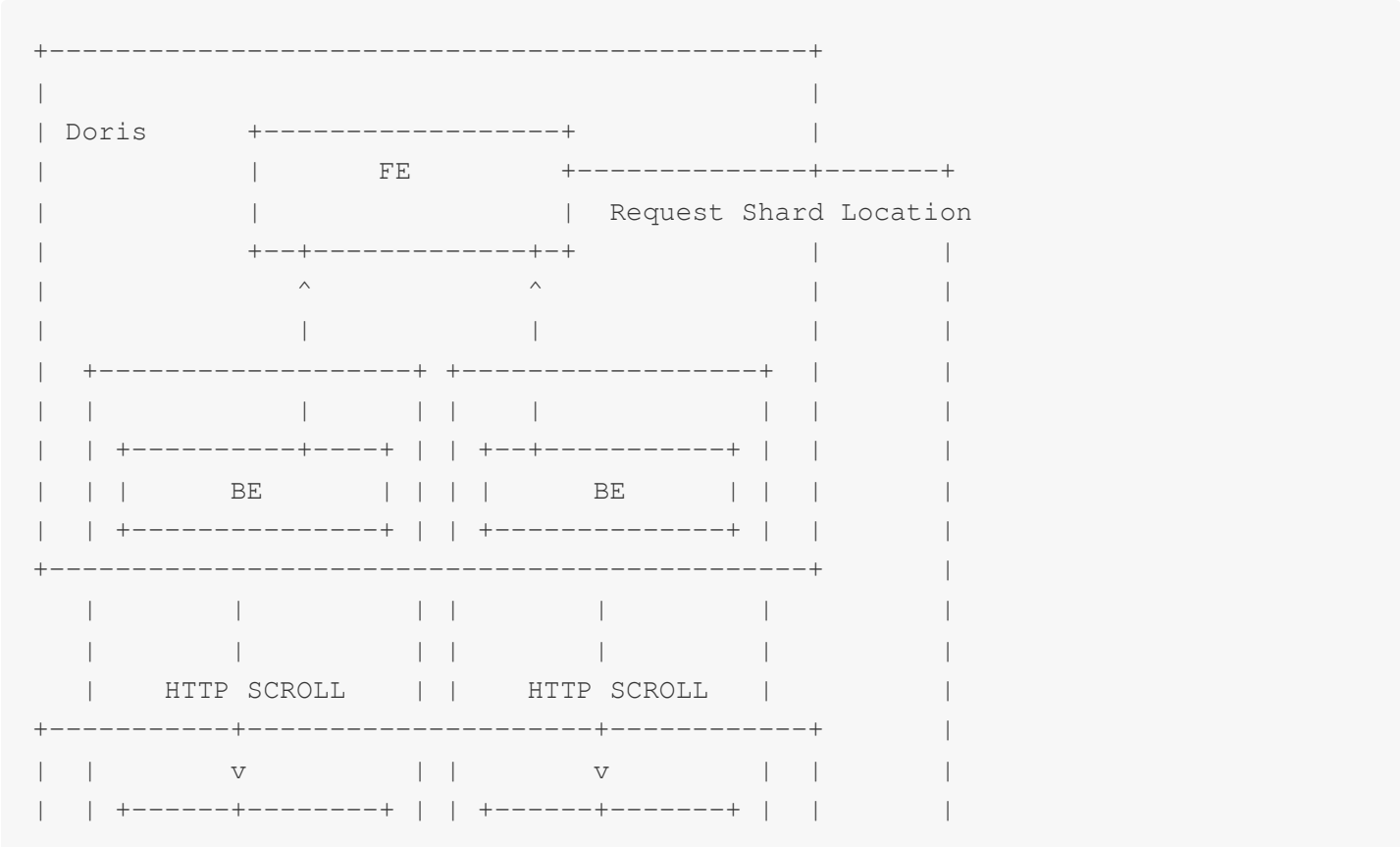
是，例如 _count 相关的 query 等，ES 内部会直接读取满足条件的文档个数相关的元数据，不需要对真实的数据进行过滤。

聚合操作是否可以下推？

目前 Doris On ES 不支持聚合操作如 sum，avg，min/max 等下推，计算方式是批量流式的从 ES 获取所有满足条件的文档，然后在 Doris 中进行计算。

附录

腾讯云数据仓库 TCHouse-D 查询 ES 原理



1. FE 会请求建表指定的主机，获取所有节点的 HTTP 端口信息以及 index 的 shard 分布信息等，如果请求失败会顺序遍历 host 列表直至成功或完全失败。
2. 查询时会根据 FE 得到的一些节点信息和 index 的元数据信息，生成查询计划并发给对应的 BE 节点。
3. BE 节点会根据**就近原则**即优先请求本地部署的 ES 节点，BE 通过 `HTTP Scroll` 方式流式的从 ES index 的每个分片中并发的从 `_source` 或 `docvalue` 中获取数据。
4. 计算完结果后，返回给用户。

Hudi Catalog

最近更新时间：2024-07-04 15:42:53

Mutil Catalog Hudi 提供了腾讯云数据仓库 TCHouse-D 直接访问 Hudi 外部表的能力，外部表省去了繁琐的数据导入工作，并借助腾讯云数据仓库 TCHouse-D 本身的 OLAP 的能力来解决 Hudi 表的数据分析问题：

1、支持 Hudi 数据源接入腾讯云数据仓库 TCHouse-D。

2、支持腾讯云数据仓库 TCHouse-D 与 Hudi 数据源中的表联合查询，进行更加复杂的分析操作。

本文档主要介绍该功能的使用方式和注意事项等。

说明：

该功能适用于腾讯云数据仓库 TCHouse-D 1.2及后续版本。

Hudi 目前仅支持 Copy On Write 表的 Snapshot Query，以及 Merge On Read 表的 Read Optimized Query。

创建方法

数据在 HDFS 上：

```
CREATE CATALOG hudi PROPERTIES (
    'type'='hms',
    'hive.metastore.uris' = 'thrift://172.21.xxx:7004',
    'hadoop.username' = 'hadoop',
    'dfs.nameservices'='your-nameservice',
    'dfs.ha.namenodes.your-nameservice'='nn1,nn2',
    'dfs.namenode.rpc-address.your-nameservice.nn1'='172.21.xxx:4007',
    'dfs.namenode.rpc-address.your-nameservice.nn2'='172.21.xxx:4007',
    'dfs.client.failover.proxy.provider.your-nameservice'='org.apache.hadoop.hdfs.s
);
```

数据在 COS 上：

```
CREATE CATALOG `hudi_cos` PROPERTIES (
"AWS_ENDPOINT" = "cos.ap-guangzhou.myqcloud.com",
"AWS_REGION" = "ap-guangzhou",
"hive.metastore.uris" = "thrift://172.16.xxxx:7004",
"type" = "hms",
"AWS_SECRET_KEY" = "Wu9ByN6g4D8seHj0770jJxxxx",
"AWS_ACCESS_KEY" = "AKIDaWJcCi9Rc4TqjV9hYHn9NRxxxxx"
);
```

类型匹配

支持的 Hudi 列类型与 Doris 对应关系如下表：

HMS Type	Doris Type	Comment

boolean	boolean	-
tinyint	tinyint	-
smallint	smallint	-
int	int	-
bigint	bigint	-
date	date	-
timestamp	datetime	-
float	float	-
double	double	-
char	char	-
varchar	varchar	-
decimal	decimal	-
array<type>	array<type>	支持 array 嵌套，如 array<array<int>>
map<KeyType, ValueType>	map<KeyType, ValueType>	暂不支持嵌套，KeyType 和 ValueType 需要为基础类型
struct<col1: Type1, col2: Type2, ...>	struct<col1: Type1, col2: Type2, ...>	暂不支持嵌套，Type1, Type2, ... 需要为基础类型
other	unsupported	-

查询用法

与普通的 Doris OLAP 表并无区别

```
select * from hudi_catalog_name.database_name.table_name;
```

Iceberg Catalog

最近更新时间：2024-07-04 15:43:38

Mutil Catalog Iceberg 提供了腾讯云数据仓库 TCHouse-D 直接访问 Iceberg 外部表的能力，外部表省去了繁琐的数据导入工作，并借助腾讯云数据仓库 TCHouse-D 本身的 OLAP 的能力来解决 Iceberg 表的数据分析问题：
支持 Iceberg 数据源接入 TCHouse-D。
支持腾讯云数据仓库 TCHouse-D 与 Iceberg 数据源中的表联合查询，进行更加复杂的分析操作。
本文档主要介绍该功能的使用方式和注意事项等。
说明：
该功能适用于腾讯云数据仓库 TCHouse-D 1.2及后续版本。
支持 Iceberg V1/V2 表格式，V2 格式仅支持 Position Delete 方式，不支持 Equality Delete。

创建方法

数据在 HDFS 上：

```
CREATE RESOURCE ice_hms_resource PROPERTIES (
'hive.metastore.uris' = 'thrift://172.16.xxxx:7004',
'type' = 'hms',
'dfs.nameservices'='HDFS1005116',     'dfs.ha.namenodes.HDFS1005116'='nn1,nn2',
'dfs.namenode.rpc-address.HDFS1005116.nn1'='172.16.xxxx:4007',
'dfs.namenode.rpc-address.HDFS1005116.nn2'='172.16.xxxx:4007',
'dfs.client.failover.proxy.provider.HDFS1005116'='org.apache.hadoop.hdfs.server.nam
);
```

数据在 COS 上：

```
CREATE CATALOG hive_cossab PROPERTIES (
"AWS_ENDPOINT" = "cos.ap-guangzhou.myqcloud.com",
"AWS_REGION" = "ap-guangzhou",
"AWS_SECRET_KEY" = "Wu9ByN6g4D8seHj0770jxxxxxxx",
"use_path_style" = "true",
"hive.metastore.uris" = "thrift://172.16.xxxx:7004",
"type" = "hms",
"AWS_ACCESS_KEY" = "AKIDaWJcCi9Rc4TqjV9hYHn9Nxxxxxxxxx"
);
```

类型匹配

支持的 Iceberg 列类型与 Doris 对应关系如下表：

HMS Type	Doris Type	Comment
boolean	boolean	-

tinyint	tinyint	-
smallint	smallint	-
int	int	-
bigint	bigint	-
date	date	-
timestamp	datetime	-
float	float	-
double	double	-
char	char	-
varchar	varchar	-
decimal	decimal	-
array<type>	array<type>	支持 array 嵌套，如 array<array<int>>
map<KeyType, ValueType>	map<KeyType, ValueType>	暂不支持嵌套，KeyType 和 ValueType 需要为基础类型
struct<col1: Type1, col2: Type2, ...>	struct<col1: Type1, col2: Type2, ...>	暂不支持嵌套，Type1, Type2, ... 需要为基础类型
other	unsupported	-

Time Travel

支持读取 Iceberg 表指定的 Snapshot。

每一次对 Iceberg 表的写操作都会产生一个新的快照。

默认情况下，读取请求只会读取最新版本的快照。

可以使用 FOR TIME AS OF 和 FOR VERSION AS OF 语句，根据快照 ID 或者快照产生的时间读取历史版本的数据。示例如下：

```
SELECT * FROM iceberg_tbl FOR TIME AS OF "2022-10-07 17:20:37";
SELECT * FROM iceberg_tbl FOR VERSION AS OF 868895038966572;
```

查询用法

与普通的 Doris OLAP 表并无区别。

```
select * from ice_catalog_name.database_name.table_name;
```

JDBC Catalog

最近更新时间：2024-07-04 15:44:22

说明：

该功能适用于腾讯云数据仓库 TCHouse-D 1.2及后续版本。

JDBC Catalog 通过标准 JDBC 协议，连接其他数据源。连接后，腾讯云数据仓库 TCHouse-D 会自动同步数据源下的 Database 和 Table 的元数据，以便快速访问这些外部数据。

创建 Catalog

MySQL

```
CREATE CATALOG jdbc_mysql PROPERTIES (
"type"="jdbc",
"user"="root",
"password"="123456",
"jdbc_url" = "jdbc:mysql://127.0.0.1:3306/demo",
"driver_url" = "mysql-connector-java-5.1.47.jar",
"driver_class" = "com.mysql.jdbc.Driver");
```

PostgreSQL

```
CREATE CATALOG jdbc_postgresql PROPERTIES (
"type"="jdbc",
"user"="root",
"password"="123456",
"jdbc_url" = "jdbc:postgresql://127.0.0.1:5449/demo",
"driver_url" = "postgresql-42.5.1.jar",
"driver_class" = "org.postgresql.Driver");
```

映射 PostgreSQL 时，Doris 的一个 Database 对应于 PostgreSQL 中指定 Catalog 下的一个 Schema（如示例中 jdbc_url 参数中 "demo"下的 schemas）。而 Doris 的 Database 下的 Table 则对应于 PostgreSQL 中，Schema 下的 Tables。即映射关系如下：

Doris	PostgreSQL
Catalog	Database
Database	Schema
Table	Table

通过 SQL 语句来获得 PG user 能够访问的所有 schema 并将其映射为 Doris 的 database

```
select nspname from pg_namespace where has_schema_privilege('<UserName>', nspname,
```

Oracle

```
CREATE CATALOG jdbc_oracle PROPERTIES (
"type"="jdbc",
"user"="root",
"password"="123456",
"jdbc_url" = "jdbc:oracle:thin:@127.0.0.1:1521:helowin",
"driver_url" = "ojdbc6.jar",
"driver_class" = "oracle.jdbc.driver.OracleDriver");
```

映射 Oracle 时，Doris 的一个 Database 对应于 Oracle 中的一个 User。而 Doris 的 Database 下的 Table 则对应于 Oracle 中，该 User 下的有权限访问的 Table。即映射关系如下：

Doris	Oracle
Catalog	Database
Database	User
Table	Table

Clickhouse

```
CREATE CATALOG jdbc_clickhouse PROPERTIES (
"type"="jdbc",
"user"="root",
"password"="123456",
"jdbc_url" = "jdbc:clickhouse://127.0.0.1:8123/demo",
"driver_url" = "clickhouse-jdbc-0.3.2-patch11-all.jar",
"driver_class" = "com.clickhouse.jdbc.ClickHouseDriver");
```

SQLServer

```
CREATE CATALOG sqlserver_catalog PROPERTIES (
"type"="jdbc",
"user"="SA",
"password"="Doris123456",
"jdbc_url" = "jdbc:sqlserver://localhost:1433;DataBaseName=doris_test",    "driver_
"driver_class" = "com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

映射 SQLServer 时，Doris 的一个 Database 对应于 SQLServer 中指定 Database（如示例中 jdbc_url 参数中的 "doris_test"）下的一个 Schema。而 Doris 的 Database 下的 Table 则对应于 SQLServer 中，Schema 下的 Tables。

即映射关系如下：

Doris	SQLServer
Catalog	Database
Database	Schema
Table	Table

Doris

JDBC Catalog 也支持连接另一个 Doris 数据库：

```
CREATE CATALOG doris_catalog PROPERTIES (
"type"="jdbc",
"user"="root",
"password"="123456",
"jdbc_url" = "jdbc:mysql://127.0.0.1:9030?useSSL=false",
"driver_url" = "mysql-connector-java-5.1.47.jar",
"driver_class" = "com.mysql.jdbc.Driver");
```

目前 JDBC Catalog 连接一个 Doris 数据库只支持用5.x版本的 JDBC jar 包。如果使用8.x JDBC jar 包，可能会出现列类型无法匹配问题。

参数说明

参数	是否必须	默认值	说明
user	是	-	对应数据库的用户名
password	是	-	对应数据库的密码
jdbc_url	是	-	JDBC 连接串
driver_url	是	-	JDBC Driver Jar 包名称*
driver_class	是	-	JDBC Driver Class 名称
only_specified_database	否	"false"	指定是否只同步指定的 database
lower_case_table_names	否	"false"	是否以小写的形式同步 JDBC 外部数据源的表名
include_database_list	否	""	当 only_specified_database=true 时，指定同步多个 database，以','分隔。db 名称是大小写敏感的
exclude_database_list	否	""	当 only_specified_database=true 时，指定不需要同步的多

			个 database，以','分割。db名称是大小写敏感的

驱动包路径

`driver_url` 可以通过以下三种方式指定：

1. 文件名。如 `mysql-connector-java-5.1.47.jar` 。需将 Jar 包预先存放在 FE 和 BE 部署目录的 `jdbc_drivers/` 目录下。系统会自动在这个目录下寻找。该目录的位置，也可以由 fe.conf 和 be.conf 中的 `jdbc_drivers_dir` 配置修改。
2. 本地绝对路径。如 `file:///path/to/mysql-connector-java-5.1.47.jar` 。需将 Jar 包预先存放在所有 FE/BE 节点指定的路径下。
3. Http 地址。如： `https://doris-community-test-1308700295.cos.ap-hongkong.myqcloud.com/jdbc_driver/mysql-connector-java-5.1.47.jar` 。系统会从这个 http 地址下载 Driver 文件。仅支持无认证的 http 服务。

指定同步数据库

`only_specified_database` ：在 jdbc 连接时可以指定链接到哪个 database/schema，如：MySQL 中 jdbc_url 中可以指定 database，pg 的 jdbc_url 中可以指定 currentSchema。

`include_database_list` ：当 only_specified_database=true 时，指定需要同步的 database，以','分割。默认为''，即不做任何过滤，同步所有 database。db 名称是大小写敏感的

`exclude_database_list` ：当 only_specified_database=true 时，指定不需要同步的多个 database，以','分割。默认为''，即不做任何过滤，同步所有database。db 名称是大小写敏感的。

当 `include_database_list` 和 `exclude_database_list` 有重合的 database 配置时， `exclude_database_list` 会优先生效。

如果使用该参数时连接 Oracle 数据库，要求使用 ojdbc8.jar 以上版本 jar 包。

数据查询

```
select * from mysql_catalog.mysql_database.mysql_table where t1 > 1000 and t2 ='ter
```

由于可能存在使用数据库内部的关键字作为字段名，为解决这种状况下仍能正确查询，所以在 SQL 语句中，会根据各个数据库的标准自动在字段名与表名上加上转义符。例如 MYSQL(``)、PostgreSQL("")、SQLServer([])、ORACLE("")，所以此时可能会造成字段名的大小写敏感，具体可以通过 explain sql，查看转义后下发到各个数据库的查询语句。

列类型映射

MySQL

MYSQL Type	Doris Type	Comment
BOOLEAN	BOOLEAN	-

TINYINT	TINYINT	-
SMALLINT	SMALLINT	-
MEDIUMINT	INT	-
INT	INT	-
BIGINT	BIGINT	-
UNSIGNED TINYINT	SMALLINT	Doris 没有 UNSIGNED 数据类型，所以扩大一个数量级
UNSIGNED MEDIUMINT	INT	Doris 没有 UNSIGNED 数据类型，所以扩大一个数量级
UNSIGNED INT	BIGINT	Doris 没有 UNSIGNED 数据类型，所以扩大一个数量级
UNSIGNED BIGINT	LARGEINT	-
FLOAT	FLOAT	-
DOUBLE	DOUBLE	-
DECIMAL	DECIMAL	-
DATE	DATE	-
TIMESTAMP	DATETIME	-
DATETIME	DATETIME	-
YEAR	SMALLINT	-
TIME	STRING	-
CHAR	CHAR	-
VARCHAR	VARCHAR	-
TINYTEXT、TEXT、MEDIUMTEXT、LONGTEXT、TINYBLOB、BLOB、MEDIUMBLOB、LONGBLOB、TINYSTRING、STRING、MEDIUMSTRING、LONGSTRING、BINARY、	STRING	-

VARBINARY、JSON、SET、BIT		
Other	UNSUPPORTED	-

PostgreSQL

POSTGRESQL Type	Doris Type	Comment
boolean	BOOLEAN	-
smallint/int2	SMALLINT	-
integer/int4	INT	-
bigint/int8	BIGINT	-
decimal/numeric	DECIMAL	-
real/float4	FLOAT	-
double precision	DOUBLE	-
smallserial	SMALLINT	-
serial	INT	-
bigserial	BIGINT	-
char	CHAR	-
varchar/text	STRING	-
timestamp	DATETIME	-
date	DATE	-
time	STRING	-
interval	STRING	-
point/line/lseg/box/path/polygon/circle	STRING	-
cidr/inet/macaddr	STRING	-
bit/bit(n)/bit varying(n)	STRING	bit 类型映射为 doris 的 STRING 类型，读出的数据是 true/false，而不是1/0
uuid/JSONB	STRING	-

Other	UNSUPPORTED	-

Oracle

ORACLE Type	Doris Type	Comment
number(p) / number(p,0)	TINYINT/SMALLINT/INT/BIGINT/LARGEINT	Doris会根据p的大小来选择对应的类型：p < 3 -> TINYINT; p < 5 -> SMALLINT; p < 10 -> INT; p < 19 -> BIGINT; p > 19 -> LARGEINT
number(p,s), [if(s>0 && p>s)]	DECIMAL(p,s)	-
number(p,s), [if(s>0 && p < s)]	DECIMAL(s,s)	-
number(p,s), [if(s<0)]	TINYINT/SMALLINT/INT/BIGINT/LARGEINT	s<0的情况下, Doris会将p设置为 p+|s|, 并进行和number(p) / number(p,0)一样的映射
number	-	Doris 目前不支持未指定 p 和 s 的 oracle 类型
decimal	DECIMAL	-
float/real	DOUBLE	-
DATE	DATETIME	-
TIMESTAMP	DATETIME	-
CHAR/NCHAR	STRING	-
VARCHAR2/NVARCHAR2	STRING	-
LONG/ RAW/ LONG RAW/ INTERVAL	STRING	-
Other	UNSUPPORTED	-

SQLServer

SQLServer Type	Doris Type	Comment

bit	BOOLEAN	-
tinyint	SMALLINT	SQLServer 的 tinyint 是无符号数，所以映射为 Doris 的 SMALLINT
smallint	SMALLINT	-
int	INT	-
bigint	BIGINT	-
real	FLOAT	-
float	DOUBLE	-
money	DECIMAL(19,4)	-
smallmoney	DECIMAL(10,4)	-
decimal/numeric	DECIMAL	-
date	DATE	-
datetime/datetime2/smalldatetime	DATETIMEV2	-
char/varchar/text/nchar/nvarchar/ntext	STRING	-
binary/varbinary	STRING	-
time/datetimeoffset	STRING	-
Other	UNSUPPORTED	-

Clickhouse

ClickHouse Type	Doris Type	Comment
Bool	BOOLEAN	-
String	STRING	-
Date/Date32	DATEV2	Jdbc Catlog 连接 ClickHouse 时默认使用 DATEV2类型
DateTime/DateTime64	DATETIMEV2	Jdbc Catlog 连接 ClickHouse 时默认使用 DATETIMEV2类型
Float32	FLOAT	-

Float64	DOUBLE	-
Int8	TINYINT	-
Int16/UInt8	SMALLINT	Doris 没有 UNSIGNED 数据类型，所以扩大一个数量级
Int32/UInt16	INT	Doris 没有 UNSIGNED 数据类型，所以扩大一个数量级
Int64/Uint32	BIGINT	Doris 没有 UNSIGNED 数据类型，所以扩大一个数量级
Int128/UInt64	LARGEINT	Doris 没有 UNSIGNED 数据类型，所以扩大一个数量级
Int256/UInt128/UInt256	STRING	Doris 没有这个数量级的数据类型，采用 STRING 处理
DECIMAL	DECIMAL/DECIMALV3/STRING	将根据 Doris DECIMAL 字段的（precision, scale）和 enable_decimal_conversion 开关选择用何种类型
Enum/IPv4/IPv6/UUID	STRING	在显示上 IPv4，IPv6会额外在数据最前面显示一个“/”，需要自己用 split_part 函数处理
Array	ARRAY	Array 内部类型适配逻辑参考上述类型，不支持嵌套类型
Other	UNSUPPORTED	-