尚硅谷大数据技术之Apache Doris

（Doris & 尚硅谷 联合出品）

版本：V1.0

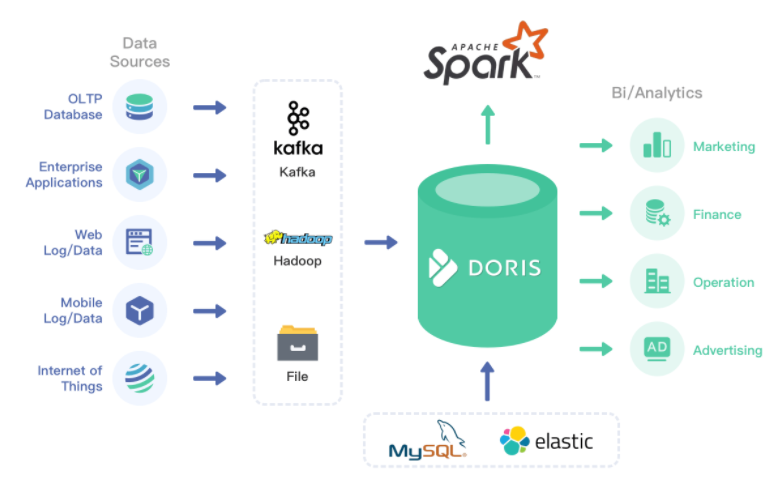
# 第1章 Doris简介

## 1.1 Doris概述

Apache Doris由百度大数据部研发（之前叫百度 Palo，2018年贡献到 Apache 社区后，更名为 Doris ），在百度内部，有超过200个产品线在使用，部署机器超过1000台，单一业务最大可达到上百 TB。

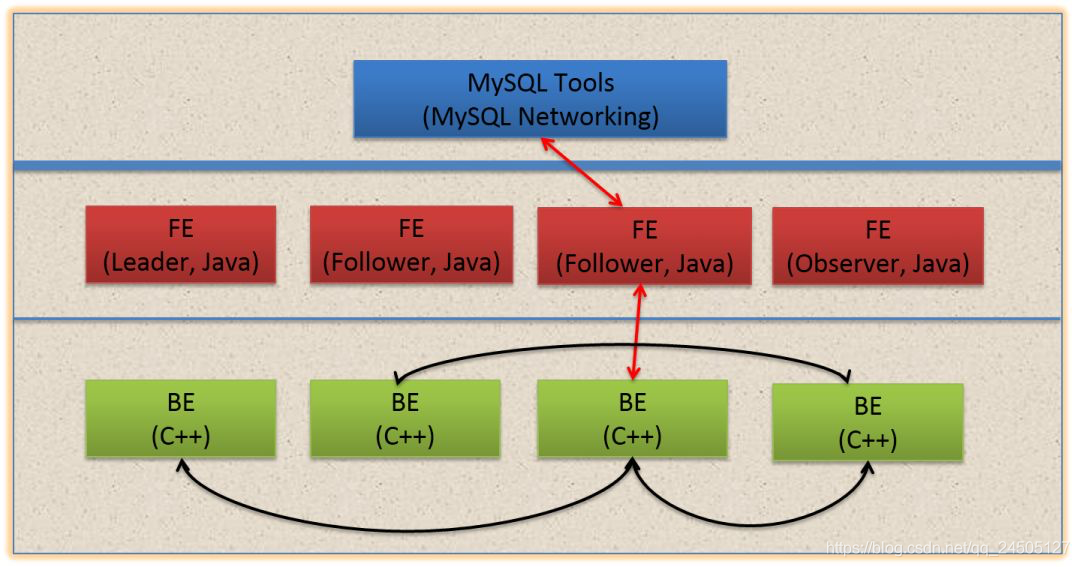
Apache Doris是一个现代化的MPP （Massively Parallel Processing，即大规模并行处理）分析型数据库产品。仅需亚秒级响应时间即可获得查询结果，有效地支持实时数据分析。Apache Doris的分布式架构非常简洁，易于运维，并且可以支持10PB以上的超大数据集。

Apache Doris可以满足多种数据分析需求，例如固定历史报表，实时数据分析，交互式数据分析和探索式数据分析等。





## 1.2 Doris架构



Doris 的架构很简洁，只设 FE(Frontend)、BE(Backend)两种角色、两个进程，不依赖于外部组件，方便部署和运维，FE、BE都可线性扩展。

* **FE（Frontend）**：存储、维护集群**元数据**；负责接收、解析查询请求，规划查询计划，调度查询执行，返回查询结果。主要有三个角色：

（1）Leader 和Follower：主要是用来达到元数据的高可用，保证单节点宕机的情况下，元数据能够实时地在线恢复，而不影响整个服务。

（2）Observer：用来扩展查询节点，同时起到元数据备份的作用。如果在发现集群压力非常大的情况下，需要去扩展整个查询的能力，那么可以加 observer 的节点。observer 不参与任何的写入，只参与读取。

* **BE（Backend）**：负责**物理数据的存储和计算**；依据 FE 生成的物理计划，分布式地执行查询。

数据的可靠性由 BE 保证，BE 会对整个数据存储多副本或者是三副本。副本数可根据需求动态调整。

* **MySQL Client**

Doris借助MySQL协议，用户使用任意MySQL的ODBC/JDBC以及MySQL的客户端，都可以直接访问Doris。

* **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，包括HDFS，S3，BOS等。

# 第2章 编译与安装

安装Doris，需要先通过源码编译，主要有两种方式：使用Docker开发镜像编译（推荐）、直接编译。

直接编译的方式，可以参考官网：<https://doris.apache.org/zh-CN/installing/compilation.html>

## 2.1 安装 Docker 环境

1）Docker要求CentOS系统的内核版本高于3.10 ，首先查看系统内核版本是否满足

uname -r

2）使用 root 权限登录系统，确保yum包更新到最新

sudo yum update -y

3）假如安装过旧版本，先卸载旧版本

sudo yum remove docker docker-common docker-selinux docker-engine

4）安装yum-util工具包和devicemapper驱动依赖

sudo yum install -y yum-utils device-mapper-persistent-data lvm2

5）设置yum源（加速yum下载速度）

sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo

如果连接超时，可以使用alibaba的镜像源：

sudo yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

6）查看所有仓库中所有docker版本，并选择特定版本安装，一般可直接安装最新版

yum list docker-ce --showduplicates | sort -r

7）安装docker

（1）安装最新稳定版本的方式：

sudo yum install docker-ce -y #安装的是最新稳定版本，因为repo中默认只开启stable仓库

（2）安装指定版本的方式：

sudo yum install <FQPN> -y

# 例如：

sudo yum install docker-ce-20.10.11.ce -y

8）启动并加入开机启动

sudo systemctl start docker #启动docker

sudo systemctl enable docker #加入开机自启动

9）查看Version，验证是否安装成功

docker version

若出现Client和Server两部分内容，则证明安装成功。

## 2.2 使用 Docker 开发镜像编译

1）下载源码并解压

通过wget下载（或者手动上传下载好的压缩包）。

wget https://dist.apache.org/repos/dist/dev/incubator/doris/0.15/0.15.0-rc04/apache-doris-0.15.0-incubating-src.tar.gz

解压到/opt/software/

tar -zxvf apache-doris-0.15.0-incubating-src.tar.gz -C /opt/software

2）下载 Docker 镜像

docker pull apache/incubator-doris:build-env-for-0.15.0

可以通过以下命令查看镜像是否下载完成。

docker images

3）挂载本地目录运行镜像

以挂载本地Doris源码目录的方式运行镜像，这样编译的产出二进制文件会存储在宿主机中，不会因为镜像退出而消失。同时将镜像中maven的 .m2 目录挂载到宿主机目录，以防止每次启动镜像编译时，重复下载maven的依赖库。

docker run -it \

-v /opt/software/.m2:/root/.m2 \

-v /opt/software/apache-doris-0.15.0-incubating-src/:/root/apache-doris-0.15.0-incubating-src/ \

apache/incubator-doris:build-env-for-0.15.0

4）切换到JDK 8

alternatives --set java java-1.8.0-openjdk.x86\_64

alternatives --set javac java-1.8.0-openjdk.x86\_64

export JAVA\_HOME=/usr/lib/jvm/java-1.8.0

5）准备Maven依赖

编译过程会下载很多依赖，可以将我们准备好的doris-repo.tar.gz解压到Docker挂载的对应目录，来避免下载依赖的过程，加速编译。

tar -zxvf doris-repo.tar.gz -C /opt/software

也可以通过指定阿里云镜像仓库来加速下载：

vim /opt/software/apache-doris-0.15.0-incubating-src/fe/pom.xml

在<repositories>标签下添加：

<repository>

<id>aliyun</id>

<url>http://maven.aliyun.com/nexus/content/groups/public/</url>

</repository>

vim /opt/software/apache-doris-0.15.0-incubating-src/be/pom.xml

在<repositories>标签下添加：

<repository>

<id>aliyun</id>

<url>http://maven.aliyun.com/nexus/content/groups/public/</url>

</repository>

6）编译Doris

sh build.sh

如果是第一次使用 build-env-for-0.15.0 或之后的版本，第一次编译的时候要使用如下命令：

sh build.sh --clean --be --fe --ui

因为build-env-for-0.15.0版本镜像升级了thrift(0.9 -> 0.13)，需要通过--clean命令强制使用新版本的thrift生成代码文件，否则会出现不兼容的代码。

## 2.3 安装要求

### 2.3.1 软硬件需求

1）Linux操作系统要求

|  |  |
| --- | --- |
| linux系统 | 版本 |
| Centos | 7.1及以上 |
| Ubuntu | 16.04及以上 |

2）软件需求

|  |  |
| --- | --- |
| 软件 | 版本 |
| Java | 1.8及以上 |
| GCC | 4.8.2及以上 |

3）开发测试环境

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 模块 | CPU | 内存 | 磁盘 | 网络 | 实例数量 |
| Frontend | 8核+ | 8GB | SSD 或 SATA，10GB+ \* | 千兆网卡 | 1 |
| Backend | 8核+ | 16GB | SSD 或 SATA，50GB+ \* | 千兆网卡 | 1-3\* |

4）生产环境

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 模块 | CPU | 内存 | 磁盘 | 网络 | 实例数量 |
| Frontend | 16核+ | 64GB | SSD 或 SATA，100GB+ \* | 万兆网卡 | 1-5\* |
| Backend | 16核+ | 64GB | SSD 或 SATA，100GB+ \* | 万兆网卡 | 10-100\* |

**5）注意事项**

（1）FE的磁盘空间主要用于存储元数据，包括日志和image。通常从几百MB到几个GB不等。

（2）BE的磁盘空间主要用于存放用户数据，总磁盘空间按用户总数据量\* 3（3副本）计算，然后再预留额外40%的空间用作后台compaction以及一些中间数据的存放。

（3）一台机器上可以部署多个BE实例，但是只能部署一个 FE。如果需要 3 副本数据，那么至少需要 3 台机器各部署一个BE实例（而不是1台机器部署3个BE实例）。多个FE所在服务器的时钟必须保持一致（允许最多5秒的时钟偏差）

（4）测试环境也可以仅适用一个BE进行测试。实际生产环境，BE实例数量直接决定了整体查询延迟。

（5）所有部署节点关闭Swap。

（6）FE节点数据至少为1（1个Follower）。当部署1个Follower和1个Observer时，可以实现读高可用。当部署3个Follower时，可以实现读写高可用（HA）。

（7）Follower的数量必须为奇数，Observer 数量随意。

（8）根据以往经验，当集群可用性要求很高时（比如提供在线业务），可以部署3个 Follower和1-3个Observer。如果是离线业务，建议部署1个Follower和1-3个Observer。

（9）Broker是用于访问外部数据源（如HDFS）的进程。通常，在每台机器上部署一个 broker实例即可。

### 2.3.2 默认端口

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实例名称 | 端口名称 | 默认端口 | 通讯方向 | 说明 |
| BE | be\_prot | 9060 | FE-->BE | BE上thrift server的端口  用于接收来自FE 的请求 |
| BE | webserver\_port | 8040 | BE<-->FE | BE上的http server端口 |
| BE | heartbeat\_service\_port | 9050 | FE-->BE | BE上心跳服务端口  用于接收来自FE的心跳 |
| BE | brpc\_prot\* | 8060 | FE<-->BE  BE<-->BE | BE上的brpc端口  用于BE之间通信 |
| FE | http\_port | 8030 | FE<-->FE  用户<--> FE | FE上的http\_server端口 |
| FE | rpc\_port | 9020 | BE-->FE  FE<-->FE | FE上thirt server端口号 |
| FE | query\_port | 9030 | 用户<--> FE | FE上的mysql server端口 |
| FE | edit\_log\_port | 9010 | FE<-->FE | FE上bdbje之间通信用的端口 |
| Broker | broker\_ipc\_port | 8000 | FE-->BROKER  BE-->BROKER | Broker上的thrift server  用于接收请求 |

当部署多个FE实例时，要保证FE的http\_port配置相同。

部署前请确保各个端口在应有方向上的访问权限。

## 2.4 集群部署

|  |  |  |
| --- | --- | --- |
| 主机1 | 主机2 | 主机3 |
| FE(LEADER) | FE(FOLLOWER) | FE(OBSERVER) |
| BE | BE | BE |
| BROKER | BROKER | BROKER |

生产环境建议FE和BE分开。

### 2.4.1 创建目录并拷贝编译后的文件

1）创建目录并拷贝编译后的文件

mkdir /opt/module/apache-doris-0.15.0

cp -r /opt/software/apache-doris-0.15.0-incubating-src/output /opt/module/apache-doris-0.15.0

2）修改可打开文件数（每个节点）

sudo vim /etc/security/limits.conf

\* soft nofile 65535

\* hard nofile 65535

\* soft nproc 65535

\* hard nproc 65535

重启永久生效，也可以用ulimit -n 65535临时生效。

### 2.4.2 部署 FE 节点

1）创建fe元数据存储的目录

mkdir /opt/module/apache-doris-0.15.0/doris-meta

2）修改fe的配置文件

vim /opt/module/apache-doris-0.15.0/fe/conf/fe.conf

#配置文件中指定元数据路径：

meta\_dir = /opt/module/apache-doris-0.15.0/doris-meta

#修改绑定ip（每台机器修改成自己的ip）

priority\_networks = 192.168.8.101/24

注意：

* 生产环境强烈建议单独指定目录不要放在Doris安装目录下，最好是单独的磁盘（如果有SSD最好）。
* 如果机器有多个ip, 比如内网外网, 虚拟机docker等, 需要进行ip绑定，才能正确识别。
* JAVA\_OPTS 默认 java 最大堆内存为 4GB，建议生产环境调整至 8G 以上。

3）启动hadoop1的FE

/opt/module/apache-doris-0.15.0/fe/bin/start\_fe.sh --daemon

### 2.4.3 配置 BE节点

1）分发BE

scp -r /opt/module/apache-doris-0.15.0/be hadoop2:/opt/module

scp -r /opt/module/apache-doris-0.15.0/be hadoop3:/opt/module

2）创建BE数据存放目录（每个节点）

mkdir /opt/module/apache-doris-0.15.0/doris-storage1

mkdir /opt/module/apache-doris-0.15.0/doris-storage2

3）修改BE的配置文件（每个节点）

vim /opt/module/apache-doris-0.15.0/be/conf/be.conf

#配置文件中指定数据存放路径：

storage\_root\_path = /opt/module/apache-doris-0.15.0/doris-storage1;/opt/module/apache-doris-0.15.0/doris-storage2

#修改绑定ip（每台机器修改成自己的ip）

priority\_networks = 192.168.8.101/24

注意：

* storage\_root\_path默认在be/storage下，需要手动创建该目录。多个路径之间使用英文状态的分号;分隔（最后一个目录后不要加）。
* 可以通过路径区别存储目录的介质，HDD或SSD。可以添加容量限制在每个路径的末尾，通过英文状态逗号，隔开，如：

storage\_root\_path=/home/disk1/doris.HDD,50;/home/disk2/doris.SSD,10;/home/disk2/doris

说明：

/home/disk1/doris.HDD,50，表示存储限制为50GB，HDD;

/home/disk2/doris.SSD,10，存储限制为10GB，SSD；

/home/disk2/doris，存储限制为磁盘最大容量，默认为HDD

* 如果机器有多个IP, 比如内网外网, 虚拟机docker等, 需要进行IP绑定，才能正确识别。

### 2.4.4 在FE中添加所有BE节点

BE节点需要先在FE中添加，才可加入集群。可以使用mysql-client连接到FE。

1）安装MySQL Client

（1）创建目录

mkdir /opt/software/mysql-client/

（2）上传相关以下三个rpm包到/opt/software/mysql-client/

* mysql-community-client-5.7.28-1.el7.x86\_64.rpm
* mysql-community-common-5.7.28-1.el7.x86\_64.rpm
* mysql-community-libs-5.7.28-1.el7.x86\_64.rpm

（3）检查当前系统是否安装过MySQL

sudo rpm -qa|grep mariadb

#如果存在，先卸载

sudo rpm -e --nodeps mariadb mariadb-libs mariadb-server

（4）安装

rpm -ivh /opt/software/mysql-client/\*

2）使用MySQL Client连接FE

mysql -h hadoop1 -P 9030 -uroot

默认root无密码，通过以下命令修改root密码。

SET PASSWORD FOR 'root' = PASSWORD('000000');

3）添加BE

ALTER SYSTEM ADD BACKEND "hadoop1:9050";

ALTER SYSTEM ADD BACKEND "hadoop2:9050";

ALTER SYSTEM ADD BACKEND "hadoop3:9050";

4）查看BE状态

SHOW PROC '/backends';

### 2.4.5 启动 BE

1）启动BE（每个节点）

/opt/module/apache-doris-0.15.0/be/bin/start\_be.sh --daemon

2）查看BE状态

mysql -h hadoop1 -P 9030 -uroot -p

SHOW PROC '/backends';

Alive为true表示该BE节点存活。

### 2.4.6 部署 FS\_Broker（可选）

Broker以插件的形式，独立于Doris部署。如果需要从第三方存储系统导入数据，需要部署相应的 Broker，默认提供了读取 HDFS、百度云BOS及Amazon S3的fs\_broker。fs\_broker是无状态的，建议每一个FE和BE节点都部署一个Broker。

1）编译FS\_BROKER并拷贝文件

（1）进入源码目录下的fs\_brokers目录，使用sh build.sh进行编译

（2）拷贝源码 fs\_broker 的 output 目录下的相应 Broker 目录到需要部署的所有节点上，改名为: apache\_hdfs\_broker。建议和 BE 或者 FE 目录保持同级。

方法同2.2。

2）启动Broker

/opt/module/apache-doris-0.15.0/apache\_hdfs\_broker/bin/start\_broker.sh --daemon

3）添加 Broker

要让 Doris 的 FE 和 BE 知道 Broker 在哪些节点上，通过 sql 命令添加 Broker 节点列表。

（1）使用 mysql-client 连接启动的 FE，执行以下命令：

mysql -h hadoop1 -P 9030 -uroot -p

ALTER SYSTEM ADD BROKER broker\_name "hadoop1:8000","hadoop2:8000","hadoop3:8000";

其中 broker\_host 为 Broker 所在节点 ip；broker\_ipc\_port 在 Broker 配置文件中的conf/apache\_hdfs\_broker.conf。

4）查看 Broker 状态

使用 mysql-client 连接任一已启动的FE，执行以下命令查看Broker状态：

SHOW PROC "/brokers";

注：在生产环境中，所有实例都应使用守护进程启动，以保证进程退出后，会被自动拉起，如Supervisor（opens new window）。如需使用守护进程启动，在0.9.0及之前版本中，需要修改各个 start\_xx.sh脚本，去掉最后的 & 符号。从0.10.0版本开始，直接调用sh start\_xx.sh启动即可。

## 2.5 扩容和缩容

Doris可以很方便的扩容和缩容FE、BE、Broker实例。

### 2.5.1 FE 扩容和缩容

可以通过将FE扩容至3个以上节点来实现FE的高可用。

1）使用MySQL登录客户端后，可以使用sql命令查看FE状态，目前就一台FE

mysql -h hadoop1 -P 9030 -uroot -p

SHOW PROC '/frontends';

也可以通过页面访问进行监控，访问8030，账户为root，密码默认为空不用填写。

2）增加FE节点

FE分为Leader，Follower和Observer三种角色。 默认一个集群，只能有一个Leader，可以有多个Follower和Observer。其中Leader和Follower组成一个Paxos选择组，如果 Leader宕机，则剩下的Follower 会自动选出新的Leader，保证写入高可用。Observer 同步 Leader的数据，但是不参加选举。

如果只部署一个FE，则FE 默认就是Leader。在此基础上，可以添加若干Follower和 Observer。

ALTER SYSTEM ADD FOLLOWER "hadoop2:9010";

ALTER SYSTEM ADD OBSERVER "hadoop3:9010";

3）配置及启动Follower和Observer

第一次启动时，启动命令需要添加参--helper leader主机: edit\_log\_port：

（1）分发FE，修改FE的配置（同2.4.2）

scp -r /opt/module/apache-doris-0.15.0/fe hadoop2:/opt/module/ apache-doris-0.15.0

scp -r /opt/module/apache-doris-0.15.0/fe hadoop3:/opt/module/ apache-doris-0.15.0

（2）在hadoop2启动Follower

/opt/module/apache-doris-0.15.0/fe/bin/start\_fe.sh --helper hadoop1:9010 --daemon

（3）在hadoop3启动Observer

/opt/module/apache-doris-0.15.0/fe/bin/start\_fe.sh --helper hadoop1:9010 --daemon

4）查看运行状态

使用mysql-client 连接到任一已启动的FE。

SHOW PROC '/frontends';

5）删除 FE 节点命令

ALTER SYSTEM DROP FOLLOWER[OBSERVER] "fe\_host:edit\_log\_port";

注意：删除 Follower FE 时，确保最终剩余的 Follower（包括 Leader）节点为奇数。

### 2.5.2 BE 扩容和缩容

1）增加BE节点

在MySQL客户端，通过 ALTER SYSTEM ADD BACKEND 命令增加BE节点。

2）DROP方式删除BE节点（不推荐）

ALTER SYSTEM DROP BACKEND "be\_host:be\_heartbeat\_service\_port";

注意：DROP BACKEND 会直接删除该BE，并且其上的数据将不能再恢复！！！所以我们强烈不推荐使用 DROP BACKEND 这种方式删除BE节点。当你使用这个语句时，会有对应的防误操作提示。

3）DECOMMISSION 方式删除BE节点（推荐）

ALTER SYSTEM DECOMMISSION BACKEND "be\_host:be\_heartbeat\_service\_port";

* 该命令用于安全删除BE节点。命令下发后，Doris 会尝试将该BE上的数据向其他BE节点迁移，当所有数据都迁移完成后，Doris会自动删除该节点。
* 该命令是一个异步操作。执行后，可以通过 SHOW PROC '/backends'; 看到该 BE 节点的isDecommission状态为true。表示该节点正在进行下线。
* 该命令不一定执行成功。比如剩余BE存储空间不足以容纳下线BE上的数据，或者剩余机器数量不满足最小副本数时，该命令都无法完成，并且BE会一直处于 isDecommission为true的状态。
* DECOMMISSION的进度，可以通过SHOW PROC '/backends'; 中的TabletNum查看，如果正在进行，TabletNum将不断减少。
* 该操作可以通过如下命令取消：

CANCEL DECOMMISSION BACKEND "be\_host:be\_heartbeat\_service\_port";

取消后，该BE上的数据将维持当前剩余的数据量。后续Doris重新进行负载均衡。

### 2.5.3 Broker 扩容缩容

Broker实例的数量没有硬性要求。通常每台物理机部署一个即可。Broker 的添加和删除可以通过以下命令完成：

ALTER SYSTEM ADD BROKER broker\_name "broker\_host:broker\_ipc\_port"; ALTER SYSTEM DROP BROKER broker\_name "broker\_host:broker\_ipc\_port"; ALTER SYSTEM DROP ALL BROKER broker\_name;

Broker是无状态的进程，可以随意启停。当然，停止后，正在其上运行的作业会失败，重试即可。

# 第3章 数据表的创建

## 3.1 创建用户和数据库

1）创建test用户

mysql -h hadoop1 -P 9030 -uroot -p

create user 'test' identified by 'test';

2）创建数据库

create database test\_db;

3）用户授权

grant all on test\_db to test;

## 3.2 基本概念

在Doris中，数据都以关系表（Table）的形式进行逻辑上的描述。

### 3.2.1 Row & Column

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。

* 在默认的数据模型中，Column只分为排序列和非排序列。存储引擎会按照排序列对数据进行排序存储，并建立稀疏索引，以便在排序数据上进行快速查找。
* 而在聚合模型中，Column可以分为两大类：Key和Value。从业务角度看，Key 和 Value可以分别对应维度列和指标列。从聚合模型的角度来说，Key 列相同的行，会聚合成一行。其中Value列的聚合方式由用户在建表时指定。

### 3.2.2 Partition & Tablet

在Doris的存储引擎中，用户数据首先被划分成若干个分区（Partition），划分的规则通常是按照用户指定的分区列进行范围划分，比如按时间划分。而在每个分区内，数据被进一步的按照Hash的方式分桶，分桶的规则是要找用户指定的分桶列的值进行Hash后分桶。每个分桶就是一个数据分片（Tablet），也是数据划分的最小逻辑单元。

* Tablet之间的数据是没有交集的，独立存储的。Tablet也是数据移动、复制等操作的最小物理存储单元。
* Partition可以视为是逻辑上最小的管理单元。数据的导入与删除，都可以或仅能针对一个 Partition 进行。

## 3.3 建表示例

### 3.3.1 建表语法

使用 CREATE TABLE 命令建立一个表(Table)。更多详细参数可以查看：

HELP CREATE TABLE;

建表语法：

CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [database.]table\_name

(column\_definition1[, column\_definition2, ...]

[, index\_definition1[, index\_definition12,]])

[ENGINE = [olap|mysql|broker|hive]]

[key\_desc]

[COMMENT "table comment"];

[partition\_desc]

[distribution\_desc]

[rollup\_index]

[PROPERTIES ("key"="value", ...)]

[BROKER PROPERTIES ("key"="value", ...)];

Doris 的建表是一个同步命令，命令返回成功，即表示建表成功。

Doris支持支持单分区和复合分区两种建表方式。

1）复合分区：既有分区也有分桶

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。

2）单分区：只做 HASH 分布，即只分桶。

### 3.3.2 字段类型

|  |  |  |
| --- | --- | --- |
| TINYINT | 1字节 | 范围：-2^7 + 1 ~ 2^7 - 1 |
| SMALLINT | 2字节 | 范围：-2^15 + 1 ~ 2^15 - 1 |
| INT | 4字节 | 范围：-2^31 + 1 ~ 2^31 - 1 |
| BIGINT | 8字节 | 范围：-2^63 + 1 ~ 2^63 - 1 |
| LARGEINT | 16字节 | 范围：-2^127 + 1 ~ 2^127 - 1 |
| FLOAT | 4字节 | 支持科学计数法 |
| DOUBLE | 12字节 | 支持科学计数法 |
| DECIMAL[(precision, scale)] | 16字节 | 保证精度的小数类型。默认是 DECIMAL(10, 0)  precision: 1 ~ 27  scale: 0 ~ 9  其中整数部分为 1 ~ 18  不支持科学计数法 |
| DATE | 3字节 | 范围：0000-01-01 ~ 9999-12-31 |
| DATETIME | 8字节 | 范围：0000-01-01 00:00:00 ~ 9999-12-31 23:59:59 |
| CHAR[(length)] |  | 定长字符串。长度范围：1 ~ 255。默认为1 |
| VARCHAR[(length)] |  | 变长字符串。长度范围：1 ~ 65533 |
| BOOLEAN |  | 与TINYINT一样，0代表false，1代表true |
| HLL | 1~16385个字节 | hll列类型，不需要指定长度和默认值、长度根据数据的聚合  程度系统内控制，并且HLL列只能通过配套的hll\_union\_agg、Hll\_cardinality、hll\_hash进行查询或使用 |
| BITMAP |  | bitmap列类型，不需要指定长度和默认值。表示整型的集合，元素最大支持到2^64 - 1 |
| STRING |  | 变长字符串，0.15版本支持，最大支持2147483643 字节（2GB-4），长度还受be 配置`string\_type\_soft\_limit`, 实际能存储的最大长度取两者最小值。只能用在value 列，不能用在 key 列和分区、分桶列 |

注：聚合模型在定义字段类型后，可以指定字段的agg\_type 聚合类型，如果不指定，则该列为 key 列。否则，该列为 value 列, 类型包括：SUM、MAX、MIN、REPLACE。

### 3.3.2 建表示例

我们以一个建表操作来说明 Doris 的数据划分。

3.3.2.1 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 "用户最后一次访问时间",

`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",

"storage\_medium" = "SSD",

"storage\_cooldown\_time" = "2018-01-01 12:00:00"

);

3.3.2.2 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) 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 "用户最后一次访问时间",

`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",

"storage\_medium" = "SSD",

"storage\_cooldown\_time" = "2018-01-01 12:00:00"

);

## 3.4 数据划分

以3.3.2的建表示例来理解。

### 3.4.1 列定义

以AGGREGATE KEY数据模型为例进行说明。更多数据模型参阅Doris数据模型。

列的基本类型，可以通过在mysql-client中执行HELP CREATE TABLE; 查看。

AGGREGATE KEY数据模型中，所有没有指定聚合方式（SUM、REPLACE、MAX、MIN）的列视为Key列。而其余则为Value列。

定义列时，可参照如下建议：

* Key 列必须在所有Value列之前。
* 尽量选择整型类型。因为整型类型的计算和查找比较效率远高于字符串。
* 对于不同长度的整型类型的选择原则，遵循够用即可。
* 对于VARCHAR和STRING类型的长度，遵循 够用即可。
* 所有列的总字节长度（包括 Key和Value）不能超过100KB。

### 3.4.2 分区与分桶

Doris支持两层的数据划分。第一层是 Partition，支持 Range和List的划分方式。第二层是 Bucket（Tablet），仅支持Hash的划分方式。

也可以仅使用一层分区。使用一层分区时，只支持Bucket划分。

3.4.2.1 Partition

* Partition列可以指定一列或多列。分区类必须为KEY列。多列分区的使用方式在后面介绍。
* 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。

**1） Range 分区**

分区列通常为时间列，以方便的管理新旧数据。不可添加范围重叠的分区。

Partition指定范围的方式

* 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。**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**。
* VALUES [...) 指定同时指定上下界，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：

（1）如上 expamle\_range\_tbl 示例，当建表完成后，会自动生成如下3个分区：

p201701: [MIN\_VALUE, 2017-02-01)

p201702: [2017-02-01, 2017-03-01)

p201703: [2017-03-01, 2017-04-01)

（2）增加一个分区 p201705 VALUES LESS THAN ("2017-06-01")，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)

（3）此时删除分区 p201703，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。

（4）继续删除分区 p201702，分区结果如下：

p201701: [MIN\_VALUE, 2017-02-01)

p201705: [2017-04-01, 2017-06-01)

空洞范围变为：[2017-02-01, 2017-04-01)

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

可以看到空洞范围缩小为：[2017-03-01, 2017-04-01)

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

即出现了一个新的空洞：[2017-01-01, 2017-02-01)

**2）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。不可添加范围重叠的分区。

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。

（1）如上example\_list\_tbl示例，当建表完成后，会自动生成如下3个分区：

p\_cn: ("Beijing", "Shanghai", "Hong Kong")

p\_usa: ("New York", "San Francisco")

p\_jp: ("Tokyo")

（2）增加一个分区p\_uk VALUES IN ("London")，分区结果如下：

p\_cn: ("Beijing", "Shanghai", "Hong Kong")

p\_usa: ("New York", "San Francisco")

p\_jp: ("Tokyo")

p\_uk: ("London")

（3）删除分区p\_jp，分区结果如下：

p\_cn: ("Beijing", "Shanghai", "Hong Kong")

p\_usa: ("New York", "San Francisco")

p\_uk: ("London")

* + - 1. Bucket

（1）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。

（2）分桶列可以是多列，但**必须为 Key 列**。分桶列可以和 Partition 列相同或不同。

（3）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 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。

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。

（4）分桶的数量理论上没有上限。

3.4.2.3 使用复合分区的场景

以下场景推荐使用复合分区

（1）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。

（2）历史数据删除需求：如有删除历史数据的需求（比如仅保留最近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。

（3）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。

3.4.2.4 多列分区

Doris 支持指定多列作为分区列，示例如下：

**1）Range 分区**

PARTITION BY RANGE(`date`, `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")

)

指定 `date`(DATE 类型) 和 `id`(INT 类型) 作为分区列。以上示例最终得到的分区如下：

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 `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

**2）List 分区**

PARTITION BY LIST(`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"))

)

指定 `id`(INT 类型) 和 `city`(VARCHAR 类型) 作为分区列。最终得到的分区如下：

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

### 3.4.3 PROPERTIES

在建表语句的最后 PROPERTIES 中，可以指定以下两个参数：

3.4.3.1 replication\_num

每个 Tablet 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 Tablet 的副本数量。

副本数量可以在运行时修改。强烈建议保持奇数。

**最大副本数量取决于集群中独立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。

对于一些小，并且更新不频繁的维度表，可以考虑设置更多的副本数。这样在Join查询时，可以有更大的概率进行本地数据Join。

3.4.3.2 storage\_medium & storage\_cooldown\_time

BE的数据存储目录可以显式的指定为 SSD 或者 HDD（通过 .SSD 或者 .HDD 后缀区分）。建表时，可以统一指定所有 Partition 初始存储的介质。注意，后缀作用是显式指定磁盘介质，而不会检查是否与实际介质类型相符。

默认初始存储介质可通过fe的配置文件 fe.conf 中指定 default\_storage\_medium=xxx，如果没有指定，则默认为HDD。如果指定为 SSD，则数据初始存放在SSD上。

如果没有指定storage\_cooldown\_time，则默认30天后，数据会从SSD自动迁移到HDD 上。如果指定了storage\_cooldown\_time，则在到达storage\_cooldown\_time时间后，数据才会迁移。

注意，当指定storage\_medium时，如果FE参数enable\_strict\_storage\_medium\_check为 False 该参数只是一个“尽力而为”的设置。即使集群内没有设置SSD存储介质，也不会报错，而是自动存储在可用的数据目录中。 同样，如果SSD介质不可访问、空间不足，都可能导致数据初始直接存储在其他可用介质上。而数据到期迁移到HDD时，如果HDD介质不可访问、空间不足，也可能迁移失败（但是会不断尝试）。 如果FE参数 enable\_strict\_storage\_medium\_check为True则当集群内没有设置SSD存储介质时，会报错 Failed to find enough host in all backends with storage medium is SSD。

### 3.4.4 ENGINE

本示例中，ENGINE的类型是olap，即默认的ENGINE类型。在Doris中，只有这个 ENGINE 类型是由Doris负责数据管理和存储的。其他ENGINE类型，如mysql、broker、es等等，本质上只是对外部其他数据库或系统中的表的映射，以保证Doris可以读取这些数据。而Doris本身并不创建、管理和存储任何非olap ENGINE 类型的表和数据。

### 3.4.5 其他

`IF NOT EXISTS` 表示如果没有创建过该表，则创建。注意这里只判断表名是否存在，而不会判断新建表结构是否与已存在的表结构相同

## 3.5 数据模型

Doris 的数据模型主要分为3类：Aggregate、Uniq、Duplicate

### 3.5.1 Aggregate 模型

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的称为 Key，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：

* SUM：求和，多行的Value进行累加。
* REPLACE：替代，下一批数据中的Value会替换之前导入过的行中的 Value。

REPLACE\_IF\_NOT\_NULL ：当遇到null值则不更新。

* MAX：保留最大值。
* MIN：保留最小值。

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。比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。（可参阅聚合模型的局限性一节获得更多详情。）

3.5.1.1 示例一：导入数据聚合

1）建表

CREATE TABLE IF NOT EXISTS test\_db.example\_site\_visit

(

`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 "用户最后一次访问时间",

`last\_visit\_date\_not\_null` DATETIME REPLACE\_IF\_NOT\_NULL DEFAULT "1970-01-01 00:00:00" COMMENT "用户最后一次访问时间",

`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 10;

2）插入数据

insert into test\_db.example\_site\_visit values\

(10000,'2017-10-01','北京',20,0,'2017-10-01 06:00:00','2017-10-01 06:00:00',20,10,10),\

(10000,'2017-10-01','北京',20,0,'2017-10-01 07:00:00','2017-10-01 07:00:00',15,2,2),\

(10001,'2017-10-01','北京',30,1,'2017-10-01 17:05:45','2017-10-01 07:00:00',2,22,22),\

(10002,'2017-10-02','上海',20,1,'2017-10-02 12:59:12',null,200,5,5),\

(10003,'2017-10-02','广州',32,0,'2017-10-02 11:20:00','2017-10-02 11:20:00',30,11,11),\

(10004,'2017-10-01','深圳',35,0,'2017-10-01 10:00:15','2017-10-01 10:00:15',100,3,3),\

(10004,'2017-10-03','深圳',35,0,'2017-10-03 10:20:22','2017-10-03 10:20:22',11,6,6);

注意：Insert into 单条数据这种操作在Doris里只能演示不能在生产使用，会引发写阻塞。

3）查看表

select \* from test\_db.example\_site\_visit;

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。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。

3.5.1.2 示例二：保留明细数据

1）建表

CREATE TABLE IF NOT EXISTS test\_db.example\_site\_visit2

(

`user\_id` LARGEINT NOT NULL COMMENT "用户id",

`date` DATE NOT NULL COMMENT "数据灌入日期时间",

`timestamp` DATETIME COMMENT "数据灌入时间，精确到秒",

`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 "用户最后一次访问时间",

`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`, `timestamp`, `city`, `age`, `sex`)

DISTRIBUTED BY HASH(`user\_id`) BUCKETS 10;

2）插入数据

insert into test\_db.example\_site\_visit2 values(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);

3）查看表

select \* from test\_db.example\_site\_visit2;

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。

3.5.1.3 示例三：导入数据与已有数据聚合

1）往实例一中继续插入数据

insert into test\_db.example\_site\_visit values(10004,'2017-10-03','深圳',35,0,'2017-10-03 11:22:00',null,44,19,19),\

(10005,'2017-10-03','长沙',29,1,'2017-10-03 18:11:02','2017-10-03 18:11:02',3,1,1);

2）查看表

select \* from test\_db.example\_site\_visit;

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。

### 3.5.2 Uniq 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 Uniq 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。

1）建表

CREATE TABLE IF NOT EXISTS test\_db.user

(

`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 10;

2）插入数据

insert into test\_db.user values\

(10000,'wuyanzu','北京',18,0,12345678910,'北京朝阳区','2017-10-01 07:00:00'),\

(10000,'wuyanzu','北京',19,0,12345678910,'北京朝阳区','2017-10-01 07:00:00'),\

(10000,'zhangsan','北京',20,0,12345678910,'北京海淀区','2017-11-15 06:10:20');

3）查询表

select \* from test\_db.user;

Uniq 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。

### 3.5.3 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。Duplicate 数据模型可以满足这类需求。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。

1）建表

CREATE TABLE IF NOT EXISTS test\_db.example\_log

(

`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(`timestamp`) BUCKETS 10;

2）插入数据

insert into test\_db.example\_log values\

('2017-10-01 08:00:05',1,404,'not found page', 101, '2017-10-01 08:00:05'),\

('2017-10-01 08:00:05',1,404,'not found page', 101, '2017-10-01 08:00:05'),\

('2017-10-01 08:00:05',2,404,'not found page', 101, '2017-10-01 08:00:06'),\

('2017-10-01 08:00:06',2,404,'not found page', 101, '2017-10-01 08:00:07');

3）查看表

select \* from test\_db.example\_log;

### 3.5.4 数据模型的选择建议

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**。

（1）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。

（2）Uniq模型针对需要唯一主键约束的场景，可以保证主键唯一性约束。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）。

（3）Duplicate适合任意维度的Ad-hoc查询。虽然同样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（只读取相关列，而不需要读取所有Key列）

### 3.5.5 聚合模型的局限性

这里我们针对Aggregate模型（包括Uniq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，任何还未聚合的数据（比如说两个不同导入批次的数据），必须通过某种方式，以保证对外展示的一致性。我们举例说明。

假设表结构如下：

|  |  |  |  |
| --- | --- | --- | --- |
| ColumnName | Type | AggregationType | Comment |
| user\_id | LARGEINT |  | 用户id |
| date | DATE |  | 数据灌入日期 |
| 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 |

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。比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。

上面的例子，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 |  | 数据灌入日期 |
| 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。

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。

## 3.6 动态分区

动态分区是在Doris 0.12版本中引入的新功能。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。

### 3.6.1 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。

### 3.6.2 使用方式

动态分区的规则可以在建表时指定，或者在运行时进行修改。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",

...

)

### 3.6.3 动态分区规则参数

3.6.3.1 主要参数

动态分区的规则参数都以 dynamic\_partition. 为前缀：

|  |  |
| --- | --- |
| dynamic\_partition.enable | 是否开启动态分区特性，可指定true或false,默认为true |
| dynamic\_partition.time\_unit | 动态分区调度的单位，可指定HOUR、DAY、WEEK、MONTH。  HOUR，后缀格式为 yyyyMMddHH，分区列数据类型不能为 DATE。  DAY，后缀格式为 yyyyMMdd。  WEEK，后缀格式为yyyy\_ww。即当前日期属于这一年的第几周。  MONTH，后缀格式为 yyyyMM。 |
| 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 |
| 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。如果不填写默认值为Interger.Min\_VALUE 即-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.prefix | 动态创建的分区名前缀 |
| dynamic\_partition.buckets | 动态创建的分区所对应分桶数量 |
| 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。 |
| 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 |

3.6.3.2 创建历史分区的参数

* dynamic\_partition.create\_history\_partition

默认为 false。当置为 true 时，Doris 会自动创建所有分区，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。

hot\_partition\_num 是往前 n 天和未来所有分区

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=9999-12-31 23:59:59

p20210518：["2021-05-18", "2021-05-19") storage\_medium=HDD storage\_cooldown\_time=9999-12-31 23:59:59

p20210519：["2021-05-19", "2021-05-20") storage\_medium=SSD storage\_cooldown\_time=2021-05-21 00:00:00

p20210520：["2021-05-20", "2021-05-21") storage\_medium=SSD storage\_cooldown\_time=2021-05-22 00:00:00

p20210521：["2021-05-21", "2021-05-22") storage\_medium=SSD storage\_cooldown\_time=2021-05-23 00:00:00

p20210522：["2021-05-22", "2021-05-23") storage\_medium=SSD storage\_cooldown\_time=2021-05-24 00:00:00

p20210523：["2021-05-23", "2021-05-24") storage\_medium=SSD storage\_cooldown\_time=2021-05-25 00:00:00

* 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="[2020-06-01,2020-06-20],[2020-10-31,2020-11-15]"。

则系统会自动保留：

["2020-06-01","2020-06-20"],

["2020-10-31","2020-11-15"]

或者

time\_unit="HOUR", \

end=3, \

start=-3, \

reserved\_history\_periods="[2020-06-01 00:00:00,2020-06-01 03:00:00]".

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``。

3.6.3.3 创建历史分区规则

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：

（1）create\_history\_partition = true

① dynamic\_partition.history\_partition\_num 未设置，即 -1.

则expect\_create\_partition\_num = end - start;

② dynamic\_partition.history\_partition\_num 已设置

则expect\_create\_partition\_num = end - max(start, -histoty\_partition\_num);

（2）create\_history\_partition = false

不会创建历史分区，expect\_create\_partition\_num = end - 0;

（3）当 expect\_create\_partition\_num > max\_dynamic\_partition\_num（默认500）时，禁止创建过多分区。

3.6.3.4 创建历史分区举例

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

history\_partition\_num=5，其余属性与 1 中保持一直，则系统会自动创建以下分区：

p20210517

p20210518

p20210519

p20210520

p20210521

p20210522

p20210523

history\_partition\_num=-1 即不设置历史分区数量，其余属性与 1 中保持一直，则系统会自动创建以下分区：

p20210517

p20210518

p20210519

p20210520

p20210521

p20210522

p20210523

3.6.3.5 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。

### 3.6.4 示例

1）创建动态分区表

分区列time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 student\_dynamic\_partition1

(id int,

time date,

name varchar(50),

age int

)

duplicate key(id,time)

PARTITION BY RANGE(time)()

DISTRIBUTED BY HASH(id) buckets 10

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" = "10",

"replication\_num" = "1"

);

2）查看动态分区表调度情况

SHOW DYNAMIC PARTITION TABLES;

* 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

3）查看表的分区

SHOW PARTITIONS FROM student\_dynamic\_partition1;

4）插入测试数据,可以全部成功（修改成对应时间）

insert into student\_dynamic\_partition1 values(1,'2022-03-31 11:00:00','name1',18);

insert into student\_dynamic\_partition1 values(1,'2022-04-01 11:00:00','name1',18);

insert into student\_dynamic\_partition1 values(1,'2022-04-02 11:00:00','name1',18);

5）设置创建历史分区

ALTER TABLE student\_dynamic\_partition1 SET ("dynamic\_partition.create\_history\_partition" = "true");

查看分区情况

SHOW PARTITIONS FROM student\_dynamic\_partition1;

6）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。

（1）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。

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。

（2）动态分区转换为手动分区

ALTER TABLE tbl\_name SET ("dynamic\_partition.enable" = "false")

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。

## 3.7 Rollup

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。

### 3.7.1 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。

### 3.7.2 Aggregate 和 Uniq 模型中的 ROLLUP

因为 Uniq 只是 Aggregate 模型的一个特例，所以这里我们不加以区别。

1）以3.5.1.2中创建的example\_site\_visit2表为例。

（1）查看表的结构信息

desc example\_site\_visit2 all;

（2）比如需要查看某个用户的总消费，那么可以建立一个只有user\_id和cost的rollup

alter table example\_site\_visit2 add rollup rollup\_cost\_userid(user\_id,cost);

（3）查看表的结构信息

desc example\_site\_visit2 all;

（4）然后可以通过explain查看执行计划，是否使用到了rollup

explain SELECT user\_id, sum(cost) FROM example\_site\_visit2 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。

（5）通过命令查看完成状态

SHOW ALTER TABLE ROLLUP;

2）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

（1）创建 ROLLUP

alter table example\_site\_visit2 add rollup rollup\_city\_age\_cost\_maxd\_mind(city,age,cost,max\_dwell\_time,min\_dwell\_time);

（2）查看rollup使用

explain SELECT city, age, sum(cost), max(max\_dwell\_time), min(min\_dwell\_time) FROM example\_site\_visit2 GROUP BY city, age;

explain SELECT city, sum(cost), max(max\_dwell\_time), min(min\_dwell\_time) FROM example\_site\_visit2 GROUP BY city;

explain SELECT city, age, sum(cost), min(min\_dwell\_time) FROM example\_site\_visit2 GROUP BY city, age;

（3）通过命令查看完成状态

SHOW ALTER TABLE ROLLUP;

### 3.7.3 Duplicate 模型中的 ROLLUP

因为 Duplicate 模型没有聚合的语意。所以该模型中的 ROLLUP，已经失去了“上卷”这一层含义。而仅仅是作为调整列顺序，以命中前缀索引的作用。下面详细介绍前缀索引，以及如何使用ROLLUP改变前缀索引，以获得更好的查询效率。

3.7.3.1 前缀索引

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。

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。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、Uniq 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、UNIQ KEY 和 DUPLICATE KEY 中指定的列进行排序存储的。而前缀索引，即**在排序的基础上，实现的一种根据给定前缀列，快速查询数据的索引方式**。

我们将一行数据的前 36 个字节 作为这行数据的前缀索引。当遇到 VARCHAR 类型时，前缀索引会直接截断。举例说明：

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 |

3）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。比如在第一个例子中，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。

3.7.3.2 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：

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。

### 3.7.4 ROLLUP 的几点说明

* 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。这也是为什么在源代码中，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。

## 3.8 物化视图

物化视图就是包含了查询结果的数据库对象，可能是对远程数据的本地copy，也可能是一个表或多表join后结果的行或列的子集，也可能是聚合后的结果。说白了，就是预先存储查询结果的一种数据库对象。

在Doris中的物化视图，就是查询结果预先存储起来的特殊的表。

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。

### 3.8.1 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，比如：时间很久的聚合操作等。
* 查询需要匹配不同前缀索引。

### 3.8.2 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。

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。

### 3.8.3 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。

### 3.8.4 物化视图原理

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。但是Doris目前创建物化视图只能在单表操作，不支持join。

3.8.4.1 创建物化视图

首先要根据查询语句的特点来决定创建一个什么样的物化视图。并不是说物化视图定义和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。

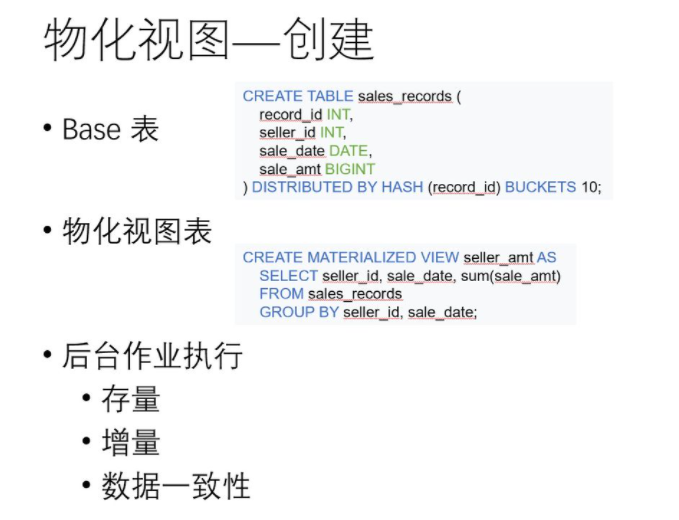
所以用户需要结合自己的查询语句，以及数据维度信息去抽象出一些物化视图的定义。

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

这里以一个销售记录表为例：

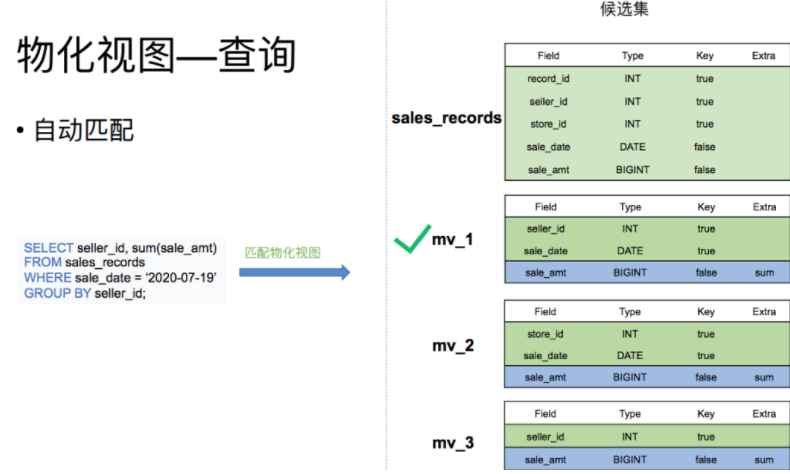


比如我们有一张销售记录明细表，存储了每个交易的时间，销售员，销售门店，和金额。

提交完创建物化视图的任务后，Doris就会异步在后台生成物化视图的数据，构建物化视图。

在构建期间，用户依然可以正常的查询和导入新的数据。创建任务会自动处理当前的存量数据和所有新到达的增量数据，从而保持和base表的数据一致性。用户不需关心一致性问题。

3.8.4.2 查询

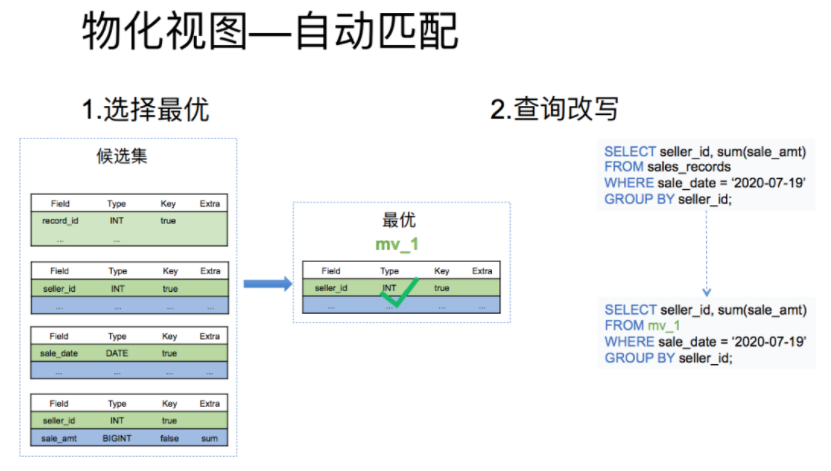


物化视图创建完成后，用户的查询会根据规则自动匹配到最优的物化视图。

比如我们有一张销售记录明细表，并且在这个明细表上创建了三张物化视图。一个存储了不同时间不同销售员的售卖量，一个存储了不同时间不同门店的销售量，以及每个销售员的总销售量。

当查询7月19日，各个销售员都买了多少钱的话。就可以匹配mv\_1物化视图。直接对mv\_1的数据进行查询。

3.8.4.3 查询自动匹配



物化视图的自动匹配分为下面两个步骤：

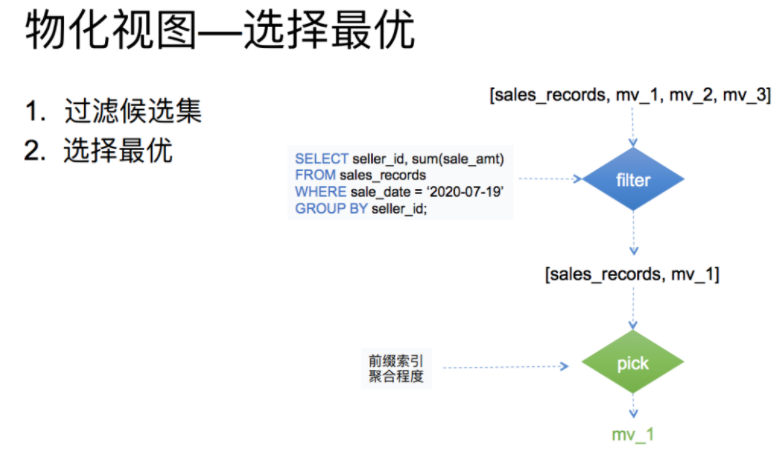
（1）根据查询条件删选出一个最优的物化视图：这一步的输入是所有候选物化视图表的元数据，根据查询的条件从候选集中输出最优的一个物化视图

（2）根据选出的物化视图对查询进行改写：这一步是结合上一步选择出的最优物化视图，进行查询的改写，最终达到直接查询物化视图的目的。



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。详细见实例2.

3.8.4.4 最优路径选择

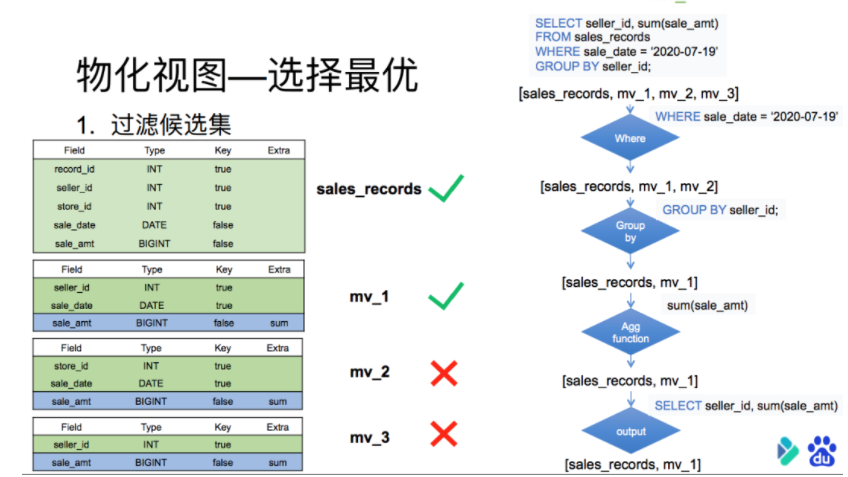


这里分为两个步骤：

（1）对候选集合进行一个过滤。只要是查询的结果能从物化视图数据计算（取部分行，部分列，或部分行列的聚合）出都可以留在候选集中，过滤完成后候选集合大小>=1。

（2）从候选集合中根据聚合程度，索引等条件选出一个最优的也就是查询花费最少物化视图。

这里再举一个相对复杂的例子，来体现这个过程：



候选集过滤目前分为4层，每一层过滤后去除不满足条件的物化视图。

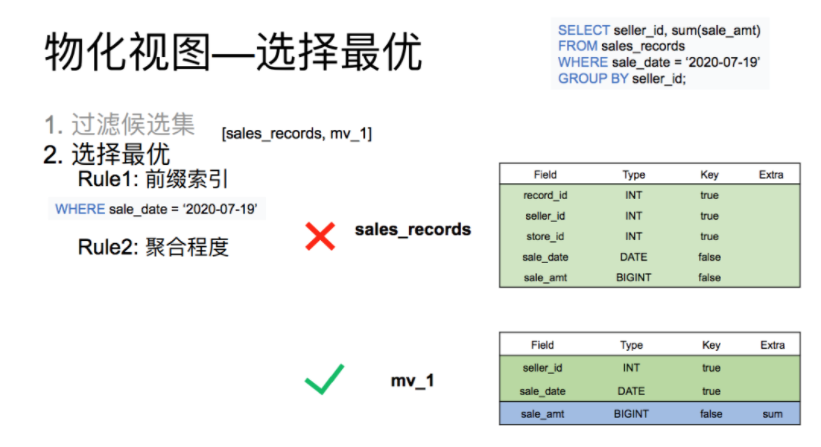
比如查询7月19日，各个销售员都买了多少钱，候选集中包括所有的物化视图以及base表共4个：

第一层过滤先判断查询where中的谓词涉及到的数据是否能从物化视图中得到。也就是销售时间列是否在表中存在。由于第三个物化视图中根本不存在销售时间列。所以在这一层过滤中，mv\_3就被淘汰了。

第二层是过滤查询的分组列是否为候选集的分组列的子集。也就是销售员id是否为表中分组列的子集。由于第二个物化视图中的分组列并不涉及销售员id。所以在这一层过滤中，mv\_2也被淘汰了。

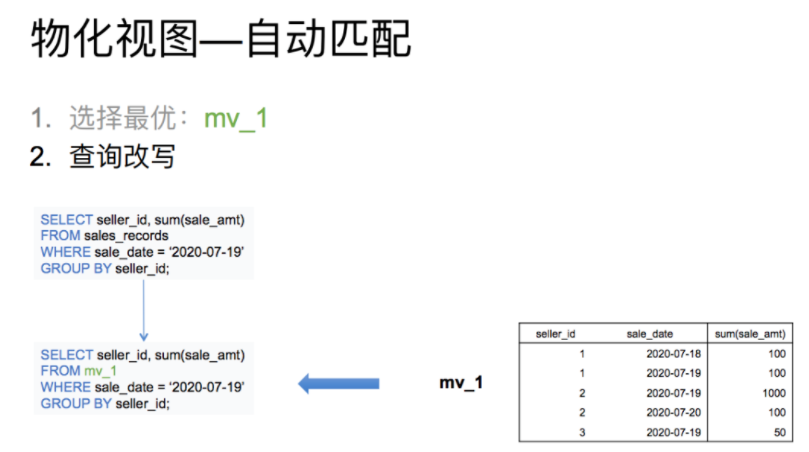
第三层过滤是看查询的聚合列是否为候选集中聚合列的子集。也就是对销售额求和是否能从候选集的表中聚合得出。这里base 表和物化视图表均满足标准。

最后一层是过滤看查询需要的列是否存在于候选集合的列中。由于候选集合中的表均满足标准，所以最终候选集合中的表为 销售明细表，以及mv\_1，这两张。



候选集过滤完后输出一个集合，这个集合中的所有表都能满足查询的需求。但每张表的查询效率都不同。这时候就需要再这个集合根据前缀索引是否能匹配到，以及聚合程度的高低来选出一个最优的物化视图。

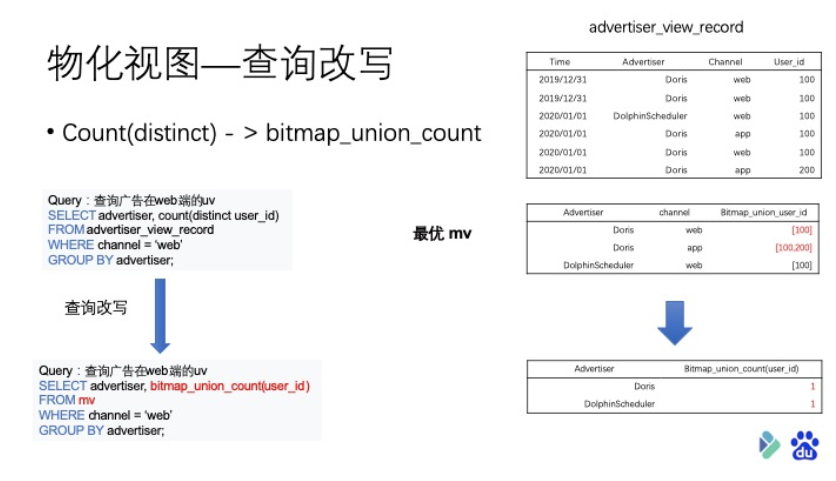
从表结构中可以看出，base表的销售日期列是一个非排序列，而物化视图表的日期是一个排序列，同时聚合程度上mv\_1表明显比base表高。所以最后选择出mv\_1作为该查询的最优匹配。



最后再根据选择出的最优解，改写查询。

刚才的查询选中mv\_1后，将查询改写为从mv\_1中读取数据，过滤出日志为7月19日的mv\_1中的数据然后返回即可。

3.8.4.5 查询改写



有些情况下的查询改写还会涉及到查询中的聚合函数的改写。

比如业务方经常会用到count distinct对PV UV进行计算。

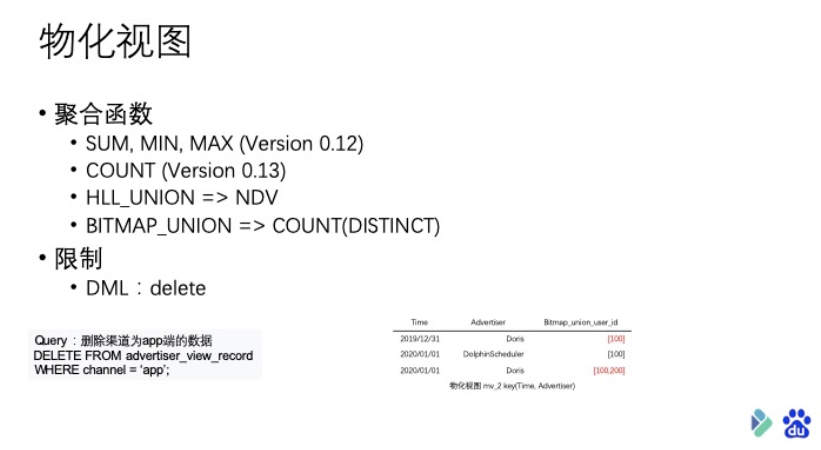
例如；

广告点击明细记录表中存放哪个用户点击了什么广告，从什么渠道点击的，以及点击的时间。并且在这个base表基础上构建了一个物化视图表，存储了不同广告不同渠道的用户bitmap值。

由于bitmap union这种聚合方式本身会对相同的用户user id进行一个去重聚合。当用户查询广告在web端的uv的时候，就可以匹配到这个物化视图。匹配到这个物化视图表后就需要对查询进行改写，将之前的对用户id求count(distinct) 改为对物化视图中bitmap union列求count。

所以最后查询取物化视图的第一和第三行求bitmap聚合中有几个值。

3.8.4.6 使用及限制



（1）目前支持的聚合函数包括，常用的sum，min，max count，以及计算pv ，uv， 留存率，等常用的去重算法 hll\_union，和用于精确去重计算count（distinct）的算法 bitmap\_union。

（2）物化视图的聚合函数的参数不支持表达式仅支持单列，比如： sum(a+b)不支持。

（3）使用物化视图功能后，由于物化视图实际上是损失了部分维度数据的。所以对表的 DML 类型操作会有一些限制：

如果表的物化视图key中不包含删除语句中的条件列，则删除语句不能执行。

比如想要删除渠道为app端的数据，由于存在一个物化视图并不包含渠道这个字段，则这个删除不能执行，因为删除在物化视图中无法被执行。这时候你只能把物化视图先删除，然后删除完数据后，重新构建一个新的物化视图。

（4）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。

（5）相同列，不同聚合函数，不能同时出现在一张物化视图中，比如：select sum(a), min(a) from table 不支持。

（6）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

### 3.8.5 案例演示

3.8.5.1 案例一

1）创建一个Base表

create table sales\_records(

record\_id int,

seller\_id int,

store\_id int,

sale\_date date,

sale\_amt bigint

)

distributed by hash(record\_id)

properties("replication\_num" = "1");

插入数据

insert into sales\_records values(1,2,3,'2020-02-02',10);

2）基于这个Base表的数据提交一个创建物化视图的任务

create materialized view store\_amt as

select store\_id, sum(sale\_amt)

from sales\_records

group by store\_id;

3）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。

SHOW ALTER TABLE MATERIALIZED VIEW FROM test\_db; (Version 0.13)

查看Base表的所有物化视图

desc sales\_records all;

4）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;

5）删除物化视图语法

DROP MATERIALIZED VIEW 物化视图名 on Base表名;

3.8.5.2 案例二：计算广告的pv、uv

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。

1）创建base表

create table advertiser\_view\_record(

time date,

advertiser varchar(10),

channel varchar(10),

user\_id int

)

distributed by hash(time)

properties("replication\_num" = "1");

插入数据

insert into advertiser\_view\_record values('2020-02-02','a','app',123);

2）创建物化视图

create materialized view advertiser\_uv as

select advertiser, channel, bitmap\_union(to\_bitmap(user\_id))

from advertiser\_view\_record

group by advertiser, channel;

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。

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。

3）查询自动匹配

SELECT advertiser, channel, count(distinct user\_id)

FROM advertiser\_view\_record

GROUP BY advertiser, channel;

会自动转换成。

SELECT advertiser, channel, bitmap\_union\_count(to\_bitmap(user\_id))

FROM advertiser\_uv

GROUP BY advertiser, channel;

4）检验是否匹配到物化视图

explain SELECT advertiser, channel, count(distinct user\_id) FROM advertiser\_view\_record GROUP BY advertiser, channel;

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。

3.8.5.3 案例三

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，比如 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）查询

explain select record\_id,seller\_id,store\_id from sales\_records where store\_id=3;

2）创建物化视图

create materialized view mv\_1 as

select

store\_id,

record\_id,

seller\_id,

sale\_date,

sale\_amt

from sales\_records;

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store\_id列。

3）查看表结构

desc sales\_records all;

4）查询匹配

explain select record\_id,seller\_id,store\_id from sales\_records where store\_id=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store\_id 是存在前缀索引的，查询效率也会提升。

## 3.9 修改表

使用 ALTER TABLE 命令可以对表进行修改，包括partition 、rollup、schema change、rename 和index五种。语法：

ALTER TABLE [database.]table

alter\_clause1[, alter\_clause2, ...];

alter\_clause 分为 partition 、rollup、schema change、rename 和index五种。

### 3.9.1 rename

1）将名为 table1 的表修改为 table2

ALTER TABLE table1 RENAME table2;

2）将表 example\_table 中名为 rollup1 的 rollup index 修改为 rollup2

ALTER TABLE example\_table RENAME ROLLUP rollup1 rollup2;

3）将表 example\_table 中名为 p1 的 partition 修改为 p2

ALTER TABLE example\_table RENAME PARTITION p1 p2;

### 3.9.2 partition

1）增加分区, 使用默认分桶方式

现有分区 [MIN, 2013-01-01)，增加分区 [2013-01-01, 2014-01-01)，

ALTER TABLE example\_db.my\_table

ADD PARTITION p1 VALUES LESS THAN ("2014-01-01");

2）增加分区，使用新的分桶数

ALTER TABLE example\_db.my\_table

ADD PARTITION p1 VALUES LESS THAN ("2015-01-01")

DISTRIBUTED BY HASH(k1) BUCKETS 20;

3）增加分区，使用新的副本数

ALTER TABLE example\_db.my\_table

ADD PARTITION p1 VALUES LESS THAN ("2015-01-01")

("replication\_num"="1");

4）修改分区副本数

ALTER TABLE example\_db.my\_table

MODIFY PARTITION p1 SET("replication\_num"="1");

5）批量修改指定分区

ALTER TABLE example\_db.my\_table

MODIFY PARTITION (p1, p2, p4) SET("in\_memory"="true");

6）批量修改所有分区

ALTER TABLE example\_db.my\_table

MODIFY PARTITION (\*) SET("storage\_medium"="HDD");

7）删除分区

ALTER TABLE example\_db.my\_table

DROP PARTITION p1;

8）增加一个指定上下界的分区

ALTER TABLE example\_db.my\_table

ADD PARTITION p1 VALUES [("2014-01-01"), ("2014-02-01"));

### 3.9.3 rollup

1）创建 index: example\_rollup\_index，基于 base index（k1,k2,k3,v1,v2）。列式存储。

ALTER TABLE example\_db.my\_table

ADD ROLLUP example\_rollup\_index(k1, k3, v1, v2);

2）创建 index: example\_rollup\_index2，基于 example\_rollup\_index（k1,k3,v1,v2）

ALTER TABLE example\_db.my\_table

ADD ROLLUP example\_rollup\_index2 (k1, v1)

FROM example\_rollup\_index;

3）创建 index: example\_rollup\_index3, 基于 base index (k1,k2,k3,v1), 自定义 rollup 超时时间一小时。

ALTER TABLE example\_db.my\_table

ADD ROLLUP example\_rollup\_index(k1, k3, v1)

PROPERTIES("timeout" = "3600");

4）删除 index: example\_rollup\_index2

ALTER TABLE example\_db.my\_table

DROP ROLLUP example\_rollup\_index2;

### 3.9.4 表结构变更

使用 ALTER TABLE 命令可以修改表的 Schema，包括如下修改：

* 增加列
* 删除列
* 修改列类型
* 改变列顺序

以增加列为例：

1）我们新增一列 uv，类型为 BIGINT，聚合类型为 SUM，默认值为 0:

ALTER TABLE table1 ADD COLUMN uv BIGINT SUM DEFAULT '0' after pv;

2）提交成功后，可以通过以下命令查看作业进度:

SHOW ALTER TABLE COLUMN;

当作业状态为 FINISHED，则表示作业完成。新的 Schema 已生效。

3）查看新的Schema

DESC table1;

4）可以使用以下命令取消当前正在执行的作业:

CANCEL ALTER TABLE ROLLUP FROM table1;

5）更多可以参阅： HELP ALTER TABLE

<https://doris.apache.org/zh-CN/sql-reference/sql-statements/Data%20Definition/ALTER%20TABLE.html>

## 3.10 删除数据（Delete）

Doris目前可以通过两种方式删除数据：DELETE FROM 语句和ALTER TABLE DROP PARTITION语句。

### 3.10.1 DELETE FROM Statement（条件删除）

delete from 语句类似标准 delete 语法，具体使用可以查看 help delete; 帮助。

语法：

DELETE FROM table\_name [PARTITION partition\_name]

WHERE

column\_name1 op { value | value\_list } [ AND column\_name2 op { value | value\_list } ...];

如：

delete from student\_kafka where id=1;

注意事项。

（1）该语句只能针对 Partition 级别进行删除。如果一个表有多个 partition 含有需要删除的数据，则需要执行多次针对不同 Partition 的 delete 语句。而如果是没有使用 Partition 的表，partition 的名称即表名。

（2）where 后面的条件谓词只能针对 Key 列，并且谓词之间，只能通过 AND 连接。如果想实现 OR 的语义，需要执行多条 delete。

（3）delete 是一个同步命令，命令返回即表示执行成功。

（4）从代码实现角度，delete 是一种特殊的导入操作。该命令所导入的内容，也是一个新的数据版本，只是该版本中只包含命令中指定的删除条件。在实际执行查询时，会根据这些条件进行查询时过滤。所以，不建议大量频繁使用 delete 命令，因为这可能导致查询效率降低。

（5）数据的真正删除是在 BE 进行数据 Compaction 时进行的。所以执行完 delete 命令后，并不会立即释放磁盘空间。

（6）delete 命令一个较强的限制条件是，在执行该命令时，对应的表，不能有正在进行的导入任务（包括 PENDING、ETL、LOADING）。而如果有 QUORUM\_FINISHED 状态的导入任务，则可能可以执行。

（7）delete 也有一个隐含的类似 QUORUM\_FINISHED 的状态。即如果 delete 只在多数副本上完成了，也会返回用户成功。但是会在后台生成一个异步的 delete job（Async Delete Job），来继续完成对剩余副本的删除操作。如果此时通过 show delete 命令，可以看到这种任务在 state 一栏会显示 QUORUM\_FINISHED。

### 3.10.2 DROP PARTITION Statement（删除分区）

该命令可以直接删除指定的分区。因为Partition是逻辑上最小的数据管理单元，所以使用 DROP PARTITION 命令可以很轻量的完成数据删除工作。并且该命令不受load以及任何其他操作的限制，同时不会影响查询效率。是比较推荐的一种数据删除方式。

该命令是同步命令，执行成功即生效。而后台数据真正删除的时间可能会延迟10分钟左右。

# 第4章 数据导入和导出

## 4.1 数据导入

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。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。

**1）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。

**2）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。

**3）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。

**4）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。

**5）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。

**6）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。

Broker load是一个异步的导入方式，支持的数据源取决于Broker进程支持的数据源。

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。

### 4.1.1 Broker Load

4.1.1.1 适用场景

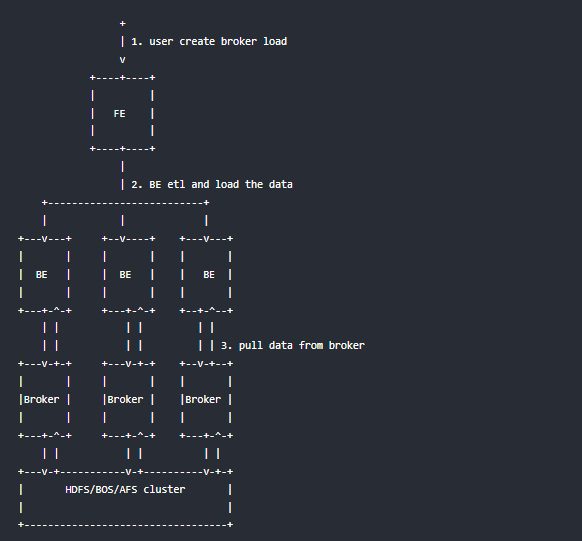
源数据在Broker可以访问的存储系统中，如HDFS。

数据量在几十到百GB级别。

[4.1.1.2](https://doris.apache.org/zh-CN/administrator-guide/load-data/broker-load-manual.html#%E5%9F%BA%E6%9C%AC%E5%8E%9F%E7%90%86) 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，由FE最终决定导入是否成功。



4.1.1.3 基本语法

LOAD LABEL db\_name.label\_name

(data\_desc, ...)

WITH BROKER broker\_name broker\_properties

[PROPERTIES (key1=value1, ... )]

\* data\_desc:

DATA INFILE ('file\_path', ...)

[NEGATIVE]

INTO TABLE tbl\_name

[PARTITION (p1, p2)]

[COLUMNS TERMINATED BY separator ]

[(col1, ...)]

[PRECEDING FILTER predicate]

[SET (k1=f1(xx), k2=f2(xx))]

[WHERE predicate]

\* broker\_properties:

(key1=value1, ...)

创建导入的详细语法执行 HELP BROKER LOAD 查看语法帮助。这里主要介绍 Broker load 的创建导入语法中参数意义和注意事项。

[**1）**](https://doris.apache.org/zh-CN/administrator-guide/load-data/broker-load-manual.html#label)**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 语义**

当 Label 对应的导入作业状态为 CANCELLED 时，可以再次使用该 Label 提交导入作业。

[**2）**](https://doris.apache.org/zh-CN/administrator-guide/load-data/broker-load-manual.html#%E6%95%B0%E6%8D%AE%E6%8F%8F%E8%BF%B0%E7%B1%BB%E5%8F%82%E6%95%B0)**数据描述类参数**

数据描述类参数主要指的是 Broker load 创建导入语句中的属于 data\_desc 部分的参数。每组 data\_desc 主要表述了本次导入涉及到的数据源地址，ETL 函数，目标表及分区等信息。

下面主要对数据描述类的部分参数详细解释：

* 多表导入

Broker load 支持一次导入任务涉及多张表，每个 Broker load 导入任务可在多个 data\_desc 声明多张表来实现多表导入。每个单独的 data\_desc 还可以指定属于该表的数据源地址。Broker load 保证了单次导入的多张表之间原子性成功或失败。

* negative

data\_desc中还可以设置数据取反导入。这个功能主要用于，当数据表中聚合列的类型都为 SUM 类型时。如果希望撤销某一批导入的数据。则可以通过 negative 参数导入同一批数据。Doris 会自动为这一批数据在聚合列上数据取反，以达到消除同一批数据的功能。

* partition

在 data\_desc 中可以指定待导入表的 partition 信息，如果待导入数据不属于指定的 partition 则不会被导入。同时，不在指定 Partition 的数据会被认为是错误数据。

* set column mapping

在 data\_desc 中的 SET 语句负责设置列函数变换，这里的列函数变换支持所有查询的等值表达式变换。如果原始数据的列和表中的列不一一对应，就需要用到这个属性。

* preceding filter predicate

用于过滤原始数据。原始数据是未经列映射、转换的数据。用户可以在对转换前的数据前进行一次过滤，选取期望的数据，再进行转换。

* where predicate

在 data\_desc 中的 WHERE 语句中负责过滤已经完成 transform 的数据，被 filter 的数据不会进入容忍率的统计中。如果多个 data\_desc 中声明了同一张表的多个条件的话，则会 merge 同一张表的多个条件，merge 策略是 AND 。

**3）导入作业参数**

导入作业参数主要指的是 Broker load 创建导入语句中的属于 opt\_properties部分的参数。导入作业参数是作用于整个导入作业的。

下面主要对导入作业参数的部分参数详细解释：

* timeout

导入作业的超时时间(以秒为单位)，用户可以在 opt\_properties 中自行设置每个导入的超时时间。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。Broker load 的默认导入超时时间为4小时。

通常情况下，用户不需要手动设置导入任务的超时时间。当在默认超时时间内无法完成导入时，可以手动设置任务的超时时间。

推荐超时时间：

总文件大小（MB） / 用户 Doris 集群最慢导入速度(MB/s) > timeout > （（总文件大小(MB) \* 待导入的表及相关 Roll up 表的个数） / (10 \* 导入并发数） ）

导入并发数见文档最后的导入系统配置说明，公式中的 10 为目前的导入限速 10MB/s。

例如一个 1G 的待导入数据，待导入表包含3个 Rollup 表，当前的导入并发数为 3。则 timeout 的 最小值为 (1 \* 1024 \* 3 ) / (10 \* 3) = 102 秒

由于每个 Doris 集群的机器环境不同且集群并发的查询任务也不同，所以用户 Doris 集群的最慢导入速度需要用户自己根据历史的导入任务速度进行推测。

* 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：

max\_filter\_ratio = (dpp.abnorm.ALL / (dpp.abnorm.ALL + dpp.norm.ALL ) )

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

* exec\_mem\_limit

导入内存限制。默认是 2GB。单位为字节。

* strict\_mode

Broker load 导入可以开启 strict mode 模式。开启方式为 properties ("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 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。

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。

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

4.1.1.4 导入示例

1）Doris中创建表

create table student\_result

(

id int ,

name varchar(50),

age int ,

score decimal(10,4)

)

DUPLICATE KEY(id)

DISTRIBUTED BY HASH(id) BUCKETS 10;

2）文件上传HDFS

启动HDFS相关服务

hadoop fs -put student.csv /

3）导入数据

csv文件导入

LOAD LABEL test\_db.student\_result

(

DATA INFILE("hdfs://my\_cluster/student.csv")

INTO TABLE `student\_result`

COLUMNS TERMINATED BY ","

FORMAT AS "csv"

(id, name, age, score)

)

WITH BROKER broker\_name

(

#开启了HA的写法，其他HDFS参数可以在这里指定

"dfs.nameservices" = "my\_cluster",

"dfs.ha.namenodes.my\_cluster" = "nn1,nn2,nn3",

"dfs.namenode.rpc-address.my\_cluster.nn1" = "hadoop1:8020",

"dfs.namenode.rpc-address.my\_cluster.nn2" = "hadoop2:8020",

"dfs.namenode.rpc-address.my\_cluster.nn3" = "hadoop3:8020",

"dfs.client.failover.proxy.provider" = "org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider"

)

PROPERTIES

(

"timeout" = "3600"

);

通用文件格式写法

LOAD LABEL test\_db.student\_result

(

DATA INFILE("hdfs://hadoop1:8020/student.csv")

INTO TABLE `student\_result`

COLUMNS TERMINATED BY ","

(c1, c2, c3, c4)

set(

id=c1,

name=c2,

age=c3,

score=c4

)

)

WITH BROKER broker\_name

(

#开启了HA的写法，其他HDFS参数可以在这里指定

"dfs.nameservices" = "my\_cluster",

"dfs.ha.namenodes.my\_cluster" = "nn1,nn2,nn3",

"dfs.namenode.rpc-address.my\_cluster.nn1" = "hadoop1:8020",

"dfs.namenode.rpc-address.my\_cluster.nn2" = "hadoop2:8020",

"dfs.namenode.rpc-address.my\_cluster.nn3" = "hadoop3:8020",

"dfs.client.failover.proxy.provider" = "org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider"

)

PROPERTIES

(

"timeout" = "3600"

);

[4.1.1.5](https://doris.apache.org/zh-CN/administrator-guide/load-data/broker-load-manual.html#%E6%9F%A5%E7%9C%8B%E5%AF%BC%E5%85%A5) 查看导入

Broker load 导入方式由于是异步的，所以用户必须将创建导入的 Label 记录，并且在**查看导入命令中使用 Label 来查看导入结果**。查看导入命令在所有导入方式中是通用的，具体语法可执行 HELP SHOW LOAD 查看。

mysql> show load order by createtime desc limit 1\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

JobId: 76391

Label: label1

State: FINISHED

Progress: ETL:N/A; LOAD:100%

Type: BROKER

EtlInfo: unselected.rows=4; dpp.abnorm.ALL=15; dpp.norm.ALL=28133376

TaskInfo: cluster:N/A; timeout(s):10800; max\_filter\_ratio:5.0E-5

ErrorMsg: N/A

CreateTime: 2019-07-27 11:46:42

EtlStartTime: 2019-07-27 11:46:44

EtlFinishTime: 2019-07-27 11:46:44

LoadStartTime: 2019-07-27 11:46:44

LoadFinishTime: 2019-07-27 11:50:16

URL: http://192.168.1.1:8040/api/\_load\_error\_log?file=\_\_shard\_4/error\_log\_insert\_stmt\_4bb00753932c491a-a6da6e2725415317\_4bb00753932c491a\_a6da6e2725415317

JobDetails: {"Unfinished backends":{"9c3441027ff948a0-8287923329a2b6a7":[10002]},"ScannedRows":2390016,"TaskNumber":1,"All backends":{"9c3441027ff948a0-8287923329a2b6a7":[10002]},"FileNumber":1,"FileSize":1073741824}

下面主要介绍了查看导入命令返回结果集中参数意义：

* JobId

导入任务的唯一ID，每个导入任务的 JobId 都不同，由系统自动生成。与 Label 不同的是，JobId永远不会相同，而 Label 则可以在导入任务失败后被复用。

* Label

导入任务的标识。

* State

导入任务当前所处的阶段。在 Broker load 导入过程中主要会出现 PENDING 和 LOADING 这两个导入中的状态。如果 Broker load 处于 PENDING 状态，则说明当前导入任务正在等待被执行；LOADING 状态则表示正在执行中。

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。其中 CANCELLED 为导入失败，FINISHED 为导入成功。

* 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。目前 Broker load 由于只有 LOADING 阶段，所以 ETL 则会永远显示为 N/A

LOAD 的进度范围为：0~100%。

LOAD 进度 = 当前完成导入的表个数 / 本次导入任务设计的总表个数 \* 100%

**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，LOAD 的进度才会改为 100%。

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。

* Type

导入任务的类型。Broker load 的 type 取值只有 BROKER。

* EtlInfo

主要显示了导入的数据量指标 unselected.rows , dpp.norm.ALL 和 dpp.abnorm.ALL。用户可以根据第一个数值判断 where 条件过滤了多少行，后两个指标验证当前导入任务的错误率是否超过 max\_filter\_ratio。

三个指标之和就是原始数据量的总行数。

* TaskInfo

主要显示了当前导入任务参数，也就是创建 Broker load 导入任务时用户指定的导入任务参数，包括：cluster，timeout 和max\_filter\_ratio。

* ErrorMsg

在导入任务状态为CANCELLED，会显示失败的原因，显示分两部分：type 和 msg，如果导入任务成功则显示 N/A。

type的取值意义：

USER\_CANCEL: 用户取消的任务

ETL\_RUN\_FAIL：在ETL阶段失败的导入任务

ETL\_QUALITY\_UNSATISFIED：数据质量不合格，也就是错误数据率超过了 max\_filter\_ratio

LOAD\_RUN\_FAIL：在LOADING阶段失败的导入任务

TIMEOUT：导入任务没在超时时间内完成

UNKNOWN：未知的导入错误

CreateTime/EtlStartTime/EtlFinishTime/LoadStartTime/LoadFinishTime

这几个值分别代表导入创建的时间，ETL阶段开始的时间，ETL阶段完成的时间，Loading阶段开始的时间和整个导入任务完成的时间。

Broker load 导入由于没有 ETL 阶段，所以其 EtlStartTime, EtlFinishTime, LoadStartTime 被设置为同一个值。

导入任务长时间停留在 CreateTime，而 LoadStartTime 为 N/A 则说明目前导入任务堆积严重。用户可减少导入提交的频率。

LoadFinishTime - CreateTime = 整个导入任务所消耗时间

LoadFinishTime - LoadStartTime = 整个 Broker load 导入任务执行时间 = 整个导入任务所消耗时间 - 导入任务等待的时间

* URL

导入任务的错误数据样例，访问 URL 地址既可获取本次导入的错误数据样例。当本次导入不存在错误数据时，URL 字段则为 N/A。

* JobDetails

显示一些作业的详细运行状态。包括导入文件的个数、总大小（字节）、子任务个数、已处理的原始行数，运行子任务的 BE 节点 Id，未完成的 BE 节点 Id。

{"Unfinished backends":{"9c3441027ff948a0-8287923329a2b6a7":[10002]},"ScannedRows":2390016,"TaskNumber":1,"All backends":{"9c3441027ff948a0-8287923329a2b6a7":[10002]},"FileNumber":1,"FileSize":1073741824}

其中已处理的原始行数，每 5 秒更新一次。该行数仅用于展示当前的进度，不代表最终实际的处理行数。实际处理行数以 EtlInfo 中显示的为准。

[4.1.1.6](https://doris.apache.org/zh-CN/administrator-guide/load-data/broker-load-manual.html#%E5%8F%96%E6%B6%88%E5%AF%BC%E5%85%A5) 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 HELP CANCEL LOAD查看。

CANCEL LOAD

[FROM db\_name]

WHERE LABEL=”load\_label”;

### 4.1.2 Stream Load

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。

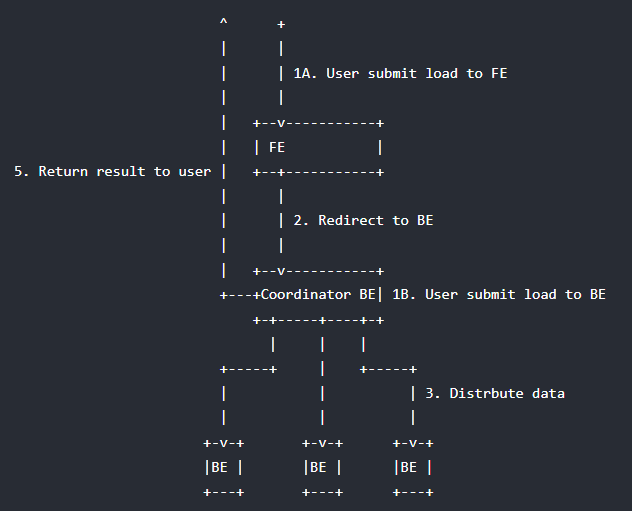
4.1.2.1 适用场景

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。

目前Stream Load支持两个数据格式：CSV（文本）和JSON。

4.1.2.2 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。

4.1.2.3 基本语法

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:http\_port/api/{db}/{table}/\_stream\_load

Header中支持属性见下面的 ‘导入任务参数’说明。

格式为： -H "key1:value1"

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。

**1）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。

**2）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

* 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

* 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。

* partition

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,dst\_c3）

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列对应目标表dst\_c3列，则写法如下：

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列对应目标表dst\_c1列，则写法如下：

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目标表的两列，则写法如下：

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
* two\_phase\_commit

Stream load 导入可以开启两阶段事务提交模式。开启方式为在 HEADER 中声明 two\_phase\_commit=true 。默认的两阶段批量事务提交为关闭。 两阶段批量事务提交模式的意思是：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。

用户可以调用如下接口对stream load事务触发commit操作：

curl -X PUT --location-trusted -u user:passwd -H "txn\_id:txnId" -H "txn\_operation:commit" http://fe\_host:http\_port/api/{db}/\_stream\_load\_2pc

或

curl -X PUT --location-trusted -u user:passwd -H "txn\_id:txnId" -H "txn\_operation:commit" http://be\_host:webserver\_port/api/{db}/\_stream\_load\_2pc

用户可以调用如下接口对stream load事务触发abort操作：

curl -X PUT --location-trusted -u user:passwd -H "txn\_id:txnId" -H "txn\_operation:abort" http://fe\_host:http\_port/api/{db}/\_stream\_load\_2pc

或

curl -X PUT --location-trusted -u user:passwd -H "txn\_id:txnId" -H "txn\_operation:abort" http://be\_host:webserver\_port/api/{db}/\_stream\_load\_2pc

4.1.2.4 导入示例

curl --location-trusted -u root -H "label:123" -H"column\_separator:," -T student.csv -X PUT http://hadoop1:8030/api/test\_db/student\_result/\_stream\_load

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。

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。

4.1.2.5 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。

Stream Load是一个同步的导入方式，用户通过发送HTTP协议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结果。用户可以直接通过返回判断导入是否成功。

### 4.1.3 Routine Load

例行导入（Routine Load）功能为用户提供了一种自动从指定数据源进行数据导入的功能。

4.1.3.1 适用场景

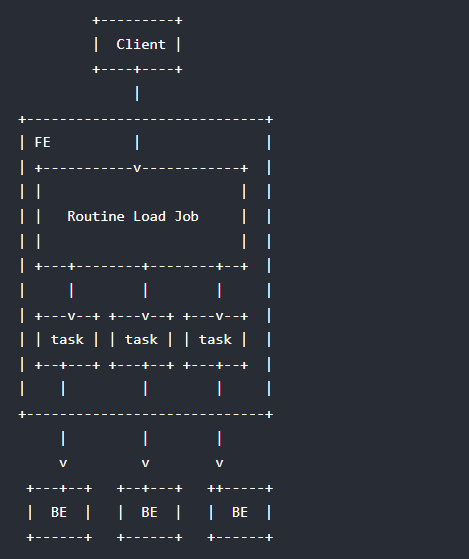
当前仅支持从Kafka系统进行例行导入，使用限制：

（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。

4.1.3.2 基本原理



如上图，Client 向 FE 提交一个例行导入作业。

（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）整个例行导入作业通过不断的产生新的 Task，来完成数据不间断的导入。

4.1.3.3 基本语法

CREATE ROUTINE LOAD [db.]job\_name ON tbl\_name

[merge\_type]

[load\_properties]

[job\_properties]

FROM data\_source

[data\_source\_properties]

执行 HELP ROUTINE LOAD可以查看语法帮助，下面是参数说明

**1）[db.]job\_name**

导入作业的名称，在同一个 database 内，相同名称只能有一个 job 在运行。

**2）tbl\_name**

指定需要导入的表的名称。

**3）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 on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, 语法为[WITH MERGE|APPEND|DELETE]

**4）load\_properties**

用于描述导入数据。语法： [column\_separator], [columns\_mapping], [where\_predicates], [delete\_on\_predicates], [source\_sequence], [partitions], [preceding\_predicates]

（1）column\_separator:

指定列分隔符，如： COLUMNS TERMINATED BY ","

这个只在文本数据导入的时候需要指定，JSON格式的数据导入不需要指定这个参数。

默认为：\t

（2）columns\_mapping:

指定源数据中列的映射关系，以及定义衍生列的生成方式。

* 映射列：

按顺序指定，源数据中各个列，对应目的表中的哪些列。对于希望跳过的列，可以指定一个不存在的列名。假设目的表有三列 k1, k2, v1。源数据有4列，其中第1、2、4列分别对应 k2, k1, v1。则书写如下：

COLUMNS (k2, k1, xxx, v1)

其中 xxx 为不存在的一列，用于跳过源数据中的第三列。

* 衍生列：

以 col\_name = expr 的形式表示的列，我们称为衍生列。即支持通过 expr 计算得出目的表中对应列的值。 衍生列通常排列在映射列之后，虽然这不是强制的规定，但是 Doris 总是先解析映射列，再解析衍生列。 接上一个示例，假设目的表还有第4列 v2，v2 由 k1 和 k2 的和产生。则可以书写如下：

COLUMNS (k2, k1, xxx, v1, v2 = k1 + k2);

再举例，假设用户需要导入只包含 k1 一列的表，列类型为 int。并且需要将源文件中的对应列进行处理：将负数转换为正数，而将正数乘以 100。这个功能可以通过 case when 函数实现，正确写法应如下：

COLUMNS (xx, k1 = case when xx < 0 then cast(-xx as varchar) else cast((xx + '100') as varchar) end)

（3）where\_predicates

用于指定过滤条件，以过滤掉不需要的列。过滤列可以是映射列或衍生列。 例如我们只希望导入 k1 大于 100 并且 k2 等于 1000 的列，则书写如下：

WHERE k1 > 100 and k2 = 1000

（4）partitions

指定导入目的表的哪些 partition 中。如果不指定，则会自动导入到对应的 partition 中。

示例：

PARTITION(p1, p2, p3)

（5）delete\_on\_predicates

表示删除条件，仅在 merge type 为MERGE 时有意义，语法与where 相同

（6）source\_sequence:

只适用于UNIQUE\_KEYS,相同key列下，保证value列按照source\_sequence列进行REPLACE, source\_sequence可以是数据源中的列，也可以是表结构中的一列。

（7）preceding\_predicates

PRECEDING FILTER predicate

用于过滤原始数据。原始数据是未经列映射、转换的数据。用户可以在对转换前的数据前进行一次过滤，选取期望的数据，再进行转换。

**5）job\_properties**

用于指定例行导入作业的通用参数。 语法：

PROPERTIES (

"key1" = "val1",

"key2" = "val2"

)

目前支持以下参数：

（1）desired\_concurrent\_number

期望的并发度。一个例行导入作业会被分成多个子任务执行。这个参数指定一个作业最多有多少任务可以同时执行。必须大于0。默认为3。 这个并发度并不是实际的并发度，实际的并发度，会通过集群的节点数、负载情况，以及数据源的情况综合考虑。

一个作业，最多有多少 task 同时在执行。对于 Kafka 导入而言，当前的实际并发度计算如下：

Min(partition num, desired\_concurrent\_number, alive\_backend\_num, Config.max\_routine\_load\_task\_concurrrent\_num)

其中 Config.max\_routine\_load\_task\_concurrrent\_num 是系统的一个默认的最大并发数限制。这是一个 FE 配置，可以通过改配置调整。默认为 5。

其中 partition num 指订阅的 Kafka topic 的 partition 数量。alive\_backend\_num 是当前正常的 BE 节点数。

（2）max\_batch\_interval/max\_batch\_rows/max\_batch\_size

这三个参数分别表示：

① 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。

② 每个子任务最多读取的行数。必须大于等于200000。默认是200000。

③ 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。 例如：

"max\_batch\_interval" = "20",

"max\_batch\_rows" = "300000",

"max\_batch\_size" = "209715200"

（3）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

（4）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"

（5）timezone

指定导入作业所使用的时区。默认为使用 Session 的 timezone 参数。该参数会影响所有导入涉及的和时区有关的函数结果

（6）format

指定导入数据格式，默认是csv，支持json格式

（7）jsonpaths

jsonpaths: 导入json方式分为：简单模式和匹配模式。如果设置了jsonpath则为匹配模式导入，否则为简单模式导入，具体可参考示例

（8）strip\_outer\_array

布尔类型，为true表示json数据以数组对象开始且将数组对象中进行展平，默认值是false

（9）json\_root

json\_root为合法的jsonpath字符串，用于指定json document的根节点，默认值为""

（10）send\_batch\_parallelism

整型，用于设置发送批处理数据的并行度，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

**6）data\_source\_properties**

数据源的类型。当前支持：Kafka

(

"key1" = "val1",

"key2" = "val2"

)

4.1.3.4 Kafka 导入示例

1）在doris中创建对应表

create table student\_kafka

(

id int,

name varchar(50),

age int

)

DUPLICATE KEY(id)

DISTRIBUTED BY HASH(id) BUCKETS 10;

2）启动kafka并准备数据

bin/kafka-topics.sh --create \

--zookeeper hadoop1:2181/kafka \

--replication-factor 1 \

--partitions 1 \

--topic test\_doris1

bin/kafka-console-producer.sh \

--broker-list hadoop1:9092,hadoop2:9092,hadoop3:9092 \

--topic test\_doris

3）创建导入任务

CREATE ROUTINE LOAD test\_db.kafka\_test ON student\_kafka

COLUMNS TERMINATED BY ",",

COLUMNS(id, name, age)

PROPERTIES

(

"desired\_concurrent\_number"="3",

"strict\_mode" = "false"

)

FROM KAFKA

(

"kafka\_broker\_list"= "hadoop1:9092,hadoop2:9092,hadoop3:9092",

"kafka\_topic" = "test\_doris1",

"property.group.id"="test\_doris\_group",

"property.kafka\_default\_offsets" = "OFFSET\_BEGINNING",

"property.enable.auto.commit"="false"

);

4）查看表

select \* from student\_kafka;

继续往kafka发送数据，查看表的变化

4.1.3.5 查看导入作业状态

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。

4.1.3.6 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。

4.1.3.7 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。

4.1.3.8 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式指定列映射关系，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：

（1）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。

（2）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

**6）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 | 系统会自动查找topic对应的所有分区并从 OFFSET\_END 开始消费 |
| 2 | No | No | Yes | 系统会自动查找topic对应的所有分区并从 default offset 指定的位置开始消费 |
| 3 | Yes | No | No | 系统会从指定分区的 OFFSET\_END 开始消费 |
| 4 | Yes | Yes | No | 系统会从指定分区的指定offset 处开始消费 |
| 5 | Yes | No | Yes | 系统会从指定分区，default offset 指定的位置开始消费 |

**7）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。

### 4.1.4 Binlog Load

Binlog Load提供了一种使Doris增量同步用户在Mysql数据库的对数据更新操作的CDC（Change Data Capture）功能。

4.1.4.1 适用场景

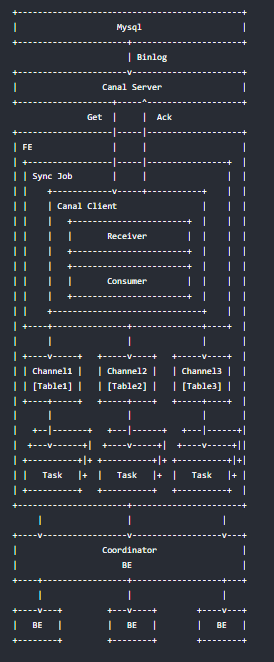
INSERT/UPDATE/DELETE支持。

过滤Query。

暂不兼容DDL语句。

4.1.4.2 基本原理

在第一期的设计中，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：



如上图，用户向FE提交一个数据同步作业。

（1）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。

（2）client中的receiver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。

（3）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。

（4）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。

（5）满足一定条件时（比如超过一定时间、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。

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

（7）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）。

（8）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。

4.1.4.3 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。

1）打开mysql的二进制binlog日志功能，编辑my.cnf配置文件

[mysqld]

log-bin = mysql-bin # 开启 binlog

binlog-format=ROW # 选择 ROW 模式

binlog-do-db=test #指定具体要同步的数据库，也可以不设置

2）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。

编辑my.cnf配置文件。

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，就能很方便的追踪事务、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）。

3）重启MySQL使配置生效

sudo systemctl restart mysqld

4）创建用户并授权

set global validate\_password\_length=4;

set global validate\_password\_policy=0;

GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT ON \*.\* TO 'canal'@'%' IDENTIFIED BY 'canal' ;

5）准备测试表

CREATE TABLE `test`.`tbl1` (

`a` int(11) NOT NULL COMMENT "",

`b` int(11) NOT NULL COMMENT ""

)

insert into test.tbl1 values(1,1),(2,2),(3,3);

4.1.4.4 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。

下载地址：<https://github.com/alibaba/canal/releases>

1）上传并解压canal deployer

mkdir /opt/module/canal-1.1.5

tar -zxvf canal.deployer-1.1.5.tar.gz -C /opt/module/canal-1.1.5

2）在conf文件夹下新建目录并重命名

一个canal服务中可以有多个instance，conf/下的每一个目录即是一个实例，每个实例下面都有独立的配置文件

mkdir /opt/module/canel-1.1.5/conf/doris-load

拷贝配置文件模板

cp /opt/module/canel-1.1.5/conf/example/instance.properties /opt/module/canel-1.1.5/conf/doris-load

3）修改conf/canal.properties的配置

canal.destinations = doris-load

4）修改instance配置文件

vim /opt/module/canel-1.1.5/conf/doris-load/instance.properties

## canal instance serverId

canal.instance.mysql.slaveId = 1234

## mysql address

canal.instance.master.address = hadoop1:3306

## 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/doris-load/doris-load.log

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。

4.1.4.5 配置目标表

1）Doris创建与Mysql对应的目标表

CREATE TABLE `binlog\_test` (

`a` int(11) NOT NULL COMMENT "",

`b` int(11) NOT NULL COMMENT ""

) ENGINE=OLAP

UNIQUE KEY(`a`)

COMMENT "OLAP"

DISTRIBUTED BY HASH(`a`) BUCKETS 8;

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。

2）开启SYNC功能

在fe.conf中将enable\_create\_sync\_job设为true，不想修改配置文件重启，可以执行如下：

使用root账号登陆

ADMIN SET FRONTEND CONFIG ("enable\_create\_sync\_job" = "true");

4.1.4.6 基本语法

创建数据同步作业的的详细语法可以连接到 Doris 后，执行 HELP CREATE SYNC JOB; 查看语法帮助。

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。

FROM mysql\_db.src\_tbl INTO des\_tbl  
[partitions]  
[columns\_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（比如增加一个 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。

FROM BINLOG

(

"key1" = "value1",

"key2" = "value2"

)

canal.server.ip: canal server的地址

canal.server.port: canal server的端口

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

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

canal.username: instance的用户名

canal.password: instance的密码

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。

4.1.4.7 示例

1）创建同步作业

CREATE SYNC test\_db.job1

(

FROM test.tbl1 INTO binlog\_test

)

FROM BINLOG

(

"type" = "canal",

"canal.server.ip" = "hadoop1",

"canal.server.port" = "11111",

"canal.destination" = "doris-load",

"canal.username" = "canal",

"canal.password" = "canal"

);

2）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 SYNC JOB; 命令查看。

展示当前数据库的所有数据同步作业状态。

SHOW SYNC JOB;

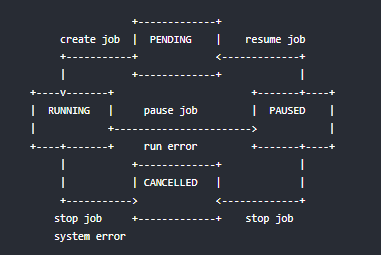
展示数据库 `test\_db` 下的所有数据同步作业状态。

SHOW SYNC JOB FROM `test\_db`;

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。

3）MySQL表继续插入数据，观察Doris的表

4）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。

停止名称为 `job\_name` 的数据同步作业

STOP SYNC JOB [db.]job\_name

暂停名称为 `job\_name` 的数据同步作业

PAUSE SYNC JOB [db.]job\_name

恢复名称为 `job\_name` 的数据同步作业

RESUME SYNC JOB `job\_name`

### 4.1.5 Insert Into

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。

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。

4.1.5.1 语法

INSERT INTO table\_name [partition\_info] [WITH LABEL label] [col\_list] [query\_stmt] [VALUES];

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。

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;

4.1.5.2 SHOW LAST INSERT

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。

### 4.1.6 S3 Load

参考官网：https://doris.apache.org/zh-CN/administrator-guide/load-data/s3-load-manual.html

## 4.2 数据导出

### 4.2.1 Export导出

数据导出是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。

4.2.1.1 基本原理

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。

**1）调度方式**



（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：

① 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。

② EXPORTING：FE 生成 ExportExportingTask，开始执行查询计划。

**2）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。

**3）查询计划执行**

一个作业的多个查询计划顺序执行。

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。

4.2.1.2 基本语法

Export 的详细命令可以通过 HELP EXPORT查看：

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，比如 '\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，比如 '\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。

4.2.1.3 导出示例

1）启动hadoop集群

2）执行导出

export table example\_site\_visit2

to "hdfs://mycluster/doris-export"

PROPERTIES

(

"label" = "mylabel",

"column\_separator"="|",

"timeout" = "3600"

)

WITH BROKER "broker\_name"

(

#HDFS开启HA需要指定，还指定其他参数

"dfs.nameservices"="mycluster",

"dfs.ha.namenodes.mycluster"="nn1,nn2,nn3",

"dfs.namenode.rpc-address.mycluster.nn1"= "hadoop1:8020",

"dfs.namenode.rpc-address.mycluster.nn2"= "hadoop2:8020",

"dfs.namenode.rpc-address.mycluster.nn3"="hadoop3:8020",

"dfs.client.failover.proxy.provider.mycluster"="org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider"

);

3）导出之后查看hdfs对应路径，会多出许多文件

4.2.1.4 查询导出作业状态

提交作业后，可以通过 SHOW EXPORT 命令查询导出作业状态。结果举例如下：

JobId: 14008

Label: mylabel

State: FINISHED

Progress: 100%

TaskInfo: {"partitions":["\*"],"exec mem limit":2147483648,"column separator":",","line delimiter":"\n","tablet num":1,"broker":"hdfs","coord num":1,"db":"default\_cluster:db1","tbl":"tbl3"}

Path: bos://bj-test-cmy/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: N/A

* 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

4.2.1.5 注意事项

（1）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。

（2）如果表数据量过大，建议按照分区导出。

（3）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。

（4）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。

（5）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。比如在百度对象存储（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。

（6）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。

（7）Export 作业只会导出 Base 表的数据，不会导出 Rollup Index 的数据。

（8）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

### 4.2.2 查询结果导出

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。

4.2.2.1 语法

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.。具体参阅Broker 文档。

HDFS 相关属性需加前缀 hdfs. 其中 hdfs.fs.defaultFS 用于填写 namenode 地址和端口。属于必填项。

S3 协议则直接执行 S3 协议配置即可。

示例：

("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。

4.2.2.2 并发导出

1）并发导出的条件

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，比如顶层不包含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。

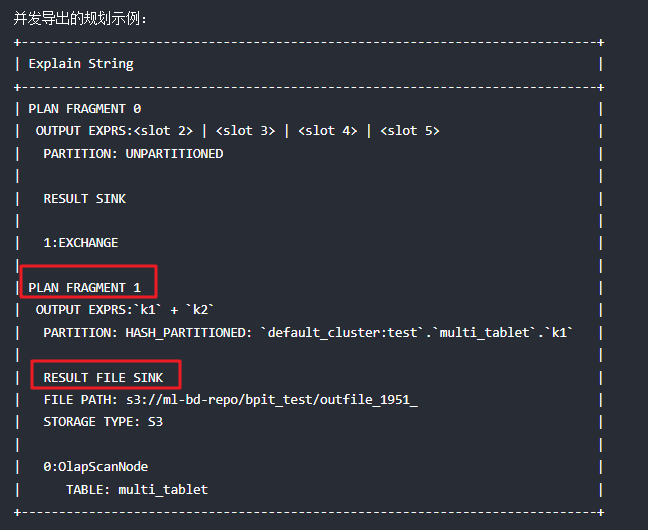
并发度 = be\_instacne\_num \* parallel\_fragment\_exec\_instance\_num

2）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 properties ("AWS\_ENDPOINT" = "xxx", ...);

对查询进行explain后，Doris会返回该查询的规划，如果发现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）。



4.2.2.3 使用示例

**示例一：**使用broker方式，将简单查询结果导出

SELECT \* FROM example\_site\_visit

INTO OUTFILE "hdfs://hadoop1:8020/doris-out/broker\_a\_"

FORMAT AS CSV

PROPERTIES

(

"broker.name" = "broker\_name",

"column\_separator" = ",",

"line\_delimiter" = "\n",

"max\_file\_size" = "100MB"

);

最终生成文件如如果不大于 100MB，则为：`result\_0.csv`。

如果大于100MB，则可能为 `result\_0.csv, result\_1.csv, ...`。

**示例二：**使用broker方式，指定导出格式为 PARQUET

SELECT city, age FROM example\_site\_visit

INTO OUTFILE "hdfs://hadoop1:8020/doris-out/broker\_b\_"

FORMAT AS PARQUET

PROPERTIES

(

"broker.name" = "broker\_name",

"schema"="required,byte\_array,city;required,int32,age"

);

查询结果导出到parquet文件需要明确指定`schema`。

**示例三：**使用 HDFS 方式导出

SELECT \* FROM example\_site\_visit

INTO OUTFILE "hdfs://doris-out/hdfs\_"

FORMAT AS CSV

PROPERTIES

(

"hdfs.fs.defaultFS" = "hdfs://hadoop1:8020",

"hdfs.hdfs\_user" = "atguigu",

"column\_separator" = ","

);

最终生成文件如如果不大于 100MB，则为：`result\_0.csv`。

如果大于 100MB，则可能为 `result\_0.csv, result\_1.csv, ...`。

**示例四：**使用HDFS方式导出，开启并发导出

set enable\_parallel\_outfile = true;

EXPLAIN SELECT \* FROM example\_site\_visit

INTO OUTFILE "hdfs://doris-out/hdfs\_"

FORMAT AS CSV

PROPERTIES

(

"hdfs.fs.defaultFS" = "hdfs://hadoop1:8020",

"hdfs.hdfs\_user" = "atguigu",

"column\_separator" = ","

);

**其他用法：**

（1） 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`my\_broker`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" = "my\_broker",

"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.namenode.ha.ConfiguredFailoverProxyProvider"

);

最终生成文件如如果不大于1GB，则为：`result\_0.csv`。

如果大于 1GB，则可能为 `result\_0.csv, result\_1.csv, ...`。

（2）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`my\_broker`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" = "my\_broker",

"broker.bos\_endpoint" = "http://bj.bcebos.com",

"broker.bos\_accesskey" = "xxxxxxxxxxxxxxxxxxxxxxxxxx",

"broker.bos\_secret\_accesskey" = "yyyyyyyyyyyyyyyyyyyyyyyyyy",

"schema"="required,int32,k1;required,byte\_array,k2"

);

（3）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 "s3a://my\_bucket/export/my\_file\_"

FORMAT AS CSV

PROPERTIES

(

"broker.name" = "hdfs\_broker",

"broker.fs.s3a.access.key" = "xxx",

"broker.fs.s3a.secret.key" = "xxxx",

"broker.fs.s3a.endpoint" = "https://cos.xxxxxx.myqcloud.com/",

"column\_separator" = ",",

"line\_delimiter" = "\n",

"max\_file\_size" = "1024MB",

"success\_file\_name" = "SUCCESS"

)

最终生成文件如如果不大于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

① 不存在的path会自动创建

② 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。

（4）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://s3.bd.bcebos.com",

"AWS\_ACCESS\_KEY" = "xxxx",

"AWS\_SECRET\_KEY" = "xxx",

"AWS\_REGION" = "bd"

)

最终生成的文件前缀为 `my\_file\_{fragment\_instance\_id}\_`。

（5）使用 s3 协议导出到 bos，并且并发导出 session 变量开启。

注意：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。

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://s3.bd.bcebos.com",

"AWS\_ACCESS\_KEY" = "xxxx",

"AWS\_SECRET\_KEY" = "xxx",

"AWS\_REGION" = "bd"

)

# 第5章 查询

## 5.1 查询设置

### 5.1.1 增大内存

一个查询任务，在单个 BE 节点上默认使用不超过 2GB 内存，内存不够时, 查询可能会出现‘Memory limit exceeded’。

SHOW VARIABLES LIKE "%mem\_limit%";

exec\_mem\_limit 的单位是 byte，可以通过 SET 命令改变 exec\_mem\_limit 的值。如改为 8GB。

SET exec\_mem\_limit = 8589934592;

上述设置仅仅在当前session有效, 如果想永久有效, 需要添加 global 参数。

SET GLOBAL exec\_mem\_limit = 8589934592;

### 5.1.2 修改超时时间

doris默认最长查询时间为300s, 如果仍然未完成, 会被cancel掉，查看配置：

SHOW VARIABLES LIKE "%query\_timeout%";

可以修改为60s

SET query\_timeout = 60;

同样, 如果需要全局生效需要添加参数global。

set global query\_timeout = 60;

当前超时的检查间隔为 5 秒，所以小于 5 秒的超时不会太准确。

### 5.1.3 查询重试和高可用

当部署多个 FE 节点时，用户可以在多个 FE 之上部署负载均衡层来实现 Doris 的高可用。

5.1.3.1 代码方式

自己在应用层代码进行重试和负载均衡。比如发现一个连接挂掉，就自动在其他连接上进行重试。应用层代码重试需要应用自己配置多个doris前端节点地址。

5.1.3.2 JDBC Connector

如果使用mysql jdbc connector来连接Doris，可以使用jdbc的自动重试机制：

jdbc:mysql://[host1][:port1],[host2][:port2][,[host3][:port3]]...[/[database]][?propertyName1=propertyValue1[&propertyName2=propertyValue2]...]

5.3.3.3 ProxySQL 方式

ProxySQL是灵活强大的MySQL代理层, 是一个能实实在在用在生产环境的MySQL中间件，可以实现读写分离，支持Query 路由功能，支持动态指定某个SQL进行cache，支持动态加载配置、故障切换和一些SQL的过滤功能。

Doris的FE进程负责接收用户连接和查询请求，其本身是可以横向扩展且高可用的，但是需要用户在多个FE上架设一层proxy，来实现自动的连接负载均衡。

1）安装ProxySQL （yum方式）

配置yum源

# vim /etc/yum.repos.d/proxysql.repo

[proxysql\_repo]

name= ProxySQL YUM repository

baseurl=http://repo.proxysql.com/ProxySQL/proxysql-1.4.x/centos/\$releasever

gpgcheck=1

gpgkey=http://repo.proxysql.com/ProxySQL/repo\_pub\_key

执行安装

# yum clean all

# yum makecache

# yum -y install proxysql

查看版本

# proxysql --version

设置开机自启动

# systemctl enable proxysql

# systemctl start proxysql

# systemctl status proxysql

启动后会监听两个端口， 默认为6032和6033。6032端口是ProxySQL的管理端口，6033是ProxySQL对外提供服务的端口 (即连接到转发后端的真正数据库的转发端口)。

# netstat -tunlp

2）ProxySQL配置

ProxySQL有配置文件 /etc/proxysql.cnf 和配置数据库文件/var/lib/proxysql/proxysql.db。这里需要特别注意：如果存在如果存在"proxysql.db"文件（在/var/lib/proxysql目录下），则 ProxySQL 服务只有在第一次启动时才会去读取proxysql.cnf文件并解析；后面启动会就不会读取proxysql.cnf文件了！如果想要让proxysql.cnf 文件里的配置在重启 proxysql 服务后生效（即想要让 proxysql 重启时读取并解析 proxysql.cnf配置文件），则需要先删除 /var/lib/proxysql/proxysql.db数据库文件，然后再重启 proxysql 服务。这样就相当于初始化启动 proxysql 服务了，会再次生产一个纯净的 proxysql.db 数据库文件(如果之前配置了 proxysql 相关路由规则等，则就会被抹掉)

（1）查看及修改配置文件：主要是几个参数，在下面已经注释出来了，可以根据自己的需要进行修改

# vim /etc/proxysql.cnf

datadir="/var/lib/proxysql" #数据目录

admin\_variables=

{

admin\_credentials="admin:admin" #连接管理端的用户名与密码

mysql\_ifaces="0.0.0.0:6032" #管理端口，用来连接proxysql的管理数据库

}

mysql\_variables=

{

threads=4 #指定转发端口开启的线程数量

max\_connections=2048

default\_query\_delay=0

default\_query\_timeout=36000000

have\_compress=true

poll\_timeout=2000

interfaces="0.0.0.0:6033" #指定转发端口，用于连接后端mysql数据库的，相当于代理作用

default\_schema="information\_schema"

stacksize=1048576

server\_version="5.7.28" #指定后端mysql的版本

connect\_timeout\_server=3000

monitor\_username="monitor"

monitor\_password="monitor"

monitor\_history=600000

monitor\_connect\_interval=60000

monitor\_ping\_interval=10000

monitor\_read\_only\_interval=1500

monitor\_read\_only\_timeout=500

ping\_interval\_server\_msec=120000

ping\_timeout\_server=500

commands\_stats=true

sessions\_sort=true

connect\_retries\_on\_failure=10

}

mysql\_servers =

(

)

mysql\_users:

(

)

mysql\_query\_rules:

(

)

scheduler=

(

)

mysql\_replication\_hostgroups=

(

)

（2）连接ProxySQL管理端口测试

# mysql -h 127.0.0.1 -P 6032 -u admin -p

查看main库（默认登陆后即在此库）的global\_variables表信息

show databases;

use main;

show tables;

（3）ProxySQL 配置后端 Doris FE

使用 insert 语句添加主机到 mysql\_servers 表中，其中：hostgroup\_id 为10表示写组，为20表示读组，我们这里不需要读写分离，无所谓随便设置哪一个都可以。

mysql -u admin -p admin -P 6032 -h 127.0.0.1

insert into mysql\_servers(hostgroup\_id,hostname,port) values(10,'192.168.8.101',9030);

insert into mysql\_servers(hostgroup\_id,hostname,port) values(10,'192.168.8.102',9030);

insert into mysql\_servers(hostgroup\_id,hostname,port) values(10,'192.168.8.103',9030);

如果在插入过程中，出现报错：

ERROR 1045 (#2800): UNIQUE constraint failed: mysql\_servers.hostgroup\_id, mysql\_servers.hostname, mysql\_servers.port

说明可能之前就已经定义了其他配置，可以清空这张表 或者 删除对应host的配置

select \* from mysql\_servers;

delete from mysql\_servers;

查看这3个节点是否插入成功，以及它们的状态。

select \* from mysql\_servers\G;

如上修改后，加载到RUNTIME，并保存到disk，下面两步非常重要，不然退出以后配置信息就没了，必须保存

load mysql servers to runtime;

save mysql servers to disk;

（4）监控Doris FE节点配置

添doris fe 节点之后，还需要监控这些后端节点。对于后端多个FE高可用负载均衡环境来说，这是必须的，因为 ProxySQL 需要通过每个节点的 read\_only 值来自动调整它们是属于读组还是写组。

首先在后端master主数据节点上创建一个用于监控的用户名。

在doris fe master主数据库节点行执行：

# mysql -h hadoop1 -P 9030 -u root -p

create user monitor@'192.168.8.%' identified by 'monitor';

grant ADMIN\_PRIV on \*.\* to monitor@'192.168.8.%';

然后回到mysql-proxy代理层节点上配置监控

# mysql -uadmin -padmin -P6032 -h127.0.0.1

set mysql-monitor\_username='monitor';

set mysql-monitor\_password='monitor';

修改后，加载到RUNTIME，并保存到disk

load mysql variables to runtime;

save mysql variables to disk;

验证监控结果：ProxySQL监控模块的指标都保存在monitor库的log表中。

以下是连接是否正常的监控(对connect指标的监控)：

注意：可能会有很多connect\_error，这是因为没有配置监控信息时的错误，配置后如果connect\_error的结果为NULL则表示正常。

select \* from mysql\_server\_connect\_log;

查看心跳信息的监控(对ping指标的监控)

select \* from mysql\_server\_ping\_log;

查看read\_only日志此时也为空(正常来说，新环境配置时，这个只读日志是为空的)

select \* from mysql\_server\_read\_only\_log;

load mysql servers to runtime;

save mysql servers to disk;

查看结果

select hostgroup\_id,hostname,port,status,weight from mysql\_servers;

（5）配置Doris用户

上面的所有配置都是关于后端 Doris FE节点的，现在可以配置关于SQL语句的，包括：发送SQL语句的用户、SQL语句的路由规则、SQL查询的缓存、SQL语句的重写等等。

本小节是SQL请求所使用的用户配置，例如root用户。这要求我们需要先在后端Doris FE节点添加好相关用户。这里以root和doris两个用户名为例。

首先，在Doris FE master主数据库节点上执行：

# mysql -h hadoop1 -P 9030 -u root -p

root用户已经存在，直接创建doris用户：

create user doris@'%' identified by 'doris';

grant ADMIN\_PRIV on \*.\* to doris@'%';

回到mysql-proxy代理层节点，配置mysql\_users表，将刚才的两个用户添加到该表中。

insert into mysql\_users(username,password,default\_hostgroup) values('root','000000',10);

insert into mysql\_users(username,password,default\_hostgroup) values('doris','doris',10);

加载用户到运行环境中，并将用户信息保存到磁盘

load mysql users to runtime;

save mysql users to disk;

select \* from mysql\_users\G

只有active=1的用户才是有效的用户。确保transaction\_persistent为1：

update mysql\_users set transaction\_persistent=1 where username='root';

update mysql\_users set transaction\_persistent=1 where username='doris';

load mysql users to runtime;

save mysql users to disk;

这里不需要读写分离，将这两个参数设为true：

UPDATE global\_variables SET variable\_value='true' WHERE variable\_name='mysql-forward\_autocommit';

UPDATE global\_variables SET variable\_value='true' WHERE variable\_name='mysql-autocommit\_false\_is\_transaction';

LOAD MYSQL VARIABLES TO RUNTIME;

SAVE MYSQL VARIABLES TO DISK;

这样就可以通过sql客户端，使用doris的用户名密码去连接了ProxySQL了

（6）通过 ProxySQL 连接 Doris 进行测试

分别使用 root 用户和 doris 用户测试下它们是否能路由到默认的 hostgroup\_id=10 （它是一个写组）读数据。下面是通过转发端口6033连接的，连接的是转发到后端真正的数据库。

mysql -udoris -pdoris -P6033 -h hadoop1 -e "show databases;"

到此就结束了，可以用MySQL客户端，JDBC等任何连接MySQL的方式连接ProxySQL 去操作doris了。

（7）验证：将hadoop1的fe停止，再执行

mysql -udoris -pdoris -P6033 -h hadoop1 -e "show databases;"

能够正常使用。

## 5.2 简单查询

1）简单查询

SELECT \* FROM example\_site\_visit LIMIT 3;

SELECT \* FROM example\_site\_visit ORDER BY user\_id;

2）Join

SELECT SUM(example\_site\_visit.cost) FROM example\_site\_visit

JOIN example\_site\_visit2

WHERE example\_site\_visit.user\_id = example\_site\_visit2.user\_id;

select

example\_site\_visit.user\_id,

sum(example\_site\_visit.cost)

from example\_site\_visit join example\_site\_visit2

where example\_site\_visit.user\_id = example\_site\_visit2.user\_id

group by example\_site\_visit.user\_id;

3）子查询

SELECT SUM(cost) FROM example\_site\_visit2 WHERE user\_id IN (SELECT user\_id FROM example\_site\_visit WHERE user\_id > 10003);

## 5.3 Join查询

### 5.3.1 Broadcast Join

系统默认实现Join的方式，是将小表进行条件过滤后，将其广播到大表所在的各个节点上，形成一个内存Hash表，然后流式读出大表的数据进行Hash Join。

Doris会自动尝试进行 Broadcast Join，如果预估小表过大则会自动切换至 Shuffle Join。注意，如果此时显式指定了 Broadcast Join 也会自动切换至 Shuffle Join。

1）默认使用Broadcast Join：

EXPLAIN SELECT SUM(example\_site\_visit.cost)

FROM example\_site\_visit

JOIN example\_site\_visit2

WHERE example\_site\_visit.city = example\_site\_visit2.city;

2）显式使用Broadcast Join：

EXPLAIN SELECT SUM(example\_site\_visit.cost)

FROM example\_site\_visit

JOIN [broadcast] example\_site\_visit2

WHERE example\_site\_visit.city = example\_site\_visit2.city;

### 5.3.2 Shuffle Join（Partitioned Join）

如果当小表过滤后的数据量无法放入内存的话，此时 Join 将无法完成，通常的报错应该是首先造成内存超限。可以显式指定 Shuffle Join，也被称作 Partitioned Join。即将小表和大表都按照 Join 的 key 进行 Hash，然后进行分布式的 Join。这个对内存的消耗就会分摊到集群的所有计算节点上。

SELECT SUM(example\_site\_visit.cost)

FROM example\_site\_visit

JOIN [shuffle] example\_site\_visit2

WHERE example\_site\_visit.city = example\_site\_visit2.city;

### 5.3.3 Colocation Join

Colocation Join是在Doris0.9版本引入的功能，旨在为Join查询提供本性优化，来减少数据在节点上的传输耗时，加速查询。

5.3.3.1 原理

Colocation Join功能，是将一组拥有CGS 的表组成一个CG。保证这些表对应的数据分片会落在同一个be节点上，那么使得两表再进行join的时候，可以通过本地数据进行直接join，减少数据在节点之间的网络传输时间。

* 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。

一个表的数据，最终会根据分桶列值 Hash、对桶数取模的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。

**使用限制：**

（1）建表时两张表的**分桶列的类型和数量需要完全一致**，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）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。

（3）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。

5.3.3.2 使用

1）建两张表，分桶列都为int类型，且桶的个数都是8个。副本数都为默认副本数。

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"

);

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"

);

2）编写查询语句，并查看执行计划

explain SELECT \* FROM tbl1 INNER JOIN tbl2 ON (tbl1.k2 = tbl2.k2);

HASH JOIN处colocate 显示为true，代表优化成功。

3）查看Group

SHOW PROC '/colocation\_group';

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。

4）修改表 Colocate Group属性

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）。

5）删除表的Colocation属性

ALTER TABLE tbl SET ("colocate\_with" = "");

6）其他操作

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。

### 5.3.4 Bucket Shuffle Join

Bucket Shuffle Join是在 Doris 0.14 版本中正式加入的新功能。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。

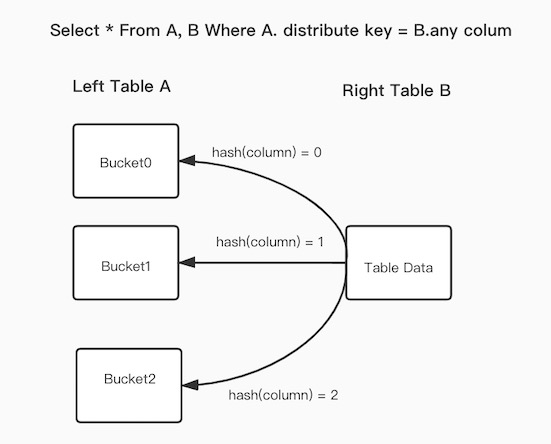
5.3.4.1 原理

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，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，原理如下图：



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：

* 首先，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。
* 其次，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。
* 最后，它可以为Join Reorder提供更多可能的优化空间。

5.3.4.2 使用

1）设置Session变量，从0.14版本开始默认为true

show variables like '%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。

2）通过explain查看join类型

EXPLAIN SELECT SUM(example\_site\_visit.cost)

FROM example\_site\_visit

JOIN example\_site\_visit2

ON example\_site\_visit.user\_id = example\_site\_visit2.user\_id;

在Join类型之中会指明使用的Join方式为：BUCKET\_SHUFFLE。

5.3.4.3 注意事项

（1）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。

（2）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。

（3）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。

（4）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。

（5）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。

（6）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。

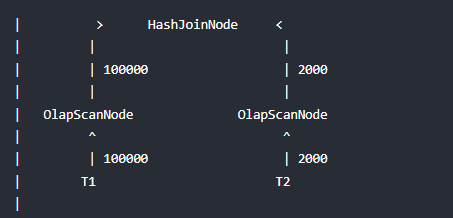
### 5.3.5 Runtime Filter

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。

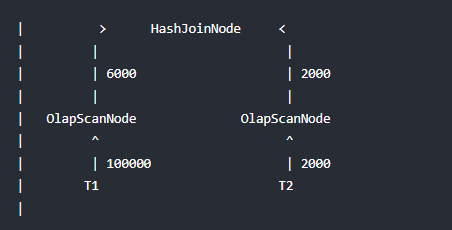
5.3.5.1 原理

Runtime Filter在查询规划时生成，在HashJoinNode中构建，在ScanNode中应用。

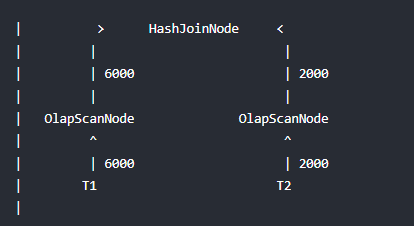
举个例子，当前存在T1表与T2表的Join查询，它的Join方式为HashJoin，T1是一张事实表，数据行数为100000，T2是一张维度表，数据行数为2000，Doris join的实际情况是：



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，比如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：



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：



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。

5.3.5.2 使用

1）指定RuntimeFilter类型

set runtime\_filter\_type="BLOOM\_FILTER,IN,MIN\_MAX";

2）建表

CREATE TABLE test (t1 INT) DISTRIBUTED BY HASH (t1) BUCKETS 2 PROPERTIES("replication\_num" = "1");

INSERT INTO test VALUES (1), (2), (3), (4);

CREATE TABLE test2 (t2 INT) DISTRIBUTED BY HASH (t2) BUCKETS 2 PROPERTIES("replication\_num" = "1");

INSERT INTO test2 VALUES (3), (4), (5);

3）查看执行计划

EXPLAIN SELECT t1 FROM test JOIN test2 where test.t1 = test2.t2;

可以看到：

HASH JOIN`生成了ID为RF000的IN predicate，其中`test2`.`t2`的key values仅在运行时可知，在OlapScanNode使用了该IN predicate用于在读取`test`.`t1`时过滤不必要的数据。

4）通过profile查看效果

set enable\_profile=true;

SELECT t1 FROM test JOIN test2 where test.t1 = test2.t2;

查看对应fe节点的webui，可以查看查询内部工作的详细信息：

http://hadoop1:8030/QueryProfile/

（1）可以看到每个Runtime Filter是否下推、等待耗时、以及OLAP\_SCAN\_NODE从prepare到接收到Runtime Filter的总时长。

RuntimeFilter:in:

- HasPushDownToEngine: true

- AWaitTimeCost: 0ns

- EffectTimeCost: 2.76ms

（2）在profile的OLAP\_SCAN\_NODE中可以查看Runtime Filter下推后的过滤效果和耗时。

- RowsVectorPredFiltered: 9.320008M (9320008)

- VectorPredEvalTime: 364.39ms

5.3.5.3 具体参数说明

1. 大多数情况下，只需要调整runtime\_filter\_type选项，其他选项保持默认即可:

包括BLOOM\_FILTER、IN、MIN\_MAX（也可以通过数字设置），默认会使用IN，部分情况下同时使用Bloom Filter、MinMax Filter、IN predicate时性能更高，每个类型含义如下：

（1）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。

① 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。

② 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。

③ 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（实际是假但误辨为真的情况）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。

（2）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。

① 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。

② 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。

（3）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。

① 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）。

② 目前IN predicate已实现合并方法。

③ 当同时指定In predicate和其他filter，并且in的过滤数值没达到runtime\_filter\_max\_in\_num时，会尝试把其他filter去除掉。原因是In predicate是精确的过滤条件，即使没有其他filter也可以高效过滤，如果同时使用则其他filter会做无用功。目前仅在Runtime filter的生产者和消费者处于同一个fragment时才会有去除非in filter的逻辑。

2）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

5.3.5.4 注意事项

（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，比如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：

（10）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；

（11）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）；

（12）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；

（13）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。

## 5.4 SQL函数

1）查看函数名：

show builtin functions in test\_db;

2）查看函数具体信息，比如查看year函数具体信息

show full builtin functions in test\_db like 'year';

3）官网

<https://doris.apache.org/zh-CN/sql-reference/sql-functions/date-time-functions/convert_tz.html>

# 第6章 集成其他系统

准备表和数据

CREATE TABLE table1

(

siteid INT DEFAULT '10',

citycode SMALLINT,

username VARCHAR(32) DEFAULT '',

pv BIGINT SUM DEFAULT '0'

)

AGGREGATE KEY(siteid, citycode, username)

DISTRIBUTED BY HASH(siteid) BUCKETS 10

PROPERTIES("replication\_num" = "1");

insert into table1 values

(1,1,'jim',2),

(2,1,'grace',2),

(3,2,'tom',2),

(4,3,'bush',3),

(5,3,'helen',3);

## 6.1 Spark读写Doris

### 6.1.1 准备Spark环境

创建maven工程，编写pom.xml文件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.atguigu.doris</groupId>

<artifactId>spark-demo</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<scala.binary.version>2.12</scala.binary.version>

<spark.version>3.0.0</spark.version>

<maven.compiler.source>8</maven.compiler.source>

<maven.compiler.target>8</maven.compiler.target>

</properties>

<dependencies>

<!-- Spark的依赖引入 -->

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-core\_${scala.binary.version}</artifactId>

<scope>provided</scope>

<version>${spark.version}</version>

</dependency>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-sql\_${scala.binary.version}</artifactId>

<scope>provided</scope>

<version>${spark.version}</version>

</dependency>

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-hive\_${scala.binary.version}</artifactId>

<scope>provided</scope>

<version>${spark.version}</version>

</dependency>

<!-- 引入Scala -->

<dependency>

<groupId>org.scala-lang</groupId>

<artifactId>scala-library</artifactId>

<version>2.12.10</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.47</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.49</version>

</dependency>

<!--spark-doris-connector-->

<dependency>

<groupId>org.apache.doris</groupId>

<artifactId>spark-doris-connector-3.1\_2.12</artifactId>

<!--<artifactId>spark-doris-connector-2.3\_2.11</artifactId>-->

<version>1.0.1</version>

</dependency>

</dependencies>

<build>

<plugins>

<!--编译scala所需插件-->

<plugin>

<groupId>org.scala-tools</groupId>

<artifactId>maven-scala-plugin</artifactId>

<version>2.15.1</version>

<executions>

<execution>

<id>compile-scala</id>

<goals>

<goal>add-source</goal>

<goal>compile</goal>

</goals>

</execution>

<execution>

<id>test-compile-scala</id>

<goals>

<goal>add-source</goal>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

</plugin>

<plugin>

<groupId>net.alchim31.maven</groupId>

<artifactId>scala-maven-plugin</artifactId>

<version>3.2.2</version>

<executions>

<execution>

<!-- 声明绑定到maven的compile阶段 -->

<goals>

<goal>compile</goal>

<goal>testCompile</goal>

</goals>

</execution>

</executions>

</plugin>

<!-- assembly打包插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-assembly-plugin</artifactId>

<version>3.0.0</version>

<executions>

<execution>

<id>make-assembly</id>

<phase>package</phase>

<goals>

<goal>single</goal>

</goals>

</execution>

</executions>

<configuration>

<archive>

<manifest>

</manifest>

</archive>

<descriptorRefs>

<descriptorRef>jar-with-dependencies</descriptorRef>

</descriptorRefs>

</configuration>

</plugin>

<!-- <plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.6.1</version>

&lt;!&ndash; 所有的编译都依照JDK1.8 &ndash;&gt;

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>-->

</plugins>

</build>

</project>

### 6.1.2 使用Spark Doris Connector

Spark Doris Connector 可以支持通过 Spark 读取 Doris 中存储的数据，也支持通过Spark写入数据到Doris。

6.1.2.1 SQL方式读写数据

package com.atuigu.doris.spark

import org.apache.spark.SparkConf

import org.apache.spark.sql.SparkSession

/\*\*

\* TODO

\*

\* @version 1.0

\* @author cjp

\*/

object SQLDemo {

def main( args: Array[String] ): Unit = {

val sparkConf = new SparkConf().setAppName("SQLDemo")

.setMaster("local[\*]") //TODO 要打包提交集群执行，注释掉

val sparkSession = SparkSession.builder().config(sparkConf).getOrCreate()

sparkSession.sql(

"""

|CREATE TEMPORARY VIEW spark\_doris

|USING doris

|OPTIONS(

| "table.identifier"="test\_db.table1",

| "fenodes"="hadoop1:8030",

| "user"="test",

| "password"="test"

|);

""".stripMargin)

//读取数据

// sparkSession.sql("select \* from spark\_doris").show()

//写入数据

sparkSession.sql("insert into spark\_doris values(99,99,'haha',5)")

}

}

6.1.2.2 DataFrame方式读写数据（batch）

package com.atuigu.doris.spark

import org.apache.spark.SparkConf

import org.apache.spark.sql.SparkSession

/\*\*

\* TODO

\*

\* @version 1.0

\* @author cjp

\*/

object DataFrameDemo {

def main( args: Array[String] ): Unit = {

val sparkConf = new SparkConf().setAppName("DataFrameDemo")

.setMaster("local[\*]") //TODO 要打包提交集群执行，注释掉

val sparkSession = SparkSession.builder().config(sparkConf).getOrCreate()

// 读取数据

// val dorisSparkDF = sparkSession.read.format("doris")

// .option("doris.table.identifier", "test\_db.table1")

// .option("doris.fenodes", "hadoop1:8030")

// .option("user", "test")

// .option("password", "test")

// .load()

// dorisSparkDF.show()

// 写入数据

import sparkSession.implicits.\_

val mockDataDF = List(

(11,23, "haha", 8),

(11, 3, "hehe", 9),

(11, 3, "heihei", 10)

).toDF("siteid", "citycode", "username","pv")

mockDataDF.show(5)

mockDataDF.write.format("doris")

.option("doris.table.identifier", "test\_db.table1")

.option("doris.fenodes", "hadoop1:8030")

.option("user", "test")

.option("password", "test")

//指定你要写入的字段

// .option("doris.write.fields", "user")

.save()

}

}

6.1.2.3 RDD方式读取数据

package com.atuigu.doris.spark

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.sql.SparkSession

/\*\*

\* TODO

\*

\* @version 1.0

\* @author cjp

\*/

object RDDDemo {

def main( args: Array[String] ): Unit = {

val sparkConf = new SparkConf().setAppName("RDDDemo")

.setMaster("local[\*]") //TODO 要打包提交集群执行，注释掉

val sc = new SparkContext(sparkConf)

import org.apache.doris.spark.\_

val dorisSparkRDD = sc.dorisRDD(

tableIdentifier = Some("test\_db.table1"),

cfg = Some(Map(

"doris.fenodes" -> "hadoop1:8030",

"doris.request.auth.user" -> "test",

"doris.request.auth.password" -> "test"

))

)

dorisSparkRDD.collect().foreach(println)

}

}

6.1.2.4 配置和字段类型映射

1）通用配置项

|  |  |  |
| --- | --- | --- |
| Key | Default Value | Comment |
| doris.fenodes | -- | Doris FE http 地址，支持多个地址，使用逗号分隔 |
| doris.table.identifier | -- | Doris 表名，如：db1.tbl1 |
| doris.request.retries | 3 | 向Doris发送请求的重试次数 |
| doris.request.connect.timeout.ms | 30000 | 向Doris发送请求的连接超时时间 |
| doris.request.read.timeout.ms | 30000 | 向Doris发送请求的读取超时时间 |
| doris.request.query.timeout.s | 3600 | 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 | 一个RDD 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Spark侧的并行度，但同时会对Doris造成更大的压力。 |
| doris.batch.size | 1024 | 一次从BE读取数据的最大行数。增大此数值可减少Spark与Doris之间建立连接的次数。从而减轻网络延迟所带来的的额外时间开销。 |
| doris.exec.mem.limit | 2147483648 | 单个查询的内存限制。默认为 2GB，单位为字节 |
| doris.deserialize.arrow.async | false | 是否支持异步转换Arrow格式到spark-doris-connector迭代所需的RowBatch |
| doris.deserialize.queue.size | 64 | 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.write.fields | -- | 指定写入Doris表的字段或者字段顺序，多列之间使用逗号分隔。默认写入时要按照Doris表字段顺序写入全部字段。 |
| sink.batch.size | 10000 | 单次写BE的最大行数 |
| sink.max-retries | 1 | 写BE失败之后的重试次数 |

2）SQL 和 Dataframe 专有配置

|  |  |  |
| --- | --- | --- |
| Key | Default Value | Comment |
| user | -- | 访问Doris的用户名 |
| password | -- | 访问Doris的密码 |
| doris.filter.query.in.max.count | 100 | 谓词下推中，in表达式value列表元素最大数量。超过此数量，则in表达式条件过滤在Spark侧处理。 |

3）RDD 专有配置

|  |  |  |
| --- | --- | --- |
| Key | Default Value | Comment |
| doris.request.auth.user | -- | 访问Doris的用户名 |
| doris.request.auth.password | -- | 访问Doris的密码 |
| doris.read.field | -- | 读取Doris表的列名列表，多列之间使用逗号分隔 |
| doris.filter.query | -- | 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。 |

4）Doris 和 Spark 列类型映射关系:

|  |  |
| --- | --- |
| Doris Type | Spark Type |
| NULL\_TYPE | DataTypes.NullType |
| BOOLEAN | DataTypes.BooleanType |
| TINYINT | DataTypes.ByteType |
| SMALLINT | DataTypes.ShortType |
| INT | DataTypes.IntegerType |
| BIGINT | DataTypes.LongType |
| FLOAT | DataTypes.FloatType |
| DOUBLE | DataTypes.DoubleType |
| DATE | DataTypes.StringType1 |
| DATETIME | DataTypes.StringType1 |
| BINARY | DataTypes.BinaryType |
| DECIMAL | DecimalType |
| CHAR | DataTypes.StringType |
| LARGEINT | DataTypes.StringType |
| VARCHAR | DataTypes.StringType |
| DECIMALV2 | DecimalType |
| TIME | DataTypes.DoubleType |
| HLL | Unsupported datatype |

注：Connector中，将DATE和DATETIME映射为String。由于Doris底层存储引擎处理逻辑，直接使用时间类型时，覆盖的时间范围无法满足需求。所以使用 String 类型直接返回对应的时间可读文本。

### 6.1.3 使用JDBC的方式（不推荐）

这种方式是早期写法，Spark无法感知Doris的数据分布，会导致打到Doris的查询压力非常大。

package com.atuigu.doris.spark

import java.util.Properties

import org.apache.spark.SparkConf

import org.apache.spark.sql.{SaveMode, SparkSession}

object JDBCDemo {

def main(args: Array[String]): Unit = {

val sparkConf = new SparkConf().setAppName("JDBCDemo").setMaster("local[\*]")

val sparkSession = SparkSession.builder().config(sparkConf).getOrCreate()

// 读取数据

// val df=sparkSession.read.format("jdbc")

// .option("url","jdbc:mysql://hadoop1:9030/test\_db")

// .option("user","test")

// .option("password","test")

// .option("dbtable","table1")

// .load()

//

// df.show()

// 写入数据

import sparkSession.implicits.\_

val mockDataDF = List(

(11,23, "haha", 8),

(11, 3, "hehe", 9),

(11, 3, "heihei", 10)

).toDF("siteid", "citycode", "username","pv")

val prop = new Properties()

prop.setProperty("user", "root")

prop.setProperty("password", "123456")

df.write.mode(SaveMode.Append)

.jdbc("jdbc:mysql://hadoop1:9030/test\_db", "table1", prop)

}

}

## 6.2 Flink Doris Connector

Flink Doris Connector 可以支持通过 Flink 操作（读取、插入、修改、删除） Doris 中存储的数据。

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 请求参数的配置设定。

版本兼容如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Connector | Flink | Doris | Java | Scala |
| 1.11.6-2.12-xx | 1.11.x | 0.13+ | 8 | 2.12 |
| 1.12.7-2.12-xx | 1.12.x | 0.13.+ | 8 | 2.12 |
| 1.13.5-2.12-xx | 1.13.x | 0.13.+ | 8 | 2.12 |
| 1.14.4-2.12-xx | 1.14.x | 0.13.+ | 8 | 2.12 |

### 6.2.1 准备Flink环境

创建maven工程，编写pom.xml文件

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.atguigu.doris</groupId>

<artifactId>flink-demo</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<maven.compiler.source>8</maven.compiler.source>

<maven.compiler.target>8</maven.compiler.target>

<flink.version>1.13.1</flink.version>

<java.version>1.8</java.version>

<scala.binary.version>2.12</scala.binary.version>

<slf4j.version>1.7.30</slf4j.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-java</artifactId>

<version>${flink.version}</version>

<scope>provided</scope> <!--不会打包到依赖中，只参与编译，不参与运行 -->

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-streaming-java\_${scala.binary.version}</artifactId>

<version>${flink.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-clients\_${scala.binary.version}</artifactId>

<version>${flink.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-table-planner-blink\_${scala.binary.version}</artifactId>

<version>${flink.version}</version>

<scope>provided</scope>

</dependency>

<!---->

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-runtime-web\_${scala.binary.version}</artifactId>

<version>${flink.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>${slf4j.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>${slf4j.version}</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-to-slf4j</artifactId>

<version>2.14.0</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.49</version>

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-statebackend-rocksdb\_${scala.binary.version}</artifactId>

<version>${flink.version}</version>

</dependency>

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-sequence-file</artifactId>

<version>${flink.version}</version>

</dependency>

<dependency>

<groupId>com.ververica</groupId>

<artifactId>flink-connector-mysql-cdc</artifactId>

<version>2.0.0</version>

</dependency>

<!--flink-doris-connector-->

<dependency>

<groupId>org.apache.doris</groupId>

<!--<artifactId>flink-doris-connector-1.14\_2.12</artifactId>-->

<artifactId>flink-doris-connector-1.13\_2.12</artifactId>

<!--<artifactId>flink-doris-connector-1.12\_2.12</artifactId>-->

<!--<artifactId>flink-doris-connector-1.11\_2.12</artifactId>-->

<version>1.0.3</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>3.2.4</version>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<artifactSet>

<excludes>

<exclude>com.google.code.findbugs:jsr305</exclude>

<exclude>org.slf4j:\*</exclude>

<exclude>log4j:\*</exclude>

<exclude>org.apache.hadoop:\*</exclude>

</excludes>

</artifactSet>

<filters>

<filter>

<!-- Do not copy the signatures in the META-INF folder.

Otherwise, this might cause SecurityExceptions when using the JAR. -->

<artifact>\*:\*</artifact>

<excludes>

<exclude>META-INF/\*.SF</exclude>

<exclude>META-INF/\*.DSA</exclude>

<exclude>META-INF/\*.RSA</exclude>

</excludes>

</filter>

</filters>

<transformers combine.children="append">

<transformer implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer">

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

### 6.2.2 SQL方式读写

package com.atuigu.doris.flink;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/\*\*

\* TODO

\*

\* @author cjp

\* @version 1.0

\*/

public class SQLDemo {

public static void main(String[] args) {

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setParallelism(1);

StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

tableEnv.executeSql("CREATE TABLE flink\_doris (\n" +

" siteid INT,\n" +

" citycode SMALLINT,\n" +

" username STRING,\n" +

" pv BIGINT\n" +

" ) \n" +

" WITH (\n" +

" 'connector' = 'doris',\n" +

" 'fenodes' = 'hadoop1:8030',\n" +

" 'table.identifier' = 'test\_db.table1',\n" +

" 'username' = 'test',\n" +

" 'password' = 'test'\n" +

")\n");

// 读取数据

// tableEnv.executeSql("select \* from flink\_doris").print();

// 写入数据

tableEnv.executeSql("insert into flink\_doris(siteid,username,pv) values(22,'wuyanzu',3)");

}

}

### 6.2.3 DataStream读写

6.2.3.1 Source

package com.atuigu.doris.flink;

import org.apache.doris.flink.cfg.DorisStreamOptions;

import org.apache.doris.flink.datastream.DorisSourceFunction;

import org.apache.doris.flink.deserialization.SimpleListDeserializationSchema;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.util.Properties;

/\*\*

\* TODO

\*

\* @author cjp

\* @version 1.0

\*/

public class DataStreamSourceDemo {

public static void main(String[] args) throws Exception {

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setParallelism(1);

Properties properties = new Properties();

properties.put("fenodes","hadoop1:8030");

properties.put("username","test");

properties.put("password","test");

properties.put("table.identifier","test\_db.table1");

env.addSource(new DorisSourceFunction(

new DorisStreamOptions(properties),

new SimpleListDeserializationSchema()

)

).print();

env.execute();

}

}

6.2.3.2 Sink

1）Json数据流写法一

package com.atuigu.doris.flink;

import org.apache.doris.flink.cfg.\*;

import org.apache.doris.flink.datastream.DorisSourceFunction;

import org.apache.doris.flink.deserialization.SimpleListDeserializationSchema;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.Properties;

/\*\*

\* TODO

\*

\* @author cjp

\* @version 1.0

\*/

public class DataStreamJsonSinkDemo {

public static void main(String[] args) throws Exception {

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setParallelism(1);

Properties pro = new Properties();

pro.setProperty("format", "json");

pro.setProperty("strip\_outer\_array", "true");

env

.fromElements(

"{\"longitude\": \"116.405419\", \"city\": \"北京\", \"latitude\": \"39.916927\"}"

)

.addSink(

DorisSink.sink(

DorisReadOptions.builder().build(),

DorisExecutionOptions.builder()

.setBatchSize(3)

.setBatchIntervalMs(0L)

.setMaxRetries(3)

.setStreamLoadProp(pro).build(),

DorisOptions.builder()

.setFenodes("FE\_IP:8030")

.setTableIdentifier("db.table")

.setUsername("root")

.setPassword("").build()

));

// .addSink(

// DorisSink.sink(

// DorisOptions.builder()

// .setFenodes("FE\_IP:8030")

// .setTableIdentifier("db.table")

// .setUsername("root")

// .setPassword("").build()

// ));

env.execute();

}

}

2）RowData数据流

package com.atuigu.doris.flink;

import org.apache.doris.flink.cfg.DorisExecutionOptions;

import org.apache.doris.flink.cfg.DorisOptions;

import org.apache.doris.flink.cfg.DorisReadOptions;

import org.apache.doris.flink.cfg.DorisSink;

import org.apache.flink.api.common.functions.MapFunction;

import org.apache.flink.streaming.api.datastream.DataStream;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import org.apache.flink.table.data.GenericRowData;

import org.apache.flink.table.data.RowData;

import org.apache.flink.table.data.StringData;

import org.apache.flink.table.types.logical.\*;

/\*\*

\* TODO

\*

\* @author cjp

\* @version 1.0

\*/

public class DataStreamRowDataSinkDemo {

public static void main(String[] args) throws Exception {

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setParallelism(1);

DataStream<RowData> source = env.fromElements("")

.map(new MapFunction<String, RowData>() {

@Override

public RowData map(String value) throws Exception {

GenericRowData genericRowData = new GenericRowData(4);

genericRowData.setField(0, 33);

genericRowData.setField(1, new Short("3"));

genericRowData.setField(2, StringData.fromString("flink-stream"));

genericRowData.setField(3, 3L);

return genericRowData;

}

});

LogicalType[] types = {new IntType(), new SmallIntType(), new VarCharType(32), new BigIntType()};

String[] fields = {"siteid", "citycode", "username", "pv"};

source.addSink(

DorisSink.sink(

fields,

types,

DorisReadOptions.builder().build(),

DorisExecutionOptions.builder()

.setBatchSize(3)

.setBatchIntervalMs(0L)

.setMaxRetries(3)

.build(),

DorisOptions.builder()

.setFenodes("hadoop1:8030")

.setTableIdentifier("test\_db.table1")

.setUsername("test")

.setPassword("test").build()

));

env.execute();

}

}

### 6.2.4 通用配置项和字段类型映射

1）通用配置项：

|  |  |  |
| --- | --- | --- |
| Key | Default Value | Comment |
| fenodes | -- | Doris FE http 地址 |
| table.identifier | -- | Doris 表名，如：db1.tbl1 |
| username | -- | 访问Doris的用户名 |
| password | -- | 访问Doris的密码 |
| doris.request.retries | 3 | 向Doris发送请求的重试次数 |
| doris.request.connect.timeout.ms | 30000 | 向Doris发送请求的连接超时时间 |
| doris.request.read.timeout.ms | 30000 | 向Doris发送请求的读取超时时间 |
| doris.request.query.timeout.s | 3600 | 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 | 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 | 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的额外时间开销。 |
| doris.exec.mem.limit | 2147483648 | 单个查询的内存限制。默认为 2GB，单位为字节 |
| doris.deserialize.arrow.async | false | 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 | 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 | -- | 读取Doris表的列名列表，多列之间使用逗号分隔 |
| doris.filter.query | -- | 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.batch.size | 10000 | 单次写BE的最大行数 |
| sink.max-retries | 1 | 写BE失败之后的重试次数 |
| sink.batch.interval | 10s | flush 间隔时间，超过该时间后异步线程将 缓存中数据写入BE。 默认值为10秒，支持时间单位ms、s、min、h和d。设置为0表示关闭定期写入。 |
| sink.properties.\* | -- | 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 'sink.properties.format' = 'json' 'sink.properties.strip\_outer\_array' = 'true' JSON格式导入 |
| sink.enable-delete | true | 是否启用删除。此选项需要Doris表开启批量删除功能(0.15+版本默认开启)，只支持Uniq模型。 |
| sink.batch.bytes | 10485760 | 单次写BE的最大数据量，当每个 batch 中记录的数据量超过该阈值时，会将缓存数据写入 BE。默认值为 10MB |

2）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 | STRING |
| DATETIME | STRING |
| DECIMAL | DECIMAL |
| CHAR | STRING |
| LARGEINT | STRING |
| VARCHAR | STRING |
| DECIMALV2 | DECIMAL |
| TIME | DOUBLE |
| HLL | Unsupported datatype |

## 6.3 DataX doriswriter

DorisWriter支持将大批量数据写入Doris中。DorisWriter 通过Doris原生支持Stream load方式导入数据， DorisWriter会将reader读取的数据进行缓存在内存中，拼接成Json文本，然后批量导入至Doris。

### 6.3.1 编译

可以自己编译，也可以直接使用我们编译好的包。

1）进入之前的容器环境

docker run -it \

-v /opt/software/.m2:/root/.m2 \

-v /opt/software/apache-doris-0.15.0-incubating-src/:/root/apache-doris-0.15.0-incubating-src/ \

apache/incubator-doris:build-env-for-0.15.0

2）运行 init-env.sh

cd /root/apache-doris-0.15.0-incubating-src/extension/DataX

sh init-env.sh

3）手动上传依赖

上传alibaba-datax-maven-m2-20210928.tar.gz，解压：

tar -zxvf alibaba-datax-maven-m2-20210928.tar.gz -C /opt/software

拷贝解压后的文件到maven仓库

sudo cp -r /opt/software/alibaba/datax/ /opt/software/.m2/repository/com/alibaba/

4）编译 doriswriter：

（1）单独编译 doriswriter 插件:

cd /root/apache-doris-0.15.0-incubating-src/extension/DataX/DataX

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

（2）编译整个 DataX 项目:

cd /root/apache-doris-0.15.0-incubating-src/extension/DataX/DataX

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

产出在target/datax/datax/.

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

5）拷贝编译好的插件到DataX

Sudo cp -r /opt/software/apache-doris-0.15.0-incubating-src/extension/DataX/doriswriter/target/datax/plugin/writer/doriswriter /opt/module/datax/plugin/writer

### 6.3.2 使用

1）准备测试表

MySQL建表、插入测试数据

CREATE TABLE `sensor` (

`id` varchar(255) NOT NULL,

`ts` bigint(255) DEFAULT NULL,

`vc` int(255) DEFAULT NULL,

PRIMARY KEY (`id`)

)

insert into sensor values('s\_2',3,3),('s\_9',9,9);

Doris建表

CREATE TABLE `sensor` (

`id` varchar(255) NOT NULL,

`ts` bigint(255) DEFAULT NULL,

`vc` int(255) DEFAULT NULL

)

DISTRIBUTED BY HASH(`id`) BUCKETS 10;

2）编写json文件

vim mysql2doris.json

{

"job": {

"setting": {

"speed": {

"channel": 1

},

"errorLimit": {

"record": 0,

"percentage": 0

}

},

"content": [

{

"reader": {

"name": "mysqlreader",

"parameter": {

"column": [

"id",

"ts",

"vc"

],

"connection": [

{

"jdbcUrl": [

"jdbc:mysql://hadoop1:3306/test"

],

"table": [

"sensor"

]

}

],

"username": "root",

"password": "000000"

}

},

"writer": {

"name": "doriswriter",

"parameter": {

"feLoadUrl": ["hadoop1:8030", "hadoop2:8030", "hadoop3:8030"],

"beLoadUrl": ["hadoop1:8040", "hadoop2:8040", "hadoop3:8040"],

"jdbcUrl": "jdbc:mysql://hadoop1:9030/",

"database": "test\_db",

"table": "sensor",

"column": ["id", "ts", "vc"],

"username": "test",

"password": "test",

"postSql": [],

"preSql": [],

"loadProps": {

},

"maxBatchRows" : 500000,

"maxBatchByteSize" : 104857600,

"labelPrefix": "my\_prefix",

"lineDelimiter": "\n"

}

}

}

]

}

}

3）运行datax任务

bin/datax.py job/mysql2doris.json

### 6.3.3 参数说明

* jdbcUrl

描述：Doris 的 JDBC 连接串，用户执行 preSql 或 postSQL。

必选：是

默认值：无

* feLoadUrl

描述：和 beLoadUrl 二选一。作为 Stream Load 的连接目标。格式为 "ip:port"。其中 IP 是 FE 节点 IP，port 是 FE 节点的 http\_port。可以填写多个，doriswriter 将以轮询的方式访问。

必选：否

默认值：无

* beLoadUrl

描述：和 feLoadUrl 二选一。作为 Stream Load 的连接目标。格式为 "ip:port"。其中 IP 是 BE 节点 IP，port 是 BE 节点的 webserver\_port。可以填写多个，doriswriter 将以轮询的方式访问。

必选：否

默认值：无

* username

描述：访问Doris数据库的用户名

必选：是

默认值：无

* password

描述：访问Doris数据库的密码

必选：否

默认值：空

* database

描述：需要写入的Doris数据库名称。

必选：是

默认值：无

* table

描述：需要写入的Doris表名称。

必选：是

默认值：无

* column

描述：目的表\*\*需要写入数据\*\*的字段，这些字段将作为生成的 Json 数据的字段名。字段之间用英文逗号分隔。例如: "column": ["id","name","age"]。

必选：是

默认值：否

* preSql

描述：写入数据到目的表前，会先执行这里的标准语句。

必选：否

默认值：无

* postSql

描述：写入数据到目的表后，会执行这里的标准语句。

必选：否

默认值：无

* maxBatchRows

描述：每批次导入数据的最大行数。和 maxBatchByteSize 共同控制每批次的导入数量。每批次数据达到两个阈值之一，即开始导入这一批次的数据。

必选：否

默认值：500000

* maxBatchByteSize

描述：每批次导入数据的最大数据量。和 \*\* maxBatchRows\*\* 共同控制每批次的导入数量。每批次数据达到两个阈值之一，即开始导入这一批次的数据。

必选：否

默认值：104857600

* labelPrefix

描述：每批次导入任务的 label 前缀。最终的 label 将有 labelPrefix + UUID + 序号 组成

必选：否

默认值：datax\_doris\_writer\_

* lineDelimiter

描述：每批次数据包含多行，每行为 Json 格式，每行的的分隔符即为 lineDelimiter。支持多个字节, 例如'\x02\x03'。

必选：否

默认值：\n

* loadProps

描述：StreamLoad 的请求参数，详情参照StreamLoad介绍页面。

必选：否

默认值：无

* connectTimeout

描述：StreamLoad单次请求的超时时间, 单位毫秒(ms)。

必选：否

默认值：-1

## 6.4 ODBC外部表

ODBC External Table Of Doris 提供了Doris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Doris可以具有了访问各式数据库的能力，并借助Doris本身的OLAP的能力来解决外部表的数据分析问题：

（1）支持各种数据源接入Doris

（2）支持Doris与各种数据源中的表联合查询，进行更加复杂的分析操作

（3）通过insert into将Doris执行的查询结果写入外部的数据源

### 6.4.1 使用方式

1）Doris中创建ODBC的外表

方式一：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"

);

方式二：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Driver名，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 |

2）ODBC Driver的安装和配置

各大主流数据库都会提供ODBC的访问Driver，用户可以执行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

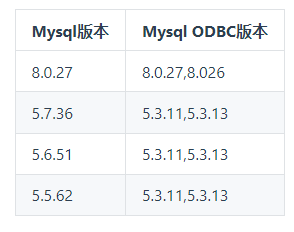
上述配置[]里的对应的是Driver名，在建立外部表时需要保持外部表的Driver名和配置文件之中的一致。

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。

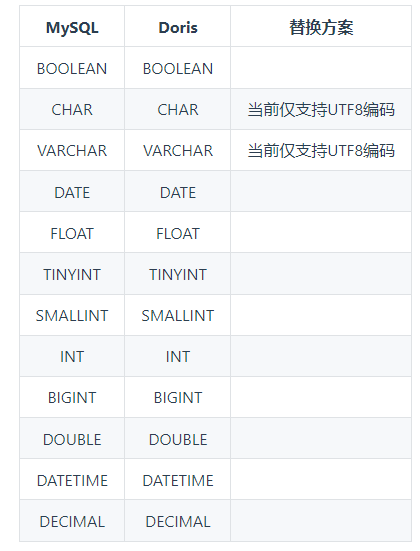
切记，这里要求所有的BE节点都安装上相同的Driver，并且安装路径相同，同时有相同的be/conf/odbcinst.ini的配置。

### 6.4.2 使用ODBC的MySQL外表

CentOS数据库ODBC版本对应关系：



MySQL与Doris的数据类型匹配：



1）安装unixODBC

安装

yum install -y unixODBC unixODBC-devel libtool-ltdl libtool-ltdl-devel

查看是否安装成功

odbcinst -j

2）安装MySQL对应版本的ODBC（每个BE节点都要）

下载

wget <https://downloads.mysql.com/archives/get/p/10/file/mysql-connector-odbc-5.3.11-1.el7.x86_64.rpm>

安装

yum install -y mysql-connector-odbc-5.3.11-1.el7.x86\_64.rpm

查看是否安装成功

myodbc-installer -d -l

3）配置unixODBC，验证通过ODBC访问Mysql

编辑ODBC配置文件

vim /etc/odbc.ini

[mysql]

Description = Data source MySQL

Driver = MySQL ODBC 5.3 Unicode Driver

Server = hadoop1

Host = hadoop1

Database = test

Port = 3306

User = root

Password = 000000

测试链接

isql -v mysql

4）准备MySQL表

CREATE TABLE `test\_cdc` (

`id` int NOT NULL AUTO\_INCREMENT,

`name` varchar(255) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=91234 DEFAULT CHARSET=utf8mb4;

INSERT INTO `test\_cdc` VALUES (123, 'this is a update');

INSERT INTO `test\_cdc` VALUES (1212, '测试flink CDC');

INSERT INTO `test\_cdc` VALUES (1234, '这是测试');

INSERT INTO `test\_cdc` VALUES (11233, 'zhangfeng\_1');

INSERT INTO `test\_cdc` VALUES (21233, 'zhangfeng\_2');

INSERT INTO `test\_cdc` VALUES (31233, 'zhangfeng\_3');

INSERT INTO `test\_cdc` VALUES (41233, 'zhangfeng\_4');

INSERT INTO `test\_cdc` VALUES (51233, 'zhangfeng\_5');

INSERT INTO `test\_cdc` VALUES (61233, 'zhangfeng\_6');

INSERT INTO `test\_cdc` VALUES (71233, 'zhangfeng\_7');

INSERT INTO `test\_cdc` VALUES (81233, 'zhangfeng\_8');

INSERT INTO `test\_cdc` VALUES (91233, 'zhangfeng\_9');

5）修改Doris的配置文件（每个BE节点都要，不用重启BE）

在BE节点的conf/odbcinst.ini，添加我们的刚才注册的的ODBC驱动（[MySQL ODBC 5.3.11]这部分）。

# Driver from the postgresql-odbc package

# Setup from the unixODBC package

[PostgreSQL]

Description = ODBC for PostgreSQL

Driver = /usr/lib/psqlodbc.so

Setup = /usr/lib/libodbcpsqlS.so

FileUsage = 1

​

​

# Driver from the mysql-connector-odbc package

# Setup from the unixODBC package

[MySQL ODBC 5.3.11]

Description = ODBC for MySQL

Driver= /usr/lib64/libmyodbc5w.so

FileUsage = 1

​

​

# Driver from the oracle-connector-odbc package

# Setup from the unixODBC package

[Oracle 19 ODBC driver]

Description=Oracle ODBC driver for Oracle 19

Driver=/usr/lib/libsqora.so.19.1

6）Doris建Resource

通过ODBC\_Resource来创建ODBC外表，这是推荐的方式，这样resource可以复用。

CREATE EXTERNAL RESOURCE `mysql\_5\_3\_11`

PROPERTIES (

"host" = "hadoop1",

"port" = "3306",

"user" = "root",

"password" = "000000",

"database" = "test",

"table" = "test\_cdc",

"driver" = "MySQL ODBC 5.3.11", --名称要和上面[]里的名称一致

"odbc\_type" = "mysql",

"type" = "odbc\_catalog")

7）基于Resource创建Doris外表

CREATE EXTERNAL TABLE `test\_odbc\_5\_3\_11` (

`id` int NOT NULL ,

`name` varchar(255) null

) ENGINE=ODBC

COMMENT "ODBC"

PROPERTIES (

"odbc\_catalog\_resource" = "mysql\_5\_3\_11", --名称就是resource的名称

"database" = "test",

"table" = "test\_cdc"

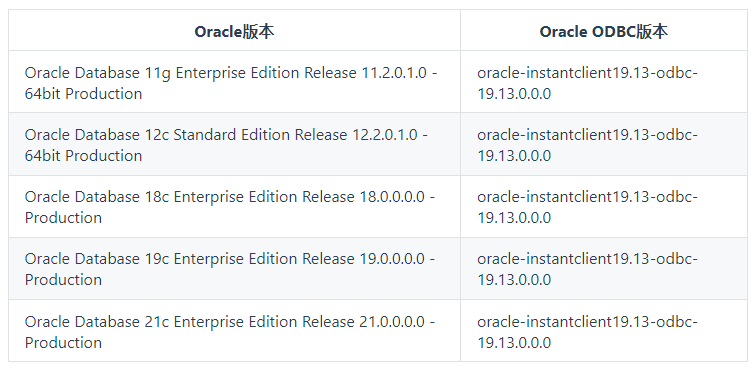
);

8）查询Doris外表

select \* from `test\_odbc\_5\_3\_11`;

### 6.4.3 使用ODBC的Oracle外表

CentOS数据库ODBC版本对应关系：



与Doris的数据类型匹配：



1）安装unixODBC

安装

yum install -y unixODBC unixODBC-devel libtool-ltdl libtool-ltdl-devel

查看是否安装成功

odbcinst -j

2）安装Oracle对应版本的ODBC（每个BE节点都要）

下载4个安装包

wget https://download.oracle.com/otn\_software/linux/instantclient/1913000/oracle-instantclient19.13-sqlplus-19.13.0.0.0-2.x86\_64.rpm

wget https://download.oracle.com/otn\_software/linux/instantclient/1913000/oracle-instantclient19.13-devel-19.13.0.0.0-2.x86\_64.rpm

wget https://download.oracle.com/otn\_software/linux/instantclient/1913000/oracle-instantclient19.13-odbc-19.13.0.0.0-2.x86\_64.rpm

wget https://download.oracle.com/otn\_software/linux/instantclient/1913000/oracle-instantclient19.13-basic-19.13.0.0.0-2.x86\_64.rpm

安装4个安装包

rpm -ivh oracle-instantclient19.13-basic-19.13.0.0.0-2.x86\_64.rpm

rpm -ivh oracle-instantclient19.13-devel-19.13.0.0.0-2.x86\_64.rpm

rpm -ivh oracle-instantclient19.13-odbc-19.13.0.0.0-2.x86\_64.rpm

rpm -ivh oracle-instantclient19.13-sqlplus-19.13.0.0.0-2.x86\_64.rpm

3）验证ODBC驱动动态链接库是否正确

ldd /usr/lib/oracle/19.13/client64/lib/libsqora.so.19.1

4）配置unixODBC，验证通过ODBC连接Oracle

vim /etc/odbcinst.ini

添加如下内容：

[Oracle 19 ODBC driver]

Description = Oracle ODBC driver for Oracle 19

Driver = /usr/lib/oracle/19.13/client64/lib/libsqora.so.19.1

vim /etc/odbc.ini

添加如下内容：

[oracle]

Driver = Oracle 19 ODBC driver ---名称是上面oracle部分用[]括起来的内容

ServerName =hadoop2:1521/orcl --oracle数据ip地址，端口及SID

UserID = atguigu

Password = 000000

验证

isql oracle

5）修改Doris的配置（每个BE节点都要，不用重启）

修改BE节点conf/odbcinst.ini文件,加入刚才/etc/odbcinst.ini添加的一样内容，并删除原先的Oracle配置。

[Oracle 19 ODBC driver]

Description = Oracle ODBC driver for Oracle 19

Driver = /usr/lib/oracle/19.13/client64/lib/libsqora.so.19.1

6）创建Resource

CREATE EXTERNAL RESOURCE `oracle\_19`

PROPERTIES (

"host" = "hadoop2",

"port" = "1521",

"user" = "atguigu",

"password" = "000000",

"database" = "orcl", --数据库示例名称，也就是ORACLE\_SID

"driver" = "Oracle 19 ODBC driver", --名称一定和be odbcinst.ini里的oracle部分的[]里的内容一样

"odbc\_type" = "oracle",

"type" = "odbc\_catalog"

);

7）基于Resource创建Doris外表

CREATE EXTERNAL TABLE `oracle\_odbc` (

id int,

name VARCHAR(20) NOT NULL

) ENGINE=ODBC

COMMENT "ODBC"

PROPERTIES (

"odbc\_catalog\_resource" = "oracle\_19",

"database" = "orcl",

"table" = "student"

);

8）查询Doris外表

select \* from oracle\_odbc;

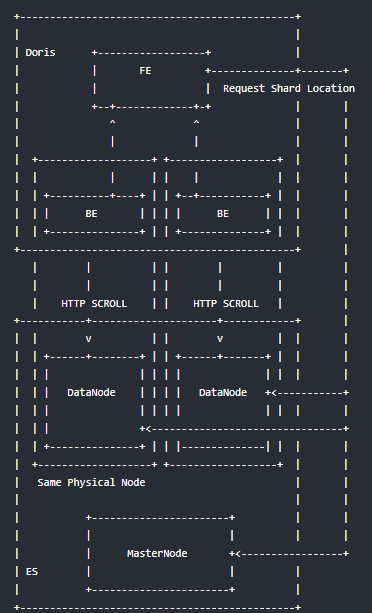
## 6.5 Doris On ES

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

### 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

### 6.5.2 使用方式

6.5.2.1 Doris中创建ES外表

1）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"

}

}

}

}

}

2）ES索引导入数据

POST /\_bulk

{"index":{"\_index":"test","\_type":"doc"}}

{ "k1" : 100, "k2": "2020-01-01", "k3": "Trying out Elasticsearch", "k4": "Trying out Elasticsearch", "k5": 10.0}

{"index":{"\_index":"test","\_type":"doc"}}

{ "k1" : 100, "k2": "2020-01-01", "k3": "Trying out Doris", "k4": "Trying out Doris", "k5": 10.0}

{"index":{"\_index":"test","\_type":"doc"}}

{ "k1" : 100, "k2": "2020-01-01", "k3": "Doris On ES", "k4": "Doris On ES", "k5": 10.0}

{"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}

3）Doris中创建ES外表

CREATE EXTERNAL TABLE `es\_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://hadoop1:9200,http://hadoop2:9200,http://hadoop3:9200",

"index" = "test",

"type" = "doc",

"user" = "",

"password" = ""

);

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，不指定的情况会使用\_doc |
| user | ES集群用户名 |
| password | 对应用户的密码信息 |

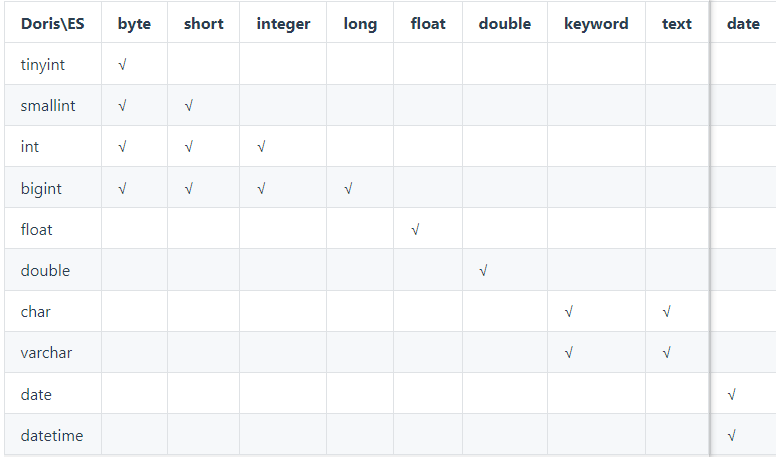
* ES 7.x之前的集群请注意在建表的时候选择正确的索引类型type
* 认证方式目前仅支持Http Basic认证，并且需要确保该用户有访问: /\_cluster/state/、\_nodes/http等路径和index的读权限; 集群未开启安全认证，用户名和密码不需要设置
* Doris表中的列名需要和ES中的字段名完全匹配，字段类型应该保持一致
* ENGINE必须是 Elasticsearch

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

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：



6.5.2.2 启用列式扫描优化查询速度

"enable\_docvalue\_scan" = "true"

**1）参数说明**

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：

（1）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

（2）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

**2）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。

**3）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。

6.5.2.3 探测keyword类型字段

"enable\_keyword\_sniff" = "true"

参数说明：

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。

6.5.2.4 开启节点自动发现,

"nodes\_discovery" = "true"

参数说明：

是否开启es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集群部署在与公共Internet隔离的内网，用户通过代理访问。

6.5.2.5 配置https访问模式

"http\_ssl\_enabled" = "true"

参数说明：

ES集群是否开启https访问模式。

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。

6.5.2.6 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。

1）基本查询

select \* from es\_table where k1 > 1000 and k3 ='term' or k4 like 'fu\*z\_'

2）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。

（1）match\_phrase查询：

select \* from es\_table where esquery(k4, '{

"match\_phrase": {

"k4": "doris on es"

}

}');

（2）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"

}

}

}');

（3）bool查询：

select \* from es\_table where esquery(k4, ' {

"bool": {

"must": [

{

"terms": {

"k1": [

11,

12

]

}

},

{

"terms": {

"k2": [

100

]

}

}

]

}

}');

### 6.5.3 最佳实践

6.5.3.1 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：

（1）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

（2）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。

6.5.3.2 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",

"type" = "doc"

}

注意:

（1）\_id字段的过滤条件仅支持=和in两种

（2）\_id字段只能是varchar类型

# 第7章 监控和报警

Doris可以使用Prometheus和Grafana进行监控和采集，官网下载最新版即可。

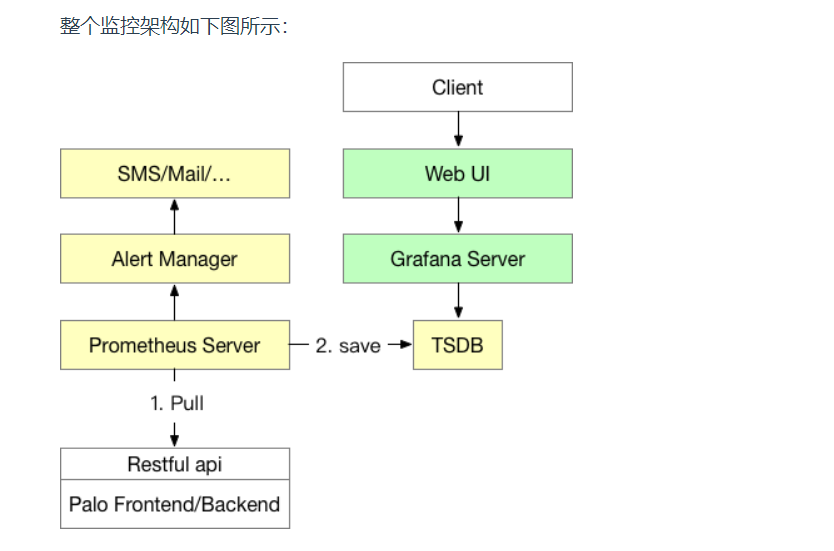
* Prometheus官网下载：<https://prometheus.io/download/>
* Grafana官网下载：<https://grafana.com/grafana/download>

Doris的监控数据通过FE和BE的http接口向外暴露。监控数据以key-value的文本形式对外展现。每个key还可能有不同的Label加以区分。当用户搭建好Doris后，可以在浏览器，通过以下接口访问监控数据.

Frontend: fe\_host:fe\_http\_port/metrics，如 http://hadoop1:8030/metrics

Backend: be\_host:be\_web\_server\_port/metrics，如 http://hadoop1:8040/metrics

整个监控架构如下图



## 7.1 [Prometheus](https://prometheus.io/)

（1）上传prometheus-2.26.0.linux-amd64.tar.gz，并进行解压

tar -zxvf prometheus-2.26.0.linux-amd64.tar.gz -C /opt/module

mv prometheus-2.26.0.linux-amd64 prometheus-2.26.0

（2）配置 promethues.yml

配置两个targets分别配置FE和BE,并且定义labels和groups指定组。如果有多个集群则再加 -job\_name标签,进行相同配置

vim /opt/module/prometheus-2.26.0/prometheus.yml

scrape\_configs:

- job\_name: 'prometheus\_doris'

static\_configs:

- targets: ['hadoop1:8030','hadoop2:8030','hadoop3:8030']

labels:

group: fe

- targets: ['hadoop1:8040','hadoop2:8040','hadoop3:8040']

labels:

group: be

（3）启动prometheus

nohup ./prometheus --web.listen-address="0.0.0.0:8181" &

该命令将后台运行 Prometheus，并指定其 web 端口为 8181。启动后，即开始采集数据，并将数据存放在 data 目录中。

（4）通过浏览器访问prometheus

http://hadoop1:8181

点击导航栏中，Status -> Targets，可以看到所有分组 Job 的监控主机节点。正常情况下，所有节点都应为 UP，表示数据采集正常。点击某一个 Endpoint，即可看到当前的监控数值。

（5）停止prometheus,直接kill -9即可

## 7.2 [Grafana](https://grafana.com/)

（1）上传grafana-7.5.2.linux-amd64.tar.gz，并进行解压

tar -zxvf grafana-7.5.2.linux-amd64.tar.gz -C /opt/module/

mv grafana-7.5.2.linux-amd64 grafana-7.5.2

（2）配置conf/defaults.ini

vim /opt/module/grafana-7.5.2/conf/defaults.ini

http\_addr = hadoop1

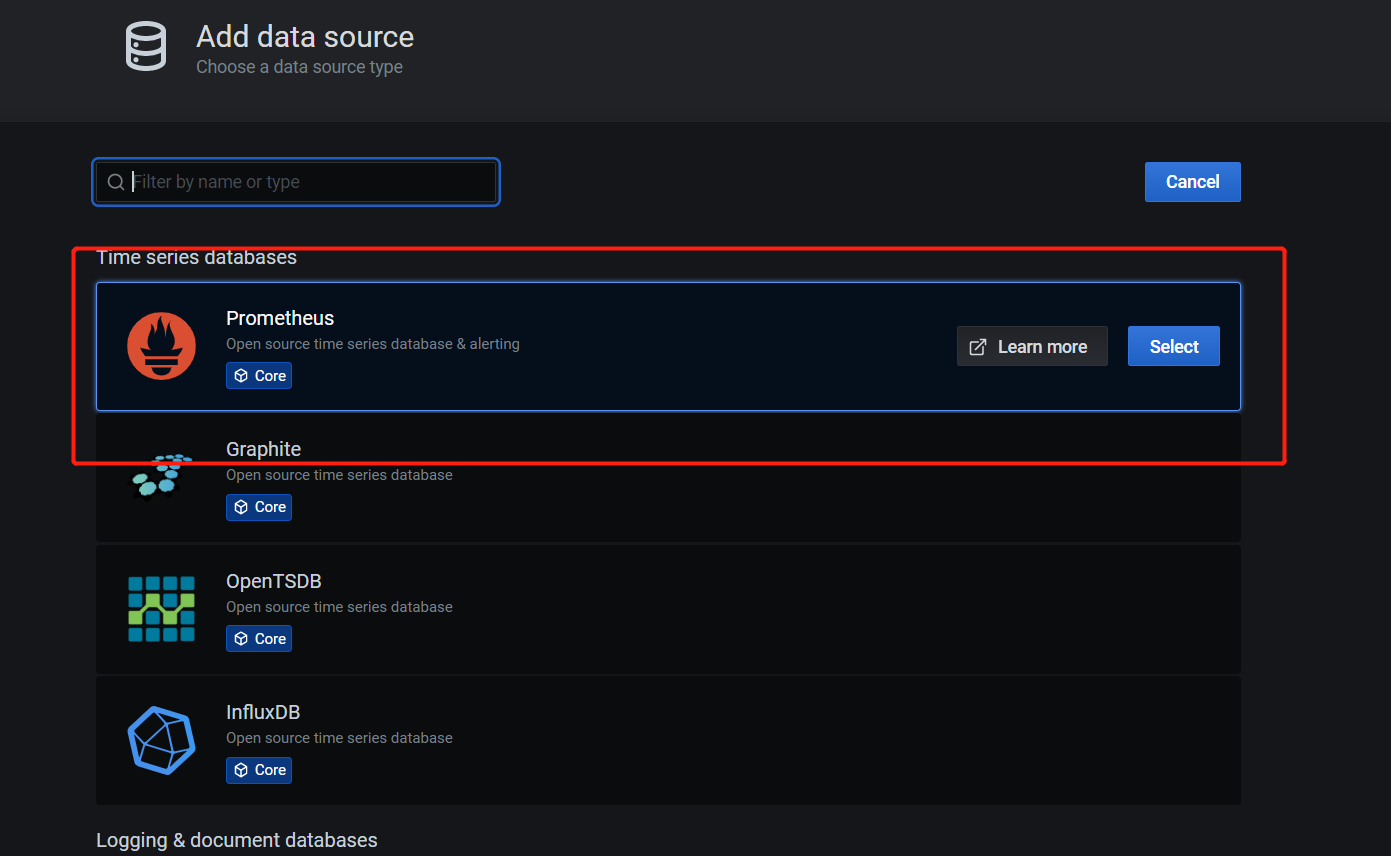
http\_port = 8182

（3）启动granafa

nohup /opt/module/grafana-7.5.2/bin/grafana-server &

（4）通过浏览器访问，配置数据源Prometheus

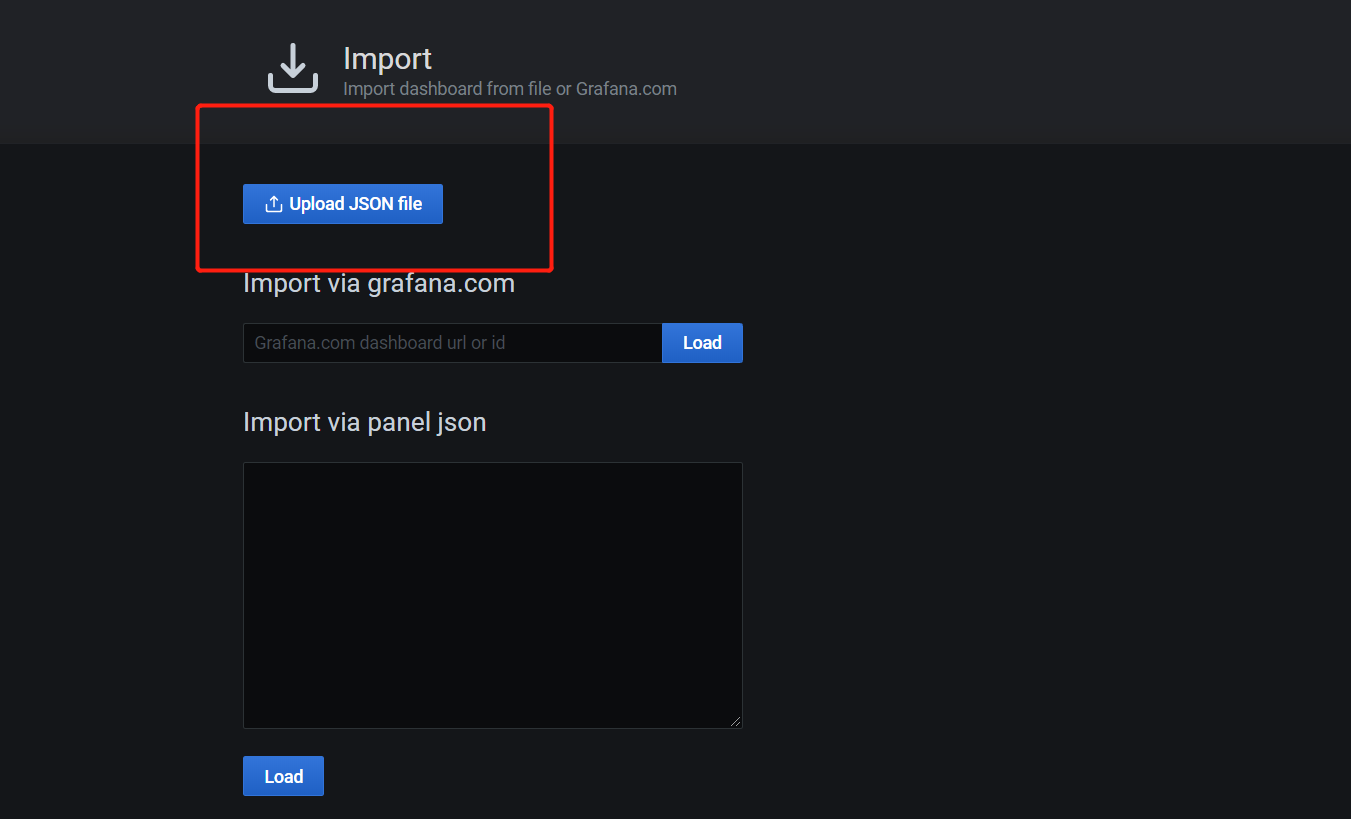
账号密码都是admin

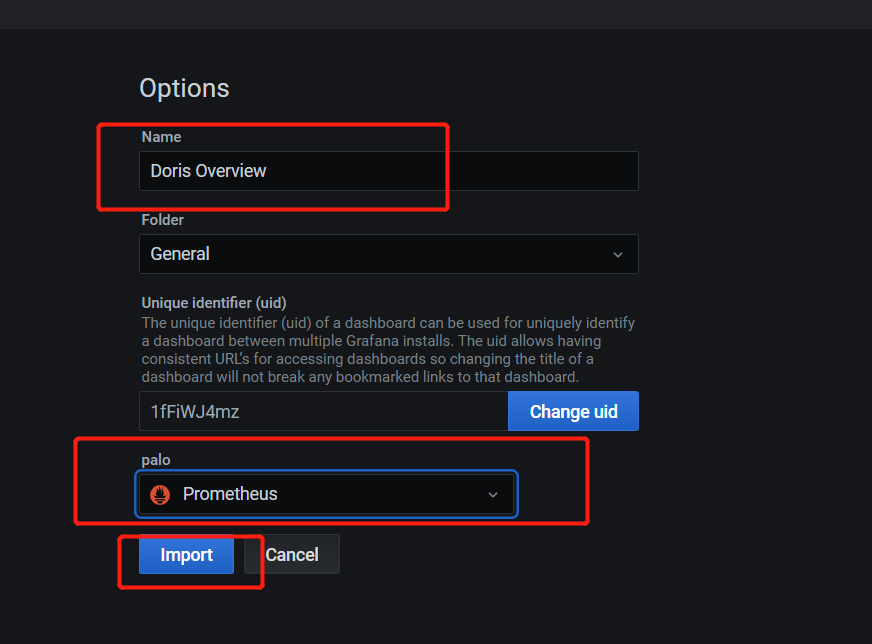


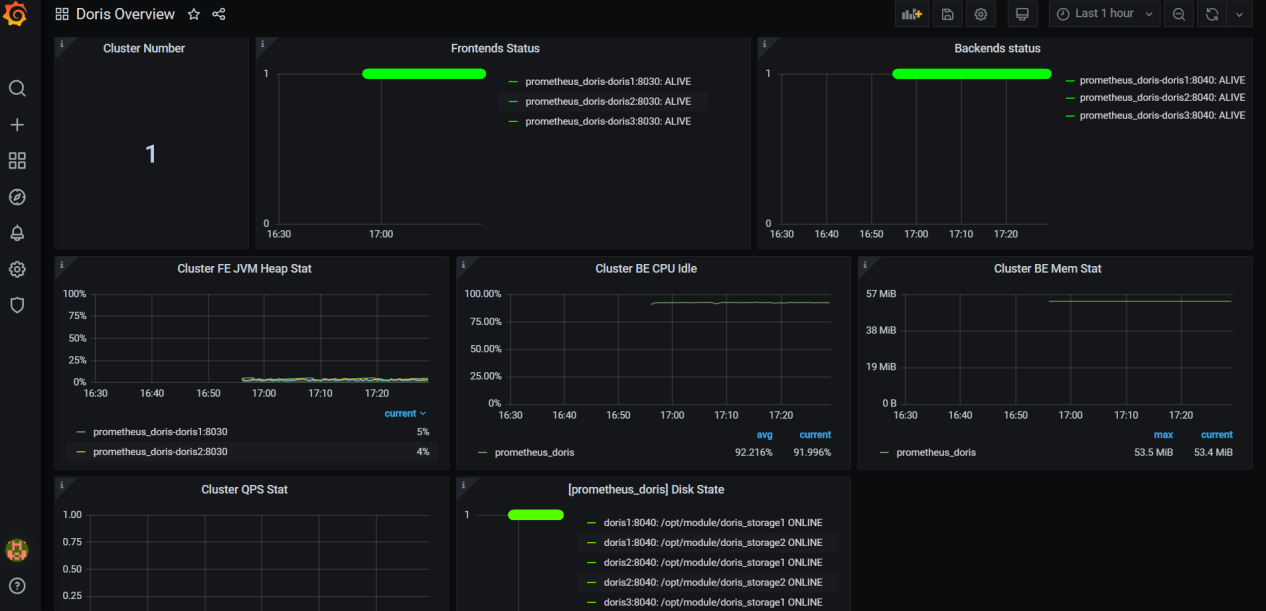
（5）添加dashboard

模板下载地址：https://grafana.com/grafana/dashboards/9734/revisions

上传准备好的doris-overview\_rev4.json







# 第8章 优化

## 8.1 查看QueryProfile

利用查询执行的统计结果，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。

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。

### 8.1.1 使用方式

1）开启profile：

set enable\_profile=true;

2）执行一个查询：

SELECT t1 FROM test JOIN test2 where test.t1 = test2.t2;

3）通过FE的UI查看：

http://hadoop1:8030/ QueryProfile/

### 8.1.2 参数说明

1）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 |

2）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 |

3）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 |

4）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 |

5）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 |

6）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 |

7）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 |

8）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 |

9）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 |

10）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 |

11）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 |

12）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。

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位整型，则数据量为 10 \* 4B = 40 Bytes。这个数据仅表示数据在内存中全展开的大小，并不代表实际的 IO 大小。

- 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 Scanner 过滤的行数。

- 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），对于IO受限的查询，接近磁盘的总吞吐量。

- ScannerBatchWaitTime: 426.886us # 用于统计transfer 线程等待scaner 线程返回rowbatch的时间。

- ScannerWorkerWaitTime: 17.745us # 用于统计scanner thread 等待线程池中可用工作线程的时间。

OlapScanner:

- BlockConvertTime: 8.941us # 将向量化Block转换为行结构的 RowBlock 的耗时。向量化 Block 在 V1 中为 VectorizedRowBatch，V2中为 RowBlockV2。

- BlockFetchTime: 468.974us # Rowset Reader 获取 Block 的时间。

- ReaderInitTime: 5.475ms # OlapScanner 初始化 Reader 的时间。V1 中包括组建 MergeHeap 的时间。V2 中包括生成各级 Iterator 并读取第一组Block的时间。

- RowsDelFiltered: 0 # 包括根据 Tablet 中存在的 Delete 信息过滤掉的行数，以及 unique key 模型下对被标记的删除行过滤的行数。

- RowsPushedCondFiltered: 0 # 根据传递下推的谓词过滤掉的条件，比如 Join 计算中从 BuildTable 传递给 ProbeTable 的条件。该数值不准确，因为如果过滤效果差，就不再过滤了。

- ScanTime: 39.24us # 从 ScanNode 返回给上层节点的时间。

- ShowHintsTime\_V1: 0ns # V2 中无意义。V1 中读取部分数据来进行 ScanRange 的切分。

SegmentIterator:

- BitmapIndexFilterTimer: 779ns # 利用 bitmap 索引过滤数据的耗时。

- BlockLoadTime: 415.925us # SegmentReader(V1) 或 SegmentIterator(V2) 获取 block 的时间。

- 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 Page 数量。

- CompressedBytesRead: 0.00 # V1 中，从文件中读取的解压前的数据大小。V2 中，读取到的没有命中 PageCache 的 Page 的压缩前的大小。

- 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询条件，完全过滤掉的 Segment 数量。

- 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，包含删除条件。V1 中还包含通过 BloomFilter 过滤掉的行数。

- 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（如果文件无需解压，则直接统计文件大小）。V2 中，仅统计未命中 PageCache 的 Page 解压后的大小（如果Page无需解压，直接统计Page大小）

- VectorPredEvalTime: 0ns # 向量化条件过滤操作的耗时。

- ShortPredEvalTime: 0ns # 短路谓词过滤操作的耗时。

- PredColumnReadTime: 0ns # 谓词列读取的耗时。

- LazyReadTime: 0ns # 非谓词列读取的耗时。

- OutputColumnTime: 0ns # 物化列的耗时。

13）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 |

### 8.1.3 调试方式

<https://doris.apache.org/zh-CN/developer-guide/debug-tool.html>

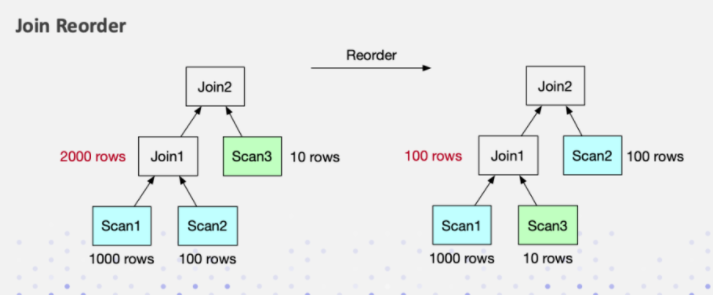
## 8.2 Join Reorder

Join Reorder 功能可以通过代价模型自动帮助调整 SQL 中 Join 的顺序，以帮助获得最优的 Join 效率。 可通过会话变量开启

set enable\_cost\_based\_join\_reorder=true

### 8.2.1 原理

数据库一旦涉及到多表 Join，Join 的顺序对整个 Join 查询的性能是影响很大的。假设有三张表 Join，参考下面这张图，左边是 a 表跟 b 张表先做 Join，中间结果的有 2000 行，然后与 c 表再进行 Join 计算。



接下来看右图，把 Join 的顺序调整了一下。把 a 表先与 c 表 Join，生成的中间结果只有 100，然后最终再与 b 表 Join 计算。最终的 Join 结果是一样的，但是它生成的中间结果有 20 倍的差距，这就会产生一个很大的性能 Diff 了。

Doris 目前支持基于规则的 Join Reorder 算法。它的逻辑是：

（1）让大表、跟小表尽量做 Join，它生成的中间结果是尽可能小的。

（2）把有条件的 Join 表往前放，也就是说尽量让有条件的 Join 表进行过滤

（3）Hash Join 的优先级高于 Nest Loop Join，因为 Hash join 本身是比 Nest Loop Join 快很多的。

### 8.2.2 示例

1）查看未开启Join Reorder的执行计划

explain graph

select \*

from example\_site\_visit

join example\_site\_visit2

on example\_site\_visit.user\_id=example\_site\_visit2.user\_id

join example\_site\_visit3

on example\_site\_visit.user\_id=example\_site\_visit3.user\_id;

2）开启Join Reorder

set enable\_cost\_based\_join\_reorder=true

3）查看开启Join Reorder后的执行计划

explain graph

select \*

from example\_site\_visit

join example\_site\_visit2

on example\_site\_visit.user\_id=example\_site\_visit2.user\_id

join example\_site\_visit3

on example\_site\_visit.user\_id=example\_site\_visit3.user\_id;

## 8.3 Join的优化原则

（1）在做 Join 的时候，要尽量选择同类型或者简单类型的列，同类型的话就减少它的数据 Cast，简单类型本身 Join 计算就很快。

（2）尽量选择 Key 列进行 Join， 原因前面在 Runtime Filter 的时候也介绍了，Key 列在延迟物化上能起到一个比较好的效果。

（3）大表之间的 Join ，尽量让它 Co-location ，因为大表之间的网络开销是很大的，如果需要去做 Shuffle 的话，代价是很高的。

（4）合理的使用 Runtime Filter，它在 Join 过滤率高的场景下效果是非常显著的。但是它并不是万灵药，而是有一定副作用的，所以需要根据具体的 SQL 的粒度做开关。

（5）涉及到多表 Join 的时候，需要去判断 Join 的合理性。尽量保证左表为大表，右表为小表，然后 Hash Join 会优于 Nest Loop Join。必要的时可以通过 SQL Rewrite，利用 Hint 去调整 Join 的顺序。



## 8.4 导入导出性能优化

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。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。

### 8.4.1 FE 配置

1）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  min\_load\_timeout\_second | 最大、最小导入超时时间，单位秒，默认最大3天,最小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最大个数，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。  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 | 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100。  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。 |

2）Broker相关的FE配置：

|  |  |
| --- | --- |
| min\_bytes\_per\_broker\_scanner | 单个BE处理的数据量的最小值，默认 64MB，单位bytes |
| max\_bytes\_per\_broker\_scanner | 单个BE处理的数据量的最大值，默认 3G，单位bytes |
| max\_broker\_concurrency | 作业的最大的导入并发数，默认10 |

* 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

3）Stream Load相关的FE配置：

|  |  |
| --- | --- |
| stream\_load\_default\_timeout\_second | 导入任务的超时时间(以秒为单位)，默认600 秒。也可以在 stream load 请求中设置单独的超时时间。 |

4）Export导出相关FE配置

|  |  |
| --- | --- |
| export\_checker\_interval\_second | 调度间隔，默认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。 |

### 8.4.2 BE 配置

1）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 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  load\_process\_max\_memory\_limit\_percent | 限制了单个 Backend 上，可用于导入任务的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。 |

2）Stream Load相关BE配置

|  |  |
| --- | --- |
| streaming\_load\_max\_mb | Stream load的最大导入大小，默认为10G，单位MB。如果用户的原始文件超过这个值，则需要调大。 |

### 8.4.3 性能分析

导入过程中的查询超时，建议先看监控，grafana 上的数据。比如是否导入占用了过多的 IO或者 cpu 等，导致了相互影响，再逐步根据 pprof （8.1.3提到的调试工具）+ 代码分析。

### 8.4.4 Broker导入大文件

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）修改 fe.conf 中配置

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。

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

比如一个 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的任务的。

2）合理设置timeout时间

在创建导入的时候自定义当前导入任务的 timeout 时间

单个BE处理数据量/最慢导入速度(MB/s)>=**timeout时间**>=单个BE处理数据量/10M/s

比如一个 100G 的文件，集群的 BE 个数为 10个

timeout >= 1000s = 10G / 10M/s

当计算出的 timeout 时间超过系统默认的导入最大超时时间 4小时，不推荐将导入最大超时时间直接改大来解决问题。最好是通过切分待导入文件并且分多次导入来解决问题。因为单次导入超过4小时的话，导入失败后重试的时间成本很高。

3）评估导入最大数据量

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

比如：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。

## 8.5 Bitmap 索引

用户可以通过创建bitmap index 加速查询

**1）创建索引**

语法：

CREATE INDEX [IF NOT EXISTS] index\_name ON table\_name (column [, ...],) [USING BITMAP] [COMMENT'balabala'];

注意： BITMAP 索引仅在单列上创建

示例：在table1 上为siteid 创建bitmap 索引

CREATE INDEX table\_bitmap ON table1 (siteid) USING BITMAP COMMENT 'table1\_bitmap\_index';

**2）查看索引**

语法：

SHOW INDEX[ES] FROM [db\_name.]table\_name [FROM database];

或者

SHOW KEY[S] FROM [db\_name.]table\_name [FROM database];

示例：展示table1索引

SHOW INDEX FROM test\_db.table1;

**3）删除索引**

语法：

DROP INDEX [IF EXISTS] index\_name ON [db\_name.]table\_name;

示例：

DROP INDEX IF EXISTS table\_bitmap ON test\_db.table1;

## 8.6 BloomFilter索引

Doris的BloomFilter索引是从通过建表的时候指定，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，则有可能在这个集合内。

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，用于在查询是快速过滤不满足条件的数据。

1）建表时指定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"

);

2）查看BloomFilter索引

SHOW CREATE TABLE sale\_detail\_bloom

3）修改BloomFilter索引

ALTER TABLE test\_db.sale\_detail\_bloom SET ("bloom\_filter\_columns" = "customer\_id,sku\_id");

4）删除BloomFilter索引

ALTER TABLE test\_db.sale\_detail\_bloom SET ("bloom\_filter\_columns" = "");

## 8.7 合理设置分桶分区数

（1）一个表的 Tablet 总数量等于 (Partition num \* Bucket num)。

（2）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。

（3）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）。

（4）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。

（5）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。

（6）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。比如当前只有 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。

（7）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。50GB：32个。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。

注：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。

# 第9章 数据备份及恢复

Doris 支持将当前数据以文件的形式，通过 broker 备份到远端存储系统中。之后可以通过 恢复 命令，从远端存储系统中将数据恢复到任意 Doris 集群。通过这个功能，Doris 可以支持将数据定期的进行快照备份。也可以通过这个功能，在不同集群间进行数据迁移。

该功能需要 Doris 版本 0.8.2+

使用该功能，需要部署对应远端存储的 broker。如 BOS、HDFS 等。可以通过 SHOW BROKER; 查看当前部署的 broker。

## 9.1 简要原理说明

### 9.1.1 备份（Backup）

备份操作是将指定表或分区的数据，直接以 Doris 存储的文件的形式，上传到远端仓库中进行存储。当用户提交 Backup 请求后，系统内部会做如下操作：

**1）快照及快照上传**

快照阶段会对指定的表或分区数据文件进行快照。之后，备份都是对快照进行操作。在快照之后，对表进行的更改、导入等操作都不再影响备份的结果。快照只是对当前数据文件产生一个硬链，耗时很少。快照完成后，会开始对这些快照文件进行逐一上传。快照上传由各个 Backend 并发完成。

**2）元数据准备及上传**

数据文件快照上传完成后，Frontend 会首先将对应元数据写成本地文件，然后通过 broker 将本地元数据文件上传到远端仓库。完成最终备份作业。

### 9.1.2 恢复（Restore）

恢复操作需要指定一个远端仓库中已存在的备份，然后将这个备份的内容恢复到本地集群中。当用户提交 Restore 请求后，系统内部会做如下操作：

**1）在本地创建对应的元数据**

这一步首先会在本地集群中，创建恢复对应的表分区等结构。创建完成后，该表可见，但是不可访问。

**2）本地snapshot**

这一步是将上一步创建的表做一个快照。这其实是一个空快照（因为刚创建的表是没有数据的），其目的主要是在 Backend 上产生对应的快照目录，用于之后接收从远端仓库下载的快照文件。

**3）下载快照**

远端仓库中的快照文件，会被下载到对应的上一步生成的快照目录中。这一步由各个 Backend 并发完成。

**4）生效快照**

快照下载完成后，我们要将各个快照映射为当前本地表的元数据。然后重新加载这些快照，使之生效，完成最终的恢复作业。

### 9.1.3 最佳实践

**1）备份**

当前我们支持最小分区（Partition）粒度的全量备份（增量备份有可能在未来版本支持）。如果需要对数据进行定期备份，首先需要在建表时，合理的规划表的分区及分桶，比如按时间进行分区。然后在之后的运行过程中，按照分区粒度进行定期的数据备份。

**2）数据迁移**

用户可以先将数据备份到远端仓库，再通过远端仓库将数据恢复到另一个集群，完成数据迁移。因为数据备份是通过快照的形式完成的，所以，在备份作业的快照阶段之后的新的导入数据，是不会备份的。因此，在快照完成后，到恢复作业完成这期间，在原集群上导入的数据，都需要在新集群上同样导入一遍。

建议在迁移完成后，对新旧两个集群并行导入一段时间。完成数据和业务正确性校验后，再将业务迁移到新的集群。

**3）重点说明**

（1）备份恢复相关的操作目前只允许拥有 ADMIN 权限的用户执行。

（2）一个 Database 内，只允许有一个正在执行的备份或恢复作业。

（3）备份和恢复都支持最小分区（Partition）级别的操作，当表的数据量很大时，建议按分区分别执行，以降低失败重试的代价。

（4）因为备份恢复操作，操作的都是实际的数据文件。所以当一个表的分片过多，或者一个分片有过多的小版本时，可能即使总数据量很小，依然需要备份或恢复很长时间。用户可以通过 SHOW PARTITIONS FROM table\_name; 和 SHOW TABLET FROM table\_name; 来查看各个分区的分片数量，以及各个分片的文件版本数量，来预估作业执行时间。文件数量对作业执行的时间影响非常大，所以建议在建表时，合理规划分区分桶，以避免过多的分片。

（5）当通过 SHOW BACKUP 或者 SHOW RESTORE 命令查看作业状态时。有可能会在 TaskErrMsg 一列中看到错误信息。但只要 State 列不为 CANCELLED，则说明作业依然在继续。这些 Task 有可能会重试成功。当然，有些 Task 错误，也会直接导致作业失败。

（6）如果恢复作业是一次覆盖操作（指定恢复数据到已经存在的表或分区中），那么从恢复作业的 COMMIT 阶段开始，当前集群上被覆盖的数据有可能不能再被还原。此时如果恢复作业失败或被取消，有可能造成之前的数据已损坏且无法访问。这种情况下，只能通过再次执行恢复操作，并等待作业完成。因此，我们建议，如无必要，尽量不要使用覆盖的方式恢复数据，除非确认当前数据已不再使用。

## 9.2 备份

### 9.2.1 创建一个远端仓库路径

CREATE REPOSITORY `hdfs\_ods\_dw\_backup`

WITH BROKER `broker\_name`

ON LOCATION "hdfs://hadoop1:8020/tmp/doris\_backup"

PROPERTIES (

"username" = "",

"password" = ""

)

### 9.2.2 执行备份

语法：

BACKUP SNAPSHOT [db\_name].{snapshot\_name}

TO `repository\_name`

ON (

`table\_name` [PARTITION (`p1`, ...)],

...

)

PROPERTIES ("key"="value", ...);

示例：

BACKUP SNAPSHOT test\_db.backup1

TO hdfs\_ods\_dw\_backup

ON

(

table1

);

### 9.2.3 查看备份任务

SHOW BACKUP [FROM db\_name]

### 9.2.4 查看远端仓库镜像

语法：

SHOW SNAPSHOT ON `repo\_name`

[WHERE SNAPSHOT = "snapshot" [AND TIMESTAMP = "backup\_timestamp"]];

示例一：查看仓库 hdfs\_ods\_dw\_backup 中已有的备份：

SHOW SNAPSHOT ON hdfs\_ods\_dw\_backup;

示例二：仅查看仓库 hdfs\_ods\_dw\_backup 中名称为 backup1 的备份：

SHOW SNAPSHOT ON hdfs\_ods\_dw\_backup WHERE SNAPSHOT = "backup1";

示例三：查看仓库 hdfs\_ods\_dw\_backup 中名称为 backup1 的备份，时间版本为 "2021-05-05-15-34-26" 的详细信息：

SHOW SNAPSHOT ON hdfs\_ods\_dw\_backup

WHERE SNAPSHOT = "backup1" AND TIMESTAMP = "2021-05-05-15-34-26";

### 9.2.5 取消备份

取消一个正在执行的备份作业语法：

CANCEL BACKUP FROM db\_name;

示例：取消 test\_db 下的 BACKUP 任务

CANCEL BACKUP FROM test\_db;

## 9.3 恢复

将之前通过 BACKUP 命令备份的数据，恢复到指定数据库下。该命令为异步操作。提交成功后，需通过 SHOW RESTORE 命令查看进度。

* 仅支持恢复 OLAP 类型的表
* 支持一次恢复多张表，这个需要和你对应的备份里的表一致

### 9.3.1 使用语法

RESTORE SNAPSHOT [db\_name].{snapshot\_name}

FROM `repository\_name`

ON (

`table\_name` [PARTITION (`p1`, ...)] [AS `tbl\_alias`],

...

)

PROPERTIES ("key"="value", ...);

说明：

（1）同一数据库下只能有一个正在执行的 BACKUP 或 RESTORE 任务。

（2）ON 子句中标识需要恢复的表和分区。如果不指定分区，则默认恢复该表的所有分区。所指定的表和分区必须已存在于仓库备份中

（3）可以通过 AS 语句将仓库中备份的表名恢复为新的表。但新表名不能已存在于数据库中。分区名称不能修改。

（4）可以将仓库中备份的表恢复替换数据库中已有的同名表，但须保证两张表的表结构完全一致。表结构包括：表名、列、分区、Rollup等等。

（5）可以指定恢复表的部分分区，系统会检查分区 Range 或者 List 是否能够匹配。

（6）PROPERTIES 目前支持以下属性：

"backup\_timestamp" = "2018-05-04-16-45-08"：指定了恢复对应备份的哪个时间版本，必填。该信息可以通过 SHOW SNAPSHOT ON repo; 语句获得。

"replication\_num" = "3"：指定恢复的表或分区的副本数。默认为3。若恢复已存在的表或分区，则副本数必须和已存在表或分区的副本数相同。同时，必须有足够的 host 容纳多个副本。

"timeout" = "3600"：任务超时时间，默认为一天。单位秒。

"meta\_version" = 40：使用指定的 meta\_version 来读取之前备份的元数据。注意，该参数作为临时方案，仅用于恢复老版本 Doris 备份的数据。最新版本的备份数据中已经包含 meta version，无需再指定。

### 9.3.2 使用示例

1）示例一

从 example\_repo 中恢复备份 snapshot\_1 中的表 backup\_tbl 到数据库 example\_db1，时间版本为 "2021-05-04-16-45-08"。恢复为 1 个副本：

RESTORE SNAPSHOT example\_db1.`snapshot\_1`

FROM `example\_repo`

ON ( `backup\_tbl` )

PROPERTIES

(

"backup\_timestamp"="2021-05-04-16-45-08",

"replication\_num" = "1"

);

2）示例二

从 example\_repo 中恢复备份 snapshot\_2 中的表 backup\_tbl 的分区 p1,p2，以及表 backup\_tbl2 到数据库 example\_db1，并重命名为 new\_tbl，时间版本为 "2021-05-04-17-11-01"。默认恢复为 3 个副本：

RESTORE SNAPSHOT example\_db1.`snapshot\_2`

FROM `example\_repo`

ON

(

`backup\_tbl` PARTITION (`p1`, `p2`),

`backup\_tbl2` AS `new\_tbl`

)

PROPERTIES

(

"backup\_timestamp"="2021-05-04-17-11-01"

);

3）演示

RESTORE SNAPSHOT test\_db.backup1

FROM `hdfs\_ods\_dw\_backup`

ON

(

table1 AS table\_restore

)

PROPERTIES

(

"backup\_timestamp"="2022-04-01-16-45-19"

);

### 9.3.3 查看恢复任务

可以通过下面的语句查看数据恢复的情况

SHOW RESTORE [FROM db\_name]

### 9.3.4 取消恢复

下面的语句用于取消一个正在执行数据恢复的作业：

CANCEL RESTORE FROM db\_name;

当取消处于 COMMIT 或之后阶段的恢复左右时，可能导致被恢复的表无法访问。此时只能通过再次执行恢复作业进行数据恢复

示例：取消 example\_db 下的 RESTORE 任务。

CANCEL RESTORE FROM example\_db;

## 9.4 删除远端仓库

该语句用于删除一个已创建的仓库。仅 root 或 superuser 用户可以删除仓库。这里的用户是指Doris的用户 语法：

DROP REPOSITORY `repo\_name`;

说明：

删除仓库，仅仅是删除该仓库在 Doris 中的映射，不会删除实际的仓库数据。删除后，可以再次通过指定相同的 broker 和 LOCATION 映射到该仓库。

示例：删除名为 hdfs\_ods\_dw\_backup 的仓库：

DROP REPOSITORY `hdfs\_ods\_dw\_backup`;

# 第10章 1.0 新特性

Doris 1.0开始官网提供了编译好的二进制包，可以直接下载使用。如果老版本想滚动升级新版本，可以参照官方说明：<https://doris.apache.org/zh-CN/installing/upgrade.html>

版本通告：<https://mp.weixin.qq.com/s/Ju3K67jOrBdJ8BX-V1IIgw>

## 10.1 向量化执行引擎

过去 Apache Doris 的 SQL 执行引擎是基于行式内存格式以及基于传统的火山模型进行设计的，在进行 SQL 算子与函数运算时存在非必要的开销，导致 Apache Doris 执行引擎的效率受限，并不适应现代 CPU 的体系结构。向量化执行引擎的目标是替换 Apache Doris 当前的行式 SQL 执行引擎，充分释放现代 CPU 的计算能力，突破在 SQL 执行引擎上的性能限制，发挥出极致的性能表现。

基于现代 CPU 的特点与火山模型的执行特点，向量化执行引擎重新设计了在列式存储系统的 SQL 执行引擎：

* 重新组织内存的数据结构，用 Column替换 Tuple，提高了计算时 Cache 亲和度，分支预测与预取内存的友好度
* 分批进行类型判断，在本次批次中都使用类型判断时确定的类型，将每一行类型判断的虚函数开销分摊到批量级别。
* 通过批级别的类型判断，消除了虚函数的调用，让编译器有函数内联以及 SIMD 优化的机会

从而大大提高了 CPU 在 SQL 执行时的效率，提升了 SQL 查询的性能。

<https://blog.csdn.net/qq_35423190/article/details/123129172>

<https://zhuanlan.zhihu.com/p/344706733>

### 10.1.1 使用方式

set enable\_vectorized\_engine = true;

set batch\_size = 4096;

batch\_size代表了SQL算子每次进行批量计算的行数。Doris默认的配置为1024,这个配置的行数会影响向量化执行引擎的性能与CPU缓存预取的行为。官方推荐配置为4096。

### 10.1.2 准备测试表

CREATE TABLE IF NOT EXISTS test\_db.user

(

`user\_id` LARGEINT NOT NULL COMMENT "用户id",

`username` VARCHAR(50) NOT NULL COMMENT "用户昵称",

`city` VARCHAR(20) NOT NULL COMMENT "用户所在城市",

`age` SMALLINT NOT NULL COMMENT "用户年龄",

`sex` TINYINT NOT NULL COMMENT "用户性别",

`phone` LARGEINT NOT NULL COMMENT "用户电话",

`address` VARCHAR(500) NOT NULL COMMENT "用户地址",

`register\_time` DATETIME NOT NULL COMMENT "用户注册时间"

)

UNIQUE KEY(`user\_id`, `username`)

DISTRIBUTED BY HASH(`user\_id`) BUCKETS 10

PROPERTIES("replication\_num" = "1");

insert into test\_db.user values\

(10000,'wuyanzu','北京',18,0,12345678910,'北京朝阳区','2017-10-01 07:00:00'),\

(20000,'wuyanzu','北京',19,0,12345678910,'北京朝阳区','2017-10-01 07:00:00'),\

(30000,'zhangsan','北京',20,0,12345678910,'北京海淀区','2017-11-15 06:10:20');

### 10.1.3 查看效果

explain select name from user where user\_id > 20000

开启了向量化执行引擎之后，在SQL的执行计划之中会在SQL算子前添加一个V的标识。

### 10.1.4 注意事项

1）NULL值

由于NULL值在向量化执行引擎中会导致性能劣化。所以在建表时，将对应的列设置为NULL通常会影响向量化执行引擎的性能。这里推荐使用一些特殊的列值表示NULL值，并在建表时设置列为NOT NULL以充分发挥向量化执行引擎的性能。

2）与行存执行引擎的部分差异

在绝大多数场景之中，用户只需要默认打开session变量的开关，就可以透明地使用向量化执行引擎，并且使SQL执行的性能得到提升。但是，目前的向量化执行引擎在下面一些微小的细节上与原先的行存执行引擎存在不同，需要使用者知晓。这部分区别分为两类

（1）a类 ：行存执行引擎需要被废弃和不推荐使用或依赖的功能

* Float与Double类型计算可能产生精度误差，仅影响小数点后5位之后的数字。如果对计算精度有特殊要求，请使用Decimal类型。
* DateTime类型不支持秒级别以下的计算或format等各种操作，向量化引擎会直接丢弃秒级别以下毫秒的计算结果。同时也不支持microseconds\_add等，对毫秒计算的函数。
* 有符号类型进行编码时，0与-0在SQL执行中被认为是相等的。这可能会影响distinct，group by等计算的结果。
* bitmap/hll 类型在向量化执行引擎中：输入均为NULL，则输出的结果为NULL而不是0。

（2）b类： 短期没有在向量化执行引擎上得到支持，但后续会得到开发支持的功能

* 不支持原有行存执行引擎的UDF与UDAF。
* string/text类型最大长度支持为1MB，而不是默认的2GB。即当开启向量化引擎后，将无法查询或导入大于1MB的字符串。但如果关闭向量化引擎，则依然可以正常查询和导入。
* 不支持 select ... into outfile 的导出方式。
* 不支持external broker外表。

## 10.2 Hive外表

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：

* 支持 Hive 数据源接入Doris
* 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

### 10.2.1 基本语法

CREATE [EXTERNAL] TABLE table\_name (

col\_name col\_type [NULL | NOT NULL] [COMMENT "comment"]

) ENGINE=**HIVE**

[COMMENT "comment"]

PROPERTIES (

'property\_name'='property\_value',

...

);

参数说明：

（1）外表列

* 列名要与 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。

（2）ENGINE 需要指定为 HIVE

（3）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

### 10.2.2 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。

### 10.2.3 使用示例

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

1）Hive中创建测试表：

CREATE TABLE `test11` (

`k1` int NOT NULL COMMENT "",

`k2` char(10) NOT NULL COMMENT "",

`k3` timestamp NOT NULL COMMENT "",

`k5` varchar(20) NOT NULL COMMENT "",

`k6` double NOT NULL COMMENT ""

)

insert into test11 values (1,'a',unix\_timestamp(),'haha',1.0);

2）Doris中创建外表

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://hadoop1:9083',

'database' = 'test',

'table' = 'test11'

);

3）查询外表

select \* from t\_hive;

## 10.3 Laterval view 语法

通过 Lateral View 语法，我们可以使用 explod\_bitmap 、explode\_split、explode\_jaon\_array 等 Table Function 表函数，将 bitmap、String 或 Json Array 由一列展开成多行，以便后续可以对展开的数据进行进一步处理（如 Filter、Join 等）。

1）创建测试表：

CREATE TABLE test3 (k1 INT,k2 varchar(30))

DISTRIBUTED BY HASH (k1) BUCKETS 2

PROPERTIES("replication\_num" = "1");

INSERT INTO test3 VALUES (1,''), (2,null), (3,','), (4,'1'),(5,'1,2,3'),(6,'a,b,c');

2）设置参数开启

set enable\_lateral\_view=true;

3）explode\_bitmap：展开一个bitmap类型

select k1, e1 from test3 lateral view explode\_bitmap(bitmap\_from\_string("1")) tmp1 as e1 order by k1, e1;

4）explode\_split：将一个字符串按指定的分隔符分割成多个子串

select k1, e1 from test3 lateral view explode\_split(k2, ',') tmp1 as e1 order by k1, e1;

5）explode\_json\_array：展开一个 json 数组

select k1, e1 from test3 lateral view explode\_json\_array\_int('[1,2,3]') tmp1 as e1 order by k1, e1;

select k1, e1 from test3 lateral view explode\_json\_array\_double('[1.0,2.0,3.0]') tmp1 as e1 order by k1, e1;

select k1, e1 from test3 lateral view explode\_json\_array\_string('[1,"b",3]') tmp1 as e1 order by k1, e1;

## 10.4 mysqldump导出

Doris 1.0支持通过mysqldump 工具导出数据或者表结构，下面几种操作：

1）导出 test 数据库中的 user 表：

mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces --databases test\_db --tables user > dump1.sql

2）导出 test\_db 数据库中的 user表结构：

mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces --databases test\_db --tables user --no-data > dump2.sql

3）导出 test\_db 数据库中所有表：

mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces --databases test\_db

4）导出所有数据库和表

mysqldump -h127.0.0.1 -P9030 -uroot --no-tablespaces --all-databases

5）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 /opt/module/doris-1.0.0/dump1.sql