尚硅谷大数据技术之Doris

（作者：尚硅谷研究院）

版本：V1.0

1. Doris简介
   1. Doris 概述

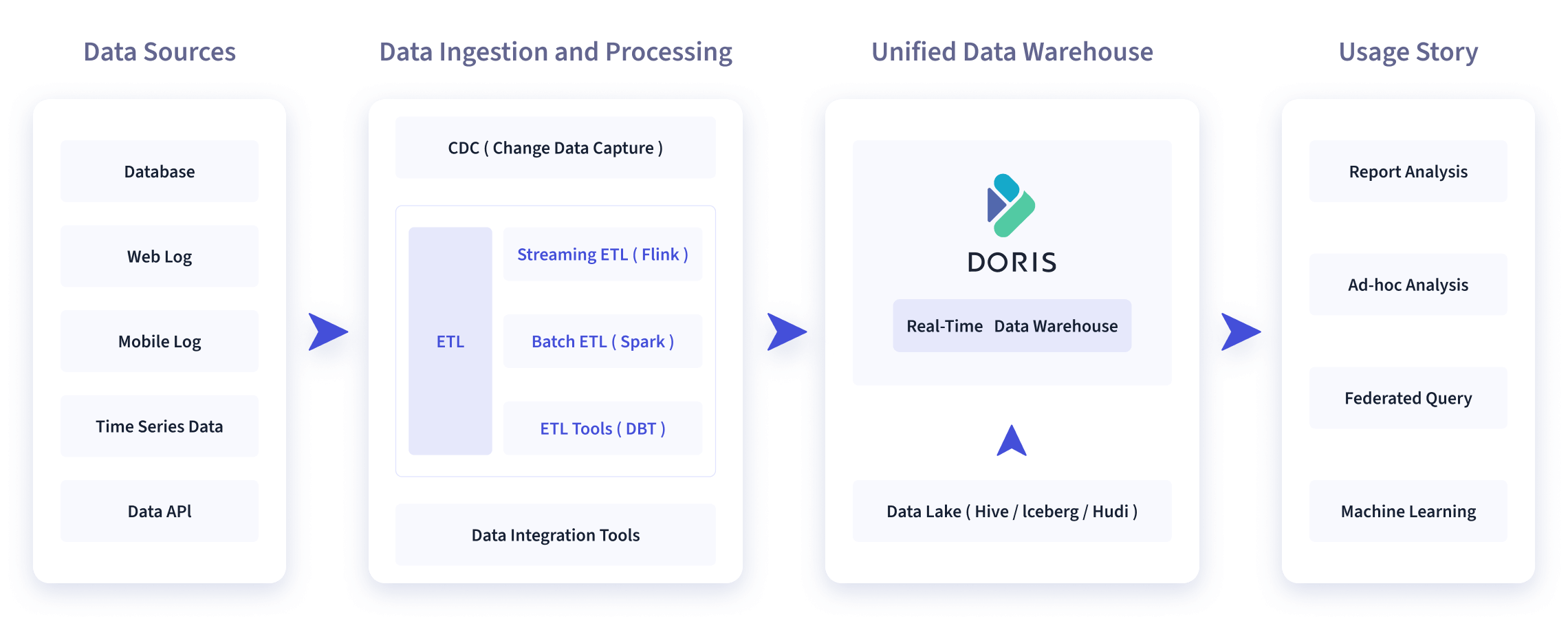
Apache Doris由百度大数据部研发（之前叫百度 Palo，2018年贡献到 Apache 社区后，更名为 Doris），在百度内部，有超过200个产品线在使用，部署机器超过1000台，单一业务最大可达到上百 TB。

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. Doris 使用场景

数据源经过各种数据集成和加工处理后，通常会入库到实时数仓 Doris 和离线湖仓（Hive，Iceberg，Hudi 中），Apache Doris 被广泛应用在以下场景中。



* + - * 1. 报表分析
* 实时看板 （Dashboards）
* 面向企业内部分析师和管理者的报表
* 面向用户或者客户的高并发报表分析（Customer Facing Analytics）。比如面向网站主的站点分析、面向广告主的广告报表，并发通常要求成千上万的QPS，查询延时要求毫秒级响应。著名的电商公司京东在广告报表中使用 Apache Doris，每天写入100亿行数据，查询并发 QPS上万，99分位的查询延时150ms。
  + - * 1. 即席查询（Ad-hoc Query）

面向分析师的自助分析，查询模式不固定，要求较高的吞吐。小米公司基于Doris构建了增长分析平台（Growing Analytics，GA），利用用户行为数据对业务进行增长分析，平均查询延时 10s，95分位的查询延时30s以内，每天的SQL查询量为数万条。

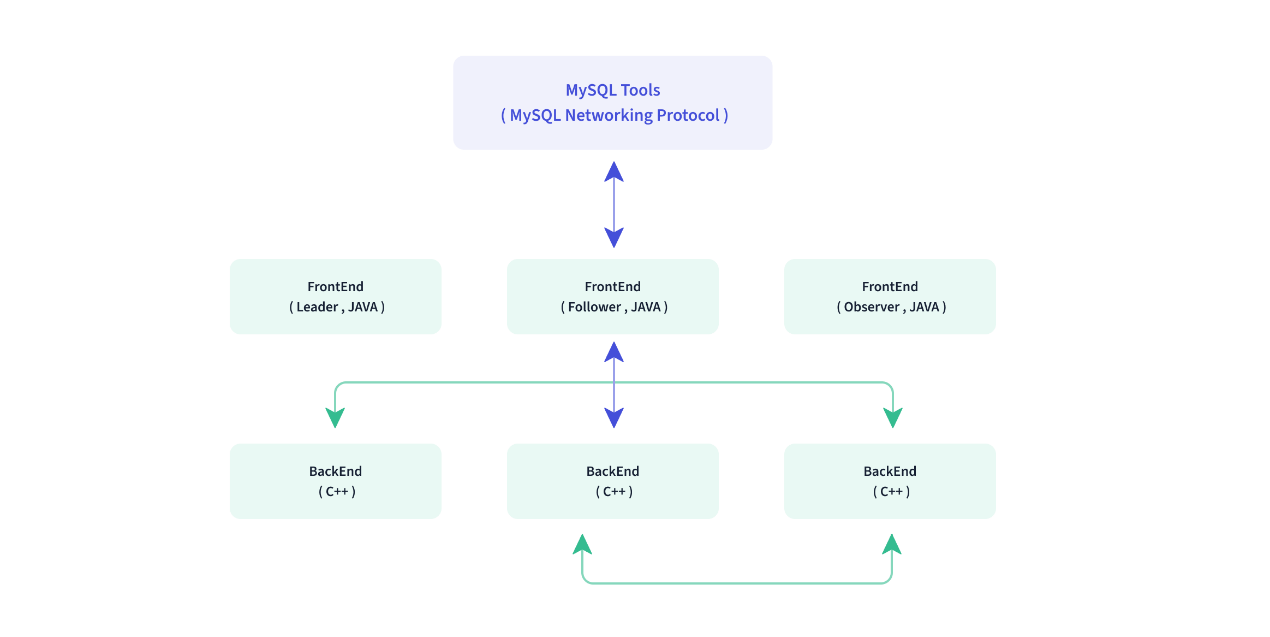
* + - * 1. 统一数仓构建

一个平台满足统一的数据仓库建设需求，简化繁琐的大数据软件栈。海底捞基于 Doris 构建的统一数仓，替换了原来由 Spark、Hive、Kudu、Hbase、Phoenix组成的旧架构，架构大大简化。

* + - * 1. 数据湖联邦查询

通过外表的方式联邦分析位于 Hive、Iceberg、Hudi中的数据，在避免数据拷贝的前提下，查询性能大幅提升。

* 1. Doris 架构



Doris**整体架构**如图所示，非常简单，只有两类进程：

* + - * 1. ****Frontend（FE）****

主要负责用户请求的接入、查询解析规划、元数据的管理、节点管理相关工作。

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。

* + - * 1. ****Backend（BE）****

主要负责数据存储、查询计划的执行。

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。

* + - * 1. ****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。

* + - * 1. ****Broker****

Broker是Doris集群中一种可选进程，主要用于支持Doris读写远端存储上的文件和目录，如 HDFS、BOS和AFS等。

1. Doris安装和部署
   1. 安装要求
      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 |

* + 1. 软件需求

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

* + 1. 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\* |

* + 1. 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\* |

* + 1. 注意事项

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。

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。

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）

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。

所有部署节点关闭Swap。

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）。

Follower的数量必须为奇数，Observer 数量随意。

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。

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。

* + 1. 内部端口

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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。

* 1. 集群部署

|  |  |  |
| --- | --- | --- |
| **hadoop102** | **hadoop103** | **hadoop104** |
| **FE（LEADER）** | FE（FOLLOWER） | FE（FOLLOWER） |
| **BE** | BE | BE |

生产环境建议FE和BE分开。

* + 1. 操作系统安装要求

设置系统最大打开文件句柄数（注意这里的\*不要去掉）。

sudo vim /etc/security/limits.conf

\* soft nofile 65536

\* hard nofile 65536

\* soft nproc 65536

\* hard nproc 65536

设置最大虚拟块的大小。

sudo vim /etc/sysctl.conf

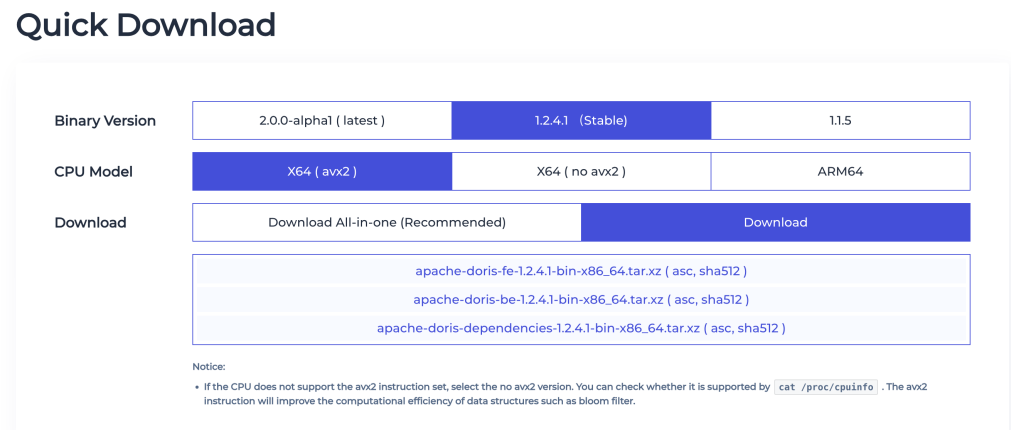
vm.max\_map\_count=2000000

重启生效。

* + 1. 下载安装包

根据自己的需要，下载合适的安装包。

<https://doris.apache.org/download/>



x86\_64架构 cpu（intel，amd)，执行命令：

cat /proc/cpuinfo | grep avx2

如果能看到avx2 字样选择带 avx2 的包，否则选择不带 avx2。

arm64 架构 cpu（apple），选择 arm64 的安装包下载。

* + 1. 解压

根据自己的CPU架构，选择合适的安装包解压（本文以 arm64 为例）

安装 fe

mkdir -p /opt/module/doris

tar -xvf apache-doris-fe-1.2.4.1-bin-arm.tar.xz -C /opt/module/doris

mv /opt/module/doris/apache-doris-fe-1.2.4.1-bin-arm /opt/module/doris/fe

安装 be

tar -xvf apache-doris-be-1.2.4.1-bin-arm.tar.xz -C /opt/module/doris

mv /opt/module/doris/apache-doris-be-1.2.4.1-bin-arm /opt/module/doris/be

安装其他依赖（java udf 函数）

tar -xvf apache-doris-dependencies-1.2.4.1-bin-arm.tar.xz -C /opt/module/doris

mv /opt/module/doris/apache-doris-dependencies-1.2.4.1-bin-arm /opt/module/doris/dependencies

cp /opt/module/doris/dependencies/java-udf-jar-with-dependencies.jar /opt/module/doris/be/lib

* + 1. 配置FE
       - 1. 修改FE配置文件

vim /opt/module/doris/fe/conf/fe.conf

# web 页面访问端口

http\_port = 7030

# 配置文件中指定元数据路径：默认在 fe 的根目录下，可以不配

# meta\_dir = /opt/module/doris/fe/doris-meta

# 修改绑定 ip

priority\_networks = 192.168.9.102/24

* 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。
  + - * 1. 启动FE

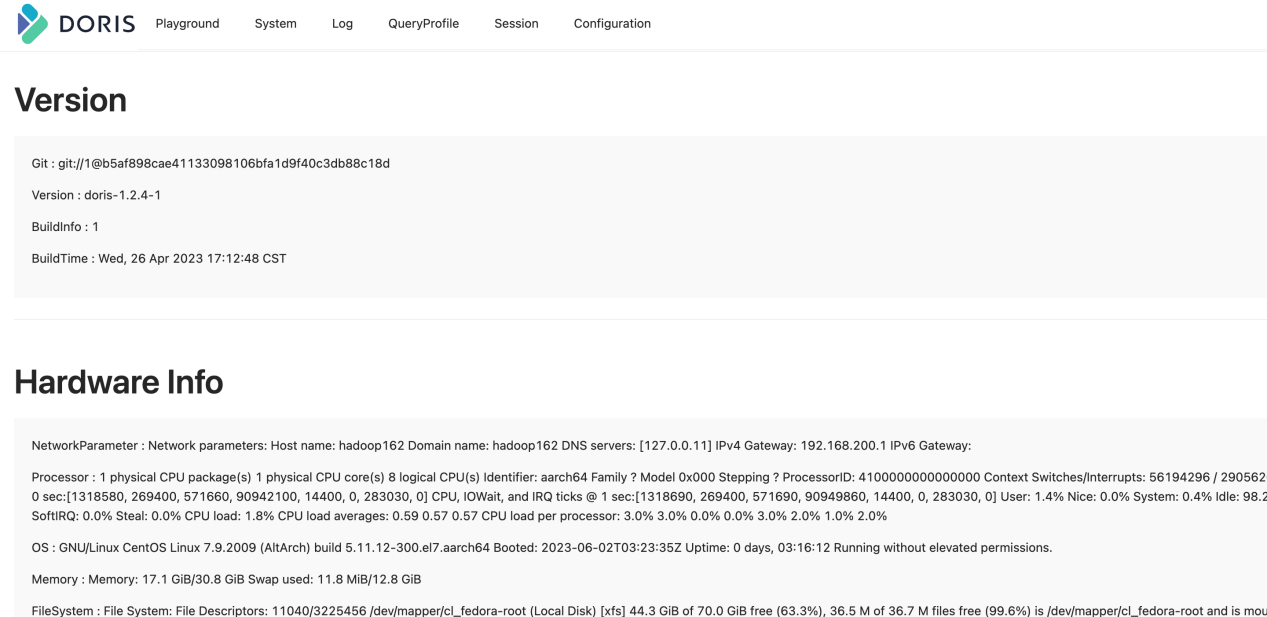
/opt/module/doris/fe/bin/start\_fe.sh --daemon

* + - * 1. 登录 FE Web页面

地址：<http://hadoop102:7030/login>

用户：root

密码：无



* + 1. 配置 BE
       - 1. 修改BE配置文件

vim /opt/module/doris/be/conf/be.conf

webserver\_port = 7040

# 不配置存储目录， 则会使用默认的存储目录

storage\_root\_path = /opt/module/doris/doris-storage1;/opt/module/doris/doris-storage2.SSD,10

priority\_networks = 192.168.9.102/24

mem\_limit=40%

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。

* 如果是 hdd，sdd 混合存储，则直接写目录即可。
* 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。
  + - * 1. 分发BE

xsync be

* + 1. 添加 BE

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。

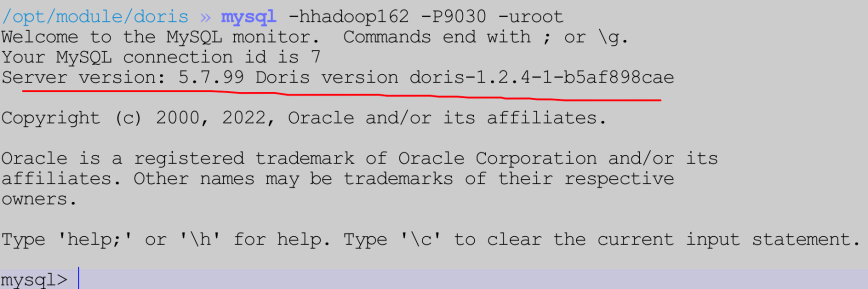
* + - * 1. 安装 Mysql 客户端

略

* + - * 1. 使用 Mysql 客户端连接到 FE

mysql -hhadoop102 -P9030 -uroot

* -P 指定端口（注意这里 P 是大写，小写 p 用来指定密码）
* FE 默认没有密码
* 设置密码：SET PASSWORD FOR 'root' = PASSWORD('aaaaaa');



* + - * 1. 添加 BE

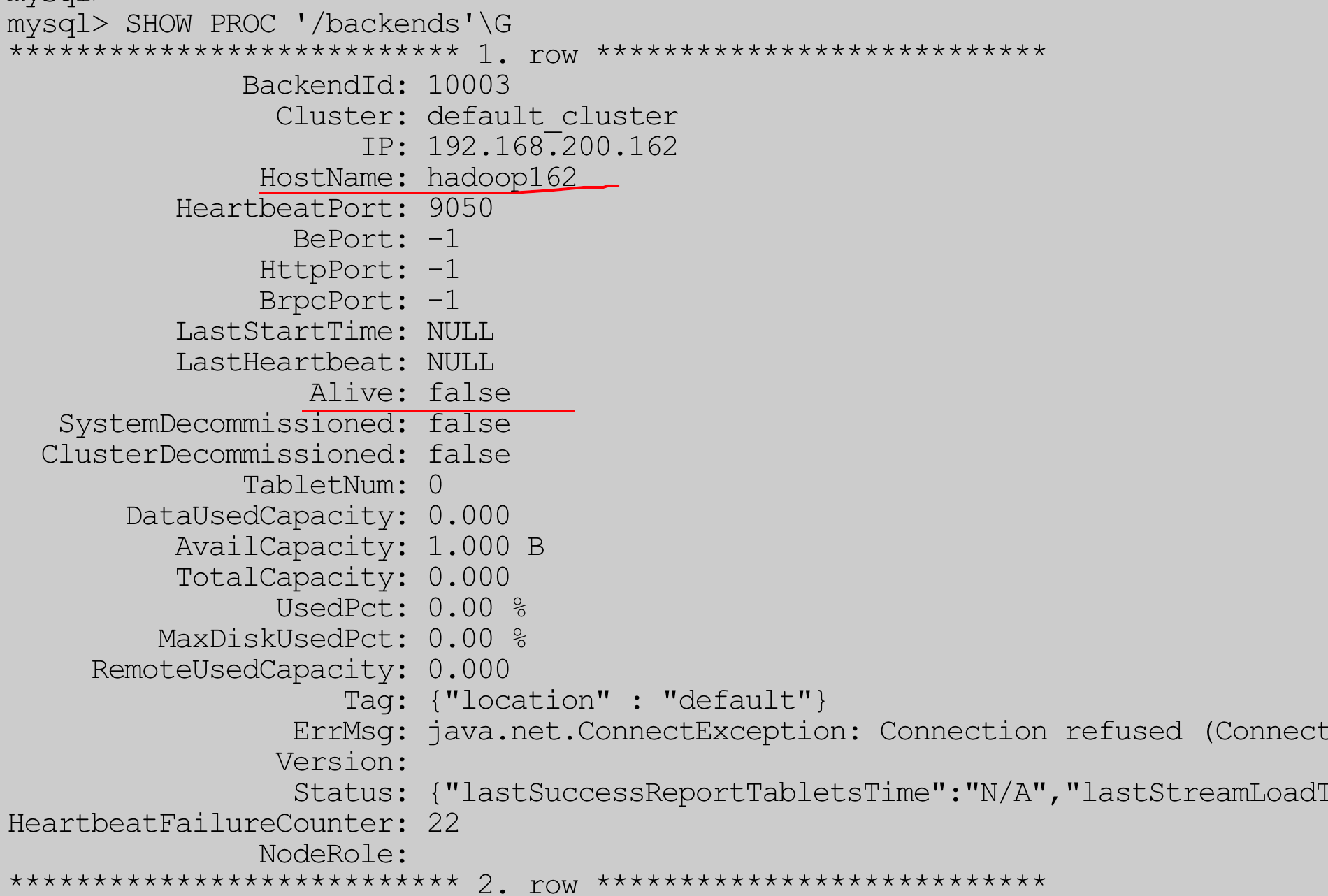
ALTER SYSTEM ADD BACKEND "hadoop102:9050";

ALTER SYSTEM ADD BACKEND "hadoop103:9050";

ALTER SYSTEM ADD BACKEND "hadoop104:9050";

* + - * 1. 查看 BE状态

SHOW PROC '/backends'\G



* + - * 1. 启动 BE

分别在三个节点执行如下命令。

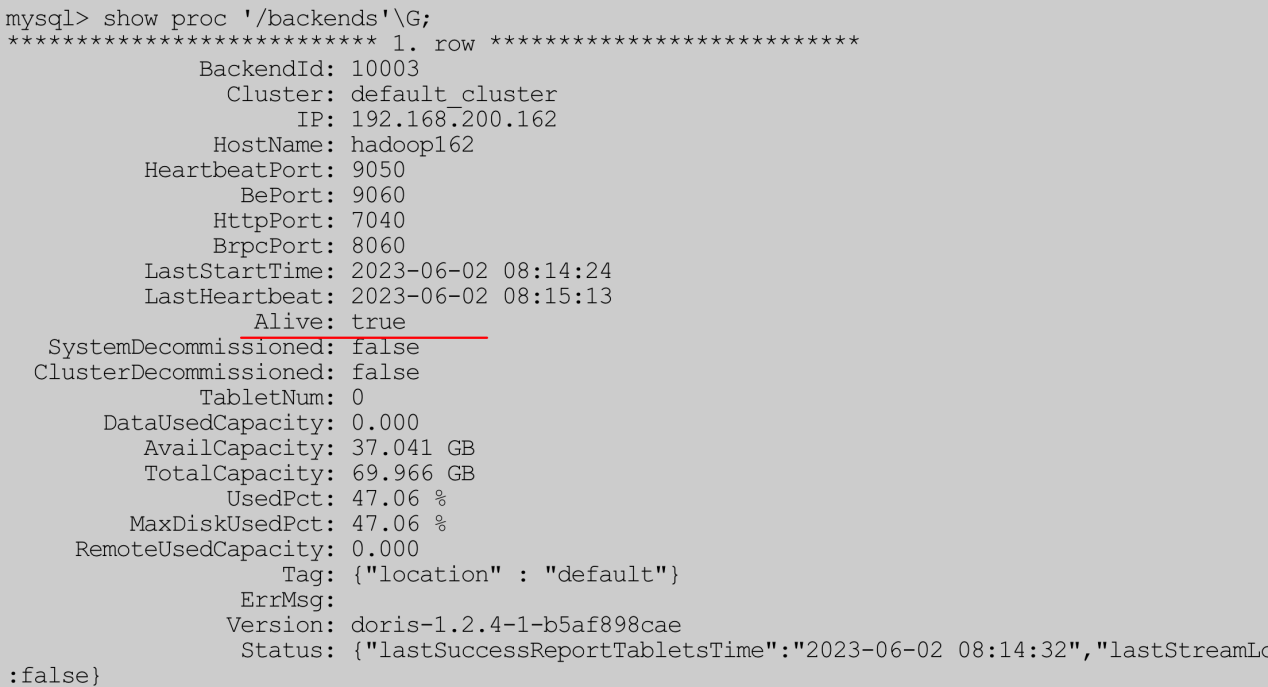
/opt/module/doris/be/bin/start\_be.sh --daemon

* + - * 1. 查询 BE 状态

mysql -h hadoop102 -P 9030 -uroot -paaaaaa

show proc '/backends';

Alive为true表示该BE节点存活。



* 1. 扩容和缩容
     1. FE 扩容和缩容

可以通过将FE扩容至3个以上节点（必须是奇数）来实现FE的高可用。

* + 1. 查看 FE 状态

mysql -h hadoop102 -uroot -P 9030 -paaaaaa

show proc '/frontends';

目前只有一个 FE。

* + - * 1. 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 OBSERVER "hadoop103:9010";

ALTER SYSTEM ADD OBSERVER "hadoop104:9010";

* + - * 1. 配置Follower和Observer

分发 FE

my\_rsync /opt/module/doris/fe

注意：需要去 hadoop103 和 hadoop104 删除 hadoop102 发过来的元数据。

rm -rf /opt/module/doris/fe/doris-meta/\*

在 hadoop103和 hadoop104 启动 FE

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

分别在hadoop103和 hadoop104 执行：

/opt/module/doris/fe/bin/start\_fe.sh --daemon --helper hadoop102:9010

在 mysql 客户端查看 FE 状态

show proc '/frontends';

* + - * 1. 删除 FE 节点（缩容）

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。

* + 1. 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。

* + - * 1. 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。

* + - * 1. 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。

* 1. Doris 集群群起脚本

#!/bin/bash

case $1 in

"start")

for host in hadoop102 hadoop103 hadoop104 ; do

echo "========== 在 $host 上启动 fe ========="

ssh $host "source /etc/profile; /opt/module/doris/fe/bin/start\_fe.sh --daemon"

done

for host in hadoop102 hadoop103 hadoop104 ; do

echo "========== 在 $host 上启动 be ========="

ssh $host "source /etc/profile; /opt/module/doris/be/bin/start\_be.sh --daemon"

done

;;

"stop")

for host in hadoop102 hadoop103 hadoop104 ; do

echo "========== 在 $host 上停止 fe ========="

ssh $host "source /etc/profile; /opt/module/doris/fe/bin/stop\_fe.sh "

done

for host in hadoop102 hadoop103 hadoop104 ; do

echo "========== 在 $host 上停止 be ========="

ssh $host "source /etc/profile; /opt/module/doris/be/bin/stop\_be.sh "

done

;;

\*)

echo "你启动的姿势不对"

echo " start 启动doris集群"

echo " stop 停止stop集群"

;;

esac

1. Doris数据表设计
   1. 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。

* + 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。

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。

* + 1. Tablet & Partition

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。

* 1. 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 |

* 1. 数据模型

Doris 的数据模型主要分为3类:

* Aggregate
* Unique
* Duplicate

准备数据库：

create database test\_db;

use test\_db;

* + 1. Aggregate 模型

假设业务有如下数据表模式。

|  |  |  |  |
| --- | --- | --- | --- |
| **ColumnName** | **Type** | **AggregationType** | **Comment** |
| **user\_id** | LARGEINT |  | 用户id |
| **date** | DATE |  | 数据灌入日期 |
| **city** | VARCHAR(20) |  | 用户所在城市 |
| **age** | SMALLINT |  | 用户年龄 |
| **sex** | TINYINT |  | 用户性别 |
| **last\_visit\_date** | DATETIME | REPLACE | 用户最后一次访问时间 |
| **cost** | BIGINT | SUM | 用户总消费 |
| **max\_dwell\_time** | INT | MAX | 用户最大停留时间 |
| **min\_dwell\_time** | INT | MIN | 用户最小停留时间 |

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

* 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",

`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

properties(

"replication\_num"="1"

);

* 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' ,20,10,10),\

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

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

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

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

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

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

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。

说明:

* 表中的REPLACE SUM MAX MIN 叫AggregationType （聚合类型），目前只有这四种聚合类型。
* 没有设置聚合类型的叫 key（维度列），设置了聚合类型的叫 value（指标列）。
* 当我们导入数据的时候, 会按照 key 对 value 使用他们自己的聚合类型进行聚合。
* 在同一个导入批次中的数据，对于 REPLACE 这种聚合方式，替换顺序不做保证。而对于不同导入批次中的数据，可以保证，后一批次的数据会替换前一批次。
* 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。
* 如果想要保留明细数据不让 doris 聚合, 则主要保证每条数据的 key 不一样就可以了。（多个key 中有一个不一样就行）
  + 1. Unique 模型

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

|  |  |  |  |
| --- | --- | --- | --- |
| **ColumnName** | **Type** | **IsKey** | **Comment** |
| **user\_id** | BIGINT | Yes | 用户id |
| **username** | VARCHAR(50) | Yes | 用户昵称 |
| **city** | VARCHAR(20) | No | 用户所在城市 |
| **age** | SMALLINT | No | 用户年龄 |
| **sex** | TINYINT | No | 用户性别 |
| **phone** | LARGEINT | No | 用户电话 |
| **address** | VARCHAR(500) | No | 用户住址 |
| **register\_time** | DATETIME | No | 用户注册时间 |

* 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

properties(

"replication\_num"="1"

)

* 插入语句

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',19,1,12345678910,'北京朝阳区','2017-10-01 08:00:00'),\

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

这种表结构完全等价于下面的聚合模型。

|  |  |  |  |
| --- | --- | --- | --- |
| **ColumnName** | **Type** | **AggregationType** | **Comment** |
| **user\_id** | BIGINT |  | 用户id |
| **username** | VARCHAR(50) |  | 用户昵称 |
| **city** | VARCHAR(20) | REPLACE | 用户所在城市 |
| **age** | SMALLINT | REPLACE | 用户年龄 |
| **sex** | TINYINT | REPLACE | 用户性别 |
| **phone** | LARGEINT | REPLACE | 用户电话 |
| **address** | VARCHAR(500) | REPLACE | 用户住址 |
| **register\_time** | DATETIME | REPLACE | 用户注册时间 |

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) REPLACE COMMENT "用户所在城市",

`age` SMALLINT REPLACE COMMENT "用户年龄",

`sex` TINYINT REPLACE COMMENT "用户性别",

`phone` LARGEINT REPLACE COMMENT "用户电话",

`address` VARCHAR(500) REPLACE COMMENT "用户地址",

`register\_time` DATETIME REPLACE COMMENT "用户注册时间"

)

AGGREGATE KEY(`user\_id`, `username`)

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

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

* + 1. Duplicate 模型

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

|  |  |  |  |
| --- | --- | --- | --- |
| **ColumnName** | **Type** | **SortKey** | **Comment** |
| **timestamp** | DATETIME | Yes | 日志时间 |
| **type** | INT | Yes | 日志类型 |
| **error\_code** | INT | Yes | 错误码 |
| **error\_msg** | VARCHAR(1024) | No | 错误详细信息 |
| **op\_id** | BIGINT | No | 负责人id |
| **op\_time** | DATETIME | No | 处理时间 |

* 建表

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

properties(

"replication\_num"="1"

);

* 插入语句

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');

这种数据模型区别于 Aggregate 和 Unique 模型。数据完全按照导入文件中的数据进行存储，不会有任何聚合。即使两行数据完全相同，也都会保留。而在建表语句中指定的 DUPLICATE KEY，只是用来指明底层数据按照那些列进行排序。

在 DUPLICATE KEY 的选择上，我们建议适当的选择前 2-4 列就可以。

这种数据模型适用于既没有聚合需求，又没有主键唯一性约束的原始数据的存储。

* 1. 建表示例

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|es]]

[key\_desc]

[COMMENT "table comment"];

[partition\_desc]

[distribution\_desc]

[rollup\_index]

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

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

Range Partition

CREATE TABLE IF NOT EXISTS test\_db.example\_range\_tbl

(

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

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

`city` VARCHAR(20) COMMENT "用户所在城市",

`age` SMALLINT COMMENT "用户年龄",

`sex` TINYINT COMMENT "用户性别",

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

`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`, `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" = "1",

"storage\_cooldown\_time" = "2024-01-01 12:00:00"

);

List partition

CREATE TABLE IF NOT EXISTS test\_db.example\_list\_tbl

(

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

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

`city` VARCHAR(20) NOT NULL COMMENT "用户所在城市",

`age` SMALLINT COMMENT "用户年龄",

`sex` TINYINT COMMENT "用户性别",

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

`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`, `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" = "1",

"storage\_cooldown\_time" = "2024-01-01 12:00:00"

);

* 1. 数据划分
     1. 列定义

这里我们只以 AGGREGATE KEY 数据模型为例进行说明。

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。
  + 1. ENGINE(引擎)

Doris 支持的引擎有：olap|mysql|broker|hive

olap 是默认的引擎, 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、hive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。

* + 1. 分区和分桶

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**。

* + - * 1. 分区（partiton）
* Partition列可以指定一列或多列。分区列必须为KEY列。多列分区的使用方式在后面介绍。
* 不论分区列是什么类型，在写分区值时，都需要加**双引号**。
* 分区数量理论上没有上限。
* 当不使用Partition建表时，系统会自动生成一个和表名同名的，全值范围的 Partition。该Partition对用户不可见，并且不可删改。
* 创建分区时不可添加范围重叠的分区。
  + - * 1. Rang 分区（范围分区）

分区列通常为**时间列**（ PARTITION BY RANGE(`date`) ），以方便的管理新旧数据

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。

VALUES [...) 指定上下界，生成一个左闭右开的区间。

通过 VALUES [...) 同时指定上下界比较容易理解。这里举例说明，当使用VALUES LESS THAN (...) 语句进行分区的增删操作时，分区范围的变化情况：

* 如上 example\_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)

查看一个表的所有分区信息

show partitions from example\_range\_tbl;

插入数据：

进入 p201701 分区

insert into test\_db.example\_range\_tbl values (10000,'2017-01-01','北京',20,0,'2017-01-01 06:00:00',20,10,10);

没有对应的分区, 插入不成功,抛出异常

insert into test\_db.example\_range\_tbl values (20000,'2017-11-01','北京',20,0,'2017-11-01 06:00:00',20,10,10);

* 当我们增加一个分区 p201705 VALUES LESS THAN ("2017-06-01")，分区结果如下。

新增分区语法:

alter table example\_range\_tbl add partition p201705 values less than ('2017-06-01');

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

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

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

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

* 此时我们删除分区 p201703，则分区结果如下：

删除分区语法:

alter table example\_range\_tbl drop partition p201703;

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

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

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

需要注意的是，这**时其他分区并不会发生变化**, p201702和p201705之间就出现了一个**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。

* 综上，**分区的删除不会改变已存在分区的范围**。删除分区可能出现空洞。通过 VALUES LESS THAN 语句增加分区时，**分区的下界紧接上一个分区的上界**。
  + - * 1. List 分区（列表分区）
* 分区列支持 BOOLEAN, TINYINT, SMALLINT, INT, BIGINT, LARGEINT, DATE, DATETIME, CHAR, VARCHAR 数据类型，分区值为枚举值。**只有当数据为目标分区枚举值其中之一时，才可以命中分区**
* Partition 支持通过 VALUES IN (...) 来指定每个分区包含的枚举值。
* 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")

进入 p\_cn 分区

insert into test\_db.example\_list\_tbl values (10000,'2017-01-01','Beijing',20,0,'2017-01-01 06:00:00',20,10,10);

插入不成功, 不进入任何分区，且报错

insert into test\_db.example\_list\_tbl values (20000,'2017-01-01','shenzhen',20,0,'2017-01-01 06:00:00',20,10,10);

* 当我们增加一个分区 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")

* 当我们删除分区 p\_jp，分区结果如下：

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

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

p\_uk: ("London")

* + - * 1. 分桶(Bucket)

DISTRIBUTED BY HASH(`user\_id`) BUCKETS 16

* 如果**使用了 Partition**，则 DISTRIBUTED ... 语句描述的是数据在**各个分区内**的划分规则。如果**不使用 Partition**，则描述的是对**整个表**的数据的划分规则。
* 分桶列可以是多列，但**必须为 Key** 列。分桶列可以和 Partition 列相同或不同。
* 分桶的数量理论上没有上限。
* 分桶列的选择，是在**查询吞吐**和**查询并发**之间的一种权衡：

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。

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。

* + - * 1. 复合分区与单分区

**复合分区**：分区和分桶

**单分区：**只分桶（其实是所有数据在一个分区，数据只做 hash 分布）

**以下场景推荐使用复合分区**

* 有时间维度或类似带有有序值的维度，可以以这类维度列作为分区列。分区粒度可以根据导入频次、分区数据量等进行评估。
* 历史数据删除需求：如有删除历史数据的需求（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。
* 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。
  + - * 1. 多列分区

Doris 支持指定多列作为分区列

* 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，**当第一列处于边界的时候, 由第二列决定**，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

* 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

* + 1. PROPERTIES
* replication\_num

副本数。默认副本数为3。如果 BE 节点数量小于3，则需指定副本数小于等于 BE 节点数量。

* storage\_medium/storage\_cooldown\_time

storage\_medium 用于声明表数据的初始存储介质，而 storage\_cooldown\_time 用于设定到期时间。

"storage\_medium" = "SSD",  
"storage\_cooldown\_time" = "2020-11-20 00:00:00"

这个示例表示数据存放在 SSD 中，并且在 2020-11-20 00:00:00 到期后，会自动迁移到 HDD 存储上。

* 1. 动态分区

前面的分区, 必须在插入数据之前手动添加需要的分区, 使用颇多不变.

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**。

* + 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。

* + 1. 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。

* + - * 1. 建表时指定

CREATE TABLE tbl1

(...)

PROPERTIES

(

"dynamic\_partition.prop1" = "value1",

"dynamic\_partition.prop2" = "value2",

...

)

* + - * 1. 运行时修改

ALTER TABLE tbl1 SET

(

"dynamic\_partition.prop1" = "value1",

"dynamic\_partition.prop2" = "value2",

...

)

* + 1. 动态分区规则主要参数

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**  **如果为 FALSE，则 Doris 会忽略该表的动态分区规则。** |
| **dynamic\_partition.time\_unit** | 动态分区调度的单位，可指定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 |
| **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.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**额外**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", \  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``。 |

* + 1. 创建历史分区规则

当 create\_history\_partition 为 true，即开启创建历史分区功能时，Doris 会根据 dynamic\_partition.start 和 dynamic\_partition.history\_partition\_num 来决定创建历史分区的个数。

假设需要创建的历史分区数量为 expect\_create\_partition\_num，根据不同的设置具体数量如下：

* create\_history\_partition = true

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

则expect\_create\_partition\_num = end - start + 1；

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

则expect\_create\_partition\_num = end - max(start, -histoty\_partition\_num) + 1;

* create\_history\_partition = false

不会创建历史分区，expect\_create\_partition\_num = end - 0 + 1;

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

总结：今天的分区（1） + end（未来的分区） + 过去的分区（start 和 history-num 谁少听谁的）

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

* + 1. 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。

* + 1. 动态分区创建示例
       - 1. 创建动态分区表

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`)

PROPERTIES(

"dynamic\_partition.enable" = "true",

"dynamic\_partition.time\_unit" = "DAY",

"dynamic\_partition.create\_history\_partition" = "true",

"dynamic\_partition.history\_partition\_num" = "3",

"dynamic\_partition.start" = "-7",

"dynamic\_partition.end" = "3",

"dynamic\_partition.prefix" = "p",

"dynamic\_partition.buckets" = "10",

"replication\_num" = "1"

);

* + - * 1. 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
  + - * 1. 查看表的分区情况

SHOW PARTITIONS FROM student\_dynamic\_partition1\G

* + - * 1. 插入测试数据（需要修改日期）

insert into student\_dynamic\_partition1 values(1,'2023-07-11 11:00:00','name1',18);

insert into student\_dynamic\_partition1 values(1,'2023-07-12 11:00:00','name1',18);

insert into student\_dynamic\_partition1 values(1,'2023-07-13 11:00:00','name1',18);

* + - * 1. 动态分区表与手动分区表相互转换

对于一个表来说，动态分区和手动分区可以自由转换，但二者不能同时存在，有且只有一种状态。

手动分区转换为动态分区

如果一个表在创建时未指定动态分区，可以通过ALTER TABLE在运行时修改动态分区相关属性来转化为动态分区，具体示例可以通过HELP ALTER TABLE查看。

注意：如果已设定dynamic\_partition.start，分区范围在动态分区起始偏移之前的历史分区将会被删除。

动态分区转换为手动分区

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。

* 1. Rollup（上卷）

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（**从细粒度到粗粒度**）。

* + 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。

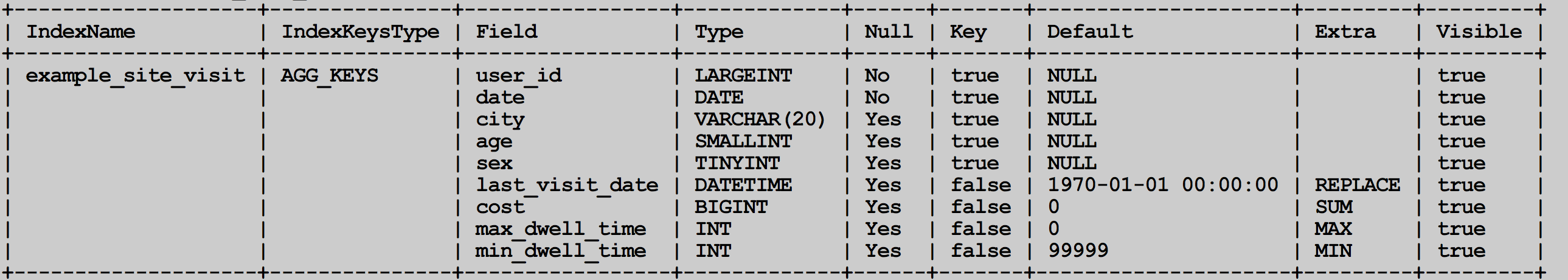
* + 1. Aggregate 和 Unique 模型中的 ROLLUP

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

以[3.3.1](#_Aggregate_模型)中创建的example\_site\_visit表为例。

* + - * 1. 查看表结构信息

desc example\_site\_visit all;



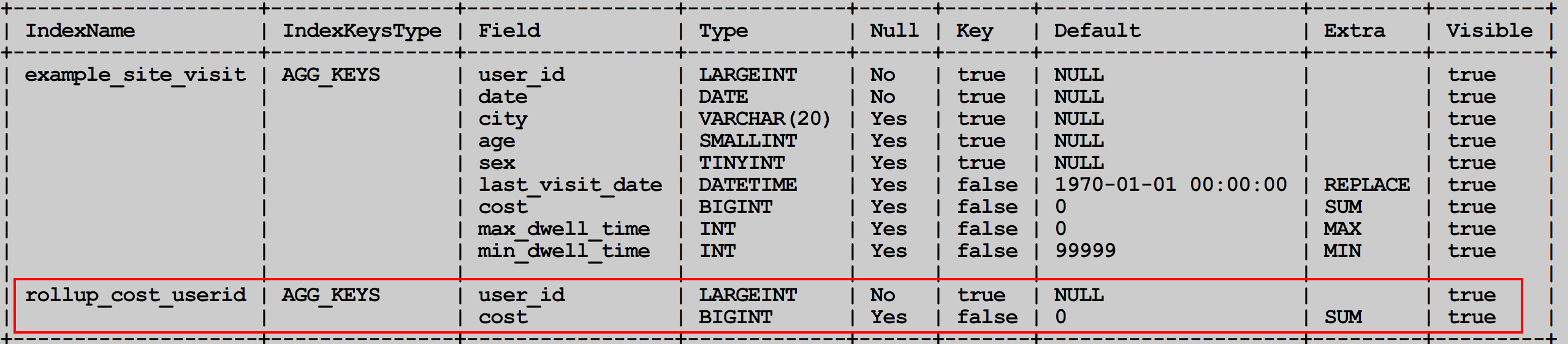
* + - * 1. 创建 rollup

示例 1

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。

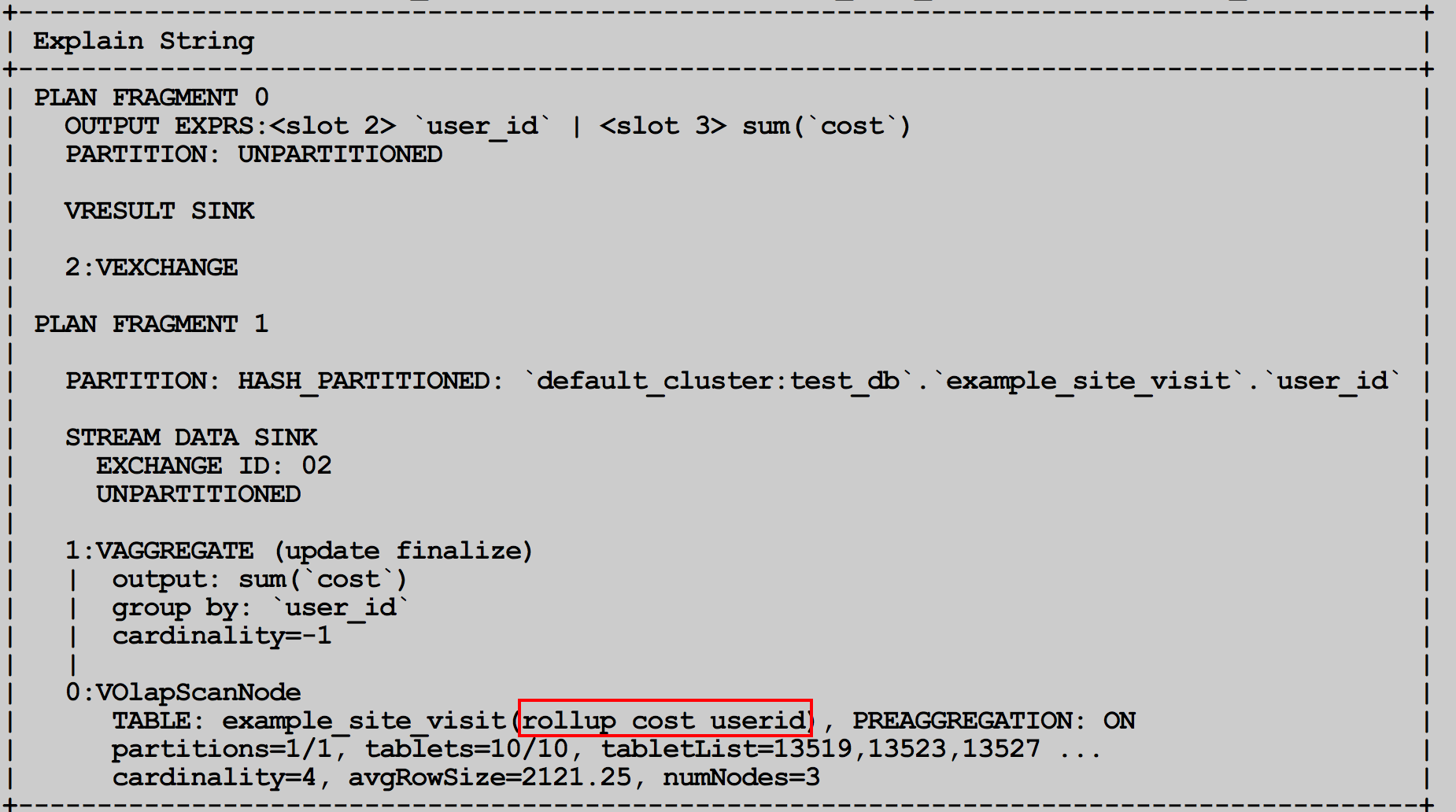
alter table example\_site\_visit add rollup rollup\_cost\_userid(user\_id,cost);

然后查看表结构信息。



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\_visit 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。

* + - * 1. 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。

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

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

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

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

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

查看命令完成状况。

SHOW ALTER TABLE ROLLUP\G;

* + 1. 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。

* + - * 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、Unique 和 Duplicate 三种数据模型中。底层的数据存储，是按照各自建表语句中，**AGGREGATE KEY、UNIQUE KEY 和 DUPLICATE KEY 中指定的列**进行**排序存储**的。

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：

* 以下表结构的前缀索引为 user\_id(8 Bytes) + age(4 Bytes) + message(prefix 24 Bytes)。

|  |  |
| --- | --- |
| **ColumnName** | **Type** |
| **user\_id** | BIGINT |
| **age** | INT |
| **message** | VARCHAR(100) |
| **max\_dwell\_time** | DATETIME |
| **min\_dwell\_time** | DATETIME |

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

|  |  |
| --- | --- |
| **ColumnName** | **Type** |
| **user\_name** | VARCHAR(20) |
| **age** | INT |
| **message** | VARCHAR(100) |
| **max\_dwell\_time** | DATETIME |
| **min\_dwell\_time** | DATETIME |

当我们的查询条件，是前缀索引的前缀时，可以极大的加快查询速度。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。

* + - * 1. 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。

* + 1. ROLLUP使用说明[​](https://doris.apache.org/zh-CN/docs/data-table/hit-the-rollup" \l "rollup%E4%BD%BF%E7%94%A8%E8%AF%B4%E6%98%8E" \o "标题的直接链接)
* 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。这也是为什么我们在源代码中，将其命名为 Materialized Index（物化索引）的原因。
* ROLLUP 是附属于 Base 表的，可以看做是 Base 表的一种**辅助数据结构**。用户可以在 Base 表的基础上，创建或删除 ROLLUP，但是不能在查询中显式的指定查询某 ROLLUP。是否命中 ROLLUP 完全由 Doris 系统自动决定。
* ROLLUP 的数据是**独立物理存储**的。因此，创建的 ROLLUP 越多，占用的磁盘空间也就越大。同时对导入速度也会有影响（导入的ETL阶段会自动产生所有 ROLLUP 的数据），但是不会降低查询效率（只会更好）。
* ROLLUP 的**数据更新与 Base 表是完全同步的**。用户无需关心这个问题。
* ROLLUP 中列的聚合方式，与 Base 表完全相同。在创建 ROLLUP 无需指定，也不能修改。
* 查询能否命中 ROLLUP 的一个必要条件（非充分条件）是，查询所涉及的**所有列**（包括 select list 和 where 中的查询条件列等）都存在于该 ROLLUP 的列中。否则，查询只能命中 Base 表。
* 某些类型的查询（如 count(\*)）在任何条件下，都无法命中 ROLLUP。
* 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。
  1. 3.8 物化视图
     1. 基本概念

物化视图是将**预先计算**（根据定义好的 SELECT 语句）好的数据集，**存储在 Doris 中的一个特殊的表**。

物化视图的出现主要是为了满足用户，既能对原始明细数据的任意维度分析，也能快速的对固定维度进行分析查询。

* + 1. 适用场景[​](https://doris.apache.org/zh-CN/docs/advanced/materialized-view/" \l "%E9%80%82%E7%94%A8%E5%9C%BA%E6%99%AF" \o "标题的直接链接)
* 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。
  + 1. 优势[​](https://doris.apache.org/zh-CN/docs/advanced/materialized-view/" \l "%E4%BC%98%E5%8A%BF" \o "标题的直接链接)

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。

* + 1. 物化视图 VS Rollup[​](https://doris.apache.org/zh-CN/docs/advanced/materialized-view/" \l "%E7%89%A9%E5%8C%96%E8%A7%86%E5%9B%BE-vs-rollup" \o "标题的直接链接)

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](https://doris.apache.org/zh-CN/docs/sql-manual/sql-reference/Data-Definition-Statements/Alter/ALTER-TABLE-ROLLUP) 语法支持的功能现在均可以通过 [CREATE MATERIALIZED VIEW](https://doris.apache.org/zh-CN/docs/sql-manual/sql-reference/Data-Definition-Statements/Create/CREATE-MATERIALIZED-VIEW) 实现。

* + 1. 使用物化视图

Doris 系统提供了一整套对物化视图的 DDL 语法，包括创建，查看，删除。DDL 的语法和 PostgreSQL, Oracle都是一致的。

* + - * 1. 创建物化视图原则

这里首先你要根据你的查询语句的特点来决定创建一个什么样的物化视图。这里并不是说你的物化视图定义和你的某个查询语句一模一样就最好。这里有两个原则：

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。

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。

原则 2

不需要给所有维度组合都创建物化视图。

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。

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。

* + - * 1. 创建物化视图示例
* 案例 1

假设用户有一张销售记录明细表，存储了每个交易的交易id，销售员，售卖门店，销售时间，以及金额。

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);

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;

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;

查看Base表的所有物化视图

desc sales\_records all;

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;

删除物化视图语法

DROP MATERIALIZED VIEW 物化视图名 on Base表名;

* 案例 2

业务场景: 计算广告的 UV，PV

假设用户的原始广告点击数据存储在 Doris，那么针对广告 PV, UV 查询就可以通过创建 bitmap\_union 的物化视图来提升查询速度。

通过下面语句首先创建一个存储广告点击数据明细的表，包含每条点击的点击时间，点击的是什么广告，通过什么渠道点击，以及点击的用户是谁。

创建一个 Base 表为

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);

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。

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;

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。

* 案例 3

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。

查询

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

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。

查看表结构

desc sales\_records all;

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。

1. Flink读写 Doris

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。

可以将 Doris 表映射为 DataStream 或者 Table。

**注意：**

* 修改和删除只支持在 Unique Key 模型上
  1. Doris 中准备表和数据

create database test;

use test;

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);

* 1. 导入 Doris 连接器

目前支持 1.17 的 doris 连接器官方仓库还没有，需要根据源码手动编译。

* + - * 1. 下载和编译源码

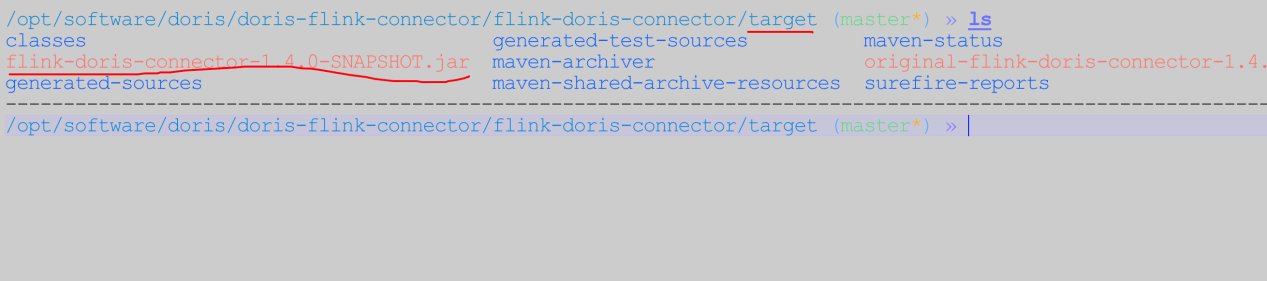
git clone <https://github.com/apache/doris-flink-connector.git>

cd doris-flink-connector/flink-doris-connector

./build.sh

选择 1.17。

编译成功之后进入 target目录 可以看到编译成功的jar 包。



* + - * 1. 把 jar 安装到本地库（windows）

mvn install:install-file -DgroupId=org.apache.doris -DartifactId=flink-doris-connector-1.17 -Dversion=1.4.0 -Dpackaging=jar -Dfile=./flink-doris-connector-1.4.0-SNAPSHOT.jar

* + - * 1. 导入依赖

<dependency>

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

<artifactId>flink-doris-connector-1.17</artifactId>

<version>1.4.0</version>

</dependency>

* 1. 流的方式读写 Doris

Flink 读写 Doris 数据主要有两种方式。

* SQL
* DataStream

Flink Doris Connector Sink 的内部实现是通过 Stream Load 服务向 Doris 写入数据, 同时也支持 Stream Load 请求参数的配置设置，具体参数可参考这里(https://doris.apache.org/zh-CN/docs/1.2/data-operate/import/import-way/stream-load-manual)，配置方法如下：

SQL 使用 WITH 参数 sink.properties. 配置。

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

* + 1. Source

DorisOptions.Builder builder = DorisOptions.builder()

.setFenodes("hadoop102:7030")

.setTableIdentifier("test.table1")

.setUsername("root")

.setPassword("aaaaaa");

DorisSource<List<?>> dorisSource = DorisSourceBuilder.<List<?>>builder()

.setDorisOptions(builder.build())

.setDorisReadOptions(DorisReadOptions.builder().build())

.setDeserializer(new SimpleListDeserializationSchema())

.build();

DataStreamSource<List<?>> stream1 = env.fromSource(dorisSource, WatermarkStrategy.noWatermarks(), "doris source");

stream1.print();

* + 1. Sink

必须开启 checkpoint。

* + - * 1. 写 json 数据

DataStreamSource<String> source = env

.fromElements(

"{\"siteid\": \"550\", \"citycode\": \"1001\", \"username\": \"ww\",\"pv\": \"100\"}");

Properties props = new Properties();

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

props.setProperty("read\_json\_by\_line", "true"); // 每行一条 json 数据

DorisSink<String> sink = DorisSink.<String>builder()

.setDorisReadOptions(DorisReadOptions.builder().build())

.setDorisOptions(DorisOptions.builder() // 设置 doris 的连接参数

.setFenodes("hadoop102:7030")

.setTableIdentifier("test.table1")

.setUsername("root")

.setPassword("aaaaaa")

.build())

.setDorisExecutionOptions(DorisExecutionOptions.builder() // 执行参数

//.setLabelPrefix("doris-label") // stream-load 导入的时候的 label 前缀

.disable2PC() // 开启两阶段提交后,labelPrefix 需要全局唯一,为了测试方便禁用两阶段提交

.setDeletable(false)

.setBufferCount(3) // 批次条数: 默认 3

.setBufferSize(8\*1024) // 批次大小: 默认 1M

.setCheckInterval(3000) // 批次输出间隔 三个对批次的限制是或的关系

.setMaxRetries(3)

.setStreamLoadProp(props) // 设置 stream load 的数据格式 默认是 csv,根据需要改成 json

.build())

.setSerializer(new SimpleStringSerializer())

.build();

source.sinkTo(sink);

* + - * 1. 写 RowData数据

其实是你把流中的数据转成 RowData，doris 再把 RowData 转成 json 或 csv 通过 streamload 的方式导入到 doris 数据库中。

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

DataType[] dataTypes = {

DataTypes.INT(),

DataTypes.SMALLINT(),

DataTypes.STRING(),

DataTypes.BIGINT()

};

Properties props = new Properties();

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

props.setProperty("read\_json\_by\_line", "true");

SingleOutputStreamOperator<RowData> source = env

.fromElements(

"{\"siteid\": \"3000\", \"citycode\": \"1001\", \"username\": \"ww\",\"pv\": \"100\"}",

"{\"siteid\": \"5000\", \"citycode\": \"1001\", \"username\": \"ww\",\"pv\": \"100\"}",

"{\"siteid\": \"6000\", \"citycode\": \"1001\", \"username\": \"ww\",\"pv\": \"100\"}"

)

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

@Override

public RowData map(String json) throws Exception {

JSONObject obj = JSON.parseObject(json);

GenericRowData rowData = new GenericRowData(4);

rowData.setField(0, obj.getIntValue("siteid"));

rowData.setField(1, obj.getShortValue("citycode"));

rowData.setField(2, StringData.fromString(obj.getString("username")));

rowData.setField(3, obj.getLongValue("pv"));

return rowData;

}

});

DorisSink<RowData> sink = DorisSink.<RowData>builder()

.setDorisReadOptions(DorisReadOptions.builder().build())

.setDorisOptions(DorisOptions.builder() // 设置 doris 的连接参数

.setFenodes("hadoop102:7030")

.setTableIdentifier("test.table1")

.setUsername("root")

.setPassword("aaaaaa")

.build())

.setDorisExecutionOptions(DorisExecutionOptions.builder() // 执行参数

//.setLabelPrefix("doris-label") // stream-load 导入的时候的 label 前缀

.disable2PC() // 开启两阶段提交后,labelPrefix 需要全局唯一,为了测试方便禁用两阶段提交

.setDeletable(false)

.setBufferCount(3) // 批次条数: 默认 3

.setBufferSize(8 \* 1024) // 批次大小: 默认 1M

.setCheckInterval(3000) // 批次输出间隔 三个对批次的限制是或的关系

.setMaxRetries(3)

.setStreamLoadProp(props)

.build())

.setSerializer(RowDataSerializer.builder()

.setType(LoadConstants.JSON)

.setFieldNames(fields)

.setFieldType(dataTypes)

.build())

.build();

source.sinkTo(sink);

* + - * 1. 写 pojo数据

以前向 doris 写数据只支持 json ，csv 字符串和 RowData。

从 1.2 开始支持 orc和 parquet。

对于 pojo 数据, 一般可以通过使用 json 解析工具把 pojo 解析成 json 字符串再写入, 或者转成RowData 再写入。

建议转成 json字符串, 相对比较方便。

* 1. SQL 的方式读写 Doris
     1. sql 读

tEnv.executeSql("CREATE TABLE flink\_doris ( " +

" siteid INT, " +

" citycode SMALLINT, " +

" username STRING, " +

" pv BIGINT " +

" ) " +

" WITH ( " +

" 'connector' = 'doris', " +

" 'fenodes' = 'hadoop102:7030', " +

" 'table.identifier' = 'test.table1', " +

" 'username' = 'root', " +

" 'password' = 'aaaaaa' " +

") ");

// 读

tEnv.sqlQuery("select \* from flink\_doris").execute().print();

* + 1. sql 写

tEnv.executeSql("CREATE TABLE flink\_doris ( " +

" siteid INT, " +

" citycode INT, " +

" username STRING, " +

" pv BIGINT " +

")WITH (" +

" 'connector' = 'doris', " +

" 'fenodes' = 'hadoop102:7030', " +

" 'table.identifier' = 'test.table1', " +

" 'username' = 'root', " +

" 'password' = 'aaaaaa', " +

" 'sink.properties.format' = 'json', " +

" 'sink.buffer-count' = '4', " +

" 'sink.buffer-size' = '4086'," +

" 'sink.enable-2pc' = 'false', " + // 测试阶段可以关闭两阶段提交,方便测试

" 'sink.properties.read\_json\_by\_line' = 'true' " +

") ");

tEnv.executeSql("insert into flink\_doris values(33, 3, '深圳', 3333)");