ClickHouse高级

V1.0

# 第1章 Explain查看执行计划

在clickhouse 20.6版本之前要查看SQL语句的执行计划需要设置日志级别为trace才能可以看到，并且只能真正执行sql，在执行日志里面查看。在20.6版本引入了原生的执行计划的语法。在20.6.3版本成为正式版本的功能。

**本文档基于目前较新稳定版21.7.3.14。**

## 1.1 基本语法

EXPLAIN [AST | SYNTAX | PLAN | PIPELINE] [setting = value, ...] SELECT ... [FORMAT ...]

* PLAN：用于查看执行计划，默认值。
  + header 打印计划中各个步骤的 head 说明，默认关闭，默认值0;
  + description 打印计划中各个步骤的描述，默认开启，默认值1；
  + actions 打印计划中各个步骤的详细信息，默认关闭，默认值0。
* AST ：用于查看语法树;
* SYNTAX：用于优化语法;
* PIPELINE：用于查看 PIPELINE 计划。
  + header 打印计划中各个步骤的 head 说明，默认关闭;
  + graph 用DOT图形语言描述管道图，默认关闭，需要查看相关的图形需要配合graphviz查看；
  + actions 如果开启了graph，紧凑打印打，默认开启。

*注：PLAN 和 PIPELINE 还可以进行额外的显示设置，如上参数所示。*

## 1.2 案例实操

### 1.2.1 新版本使用 EXPLAIN

可以再安装一个20.6以上版本，或者直接在官网的在线demo，选择高版本进行测试。

官网在线测试链接：<https://play.clickhouse.tech/?file=welcome>

1）查看PLAIN

简单查询

explain plan select arrayJoin([1,2,3,null,null]);

复杂SQL的执行计划

explain select database,table,count(1) cnt from system.parts where database in ('datasets','system') group by database,table order by database,cnt desc limit 2 by database;

打开全部的参数的执行计划

EXPLAIN header=1, actions=1,description=1 SELECT number from system.numbers limit 10;

2）AST语法树

EXPLAIN AST SELECT number from system.numbers limit 10;

3）SYNTAX语法优化

//先做一次查询

SELECT number = 1 ? 'hello' : (number = 2 ? 'world' : 'atguigu') FROM numbers(10);

//查看语法优化

EXPLAIN SYNTAX SELECT number = 1 ? 'hello' : (number = 2 ? 'world' : 'atguigu') FROM numbers(10);

//开启三元运算符优化

SET optimize\_if\_chain\_to\_multiif = 1;

//再次查看语法优化

EXPLAIN SYNTAX SELECT number = 1 ? 'hello' : (number = 2 ? 'world' : 'atguigu') FROM numbers(10);

//返回优化后的语句

SELECT multiIf(number = 1, \'hello\', number = 2, \'world\', \'xyz\')

FROM numbers(10)

4）查看PIPELINE

EXPLAIN PIPELINE SELECT sum(number) FROM numbers\_mt(100000) GROUP BY number % 20;

//打开其他参数

EXPLAIN PIPELINE header=1,graph=1 SELECT sum(number) FROM numbers\_mt(10000) GROUP BY number%20;

### 1.2.2 老版本查看执行计划

clickhouse-client -h 主机名 --send\_logs\_level=trace <<< "sql" > /dev/null

其中，send\_logs\_level参数指定日志等级为trace，<<<将SQL语句重定向至clickhouse-client进行查询，> /dev/null将查询结果重定向到空设备吞掉，以便观察日志。

注意：

1、通过将ClickHouse的服务日志，设置到DEBUG或者TRACE级别，才可以变相实现EXPLAIN查询的作用。

2、需要真正的执行SQL查询，CH才能打印计划日志，所以如果表的数据量很大，最好借助LIMIT子句，减小查询返回的数据量。

# 第2章 建表优化

## 2.1 数据类型

### 2.1.1 时间字段的类型

建表时能用数值型或日期时间型表示的字段就不要用字符串，全String类型在以Hive为中心的数仓建设中常见，但ClickHouse环境不应受此影响。

虽然ClickHouse底层将DateTime存储为时间戳Long类型，但不建议存储Long类型，因为DateTime不需要经过函数转换处理，执行效率高、可读性好。

create table t\_type2(

id UInt32,

sku\_id String,

total\_amount Decimal(16,2) ,

create\_time Int32

) engine =ReplacingMergeTree(create\_time)

partition by toYYYYMMDD(toDate(create\_time)) –-需要转换一次，否则报错

primary key (id)

order by (id, sku\_id);

### 2.1.2 空值存储类型

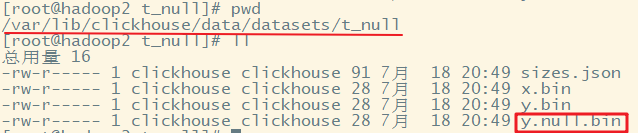
官方已经指出**Nullable类型几乎总是会拖累性能**，因为存储Nullable列时需要创建一个额外的文件来存储NULL的标记，并且Nullable列无法被索引。因此除非极特殊情况，应直接使用字段默认值表示空，或者自行指定一个在业务中无意义的值（例如用-1表示没有商品ID）。

CREATE TABLE t\_null(x Int8, y Nullable(Int8)) ENGINE TinyLog;

INSERT INTO t\_null VALUES (1, NULL), (2, 3);

SELECT x + y FROM t\_null;

查看存储的文件：（没有权限就用root用户）



官网说明：<https://clickhouse.tech/docs/zh/sql-reference/data-types/nullable/>

## 2.2 分区和索引

分区粒度根据业务特点决定，不宜过粗或过细。一般选择**按天分区**，也可以指定为Tuple()，以单表一亿数据为例，分区大小控制在10-30个为最佳。

必须指定索引列，ClickHouse中的索引列即排序列，通过order by指定，一般在查询条件中经常被用来充当筛选条件的属性被纳入进来；可以是单一维度，也可以是组合维度的索引；通常需要满足高级列在前、查询频率大的在前原则；还有基数特别大的不适合做索引列，如用户表的userid字段；通常**筛选后的数据满足在百万以内为最佳**。

比如官方案例的hits\_v1表：

……

PARTITION BY toYYYYMM(EventDate)

ORDER BY (CounterID, EventDate, intHash32(UserID))

……

visits\_v1表：

……

PARTITION BY toYYYYMM(StartDate)

ORDER BY (CounterID, StartDate, intHash32(UserID), VisitID)

……

## 2.3 表参数

Index\_granularity是用来控制索引粒度的，默认是8192，如非必须不建议调整。

如果表中不是必须保留全量历史数据，建议指定TTL（生存时间值），可以免去手动过期历史数据的麻烦，TTL 也可以通过alter table语句随时修改。（参考基础文档4.4.5 数据TTL）

## 2.4 写入和删除优化

（1）尽量不要执行单条或小批量删除和插入操作，这样会产生小分区文件，给后台Merge任务带来巨大压力

（2）不要一次写入太多分区，或数据写入太快，数据写入太快会导致Merge速度跟不上而报错，一般建议每秒钟发起2-3次写入操作，每次操作写入2w~5w条数据（依服务器性能而定）

写入过快报错，报错信息：

1. Code: 252, e.displayText() = DB::Exception: Too many parts(304). Merges are processing significantly slower than inserts

2. Code: 241, e.displayText() = DB::Exception: Memory limit (for query) exceeded:would use 9.37 GiB (attempt to allocate chunk of 301989888 bytes), maximum: 9.31 GiB

处理方式：

“ Too many parts 处理 ” ：使用WAL预写日志，提高写入性能。

in\_memory\_parts\_enable\_wal 默认为 true

在服务器内存充裕的情况下增加内存配额，一般通过max\_memory\_usage来实现

在服务器内存不充裕的情况下，建议将超出部分内容分配到系统硬盘上，但会降低执行速度，一般通过max\_bytes\_before\_external\_group\_by、max\_bytes\_before\_external\_sort参数来实现。

## 2.5 常见配置

配置项主要在config.xml或users.xml中， 基本上都在users.xml里

* config.xml的配置项

<https://clickhouse.tech/docs/en/operations/server-configuration-parameters/settings/>

* users.xml的配置项

<https://clickhouse.tech/docs/en/operations/settings/settings/>

### 2.5.1 CPU资源

|  |  |
| --- | --- |
| 配置 | 描述 |
| background\_pool\_size | 后台线程池的大小，merge线程就是在该线程池中执行，该线程池不仅仅是给merge线程用的，默认值16，允许的前提下建议改成cpu个数的2倍（线程数）。 |
| background\_schedule\_pool\_size | 执行后台任务（复制表、Kafka流、DNS缓存更新）的线程数。默认128，建议改成cpu个数的2倍（线程数）。 |
| background\_distributed\_schedule\_pool\_size | 设置为分布式发送执行后台任务的线程数，默认16，建议改成cpu个数的2倍（线程数）。 |
| max\_concurrent\_queries | 最大并发处理的请求数(包含select,insert等)，默认值100，推荐150(不够再加)~300。 |
| max\_threads | 设置单个查询所能使用的最大cpu个数，默认是cpu核数 |

### 2.5.2 内存资源

|  |  |
| --- | --- |
| 配置 | 描述 |
| max\_memory\_usage | 此参数在users.xml  中,表示单次Query占用内存最大值，该值可以设置的比较大，这样可以提升集群查询的上限。  保留一点给OS，比如128G内存的机器，设置为100GB。 |
| max\_bytes\_before\_external\_group\_by | 一般按照max\_memory\_usage的一半设置内存，当group使用内存超过阈值后会刷新到磁盘进行。  因为clickhouse聚合分两个阶段：查询并及建立中间数据、合并中间数据，结合上一项，建议50GB。 |
| max\_bytes\_before\_external\_sort | 当order by已使用max\_bytes\_before\_external\_sort内存就进行溢写磁盘(基于磁盘排序)，如果不设置该值，那么当内存不够时直接抛错，设置了该值order by可以正常完成，但是速度相对存内存来说肯定要慢点(实测慢的非常多，无法接受)。 |
| max\_table\_size\_to\_drop | 此参数在 config.xml 中，应用于需要删除表或分区的情况，默认是50GB，意思是如果删除50GB以上的分区表会失败。建议修改为0，这样不管多大的分区表都可以删除。 |

### 2.5.3 存储

ClickHouse不支持设置多数据目录，为了提升数据io性能，可以挂载虚拟券组，一个券组绑定多块物理磁盘提升读写性能，多数据查询场景SSD会比普通机械硬盘快2-3倍。

# 第3章 ClickHouse 语法优化规则

ClickHouse 的 SQL 优化规则是基于RBO(Rule Based Optimization)，下面是一些优化规则

## 3.1 准备测试用表

**1）上传官方的数据集**

将visits\_v1.tar和hits\_v1.tar上传到虚拟机，解压到clickhouse数据路径下

// 解压到clickhouse数据路径

sudo tar -xvf hits\_v1.tar -C /var/lib/clickhouse

sudo tar -xvf visits\_v1.tar -C /var/lib/clickhouse

//修改所属用户

sudo chown -R clickhouse:clickhouse /var/lib/clickhouse/data/datasets

sudo chown -R clickhouse:clickhouse /var/lib/clickhouse/metadata/datasets

**2）重启clickhouse-server**

sudo clickhouse restart

**3）执行查询**

clickhouse-client --query "SELECT COUNT(\*) FROM datasets.hits\_v1"

clickhouse-client --query "SELECT COUNT(\*) FROM datasets.visits\_v1"

*注意：官方的tar包，包含了建库、建表语句、数据内容，这种方式不需要手动建库、建表，最方便。*

*hits\_v1表有130多个字段，880多万条数据*

*visits\_v1表有180多个字段，160多万条数据*

## 3.2 COUNT 优化

在调用 count 函数时，如果使用的是 count() 或者 count(\*)，且没有 where 条件，则会直接使用 system.tables 的 total\_rows，例如:

EXPLAIN SELECT count()FROM datasets.hits\_v1;

Union

Expression (Projection)

Expression (Before ORDER BY and SELECT)

MergingAggregated

ReadNothing (Optimized trivial count)

注意 Optimized trivial count ，这是对 count 的优化。

如果 count 具体的列字段，则不会使用此项优化：

EXPLAIN SELECT count(CounterID) FROM datasets.hits\_v1;

Union

Expression (Projection)

Expression (Before ORDER BY and SELECT)

Aggregating

Expression (Before GROUP BY)

ReadFromStorage (Read from MergeTree)

## 3.3 消除子查询重复字段

下面语句子查询中有两个重复的 id 字段，会被去重:

EXPLAIN SYNTAX SELECT

a.UserID,

b.VisitID,

a.URL,

b.UserID

FROM

hits\_v1 AS a

LEFT JOIN (

SELECT

UserID,

UserID as HaHa,

VisitID

FROM visits\_v1) AS b

USING (UserID)

limit 3;

//返回优化语句：

SELECT

UserID,

VisitID,

URL,

b.UserID

FROM hits\_v1 AS a

ALL LEFT JOIN

(

SELECT

UserID,

VisitID

FROM visits\_v1

) AS b USING (UserID)

LIMIT 3

## 3.4 谓词下推

当group by有having子句，但是没有with cube、with rollup 或者with totals修饰的时候，having过滤会下推到where提前过滤。例如下面的查询，HAVING name变成了WHERE name，在group by之前过滤：

EXPLAIN SYNTAX SELECT UserID FROM hits\_v1 GROUP BY UserID HAVING UserID = '8585742290196126178';

//返回优化语句

SELECT UserID

FROM hits\_v1

WHERE UserID = \'8585742290196126178\'

GROUP BY UserID

子查询也支持谓词下推：

EXPLAIN SYNTAX

SELECT \*

FROM

(

SELECT UserID

FROM visits\_v1

)

WHERE UserID = '8585742290196126178'

//返回优化后的语句

SELECT UserID

FROM

(

SELECT UserID

FROM visits\_v1

WHERE UserID = \'8585742290196126178\'

)

WHERE UserID = \'8585742290196126178\'

再来一个复杂例子：

EXPLAIN SYNTAX

SELECT \* FROM (

SELECT

\*

FROM

(

SELECT

UserID

FROM visits\_v1)

UNION ALL

SELECT

\*

FROM

(

SELECT

UserID

FROM visits\_v1)

)

WHERE UserID = '8585742290196126178'

//返回优化后的语句

SELECT UserID

FROM

(

SELECT UserID

FROM

(

SELECT UserID

FROM visits\_v1

WHERE UserID = \'8585742290196126178\'

)

WHERE UserID = \'8585742290196126178\'

UNION ALL

SELECT UserID

FROM

(

SELECT UserID

FROM visits\_v1

WHERE UserID = \'8585742290196126178\'

)

WHERE UserID = \'8585742290196126178\'

)

WHERE UserID = \'8585742290196126178\'

## 3.5 聚合计算外推

聚合函数内的计算，会外推，例如：

EXPLAIN SYNTAX

SELECT sum(UserID \* 2)

FROM visits\_v1

//返回优化后的语句

SELECT sum(UserID) \* 2

FROM visits\_v1

## 3.6 聚合函数消除

如果对聚合键，也就是 group by key 使用 min、max、any 聚合函数，则将函数消除，例如：

EXPLAIN SYNTAX

SELECT

sum(UserID \* 2),

max(VisitID),

max(UserID)

FROM visits\_v1

GROUP BY UserID

//返回优化后的语句

SELECT

sum(UserID) \* 2,

max(VisitID),

UserID

FROM visits\_v1

GROUP BY UserID

## 3.7 删除重复的 order by key

例如下面的语句，重复的聚合键 id 字段会被去重:

EXPLAIN SYNTAX

SELECT \*

FROM visits\_v1

ORDER BY

UserID ASC,

UserID ASC,

VisitID ASC,

VisitID ASC

//返回优化后的语句：

select

……

FROM visits\_v1

ORDER BY

UserID ASC,

VisitID ASC

## 3.8 删除重复的 limit by key

例如下面的语句，重复声明的 name 字段会被去重：

EXPLAIN SYNTAX

SELECT \*

FROM visits\_v1

LIMIT 3 BY

VisitID,

VisitID

LIMIT 10

//返回优化后的语句：

select

……

FROM visits\_v1

LIMIT 3 BY VisitID

LIMIT 10

## 3.9 删除重复的 USING Key

例如下面的语句，重复的关联键 id 字段会被去重：

EXPLAIN SYNTAX

SELECT

a.UserID,

a.UserID,

b.VisitID,

a.URL,

b.UserID

FROM hits\_v1 AS a

LEFT JOIN visits\_v1 AS b USING (UserID, UserID)

//返回优化后的语句：

SELECT

UserID,

UserID,

VisitID,

URL,

b.UserID

FROM hits\_v1 AS a

ALL LEFT JOIN visits\_v1 AS b USING (UserID)

## 3.10 标量替换

如果子查询只返回一行数据，在被引用的时候用标量替换，例如下面语句中的 total\_disk\_usage 字段：

EXPLAIN SYNTAX

WITH

(

SELECT sum(bytes)

FROM system.parts

WHERE active

) AS total\_disk\_usage

SELECT

(sum(bytes) / total\_disk\_usage) \* 100 AS table\_disk\_usage,

table

FROM system.parts

GROUP BY table

ORDER BY table\_disk\_usage DESC

LIMIT 10;

//返回优化后的语句：

WITH CAST(0, \'UInt64\') AS total\_disk\_usage

SELECT

(sum(bytes) / total\_disk\_usage) \* 100 AS table\_disk\_usage,

table

FROM system.parts

GROUP BY table

ORDER BY table\_disk\_usage DESC

LIMIT 10

## 3.11 三元运算优化

如果开启了 optimize\_if\_chain\_to\_multiif 参数，三元运算符会被替换成 multiIf 函数，例如：

EXPLAIN SYNTAX

SELECT number = 1 ? 'hello' : (number = 2 ? 'world' : 'atguigu')

FROM numbers(10)

settings optimize\_if\_chain\_to\_multiif = 1;

//返回优化后的语句：

SELECT multiIf(number = 1, \'hello\', number = 2, \'world\', \'atguigu\')

FROM numbers(10)

SETTINGS optimize\_if\_chain\_to\_multiif = 1

# 第4章 查询优化

## 4.1 单表查询

### 4.1.1 Prewhere替代where

Prewhere和where语句的作用相同，用来过滤数据。不同之处在于prewhere只支持 \*MergeTree 族系列引擎的表，首先会读取指定的列数据，来判断数据过滤，等待数据过滤之后再读取select 声明的列字段来补全其余属性。

当查询列明显多于筛选列时使用Prewhere可十倍提升查询性能，Prewhere会自动优化执行过滤阶段的数据读取方式，降低io操作。

在某些场合下，prewhere语句比where语句处理的数据量更少性能更高。

#关闭where自动转prewhere(默认情况下， where条件会自动优化成prewhere)

set optimize\_move\_to\_prewhere=0;

# 使用where

select WatchID,

JavaEnable,

Title,

GoodEvent,

EventTime,

EventDate,

CounterID,

ClientIP,

ClientIP6,

RegionID,

UserID,

CounterClass,

OS,

UserAgent,

URL,

Referer,

URLDomain,

RefererDomain,

Refresh,

IsRobot,

RefererCategories,

URLCategories,

URLRegions,

RefererRegions,

ResolutionWidth,

ResolutionHeight,

ResolutionDepth,

FlashMajor,

FlashMinor,

FlashMinor2

from datasets.hits\_v1 where UserID='3198390223272470366';

# 使用prewhere关键字

select WatchID,

JavaEnable,

Title,

GoodEvent,

EventTime,

EventDate,

CounterID,

ClientIP,

ClientIP6,

RegionID,

UserID,

CounterClass,

OS,

UserAgent,

URL,

Referer,

URLDomain,

RefererDomain,

Refresh,

IsRobot,

RefererCategories,

URLCategories,

URLRegions,

RefererRegions,

ResolutionWidth,

ResolutionHeight,

ResolutionDepth,

FlashMajor,

FlashMinor,

FlashMinor2

from datasets.hits\_v1 prewhere UserID='3198390223272470366';

默认情况，我们肯定不会关闭where自动优化成prewhere，但是某些场景即使开启优化，也不会自动转换成prewhere，需要手动指定prewhere：

* 使用常量表达式
* 使用默认值为alias类型的字段
* 包含了arrayJOIN，globalIn，globalNotIn或者indexHint的查询
* select查询的列字段和where的谓词相同
* 使用了主键字段

### 4.1.2 数据采样

通过采样运算可极大提升数据分析的性能

SELECT Title,count(\*) AS PageViews

FROM hits\_v1

SAMPLE 0.1 #代表采样10%的数据,也可以是具体的条数

WHERE CounterID =57

GROUP BY Title

ORDER BY PageViews DESC LIMIT 1000

采样修饰符只有在MergeTree engine表中才有效，且在创建表时需要指定采样策略。

### 4.1.3 列裁剪与分区裁剪

数据量太大时应避免使用select \* 操作，查询的性能会与查询的字段大小和数量成线性表换，字段越少，消耗的io资源越少，性能就会越高。

反例：

select \* from datasets.hits\_v1;

正例：

select WatchID,

JavaEnable,

Title,

GoodEvent,

EventTime,

EventDate,

CounterID,

ClientIP,

ClientIP6,

RegionID,

UserID

from datasets.hits\_v1;

分区裁剪就是只读取需要的分区，在过滤条件中指定。

select WatchID,

JavaEnable,

Title,

GoodEvent,

EventTime,

EventDate,

CounterID,

ClientIP,

ClientIP6,

RegionID,

UserID

from datasets.hits\_v1

where EventDate='2014-03-23';

### 4.1.4 orderby 结合 where、limit

千万以上数据集进行order by查询时需要搭配where条件和limit语句一起使用。

#正例：

SELECT UserID,Age

FROM hits\_v1

WHERE CounterID=57

ORDER BY Age DESC LIMIT 1000

#反例：

SELECT UserID,Age

FROM hits\_v1

ORDER BY Age DESC

### 4.1.5 避免构建虚拟列

如非必须，不要在结果集上构建虚拟列，虚拟列非常消耗资源浪费性能，可以考虑在前端进行处理，或者在表中构造实际字段进行额外存储。

反例：

SELECT Income,Age,Income/Age as IncRate FROM datasets.hits\_v1;

正例：拿到Income和Age后，考虑在前端进行处理，或者在表中构造实际字段进行额外存储

SELECT Income,Age FROM datasets.hits\_v1;

### 4.1.6 uniqCombined替代distinct

性能可提升10倍以上，uniqCombined底层采用类似HyperLogLog算法实现，能接收2%左右的数据误差，可直接使用这种去重方式提升查询性能。Count(distinct )会使用uniqExact精确去重。

不建议在千万级不同数据上执行distinct去重查询，改为近似去重uniqCombined

反例：

select count(distinct rand()) from hits\_v1;

正例：

SELECT uniqCombined(rand()) from datasets.hits\_v1

### 4.1.7 使用物化视图

参考第6章。

### 4.1.8 其他注意事项

**（1）查询熔断**

为了避免因个别慢查询引起的服务雪崩的问题，除了可以为单个查询设置超时以外，还可以配置周期熔断，在一个查询周期内，如果用户频繁进行慢查询操作超出规定阈值后将无法继续进行查询操作。

**（2）关闭虚拟内存**

物理内存和虚拟内存的数据交换，会导致查询变慢，资源允许的情况下关闭虚拟内存。

**（3）配置join\_use\_nulls**

为每一个账户添加 join\_use\_nulls 配置，左表中的一条记录在右表中不存在，右表的相应字段会返回该字段相应数据类型的默认值，而不是标准SQL中的Null值。

**（4）批量写入时先排序**

批量写入数据时，必须控制每个批次的数据中涉及到的分区的数量，在写入之前最好对需要导入的数据进行排序。无序的数据或者涉及的分区太多，会导致ClickHouse无法及时对新导入的数据进行合并，从而影响查询性能。

**（5）关注CPU**

cpu一般在50%左右会出现查询波动，达到70%会出现大范围的查询超时，cpu是最关键的指标，要非常关注。

## 4.2 多表关联

### 4.2.1 准备表和数据

#创建小表

CREATE TABLE visits\_v2

ENGINE = CollapsingMergeTree(Sign)

PARTITION BY toYYYYMM(StartDate)

ORDER BY (CounterID, StartDate, intHash32(UserID), VisitID)

SAMPLE BY intHash32(UserID)

SETTINGS index\_granularity = 8192

as select \* from visits\_v1 limit 10000;

#创建join结果表：避免控制台疯狂打印数据

CREATE TABLE hits\_v2

ENGINE = MergeTree()

PARTITION BY toYYYYMM(EventDate)

ORDER BY (CounterID, EventDate, intHash32(UserID))

SAMPLE BY intHash32(UserID)

SETTINGS index\_granularity = 8192

as select \* from hits\_v1 where 1=0;

### 4.2.2 用 IN 代替 JOIN

当多表联查时，查询的数据仅从其中一张表出时，可考虑用 IN 操作而不是JOIN

insert into hits\_v2

select a.\* from hits\_v1 a where a. CounterID in (select CounterID from visits\_v1);

#反例：使用join

insert into table hits\_v2

select a.\* from hits\_v1 a left join visits\_v1 b on a. CounterID=b. CounterID;

|  |
| --- |
|  |

### 4.2.3 大小表JOIN

多表join时要满足小表在右的原则，右表关联时被加载到内存中与左表进行比较，ClickHouse中无论是Left join 、Right join 还是 Inner join 永远都是拿着右表中的每一条记录到左表中查找该记录是否存在，所以右表必须是小表。

**（1）小表在右**

insert into table hits\_v2

select a.\* from hits\_v1 a left join visits\_v2 b on a. CounterID=b. CounterID;

**（2）大表在右**

insert into table hits\_v2

select a.\* from visits\_v2 b left join hits\_v1 a on a. CounterID=b. CounterID;

### 4.2.4 注意谓词下推（版本差异）

ClickHouse在join查询时不会主动发起谓词下推的操作，需要每个子查询提前完成过滤操作，需要注意的是，是否执行谓词下推，对性能影响差别很大（新版本中已经不存在此问题，但是需要注意谓词的位置的不同依然有性能的差异）

Explain syntax

select a.\* from hits\_v1 a left join visits\_v2 b on a. CounterID=b. CounterID

having a.EventDate = '2014-03-17';

Explain syntax

select a.\* from hits\_v1 a left join visits\_v2 b on a. CounterID=b. CounterID

having b.StartDate = '2014-03-17';

insert into hits\_v2

select a.\* from hits\_v1 a left join visits\_v2 b on a. CounterID=b. CounterID

where a.EventDate = '2014-03-17';

insert into hits\_v2

select a.\* from (

select \* from

hits\_v1

where EventDate = '2014-03-17'

) a left join visits\_v2 b on a. CounterID=b. CounterID;

### 4.2.5 分布式表使用GLOBAL

两张分布式表上的IN和JOIN之前必须加上GLOBAL关键字，右表只会在接收查询请求的那个节点查询一次，并将其分发到其他节点上。如果不加GLOBAL关键字的话，每个节点都会单独发起一次对右表的查询，而右表又是分布式表，就导致右表一共会被查询N²次（N是该分布式表的分片数量），这就是查询放大，会带来很大开销。

### 4.2.6 使用字典表

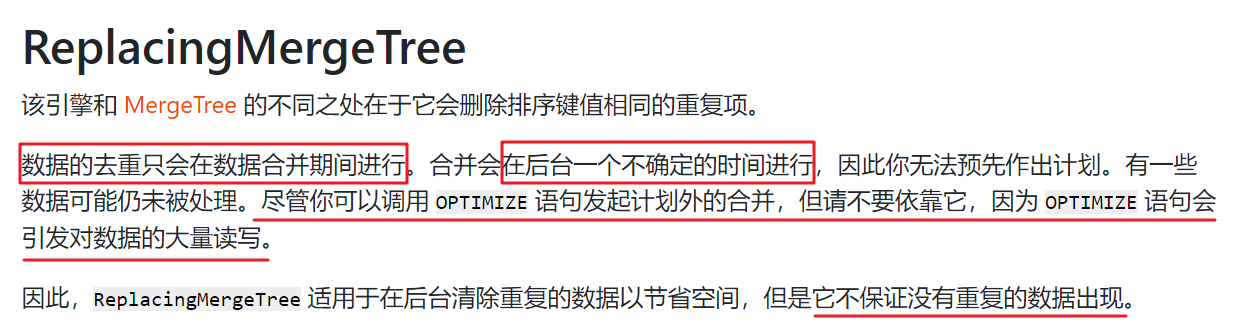
将一些需要关联分析的业务创建成字典表进行join操作，前提是字典表不宜太大，因为字典表会常驻内存

### 4.2.7 提前过滤

通过增加逻辑过滤可以减少数据扫描，达到提高执行速度及降低内存消耗的目的

# 第5章 数据一致性（重点）

查询CK手册发现，即便对数据一致性支持最好的Mergetree，也只是保证**最终一致性**：



我们在使用 ReplacingMergeTree、SummingMergeTree 这类表引擎的时候，会出现短暂数据不一致的情况。

在某些对一致性非常敏感的场景，通常有以下几种解决方案。

## 5.1 准备测试表和数据

**（1）创建表**

CREATE TABLE test\_a(

user\_id UInt64,

score String,

deleted UInt8 DEFAULT 0,

create\_time DateTime DEFAULT toDateTime(0)

)ENGINE= ReplacingMergeTree(create\_time)

ORDER BY user\_id;

其中:

user\_id 是数据去重更新的标识;

create\_time 是版本号字段，每组数据中 create\_time 最大的一行表示最新的数据;

deleted 是自定的一个标记位，比如 0 代表未删除，1 代表删除数据。

**（2）写入 1000万 测试数据**

INSERT INTO TABLE test\_a(user\_id,score)

WITH(

SELECT ['A','B','C','D','E','F','G']

)AS dict

SELECT number AS user\_id, dict[number%7+1] FROM numbers(10000000);

**（3）修改前 50万 行数据，修改内容包括 name 字段和 create\_time 版本号字段**

INSERT INTO TABLE test\_a(user\_id,score,create\_time)

WITH(

SELECT ['AA','BB','CC','DD','EE','FF','GG']

)AS dict

SELECT number AS user\_id, dict[number%7+1], now() AS create\_time FROM numbers(500000);

**（4）统计总数**

SELECT COUNT() FROM test\_a;

10500000

还未触发分区合并，所以还未去重。

## 5.2 手动 OPTIMIZE

在写入数据后，立刻执行OPTIMIZE强制触发新写入分区的合并动作。

OPTIMIZE TABLE test\_a FINAL;

语法：OPTIMIZE TABLE [db.]name [ON CLUSTER cluster] [PARTITION partition | PARTITION ID 'partition\_id'] [FINAL] [DEDUPLICATE [BY expression]]

## 5.3 通过 Group by 去重

**（1）执行去重的查询**

SELECT

user\_id ,

argMax(score, create\_time) AS score,

argMax(deleted, create\_time) AS deleted,

max(create\_time) AS ctime

FROM test\_a

GROUP BY user\_id

HAVING deleted = 0;

函数说明：

* argMax(field1，field2):按照 field2 的最大值取 field1 的值。

当我们更新数据时，会写入一行新的数据，例如上面语句中，通过查询最大的 create\_time 得到修改后的score字段值。

**（2）创建视图，方便测试**

CREATE VIEW view\_test\_a AS

SELECT

user\_id ,

argMax(score, create\_time) AS score,

argMax(deleted, create\_time) AS deleted,

max(create\_time) AS ctime

FROM test\_a

GROUP BY user\_id

HAVING deleted = 0;

**（3）插入重复数据，再次查询**

#再次插入一条数据

INSERT INTO TABLE test\_a(user\_id,score,create\_time) VALUES(0,'AAAA',now())

#再次查询

SELECT \*

FROM view\_test\_a

WHERE user\_id = 0;

**（4）删除数据测试**

#再次插入一条标记为删除的数据

INSERT INTO TABLE test\_a(user\_id,score,deleted,create\_time) VALUES(0,'AAAA',1,now());

#再次查询，刚才那条数据看不到了

SELECT \*

FROM view\_test\_a

WHERE user\_id = 0;

这行数据并没有被真正的删除，而是被过滤掉了。在一些合适的场景下，可以结合 表级别的 TTL 最终将物理数据删除。

## 5.4 通过 FINAL 查询

在查询语句后增加FINAL修饰符，这样在查询的过程中将会执行Merge的特殊逻辑（例如数据去重，预聚合等）。

但是这种方法在早期版本基本没有人使用，因为在增加 FINAL之后，我们的查询将会变成一个单线程的执行过程，查询速度非常慢。

在v20.5.2.7-stable版本中，FINAL查询支持多线程执行，并且可以通过max\_final\_threads 参数控制单个查询的线程数。但是目前读取part部分的动作依然是串行的。

FINAL查询最终的性能和很多因素相关，列字段的大小、分区的数量等等都会影响到最终的查询时间，所以还要结合实际场景取舍。

*参考链接：https://github.com/ClickHouse/ClickHouse/pull/10463*

使用hits\_v1表进行测试：

分别安装了20.4.5.36 和 21.7.3.14 两个版本的ClickHouse进行对比。

### 5.4.1 老版本测试

（1）普通查询语句

select \* from visits\_v1 WHERE StartDate = '2014-03-17' limit 100;

（2）FINAL查询

select \* from visits\_v1 FINAL WHERE StartDate = '2014-03-17' limit 100;

先前的并行查询变成了单线程。

### 5.4.2 新版本测试

**（1）普通语句查询**

select \* from visits\_v1 WHERE StartDate = '2014-03-17' limit 100 settings max\_threads = 2;

查看执行计划：

explain pipeline select \* from visits\_v1 WHERE StartDate = '2014-03-17' limit 100 settings max\_threads = 2;

(Expression)

ExpressionTransform × 2

(SettingQuotaAndLimits)

(Limit)

Limit 2 → 2

(ReadFromMergeTree)

MergeTreeThread × 2 0 → 1

明显将由2个线程并行读取 part 查询。

**（2）FINAL查询**

select \* from visits\_v1 final WHERE StartDate = '2014-03-17' limit 100 settings max\_final\_threads = 2;

查询速度没有普通的查询快，但是相比之前已经有了一些提升,查看 FINAL 查询的执行计划：

explain pipeline select \* from visits\_v1 final WHERE StartDate = '2014-03-17' limit 100 settings max\_final\_threads = 2;

(Expression)

ExpressionTransform × 2

(SettingQuotaAndLimits)

(Limit)

Limit 2 → 2

(ReadFromMergeTree)

ExpressionTransform × 2

CollapsingSortedTransform × 2

Copy 1 → 2

AddingSelector

ExpressionTransform

MergeTree 0 → 1

从CollapsingSortedTransform这一步开始已经是多线程执行，但是读取 part 部分的动作还是串行。

# 第6章 物化视图

ClickHouse的物化视图是一种查询结果的持久化，它确实是给我们带来了查询效率的提升。用户查起来跟表没有区别，它就是一张表，它也像是一张时刻在预计算的表，创建的过程它是用了一个特殊引擎，加上后来 as select，就是create一个table as select的写法。

“查询结果集”的范围很宽泛，可以是基础表中部分数据的一份简单拷贝，也可以是多表join之后产生的结果或其子集，或者原始数据的聚合指标等等。所以，物化视图不会随着基础表的变化而变化，所以它也称为快照（snapshot）

## 6.1 概述

### 6.1.1 物化视图与普通视图的区别

**普通视图不保存数据，保存的仅仅是查询语句**，查询的时候还是从原表读取数据，可以将普通视图理解为是个子查询。**物化视图则是把查询的结果根据相应的引擎存入到了磁盘或内存中**，对数据重新进行了组织，你可以理解物化视图是完全的一张新表。

### 6.1.2 优缺点

优点：查询速度**快**，要是把物化视图这些规则全部写好，它比原数据查询快了很多，总的行数少了，因为都预计算好了。

缺点：它的本质是一个流式数据的使用场景，是累加式的技术，所以要用历史数据做去重、去核这样的分析，在物化视图里面是不太好用的。在某些场景的使用也是有限的。而且如果一张表加了好多物化视图，在写这张表的时候，就会消耗很多机器的资源，比如数据带宽占满、存储一下子增加了很多。

### 6.1.3 基本语法

也是create语法，会创建一个隐藏的目标表来保存视图数据。也可以TO 表名，保存到一张显式的表。没有加TO表名，表名默认就是 .inner.物化视图名

CREATE [MATERIALIZED] VIEW [IF NOT EXISTS] [db.]table\_name [TO[db.]name] [ENGINE = engine] [POPULATE] AS SELECT ...

**1）创建物化视图的限制**

1.必须指定物化视图的engine 用于数据存储

2.TO [db].[table]语法的时候，不得使用POPULATE。

3.查询语句(select）可以包含下面的子句： DISTINCT, GROUP BY, ORDER BY, LIMIT…

4.物化视图的alter操作有些限制，操作起来不大方便。

5.若物化视图的定义使用了TO [db.]name 子语句，则可以将目标表的视图 卸载 DETACH 再装载 ATTACH

**2）物化视图的数据更新**

（1）物化视图创建好之后，若源表被写入新数据则物化视图也会同步更新

（2）POPULATE 关键字决定了物化视图的更新策略：

* + 若有POPULATE 则在创建视图的过程会将源表已经存在的数据一并导入，类似于 create table ... as
  + 若无POPULATE 则物化视图在创建之后没有数据，只会在创建只有同步之后写入源表的数据
  + clickhouse 官方并不推荐使用POPULATE，因为在创建物化视图的过程中同时写入的数据不能被插入物化视图。

（3）物化视图不支持同步删除，若源表的数据不存在（删除了）则物化视图的数据仍然保留

（4）物化视图是一种特殊的数据表，可以用show tables 查看

（5）物化视图数据的删除：

（6）物化视图的删除：

## 6.2 案例实操

对于一些确定的数据模型，可将统计指标通过物化视图的方式进行构建，这样可避免查询时重复计算的过程，物化视图会在有新数据插入时进行更新。

### 6.2.1 准备测试用表和数据

1）建表

#建表语句

CREATE TABLE hits\_test

(

EventDate Date,

CounterID UInt32,

UserID UInt64,

URL String,

Income UInt8

)

ENGINE = MergeTree()

PARTITION BY toYYYYMM(EventDate)

ORDER BY (CounterID, EventDate, intHash32(UserID))

SAMPLE BY intHash32(UserID)

SETTINGS index\_granularity = 8192

2）导入一些数据

INSERT INTO hits\_test

SELECT

EventDate,

CounterID,

UserID,

URL,

Income

FROM hits\_v1

limit 10000;

### 6.2.2 创建物化视图

#建表语句

CREATE MATERIALIZED VIEW hits\_mv

ENGINE=SummingMergeTree

PARTITION BY toYYYYMM(EventDate) ORDER BY (EventDate, intHash32(UserID))

AS SELECT

UserID,

EventDate,

count(URL) as ClickCount,

sum(Income) AS IncomeSum

FROM hits\_test

WHERE EventDate >= '2014-03-20' #设置更新点,该时间点之前的数据可以另外通过 #insert into select …… 的方式进行插入

GROUP BY UserID,EventDate;

##或者可以用下列语法，表A可以是一张mergetree表

CREATE MATERIALIZED VIEW 物化视图名 TO 表A

AS SELECT FROM 表B;

#不建议添加populate关键字进行全量更新

### 6.3.3 导入增量数据

#导入增量数据

INSERT INTO hits\_test

SELECT

EventDate,

CounterID,

UserID,

URL,

Income

FROM hits\_v1

WHERE EventDate >= '2014-03-23'

limit 10;

#查询物化视图

SELECT \* FROM hits\_mv;

### 6.3.4 导入历史数据

#导入增量数据

INSERT INTO hits\_mv

SELECT

UserID,

EventDate,

count(URL) as ClickCount,

sum(Income) AS IncomeSum

FROM hits\_test

WHERE EventDate = '2014-03-20'

GROUP BY UserID,EventDate

#查询物化视图

SELECT \* FROM hits\_mv;

# 第7章 MaterializeMySQL引擎

## 7.1 概述

MySQL 的用户群体很大，为了能够增强数据的实时性，很多解决方案会利用 binlog 将数据写入到 ClickHouse。为了能够监听 binlog 事件，我们需要用到类似 canal 这样的第三方中间件，这无疑增加了系统的复杂度。

ClickHouse 20.8.2.3版本新增加了 MaterializeMySQL 的 database 引擎，该 database 能映射到 MySQL 中的某个 database，并自动在 ClickHouse 中创建对应的 ReplacingMergeTree。ClickHouse服务做为MySQL副本，读取Binlog并执行DDL和DML请求，实现了基于MySQL Binlog机制的业务数据库实时同步功能。

### 7.1.1 特点

（1）MaterializeMySQL 同时支持**全量**和**增量**同步，在 database 创建之初会全量同步 MySQL 中的表和数据，之后则会通过 binlog 进行增量同步。

（2）MaterializeMySQL database 为其所创建的每张 ReplacingMergeTree 自动增加了 \_sign 和 \_version 字段。

其中， \_version 用作 ReplacingMergeTree 的 ver 版本参数，每当监听到 insert、update 和 delete 事件时，在 databse 内全局自增。而 \_sign 则用于标记是否被删除，取值 1 或者 -1。

目前 MaterializeMySQL 支持如下几种 binlog 事件:

* MYSQL\_WRITE\_ROWS\_EVENT: \_sign = 1，\_version ++
* MYSQL\_DELETE\_ROWS\_EVENT: \_sign = -1，\_version ++
* MYSQL\_UPDATE\_ROWS\_EVENT: 新数据 \_sign = 1
* MYSQL\_QUERY\_EVENT: 支持 CREATE TABLE 、DROP TABLE 、RENAME TABLE等。

### 7.1.2 使用细则

**（1）DDL查询**

MySQL DDL查询被转换成相应的ClickHouse DDL查询（ALTER, CREATE, DROP, RENAME）。如果ClickHouse不能解析某些DDL查询，该查询将被忽略。

**（2）数据复制**

MaterializeMySQL不支持直接插入、删除和更新查询，而是将DDL语句进行相应转换：

MySQL INSERT查询被转换为INSERT with \_sign=1。

MySQL DELETE查询被转换为INSERT with \_sign=-1。

MySQL UPDATE查询被转换成INSERT with \_sign=1和INSERT with \_sign=-1。

**（3）SELECT查询**

如果在SELECT查询中没有指定\_version，则使用FINAL修饰符，返回\_version的最大值对应的数据，即最新版本的数据。

如果在SELECT查询中没有指定\_sign，则默认使用WHERE \_sign=1，即返回未删除状态（\_sign=1)的数据。

**（4）索引转换**

ClickHouse数据库表会自动将MySQL主键和索引子句转换为ORDER BY元组。

ClickHouse只有一个物理顺序，由ORDER BY子句决定。如果需要创建新的物理顺序，请使用物化视图。

## 7.2 案例实操

### 7.2.1 MySQL开启binlog和GTID模式

**（1）确保 MySQL 开启了 binlog 功能，且格式为 ROW**

打开/etc/my.cnf,在[mysqld]下添加：

server-id=1

log-bin=mysql-bin

binlog\_format=ROW

**（2）开启GTID模式**

如果如果clickhouse使用的是20.8 prestable之后发布的版本，那么MySQL还需要配置开启GTID模式, 这种方式在mysql主从模式下可以确保数据同步的一致性(主从切换时)。

gtid-mode=on

enforce-gtid-consistency=1 # 设置为主从强一致性

log-slave-updates=1 # 记录日志

*GTID 是 MySQL 复制增强版，从 MySQL 5.6 版本开始支持，目前已经是 MySQL 主流复制模式。它为每个 event 分配一个全局唯一ID和序号，我们可以不用关心 MySQL 集群主从拓扑结构，直接告知 MySQL 这个 GTID 即可。*

**（3）重启MySQL**

sudo systemctl restart mysqld

### 7.2.2 准备MySQL表和数据

**（1）在 MySQL 中创建数据表并写入数据**

CREATE DATABASE testck;

CREATE TABLE `testck`.`t\_organization` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`code` int NOT NULL,

`name` text DEFAULT NULL,

`updatetime` datetime DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY (`code`)

) ENGINE=InnoDB;

INSERT INTO testck.t\_organization (code, name,updatetime) VALUES(1000,'Realinsight',NOW());

INSERT INTO testck.t\_organization (code, name,updatetime)

VALUES(1001, 'Realindex',NOW());

INSERT INTO testck.t\_organization (code, name,updatetime) VALUES(1002,'EDT',NOW());

**（2）创建第二张表**

CREATE TABLE `testck`.`t\_user` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`code` int,

PRIMARY KEY (`id`)

) ENGINE=InnoDB;

INSERT INTO testck.t\_user (code) VALUES(1);

### 7.2.3 开启ClickHouse物化引擎

set allow\_experimental\_database\_materialize\_mysql=1;

### 7.2.4 创建复制管道

（1）ClickHouse中创建 MaterializeMySQL 数据库

CREATE DATABASE test\_binlog ENGINE = MaterializeMySQL('hadoop1:3306','testck','root','000000');

其中 4 个参数分别是 MySQL地址、databse、username 和 password。

（2）查看ClickHouse的数据

use test\_binlog;

show tables;

select \* from t\_organization;

select \* from t\_user;

### 7.2.5 修改数据

（1）在 MySQL 中修改数据:

update t\_organization set name = CONCAT(name,'-v1') where id = 1

（2）查看clickhouse日志可以看到binlog监听事件，查询clickhouse

select \* from t\_organization;

### 7.2.6 删除数据

（1）MySQL删除数据:

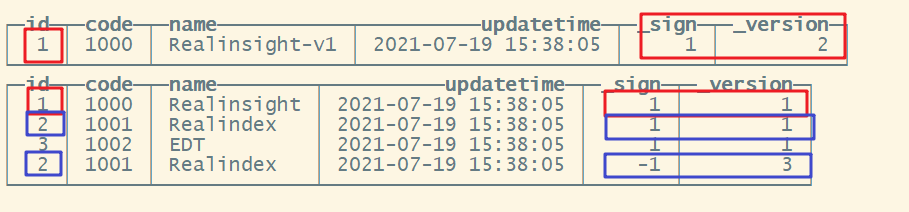
DELETE FROM t\_organization where id = 2;

（2）ClicKHouse，日志有 DeleteRows 的 binlog 监听事件，查看数据：

select \* from t\_organization;

（3）在刚才的查询中增加 \_sign 和 \_version 虚拟字段

select \*,\_sign,\_version from t\_organization order by \_sign desc,\_version desc;



在查询时，对于已经被删除的数据，\_sign=-1，ClickHouse 会自动重写 SQL，将 \_sign = -1 的数据过滤掉;

对于修改的数据，则自动重写 SQL，为其增加 FINAL 修饰符。

select \* from t\_organization

等同于

select \* from t\_organization final where \_sign = 1

### 7.2.7 删除表

（1）在mysql执行删除表

drop table t\_user;

（2）此时在clickhouse处会同步删除对应表，如果查询会报错

show tables;

select \* from t\_user;

DB::Exception: Table scene\_mms.scene doesn't exist..

（3）mysql新建表，clickhouse可以查询到

CREATE TABLE `testck`.`t\_user` (

`id` int(11) NOT NULL AUTO\_INCREMENT,

`code` int,

PRIMARY KEY (`id`)

) ENGINE=InnoDB;

INSERT INTO testck.t\_user (code) VALUES(1);

#ClickHouse查询

show tables;

select \* from t\_user;

# 第8章 常见问题排查

## 8.1 分布式DDL某数据节点的副本不执行

**（1）问题：**使用分布式ddl执行命令create table on cluster xxxx 某个节点上没有创建表，但是client返回正常，查看日志有如下报错。

<Error> xxx.xxx: Retrying createReplica(), because some other replicas were created at the same time

**（2）解决办法：**重启该不执行的节点。

## 8.2 数据副本表和数据不一致

**（1）问题：**由于某个数据节点副本异常，导致两数据副本表不一致，某个数据副本缺少表，需要将两个数据副本调整一致。

**（2）解决办法：**

在缺少表的数据副本节点上创建缺少的表，创建为本地表，表结构可以在其他数据副本通过show crete table xxxx获取。

表结构创建后，clickhouse会自动从其他副本同步该表数据，验证数据量是否一致即可。

## 8.3 副本节点全量恢复

**（1）问题：**某个数据副本异常无法启动，需要重新搭建副本。

**（2）解决办法：**

清空异常副本节点的metadata和data目录。

从另一个正常副本将metadata目录拷贝过来（这一步之后可以启动数据库，但是只有表结构没有数据）。

执行sudo -u clickhouse touch /data/clickhouse/flags/force\_restore\_data

启动数据库。

## 8.4 数据副本启动缺少zk表

**（1）问题：**某个数据副本表在zk上丢失数据，或者不存在，但是metadata元数据里存在，导致启动异常，报错：

Can’t get data for node /clickhouse/tables/01-02/xxxxx/xxxxxxx/replicas/xxx/metadata: node doesn’t exist (No node): Cannot attach table xxxxxxx

**（2）解决办法：**

metadata中移除该表的结构文件，如果多个表报错都移除

mv metadata/xxxxxx/xxxxxxxx.sql /tmp/

启动数据库

手工创建缺少的表，表结构从其他节点show create table获取。

创建后会自动同步数据，验证数据是否一致。

## 8.5 ZK table replicas数据未删除，导致重建表报错

**（1）问题：**重建表过程中，先使用drop table xxx on cluster xxx ,各节点在clickhouse上table已物理删除，但是zk里面针对某个clickhouse节点的table meta信息未被删除（低概率事件），因zk里仍存在该表的meta信息，导致再次创建该表create table xxx on cluster, 该节点无法创建表(其他节点创建表成功)，报错：

Replica /clickhouse/tables/01-03/xxxxxx/xxx/replicas/xxx already exists..

**（2）解决办法：**

从其他数据副本cp该table的metadata sql过来.

重启节点。

## 8.6 Clickhouse节点意外关闭

**（1）问题：**模拟其中一个节点意外宕机，在大量insert数据的情况下，关闭某个节点。

**（2）现象：**数据写入不受影响、数据查询不受影响、建表DDL执行到异常节点会卡住，报错：

Code: 159. DB::Exception: Received from localhost:9000. DB::Exception: Watching task /clickhouse/task\_queue/ddl/query-0000565925 is executing longer than distributed\_ddl\_task\_timeout (=180) seconds. There are 1 unfinished hosts (0 of them are currently active), they are going to execute the query in background.

**（3）解决办法：**启动异常节点，期间其他副本写入数据会自动同步过来，其他副本的建表DDL也会同步。

## 8.7 其他问题参考

<https://help.aliyun.com/document_detail/162815.html?spm=a2c4g.11186623.6.652.312e79bd17U8IO>