ClickHouse

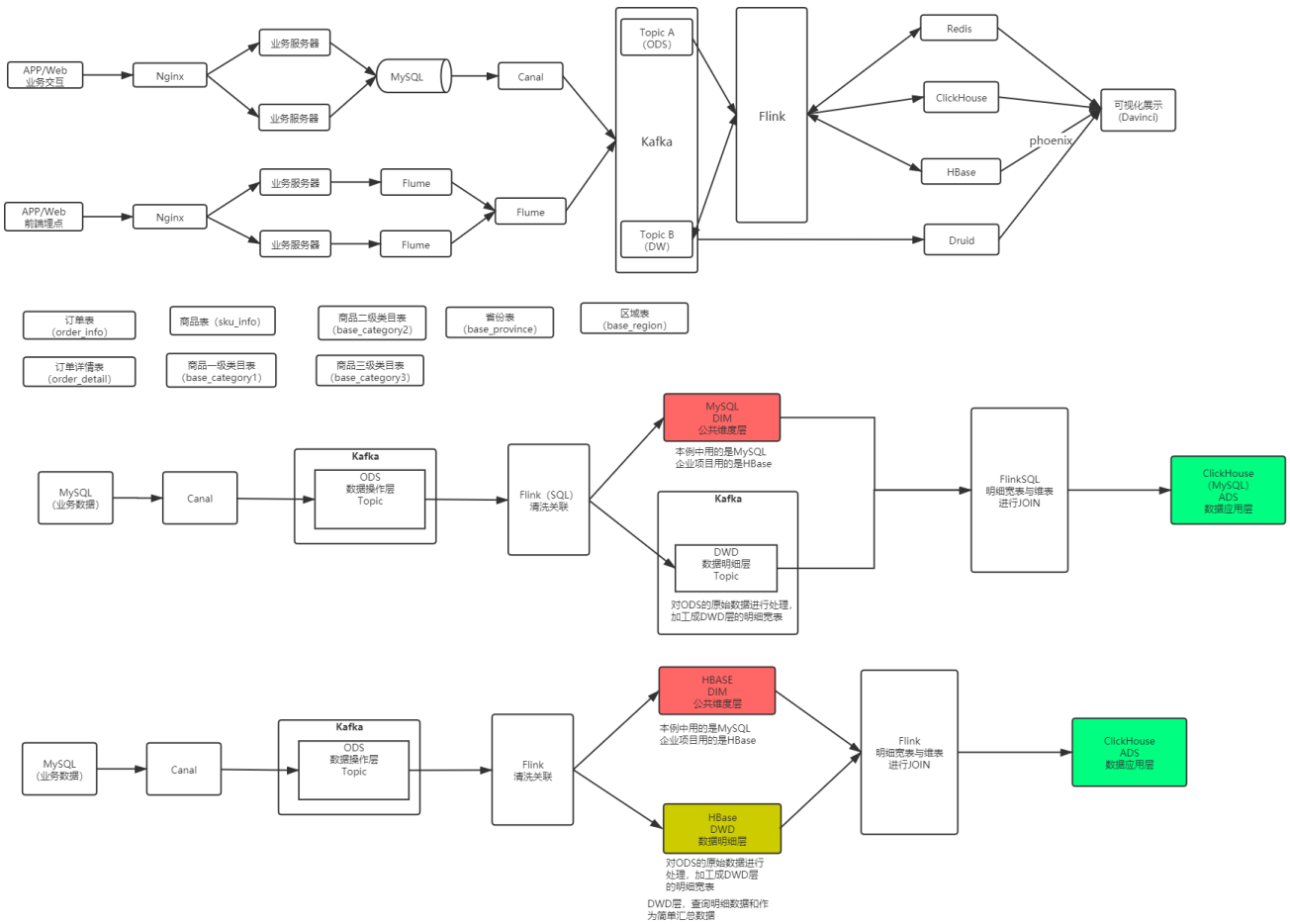
第一部分 概述

1.1 概述

ClickHouse是一个快速开源的OLAP数据库管理系统，它是面向列的，允许使用SQL查询实时生成分析报告。

随着物联网IOT时代的来临，IOT设备感知和报警存储的数据越来越大，有用的价值数据需要数据分析师去分析。大数据分析成了非常重要的环节。当然近两年开启的开源大潮，为大数据分析工程师提供了十分富余的工具。但这同时也增加了开发者选择合适的工具的难度，尤其对于新入行的开发者来说。学习成本，框架的多样化和复杂度成了很大的难题。例如kafka,hdfs,spark,hive 等等组合才能产生最后的分析结果。把各种开源框架、工具、库、平台人工整合到一起所需工作之复杂，是大数据领域开发和数据分析师常有的抱怨之一，也是他们支持大数据分析平台简单化和统一化的首要原因。

复杂的架构图：



从业务维度来分析，用户需求会反向促使技术发展

传统OLTP、OLAP

OLTP On-Line Transaction Processing： 联机事务处理过程。

`ERP: Enterprise Resource Planning 企业资源计划`

`CRM: Customer Relationship Management 客户关系管理`

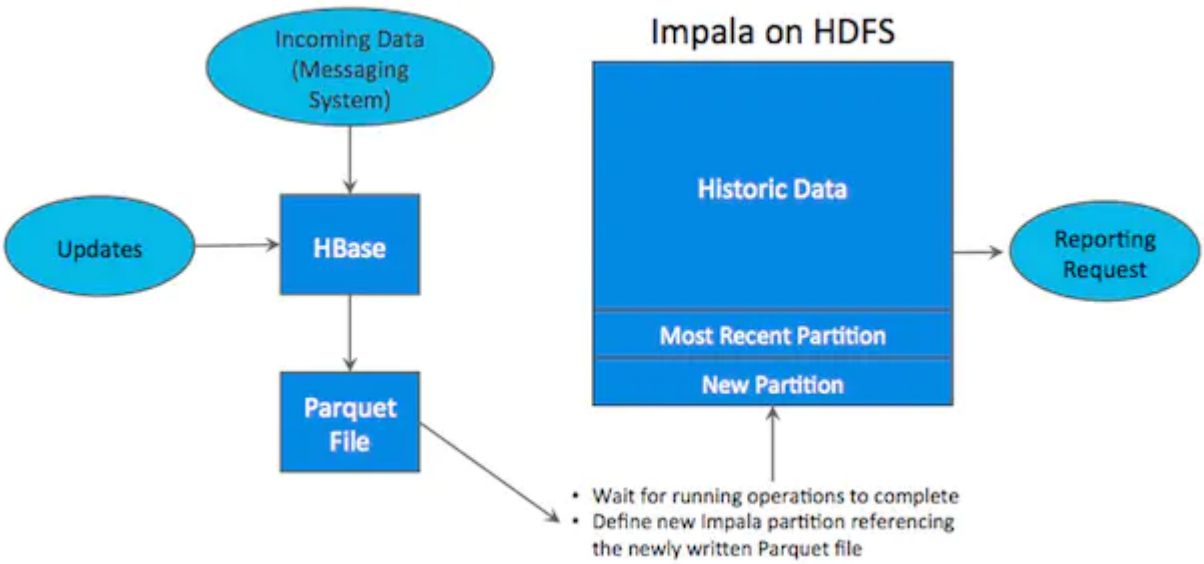
流程审批、数据录入、填报等

特点：线下工作线上化，数据保存在各自的系统中，互不相通（数据孤岛）

OLAP On-Line Analytical Processing： 联机分析系统

分析报表，分析决策等

OLAP的实现方案一：（数仓）



如上图所示，数据实时写入 HBase，实时的数据更新也在 HBase 完成，为了应对 OLAP 需求，我们定时（通常是 T+1 或者 T+H）将 HBase 数据写成静态的文件（如：Parquet）导入到 OLAP 引擎（如：HDFS，比较常见的是Impala操作Hive）。这一架构能满足既需要随机读写，又可以支持 OLAP 分析的场景，但他有如下缺点：

- **架构复杂。**从架构上看，数据在 HBase、消息队列、HDFS 间流转，涉及环节太多，运维成本很高。并且每个环节需要保证高可用，都需要维护多个副本，存储空间也有一定的浪费。最后数据在多个系统上，对数据安全策略、监控等都提出了挑战。
- **时效性低。**数据从 HBase 导出成静态文件是周期性的，一般这个周期是一天（或一小时），在时效性上不是很高。
- **难以应对后续的更新。**真实场景中，总会有数据是「延迟」到达的。如果这些数据之前已经从 HBase 导出到 HDFS，新到的变更数据就难以处理了，一个方案是把原有数据应用上新的变更后重写一遍，但这代价又很高。

OLAP的实现方案二：ClickHouse、Kudu等

1.2 Clickhouse 发展历史

Yandex在2016年6月15日开源了一个数据分析的数据库，名字叫做ClickHouse，这对保守俄罗斯人来说是个特大事。更让人惊讶的是，这个列式存储数据库的跑分要超过很多流行的商业MPP数据库软件，例如Vertica。如果你没有听过Vertica，那你一定听过 Michael Stonebraker，2014年图灵奖的获得者，PostgreSQL和Ingres发明者（Sybase和SQL Server都是继承 Ingres而来的），Paradigm4和SciDB的创办者。Michael Stonebraker于2005年创办Vertica公司，后来该公司被HP收购，HP Vertica成为MPP列式存储商业数据库的高性能代表，Facebook就购买了Vertica数据用于用户行为分析。

ClickHouse的技术演变之路：

ClickHouse的背后研发团队是俄罗斯的Yandex公司，2011年在纳斯达克上市，它的核心产品是搜索引擎。

我们知道，做搜索引擎的公司营收非常依赖流量和在线广告，所以做搜索引擎的公司一般会并行推出在线流量分析产品，比如说百度的百度统计，Google的 Google Analytics等。Yandex的Yandex.Metricah。ClickHouse就是在这种背景下诞生的。

ROLAP： 传统关系型数据库OLAP，基于MySQL的MyISAM表引擎。

MOLAP： 借助物化视图的形式实现数据立方体。预处理的结果存在HBase这类高性能的分布式数据库

HOLAP：R和M的结合体H

ROLAP： ClickHouse

1.3 Clickhouse 支持特性剖析

在看Clickhouse 运行场景之前要了解技术的功能特性以及弊端是一个技术架构以及开发人员所要了解的。只有”知己知彼”才可以”百战不殆”,接下来我们看一下Clickhouse的具体特点；

1.真正的面向列的DBMS

2.数据高效压缩

3.磁盘存储的数据

4.多核并行处理

5.在多个服务器上分布式处理

6.SQL语法支持

7.向量化引擎

8.实时数据更新

9.索引

10.适合在线查询

11.支持近似预估计算

12.支持嵌套的数据结构

13.支持数组作为数据类型

14.支持限制查询复杂性以及配额

15.复制数据和对数据完整性的支持

1.3.1 真正的面向列的DBMS

如果你想让查询变得更快：

1、减少数据扫描范围

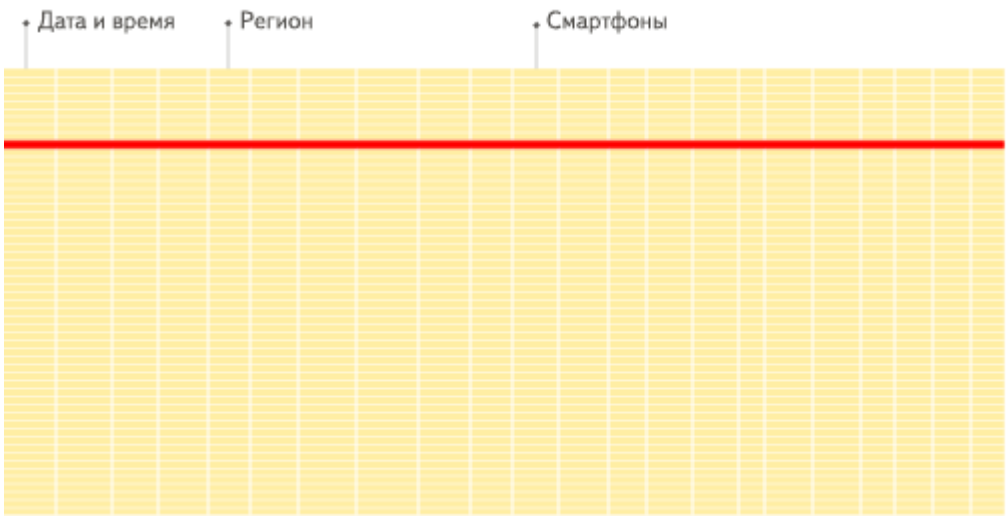
2、减少数据传输时的大小

在一个真正的面向列的DBMS中，没有任何“垃圾”存储在值中。例如，必须支持定长数值，以避免在数值旁边存储长度“数字”。例如，十亿个UInt8类型的值实际上应该消耗大约1 GB的未压缩磁盘空间，否则这将强烈影响CPU的使用。由于解压缩的速度（CPU使用率）主要取决于未压缩的数据量，所以即使在未压缩的情况下，紧凑地存储数据（没有任何“垃圾”）也是非常重要的。

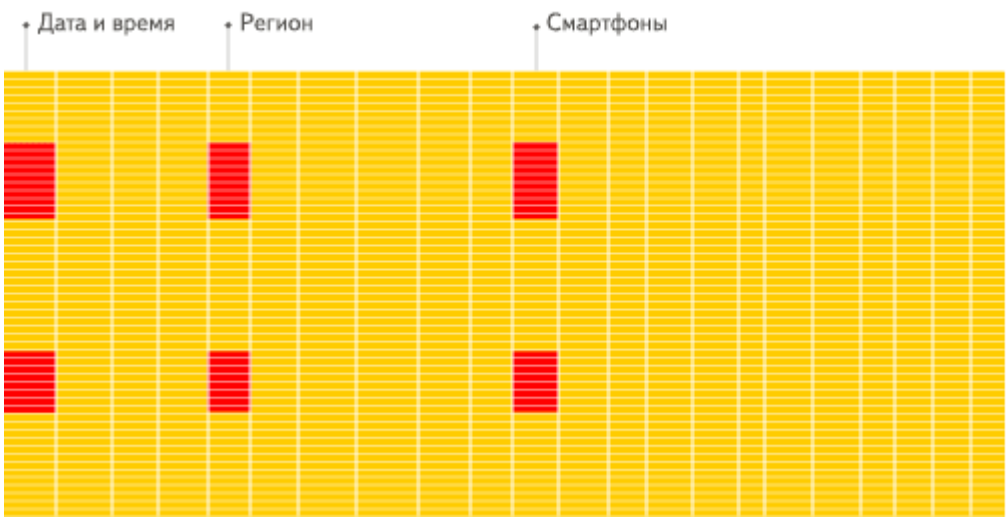
因为有些系统可以单独存储单独列的值，但由于其他场景的优化，无法有效处理分析查询。例如HBase，BigTable，Cassandra和HyperTable。在这些系统中，每秒钟可以获得大约十万行的吞吐量，但是每秒不会达到数亿行。

另外，ClickHouse是一个DBMS，而不是一个单一的数据库。ClickHouse允许在运行时创建表和数据库，加载数据和运行查询，而无需重新配置和重新启动服务器。

Row-oriented DBMS



Column-oriented DBMS



DBMS：数据库管理系统

之所以称之为DBMS，是因为ClickHouse拥有：

1、DDL

2、DML

3、权限管理

4、数据备份

5、分布式存储

等功能。

1.3.2 数据压缩

一些面向列的DBMS（InfiniDB CE和MonetDB）不使用数据压缩。但是，数据压缩确实提高了性能。

1.3.3 磁盘存储的数据

许多面向列的DBMS（SAP HANA和GooglePowerDrill）只能在内存中工作。但即使在数千台服务器上，内存也太小，无法在Yandex.Metrica中存储所有浏览量和会话。

1.3.4 多核并行处理

多核并行化大型查询。

1.3.5 在多个服务器上分布式处理

上面列出的列式DBMS几乎都不支持分布式处理。在ClickHouse中，数据可以驻留在不同的分片上。每个分片可以是用于容错的一组副本。查询在所有分片上并行处理。这对用户来说是透明的。

1.3.6 SQL支持

支持的查询包括GROUP BY, ORDER BY

子查询在FROM，IN，JOIN子句中被支持;

标量子查询支持。

关联子查询不支持。

真是因为ClickHouse提供了标准协议的SQL查询接口，使得现有可视化分析系统能够轻松与他集成对接

1.3.7 向量化引擎

数据不仅按列存储，而且由矢量 - 列的部分进行处理。这使我们能够实现高CPU性能。

向量化执行时寄存器硬件层面上的特性。可以理解为消除程序中循环的优化。

为了实现向量化执行，需要利用CPU的SIMD指令（Single Instruction Multiple Data），即用单条指令处理多条数据。现代计算机系统概念中，他是利用数据并行来提高性能的一种实现方式，，他的原理是在CPU寄存器层面实现数据并行的实现原理。

1.3.8 实时数据更新

ClickHouse支持主键表。为了快速执行对主键范围的查询，数据使用合并树(MergeTree)进行递增排序。由于这个原因，数据可以不断地添加到表中。添加数据时无锁处理。

1.3.9 索引

例如，带有主键可以在特定的时间范围内为特定客户端（Metrica计数器）抽取数据，并且延迟时间小于几十毫秒。

1.3.10 支持在线查询

我们可以使用该系统作为Web界面的后端。低延迟意味着可以无延迟实时地处理查询。

1.3.11 支持近似计算

(1) 系统包含用于近似计算各种值，中位数和分位数的集合函数。

(2) 支持基于部分（样本）数据运行查询并获得近似结果。在这种情况下，从磁盘检索比例较少的数据。

(3) 支持为有限数量的随机密钥（而不是所有密钥）运行聚合。在数据中密钥分发的特定条件下，这提供了相对准确的结果，同时使用较少的资源。

1.3.12 数据复制和对数据完整性的支持。

使用异步多主复制。写入任何可用的副本后，数据将分发到所有剩余的副本。系统在不同的副本上保持相同的数据。数据在失败后自动恢复

> ClickHouse的不完美：

Ø 1.不支持事物。

Ø 2.不支持Update/Delete操作。

Ø 3.支持有限操作系统。

现在支持ubuntu,centos 需要自己编译，不过有热心人已经编译好了，拿来用就行。对于Windows 不支持。

1.4 ClickHouse应用场景

自从ClickHouse2016年6月15日开源后，ClickHouse中文社区随后成立。中文开源组开始以易观，海康威视,美团，新浪，京东,58,腾讯,酷狗音乐和俄罗斯开源社区等人员组成，随着开源社区的不断活跃，陆续有神州数码，青云，PingCAP，中软国际等公司成员加入以及其他公司成员加入。初始在群里讨论技术后续有一些大型公司陆续运用到项目中，介于分享不方便问题解决，建立了相应的论坛。根据交流得知一些大公司已经运用。

可以应用以下场景：

1.电信行业用于存储数据和统计数据使用。

2.新浪微博用于用户行为数据记录和分析工作。

3.用于广告网络和RTB,电子商务的用户行为分析。

4.信息安全里面的日志分析。

5.检测和遥感信息的挖掘。

6.商业智能。

7.网络游戏以及物联网的数据处理和价值数据分析。

8.最大的应用来自于Yandex的统计分析服务Yandex.Metrica，类似于谷歌Analytics(GA)，或友盟统计，小米统计，帮助网站或移动应用进行数据分析和精细化运营工具，据称Yandex.Metrica为世界上第二大的网站分析平台。ClickHouse在这个应用中，部署了近四百台机器，每天支持200亿的事件和历史总记录超过13万亿条记录，这些记录都存有原始数据（非聚合数据），随时可以使用SQL查询和分析，生成用户报告。

1.5 ClickHouse 和一些技术的比较

1.商业OLAP数据库

例如：HP Vertica, Actian the Vector,

区别：ClickHouse是开源而且免费的

2.云解决方案

例如：亚马逊RedShift和谷歌的BigQuery

区别：ClickHouse可以使用自己机器部署，无需为云付费

3.Hadoop生态软件

例如：Cloudera Impala, Spark SQL, Facebook Presto , Apache Drill

区别：

ClickHouse支持实时的高并发系统

ClickHouse不依赖于Hadoop生态软件和基础

ClickHouse支持分布式机房的部署

4.开源OLAP数据库

例如：InfiniDB, MonetDB, LucidDB

区别：这些项目的应用的规模较小，并没有应用在大型的互联网服务当中，相比之下，ClickHouse的成熟度和稳定性远远超过这些软件。

5.开源分析，非关系型数据库

例如：Druid , Apache Kylin

区别：ClickHouse可以支持从原始数据的直接查询，ClickHouse支持类SQL语言，提供了传统关系型数据的便利。

1.6 总结

在大数据分析领域中，传统的大数据分析需要不同框架和技术组合才能达到最终的效果，在人力成本，技术能力和硬件成本上以及维护成本让大数据分析变得成为昂贵的事情。让很多中小型企业非常苦恼，不得不被迫租赁第三方大型公司的数据分析服务。

ClickHouse开源的出现让许多想做大数据并且想做大数据分析的很多公司和企业耳目一新。ClickHouse 正是以不依赖Hadoop 生态、安装和维护简单、查询速度快、可以支持SQL等特点在大数据分析领域越走越远。

第二部分 安装：

2.1 下载地址

官网：https://clickhouse.yandex/

下载地址：http://repo.red-soft.biz/repos/clickhouse/stable/el6/

2.2 单机模式

2.2.1上传4个文件到/root/apps/software/clickhouse_rpm

```
-rw-r--r--. 1 root root     6376 Aug 25 11:19 clickhouse-client-20.5.4.40-1.el7.x86_64.rpm
-rw-r--r--. 1 root root 57490408 Aug 25 11:19 clickhouse-common-static-20.5.4.40-
1.el7.x86_64.rpm
-rw-r--r--. 1 root root 35102796 Aug 25 11:19 clickhouse-server-20.5.4.40-1.el7.x86_64.rpm
-rw-r--r--. 1 root root    12988 Aug 25 11:19 clickhouse-server-common-20.5.4.40-
1.el7.x86_64.rpm
```

2.2.2分别安装这4个rpm文件

```
rpm -ivh ./*.rpm
```

2.2.3 启动ClickServer

前台启动:

```
[root@hdp-1 ~]# sudo -u clickhouse clickhouse-server --config-file=/etc/clickhouse-
server/config.xml
```

后台启动:

```
[root@hdp-1 ~]# nohup sudo -u clickhouse clickhouse-server --config-file=/etc/clickhouse-
server/config.xml  >null 2>&1 &
```

2.2.4 使用client连接server

```

[root@hdp-1 ~]# clickhouse-client -m
ClickHouse client version 20.5.4.40.
Connecting to localhost:9000 as user default.
Connected to ClickHouse server version 20.5.4 revision 54435.

hdp-1 :)
```

2.3 分布式集群安装

2.3.1 在hdp-2,hdp-3上面执行单机安装的所有步骤

2.3.2 三台机器修改配置文件config.xml

```
[root@hdp-1 ~]# vim /etc/clickhouse-server/config.xml
```

注意点：

```
<!-- Path to data directory, with trailing slash. -->
    <path>/var/lib/clickhouse/</path>
```

zookeeper标签上面增加：

```
<include_from>/etc/clickhouse-server/config.d/metrika.xml</include_from>
```

2.3.3 在三台机器的/etc/clickhouse-server/config.d目录下新建metrika.xml文件

[root@hdp-1 config.d]# vim metrika.xml

添加如下内容：

注意：标签中的内容对应自己的主机名

```
<yandex>
    <clickhouse_remote_servers>
        <perftest_3shards_1replicas>
            <shard>
                <internal_replication>true</internal_replication>
                <replica>
                    <host>hdp-1</host>
                    <port>9000</port>
                </replica>
            </shard>
            <shard>
                <replica>
                    <internal_replication>true</internal_replication>
                    <host>hdp-2</host>
                    <port>9000</port>
                </replica>
            </shard>
            <shard>
                <internal_replication>true</internal_replication>
                <replica>
                    <host>hdp-3</host>
                    <port>9000</port>
                </replica>
            </shard>
        </perftest_3shards_1replicas>
    </clickhouse_remote_servers>

    <zookeeper-servers>
        <node index="1">
```

```
            <host>hdp-1</host>
            <port>2181</port>
        </node>
        <node index="2">
            <host>hdp-2</host>
            <port>2181</port>
        </node>
        <node index="3">
            <host>hdp-3</host>
            <port>2181</port>
        </node>
    </zookeeper-servers>

    <macros>
        <shard>01</shard>
        <replica>hdp-1</replica>
    </macros>

    <networks>
        <ip>::/0</ip>
    </networks>

    <clickhouse_compression>
        <case>
            <min_part_size>10000000000</min_part_size>
            <min_part_size_ratio>0.01</min_part_size_ratio>
            <method>lz4</method>
        </case>
    </clickhouse_compression>

</yandex>
```

注意：需要根据机器名字不同去修改 hdp-1

```
<macros>
<replica>hdp-1</replica>
</macros>
```

2.3.4 三台机器启动ClickServer

首先在三台机器开启Zookeeper

前台启动：

[root@hdp-1 ~]# sudo -u clickhouse clickhouse-server --config-file=/etc/clickhouse-server/config.xml

注意：此方式为clickhouse官方提供的启动方式

另外也可以：

service clickhouse-server start或者systemctl start clickhouse-server方式启动

几种方式首选官方提供的方式。

后台启动：

[root@hdp-1 ~]# nohup sudo -u clickhouse clickhouse-server --config-file=/etc/clickhouse-server/config.xml >null 2>&1 &

第三部分 数据类型

支持DML，

为了提高性能，较传统数据库而言，clickhouse提供了复合数据类型。

ClickHouse的Upadate和Delete是由Alter变种实现。

3.1 整型

固定长度的整型，包括**有符号整型**或**无符号整型。**

整型范围（-2n-1~2n-1-1）：

Int8	**[-128 : 127]**	**1**
Int16	- [-32768 : 32767]	2
Int32	[-2147483648 : 2147483647]	3
Int64	[-9223372036854775808 : 9223372036854775807]	4
名称	范围	大小（字节）

无符号整型范围（0~2n-1）：

名称	**范围**	**大小（字节）**
UInt8	[0 : 255]	1
UInt16[0 : 65535]	[0 : 65535]	2
UInt32	[0 : 4294967295]	3
UInt64	[0 : 18446744073709551615]	4

3.2 浮点型

名称	**大小（字节）**	**有效精度（位数）**	**传统**
Float32	4	7	Float
Float64	8	16	Double

建议尽可能以整数形式存储数据。例如，将固定精度的数字转换为整数值，如时间用毫秒为单位表示，因为浮点型进行计算时可能引起四舍五入的误差。

```
:) select 1-0.9

┌────────minus(1, 0.9)─┐

| 0.09999999999999998 |

└──────────────────────┘
```

与标准SQL相比，ClickHouse 支持以下类别的浮点数：

Inf-正无穷：

```
:) select 1/0

┌─divide(1, 0)─┐

|      inf |

└──────────────┘
```

-Inf-负无穷：

```
:) select -1/0

┌─divide(1, 0)─┐

|      -inf |

└──────────────┘
```

NaN-非数字：

```
:) select 0/0

┌─divide(0, 0)─┐

|      nan |

└──────────────┘
```

3.3 Decimal

如果要求更高精度，可以选择Decimal类型

格式：Decimal（P,S)

P:代表精度，决定总位数（正数部分+小数部分），取值范围0-38

S:代表规模，决定小数位数，取值范围是0-P

ClickHouse对Decimal提供三种简写：

Decimal32，Decimal64，Decimal128

3.3.1 相加、减精度取大

```
SELECT toDecimal32(2, 4) + toDecimal32(2, 2)

┌─plus(toDecimal64(2, 4), toDecimal32(2, 2))─┐
│                                     4.0000 │
└────────────────────────────────────────────┘



SELECT toDecimal32(4, 4) - toDecimal32(2, 2)

┌─minus(toDecimal32(4, 4), toDecimal32(2, 2))─┐
│                                      2.0000 │
└─────────────────────────────────────────────┘
```

3.3.2 相乘精度取和

```
SELECT toDecimal32(2, 2) * toDecimal32(4, 4)

┌─multiply(toDecimal32(2, 2), toDecimal32(4, 4))─┐
│                                       8.000000 │
└────────────────────────────────────────────────┘
```

3.3.3 相除精度取被除数

```
SELECT toDecimal32(4, 4) / toDecimal32(2, 2)

┌─divide(toDecimal32(4, 4), toDecimal32(2, 2))─┐
│                                       2.0000 │
└──────────────────────────────────────────────┘
```

3.4 字符串

String

字符串可以任意长度的。它可以包含任意的字节集，包含空字节。

FixedString(N)

固定长度 N 的字符串，N 必须是严格的正自然数。当服务端读取长度小于 N 的字符串时候，通过在字符串末尾添加空字节来达到 N 字节长度。 当服务端读取长度大于 N 的字符串时候，将返回错误消息。

```
SELECT
    toFixedString('abc', 5),
    LENGTH(toFixedString('abc', 5)) AS LENGTH

┌─toFixedString('abc', 5)─┬─LENGTH─┐
| abc                     |      5 |
└─────────────────────────┴────────┘
```

UUID

ClickHouse将UUID这种在传统数据库中充当主键的类型直接做成了数据类型

```
CREATE TABLE UUID_TEST
(
    `c1` UUID,
    `c2` String
)
ENGINE = Memory
```

插入数据：

```
insert into UUID_TEST select generateUUIDv4(),'t1';
```

```
insert into UUID_TEST(c2) values('t2');
```

查询结果：

```
SELECT *
FROM UUID_TEST

┌──────────────────────────────────────c1─┬─c2─┐
| 7508ccbd-16b9-451d-b74e-3768bc6b91a9 | t1 |
└──────────────────────────────────────┴────┘
┌──────────────────────────────────────c1─┬─c2─┐
| 00000000-0000-0000-0000-000000000000 | t2 |
└──────────────────────────────────────┴────┘
```

3.5 枚举类型

包括 Enum8 和 Enum16 类型。Enum 保存 'string'= integer 的对应关系。

Enum8 用 'String'= Int8 对描述。

Enum16 用 'String'= Int16 对描述。

用法演示：

创建一个带有一个枚举 Enum8('hello' = 1, 'world' = 2) 类型的列：

```
CREATE TABLE t_enum

(

  x Enum8('hello' = 1, 'world' = 2)

)

ENGINE = TinyLog
```

这个 x 列只能存储类型定义中列出的值：'hello'或'world'。如果尝试保存任何其他值，ClickHouse 抛出异常。

```
:) INSERT INTO t_enum VALUES ('hello'), ('world'), ('hello')
```

```
INSERT INTO t_enum VALUES

Ok.

3 rows in set. Elapsed: 0.002 sec.

:) insert into t_enum values('a')

INSERT INTO t_enum VALUES

Exception on client:

Code: 49. DB::Exception: Unknown element 'a' for type Enum8('hello' = 1, 'world' = 2)
```

从表中查询数据时，ClickHouse 从 Enum 中输出字符串值。

```
SELECT * FROM t_enum

┌─x─────┐

| hello |

| world |

| hello |

└───────┘
```

如果需要看到对应行的数值，则必须将 Enum 值转换为整数类型。

```
SELECT CAST(x, 'Int8') FROM t_enum



┌─CAST(x, 'Int8')─┐

|        1 |

|        2 |

|        1 |

└─────────────────┘
```

为什么需要枚举类型?

后续对枚举的操作：排序、分组、去重、过滤等，会使用Int类型的Value值

3.6 数组

Array(T)：由 T 类型元素组成的数组。

T 可以是任意类型，包含数组类型。 但不推荐使用多维数组，ClickHouse 对多维数组的支持有限。例如，不能在 MergeTree 表中存储多维数组。

可以使用array函数来创建数组：

array(T)

也可以使用方括号：ClickHouse能够自动推断数据类型

[]

创建数组案例：

```
:) SELECT array(1, 2.0) AS x, toTypeName(x)      --数组中可以有不同的数据类型，但是需要相互兼容。

┌─x─────┬─toTypeName(array(1, 2))─┐

| [1,2] |                Array(UInt8)       |

└───────┴────────────────────────────┘

1 rows in set. Elapsed: 0.002 sec.

:) SELECT [1, 2] AS x, toTypeName(x)

┌─x─────┬─toTypeName([1, 2])─┐

| [1,2] | Array(UInt8)    |

└───────┴────────────────────┘

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

如果是声明表字段的时候，需要指明数据类型：

```
CREATE TABLE Array_test
(
    `c1` Array(String)
)
ENGINE = memory
```

3.7 元组

Tuple(T1, T2, ...)：元组，其中每个元素都有单独的类型。

创建元组的示例：

```
:) SELECT tuple(1,'a') AS x, toTypeName(x)


SELECT

  (1, 'a') AS x,

  toTypeName(x)


┌─x───────┬─toTypeName(tuple(1, 'a'))─┐

| (1,'a') | Tuple(UInt8, String)    |

└─────────┴───────────────────────────┘

1 rows in set. Elapsed: 0.021 sec.
```

在定义表字段的时候也需要指明数据类型

3.8 Date、DateTime

日期类型，用两个字节存储，表示从 1970-01-01 (无符号) 到当前的日期值。

3.9 布尔型

没有单独的类型来存储布尔值。可以使用 UInt8 类型，取值限制为 0 或 1。

第4部分 表引擎

表引擎（即表的类型）决定了：

1）数据的存储方式和位置，写到哪里以及从哪里读取数据

2）支持哪些查询以及如何支持。

3）并发数据访问。

4）索引的使用（如果存在）。

5）是否可以执行多线程请求。

6）数据复制参数。

ClickHouse的表引擎有很多，下面介绍其中几种，对其他引擎有兴趣的可以去查阅官方文档：https://clickhouse.yandex/docs/zh/operations/table_engines/

4.1 日志

4.1.1 TinyLog

最简单的表引擎，用于将数据存储在磁盘上。每列都存储在单独的压缩文件中，写入时，数据将附加到文件末尾。

该引擎没有并发控制

- 如果同时从表中读取和写入数据，则读取操作将抛出异常；
- 如果同时写入多个查询中的表，则数据将被破坏。

这种表引擎的典型用法是 write-once：首先只写入一次数据，然后根据需要多次读取。此引擎适用于相对较小的表（建议最多1,000,000行）。如果有许多小表，则使用此表引擎是适合的，因为它需要打开的文件更少。当拥有大量小表时，可能会导致性能低下。 不支持索引。

案例：创建一个TinyLog引擎的表并插入一条数据

```
:)create table t (a UInt16, b String) ENGINE=TinyLog;

:)insert into t (a, b) values (1, 'abc');
```

此时我们到保存数据的目录/var/lib/clickhouse/data/default/t中可以看到如下目录结构：

[root@hdp-2]# ls

a.bin b.bin sizes.json

a.bin 和 b.bin 是压缩过的对应的列的数据， sizes.json 中记录了每个 *.bin 文件的大小：

[root@hdp-2]# cat sizes.json

{"yandex":{"a%2Ebin":{"size":"28"},"b%2Ebin":{"size":"30"}}}

4,1.2 Log

Log与 TinyLog 的不同之处在于，«标记» 的小文件与列文件存在一起。这些标记写在每个数据块上，并且包含偏移量，这些偏移量指示从哪里开始读取文件以便跳过指定的行数。这使得可以在多个线程中读取表数据。对于并发数据访问，可以同时执行读取操作，而写入操作则阻塞读取和其它写入。Log 引擎不支持索引。同样，如果写入表失败，则该表将被破坏，并且从该表读取将返回错误。Log 引擎适用于临时数据，write-once 表以及测试或演示目的。

4.1.3 StripeLog

该引擎属于日志引擎系列。请在日志引擎系列文章中（https://clickhouse.tech/docs/zh/engines/table-engines/log-family/）查看引擎的共同属性和差异。

在你需要写入许多小数据量（小于一百万行）的表的场景下使用这个引擎。

建表

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    column1_name [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    column2_name [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = StripeLog
```

查看建表请求的详细说明。(https://clickhouse.tech/docs/zh/engines/table-engines/log-family/stripelog/#create-table-query)

写数据

StripeLog 引擎将所有列存储在一个文件中。对每一次 Insert 请求，ClickHouse 将数据块追加在表文件的末尾，逐列写入。

ClickHouse 为每张表写入以下文件：

- data.bin — 数据文件。
- index.mrk — 带标记的文件。标记包含了已插入的每个数据块中每列的偏移量。

StripeLog 引擎不支持 ALTER UPDATE 和 ALTER DELETE 操作。

读数据

带标记的文件使得 ClickHouse 可以并行的读取数据。这意味着 SELECT 请求返回行的顺序是不可预测的。使用 ORDER BY 子句对行进行排序。

使用示例

建表：

```
CREATE TABLE stripe_log_table
(
    timestamp DateTime,
    message_type String,
    message String
)
ENGINE = StripeLog
```

插入数据：

```
INSERT INTO stripe_log_table VALUES (now(),'REGULAR','The first regular message')
INSERT INTO stripe_log_table VALUES (now(),'REGULAR','The second regular message'),
(now(),'WARNING','The first warning message')
```

我们使用两次 INSERT 请求从而在 data.bin 文件中创建两个数据块。

ClickHouse 在查询数据时使用多线程。每个线程读取单独的数据块并在完成后独立的返回结果行。这样的结果是，大多数情况下，输出中块的顺序和输入时相应块的顺序是不同的。例如：

```
SELECT * FROM stripe_log_table
┌─────────────timestamp─┬─message_type─┬─message────────────────────┐
│ 2019-01-18 14:27:32 │ REGULAR      │ The second regular message │
│ 2019-01-18 14:34:53 │ WARNING      │ The first warning message  │
└─────────────────────┴──────────────┴────────────────────────────┘

┌─────────────timestamp─┬─message_type─┬─message───────────────────┐
│ 2019-01-18 14:23:43 │ REGULAR      │ The first regular message │
└─────────────────────┴──────────────┴───────────────────────────┘
```

对结果排序（默认增序）：

```
SELECT * FROM stripe_log_table ORDER BY timestamp
┌─────────────timestamp─┬─message_type─┬─message────────────────────┐
│ 2019-01-18 14:23:43 │ REGULAR      │ The first regular message  │
│ 2019-01-18 14:27:32 │ REGULAR      │ The second regular message │
│ 2019-01-18 14:34:53 │ WARNING      │ The first warning message  │
└─────────────────────┴──────────────┴────────────────────────────┘
```

4.2 Memory

内存引擎，数据以未压缩的原始形式直接保存在内存当中，服务器重启数据就会消失。读写操作不会相互阻塞，不支持索引。简单查询下有非常非常高的性能表现（超过10G/s）。

一般用到它的地方不多，除了用来测试，就是在需要非常高的性能，同时数据量又不太大（上限大概 1 亿行）的场景。

4.3 Merge

Merge 引擎 (不要跟 MergeTree 引擎混淆) 本身不存储数据，但可用于同时从任意多个其他的表中读取数据。 读是自动并行的，不支持写入。读取时，那些被真正读取到数据的表的索引（如果有的话）会被使用。

Merge 引擎的参数：一个数据库名和一个用于匹配表名的正则表达式。

案例：先建t1，t2，t3三个表，然后用 Merge 引擎的 t 表再把它们链接起来。

```
:)create table t1 (id UInt16, name String) ENGINE=TinyLog;

:)create table t2 (id UInt16, name String) ENGINE=TinyLog;

:)create table t3 (id UInt16, name String) ENGINE=TinyLog;
```

```
:)insert into t1(id, name) values (1, 'first');

:)insert into t2(id, name) values (2, 'second');

:)insert into t3(id, name) values (3, 'i am in t3');
```

```
:)create table t (id UInt16, name String) ENGINE=Merge(currentDatabase(), '^t');
```

```
:) select * from t;

┌─id─┬─name─┐

|  2 | second |

└────┴──────┘

┌─id─┬─name──┐

|  1 | first |

└────┴───────┘

┌─id─┬─name─────┐

| 3      | i am in t3 |

└────┴──────────┘
```

4.4 MergeTree

Clickhouse 中最强大的表引擎当属 MergeTree （合并树）引擎及该系列（*MergeTree）中的其他引擎。

MergeTree 引擎系列的基本理念如下。当你有巨量数据要插入到表中，你要高效地一批批写入数据片段，并希望这些数据片段在后台按照一定规则合并。相比在插入时不断修改（重写）数据进存储，这种策略会高效很多。

4.4.1MergeTree的创建方式与存储结构

4.4.1.1 MergeTree的创建方式：

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1] [TTL expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2] [TTL expr2],
    ...
    INDEX index_name1 expr1 TYPE type1(...) GRANULARITY value1,
    INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2
) ENGINE = MergeTree()
ORDER BY expr
[PARTITION BY expr]
[PRIMARY KEY expr]
[SAMPLE BY expr]
[TTL expr [DELETE|TO DISK 'xxx'|TO VOLUME 'xxx'], ...]
```

```
[SETTINGS name=value, ...]
```

案例:

```
create table mt_table(date Date,id UInt8,name String) engine = MergeTree partition by
toYYYYMM(date) order by id;

CREATE TABLE mt_table3
(
    `date` Date,
    `id` UInt8,
    `name` String
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(date)
ORDER BY id
```

```
insert into mt_table values ('2019-05-01', 1, 'zhangsan');

insert into mt_table values ('2019-06-01', 2, 'lisi');

insert into mt_table values ('2019-05-03', 3, 'wangwu');
```

在/var/lib/clickhouse/data/default/mt_tree下可以看到:

ls

20190501_20190501_2_2_0 20190503_20190503_6_6_0 20190601_20190601_4_4_0 detached

---

随便进入一个目录:

ls

checksums.txt columns.txt date.bin date.mrk id.bin id.mrk name.bin name.mrk primary.idx

- *.bin是按列保存数据的文件
- *.mrk保存块偏移量
- primary.idx保存主键索引

[PARTITION BY expr]

[PRIMARY KEY expr]

[SAMPLE BY expr]

[TTL expr [DELETE|TO DISK 'xxx'|TO VOLUME 'xxx'], ...]

[SETTINGS name=value, ...]

- ENGINE - 引擎名和参数。 ENGINE = MergeTree(). MergeTree 引擎没有参数。

- PARTITION BY — 分区键 。

  要按月分区，可以使用表达式 `toYYYYMM(date_column)` ，这里的 `date_column` 是一个 Date 类型的列。这里该分区名格式会是 `"YYYYMM"` 这样。

- ORDER BY — 表的排序键。必选!

  可以是一组列的元组或任意的表达式。 例如: `ORDER BY (CounterID, EventDate)` 。

- PRIMARY KEY - 主键，如果要设成 跟排序键不相同。

  默认情况下主键跟排序键（由 `ORDER BY` 子句指定）相同。 因此，大部分情况下不需要再专门指定一个 `PRIMARY KEY` 子句。

- SAMPLE BY — 用于抽样的表达式。

  如果要用抽样表达式，主键中必须包含这个表达式。例如： `SAMPLE BY intHash32(UserID) ORDER BY (CounterID, EventDate, intHash32(UserID))` 。

- SETTINGS — 影响 MergeTree 性能的额外参数:
  - index_granularity — 索引粒度。即索引中相邻『标记』间的数据行数。默认值，8192 。该列表中所有可用的参数可以从这里查看 MergeTreeSettings.h
  - index_granularity_bytes — 索引粒度，以字节为单位，默认值: 10Mb。如果仅按数据行数限制索引粒度, 请设置为0(不建议)。
  - enable_mixed_granularity_parts — 启用或禁用通过 index_granularity_bytes 控制索引粒度的大小。在19.11版本之前, 只有 index_granularity 配置能够用于限制索引粒度的大小。当从大表(数十或数百兆)中查询数据时候，index_granularity_bytes 配置能够提升ClickHouse的性能。如果你的表内数据量很大，可以开启这项配置用以提升SELECT 查询的性能。
  - use_minimalistic_part_header_in_zookeeper — 数据片段头在 ZooKeeper 中的存储方式。如果设置了 use_minimalistic_part_header_in_zookeeper=1 ， ZooKeeper 会存储更少的数据。更多信息参考『服务配置参数』这章中的 设置描述 。
  - min_merge_bytes_to_use_direct_io — 使用直接 I/O 来操作磁盘的合并操作时要求的最小数据量。合并数据片段时，ClickHouse 会计算要被合并的所有数据的总存储空间。如果大小超过了 min_merge_bytes_to_use_direct_io 设置的字节数，则 ClickHouse 将使用直接 I/O 接口（O_DIRECT 选项）对磁盘读写。如果设置 min_merge_bytes_to_use_direct_io = 0 ，则会禁用直接 I/O。默认值：10 * 1024 * 1024 * 1024 字节。
  - merge_with_ttl_timeout — TTL合并频率的最小间隔时间。默认值: 86400 (1 天)。
  - write_final_mark — 启用或禁用在数据片段尾部写入最终索引标记。默认值: 1（不建议更改）。
  - storage_policy — 存储策略。 参见 使用多个区块装置进行数据存储.

4.4.1.2 MergeTree的存储结构

```
-- mt_table
   |-- 20180301_20180330_1_100_20
   |   |-- checksums.txt
   |   |-- columns.txt
   |   |-- date.bin
   |   |-- date.mrk2
   |   |-- id.bin
   |   |-- id.mrk2
   |   |-- name.bin
   |   |-- name.mrk2
   |   `-- primary.idx
   |-- 20180601_20180629_101_200_20
```

```
|   |-- checksums.txt
|   |-- columns.txt
|   |-- date.bin
|   |-- date.mrk2
|   |-- id.bin
|   |-- id.mrk2
|   |-- name.bin
|   |-- name.mrk2
|   `-- primary.idx
|-- detached
 -- format_version.txt
```

`checksums.txt`

二进制的校验文件，保存了余下文件的大小size和size的Hash值，用于快速校验文件的完整和正确性

`columns.txt`

明文的列信息文件，如：

```
[root@hdp-1 20180301_20180330_1_100_20]# cat columns.txt
columns format version: 1
3 columns:
`date` Date
`id` UInt8
`name` String
```

`date.bin`

压缩格式（默认LZ4）的数据文件，保存了原始数据。以列名.bin命名。

`date.mrk2`

使用了自适应大小的索引间隔,名字为 .mrk2

二进制的列字段标记文件，作用是把稀疏索引.idx文件和存数据的文件.bin联系起来。详见数据标记一节

id.bin id.mrk2 name.bin name.mrk2

`primary.idx`

二进制的一级索引文件，在建表的时候通过OrderBy或者PrimaryKey声明的稀疏索引。

4.4.2、数据分区

数据是以分区目录的形式组织的，每个分区独立分开存储。

这种形式，查询数据时，可以有效的跳过无用的数据文件。

4.4.2.1 数据分区的规则

分区键的取值，生成分区ID，分区根据ID决定。根据分区键的数据类型不同，分区ID的生成目前有四种规则：

（1）不指定分区键

（2）使用整形

（3）使用日期类型 toYYYYMM(date)

（4）使用其他类型

数据在写入时，会对照分区ID落入对应的分区。

4.4.2.2 分区目录的生成规则

partitionID_MinBlockNum_MaxBlockNum_Level

BlockNum是一个全局整型，从1开始，每当新创建一个分区目录，此数字就累加1。

MinBlockNum:最小数据块编号

MaxBlockNum:最大数据块编号

对于一个新的分区，MinBlockNum和MaxBlockNum的值相同

如：2020_03_1_1_0, 2020_03_2_2_0

Level:合并的层级，即某个分区被合并过得次数。不是全局的，而是针对某一个分区。

4.4.2.3 分区目录的合并过程

MergeTree的分区目录在数据写入过程中被创建。

不同的批次写入数据属于同一分区，也会生成不同的目录，在之后的某个时刻再合并（写入后的10-15分钟），合并后的旧分区目录默认8分钟后删除。

同一个分区的多个目录合并以后的命名规则：

- MinBlockNum:取同一分区中MinBlockNum值最小的
- MaxBlockNum：取同一分区中MaxBlockNum值最大的
- Level:取同一分区最大的Level值加1

4.4.3、索引

4.4.3.1 一级索引

文件：primary.idx

MergeTree的主键使用Primary Key定义，主键定义之后，MergeTree会根据index_granularity间隔（默认8192)为数据生成一级索引并保存至primary.idx文件中。这种方式是稀疏索引

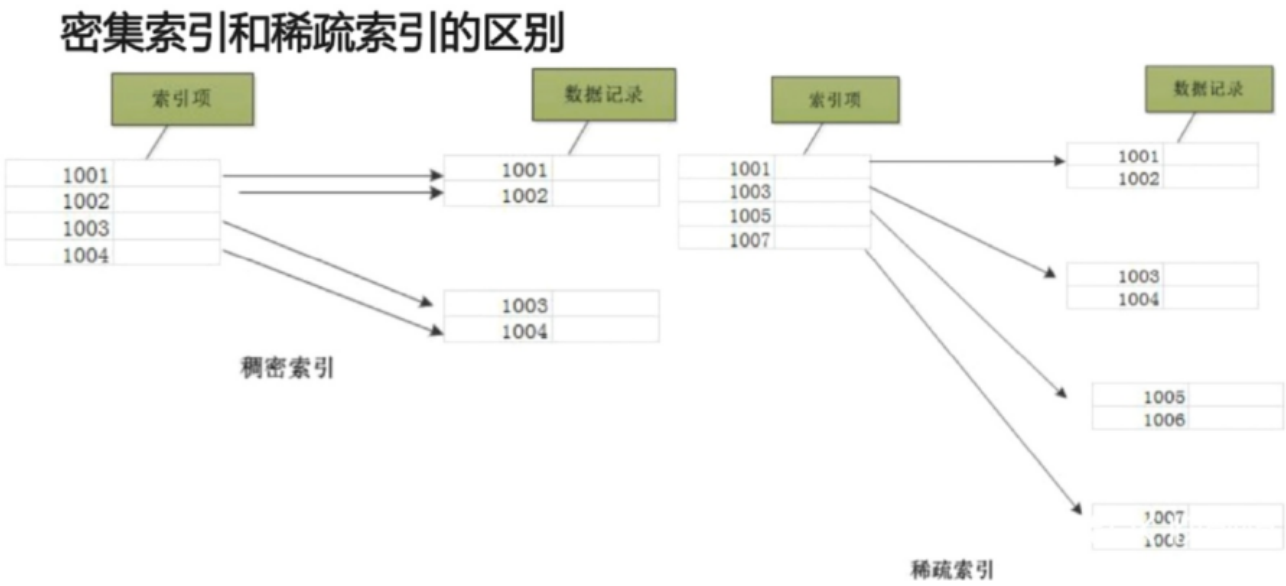
简化形式：通过order by指代主键

4.4.3.1.1 稀疏索引

primary.idx文件的一级索引采用稀疏索引。

稠密索引：每一行索引标记对应一行具体的数据记录

稀疏索引：每一行索引标记对应一段数据记录（默认索引粒度为8192）



稀疏索引占用空间小，所以primary.idx内的索引数据常驻内存，取用速度快！

4.4.3.1.2 索引粒度

index_granularity参数，表示索引粒度。新版本中clickhouse提供了自适应索引粒度。

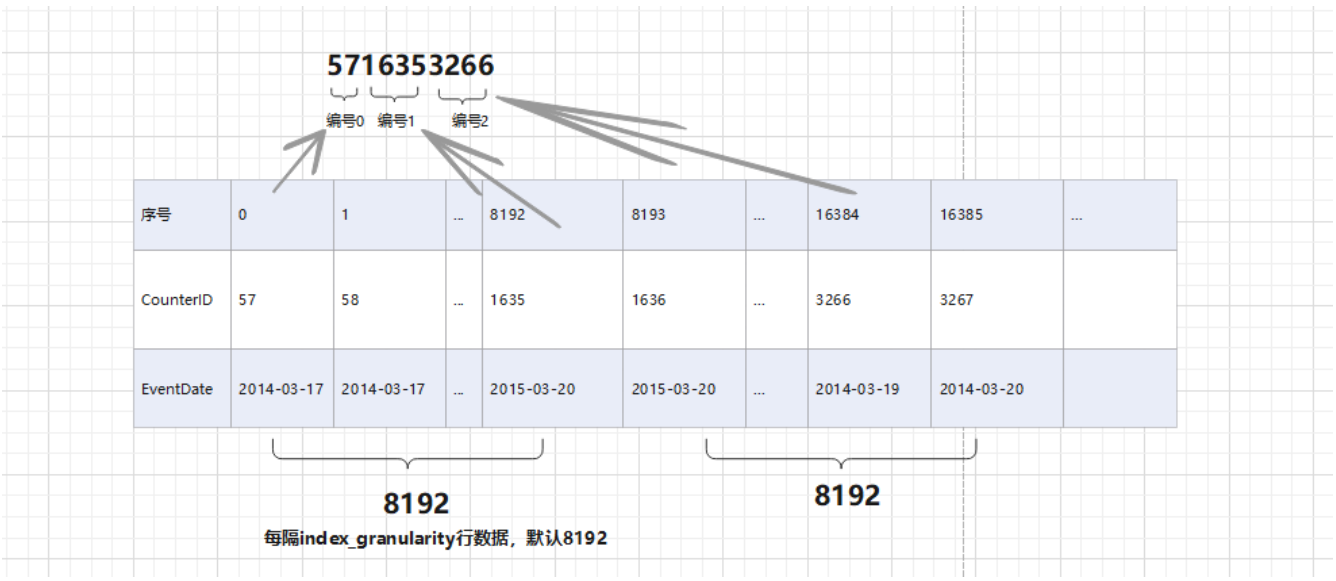
索引粒度在MergeTree引擎中很重要。

4.4.3.1.3 索引数据的生成规则

借助hits_v1表中的真实数据观察：

primary.idx文件

由于稀疏索引，所以MergeTree要间隔index_granularity行数据才会生成一个索引记录，其索引值会根据声明的主键字段获取。



4.4.3.1.4 索引的查询过程

索引是如何工作的？对primary.idx文件的查询过程

MarkRange：一小段数据区间

按照index_granularity的间隔粒度，将一段完整的数据划分成多个小的数据段，小的数据段就是MarkRange，

MarkRange与索引编号对应。

案例：

共200行数据

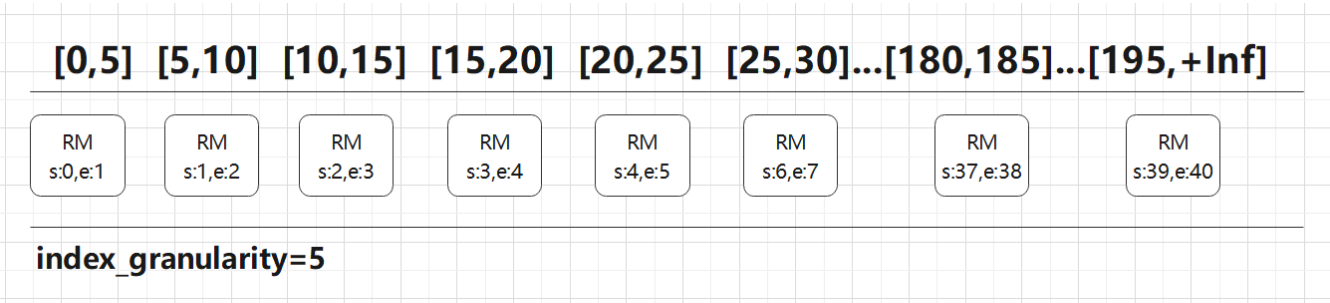
index_granularity大小为5

主键ID为Int，取值从0开始

根据索引生成规则，primary.idx文件内容为：

05101520253035404550...200

共200行数据 / 5 = 40个MarkRange

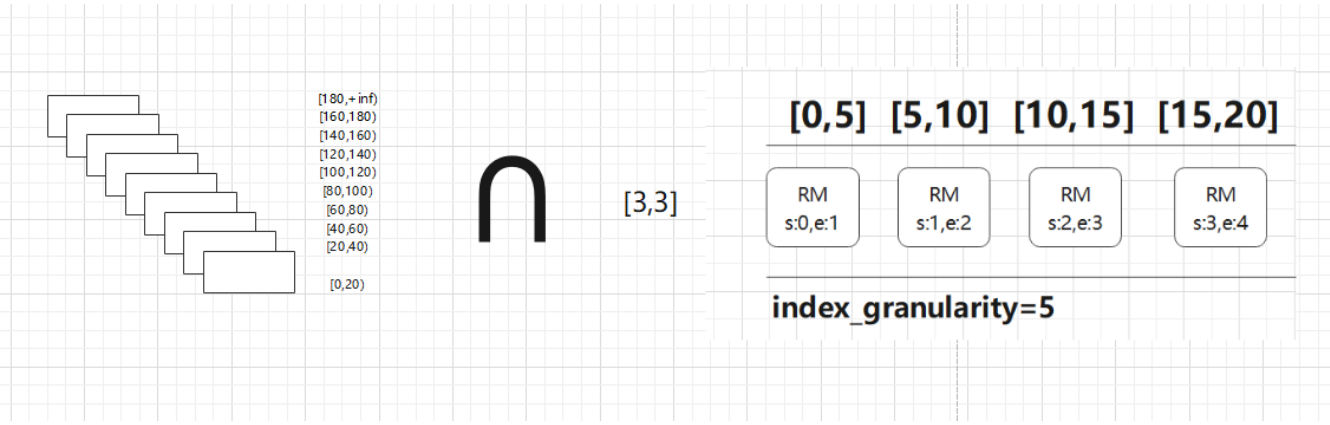


索引查询 where id = 3

第一步：形成区间格式： [3,3]

第二步：进行交集 [3,3]∩[0,199]

以MarkRange的步长大于8分块，进行剪枝



第三步：合并

MarkRange:(start0,end 20)

在ClickHouse中，MergeTree引擎表的索引列在建表时使用ORDER BY语法来指定。而在官方文档中，用了下面一幅图来说明。

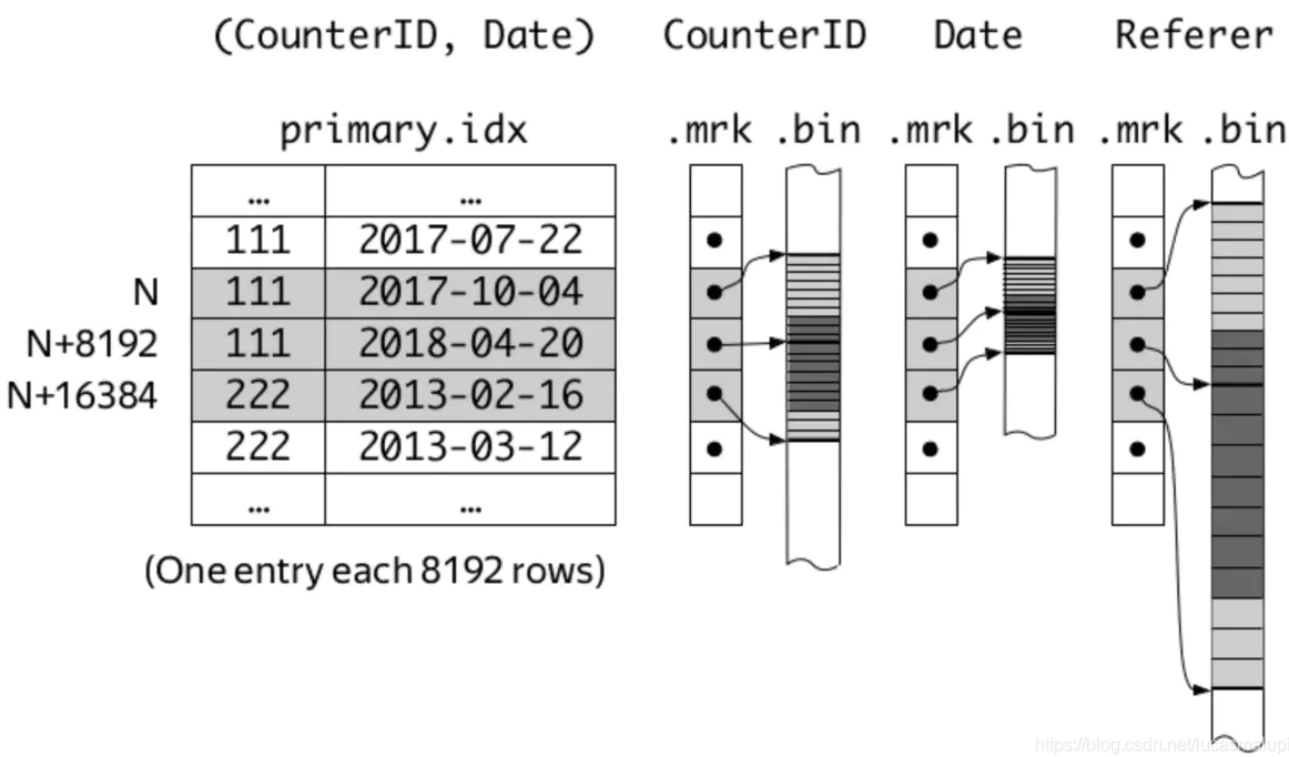
```
Whole data:     [-----------------------------------------]
CounterID:      [aaaaaaaaaaaaaaaaabbbbcdeeeeeeeeeeeeefggggggggghhhhhhhhhiiiiiiiiiklllllll]
Date:           [1111111222222233331233211111222222333211111112122222223111112223311122333]
Marks:           |      |      |      |      |      |      |      |      |      |      |
                a,1    a,2    a,3    b,3    e,2    e,3    g,1    h,2    i,1    i,3    l,3
Marks numbers:   0      1      2      3      4      5      6      7      8      9      10
```

这张图示出了以CounterID、Date两列为索引列的情况，即先以CounterID为主要关键字排序，再以Date为次要关键字排序，最后用两列的组合作为索引键。marks与mark numbers就是索引标记，且marks之间的间隔就由建表时的索引粒度参数index_granularity来指定，默认值为8192。

ClickHouse MergeTree引擎表中，每个part的数据大致以下面的结构存储。

. ├── business_area_id.bin ├── business_area_id.mrk2 ├── coupon_money.bin ├── coupon_money.mrk2 ├── groupon_id.bin ├── groupon_id.mrk2 ├── is_new_order.bin ├── is_new_order.mrk2 ... ├── primary.idx ... 其中，bin文件存储的是每一列的原始数据（可能被压缩存储），mrk2文件存储的是图中的mark numbers与bin文件中数据位置的映射关系。另外，还有一个primary.idx文件存储被索引列的具体数据。另外，每个part的数据都存储在单独的目录中，目录名形如20200708_92_121_7，即包含了分区键、起始mark number和结束mark number，方便定位。

在ClickHouse之父Alexey Milovidov分享的PPT中，有更加详细的图示。



这样，每一列都通过ORDER BY列进行了索引。查询时，先查找到数据所在的parts，再通过mrk2文件确定bin文件中数据的范围即可。

不过，ClickHouse的稀疏索引与Kafka的稀疏索引不同，可以由用户自由组合多列，因此也要格外注意不要加入太多索引列，防止索引数据过于稀疏，增大存储和查找成本。另外，基数太小（即区分度太低）的列不适合做索引列，因为很可能横跨多个mark的值仍然相同，没有索引的意义了。

4.4.3.2 跳数索引

4.4.3.2.1granularity和index_granularity的关系

index_granularity定义了数据的粒度 granularity定义了聚合信息汇总的粒度 换言之，granularity定义了一行跳数索引能够跳过多少个index_granularity区间的数据

4.4.3.2.2索引的可用类型

- minmax 存储指定表达式的极值（如果表达式是 tuple ，则存储 tuple 中每个元素的极值），这些信息用于跳过数据块，类似主键。
- set(max_rows) 存储指定表达式的惟一值（不超过 max_rows 个，max_rows=0 则表示『无限制』）。这些信息可用于检查 WHERE 表达式是否满足某个数据块。
- ngrambf_v1(n, size_of_bloom_filter_in_bytes, number_of_hash_functions, random_seed) 存储包含数据块中所有 n 元短语的 布隆过滤器 。只可用在字符串上。 可用于优化 equals， like 和 in 表达式的性能。 n – 短语长度。 size_of_bloom_filter_in_bytes – 布隆过滤器大小，单位字节。（因为压缩得好，可以指定比较大的值，如256或512）。 number_of_hash_functions – 布隆过滤器中使用的 hash 函数的个数。 random_seed – hash 函数的随机种子。
- tokenbf_v1(size_of_bloom_filter_in_bytes, number_of_hash_functions, random_seed) 跟 ngrambf_v1 类似，不同于 ngrams 存储字符串指定长度的所有片段。它只存储被非字母数据字符分割的片段。

```
INDEX sample_index (u64 * length(s)) TYPE minmax GRANULARITY 4
INDEX sample_index2 (u64 * length(str), i32 + f64 * 100, date, str) TYPE set(100)
GRANULARITY 4
INDEX sample_index3 (lower(str), str) TYPE ngrambf_v1(3, 256, 2, 0) GRANULARITY 4
```

4.4.4 、数据存储

表由按主键排序的数据 片段 组成。

当数据被插入到表中时，会分成数据片段并按主键的字典序排序。例如，主键是 (CounterID, Date) 时，片段中数据按 CounterID 排序，具有相同 CounterID 的部分按 Date 排序。

不同分区的数据会被分成不同的片段，ClickHouse 在后台合并数据片段以便更高效存储。不会合并来自不同分区的数据片段。这个合并机制并不保证相同主键的所有行都会合并到同一个数据片段中。

ClickHouse 会为每个数据片段创建一个索引文件，索引文件包含每个索引行（『标记』）的主键值。索引行号定义为 n * index_granularity 。最大的 n 等于总行数除以 index_granularity 的值的整数部分。对于每列，跟主键相同的索引行处也会写入『标记』。这些『标记』让你可以直接找到数据所在的列。

你可以只用一单一大表并不断地一块块往里面加入数据 – MergeTree 引擎的就是为了这样的场景

4.4.4.1 按列存储

在MergeTree中数据按列存储，具体到每个列字段，都拥有一个.bin数据文件，是最终存储数据的文件。 按列存储的好处： 1、更好的压缩 2、最小化数据扫描范围

MergeTree往.bin文件存数据的步骤：

1、对数据进行压缩

2、根据OrderBy排序

3、数据以压缩数据块的形式写入.bin文件

4.4.4.2 压缩数据块

CompressionMethod_CompressedSize_UnccompressedSize

一个压缩数据块有两部分组成：

1、头信息 2、压缩数据

头信息固定使用9位字节表示，1个UInt8（1字节）+2个UInt32(4字节)，分别表示压缩算法、压缩后数据大小、压缩前数据大小

如：0x821200065536

0x82:是压缩方法

12000：压缩后数据大小

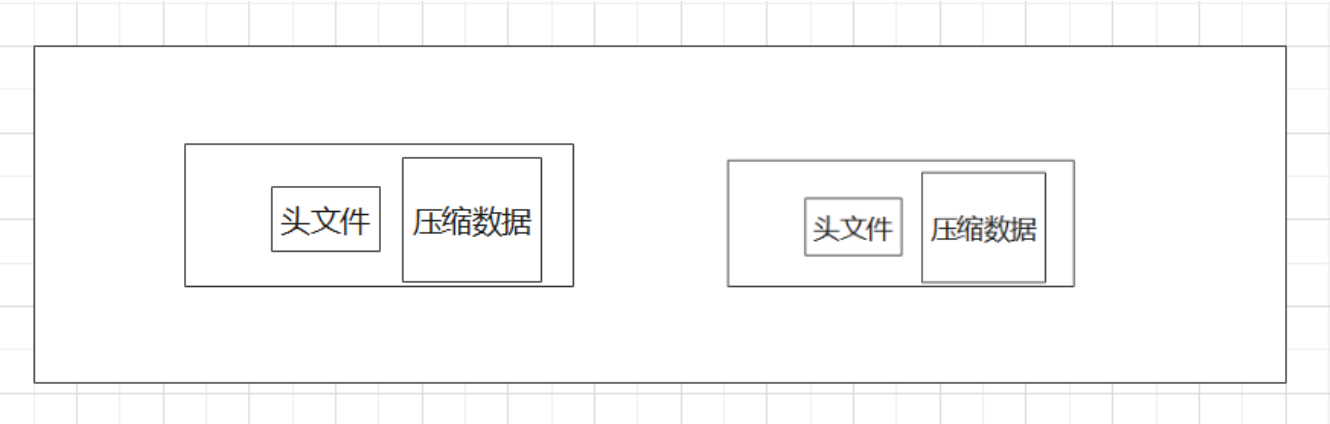
65536：压缩前数据大小

clickhouse-compressor --stat命令

```
[root@hdp-1 20180301_20180330_1_100_20]# clickhouse-compressor --stat <./date.bin > out.log
[root@hdp-1 20180301_20180330_1_100_20]# cat out.log
200     207
[root@hdp-1 20180301_20180330_1_100_20]#
```

out1.log文件中显示的数据前面的是压缩的，后面是未压缩的。

[Column].bin:



如果按照默认8192的索引粒度把数据分成批次，每批次读入数据的规则：

设x为批次数据的大小，

如果单批次获取的数据 x<64k,则继续读下一个批次，找到size>64k则生成下一个数据块

如果单批次数据64k<x<1M则直接生成下一个数据块

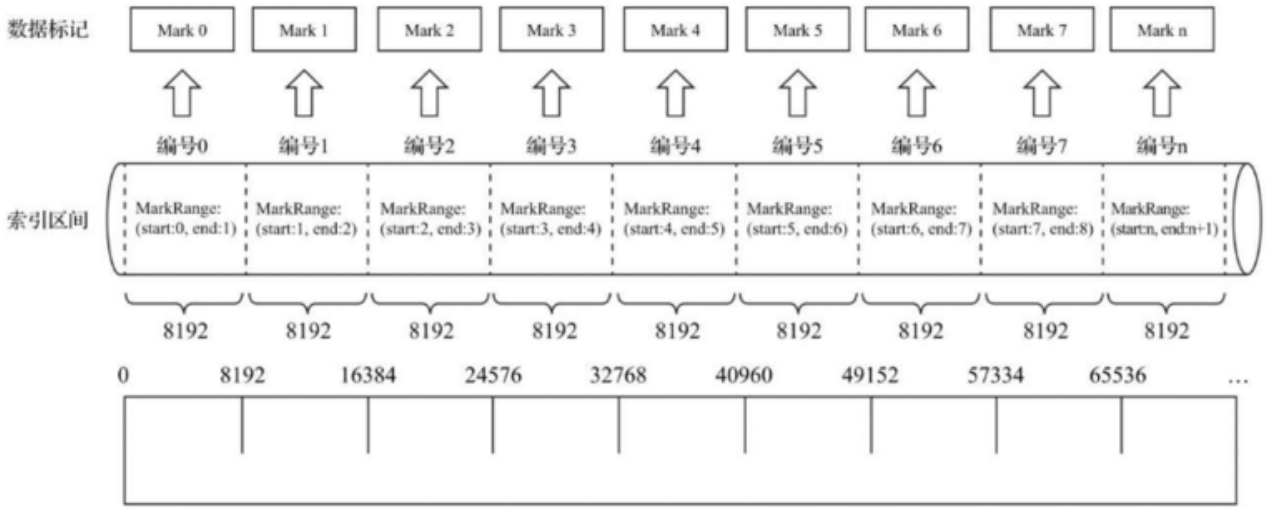
如果x>1M,则按照1M切分数据，剩下的数据继续按照上述规则执行。

4.4.5 数据标记

.mrk文件

将以及索引primary.idx和数据文件.bin建立映射关系

通用用hits_v1表说明：



1、数据标记和索引区间是对齐的，根据索引区间的下标编号，就能找到数据标记---索引编号和数据标记数值相同

2、每一个[Column].bin都有一个[Column].mrk与之对应---.mrk文件记录数据在.bin文件中的偏移量

拿JavaEnable字段说明：

1 b * 8192 = 8192b 8192b * 8 = 64k

.mrk（标记）

编号	压缩文件中的偏移量	解压缩块中的偏移量
0	0	0
1	0	8192
2	0	16384
3	0	24576
4	0	32768
5	0	40960
6	0	49152
7	0	57344
8	12016	0
9	12016	8192
…	…	…

4.4.5.1 .mrk文件内容的生成规则

数据标记和区间是对齐的。均按照index_granularity粒度间隔。可以通过索引区间的下标编号找到对应的数据标记。

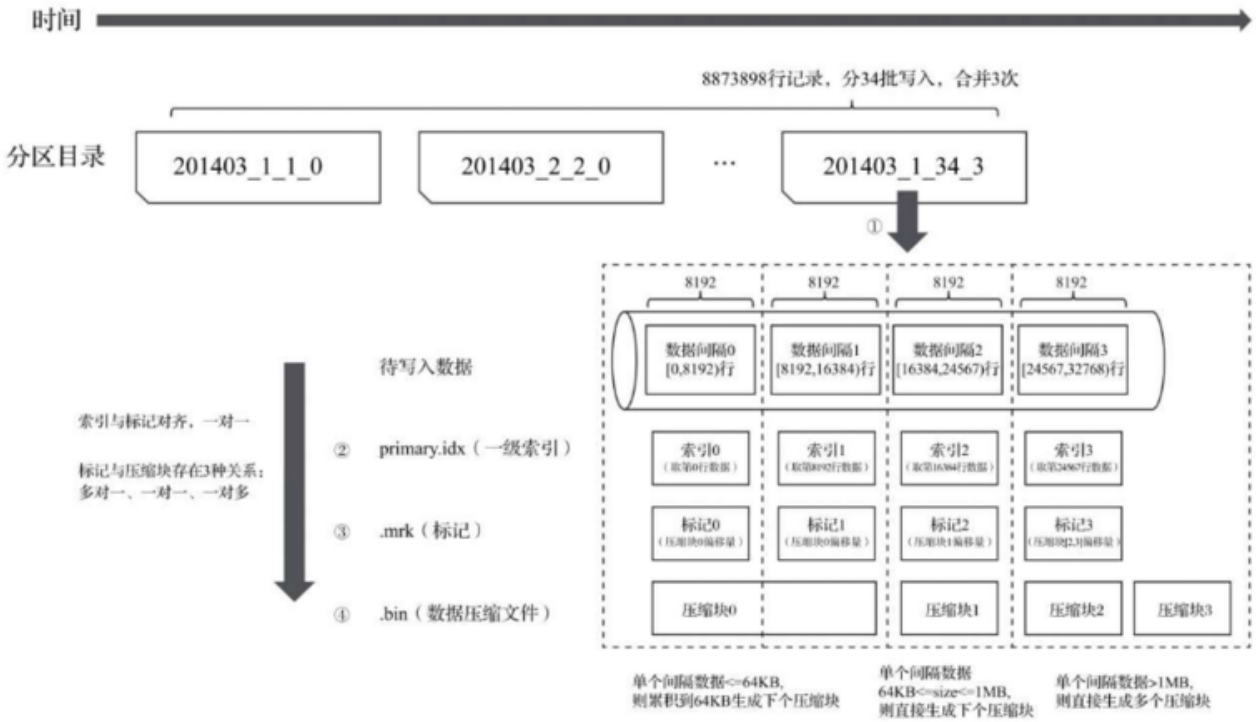
每一个列字段的.bin文件都有一个.mrk数据标记文件，用于记录数据在.bin文件中的偏移量信息。

标记数据采用LRU缓存策略加快其取用速度

4.4.5.2 .mrk文件的工作方式

4.4.6 分区、索引、标记和压缩协同

4.4.6.1 写入过程

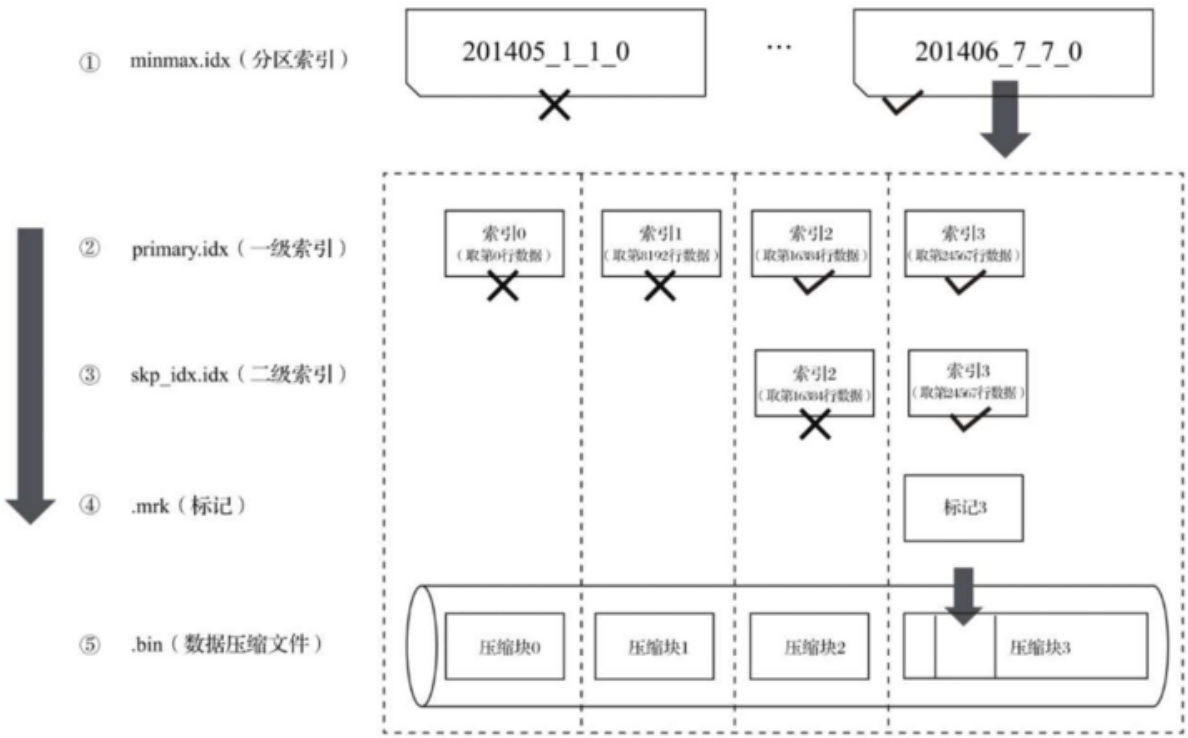


1、生成分区目录

2、合并分区目录

3、生成primary.idx索引文件、每一列的.bin和.mrk文件

4.4.6.2 查询过程



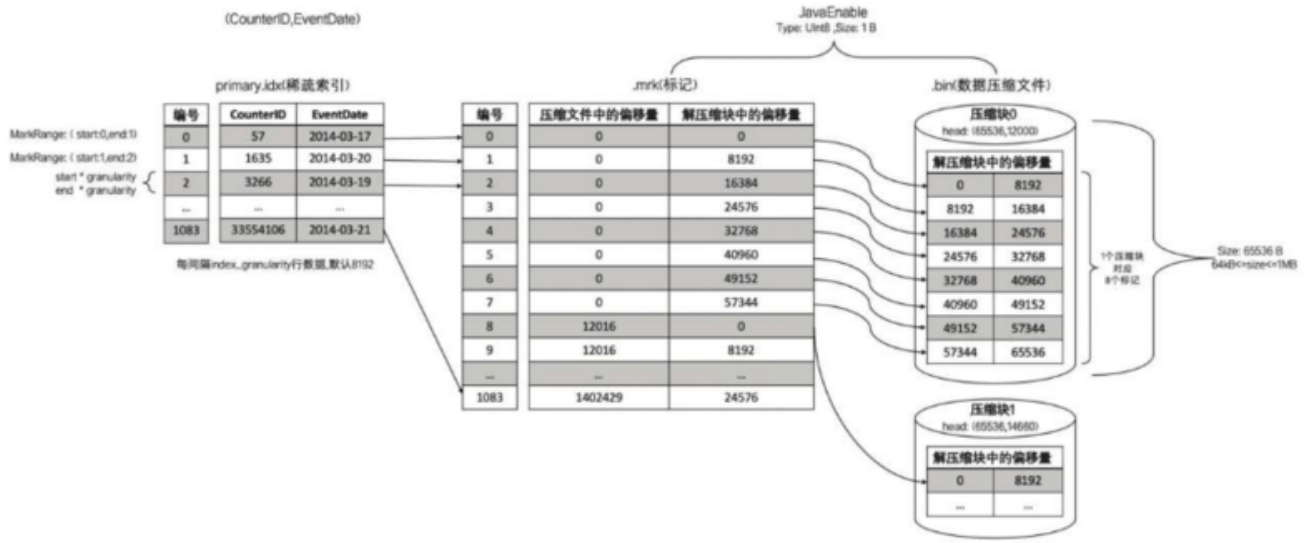
1、根据分区索引缩小查询范围

2、根据数据标记，缩小查询范围

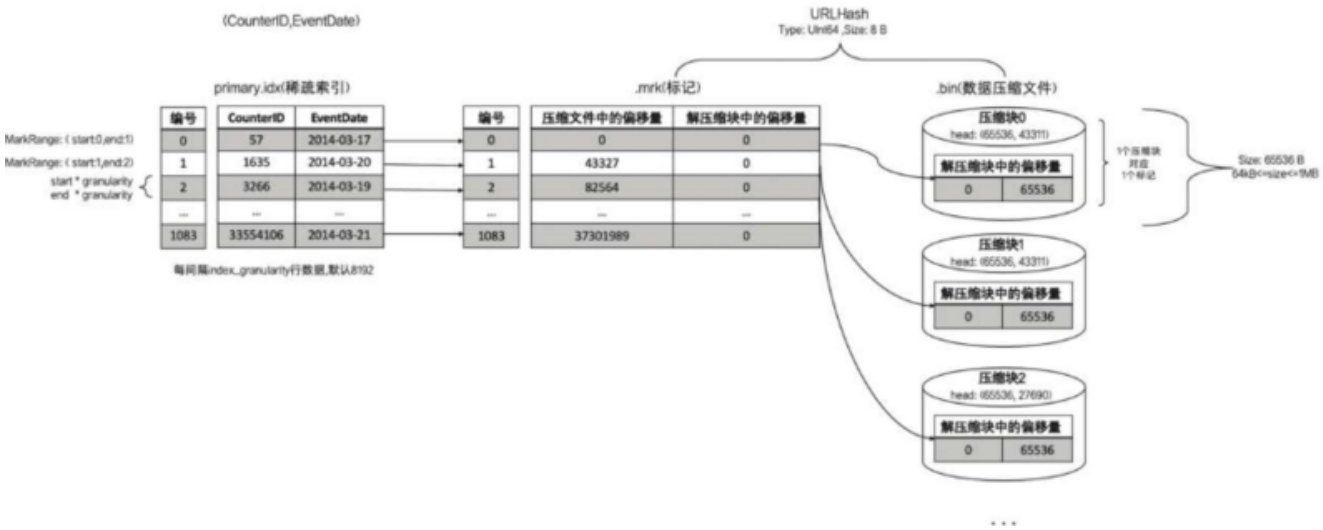
3、解压压缩块

4.4.6.3 数据标记与压缩数据块的对应关系

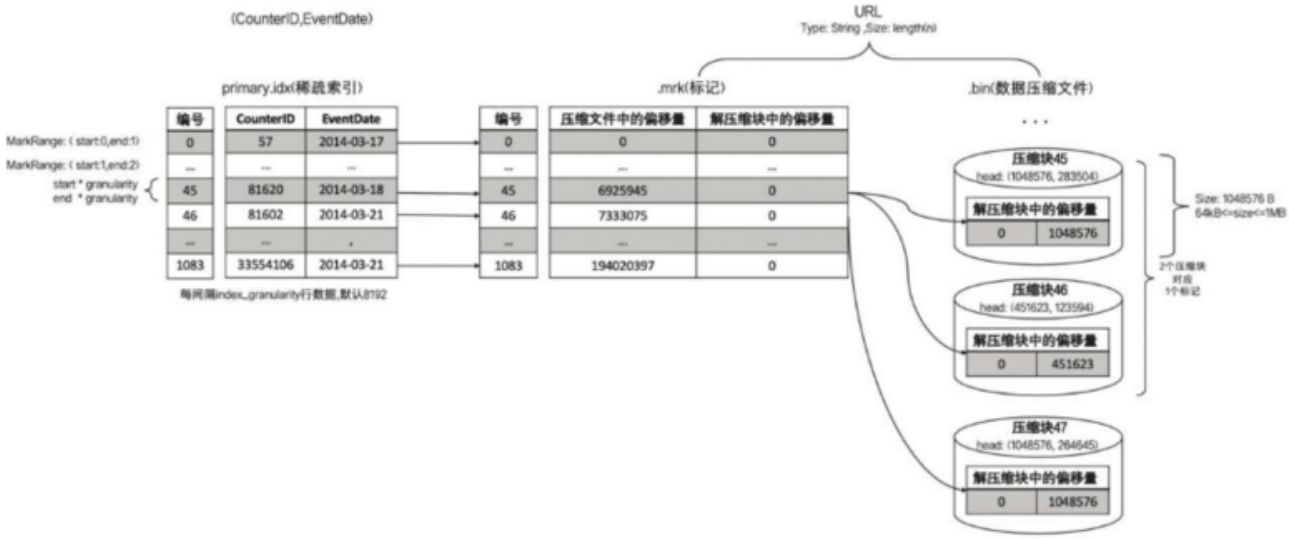
多对一：



一对一：



一对多：



4.4.7 MergTree的TTL

TTL：time to live 数据存活时间。TTL既可以设置在表上，也可以设置在列上。TTL指定的时间到期后则删除相应的表或列，如果同时设置了TTL，则根据先过期时间删除相应数据。

用法：

TTL time_col + INTERVAL 3 DAY

表示数据存活时间是time_col时间的3天后

INTERVAL可以设定的时间：SECOND MINUTE HOUR DAY WEEK MONTH QUARTER YEAR

4.4.7.1 TTL设置在列上

例：

```
create table ttl_table_v1 (
id String,
create_time DateTime,
code String TTL create_time + INTERVAL 10 SECOND,
type UInt8 TTL create_time + INTERVAL 10 SECOND
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(create_time)
ORDER BY id;
```

```
insert into ttl_table_v1 values('A000',now(),'C1',1),('A000',now()+INTERVAL 10
MINUTE,'C1',1);
```

```
SELECT *
FROM ttl_table_v1

┌─id──┬─────────create_time─┬─code─┬─type─┐
| A000 | 2020-08-19 20:11:51 | C1   |    1 |
| A000 | 2020-08-19 20:21:51 | C1   |    1 |
└─────┴─────────────────────┴──────┴──────┘
```

```
optimize table ttl_table_v1 FINAL;
```

```
SELECT *
FROM ttl_table_v1

┌─id──┬─────────create_time─┬─code─┬─type─┐
| A000 | 2020-08-19 20:11:51 |      |    0 |
| A000 | 2020-08-19 20:21:51 | C1   |    1 |
└─────┴─────────────────────┴──────┴──────┘
```

4.4.7.2 TTL设置在表上

```
create table tt1_table_v2 (
id String,
create_time DateTime,
code String TTL create_time + INTERVAL 10 SECOND,
type UInt8
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(create_time)
ORDER BY create_time
TTL create_time + INTERVAL 1 DAY;
```

```
ALTER TABLE tt1_table_v1 MODIFY TTL create_time + INTERVAL + 3 DAY;
```

TTL目前没有取消方法

4.4.7.3 TTL文件说明

```
[root@hdp-1 202008_1_1_0]# cat ttl.txt
ttl format version: 1
{"columns":[{"name":"code","min":1597839121,"max":1597839721},
{"name":"type","min":1597839121,"max":1597839721}]}
```

```
SELECT
    toDateTime(1597839121) AS ttl_min,
    toDateTime(1597839721) AS ttl_max

┌──────────────ttl_min─┬──────────────ttl_max─┐
| 2020-08-19 20:12:01 | 2020-08-19 20:22:01 |
└─────────────────────┴─────────────────────┘
```

```
SELECT *
FROM ttl_table_v1

┌─id──┬──────────create_time─┬─code─┬─type─┐
| A000 | 2020-08-19 20:11:51 |      |    0 |
| A000 | 2020-08-19 20:12:01 | C1   |    1 |
└─────┴─────────────────────┴──────┴──────┘
```

一列数据中最小时间：2020-08-19 20:11:51 + INTERVAL的10秒钟 = 2020-08-19 20:12:01 时间戳："min":1597839121

一列数据中最大时间：2020-08-19 20:12:01 + INTERVAL的10秒钟 = 2020-08-19 20:22:01 时间戳："max":1597839721

文件ttl.txt记录的是列字段的过期时间。

4.4.8 MergeTree的存储策略

整体配置：

```
<storage_configuration>
    <disks>
        <disk_hot1>
            <path>/var/lib/clickhouse/chbase/hotdata1/</path>
        </disk_hot1>
        <disk_hot2>
            <path>/var/lib/clickhouse/chbase/hotdata2/</path>
        </disk_hot2>
        <disk_cold>
            <path>/var/lib/clickhouse/chbase/colddata/</path>
        </disk_cold>
    </disks>
    <policies>
```

```
        <default_jbod>
            <volumes>
                <jbod>
                    <disk>disk_hot1</disk>:
                    <disk>disk_hot2</disk>
                </jbod>

            </volumes>
            <move_factor>0.2</move_factor>
        </default_jbod>
        <moving_from_hot_to_cold>
            <volumes>
                <hot>
                    <disk>disk_hot1</disk>
                    <max_data_part_size_bytes>1073741824</max_data_part_size_bytes>
                </hot>
                <cold>
                    <disk>disk_cold</disk>
                </cold>
            </volumes>
            <move_factor>0.2</move_factor>
        </moving_from_hot_to_cold>
        <moving_from_hot_to_cold_new>
            <volumes>
                <hot>
                    <disk>disk_hot2</disk>
                    <max_data_part_size_bytes>1048576</max_data_part_size_bytes>
                </hot>
                <cold>
                    <disk>disk_cold</disk>
                </cold>
            </volumes>
            <move_factor>0.2</move_factor>
        </moving_from_hot_to_cold_new>
    </policies>
</storage_configuration>
```

4.4.8.1 默认策略

19.15之前,只能单路径存储，存储位置为在config.xml配置文件中指定

```
<!-- Path to data directory, with trailing slash. -->
<path>/var/lib/clickhouse/</path>
```

19.15之后，支持多路径存储策略的自定义存储策略，目前有三类策略：

4.4.8.2 JBOD策略

配置方式在config.xml配置文件中指定：

```
<storage_configuration>
    <disks>
        <disk_hot1>
            <path>/var/lib/clickhouse/chbase/hotdata1/</path>
        </disk_hot1>
        <disk_hot2>
            <path>/var/lib/clickhouse/chbase/hotdata2/</path>
        </disk_hot2>
                <disk_cold>
                        <path>/var/lib/clickhouse/chbase/colddata/</path>
                </disk_cold>
    </disks>
        <policies>
        <default_jbod>
            <volumes>
                <jbod>
                    <disk>disk_hot1</disk>:
                                        <disk>disk_hot2</disk>
                </jbod>

            </volumes>
            <move_factor>0.2</move_factor>
        </default_jbod>
    </policies>
</storage_configuration>
```

```
service clickhouse-server restart
```

```
select
name,
path,
formatReadableSize(free_space) as free,
formatReadableSize(total_space) as total,
formatReadableSize(keep_free_space) as reserved
from system.disks;
```

```
SELECT
    name,
    path,
    formatReadableSize(free_space) AS free,
    formatReadableSize(total_space) AS total,
    formatReadableSize(keep_free_space) AS reserved
FROM system.disks
```

```
┌─name─────┬─path──────────────────────────────────┬─free─────┬─total────┬─reserved─┐
| default   | /var/lib/clickhouse/                  | 5.02 GiB | 16.99 GiB | 0.00 B   |
| disk_cold | /var/lib/clickhouse/chbase/colddata/ | 5.02 GiB | 16.99 GiB | 0.00 B   |
| disk_hot1 | /var/lib/clickhouse/chbase/hotdata1/ | 5.02 GiB | 16.99 GiB | 0.00 B   |
| disk_hot2 | /var/lib/clickhouse/chbase/hotdata2/ | 5.02 GiB | 16.99 GiB | 0.00 B   |
└──────────┴──────────────────────────────────────┴──────────┴──────────┴──────────┘
```

system.disk系统表，刚才配置的三块磁盘已经生效

```
select
policy_name,
volume_name,
volume_priority,
disks,
formatReadableSize(max_data_part_size) max_data_part_size,
move_factor
from system.storage_policies;
```

```
SELECT
    policy_name,
    volume_name,
    volume_priority,
    disks,
    formatReadableSize(max_data_part_size) AS max_data_part_size,
    move_factor
FROM system.storage_policies

┌─policy_name─┬─volume_name─┬─disks─────────────────┬─move_factor─┐
| default      | default     |  ['default']          |           0 |
| default_jbod | jbod        |  ['disk_hot1','disk_hot2'] |         0.2 |
└─────────────┴─────────────┴──────────────────────┴─────────────┘
```

system.storage_policies系统表可以看到刚才配置的策略也生效了。

```
insert into table jbod_table select rand() from numbers(10);

select name,disk_name from system.parts where table='jbod_table';
```

```
SELECT
    name,
    disk_name
FROM system.parts
WHERE table = 'jbod_table'

┌─name──────┬─disk_name─┐
| all_1_1_0 | disk_hot1 |
└───────────┴───────────┘
```

在此插入一条数据：

```
insert into table jbod_table select rand() from numbers(10);

select name,disk_name from system.parts where table='jbod_table';
```

```
SELECT
    name,
    disk_name
FROM system.parts
WHERE table = 'jbod_table'

┌─name──────┬─disk_name─┐
| all_1_1_0 | disk_hot1 |
| all_2_2_0 | disk_hot2 |
└───────────┴───────────┘
```

触发合并操作：

```
optimize table jbod_table;
```

```
SELECT
    name,
    disk_name
FROM system.parts
WHERE table = 'jbod_table'

┌─name──────┬─disk_name─┐
| all_1_1_0 | disk_hot1 |
| all_1_2_1 | disk_hot1 |
| all_2_2_0 | disk_hot2 |
└───────────┴───────────┘
```

验证JBOD策略的工作方式，多个磁盘组成了一个磁盘组volume卷，新生成的数据分区，分区目录会按照volume卷中的磁盘定义顺序，轮询写入数据。

4.4.8.3 HOT/COLD策略

conf.xml配置文件：

```
<storage_configuration>
    <disks>
        <disk_hot1>
            <path>/var/lib/clickhouse/chbase/hotdata1/</path>
        </disk_hot1>
        <disk_hot2>
            <path>/var/lib/clickhouse/chbase/hotdata2/</path>
        </disk_hot2>
                <disk_cold>
                        <path>/var/lib/clickhouse/chbase/colddata/</path>
                </disk_cold>
    </disks>
        <policies>
        <default_jbod>
            <volumes>
                <jbod>
                    <disk>disk_hot1</disk>:
                                        <disk>disk_hot2</disk>
                </jbod>

            </volumes>
            <move_factor>0.2</move_factor>
        </default_jbod>
                <moving_from_hot_to_cold>
                        <volumes>
                                <hot>
                                        <disk>disk_hot1</disk>

<max_data_part_size_bytes>1073741824</max_data_part_size_bytes>
                                </hot>
                                <cold>
                                        <disk>disk_cold</disk>
```

```
                                </cold>
                        </volumes>
                        <move_factor>0.2</move_factor>
                </moving_from_hot_to_cold>
    </policies>
</storage_configuration>
```

重启服务：service clickhouse-server restart

创建表，测试moving_from_hot_to_cold存储策略：

```
create table hot_cold_table(
id UInt64
)ENGINE=MergeTree()
ORDER BY id
SETTINGS storage_policy='moving_from_hot_to_cold_new';
```

写入一批500K的数据，生成一个分区目录：

```
insert into hot_cold_table select rand() from numbers(100000);
```

```
select name,disk_name from system.parts where table = 'hot_cold_table';
```

```
SELECT
    name,
    disk_name
FROM system.parts
WHERE table = 'hot_cold_table'

┌─name──────┬─disk_name─┐
| all_1_1_0 | disk_hot1 |
└───────────┴───────────┘
```

第一个分区写入hot1卷。

接着写入第二批500k的数据

```
insert into hot_cold_table select rand() from numbers(100000);
```

```
select name,disk_name from system.parts where table = 'hot_cold_table';
```

```
SELECT
    name,
    disk_name
FROM system.parts
WHERE table = 'hot_cold_table'

┌─name────────┬─disk_name─┐
| all_1_1_0 | disk_hot1 |
| all_2_2_0 | disk_hot1 |
└─────────────┴───────────┘
```

每一个分区大小为500k

接下来触发合并：

```
OPTIMIZE TABLE hot_cold_table_new

Ok.

0 rows in set. Elapsed: 0.013 sec.

hdp-1 :) select name, disk_name from system.parts where table = 'hot_cold_table_new';

SELECT
    name,
    disk_name
FROM system.parts
WHERE table = 'hot_cold_table_new'

┌─name────────┬─disk_name─┐
| all_1_1_0 | disk_hot2 |
| all_1_2_1 | disk_cold |
| all_2_2_0 | disk_hot2 |
└─────────────┴───────────┘
```

查询大小：

```
select
disk_name,
formatReadableSize(bytes_on_disk) as size
from system.parts where (table = 'hot_cold_table_new') and active = 1;
```

```
SELECT
    disk_name,
    formatReadableSize(bytes_on_disk) AS size
FROM system.parts
WHERE (table = 'hot_cold_table_new') AND (active = 1)

┌─disk_name─┬─size─────┐
| disk_cold | 1.01 MiB |
└───────────┴──────────┘
```

虽然MergeTree的存储策略不能修改，但是分区目录却支持移动。

第5部分 MergeTree家族表引擎

5.1 ReplacingMergeTree

这个引擎是在 MergeTree 的基础上，添加了“处理重复数据”的功能，该引擎和MergeTree的不同之处在于它会删除具有相同主键的重复项。

特点：

1. 使用ORDER BY排序键作为判断重复的唯一键
2. 数据的去重只会在合并的过程中触发
3. 以数据分区为单位删除重复数据，不同分区的的重复数据不会被删除
4. 找到重复数据的方式依赖数据已经ORDER BY排好序了
5. 如果没有ver版本号，则保留重复数据的最后一行
6. 如果设置了ver版本号，则保留重复数据中ver版本号最大的数据

格式：

ENGINE [=] ReplacingMergeTree(date-column [, sampling_expression], (primary, key), index_granularity, [ver])

可以看出他比MergeTree只多了一个ver，这个ver指代版本列。

案例：

```
create table replace_table(
id String,
code String,
create_time DateTime
)ENGINE=ReplacingMergeTree()
PARTITION BY toYYYYMM(create_time)
ORDER BY (id,code)
PRIMARY KEY id;
```

```
insert into replace_table values('A001','C1','2020-08-21 08:00:00');
insert into replace_table values('A001','C1','2020-08-22 08:00:00');
insert into replace_table values('A001','C8','2020-08-23 08:00:00');
insert into replace_table values('A001','C9','2020-08-24 08:00:00');
insert into replace_table values('A002','C2','2020-08-25 08:00:00');
insert into replace_table values('A003','C3','2020-08-26 08:00:00');
```

optimize强制触发：

```
SELECT *
FROM replace_table

┌─id──┬─code─┬─────────create_time─┐
| A001 | C1   | 2020-08-22 08:00:00 |
| A001 | C8   | 2020-08-23 08:00:00 |
| A001 | C9   | 2020-08-24 08:00:00 |
| A002 | C2   | 2020-08-25 08:00:00 |
| A003 | C3   | 2020-08-26 08:00:00 |
└──────┴──────┴─────────────────────┘
```

通过观察，去重是根据ORDER BY来的，并非PRIMARY KEY

在继续插入一条数据：

```
insert into replace_table values('A001','C1','2020-05-21 08:00:00');
```

```
OPTIMIZE TABLE replace_table

Ok.

0 rows in set. Elapsed: 0.001 sec.

hdp-1 :) select * from replace_table;

SELECT *
FROM replace_table

┌─id──┬─code─┬─────────create_time─┐
| A001 | C1   | 2020-05-21 08:00:00 |
└──────┴──────┴─────────────────────┘
┌─id──┬─code─┬─────────create_time─┐
| A001 | C1   | 2020-08-22 08:00:00 |
| A001 | C8   | 2020-08-23 08:00:00 |
| A001 | C9   | 2020-08-24 08:00:00 |
| A002 | C2   | 2020-08-25 08:00:00 |
| A003 | C3   | 2020-08-26 08:00:00 |
└──────┴──────┴─────────────────────┘
```

通过观察，不同分区的数据不会去重。

5.2 SummingMergeTree

该引擎继承自 MergeTree。区别在于，当合并 SummingMergeTree 表的数据片段时，ClickHouse 会把所有具有相同聚合数据的条件Key的行合并为一行，该行包含了被合并的行中具有数值数据类型的列的汇总值。如果聚合数据的条件Key的组合方式使得单个键值对应于大量的行，则可以显著的减少存储空间并加快数据查询的速度，对于不可加的列，会取一个最先出现的值。

特征：

1. 用ORDER BY排序键作为聚合数据的条件Key
2. 合并分区的时候触发汇总逻辑
3. 以数据分区为单位聚合数据，不同分区的数据不会被汇总
4. 如果在定义引擎时指定了Columns汇总列（非主键）则SUM汇总这些字段
5. 如果没有指定，则汇总所有非主键的数值类型字段
6. SUM汇总相同的聚合Key的数据，依赖ORDER BY排序
7. 同一分区的SUM汇总过程中，非汇总字段的数据保留第一行取值
8. 支持嵌套结构，但列字段名称必须以Map后缀结束。

语法：

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = SummingMergeTree([columns])
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

columns — 包含将要被汇总的列的列名的元组

案例1：

```
create table smt_table (date Date, name String, a UInt16, b UInt16)
ENGINE=SummingMergeTree(date, (date, name), 8192, (a))
```

插入数据：

```
insert into smt_table (date, name, a, b) values ('2019-07-10', 'a', 1, 2);

insert into smt_table (date, name, a, b) values ('2019-07-10', 'b', 2, 1);

insert into smt_table (date, name, a, b) values ('2019-07-11', 'b', 3, 8);

insert into smt_table (date, name, a, b) values ('2019-07-11', 'b', 3, 8);

insert into smt_table (date, name, a, b) values ('2019-07-11', 'a', 3, 1);

insert into smt_table (date, name, a, b) values ('2019-07-12', 'c', 1, 3);
```

等待一段时间或optimize table smt_table手动触发merge，后查询

```
:) select * from smt_table



┌──────date─┬─name─┬─a─┬─b─┐

| 2019-07-10 | a    | 1 | 2 |

| 2019-07-10 | b    | 2 | 1 |

| 2019-07-11 | a    | 3 | 1 |

| 2019-07-11 | b    | 6 | 8 |

| 2019-07-12 | c    | 1 | 3 |

└────────────┴──────┴───┴───┘
```

发现2019-07-11，b的a列合并相加了，b列取了8（因为b列为8的数据最先插入）。

案例2:

```
create table summing_table(
id String,
city String,
v1 UInt32,
v2 Float64,
create_time DateTime
)ENGINE=SummingMergeTree()
PARTITION BY toYYYYMM(create_time)
ORDER BY (id,city)
PRIMARY KEY id;
```

```
insert into table summing_table values('A000','beijing',10,20,'2020-08-20 08:00:00');
insert into table summing_table values('A000','beijing',20,30,'2020-08-30 08:00:00');
insert into table summing_table values('A000','shanghai',10,20,'2020-08-20 08:00:00');
insert into table summing_table values('A000','beijing',10,20,'2020-06-20 08:00:00');
insert into table summing_table values('A001','beijing',50,60,'2020-02-20 08:00:00');
```

```
OPTIMIZE TABLE summing_table

Ok.

0 rows in set. Elapsed: 0.003 sec.

hdp-1 :) select * from summing_table;

SELECT *
FROM summing_table

┌─id───┬─city─────┬─v1─┬─v2─┬─────────create_time─┐
| A001 | beijing | 50 | 60 | 2020-02-20 08:00:00 |
└──────┴──────────┴────┴────┴─────────────────────┘
┌─id───┬─city─────┬─v1─┬─v2─┬─────────create_time─┐
| A000 | beijing | 10 | 20 | 2020-06-20 08:00:00 |
└──────┴──────────┴────┴────┴─────────────────────┘
┌─id───┬─city─────┬─v1─┬─v2─┬─────────create_time─┐
| A000 | beijing  | 30 | 50 | 2020-08-20 08:00:00 |
| A000 | shanghai | 10 | 20 | 2020-08-20 08:00:00 |
└──────┴──────────┴────┴────┴─────────────────────┘
```

通过观察，根据ORDER BY排序键(id,city)作为聚合Key,因为没有在建表指定SummingMergeTree的时候没有指定Sum列，所以把所有非主键数值类型的列都进行了SUM逻辑

```
┌─id───┬─city─────┬─v1─┬─v2─┬─────────create_time─┐
| A000 | beijing | 20 | 30 | 2020-08-30 08:00:00 |
└──────┴──────────┴────┴────┴─────────────────────┘
┌─id───┬─city─────┬─v1─┬─v2─┬─────────create_time─┐
| A000 | shanghai | 10 | 20 | 2020-08-20 08:00:00 |
└──────┴──────────┴────┴────┴─────────────────────┘
```

这两条数据v1和v2分别相加SUM后结果为30,50

```
┌─id───┬─city─────┬─v1─┬─v2─┬─────────create_time─┐
| A000 | beijing  | 30 | 50 | 2020-08-20 08:00:00 |
| A000 | shanghai | 10 | 20 | 2020-08-20 08:00:00 |
└──────┴──────────┴────┴────┴─────────────────────┘
```

案例3:

SummingMergeTree支持嵌套类型的字段，但列字段名称必须以Map后缀结束。

```
create table summing_table_nested(
id String,
nestMap Nested(
id UInt32,
key UInt32,
val UInt64
),
create_time DateTime
)ENGINE=SummingMergeTree()
PARTITION BY toYYYYMM(create_time)
ORDER BY id;
```

id:A001

nestMap.id: [1,1,2]

nestMap.key[10,20,100]

nestMap.val[40,60,20]

create_time: 2020-08-22 08:00:00

```
[(1, 10, 40)] + [(1, 20, 60)] -> [(1, 30, 100)]
[(2, 100, 20)] -> [(2, 100, 20)]
```

5.3 AggregateMergeTree

说明： 该引擎继承自 MergeTree，并改变了数据片段的合并逻辑。 ClickHouse 会将相同主键的所有行（在一个数据片段内）替换为单个存储一系列聚合函数状态的行。 可以使用 AggregatingMergeTree 表来做增量数据统计聚合，包括物化视图的数据聚合。 引擎需使用 AggregateFunction 类型来处理所有列。 如果要按一组规则来合并减少行数，则使用 AggregatingMergeTree 是合适的。 对于AggregatingMergeTree不能直接使用insert来查询写入数据。一般是用insert select。但更常用的是创建物化视图

a. 先创建一个MergeTree引擎的基表

```
hdp-1 :) create table arr_table_base (id String, city String, code String,value UInt32)
engine=MergeTree partition by city order by (id,city);

CREATE TABLE arr_table_base
(
    `id` String,
    `city` String,
    `code` String,
    `value` UInt32
)
ENGINE = MergeTree
PARTITION BY city
ORDER BY (id, city)
```

b. 往基表写入数据

```

```

c. 创建一个AggregatingMergeTree的物化视图

```
create materialized view agg_view engine=AggregatingMergeTree() partition by city order
by(id,city) as select id,city,uniqState(code) as code, sumState(value) as value from
arr_table_base group by id,city;

CREATE MATERIALIZED VIEW agg_view
ENGINE = AggregatingMergeTree()
PARTITION BY city
ORDER BY (id, city) AS
SELECT
    id,
    city,
    uniqState(code) AS code,
    sumState(value) AS value
FROM arr_table_base
GROUP BY
    id,
    city
```

d. 根据b往基表写数据的方法重写写一次将数据填充到物化视图amt_tab_view中并查询

```
insert into table arr_table_base values
('A000','wuhan','code1',1),
('A000','wuhan','code2',200),
('A000','zhuhai','code1',200);
```

e. 通过optimize命令手动Merge后查询

```
select id,sumMerge(value),uniqMerge(code) from agg_view group by id,city;

SELECT
    id,
    sumMerge(value),
    uniqMerge(code)
FROM agg_view
GROUP BY
    id,
    city

┌─id───┬─sumMerge(value)─┬─uniqMerge(code)─┐
│ A000 │             200 │               1 │
│ A000 │             201 │               2 │
└──────┴─────────────────┴─────────────────┘
```

f. 使用场景

可以使用AggregatingMergeTree表来做增量数据统计聚合，包括物化视图的数据聚合。

5.4 CollapsingMergeTree

以增代删：

说明： yandex官方给出的介绍是CollapsingMergeTree 会异步的删除（折叠）这些除了特定列 Sign 有 1 和 -1 的值以外，其余所有字段的值都相等的成对的行。没有成对的行会被保留。该引擎可以显著的降低存储量并提高 SELECT 查询效率。 CollapsingMergeTree引擎有个状态列sign，这个值1为”状态”行，-1为”取消”行，对于数据只关心状态列为状态的数据，不关心状态列为取消的数据

a. 创建CollapsingMergeTree表

创表语法：

```
 CREATE TABLE cmt_tab(
     sign Int8,
     date Date,
     name String,
     point String) ENGINE=CollapsingMergeTree(sign)
     PARTITION BY date
     ORDER BY (name)
     SAMPLE BY name;

CREATE TABLE cmt_tab
(
    `sign` Int8,
    `date` Date,
    `name` String,
    `point` String
)
ENGINE = CollapsingMergeTree(sign)
PARTITION BY date
```

```
ORDER BY name
SAMPLE BY name
```

b. 插入数据：

```
insert into cmt_tab(sign,date,name,point) values (1,'2019-12-13','cctv','100000');
insert into cmt_tab(sign,date,name,point) values (-1,'2019-12-13','cctv','100000');
insert into cmt_tab(sign,date,name,point) values (1,'2019-12-13','hntv','10000');
insert into cmt_tab(sign,date,name,point) values (-1,'2019-12-13','hntv','10000');
insert into cmt_tab(sign,date,name,point) values (1,'2019-12-13','hbtv','11000');
insert into cmt_tab(sign,date,name,point) values (-1,'2019-12-13','hbtv','11000');
insert into cmt_tab(sign,date,name,point) values (1,'2019-12-14','cctv','200000');
insert into cmt_tab(sign,date,name,point) values (1,'2019-12-14','hntv','15000');
insert into cmt_tab(sign,date,name,point) values (1,'2019-12-14','hbtv','16000');
```

c. 通过optimize table amt_tab_view命令手动Merge后查询

```
SELECT *
FROM cmt_tab

┌─sign─┬────────date─┬─name─┬─point─┐
│    1 │ 2019-12-14 │ cctv │ 200000 │
└──────┴────────────┴──────┴────────┘
┌─sign─┬────────date─┬─name─┬─point─┐
│    1 │ 2019-12-14 │ hntv │ 15000 │
└──────┴────────────┴──────┴────────┘
┌─sign─┬────────date─┬─name─┬─point─┐
│    1 │ 2019-12-14 │ hbtv │ 16000 │
└──────┴────────────┴──────┴────────┘
```

d.使用场景

大数据中对于数据更新很难做到，比如统计一个网站或TV的在用户数，更多场景都是选择用记录每个点的数据，再对数据进行一定聚合查询。而clickhouse通过CollapsingMergeTree就可以实现，所以使得CollapsingMergeTree大部分用于OLAP场景

5.5 VersionedCollapsingMergeTree

这个引擎和CollapsingMergeTree差不多，只是对CollapsingMergeTree引擎加了一个版本，比如可以适用于非实时用户在线统计，统计每个节点用户在在线业务 a. 创表语法

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
) ENGINE = VersionedCollapsingMergeTree(sign, version)
[PARTITION BY expr]
[ORDER BY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]
```

第6部分

6.1 HDFS

该引擎提供了集成 Apache Hadoop 生态系统通过允许管理数据 HDFS通过ClickHouse. 这个引擎是相似的 到 文件 和 URL 引擎，但提供Hadoop特定的功能。

用途

```
ENGINE = HDFS(URI, format)
```

该 URI 参数是HDFS中的整个文件URI。 该 format 参数指定一种可用的文件格式。 执行 SELECT 查询时，格式必须支持输入，并执行 INSERT queries – for output. The available formats are listed in the 格式 科。 路径部分 URI 可能包含水珠。 在这种情况下，表将是只读的。

示例:

1. 设置 hdfs_engine_table 表:

```
CREATE TABLE hdfs_engine_table (name String, value UInt32)
ENGINE=HDFS('hdfs://hdfs1:9000/other_storage', 'TSV')
```

2. 填充文件:

```
INSERT INTO hdfs_engine_table VALUES ('one', 1), ('two', 2), ('three', 3)
```

3. 查询数据:

```
SELECT * FROM hdfs_engine_table LIMIT 2
┌─name─┬─value─┐
| one  |     1 |
| two  |     2 |
└──────┴───────┘
```

实施细节

- 读取和写入可以并行
- 不支持:
  - ALTER 和 SELECT...SAMPLE 操作。
  - 索引。
  - 复制。

路径中的水珠

多个路径组件可以具有globs。 对于正在处理的文件应该存在并匹配到整个路径模式。 文件列表确定在 SELECT （不在 CREATE 时刻）。

- — Substitutes any number of any characters except / 包括空字符串。
- ? — Substitutes any single character.
- {some_string,another_string,yet_another_one} — Substitutes any of strings 'some_string', 'another_string', 'yet_another_one'.
- {N..M} — Substitutes any number in range from N to M including both borders.

建筑与 {} 类似于 远程 表功能。

示例

1. 假设我们在HDFS上有几个TSV格式的文件，其中包含以下Uri:

- ‘hdfs://hdfs1:9000/some_dir/some_file_1’
- ‘hdfs://hdfs1:9000/some_dir/some_file_2’
- ‘hdfs://hdfs1:9000/some_dir/some_file_3’
- ‘hdfs://hdfs1:9000/another_dir/some_file_1’
- ‘hdfs://hdfs1:9000/another_dir/some_file_2’
- ‘hdfs://hdfs1:9000/another_dir/some_file_3’

1. 有几种方法可以创建由所有六个文件组成的表:

```
CREATE TABLE table_with_range (name String, value UInt32) ENGINE =
HDFS('hdfs://hdfs1:9000/{some,another}_dir/some_file_{1..3}', 'TSV')
```

另一种方式:

```
CREATE TABLE table_with_question_mark (name String, value UInt32) ENGINE =
HDFS('hdfs://hdfs1:9000/{some,another}_dir/some_file_?', 'TSV')
```

表由两个目录中的所有文件组成（所有文件都应满足query中描述的格式和模式):

```
CREATE TABLE table_with_asterisk (name String, value UInt32) ENGINE =
HDFS('hdfs://hdfs1:9000/{some,another}_dir/*', 'TSV')
```

警告

如果文件列表包含带有前导零的数字范围，请单独使用带有大括号的构造或使用 ?.

示例

创建具有名为文件的表 file000, file001, ... , file999:

```
CREARE TABLE big_table (name String, value UInt32) ENGINE =
HDFS('hdfs://hdfs1:9000/big_dir/file{0..9}{0..9}{0..9}', 'CSV')
```

虚拟列

- _path — Path to the file.
- _file — Name of the file.

另请参阅

- 虚拟列

6.2 MySQL

MySQL 引擎可以对存储在远程 MySQL 服务器上的数据执行 SELECT 查询。

调用格式：

```
MySQL('host:port', 'database', 'table', 'user', 'password'[, replace_query,
'on_duplicate_clause']);
```

调用参数

- host:port — MySQL 服务器地址。
- database — 数据库的名称。
- table — 表名称。
- user — 数据库用户。
- password — 用户密码。
- replace_query — 将 INSERT INTO 查询是否替换为 REPLACE INTO 的标志。如果 replace_query=1，则替换查询
- 'on_duplicate_clause' — 将 ON DUPLICATE KEY UPDATE 'on_duplicate_clause' 表达式添加到 INSERT 查询语句中。例如：impression = VALUES(impression) + impression。如果需要指定 'on_duplicate_clause'，则需要设置 replace_query=0。如果同时设置 replace_query = 1 和 'on_duplicate_clause'，则会抛出异常。

此时，简单的 WHERE 子句（例如 =, !=, >, >=, <, <=）是在 MySQL 服务器上执行。

其余条件以及 LIMIT 采样约束语句仅在对MySQL的查询完成后才在ClickHouse中执行。

MySQL 引擎不支持 可为空 数据类型，因此，当从MySQL表中读取数据时，NULL 将转换为指定列类型的默认值（通常为0或空字符串）。

案例：

```
CREATE TABLE mysql_table2
(
    `id` UInt32,
    `name` String,
    `age` UInt32
)
ENGINE = MySQL('10.1.192.183:3306', 'bigdata', 'student', 'root', 'lucas')
```

```
SELECT *
FROM mysql_table2

┌─id─┬─name──┬─age─┐
│  3 | jack  |  28 |
│  4 | lucas |  18 |
└────┴───────┴─────┘
```

6.3 Kafka

此引擎与 Apache Kafka 结合使用。

Kafka 特性：

- 发布或者订阅数据流。
- 容错存储机制。
- 处理流数据。

老版格式：

```
Kafka(kafka_broker_list, kafka_topic_list, kafka_group_name, kafka_format
      [, kafka_row_delimiter, kafka_schema, kafka_num_consumers])
```

新版格式：

```
Kafka SETTINGS
  kafka_broker_list = 'localhost:9092',
  kafka_topic_list = 'topic1,topic2',
  kafka_group_name = 'group1',
  kafka_format = 'JSONEachRow',
  kafka_row_delimiter = '\n',
  kafka_schema = '',
  kafka_num_consumers = 2
```

必要参数：

- kafka_broker_list – 以逗号分隔的 brokers 列表 (localhost:9092)。
- kafka_topic_list – topic 列表 (my_topic)。
- kafka_group_name – Kafka 消费组名称 (group1)。如果不希望消息在集群中重复，请在每个分片中使用相同的组名。
- kafka_format – 消息体格式。使用与 SQL 部分的 FORMAT 函数相同表示方法，例如 JSONEachRow。了解详细信息，请参考 Formats 部分。

可选参数：

- kafka_row_delimiter - 每个消息体（记录）之间的分隔符。
- kafka_schema – 如果解析格式需要一个 schema 时，此参数必填。例如，普罗托船长 需要 schema 文件路径以及根对象 schema.capnp:Message 的名字。

- kafka_num_consumers – 单个表的消费者数量。默认值是：1，如果一个消费者的吞吐量不足，则指定更多的消费者。消费者的总数不应该超过 topic 中分区的数量，因为每个分区只能分配一个消费者。

示例：

```
CREATE TABLE queue (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka('localhost:9092', 'topic', 'group1', 'JSONEachRow');

SELECT * FROM queue LIMIT 5;

CREATE TABLE queue2 (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka SETTINGS kafka_broker_list = 'localhost:9092',
                          kafka_topic_list = 'topic',
                          kafka_group_name = 'group1',
                          kafka_format = 'JSONEachRow',
                          kafka_num_consumers = 4;

CREATE TABLE queue2 (
  timestamp UInt64,
  level String,
  message String
) ENGINE = Kafka('localhost:9092', 'topic', 'group1')
            SETTINGS kafka_format = 'JSONEachRow',
                     kafka_num_consumers = 4;
```

消费的消息会被自动追踪，因此每个消息在不同的消费组里只会记录一次。如果希望获得两次数据，则使用另一个组名创建副本。

消费组可以灵活配置并且在集群之间同步。例如，如果群集中有10个主题和5个表副本，则每个副本将获得2个主题。 如果副本数量发生变化，主题将自动在副本中重新分配。了解更多信息请访问 http://kafka.apache.org/intro。

SELECT 查询对于读取消息并不是很有用（调试除外），因为每条消息只能被读取一次。使用物化视图创建实时线程更实用。您可以这样做：

1. 使用引擎创建一个 Kafka 消费者并作为一条数据流。
2. 创建一个结构表。
3. 创建物化视图，改视图会在后台转换引擎中的数据并将其放入之前创建的表中。

当 MATERIALIZED VIEW 添加至引擎，它将会在后台收集数据。可以持续不断地从 Kafka 收集数据并通过 SELECT 将数据转换为所需要的格式。

示例：

```
CREATE TABLE queue (
  timestamp UInt64,
  level String,
  message String
```

```
) ENGINE = Kafka('localhost:9092', 'topic', 'group1', 'JSONEachRow');

CREATE TABLE daily (
  day Date,
  level String,
  total UInt64
) ENGINE = SummingMergeTree(day, (day, level), 8192);

CREATE MATERIALIZED VIEW consumer TO daily
  AS SELECT toDate(toDateTime(timestamp)) AS day, level, count() as total
  FROM queue GROUP BY day, level;

SELECT level, sum(total) FROM daily GROUP BY level;
```

为了提高性能，接受的消息被分组为 max_insert_block_size 大小的块。如果未在 stream_flush_interval_ms 毫秒内形成块，则不关心块的完整性，都会将数据刷新到表中。

停止接收主题数据或更改转换逻辑，请 detach 物化视图：

```
DETACH TABLE consumer;
ATTACH TABLE consumer;
```

如果使用 ALTER 更改目标表，为了避免目标表与视图中的数据之间存在差异，推荐停止物化视图。

配置

与 GraphiteMergeTree 类似，Kafka 引擎支持使用ClickHouse配置文件进行扩展配置。可以使用两个配置键：全局 (kafka) 和 主题级别 (kafka_*)。首先应用全局配置，然后应用主题级配置（如果存在）。

```
<!-- Global configuration options for all tables of Kafka engine type -->
<kafka>
  <debug>cgrp</debug>
  <auto_offset_reset>smallest</auto_offset_reset>
</kafka>

<!-- Configuration specific for topic "logs" -->
<kafka_logs>
  <retry_backoff_ms>250</retry_backoff_ms>
  <fetch_min_bytes>100000</fetch_min_bytes>
</kafka_logs>
```

有关详细配置选项列表，请参阅 librdkafka配置参考。在 ClickHouse 配置中使用下划线 (_)，并不是使用点 (.)。例如，check.crcs=true 将是 true。

6.4 JDBC

允许CH通过 JDBC 连接到外部数据库。

要实现JDBC连接，CH需要使用以后台进程运行的程序 clickhouse-jdbc-bridge。

该引擎支持 Nullable 数据类型。

建表

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name
(
    columns list...
)
ENGINE = JDBC(dbms_uri, external_database, external_table)
```

引擎参数

- dbms_uri — 外部DBMS的uri.

  格式: jdbc:://:/?user=&password=. MySQL示例: jdbc:mysql://localhost:3306/?user=root&password=root.
- external_database — 外部DBMS的数据库名.
- external_table — external_database中的外部表名.

用法示例

通过mysql控制台客户端来创建表

Creating a table in MySQL server by connecting directly with it's console client:

```
mysql> CREATE TABLE `test`.`test` (
    ->   `int_id` INT NOT NULL AUTO_INCREMENT,
    ->   `int_nullable` INT NULL DEFAULT NULL,
    ->   `float` FLOAT NOT NULL,
    ->   `float_nullable` FLOAT NULL DEFAULT NULL,
    ->   PRIMARY KEY (`int_id`));
Query OK, 0 rows affected (0,09 sec)

mysql> insert into test (`int_id`, `float`) VALUES (1,2);
Query OK, 1 row affected (0,00 sec)

mysql> select * from test;
+------+----------+-----+----------+
| int_id | int_nullable | float | float_nullable |
+------+----------+-----+----------+
|      1 |         NULL |     2 |           NULL |
+------+----------+-----+----------+
1 row in set (0,00 sec)
```

在CH服务端创建表，并从中查询数据：

```
CREATE TABLE jdbc_table
(
    `int_id` Int32,
    `int_nullable` Nullable(Int32),
    `float` Float32,
    `float_nullable` Nullable(Float32)
)
ENGINE JDBC('jdbc:mysql://localhost:3306/?user=root&password=root', 'test', 'test')
SELECT *
FROM jdbc_table
┌─int_id─┬─int_nullable─┬─float─┬─float_nullable─┐
│      1 │         ᴺᵁᴸᴸ │     2 │           ᴺᵁᴸᴸ │
└────────┴──────────────┴───────┴────────────────┘
```

第7部分 SQL语法

参见附录

第8部分 副本和分片

8.1、副本

ReplicatedMergeTree

zk: 实现多个实例之间的通信。

8.1.1 副本的特点

作为数据副本的主要实现载体，ReplicatedMergeTree在设计上有一些显著特点:

- 依赖ZooKeeper：在执行INSERT和ALTER查询的时候，ReplicatedMergeTree需要借助ZooKeeper的分布式协同能力，以实现多个副本之间的同步。但是在查询副本的时候，并不需要使用ZooKeeper。关于这方面的更多信息，会在稍后详细介绍。
- 表级别的副本：副本是在表级别定义的，所以每张表的副本配置都可以按照它的实际需求进行个性化定义，包括副本的数量，以及副本在集群内的分布位置等。
- 多主架构（Multi Master）：可以在任意一个副本上执行INSERT和ALTER查询，它们的效果是相同的。这些操作会借助ZooKeeper的协同能力被分发至每个副本以本地形式执行。
- Block数据块：在执行INSERT命令写入数据时，会依据max_insert_block_size的大小（默认1048576行）将数据切分成若干个Block数据块。所以Block数据块是数据写入的基本单元，并且具有写入的原子性和唯一性。
- 原子性：在数据写入时，一个Block块内的数据要么全部写入成功，要么全部失败。
- 唯一性：在写入一个Block数据块的时候，会按照当前Block数据块的数据顺序、数据行和数据大小等指标，计算Hash信息摘要并记录在案。在此之后，如果某个待写入的Block数据块与先前已被写入的Block数据块拥有相同的Hash摘要（Block数据块内数据顺序、数据大小和数据行均相同），则该Block数据块会被忽略。这项设计可以预防由异常原因引起的Block数据块重复写入的问题。如果只是单纯地看这些特点的说明，可能不够直观。没关系，接下来会逐步展开，并附带一系列具体的示例。

8.1.2 zk的配置方式

新建配置文件 /etc/clickhouse-server/config.d/metrika.xml

```
<yandex>
  <zookeeper-servers>
        <node index="1">
            <host>hdp-1</host>
            <port>2181</port>
        </node>
        <node index="2">
            <host>hdp-2</host>
            <port>2181</port>
        </node>
        <node index="3">
            <host>hdp-3</host>
            <port>2181</port>
        </node>
    </zookeeper-servers>
</yandex>
```

修改配置文件/etc/clickhouse-server/config.xml

```
 <include_from>/etc/clickhouse-server/config.d/metrika.xml</include_from>
 <zookeeper incl="zookeeper-servers" optional="true" />
```

8.1.3 副本的定义形式

```
CREATE TABLE table_name
(
    EventDate DateTime,
    CounterID UInt32,
    UserID UInt32
) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/table_name', '{replica}')
PARTITION BY toYYYYMM(EventDate)
ORDER BY (CounterID, EventDate, intHash32(UserID))
SAMPLE BY intHash32(UserID)
```

创建一个副本表

```
hdp-1 :) create table replicated_sales_5( id String, price Float64, create_time DateTime
)ENGINE=ReplicatedMergeTree('/clickhouse/tables/01/replicated_sales_5','ch5.nauu.com')
PARTITION BY toYYYYMM(create_time) ORDER BY id;

CREATE TABLE replicated_sales_5
(
    `id` String,
    `price` Float64,
    `create_time` DateTime
)
ENGINE = ReplicatedMergeTree('/clickhouse/tables/01/replicated_sales_5', 'hdp-1')
PARTITION BY toYYYYMM(create_time)
ORDER BY id
```

```
Ok.

0 rows in set. Elapsed: 0.071 sec.
```

ENGINE = ReplicatedMergeTree('/clickhouse/tables/01/replicated_sales_5', 'hdp-1')说明：

/clickhouse/tables: 约定俗称的路径固定前缀

01：分片编号

replicated_sales_5：数据表的名字，建议与物理表名相同

hdp-1： 在zk中创建副本的名称，约定俗成使用服务器的名称。

CH提供了一张zookeeper的代理表，可用通过SQL语句读取远端ZooKeeper的数据：

```
hdp-1 :) select * from zookeeper where path = '/clickhouse';

SELECT *
FROM zookeeper
WHERE path = '/clickhouse'

┌─name─┬─value─┬──────czxid─┬──────mzxid─┬────────────────ctime─┬────────────────mtime─┬─ver
sion
| 01   |       | 21474837087 | 21474837087 | 2020-08-24 14:52:24 | 2020-08-24 14:52:24 |
0
└──────┴───────┴─────────────┴─────────────┴─────────────────────┴─────────────────────┴─
```

通过zk查看

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls
[zk: localhost:2181(CONNECTED) 1] ls /
[itmayeidu_Lasting, cluster, controller, brokers, zookeeper, admin, isr_change_notification,
log_dir_event_notification, controller_epoch, clickhouse, edu-front-boot,
sentinel_rule_config, consumers, front, latest_producer_id_block, config, hbase]
[zk: localhost:2181(CONNECTED) 2] ls /clickhouse
[tables, task_queue]
[zk: localhost:2181(CONNECTED) 3] ls /clickhouse/tables/
Command failed: java.lang.IllegalArgumentException: Path must not end with / character
[zk: localhost:2181(CONNECTED) 4] ls /clickhouse/tables
[01]
[zk: localhost:2181(CONNECTED) 5] ls /clickhouse/tables/01
[replicated_sales_5]
[zk: localhost:2181(CONNECTED) 6] ls /clickhouse/tables/01/replicated_sales_5
[metadata, temp, mutations, log, leader_election, columns, blocks,
```

```
nonincrement_block_numbers, replicas, quorum, block_numbers]
[zk: localhost:2181(CONNECTED) 7]
```

8.2、ReplicatedMergeTree原理

8.2.1 数据结构

```
[zk: localhost:2181(CONNECTED) 6] ls /clickhouse/tables/01/replicated_sales_5
[metadata, temp, mutations, log, leader_election, columns, blocks,
nonincrement_block_numbers, replicas, quorum, block_numbers]
```

元数据

metadata:元数信息： 主键、采样表达式、分区键

columns:列的字段的数据类型、字段名

replicats:副本的名称

标志：

leder_election:主副本的选举路径

blocks:hash值（复制数据重复插入）、partition_id

max_insert_block_size: 1048576行

block_numbers:在同一分区下block的顺序

quorum:副本的数据量

操作类：

log:log-000000 常规操作

mutations: delete update

replicas:

Entry：

LogEntry和MutationEntry

```
format version: 4
create_time: 2020-10-31 20:24:25
source replica: hdp-1
block_id: 202008_780388339403124112_11960209063776297222
get
202008_0_0_0
```

get:指令（获取数据的指令）

谁会获取这个指令？ --- - hdp-2会获取，并执行

202008_0_0_0：分区信息、告诉hdp-2 你要获取哪一个分区的数据

8.2.2 副本协同的核心流程

8.2.2.1 INSERT

在hdp-1机器上创建一个副本实例：

```
create table a1(
id String,
price Float64,
create_time DateTime
)ENGINE=ReplicatedMergeTree('/clickhouse/tables/01/a1','hdp-1')
PARTITION BY toYYYYMM(create_time)
ORDER BY id;
```

- 根据zk_path初始化所有的zk节点
- 在replicas节点下注册自己的副本实例hdp-1
- 启动监听任务，监听/log日志节点
- 参与副本选举，选出主副本。选举的方式是向leader_election/插入子节点，第一个插入成功的副本就是主副本

创建第二个副本实例：

```
create table a1(
id String,
price Float64,
create_time DateTime
)ENGINE=ReplicatedMergeTree('/clickhouse/tables/01/a1','hdp-2')
PARTITION BY toYYYYMM(create_time)
ORDER BY id;
```

参与副本选举，hdp-1副本成为主副本。

向第一个副本实例插入数据：

```
insert into table a1 values('A001',100,'2020-08-20 08:00:00');
```

插入命令执行后，在本地完成分区目录的写入，接着向block写入该分区的block_id

```
 ls /clickhouse/tables/01/a1/blocks
[202008_780388339403124112_11960209063776297222]
```

如果设置了inser_quorum参数，且insert_quorum>=2,则hdp-2会进一步监控已完成写入操作的副本个数，直到写入副本个数>= insert_quorum的时候，整个写入操作才算完成。

接下来，hdp-1副本发起向log日志推送操作日志[log-0000000000]

```
[zk: localhost:2181(CONNECTED) 11] ls /clickhouse/tables/01/a1/log
[log-0000000000]
```

操作日式的内容为：

```
[zk: localhost:2181(CONNECTED) 12] get /clickhouse/tables/01/a1/log/log-0000000000
format version: 4
create_time: 2020-08-24 18:26:22
source replica: hdp-1
block_id: 202008_780388339403124112_11960209063776297222
get
202008_0_0_0

cZxid = 0x5000002a0
ctime = Mon Aug 24 18:26:22 CST 2020
mZxid = 0x5000002a0
mtime = Mon Aug 24 18:26:22 CST 2020
pZxid = 0x5000002a0
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 150
numChildren = 0
[zk: localhost:2181(CONNECTED) 13]
```

LogEntry:

source replica: 发送这条Log指令的副本来源，对应replica_name

ENGINE = ReplicatedMergeTree('/clickhouse/tables/{layer}-{shard}/table_name', '{replica_name}')

get: 操作指令类型

```
get:从远程副本下载分区

merge：合并分区

mutate：MUTATION操作
```

block_id：当前分区的blockId,对应/blocks路径下的子节点名称

202008_0_0_0： 当前分区目录的名称

从日志内容可以看到，操作类型为get下载，需要下载的分区是202008_0_0_0，其余所有副本都会基于Log日志以相同的顺序执行。

接下来：第二个副本实例拉取Log日志：

hdp-2会一直监听/log节点变化，当hdp-1推送了/log/log-0000000000之后，hdp-2便会触发日志的拉取任务，并更新log_pointer,

```
[zk: localhost:2181(CONNECTED) 15] get /clickhouse/tables/01/a1/replicas/hdp-2/log_pointer
1
cZxid = 0x500000291
ctime = Mon Aug 24 18:25:32 CST 2020
mZxid = 0x5000002a2
mtime = Mon Aug 24 18:26:22 CST 2020
pZxid = 0x500000291
cversion = 0
dataVersion = 2
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 1
numChildren = 0
```

在拉取LogEntry之后，它并不会立即执行，而是将其转成任务对象放入队列

```
[zk: localhost:2181(CONNECTED) 8] ls /clickhouse/tables/01/a1/replicas/hdp-2/queue
[queue-0000000000]
[zk: localhost:2181(CONNECTED) 9] get /clickhouse/tables/01/a1/replicas/hdp-2/queue/queue-
0000000000
format version: 4
create_time: 2020-08-24 18:26:22
source replica:hdp-1
block_id: 202008_780388339403124112_11960209063776297222
get
202008_0_0_0

cZxid = 0x5000002a2
ctime = Mon Aug 24 18:26:22 CST 2020
mZxid = 0x5000002a2
mtime = Mon Aug 24 18:26:22 CST 2020
pZxid = 0x5000002a2
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 150
numChildren = 0
```

第二个副本实例向其他副本发起下载请求。

当看到type为get的时候，ReplicatedMergeTree会明白在远端的其它副本已经成功写入了数据分区，并根据log_pointer下标做大的下载数据。

hdp-1的DataPartsExchange端口服务就收到调用请求，在得知对方来意之后，根据参数做出相应，将本地的202008_0_0_0基于DataPartsExchange的服务相应发送给hdp-2

8.3 分片

配置文件：

```
<yandex>
    <clickhouse_remote_servers>
        <perftest_3shards_1replicas>  --- 集群的名字
            <shard>
                <internal_replication>true</internal_replication>
                <replica>
                    <host>hdp-1</host>
                    <port>9000</port>
                </replica>
            </shard>
            <shard>
                <replica>
                    <internal_replication>true</internal_replication>
                    <host>hdp-2</host>
                    <port>9000</port>
                </replica>
            </shard>
            <shard>
                <internal_replication>true</internal_replication>
                <replica>
                    <host>hdp-3</host>
                    <port>9000</port>
                </replica>
            </shard>
        </perftest_3shards_1replicas>
    </clickhouse_remote_servers>

    <zookeeper-servers>
        <node index="1">
            <host>hdp-1</host>
            <port>2181</port>
        </node>
        <node index="2">
            <host>hdp-2</host>
            <port>2181</port>
        </node>
        <node index="3">
            <host>hdp-3</host>
            <port>2181</port>
        </node>
    </zookeeper-servers>
```

```
    <macros>
        <shard>01</shard>
        <replica>hdp-1</replica>
    </macros>

    <networks>
        <ip>::/0</ip>
    </networks>

    <clickhouse_compression>
        <case>
            <min_part_size>10000000000</min_part_size>
            <min_part_size_ratio>0.01</min_part_size_ratio>
            <method>lz4</method>
        </case>
    </clickhouse_compression>

</yandex>
```

```
create table clutable on cluster perftest_3shards_1replicas(id UInt64)
engine = ReplicatedMergeTree('/clickhouse/tables/{shard}/clutable','{replica}')
order by id;
```

8.4 Distributed用法

Distributed表引擎：

all： 全局查询的

local：真正的保存数据的表

5.3 Distributed

分布式引擎，本身不存储数据, 但可以在多个服务器上进行分布式查询。 读是自动并行的。读取时，远程服务器表的索引（如果有的话）会被使用。

Distributed(cluster_name, database, table [, sharding_key])

参数解析：

cluster_name - 服务器配置文件中的集群名,在/etc/metrika.xml中配置的

database – 数据库名

table – 表名

sharding_key – 数据分片键

案例演示：

1）在hdp-1，hdp-2，hdp-3上分别创建一个表t

```
:)create table t(id UInt16, name String) ENGINE=TinyLog;
```

2）在三台机器的t表中插入一些数据

```
:)insert into t(id, name) values (1, 'zhangsan');

:)insert into t(id, name) values (2, 'lisi');
```

3）在hdp-1上创建分布式表

```
:)create table dis_table(id UInt16, name String)
ENGINE=Distributed(perftest_3shards_1replicas, default, t, id);
```

4）往dis_table中插入数据

```
:) insert into dis_table select * from t
```

5）查看数据量

```
:) select count() from dis_table

FROM dis_table

┌─count()─┐

|    8 |

└─────────┘

:) select count() from t


SELECT count()

FROM t


┌─count()─┐

|    3 |

└─────────┘
```

可以看到每个节点大约有1/3的数据

附录1 SQL语法

DDL

建库：

create databases mydatabase;

执行完成以后，会在clickhouse的安装路径后生成mydatabase的文件目录：

cd /var/lib/clickhouse/data

ll

在/var/lib/clickhouse/metadata路径下，会生成用于恢复数据库的.sql文件

show databases查询数据库

建表：三种方式

```
create table my_table ( Title String, URL String ,EventTime DateTime) ENGINE=Memory

create table if not exists new_db.hits_v1 as default.hits_v1 engine=TinyLog

create table if not exists hits_v1_1 engine=Memory as select * from hits_v1;
```

查询表结构：

desc

删除表：

drop table

默认值表达式：

```
hdp-1 :) create table dfv_v1 (
:-] id String,
:-] c1 DEFAULT 1000,
:-] c2 String DEFAULT c1
:-] ) engine = TinyLog;

CREATE TABLE dfv_v1
(
    `id` String,
    `c1` DEFAULT 1000,
    `c2` String DEFAULT c1
)
ENGINE = TinyLog

Ok.

0 rows in set. Elapsed: 0.018 sec.

hdp-1 :) insert into dfv_v1(id) values ('A000');
```

```
INSERT INTO dfv_v1 (id) VALUES

Ok.

1 rows in set. Elapsed: 0.009 sec.

hdp-1 :) select c1,c2,toTypeName(c1),toTypeName(c2) from dfv_v1;

SELECT
    c1,
    c2,
    toTypeName(c1),
    toTypeName(c2)
FROM dfv_v1

┌──c1─┬─c2──┬─toTypeName(c1)─┬─toTypeName(c2)─┐
| 1000 | 1000 | UInt16         | String         |
└──────┴─────┴────────────────┴────────────────┘

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

临时表：

语法：

```
 create temporary table tmp_v1 (createtime Datetime);
```

案例：

如果临时表和正常表名字相同，临时表优先

临时表的表引擎只能是Memory，数据是临时的，断电即无的数据。

更多的是应用在clickhouse内部，是数据在集群间传播的载体

```
hdp-1 :) create table tmp_v1( title String)engine = Memory;

CREATE TABLE tmp_v1
(
    `title` String
)
ENGINE = Memory

Ok.

0 rows in set. Elapsed: 0.003 sec.

hdp-1 :) insert into tmp_v1 values ('click');

INSERT INTO tmp_v1 VALUES
```

```
Ok.

1 rows in set. Elapsed: 0.001 sec.

hdp-1 :) create temporary table tmp_v1 (createtime Datetime);

CREATE TEMPORARY TABLE tmp_v1
(
    `createtime` Datetime
)

Ok.

0 rows in set. Elapsed: 0.003 sec.

hdp-1 :) insert into tmp_v1 values (now());

INSERT INTO tmp_v1 VALUES

Ok.

1 rows in set. Elapsed: 0.010 sec.

hdp-1 :) select * from tmp_v1;

SELECT *
FROM tmp_v1

┌──────────createtime─┐
| 2020-10-21 06:31:37 |
└─────────────────────┘
```

分区表：

只有合并数（MergeTree）家族的表引擎支持分区表

可以利用分区表，做定位查询，缩小查询范围。

分区字段不易设的太小

案例：

```
CREATE TABLE partition_v1
(
    `ID` String,
    `URL` String,
    `EventTime` Date
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(EventTime)
```

```
ORDER BY ID

 insert into partition_v1 values ('A000','www.baidu.com','2020-05-01'),
('A001','www.sina.com','2020-06-01');

select table,partition,path from system.parts where table = 'partition_v1';

SELECT
    table,
    partition,
    path
FROM system.parts
WHERE table = 'partition_v1'

┌─table────────┬─partition─┬─path──────────────────────────────────────────────────────┐
| partition_v1 | 202005    | /var/lib/clickhouse/data/default/partition_v1/202005_1_1_0/ |
| partition_v1 | 202006    | /var/lib/clickhouse/data/default/partition_v1/202006_2_2_0/ |
└──────────────┴───────────┴───────────────────────────────────────────────────────┘
```

视图：

普通视图和物化视图

普通视图：不保存数据，只是一层单纯的select查询映射，起着简化查询、明晰语义的作用。

物化视图：保存数据，如果源表被写入新数据，物化视图也会同步更新。

POPULATE修饰符：决定在创建物化视图的过程中是否将源表的数据同步到物化视图里。

数据表的基本操作：

只有MergeTree、Merge、Distribution这三类表引擎支持alter操作。

追加字段，两种方式：

```
1、alter table partition_v1 add column os String default 'mac';

2、alter table partition_v1 add column IP String after ID;
```

```
hdp-1 :) desc partition_v1;

DESCRIBE TABLE partition_v1

┌─name─────┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─ttl_
expression─┐
| ID       | String |              |                    |         |                  |
```

```
|
| URL       | String |              |                    |         |                  |
|
| EventTime | Date   |              |                    |         |                  |
|
└───────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴─────
──────────┘

3 rows in set. Elapsed: 0.001 sec.

hdp-1 :) alter table partition_v1 add column os String default 'mac';

ALTER TABLE partition_v1
    ADD COLUMN `os` String DEFAULT 'mac'


Ok.

0 rows in set. Elapsed: 0.004 sec.

hdp-1 :) desc partition_v1;

DESCRIBE TABLE partition_v1

┌─name──────┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─ttl_
expression─┐
| ID        | String |              |                    |         |                  |
|
| URL       | String |              |                    |         |                  |
|
| EventTime | Date   |              |                    |         |                  |
|
| os        | String | DEFAULT      | 'mac'              |         |                  |
|
└───────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴─────
──────────┘


hdp-1 :) alter table partition_v1 add column IP String after ID;

ALTER TABLE partition_v1
    ADD COLUMN `IP` String     AFTER ID


Ok.

0 rows in set. Elapsed: 0.002 sec.

hdp-1 :) desc partition_v1;

DESCRIBE TABLE partition_v1

┌─name──────┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─ttl_
```

```
expression┐
| ID        | String |              |                    |         |                  |
|
| IP        | String |              |                    |         |                  |
|
| URL       | String |              |                    |         |                  |
|
| EventTime | Date   |              |                    |         |                  |
|
| os        | String | DEFAULT      | 'mac'              |         |                  |
|
└───────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴────
──────────┘

5 rows in set. Elapsed: 0.001 sec.
```

修改数据类型：

alter...modify column...

注意：类型需要相互兼容

```
hdp-1 :) alter table partition_v1 modify column IP IPv4;

ALTER TABLE partition_v1
    MODIFY COLUMN `IP` IPv4


Ok.

0 rows in set. Elapsed: 0.012 sec.

hdp-1 :) desc partition_v1;

DESCRIBE TABLE partition_v1

┌─name──────┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─ttl_
expression─┐
| ID        | String |              |                    |         |                  |
|
| IP        | IPv4   |              |                    |         |                  |
|
| URL       | String |              |                    |         |                  |
|
| EventTime | Date   |              |                    |         |                  |
|
| os        | String | DEFAULT      | 'mac'              |         |                  |
|
└───────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴────
──────────┘
```

修改备注：

alter...comment column ...

```
ALTER TABLE partition_v1
    COMMENT COLUMN ID '主键ID'


Ok.

0 rows in set. Elapsed: 0.006 sec.

hdp-1 :) desc partition_v1;

DESCRIBE TABLE partition_v1

┌─name─────┬─type───┬─default_type─┬─default_expression─┬─comment─┬─codec_expression─┬─ttl_
expression─┐
| ID        | String |              |                    | 主键ID  |                  |
|
| IP        | IPv4   |              |                    |         |                  |
|
| URL       | String |              |                    |         |                  |
|
| EventTime | Date   |              |                    |         |                  |
|
| os        | String | DEFAULT      | 'mac'              |         |                  |
|
└──────────┴────────┴──────────────┴────────────────────┴─────────┴──────────────────┴─────
──────────┘

5 rows in set. Elapsed: 0.001 sec.
```

删除已有字段：

会把该字段下的数据一起删除

```
alter table partition_v1 drop column URL
```

移动数据表

rename... to...

注意：表的移动只能在单节点内完成

```
rename table default.partition_v1 to system.partition_v1;
```

数据分区的基本操作：

查询分区信息：

ClickHouse内置了很多system系统表，用于查询自身状态信息。

```
USE system

Ok.

0 rows in set. Elapsed: 0.001 sec.

hdp-1 :) show tables;

SHOW TABLES

┌─name─────────────────────────┐
| aggregate_function_combinators |
| asynchronous_metric_log        |
| asynchronous_metrics           |
| build_options                  |
| clusters                       |
| collations                     |
| columns                        |
| contributors                   |
| current_roles                  |
| data_type_families             |
| databases                      |
| detached_parts                 |
| dictionaries                   |
| disks                          |
| distribution_queue             |
| enabled_roles                  |
| events                         |
| formats                        |
| functions                      |
| grants                         |
| graphite_retentions            |
| licenses                       |
| macros                         |
| merge_tree_settings            |
| merges                         |
| metric_log                     |
| metrics                        |
| models                         |
| mutations                      |
| numbers                        |
| numbers_mt                     |
| one                            |
| parts                          |
| parts_columns                  |
| privileges                     |
```

```
| processes                       |
| query_log                       |
| query_thread_log                |
| quota_limits                    |
| quota_usage                     |
| quotas                          |
| quotas_usage                    |
| replicas                        |
| replication_queue               |
| role_grants                     |
| roles                           |
| row_policies                    |
| settings                        |
| settings_profile_elements       |
| settings_profiles               |
| stack_trace                     |
| storage_policies                |
| table_engines                   |
| table_functions                 |
| tables                          |
| trace_log                       |
| users                           |
| zeros                           |
| zeros_mt                        |
| zookeeper                       |
└─────────────────────────────────┘

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

其中parts是专门用来查询分区信息的表

desc parts

```
┌─name──────────────────────────────┬─type─────────┬─default_type─┬─default_expressio
n
| partition                         | String       |              |
| name                              | String       |              |
| part_type                         | String       |              |
| active                            | UInt8        |              |
| marks                             | UInt64       |              |
| rows                              | UInt64       |              |
| bytes_on_disk                     | UInt64       |              |
| data_compressed_bytes             | UInt64       |              |
| data_uncompressed_bytes           | UInt64       |              |
| marks_bytes                       | UInt64       |              |
| modification_time                 | DateTime     |              |
| remove_time                       | DateTime     |              |
| refcount                          | UInt32       |              |
| min_date                          | Date         |              |
| max_date                          | Date         |              |
| min_time                          | DateTime     |              |
| max_time                          | DateTime     |              |
| partition_id                      | String       |              |
```

```
| min_block_number                          | Int64           |               |
| max_block_number                          | Int64           |               |
| level                                     | UInt32          |               |
| data_version                              | UInt64          |               |
| primary_key_bytes_in_memory               | UInt64          |               |
| primary_key_bytes_in_memory_allocated     | UInt64          |               |
| is_frozen                                 | UInt8           |               |
| database                                  | String          |               |
| table                                     | String          |               |
| engine                                    | String          |               |
| disk_name                                 | String          |               |
| path                                      | String          |               |
| hash_of_all_files                         | String          |               |
| hash_of_uncompressed_files                | String          |               |
| uncompressed_hash_of_compressed_files     | String          |               |
| delete_ttl_info_min                       | DateTime        |               |
| delete_ttl_info_max                       | DateTime        |               |
| move_ttl_info.expression                  | Array(String)   |               |
| move_ttl_info.min                         | Array(DateTime) |               |
| move_ttl_info.max                         | Array(DateTime) |               |
| bytes                                     | UInt64          | ALIAS         | bytes_on_disk
| marks_size                                | UInt64          | ALIAS         | marks_bytes
└───────────────────────────────────────────┴─────────────────┴───────────────┴──────────────────
_
```

```
hdp-1 :) select partition_id,name,table,database from system.parts where table =
'partition_v1';

SELECT
    partition_id,
    name,
    table,
    database
FROM system.parts
WHERE table = 'partition_v1'

┌─partition_id─┬─name──────────┬─table────────┬─database─┐
| 202005       | 202005_1_1_0_3 | partition_v1 | default  |
| 202006       | 202006_2_2_0_3 | partition_v1 | default  |
└──────────────┴────────────────┴──────────────┴──────────┘

2 rows in set. Elapsed: 0.006 sec.

hdp-1 :)
```

删除指定分区：

```
alter table partition_v1 drop partition 202005
```

可以利用删除完成更新操作---先删除，再insert插入

复制分区数据

需要满足两个条件

1、两张表需要有相同的分区键

2、两张表需要有相同的表结构

```
alter table partition_v2 replace partition 202005 from partition_v1;
```

重置分区数据

```
alter table partition_v1 clear clumn URL in partition 202005
```

卸载和装载分区

卸载：alter table ...detach..

装载： alter table ...attach...

```
alter table partition_v1 detach partition 202005
alter table partition_v1 attach partition 202005
```

分布式DDL执行

在集群上的任意一个节点上执行DDL语句，那么集群上的任意一个节点都会以相同的顺序执行相同的语义。

数据写入：

方式1：

```
insert into partition_v1 values (...)
```

方式2：指定格式 format CSV

```
[root@hdp-1 input]# cat mycsv.csv
A003,0.0.0.0,www.aa.com,2020-05-05,mac
A004,0.0.0.0,www.cc.com,2020-05-06,window
```

```
[root@hdp-1 input]# cat mycsv.csv | clickhouse-client --query="insert into
default.partition_v1 format CSV"

hdp-1 :) select * from partition_v1;

SELECT *
FROM partition_v1

┌─ID──┬─────IP─┬─URL─────────┬──EventTime─┬─os────┐
| A003 | 0.0.0.0 | www.aa.com | 2020-05-05 | mac    |
| A004 | 0.0.0.0 | www.cc.com | 2020-05-06 | window |
└─────┴────────┴─────────────┴────────────┴────────┘
┌─ID──┬─────IP─┬─URL─────────────┬──EventTime─┬─os──┐
| A000 | 0.0.0.0 | www.baidu.com | 2020-05-01 | mac |
└─────┴────────┴─────────────────┴────────────┴─────┘
┌─ID──┬─────IP─┬─URL─────────────┬──EventTime─┬─os──┐
| A001 | 0.0.0.0 | www.sina.com | 2020-06-01 | mac |
└─────┴────────┴─────────────────┴────────────┴─────┘
┌─ID──┬─────IP─┬─URL─────────────────┬──EventTime─┬─os────┐
| A002 | 0.0.0.0 | www.youtube.com | 2020-05-03 | window |
└─────┴────────┴─────────────────────┴────────────┴────────┘

5 rows in set. Elapsed: 0.004 sec.
```

创建partition_v2

```
hdp-1 :) create table partition_v2 (ID String, IP IPv4, URL String,EventTime Date, os
String) engine=MergeTree partition by toYYYYMM(EventTime) order by ID;

CREATE TABLE partition_v2
(
    `ID` String,
    `IP` IPv4,
    `URL` String,
    `EventTime` Date,
    `os` String
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(EventTime)
ORDER BY ID
```

方式3：select方式

```
hdp-1 :) insert into partition_v2 select * from partition_v1;

INSERT INTO partition_v2 SELECT *
FROM partition_v1

Ok.
```

```
0 rows in set. Elapsed: 0.006 sec.

hdp-1 :) select * from partition_v2;

SELECT *
FROM partition_v2

┌─ID──┬──────IP─┬─URL─────────┬──EventTime─┬─os──┐
│ A001 │ 0.0.0.0 │ www.sina.com │ 2020-06-01 │ mac │
└──────┴─────────┴──────────────┴────────────┴─────┘
┌─ID──┬──────IP─┬─URL─────────────┬──EventTime─┬─os─────┐
│ A000 │ 0.0.0.0 │ www.baidu.com   │ 2020-05-01 │ mac    │
│ A002 │ 0.0.0.0 │ www.youtube.com │ 2020-05-03 │ window │
│ A003 │ 0.0.0.0 │ www.aa.com      │ 2020-05-05 │ mac    │
│ A004 │ 0.0.0.0 │ www.cc.com      │ 2020-05-06 │ window │
└──────┴─────────┴─────────────────┴────────────┴────────┘
```

1.1 CREATE

1.1.1 CREATE DATABASE

用于创建指定名称的数据库，语法如下：

CREATE DATABASE [IF NOT EXISTS] db_name

如果查询中存在IF NOT EXISTS，则当数据库已经存在时，该查询不会返回任何错误。

```
:) create database test;
Ok.
0 rows in set. Elapsed: 0.018 sec.
```

1.1.2 CREATE TABLE

对于创建表，语法如下：

CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]

(

name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],

name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],

...

) ENGINE = engine

DEFAULT expr – 默认值，用法与SQL类似。

MATERIALIZED expr – 物化表达式，被该表达式指定的列不能被INSERT，因为它总是被计算出来的。 对于INSERT而言，不需要考虑这些列。 另外，在SELECT查询中如果包含星号，此列不会被查询。

ALIAS expr – 别名。

有三种方式创建表：

1）直接创建

:) create table t1(id UInt16,name String) engine=TinyLog

2）创建一个与其他表具有相同结构的表

CREATE TABLE [IF NOT EXISTS] [db.]table_name AS [db2.]name2 [ENGINE = engine]

可以对其指定不同的表引擎声明。如果没有表引擎声明，则创建的表将与db2.name2使用相同的表引擎。

:) create table t2 as t1 engine=Memory

:) desc t2

DESCRIBE TABLE t2

```
┌─name─┬─type───┬─default_type─┬─default_expression─┐
│ id   │ UInt16 │              │                    │
│ name │ String │              │                    │
└──────┴────────┴──────────────┴────────────────────┘
```

3）使用指定的引擎创建一个与SELECT子句的结果具有相同结构的表，并使用SELECT子句的结果填充它。

语法：

CREATE TABLE [IF NOT EXISTS] [db.]table_name ENGINE = engine AS SELECT ...

实例：

先在t2中插入几条数据

:) insert into t1 values(1,'zhangsan'),(2,'lisi'),(3,'wangwu')

:) create table t3 engine=TinyLog as select * from t1

:) select * from t3

```
┌─id─┬─name─────┐
│ 1  │ zhangsan │
│ 2  │ lisi     │
│ 3  │ wangwu   │
└────┴──────────┘
```

1.2 INSERT INTO

主要用于向表中添加数据，基本格式如下：

INSERT INTO [db.]table [(c1, c2, c3)] VALUES (v11, v12, v13), (v21, v22, v23), ...

实例：

:) insert into t1 values(1,'zhangsan'),(2,'lisi'),(3,'wangwu')

还可以使用select来写入数据：

INSERT INTO [db.]table [(c1, c2, c3)] SELECT ...

实例：

:) insert into t2 select * from t3

:) select * from t2

```
┌─id─┬─name─────┐
│ 1 │ zhangsan │
│ 2 │ lisi │
│ 3 │ wangwu │
└────┴──────────┘
```

ClickHouse不支持的修改数据的查询：UPDATE, DELETE, REPLACE, MERGE, UPSERT, INSERT UPDATE。

1.3 ALTER

ALTER只支持MergeTree系列，Merge和Distributed引擎的表，基本语法：

ALTER TABLE [db].name [ON CLUSTER cluster] ADD|DROP|MODIFY COLUMN ...

参数解析：

ADD COLUMN – 向表中添加新列

DROP COLUMN – 在表中删除列

MODIFY COLUMN – 更改列的类型

案例演示：

1）创建一个MergerTree引擎的表

create table mt_table (date Date, id UInt8, name String) ENGINE=MergeTree(date, (id, name), 8192);

2）向表中插入一些值

insert into mt_table values ('2019-05-01', 1, 'zhangsan');

insert into mt_table values ('2019-06-01', 2, 'lisi');

insert into mt_table values ('2019-05-03', 3, 'wangwu');

3）在末尾添加一个新列age

:)alter table mt_table add column age UInt8

:)desc mt_table

```
┌─name─┬─type───┬─default_type─┬─default_expression─┐
│ date │ Date   │              │                    │
│ id   │ UInt8  │              │                    │
│ name │ String │              │                    │
│ age  │ UInt8  │              │                    │
└──────┴────────┴──────────────┴────────────────────┘
```

:) select * from mt_table

```
┌──────date─┬─id─┬─name─┬─age─┐
│ 2019-06-01 │ 2 │ lisi │ 0 │
└────────────┴────┴──────┴─────┘

┌──────date─┬─id─┬─name─────┬─age─┐
│ 2019-05-01 │ 1 │ zhangsan │ 0 │
│ 2019-05-03 │ 3 │ wangwu   │ 0 │
└────────────┴────┴──────────┴─────┘
```

4）更改age列的类型

:)alter table mt_table modify column age UInt16

:)desc mt_table

```
┌─name─┬─type───┬─default_type─┬─default_expression─┐
│ date │ Date   │              │                    │
│ id   │ UInt8  │              │                    │
│ name │ String │              │                    │
│ age  │ UInt16 │              │                    │
└──────┴────────┴──────────────┴────────────────────┘
```

5）删除刚才创建的age列

:)alter table mt_table drop column age

:)desc mt_table

```
┌─name─┬─type───┬─default_type─┬─default_expression─┐
│ date │ Date   │              │                    │
│ id   │ UInt8  │              │                    │
│ name │ String │              │                    │
```

└─────────┴────────────┴──────────────────┴──────────────────────────────┘

1.4 DESCRIBE TABLE

查看表结构

:)desc mt_table

┌─name─┬─type─────┬─default_type─┬─default_expression─┐

│ date │ Date │ │ │

│ id │ UInt8 │ │ │

│ name │ String │ │ │

└─────────┴────────────┴──────────────────┴──────────────────────────────┘

1.5 CHECK TABLE

检查表中的数据是否损坏，他会返回两种结果：

0 – 数据已损坏

1 – 数据完整

该命令只支持Log，TinyLog和StripeLog引擎。

附录2 选择查询 附录

2.1 with子句

本节提供对公共表表达式的支持 (CTE），所以结果 WITH 子句可以在其余部分中使用 SELECT 查询。

限制

1. 不支持递归查询。
2. 当在section中使用子查询时，它的结果应该是只有一行的标量。
3. Expression的结果在子查询中不可用。

例 示例1: 使用常量表达式作为 “variable”

```
hdp-1 :) with '2020-08-02 14:20:22' as tm select toDate(tm);

WITH '2020-08-02 14:20:22' AS tm
SELECT toDate(tm)

┌─toDate(tm)─┐
| 2020-08-02 |
└────────────┘

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

```
WITH '2019-08-01 15:23:00' as ts_upper_bound
SELECT *
FROM hits
WHERE
    EventDate = toDate(ts_upper_bound) AND
    EventTime <= ts_upper_bound
```

示例2: 从SELECT子句列表中逐出sum(bytes)表达式结果

```
WITH sum(bytes) as s
SELECT
    formatReadableSize(s),
    table
FROM system.parts
GROUP BY table
ORDER BY s
```

例3: 使用标量子查询的结果

没有 where acitive条件限制

```
hdp-1 :) with sum(bytes) as s
:-] select formatReadableSize(s) from parts;

WITH sum(bytes) AS s
SELECT formatReadableSize(s)
FROM parts

┌─formatReadableSize(s)─┐
│ 1.36 GiB              │
└───────────────────────┘

1 rows in set. Elapsed: 0.009 sec.
```

有where active条件限制

```
hdp-1 :) with sum(bytes) as s
:-] select formatReadableSize(s)
:-] from parts
:-] where active;

WITH sum(bytes) AS s
SELECT formatReadableSize(s)
FROM parts
WHERE active

┌─formatReadableSize(s)─┐
│ 1.24 GiB              │
└───────────────────────┘
```

```
1 rows in set. Elapsed: 0.003 sec.
```

-- 不用with定义变量的写法

```
hdp-1 :) select formatReadableSize(sum(bytes)) as f, table from parts group by table;

SELECT
    formatReadableSize(sum(bytes)) AS f,
    table
FROM parts
GROUP BY table

┌─f──────────┬─table──────────────────┐
| 11.62 KiB  | mt_table               |
| 277.00 B   | r1                     |
| 1.18 GiB   | hits_v1                |
| 182.74 MiB | metric_log             |
| 1.49 MiB   | hot_cold_table         |
| 1.14 KiB   | partition_v1           |
| 38.65 KiB  | trace_log              |
| 185.61 KiB | query_thread_log       |
| 1.50 KiB   | mt_table2              |
| 1.26 KiB   | summing_table          |
| 189.00 B   | ttl_table_v1           |
| 189.25 KiB | query_log              |
| 233.00 B   | jbod_table             |
| 553.00 B   | replace_table          |
| 900.00 B   | partition_v2           |
| 1.01 MiB   | hot_cold_table_new     |
| 4.92 MiB   | asynchronous_metric_log |
└────────────┴────────────────────────┘

17 rows in set. Elapsed: 0.010 sec.
```

相同结果，用with定义变量

```
hdp-1 :) with sum(bytes) as a select formatReadableSize(a) as s,table from parts group by
table;

WITH sum(bytes) AS a
SELECT
    formatReadableSize(a) AS s,
    table
FROM parts
GROUP BY table

┌─s──────────┬─table──────────────────┐
| 11.62 KiB  | mt_table               |
| 277.00 B   | r1                     |
```

```
| 1.18 GiB   | hits_v1                |
| 182.37 MiB | metric_log             |
| 1.49 MiB   | hot_cold_table         |
| 1.14 KiB   | partition_v1           |
| 38.65 KiB  | trace_log              |
| 228.61 KiB | query_thread_log       |
| 1.50 KiB   | mt_table2              |
| 1.26 KiB   | summing_table          |
| 189.00 B   | ttl_table_v1           |
| 230.87 KiB | query_log              |
| 233.00 B   | jbod_table             |
| 553.00 B   | replace_table          |
| 900.00 B   | partition_v2           |
| 1.01 MiB   | hot_cold_table_new     |
| 4.92 MiB   | asynchronous_metric_log |
└────────────┴────────────────────────┘

17 rows in set. Elapsed: 0.014 sec.
```

```
/* this example would return TOP 10 of most huge tables */
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
```

例4: 在子查询中重用表达式

作为子查询中表达式使用的当前限制的解决方法，您可以复制它。

```
WITH ['hello'] AS hello
SELECT
    hello,
    *
FROM
(
    WITH ['hello'] AS hello
    SELECT hello
)
┌─hello─────┬─hello─────┐
| ['hello'] | ['hello'] |
└───────────┴───────────┘
```

2.2 FROM子句

FROM 子句指定从以下数据源中读取数据:

- 表
- 子查询
- 表函数

JOIN 和 ARRAY JOIN 子句也可以用来扩展 FROM 的功能

子查询是另一个 SELECT 可以指定在 FROM 后的括号内的查询。

FROM 子句可以包含多个数据源，用逗号分隔，这相当于在他们身上执行 CROSS JOIN

FINAL 修饰符

当 FINAL 被指定，ClickHouse会在返回结果之前完全合并数据，从而执行给定表引擎合并期间发生的所有数据转换。

它适用于从使用 MergeTree-引擎族（除了 GraphiteMergeTree). 还支持:

- Replicated 版本 MergeTree 引擎
- View, Buffer, Distributed，和 MaterializedView 在其他引擎上运行的引擎，只要是它们底层是 MergeTree-引擎表即可。

缺点

使用的查询 FINAL 执行速度不如类似的查询那么快，因为:

- 查询在单个线程中执行，并在查询执行期间合并数据。
- 查询与 FINAL 除了读取查询中指定的列之外，还读取主键列。

在大多数情况下，避免使用 FINAL. 常见的方法是使用假设后台进程的不同查询 MergeTree 引擎还没有发生，并通过应用聚合（例如，丢弃重复项）来处理它。

实现细节

如果 FROM 子句被省略，数据将从读取 system.one 表。 该 system.one 表只包含一行（此表满足与其他 DBMS 中的 DUAL 表有相同的作用）。

若要执行查询，将从相应的表中提取查询中列出的所有列。 外部查询不需要的任何列都将从子查询中抛出。 如果查询未列出任何列（例如, SELECT count() FROM t），无论如何都会从表中提取一些列（首选是最小的列），以便计算行数。

2.3 SAMPLE 子句

该 SAMPLE 子句允许近似于 SELECT 查询处理。

启用数据采样时，不会对所有数据执行查询，而只对特定部分数据（样本）执行查询。 例如，如果您需要计算所有访问的统计信息，只需对所有访问的1/10分数执行查询，然后将结果乘以10即可。

近似查询处理在以下情况下可能很有用:

- 当你有严格的时间需求（如<100ms），但你不能通过额外的硬件资源来满足他们的成本。
- 当您的原始数据不准确时，所以近似不会明显降低质量。
- 业务需求的目标是近似结果（为了成本效益，或者向高级用户推销确切结果）。

注

您只能使用采样中的表 MergeTree 族，并且只有在表创建过程中指定了采样表达式（请参阅 MergeTree引擎).

下面列出了数据采样的功能:

- 数据采样是一种确定性机制。 同样的结果 SELECT .. SAMPLE 查询始终是相同的。
- 对于不同的表，采样工作始终如一。 对于具有单个采样键的表，具有相同系数的采样总是选择相同的可能数据子集。 例如，用户Id的示例采用来自不同表的所有可能的用户Id的相同子集的行。 这意味着您可以在子查询中使用采样 IN 此外，您可以使用 JOIN 。
- 采样允许从磁盘读取更少的数据。 请注意，您必须正确指定采样键。 有关详细信息，请参阅 创建MergeTree表.

为 SAMPLE 子句支持以下语法:

SAMPLE Clause Syntax	**产品描述**
SAMPLE k	这里 k 是从0到1的数字。

查询执行于 k`SAMPLE 0.1Read moreSAMPLE n`n`n`SAMPLE 10000000Read moreSAMPLE k OFFSET m`k`m`k`mRead more

SAMPLE K

这里 k 从0到1的数字（支持小数和小数表示法）。 例如, SAMPLE 1/2 或 SAMPLE 0.5.

注意：为了得到最终的统计结果，需要将sample查询结果乘系数

在一个 SAMPLE k 子句，样品是从 k 数据的分数。 示例如下所示:

```
hdp-1 :) select count(CounterID) from hits_v1 sample 0.1;

SELECT count(CounterID)
FROM hits_v1
SAMPLE 1 / 10
```

```
┌─count(CounterID)─┐
│           839889 │
└──────────────────┘

1 rows in set. Elapsed: 0.033 sec. Processed 7.36 million rows, 88.30 MB (223.32 million rows/s., 2.68 GB/s.)

hdp-1 :) select count(CounterID) from hits_v1;

SELECT count(CounterID)
FROM hits_v1

┌─count(CounterID)─┐
│          8873898 │
└──────────────────┘

1 rows in set. Elapsed: 0.015 sec. Processed 8.87 million rows, 35.50 MB (579.86 million rows/s., 2.32 GB/s.)

hdp-1 :)
```

利用虚拟字段_sample_factor来获取采样系数

```
hdp-1 :) select CounterID,_sample_factor from hits_v1 sample 0.1 limit 2;

SELECT
    CounterID,
    _sample_factor
FROM hits_v1
SAMPLE 1 / 10
LIMIT 2

┌─CounterID─┬─_sample_factor─┐
│        57 │             10 │
└───────────┴────────────────┘
┌─CounterID─┬─_sample_factor─┐
│        57 │             10 │
└───────────┴────────────────┘

2 rows in set. Elapsed: 0.004 sec.

hdp-1 :) select CounterID,_sample_factor from hits_v1 limit 2;

SELECT
    CounterID,
    _sample_factor
FROM hits_v1
LIMIT 2

┌─CounterID─┬─_sample_factor─┐
│        57 │              1 │
│        57 │              1 │
└───────────┴────────────────┘
```

```
2 rows in set. Elapsed: 0.003 sec.

hdp-1 :) select CounterID,_sample_factor from hits_v1 sample 0.2 limit 2;

SELECT
    CounterID,
    _sample_factor
FROM hits_v1
SAMPLE 2 / 10
LIMIT 2

┌─CounterID─┬─_sample_factor─┐
│        57 │              5 │
└───────────┴────────────────┘
┌─CounterID─┬─_sample_factor─┐
│        57 │              5 │
└───────────┴────────────────┘

2 rows in set. Elapsed: 0.006 sec.
```

```
SELECT
    Title,
    count() * 10 AS PageViews
FROM hits_distributed
SAMPLE 0.1
WHERE
    CounterID = 34
GROUP BY Title
ORDER BY PageViews DESC LIMIT 1000
```

在此示例中，对0.1(10%)数据的样本执行查询。 聚合函数的值不会自动修正，因此要获得近似结果，值 count() 手动乘以10。

SAMPLE N

这里 n 是足够大的整数。 例如, SAMPLE 10000000.

在这种情况下，查询在至少一个样本上执行 n 行（但不超过这个）。 例如, SAMPLE 10000000 在至少10,000,000行上运行查询。

由于数据读取的最小单位是一个颗粒（其大小由 index_granularity 设置），是有意义的设置一个样品，其大小远大于颗粒。

使用时 SAMPLE n 子句，你不知道处理了哪些数据的相对百分比。 所以你不知道聚合函数应该乘以的系数。 使用 _sample_factor 虚拟列得到近似结果。

该 _sample_factor 列包含动态计算的相对系数。 当您执行以下操作时，将自动创建此列 创建 具有指定采样键的表。 的使用示例 _sample_factor 列如下所示。

```
hdp-1 :) select count() from hits_v1 sample 10000;

SELECT count()
FROM hits_v1
SAMPLE 10000

┌─count()─┐
│    9251 │
└─────────┘

1 rows in set. Elapsed: 0.066 sec. Processed 7.15 million rows, 119.16 MB (108.92 million
rows/s., 1.82 GB/s.)
```

```
hdp-1 :) select CounterID,_sample_factor from hits_v1 sample 10000 limit 1;

SELECT
    CounterID,
    _sample_factor
FROM hits_v1
SAMPLE 10000
LIMIT 1

┌─CounterID─┬─_sample_factor─┐
│      1294 │       886.3744 │
└───────────┴────────────────┘

1 rows in set. Elapsed: 0.045 sec. Processed 7.07 thousand rows, 84.88 KB (156.61 thousand
rows/s., 1.88 MB/s.)
```

让我们考虑表 visits，其中包含有关网站访问的统计信息。 第一个示例演示如何计算页面浏览量:

```
hdp-1 :) select sum(PageViews) from visits_v1;

SELECT sum(PageViews)
FROM visits_v1

┌─sum(PageViews)─┐
│        6738137 │
└────────────────┘

1 rows in set. Elapsed: 0.010 sec. Processed 1.68 million rows, 6.71 MB (160.58 million
rows/s., 642.32 MB/s.)
```

```
hdp-1 :) select sum(PageViews * _sample_factor) from visits_v1 sample 10000000;

SELECT sum(PageViews * _sample_factor)
FROM visits_v1
SAMPLE 10000000

┌─sum(multiply(PageViews, _sample_factor))─┐
│                                  6738137 │
└──────────────────────────────────────────┘

1 rows in set. Elapsed: 0.014 sec. Processed 1.68 million rows, 6.71 MB (123.55 million
rows/s., 494.20 MB/s.)
```

下一个示例演示如何计算访问总数:

```
hdp-1 :) select sum(_sample_factor) from visits_v1 sample 10000000;

SELECT sum(_sample_factor)
FROM visits_v1
SAMPLE 10000000

┌─sum(_sample_factor)─┐
│             1676861 │
└─────────────────────┘

1 rows in set. Elapsed: 0.003 sec. Processed 1.68 million rows, 1.68 MB (529.35 million
rows/s., 529.35 MB/s.)
```

下面的示例显示了如何计算平均会话持续时间。 请注意，您不需要使用相对系数来计算平均值。

```
hdp-1 :) select avg(Duration) from visits_v1 sample 10000000;

SELECT avg(Duration)
FROM visits_v1
SAMPLE 10000000

┌──────avg(Duration)─┐
│ 359.92117176080785 │
└────────────────────┘

1 rows in set. Elapsed: 0.011 sec. Processed 1.68 million rows, 6.71 MB (148.60 million
rows/s., 594.38 MB/s.)

hdp-1 :)
```

SAMPLE K OFFSET M

这里 k 和 m 是从0到1的数字。 示例如下所示。

示例1

```
SAMPLE 1/10
```

在此示例中，示例是所有数据的十分之一:

```
[++----------]
```

示例2

```
SAMPLE 1/10 OFFSET 1/2
```

这里，从数据的后半部分1/2取出10%的样本。如果10%超出自动截断

2.4 JOIN子句

array join

```
hdp-1 :) use default;

USE default

Ok.

0 rows in set. Elapsed: 0.001 sec.

hdp-1 :) create table query_v1 (title String, value Array(UInt8))engine=Log;

CREATE TABLE query_v1
(
    `title` String,
    `value` Array(UInt8)
)
ENGINE = Log

Ok.

0 rows in set. Elapsed: 0.002 sec.

hdp-1 :) show tables;

SHOW TABLES

┌─name──────────────┐
| UUID_TEST         |
| dfv_v1            |
| hot_cold_table    |
| hot_cold_table_new|
| jbod_table        |
```

```
| jdbc_example         |
| mt_table             |
| mt_table2            |
| partition_v1         |
| partition_v2         |
| query_v1             |
| r1                   |
| replace_table        |
| summing_table        |
| summing_table_nested |
| tmp_v1               |
| ttl_table_v1         |
└──────────────────────┘

17 rows in set. Elapsed: 0.001 sec.

hdp-1 :) insert into query_v1 values('student',[1,2,3]),('teacher',[4,5]),('suguan',[]);

INSERT INTO query_v1 VALUES

Ok.

3 rows in set. Elapsed: 0.004 sec.

hdp-1 :) select * from query_v1;

SELECT *
FROM query_v1

┌─title───┬─value───┐
| student | [1,2,3] |
| teacher | [4,5]   |
| suguan  | []      |
└─────────┴─────────┘

3 rows in set. Elapsed: 0.001 sec.

hdp-1 :) select title,value from query_v1 array join value;

SELECT
    title,
    value
FROM query_v1
ARRAY JOIN value

┌─title───┬─value─┐
| student |     1 |
| student |     2 |
| student |     3 |
| teacher |     4 |
| teacher |     5 |
└─────────┴───────┘
```

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

hdp-1 :) select title,value,v from query_v1 array join value as v;

SELECT
    title,
    value,
    v
FROM query_v1
ARRAY JOIN value AS v

┌─title───┬─value───┬─v─┐
| student | [1,2,3] | 1 |
| student | [1,2,3] | 2 |
| student | [1,2,3] | 3 |
| teacher | [4,5]   | 4 |
| teacher | [4,5]   | 5 |
└─────────┴─────────┴───┘

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

arrayMap(func, arr1, ...)

Returns an array obtained from the original application of the `func` function to each element in the `arr` array.

Examples:

```
SELECT arrayMap(x -> (x + 2), [1, 2, 3]) as res;
┌─res─────┐
| [3,4,5] |
└─────────┘
```

The following example shows how to create a tuple of elements from different arrays:

```
SELECT arrayMap((x, y) -> (x, y), [1, 2, 3], [4, 5, 6]) AS res
┌─res─────────────────┐
| [(1,4),(2,5),(3,6)] |
└─────────────────────┘
```

Note that the `arrayMap` is a higher-order function. You must pass a lambda function to it as the first argument, and it can't be omitted.

```
hdp-1 :) select arrayMap(x->x*2,value) from query_v1;

SELECT arrayMap(x -> (x * 2), value)
FROM query_v1

┌─arrayMap(lambda(tuple(x), multiply(x, 2)), value)─┐
| [2,4,6]                                           |
| [8,10]                                            |
| []                                                |
└───────────────────────────────────────────────────┘

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

```
hdp-1 :) select title,value,v,arrayMap(x->x*2,value) as mapv,v_1 from query_v1 left array
join value as v,mapv as v_1;

SELECT
    title,
    value,
    v,
    arrayMap(x -> (x * 2), value) AS mapv,
    v_1
FROM query_v1
LEFT ARRAY JOIN
    value AS v,
    mapv AS v_1

┌─title───┬─value───┬─v─┬─mapv────┬─v_1─┐
| student | [1,2,3] | 1 | [2,4,6] |   2 |
| student | [1,2,3] | 2 | [2,4,6] |   4 |
| student | [1,2,3] | 3 | [2,4,6] |   6 |
| teacher | [4,5]   | 4 | [8,10]  |   8 |
| teacher | [4,5]   | 5 | [8,10]  |  10 |
| suguan  | []      | 0 | []      |   0 |
└─────────┴─────────┴───┴─────────┴─────┘

6 rows in set. Elapsed: 0.002 sec.

hdp-1 :) select title,value,v from query_v1 left array join value as v;

SELECT
    title,
    value,
    v
FROM query_v1
LEFT ARRAY JOIN value AS v

┌─title───┬─value───┬─v─┐
| student | [1,2,3] | 1 |
```

```
| student | [1,2,3] | 2 |
| student | [1,2,3] | 3 |
| teacher | [4,5]   | 4 |
| teacher | [4,5]   | 5 |
| suguan  | []      | 0 |
└─────────┴─────────┴───┘
```

JOIN

三张表：

```
create table join_tb1 (id UInt8,name String,time DateTime) engine=Log;
insert into join_tb1 values(1,'ClickHouse','2020-02-01 12:00:00');
insert into join_tb1 values(2,'Spark','2020-02-01 12:30:00');
insert into join_tb1 values(3,'ElasticSearch','2020-02-01 13:00:00');
insert into join_tb1 values(4,'HBase','2020-02-01 13:30:00');
insert into join_tb1 (name,time)values('ClickHouse','2020-02-01 12:00:00');
insert into join_tb1 (name,time)values('Spark','2020-02-01 12:00:00');

create table join_tb2 (id UInt8, rate UInt8, time DateTime) engine=Log;
insert into join_tb2 values(1,100,'2020-05-02 10:00:00');
insert into join_tb2 values(2,90,'2020-05-02 10:01:00');
insert into join_tb2 values(3,80,'2020-05-02 10:02:00');
insert into join_tb2 values(5,70,'2020-05-02 10:03:00');
insert into join_tb2 values(6,60,'2020-05-02 10:04:00');

create table join_tb3 (id UInt8, star UInt8) engine=Log;
insert into join_tb3 values(1,1000);
insert into join_tb3 values(2,900);
```

```
SELECT *
FROM join_tb1

┌─id─┬─name──────────┬────────────────time─┐
│  1 | ClickHouse    | 2020-02-01 12:00:00 |
│  2 | Spark         | 2020-02-01 12:30:00 |
│  3 | ElasticSearch | 2020-02-01 13:00:00 |
└────┴───────────────┴─────────────────────┘
┌─id─┬─name────────┬────────────────time─┐
│  4 | HBase       | 2020-02-01 13:30:00 |
│  0 | ClickHouse | 2020-02-01 12:00:00 |
│  0 | Spark       | 2020-02-01 12:00:00 |
└────┴─────────────┴─────────────────────┘
```

```
SELECT *
FROM join_tb2

┌─id─┬─rate─┬────────────time─┐
│  1 │  100 │ 2020-05-02 10:00:00 │
│  2 │   90 │ 2020-05-02 10:01:00 │
└────┴──────┴─────────────────────┘
┌─id─┬─rate─┬────────────time─┐
│  3 │   80 │ 2020-05-02 10:02:00 │
│  5 │   70 │ 2020-05-02 10:03:00 │
│  6 │   60 │ 2020-05-02 10:04:00 │
└────┴──────┴─────────────────────┘
```

```
SELECT *
FROM join_tb3

┌─id─┬─star─┐
│  1 │  232 │
└────┴──────┘
┌─id─┬─star─┐
│  2 │  132 │
└────┴──────┘
```

查询：ALL

```
hdp-1 :) select a.id,a.name,b.rate from join_tb1 as a all inner join join_tb2 as b on
a.id=b.id;

SELECT
    a.id,
    a.name,
    b.rate
FROM join_tb1 AS a
ALL INNER JOIN join_tb2 AS b ON a.id = b.id

┌─id─┬─name──────────┬─rate─┐
│  1 │ ClickHouse    │  100 │
│  1 │ ClickHouse    │  104 │
│  2 │ Spark         │   90 │
│  3 │ ElasticSearch │   80 │
└────┴───────────────┴──────┘

4 rows in set. Elapsed: 0.003 sec.
```

ANY

```
hdp-1 :) select a.id,a.name,b.rate from join_tb1 as a any inner join join_tb2 as b on
a.id=b.id;

SELECT
    a.id,
    a.name,
    b.rate
FROM join_tb1 AS a
ANY INNER JOIN join_tb2 AS b ON a.id = b.id

┌─id─┬─name─────────┬─rate─┐
│  1 │ ClickHouse   │  100 │
│  2 │ Spark        │   90 │
│  3 │ ElasticSearch │   80 │
└────┴──────────────┴──────┘

3 rows in set. Elapsed: 0.004 sec.
```

ASOF

```
hdp-1 :) select a.id,a.name,b.rate,a.time,b.time from join_tb1 as a any inner join join_tb2
as b on a.id=b.id;

SELECT
    a.id,
    a.name,
    b.rate,
    a.time,
    b.time
FROM join_tb1 AS a
ANY INNER JOIN join_tb2 AS b ON a.id = b.id

┌─id─┬─name─────────┬─rate─┬────────────time─┬──────────b.time─┐
│  1 │ ClickHouse   │  100 │ 2020-02-01 12:00:00 │ 2020-05-02 10:00:00 │
│  2 │ Spark        │   90 │ 2020-02-01 12:30:00 │ 2020-05-02 10:01:00 │
│  3 │ ElasticSearch │   80 │ 2020-02-01 13:00:00 │ 2020-05-02 10:02:00 │
└────┴──────────────┴──────┴─────────────────────┴─────────────────────┘

3 rows in set. Elapsed: 0.003 sec.

hdp-1 :) select a.id,a.name,b.rate,a.time,b.time from join_tb1 as a asof inner join join_tb2
as b on a.id=b.id and a.time <= b.time;

SELECT
    a.id,
    a.name,
    b.rate,
    a.time,
    b.time
FROM join_tb1 AS a
```

```
ASOF INNER JOIN join_tb2 AS b ON (a.id = b.id) AND (a.time <= b.time)

┌─id─┬─name──────────┬─rate─┬────────────time─┬──────────────b.time─┐
│  1 │ ClickHouse    │  100 │ 2020-02-01 12:00:00 │ 2020-05-02 10:00:00 │
│  2 │ Spark         │   90 │ 2020-02-01 12:30:00 │ 2020-05-02 10:01:00 │
│  3 │ ElasticSearch │   80 │ 2020-02-01 13:00:00 │ 2020-05-02 10:02:00 │
└────┴───────────────┴──────┴─────────────────────┴─────────────────────┘

3 rows in set. Elapsed: 0.003 sec.
```

Join通过使用一个或多个表的公共值合并来自一个或多个表的列来生成新表。 它是支持SQL的数据库中的常见操作，它对应于 关系代数 加入。 一个表连接的特殊情况通常被称为 “self-join”.

语法:

```
SELECT <expr_list>
FROM <left_table>
[GLOBAL] [INNER|LEFT|RIGHT|FULL|CROSS] [OUTER|SEMI|ANTI|ANY|ASOF] JOIN <right_table>
(ON <expr_list>)|(USING <column_list>) ...
```

从表达式 ON 从子句和列 USING 子句被称为 “join keys”. 除非另有说明，加入产生一个 笛卡尔积 从具有匹配的行 “join keys”，这可能会产生比源表更多的行的结果。

支持的联接类型

所有标准 SQL JOIN 支持类型:

- INNER JOIN，只返回匹配的行。
- LEFT OUTER JOIN，除了匹配的行之外，还返回左表中的非匹配行。
- RIGHT OUTER JOIN，除了匹配的行之外，还返回右表中的非匹配行。
- FULL OUTER JOIN，除了匹配的行之外，还会返回两个表中的非匹配行。
- CROSS JOIN，产生整个表的笛卡尔积, “join keys” 是 不 指定。

JOIN 没有指定类型暗指 INNER. 关键字 OUTER 可以安全地省略。 替代语法 CROSS JOIN 在指定多个表 FROM 用逗号分隔。

ClickHouse中提供的其他联接类型:

- LEFT SEMI JOIN 和 RIGHT SEMI JOIN,白名单 “join keys”，而不产生笛卡尔积。
- LEFT ANTI JOIN 和 RIGHT ANTI JOIN，黑名单 “join keys”，而不产生笛卡尔积。
- LEFT ANY JOIN, RIGHT ANY JOIN and INNER ANY JOIN, partially (for opposite side of LEFT and RIGHT) or completely (for INNER and FULL) disables the cartesian product for standard JOIN types.
- ASOF JOIN and LEFT ASOF JOIN, joining sequences with a non-exact match. ASOF JOIN usage is described below.

严格

注

可以使用以下方式复盖默认的严格性值 join_default_strictness 设置。

Also the behavior of ClickHouse server for ANY JOIN operations depends on the any_join_distinct_right_table_keys setting.

ASOF加入使用

ASOF JOIN 当您需要连接没有完全匹配的记录时非常有用。

算法需要表中的特殊列。 本专栏:

- 必须包含有序序列。
- 可以是以下类型之一: Int，UInt, *浮动, 日期, 日期时间, 十进制.*
- 不能是唯一的列 JOIN

语法 ASOF JOIN ... ON:

```
SELECT expressions_list
FROM table_1
ASOF LEFT JOIN table_2
ON equi_cond AND closest_match_cond
```

您可以使用任意数量的相等条件和恰好一个最接近的匹配条件。 例如, SELECT count() FROM table_1 ASOF LEFT JOIN table_2 ON table_1.a == table_2.b AND table_2.t <= table_1.t.

支持最接近匹配的条件: >, >=, <, <=.

语法 ASOF JOIN ... USING:

```
SELECT expressions_list
FROM table_1
ASOF JOIN table_2
USING (equi_column1, ... equi_columnN, asof_column)
```

ASOF JOIN 用途 equi_columnX 对于加入平等和 asof_column 用于加入与最接近的比赛 table_1.asof_column >= table_2.asof_column 条件。 该 asof_column 列总是在最后一个 USING 条款

例如，请考虑下表:

```
     table_1                           table_2
  event   | ev_time | user_id       event   | ev_time | user_id
----------|---------|----------   ----------|---------|----------
              ...                               ...
event_1_1 |  12:00  |  42         event_2_1 |  11:59  |   42
              ...                 event_2_2 |  12:30  |   42
event_1_2 |  13:00  |  42         event_2_3 |  13:00  |   42
              ...                               ...
```

ASOF JOIN 可以从用户事件的时间戳 table_1 并找到一个事件 table_2 其中时间戳最接近事件的时间戳 table_1 对应于最接近的匹配条件。 如果可用，则相等的时间戳值是最接近的值。 在这里，该 user_id 列可用于连接相等和 ev_time 列可用于在最接近的匹配加入。 在我们的例子中, event_1_1 可以加入 event_2_1 和 event_1_2 可以加入 event_2_3，但是 event_2_2 不能加入。

注

ASOF 加入是 不 支持在 加入我们 表引擎。

分布式联接

有两种方法可以执行涉及分布式表的join:

- 当使用正常 JOIN，将查询发送到远程服务器。 为了创建正确的表，在每个子查询上运行子查询，并使用此表执行联接。 换句话说，在每个服务器上单独形成右表。
- 使用时 GLOBAL ... JOIN，首先请求者服务器运行一个子查询来计算正确的表。 此临时表将传递到每个远程服务器，并使用传输的临时数据对其运行查询。

使用时要小心 GLOBAL. 有关详细信息，请参阅 分布式子查询 科。

使用建议

处理空单元格或空单元格

在连接表时，可能会出现空单元格。 设置 join_use_nulls 定义ClickHouse如何填充这些单元格。

如果 JOIN 键是 可为空 字段，其中至少有一个键具有值的行 NULL 没有加入。

语法

在指定的列 USING 两个子查询中必须具有相同的名称，并且其他列必须以不同的方式命名。 您可以使用别名更改子查询中的列名。

该 USING 子句指定一个或多个要联接的列，这将建立这些列的相等性。 列的列表设置不带括号。 不支持更复杂的连接条件。

语法限制

对于多个 JOIN 单个子句 SELECT 查询:

- 通过以所有列 * 仅在联接表时才可用，而不是子查询。
- 该 PREWHERE 条款不可用。

为 ON, WHERE，和 GROUP BY 条款:

- 任意表达式不能用于 ON, WHERE，和 GROUP BY 子句，但你可以定义一个表达式 SELECT 子句，然后通过别名在这些子句中使用它。

性能

当运行 JOIN，与查询的其他阶段相关的执行顺序没有优化。 连接（在右表中搜索）在过滤之前运行 WHERE 和聚集之前。

每次使用相同的查询运行 JOIN，子查询再次运行，因为结果未缓存。 为了避免这种情况，使用特殊的 加入我们 表引擎，它是一个用于连接的准备好的数组，总是在RAM中。

在某些情况下，使用效率更高 IN 而不是 JOIN.

如果你需要一个 JOIN 对于连接维度表（这些是包含维度属性的相对较小的表，例如广告活动的名称）， JOIN 由于每个查询都会重新访问正确的表，因此可能不太方便。 对于这种情况下，有一个 “external dictionaries” 您应该使用的功能 JOIN. 有关详细信息，请参阅 外部字典 科。

内存限制

默认情况下，ClickHouse使用 哈希联接 算法。 ClickHouse采取 `` 并在RAM中为其创建哈希表。 在某个内存消耗阈值之后，ClickHouse回退到合并联接算法。

如果需要限制联接操作内存消耗，请使用以下设置:

- max_rows_in_join — Limits number of rows in the hash table.
- max_bytes_in_join — Limits size of the hash table.

当任何这些限制达到，ClickHouse作为 join_overflow_mode 设置指示。

例子

示例:

```
SELECT
    CounterID,
    hits,
    visits
FROM
(
    SELECT
        CounterID,
        count() AS hits
    FROM test.hits
    GROUP BY CounterID
) ANY LEFT JOIN
(
    SELECT
        CounterID,
        sum(Sign) AS visits
    FROM test.visits
    GROUP BY CounterID
) USING CounterID
ORDER BY hits DESC
LIMIT 10
┌─CounterID─┬───hits─┬─visits─┐
│   1143050 | 523264 |  13665 |
│    731962 | 475698 | 102716 |
│    722545 | 337212 | 108187 |
│    722889 | 252197 |  10547 |
│   2237260 | 196036 |   9522 |
│  23057320 | 147211 |   7689 |
│    722818 |  90109 |  17847 |
│     48221 |  85379 |   4652 |
│  19762435 |  77807 |   7026 |
│    722884 |  77492 |  11056 |
└───────────┴────────┴────────┘
```

拆分分析:

```
hdp-1 :) select CounterID as id, count() as hits from hits_v1 group by id limit 10;

SELECT
    CounterID AS id,
    count() AS hits
FROM hits_v1
GROUP BY id
LIMIT 10
```

```
┌────────id─┬─hits─┐
| 24141121 |    4 |
| 31790479 |    2 |
| 17424666 |    1 |
| 29229146 |    1 |
|  8325557 |    2 |
|  2314381 |   26 |
|   899083 |    2 |
| 19151207 |   96 |
| 31138002 |    7 |
|   129995 |    4 |
└──────────┴──────┘

10 rows in set. Elapsed: 0.042 sec. Processed 8.87 million rows, 35.50 MB (209.50 million rows/s., 837.99 MB/s.)

hdp-1 :) select CounterID as id,sum(Sign) as visits from visits_v1 group by id limit 10;

SELECT
    CounterID AS id,
    sum(Sign) AS visits
FROM visits_v1
GROUP BY id
LIMIT 10

┌────────id─┬─visits─┐
|  1602763 |     10 |
|   205593 |      1 |
|   406018 |      1 |
|  1398990 |      1 |
| 17891807 |      1 |
| 24614464 |      1 |
| 14339903 |      3 |
|  9991977 |      5 |
| 31934585 |      1 |
| 15525311 |      3 |
└──────────┴────────┘

--LEFT JOIN 各种join

hdp-1 :) select * from (select CounterID as id, count() as hits from hits_v1 group by id limit 10) as a left join (select CounterID as id,sum(Sign) as visits from visits_v1 group by id limit 10) as b on a.id = b.id;

SELECT *
FROM
(
    SELECT
        CounterID AS id,
        count() AS hits
    FROM hits_v1
    GROUP BY id
```

```
    LIMIT 10
) AS a
LEFT JOIN
(
    SELECT
        CounterID AS id,
        sum(Sign) AS visits
    FROM visits_v1
    GROUP BY id
    LIMIT 10
) AS b ON a.id = b.id

┌────────id─┬─hits─┬─b.id─┬─visits─┐
| 24141121 |    4 |    0 |      0 |
| 31790479 |    2 |    0 |      0 |
| 17424666 |    1 |    0 |      0 |
| 29229146 |    1 |    0 |      0 |
|  8325557 |    2 |    0 |      0 |
|  2314381 |   26 |    0 |      0 |
|   899083 |    2 |    0 |      0 |
| 19151207 |   96 |    0 |      0 |
| 31138002 |    7 |    0 |      0 |
|   129995 |    4 |    0 |      0 |
└──────────┴──────┴──────┴────────┘
```

2.5 PREWHERE 子句

```
select WatchID,Title,GoodEvent from hits_v1 where JavaEnable=1;
6535088 rows in set. Elapsed: 40.853 sec. Processed 8.87 million rows, 863.90 MB (217.22
thousand rows/s., 21.15 MB/s.)

select WatchID,Title,GoodEvent from hits_v1 prewhere JavaEnable=1;
6535088 rows in set. Elapsed: 22.150 sec. Processed 8.87 million rows, 863.90 MB (400.62
thousand rows/s., 39.00 MB/s.)


SET optimize_move_to_prewhere = 0

Ok.

0 rows in set. Elapsed: 0.011 sec.

hdp-1 :) select WatchID,Title,GoodEvent from hits_v1 where JavaEnable=1;
6535088 rows in set. Elapsed: 35.728 sec. Processed 8.87 million rows, 864.55 MB (248.38
thousand rows/s., 24.20 MB/s.)
```

```
hdp-1 :) select WatchID,Title,GoodEvent from hits_v1 prewhere JavaEnable=1;
6535088 rows in set. Elapsed: 32.021 sec. Processed 8.87 million rows, 863.90 MB (277.13
thousand rows/s., 26.98 MB/s.)
```

Prewhere是更有效地进行过滤的优化。 默认情况下，即使在 PREWHERE 子句未显式指定。 它也会自动移动 WHERE 条件到prewhere阶段。 PREWHERE 子句只是控制这个优化，如果你认为你知道如何做得比默认情况下更好才去控制它。

使用prewhere优化，首先只读取执行prewhere表达式所需的列。 然后读取运行其余查询所需的其他列，但只读取prewhere表达式所在的那些块 “true” 至少对于一些行。 如果有很多块，其中prewhere表达式是 “false” 对于所有行和prewhere需要比查询的其他部分更少的列，这通常允许从磁盘读取更少的数据以执行查询。

手动控制Prewhere

该子句具有与 WHERE 相同的含义，区别在于从表中读取数据。 当手动控制 PREWHERE 对于查询中的少数列使用的过滤条件，但这些过滤条件提供了强大的数据过滤。 这减少了要读取的数据量。

查询可以同时指定 PREWHERE 和 WHERE. 在这种情况下, PREWHERE 先于 WHERE.

如果 optimize_move_to_prewhere 设置为0，启发式自动移动部分表达式 WHERE 到 PREWHERE 被禁用。

限制

PREWHERE 只有支持 *MergeTree 族系列引擎的表。

```
vi /var/log/clickhouse-server/clickhouse-server.log

2020.10.24 13:36:24.818179 [ 13472 ] {6f7289cb-4332-4cfd-a1ac-aa98025bf56f} <Debug>
InterpreterSelectQuery: MergeTreeWhereOptimizer: condition "JavaEnable = 1" moved to
PREWHERE
```

2.6 WHERE

WHERE 子句允许过滤从 FROM 子句 SELECT.

如果有一个 WHERE 子句，它必须包含一个表达式与 UInt8 类型。 这通常是一个带有比较和逻辑运算符的表达式。此表达式计算结果为0的行将从进一步的转换或结果中解释出来。

WHERE 如果基础表引擎支持，则根据使用索引和分区修剪的能力评估表达式。

注

有一个叫做过滤优化 prewhere 的东西.

2.7 GROUP BY子句

GROUP BY 子句将 SELECT 查询结果转换为聚合模式，其工作原理如下:

- GROUP BY 子句包含表达式列表（或单个表达式 -- 可以认为是长度为1的列表）。 这份名单充当 “grouping key”，而每个单独的表达式将被称为 “key expressions”.

- 在所有的表达式在 SELECT, HAVING，和 ORDER BY 子句中 必须 基于键表达式进行计算 或 上 聚合函数 在非键表达式（包括纯列）上。 换句话说，从表中选择的每个列必须用于键表达式或聚合函数内，但不能同时使用。
- 聚合结果 SELECT 查询将包含尽可能多的行，因为有唯一值 “grouping key” 在源表中。 通常这会显着减少行数，通常是数量级，但不一定：如果所有行数保持不变 “grouping key” 值是不同的。

注

还有一种额外的方法可以在表上运行聚合。 如果查询仅在聚合函数中包含表列，则 GROUP BY 可以省略，并且通过一个空的键集合来假定聚合。 这样的查询总是只返回一行。

空处理

对于分组，ClickHouse解释 NULL 作为一个值，并且 NULL==NULL. 它不同于 NULL 在大多数其他上下文中的处理方式。

这里有一个例子来说明这意味着什么。

假设你有一张表:

```
┌─x─┬────y─┐
│ 1 │    2 │
│ 2 │ NULL │
│ 3 │    2 │
│ 3 │    3 │
│ 3 │ NULL │
└───┴──────┘
```

查询 SELECT sum(x), y FROM t_null_big GROUP BY y 结果:

```
┌─sum(x)─┬────y─┐
│      4 │    2 │
│      3 │    3 │
│      5 │ NULL │
└────────┴──────┘
```

你可以看到 GROUP BY 为 y = NULL 总结 x，仿佛 NULL 是这个值。

如果你通过几个键 GROUP BY，结果会给你选择的所有组合，就好像 NULL 是一个特定的值。

WITH TOTAL 修饰符

如果 WITH TOTALS 被指定，将计算另一行。 此行将具有包含默认值（零或空行）的关键列，以及包含跨所有行计算值的聚合函数列（ “total” 值）。

这个额外的行仅产生于 JSON*, *TabSeparated*，和 Pretty* 格式，与其他行分开:

- 在 JSON* 格式，这一行是作为一个单独的输出 ‘totals’ 字段。
- 在 TabSeparated* 格式，该行位于主结果之后，前面有一个空行（在其他数据之后）。
- 在 Pretty* 格式时，该行在主结果之后作为单独的表输出。
- 在其他格式中，它不可用。

WITH TOTALS 可以以不同的方式运行时 HAVING 是存在的。 该行为取决于 totals_mode 设置。

配置总和处理

默认情况下, totals_mode = 'before_having'. 在这种情况下, 'totals' 是跨所有行计算，包括那些不通过具有和 max_rows_to_group_by.

其他替代方案仅包括通过具有在 'totals'，并与设置不同的行为 max_rows_to_group_by 和 group_by_overflow_mode = 'any'.

after_having_exclusive – Don't include rows that didn't pass through max_rows_to_group_by. 换句话说, 'totals' 将有少于或相同数量的行，因为它会 max_rows_to_group_by 被省略。

after_having_inclusive – Include all the rows that didn't pass through 'max_rows_to_group_by' 在 'totals'. 换句话说, 'totals' 将有多个或相同数量的行，因为它会 max_rows_to_group_by 被省略。

after_having_auto – Count the number of rows that passed through HAVING. If it is more than a certain amount (by default, 50%), include all the rows that didn't pass through 'max_rows_to_group_by' 在 'totals'. 否则，不包括它们。

totals_auto_threshold – By default, 0.5. The coefficient for after_having_auto.

如果 max_rows_to_group_by 和 group_by_overflow_mode = 'any' 不使用，所有的变化 after_having 是相同的，你可以使用它们中的任何一个（例如, after_having_auto).

您可以使用 WITH TOTALS 在子查询中，包括在子查询 JOIN 子句（在这种情况下，将各自的总值合并）。

例子

示例:

```
SELECT
    count(),
    median(FetchTiming > 60 ? 60 : FetchTiming),
    count() - sum(Refresh)
FROM hits
```

但是，与标准SQL相比，如果表没有任何行（根本没有任何行，或者使用 WHERE 过滤之后没有任何行），则返回一个空结果，而不是来自包含聚合函数初始值的行。

相对于MySQL（并且符合标准SQL），您无法获取不在键或聚合函数（常量表达式除外）中的某些列的某些值。 要解决此问题，您可以使用 'any' 聚合函数（获取第一个遇到的值）或 'min/max'.

示例:

```
SELECT
    domainWithoutWWW(URL) AS domain,
    count(),
    any(Title) AS title -- getting the first occurred page header for each domain.
FROM hits
GROUP BY domain
```

对于遇到的每个不同的键值, GROUP BY 计算一组聚合函数值。

GROUP BY 不支持数组列。

不能将常量指定为聚合函数的参数。 示例: sum(1). 相反，你可以摆脱常数。 示例: count().

实现细节

聚合是面向列的 DBMS 最重要的功能之一，因此它的实现是ClickHouse中最优化的部分之一。 默认情况下，聚合使用哈希表在内存中完成。 它有 40+ 的特殊化自动选择取决于 “grouping key” 数据类型。

在外部存储器中分组

您可以启用将临时数据转储到磁盘以限制内存使用期间 GROUP BY. 该 max_bytes_before_external_group_by 设置确定倾销的阈值RAM消耗 GROUP BY 临时数据到文件系统。 如果设置为0（默认值），它将被禁用。

使用时 max_bytes_before_external_group_by，我们建议您设置 max_memory_usage 大约两倍高。 这是必要的，因为聚合有两个阶段：读取数据和形成中间数据（1）和合并中间数据（2）。 将数据转储到文件系统只能在阶段1中发生。 如果未转储临时数据，则阶段2可能需要与阶段1相同的内存量。

例如，如果 max_memory_usage 设置为10000000000，你想使用外部聚合，这是有意义的设置 max_bytes_before_external_group_by 到10000000000，和 max_memory_usage 到20000000000。 当触发外部聚合（如果至少有一个临时数据转储）时，RAM的最大消耗仅略高于 max_bytes_before_external_group_by.

通过分布式查询处理，在远程服务器上执行外部聚合。 为了使请求者服务器只使用少量的RAM，设置 distributed_aggregation_memory_efficient 到1。

当合并数据刷新到磁盘时，以及当合并来自远程服务器的结果时， distributed_aggregation_memory_efficient 设置被启用，消耗高达 1/256 * the_number_of_threads 从RAM的总量。

当启用外部聚合时，如果数据量小于 max_bytes_before_external_group_by (例如数据没有被 flushed), 查询执行速度和不在外部聚合的速度一样快. 如果临时数据被flushed到外部存储, 执行的速度会慢几倍 (大概是三倍).

如果你有一个 ORDER BY 用一个 LIMIT 后 GROUP BY，然后使用的RAM的量取决于数据的量 LIMIT，不是在整个表。 但如果 ORDER BY 没有 LIMIT，不要忘记启用外部排序 (max_bytes_before_external_sort).

2.8 LIMIT BY子句

与查询 LIMIT n BY expressions 子句选择第一个 n 每个不同值的行 expressions. LIMIT BY 可以包含任意数量的 表达式.

ClickHouse支持以下语法变体:

- LIMIT [offset_value,]n BY expressions
- LIMIT n OFFSET offset_value BY expressions

在查询处理过程中，ClickHouse会选择按排序键排序的数据。 排序键使用以下命令显式设置 ORDER BY 子句或隐式作为表引擎的属性。 然后ClickHouse应用 LIMIT n BY expressions 并返回第一 n 每个不同组合的行 expressions. 如果 OFFSET 被指定，则对于每个数据块属于一个不同的组合 expressions，ClickHouse跳过 offset_value 从块开始的行数，并返回最大值 n 行的结果。 如果 offset_value 如果数据块中的行数大于数据块中的行数，ClickHouse将从该块返回零行。

注

LIMIT BY 是不相关的 LIMIT. 它们都可以在同一个查询中使用。

例

样例表:

```
CREATE TABLE limit_by(id Int, val Int) ENGINE = Memory;
INSERT INTO limit_by VALUES (1, 10), (1, 11), (1, 12), (2, 20), (2, 21);
```

查询:

```
SELECT * FROM limit_by ORDER BY id, val LIMIT 2 BY id
┌─id─┬─val─┐
│  1 │  10 │
│  1 │  11 │
│  2 │  20 │
│  2 │  21 │
└────┴─────┘
SELECT * FROM limit_by ORDER BY id, val LIMIT 1, 2 BY id
┌─id─┬─val─┐
│  1 │  11 │
│  1 │  12 │
│  2 │  21 │
└────┴─────┘
```

该 SELECT * FROM limit_by ORDER BY id, val LIMIT 2 OFFSET 1 BY id 查询返回相同的结果。

以下查询返回每个引用的前5个引用 domain, device_type 最多可与100行配对 (LIMIT n BY + LIMIT).

```
SELECT
    domainWithoutWWW(URL) AS domain,
    domainWithoutWWW(REFERRER_URL) AS referrer,
    device_type,
    count() cnt
FROM hits
GROUP BY domain, referrer, device_type
ORDER BY cnt DESC
LIMIT 5 BY domain, device_type
LIMIT 100
```

2.9 HAVING 子句

允许过滤由 GROUP BY 生成的聚合结果. 它类似于 WHERE ， 但不同的是 WHERE 在聚合之前执行，而 HAVING 之后进行。

可以从 SELECT 生成的聚合结果中通过他们的别名来执行 HAVING 子句。 或者 HAVING 子句可以筛选查询结果中未返回的其他聚合的结果。

限制

HAVING 如果不执行聚合则无法使用。 使用 WHERE 则相反。

2.10 SELECT 子句

表达式 指定 SELECT 子句是在上述子句中的所有操作完成后计算的。 这些表达式的工作方式就好像它们应用于结果中的单独行一样。 如果表达式 SELECT 子句包含聚合函数，然后ClickHouse将使用 GROUP BY 聚合参数应用在聚合函数和表达式上。

如果在结果中包含所有列，请使用星号 (*）符号。 例如, SELECT * FROM ....

将结果中的某些列与 re2 正则表达式匹配，可以使用 COLUMNS 表达。

```
COLUMNS('regexp')
```

例如表:

```
CREATE TABLE default.col_names (aa Int8, ab Int8, bc Int8) ENGINE = TinyLog
```

以下查询所有列名包含 a 。

```
SELECT COLUMNS('a') FROM col_names
┌─aa─┬─ab─┐
│  1 │  1 │
└────┴────┘
```

所选列不按字母顺序返回。

您可以使用多个 COLUMNS 表达式并将函数应用于它们。

例如:

```
SELECT COLUMNS('a'), COLUMNS('c'), toTypeName(COLUMNS('c')) FROM col_names
┌─aa─┬─ab─┬─bc─┬─toTypeName(bc)─┐
│  1 │  1 │  1 │ Int8           │
└────┴────┴────┴────────────────┘
```

返回的每一列 COLUMNS 表达式作为单独的参数传递给函数。 如果函数支持其他参数，您也可以将其他参数传递给函数。 使用函数时要小心，如果函数不支持传递给它的参数，ClickHouse将抛出异常。

例如:

```
SELECT COLUMNS('a') + COLUMNS('c') FROM col_names
Received exception from server (version 19.14.1):
Code: 42. DB::Exception: Received from localhost:9000. DB::Exception: Number of arguments
for function plus doesn't match: passed 3, should be 2.
```

该例子中, COLUMNS('a') 返回两列: aa 和 ab. COLUMNS('c') 返回 bc 列。 该 + 运算符不能应用于3个参数，因此 ClickHouse抛出一个带有相关消息的异常。

匹配的列 COLUMNS 表达式可以具有不同的数据类型。 如果 COLUMNS 不匹配任何列，并且是在 SELECT 唯一的表达式，ClickHouse则抛出异常。

星号

您可以在查询的任何部分使用星号替代表达式。进行查询分析、时，星号将展开为所有表的列（不包括 MATERIALIZED 和 ALIAS 列）。 只有少数情况下使用星号是合理的:

- 创建转储表时。
- 对于只包含几列的表，例如系统表。
- 获取表中列的信息。 在这种情况下，设置 LIMIT 1. 但最好使用 DESC TABLE 查询。
- 当对少量列使用 PREWHERE 进行强过滤时。
- 在子查询中（因为外部查询不需要的列从子查询中排除）。

在所有其他情况下，我们不建议使用星号，因为它只给你一个列DBMS的缺点，而不是优点。 换句话说，不建议使用星号。

极端值

除结果之外，还可以获取结果列的最小值和最大值。 要做到这一点，设置 extremes 设置为1。 最小值和最大值是针对数字类型、日期和带有时间的日期计算的。 对于其他类型列，输出默认值。

分别的额外计算两行 – 最小值和最大值。 这额外的两行采用输出格式为 JSON*, TabSeparated*，和 Pretty* formats，与其他行分开。 它们不以其他格式输出。

为 JSON* 格式时，极端值单独的输出在 'extremes' 字段。 为 TabSeparated* 格式时，此行来的主要结果集后，然后显示 'totals' 字段。 它前面有一个空行（在其他数据之后）。 在 Pretty* 格式时，该行在主结果之后输出为一个单独的表，然后显示 'totals' 字段。

极端值在 LIMIT 之前被计算，但在 LIMIT BY 之后被计算. 然而，使用 LIMIT offset, size， offset 之前的行都包含在 extremes. 在流请求中，结果还可能包括少量通过 LIMIT 过滤的行.

备注

您可以在查询的任何部分使用同义词 (AS 别名）。

GROUP BY 和 ORDER BY 子句不支持位置参数。 这与MySQL相矛盾，但符合标准SQL。 例如, GROUP BY 1, 2 将被理解为根据常量分组 (i.e. aggregation of all rows into one).

实现细节

如果查询省略 DISTINCT, GROUP BY， ORDER BY， IN， JOIN 子查询，查询将被完全流处理，使用O(1)量的 RAM。 若未指定适当的限制，则查询可能会消耗大量RAM:

- max_memory_usage
- max_rows_to_group_by
- max_rows_to_sort
- max_rows_in_distinct
- max_bytes_in_distinct
- max_rows_in_set
- max_bytes_in_set
- max_rows_in_join
- max_bytes_in_join
- max_bytes_before_external_sort
- max_bytes_before_external_group_by

有关详细信息，请参阅部分 "Settings". 可以使用外部排序（将临时表保存到磁盘）和外部聚合。

2.11 DISTINCT子句

如果 SELECT DISTINCT 被声明，则查询结果中只保留唯一行。 因此，在结果中所有完全匹配的行集合中，只有一行被保留。

空处理

DISTINCT 适用于 NULL 就好像 NULL 是一个特定的值，并且 NULL==NULL. 换句话说，在 DISTINCT 结果，不同的组合 NULL 仅发生一次。 它不同于 NULL 在大多数其他情况中的处理方式。

替代办法

通过应用可以获得相同的结果 GROUP BY 在同一组值指定为 SELECT 子句，并且不使用任何聚合函数。 但与 GROUP BY 有几个不同的地方:

- DISTINCT 可以与 GROUP BY 一起使用.
- 当 ORDER BY 被省略并且 LIMIT 被定义时，在读取所需数量的不同行后立即停止运行。
- 数据块在处理时输出，而无需等待整个查询完成运行。

限制

DISTINCT 不支持当 SELECT 包含有数组的列。

例子

ClickHouse支持使用 DISTINCT 和 ORDER BY 在一个查询中的不同的列。 DISTINCT 子句在 ORDER BY 子句前被执行。

示例表:

```
┌─a─┬─b─┐
│ 2 │ 1 │
│ 1 │ 2 │
│ 3 │ 3 │
│ 2 │ 4 │
└───┴───┘
```

当执行 SELECT DISTINCT a FROM t1 ORDER BY b ASC 来查询数据，我们得到以下结果:

```
┌─a─┐
│ 2 │
│ 1 │
│ 3 │
└───┘
```

如果我们改变排序方向 SELECT DISTINCT a FROM t1 ORDER BY b DESC，我们得到以下结果:

```
┌─a─┐
│ 3 │
│ 1 │
│ 2 │
└───┘
```

行 2, 4 排序前被切割。

在编程查询时考虑这种实现特性。

2.12 LIMIT

LIMIT m 允许选择结果中起始的 m 行。

LIMIT n, m 允许选择个 m 从跳过第一个结果后的行 n 行。 与 LIMIT m OFFSET n 语法是等效的。

n 和 m 必须是非负整数。

如果没有 ORDER BY 子句显式排序结果，结果的行选择可能是任意的和非确定性的。

LIMIT ... WITH TIES 修饰符

如果为 LIMIT n[,m] 设置了 WITH TIES， 并且声明了 ORDER BY expr_list, you will get in result first n or n,m rows and all rows with same ORDER BY fields values equal to row at position n for LIMIT n and m for LIMIT n,m.

此修饰符可以与： ORDER BY ... WITH FILL modifier 组合使用.

例如以下查询：

```
SELECT * FROM (
    SELECT number%50 AS n FROM numbers(100)
) ORDER BY n LIMIT 0,5
```

返回

```
┌─n─┐
| 0 |
| 0 |
| 1 |
| 1 |
| 2 |
└───┘
```

单子执行了 WITH TIES 修饰符后

```
SELECT * FROM (
    SELECT number%50 AS n FROM numbers(100)
) ORDER BY n LIMIT 0,5 WITH TIES
```

则返回了以下的数据行

```
┌─n─┐
| 0 |
| 0 |
| 1 |
| 1 |
| 2 |
| 2 |
└───┘
```

cause row number 6 have same value “2” for field n as row number 5

2.13 UNION ALL子句

你可以使用 UNION ALL 结合任意数量的 SELECT 来扩展其结果。 示例:

```
SELECT CounterID, 1 AS table, toInt64(count()) AS c
    FROM test.hits
    GROUP BY CounterID

UNION ALL

SELECT CounterID, 2 AS table, sum(Sign) AS c
    FROM test.visits
    GROUP BY CounterID
    HAVING c > 0
```

结果列通过它们的索引进行匹配（在内部的顺序 SELECT). 如果列名称不匹配，则从第一个查询中获取最终结果的名称。

对联合执行类型转换。 例如，如果合并的两个查询具有相同的字段与非-Nullable 和 Nullable 从兼容类型的类型，由此产生的 UNION ALL 有一个 Nullable 类型字段。

属于以下部分的查询 UNION ALL 不能用圆括号括起来。 ORDER BY 和 LIMIT 应用于单独的查询，而不是最终结果。如果您需要将转换应用于最终结果，则可以将所有查询 UNION ALL 在子查询中 FROM 子句。

限制

只有 UNION ALL 支持。 UNION (UNION DISTINCT）不支持。 如果你需要 UNION DISTINCT，你可以写 SELECT DISTINCT 子查询中包含 UNION ALL.

实现细节

属于 UNION ALL 的查询可以同时运行，并且它们的结果可以混合在一起。

2.14 INTO OUTFILE 子句

添加 INTO OUTFILE filename 子句（其中filename是字符串） SELECT query 将其输出重定向到客户端上的指定文件。

实现细节

- 此功能是在可用 命令行客户端 和 clickhouse-local. 因此通过 HTTP接口 发送查询将会失败。
- 如果具有相同文件名的文件已经存在，则查询将失败。
- 默认值 输出格式 是 TabSeparated （就像在命令行客户端批处理模式中一样）。

2.15 Format子句

ClickHouse支持广泛的 序列化格式https://clickhouse.tech/docs/zh/interfaces/formats/ 可用于查询结果等。 有多种方法可以选择格式化 SELECT 的输出，其中之一是指定 FORMAT format 在查询结束时以任何特定格式获取结果集。

特定的格式方便使用，与其他系统集成或增强性能。

默认格式

如果 FORMAT 被省略则使用默认格式，这取决于用于访问ClickHouse服务器的设置和接口。 为 HTTP接口 和 命令行客户端 在批处理模式下，默认格式为 TabSeparated. 对于交互模式下的命令行客户端，默认格式为 PrettyCompact（它生成紧凑的人类可读表）。

实现细节

使用命令行客户端时，数据始终以内部高效格式通过网络传递 (Native). 客户端独立解释 FORMAT 查询子句并格式化数据本身（以减轻网络和服务器的额外负担）。