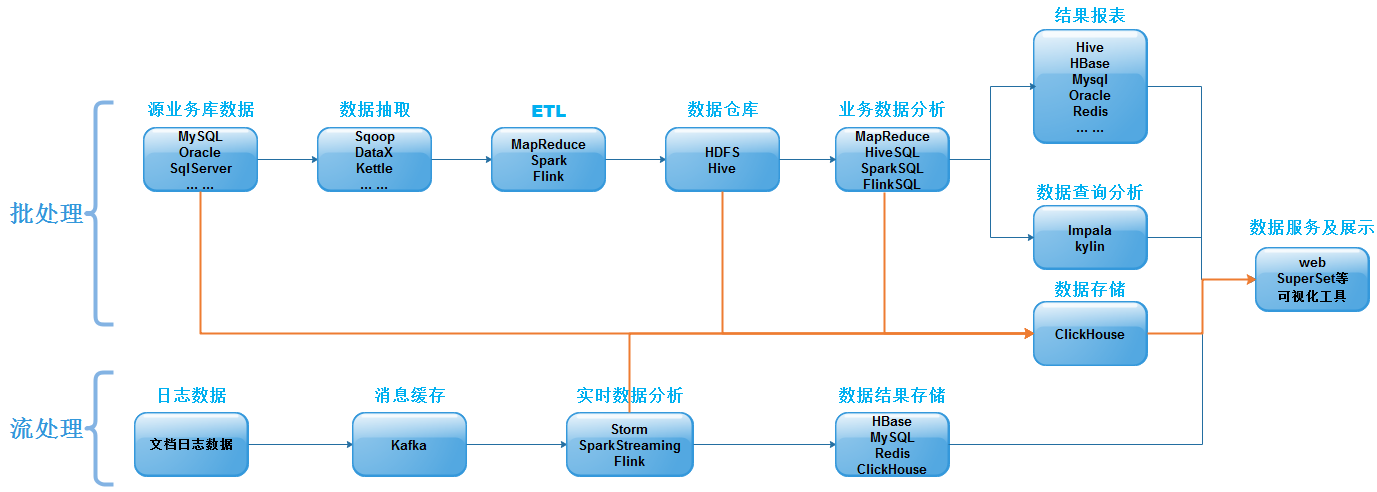
**ClickHouse文档**

## ClickHouse与其特性

在大数据处理场景中，流处理和批处理使用到的技术大致如下：



批处理会将源业务系统中的数据通过数据抽取工具（例如Sqoop）将数据抽取到HDFS中，这个过程可以使用MapReduce、Spark、Flink技术对数据进行ETL清洗处理，也可以直接将数据抽取到Hive数仓中，一般可以将结构化的数据直接抽取到Hive数据仓库中，然后使用HiveSQL或者SparkSQL进行业务指标分析，如果涉及到的分析业务非常复杂，可以使用Hive的自定义函数或者Spark、Flink进行复杂分析，这就是我们通常说的数据指标分析。分析之后的结果可以保存到Hive、HBase、MySQL、Redis等，供后续查询使用。一般在数仓构建中，如果指标存入Hive中，我们可以使用Sqoop工具将结果导入到关系型数据库中供后续查询。HBase中更擅长存储原子性非聚合查询数据，如果有大量结果数据后期不需要聚合查询，也可以通过业务分析处理考虑存入HBase中。对于一些查询需求结果反馈非常快的场景可以考虑将结果存入Redis中。

对于大多数企业构建数仓之后，会将结果存入到Hive中的DM层中。DM层数据存入的是与业务强相关的报表数据，DM层数据是由数仓中DWS层主题宽表聚合统计得到，这种报表层设计适合查询固定的场景。对于一些查询需求多变场景，我们也可以使用impala来直接将主题宽表数据基于内存进行交互式查询，对web或者数据分析做到交互式返回结果，使用impala对内存开销非常大。还有另外一种方式是使用Kylin进行预计算，将结果提前计算好存入Hbase中，以供后续交互式查询结果，Kylin是使用空间获取时间的一种方式，预先将各种维度组合对应的度量计算出来存入HBase,用户写SQL交互式查询的是HBase中预计算好的结果数据。最后将数据分析结果可以直接对web以接口服务提供使用或者公司内部使用可视化工具展示使用。

以上无论批处理过程还是流处理过程，使用到的技术几乎离不开Hadoop生态圈。

### **什么是ClickHouse**

ClickHouse是一个开源的，用于联机分析（OLAP）的列式数据库管理系统（DBMS-database manager system）, 它是面向列的，并允许使用SQL查询，实时生成分析报告。ClickHouse最初是一款名为Yandex.Metrica的产品，主要用于WEB流量分析。ClickHouse的全称是Click Stream，Data WareHouse，简称ClickHouse。

ClickHouse不是一个单一的数据库,它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。ClickHouse同时支持列式存储和数据压缩，这是对于一款高性能数据库来说是必不可少的特性。一个非常流行的观点认为，如果你想让查询变得更快，最简单且有效的方法是**减少数据扫描范围**和**数据传输时的大小**，而列式存储和数据压缩就可以帮助我们实现上述两点，列式存储和数据压缩通常是伴生的，因为一般来说列式存储是数据压缩的前提。

### **OLAP场景的特征**

* 绝大多数是读请求。
* 数据以相当大的批次(> 1000行)更新，而不是单行更新;或者根本没有更新。
* 已添加到数据库的数据不能修改。
* 对于读取，从数据库中提取相当多的行，但只提取列的一小部分。
* 宽表，即每个表包含着大量的列。
* 查询相对较少(通常每台服务器每秒查询数百次或更少)。
* 对于简单查询，允许延迟大约50毫秒。
* 列中的数据相对较小：数字和短字符串(例如，每个URL 60个字节)。
* 处理单个查询时需要高吞吐量(每台服务器每秒可达数十亿行)。
* 事务不是必须的。
* 对数据一致性要求低。有副本情况下，写入一个即可，后台自动同步。
* 每个查询有一个大表。除了他以外，其他的都很小。
* 查询结果明显小于源数据。换句话说，数据经过过滤或聚合，因此结果适合于单个服务器的RAM中。

通过以上OLAP场景分析特点很容易可以看出，OLAP场景与其他通常业务场景(例如,OLTP或K/V)有很大的不同， 因此想要使用OLTP或Key-Value数据库去高效的处理分析查询场景，并不是非常完美的适用方案。例如，使用OLAP数据库去处理分析请求通常要优于使用MongoDB或Redis去处理分析请求。

### **ClickHouse特性**

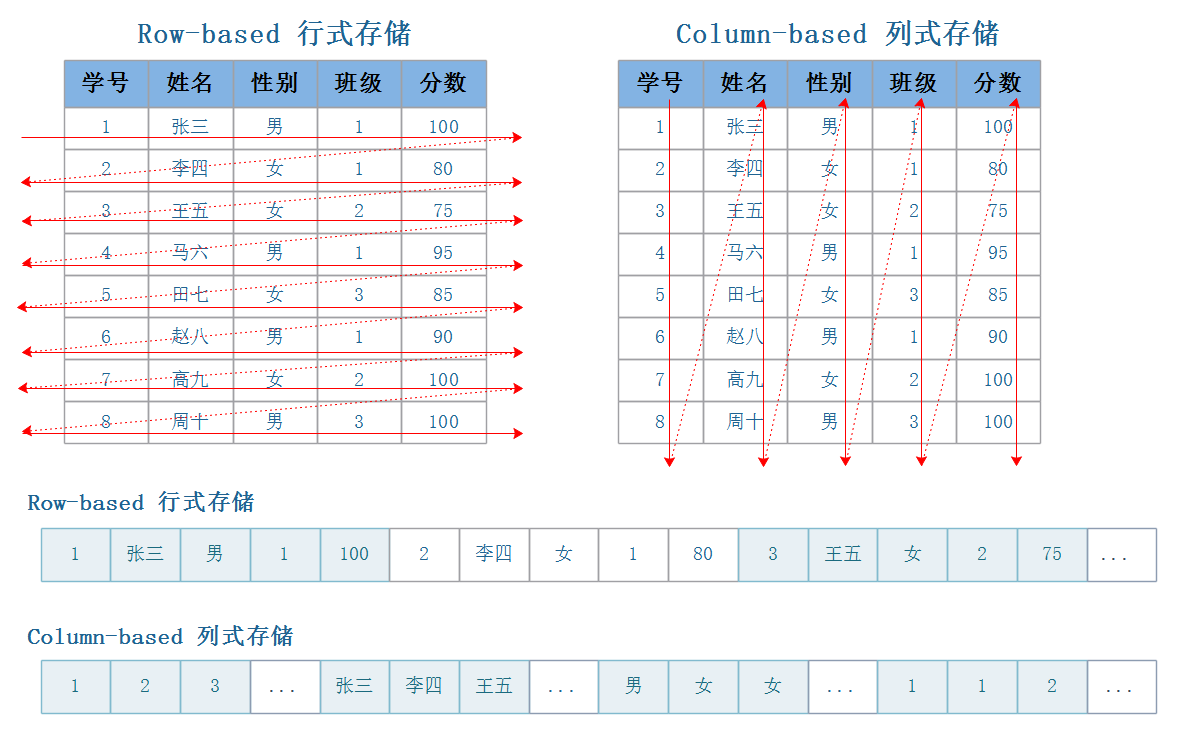
#### 完备的DBMS功能

ClickHouse是一个数据库管理系统，而不仅是一个数据库，作为数据库管理系统具备完备的管理功能：

* DDL(Data Definition Language-数据定义语言)：可以动态地创建、修改或删除数据库、表和视图，而无须重启服务。
* DML(Data Manipulation Language)：可以动态查询、插入、修改或删除数据。
* 分布式管理：提供集群模式，能够自动管理多个数据库节点。
* 权限控制：可以按照用户粒度设置数据库或者表的操作权限，保障数据的安全性。
* 数据备份与恢复：提供了数据备份导出与导入恢复机制，满足生产环境的要求。

#### 列式存储

目前大数据存储有两种方案可以选择，行式存储(Row-Based)和列式存储(Column-Based)。



* **行式存储在数据写入和修改上具有优势。**

行存储的写入是一次完成的，如果这种写入建立在操作系统的文件系统上，可以保证写入过程的成功或者失败，可以保证数据的完整性。列式存储需要把一行记录拆分成单列保存，写入次数明显比行存储多(因为磁头调度次数多，而磁头调度是需要时间的，一般在1ms~10ms)，再加上磁头需要在盘片上移动和定位花费的时间，实际消耗更大。

数据修改实际上也是一次写入过程，不同的是，数据修改是对磁盘上的记录做删除标记。行存储是在指定位置写入一次，列存储是将磁盘定位到多个列上分别写入，这个过程仍是行存储的列数倍。

所以，行式存储在数据写入和修改上具有很大优势。

* **列式存储在数据读取和解析、分析数据上具有优势。**

数据读取时，行存储通常将一行数据完全读出，如果只需要其中几列数据的情况，就会存在冗余列，出于缩短处理时间的考量，消除冗余列的过程通常是在内存中进行的。列存储每次读取的数据是集合的一段或者全部，不存在冗余性问题。

列式存储中的每一列数据类型是相同的，不存在二义性问题，例如，某列类型为整型int,那么它的数据集合一定是整型数据，这种情况使数据解析变得十分容易。相比之下，行存储则要复杂得多，因为在一行记录中保存了多种类型的数据，数据解析需要在多种数据类型之间频繁转换，这个操作很消耗CPU，增加了解析的时间。

所以，列式存储在数据读取和解析数据做数据分析上更具优势。

综上所述，行存储的写入是一次性完成，消耗的时间比列存储少，并且能够保证数据的完整性，缺点是数据读取过程中会产生冗余数据，如果只有少量数据，此影响可以忽略，数量大可能会影响到数据的处理效率。列存储在写入效率、保证数据完整性上都不如行存储，它的优势是在读取过程，不会产生冗余数据，这对数据完整性要求不高的大数据处理领域比较重要。一般来说一个OLAP类型的查询可能需要访问几百万或者几十亿行的数据，但是OLAP分析时只是获取少数的列，对于这种场景列式数据库只需要读取对应的列即可，行式数据库需要读取所有的数据列，因此这种场景更适合列式数据库，可以大大提高OLAP数据分析的效率。ClickHouse就是一款使用列式存储的数据库，数据按列进行组织，属于同一列的数据会被保存在一起，列与列之间也会由不同的文件分别保存，在对OLAP场景分析时，效率很高。

#### 数据压缩

为了使数据在传输上更小，处理起来更快，可以对数据进行压缩，ClickHouse默认使用LZ4算法压缩，数据总体压缩比可达8:1。

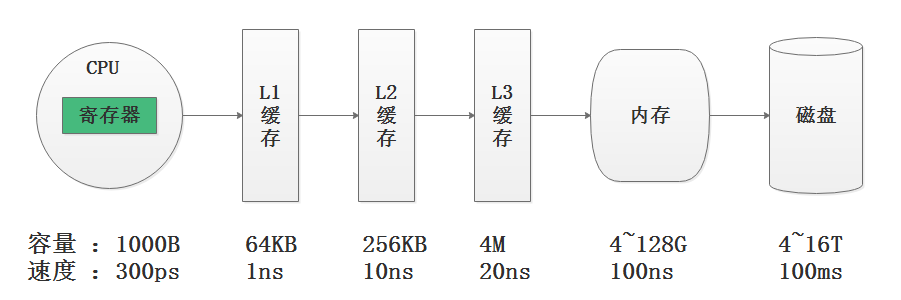
ClickHouse采用列式存储，列式存储相对于行式存储另一个优势就是对数据压缩的友好性。例如：有两个字符串“ABCDE”，“BCD”，现在对它们进行压缩：

|  |
| --- |
| 压缩前：ABCDE\_BCD  压缩后：ABCDE\_(5,3) |

通过以上案例可以看到，压缩的本质是按照一定步长对数据进行匹配扫描，当发现重复部分的时候就进行编码转换。例如：(5,3)代表从下划线往前数5个字节，会匹配上3个字节长度的重复项，即:“BCD”。当然，真实的压缩算法比以上举例更复杂，但压缩的本质就是如此，数据中重复性项越多，则压缩率越高，压缩率越高，则数据体量越小，而数据体量越小，则数据在网络中的传输越快，对网络带宽和磁盘IO的压力也就越小。

列式存储中同一个列的数据由于它们拥有相同的数据类型和现实语义，可能具备重复项的可能性更高，更利于数据的压缩。所以ClickHouse在数据压缩上比例很大。

#### 向量化执行引擎



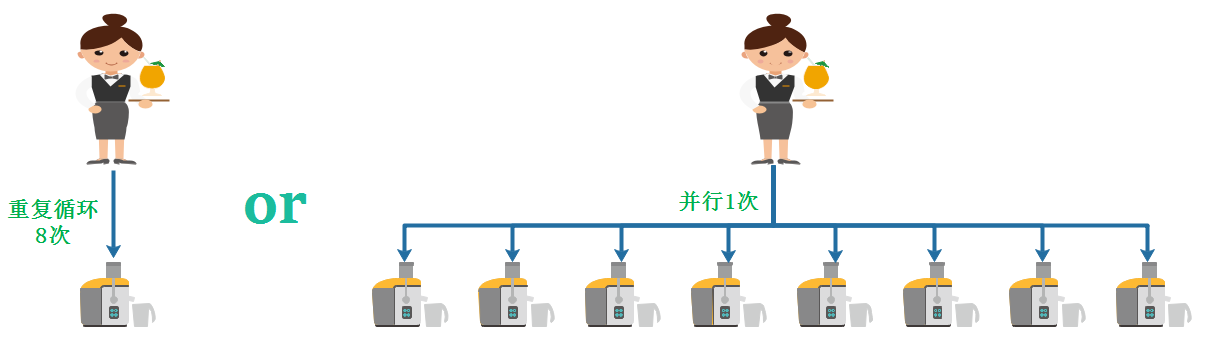
在计算机系统的体系结构中，存储系统是一种层次结构，典型服务器计算机的存储层次结构如上图，上图表述了CPU、CPU三级缓存、内存、磁盘数据容量与数据读取速度对比，我们可以看出存储媒介距离CPU越近，则访问数据的速度越快。

*注意：缓存就是数据交换的缓冲区，缓存往往都是RAM(断电即掉的非永久储存),它们的作用就是帮助硬件更快地响应。CPU缓存的定义为CPU与内存之间的临时数据交换器，它的出现是为了解决CPU运行处理速度与内存读写速度不匹配的矛盾，CPU缓存一般直接跟CPU芯片集成或位于主板总线互连的独立芯片上，现阶段的CPU缓存一般直接集成在CPU上。CPU往往需要重复处理相同的数据、重复执行相同的指令，如果这部分数据、指令，CPU能在CPU缓存中找到，CPU就不需要从内存或硬盘中再读取数据、指令，从而减少了整机的响应时间。*

由上图可知，从内存读取数据速度比磁盘读取数据速度要快1000倍，从CPU缓存中读取数据的速度比从内存中读取数据的速度最快要快100倍，从CPU寄存器中读取数据的速度为300ps(1000ps 皮秒 = 1ns)，比CPU缓存要快3倍还多。从寄存器中访问数据的速度，是从内存访问数据速度的300倍，是从磁盘中访问数据速度的30万倍。

如果能从CPU寄存器中访问数据对程序的性能提升意义非凡，向量化执行就是在寄存器层面操作数据，为上层应用程序的性能带来了指数级的提升。

**何为向量化执行？向量化执行，可以简单地看作一项消除程序中循环的优化。**这里用一个形象的例子比喻。小胡经营了一家果汁店，虽然店里的鲜榨苹果汁深受大家喜爱，但客户总是抱怨制作果汁的速度太慢。小胡的店里只有一台榨汁机，每次他都会从篮子里拿出一个苹果，放到榨汁机内等待出汁。如果有8个客户，每个客户都点了一杯苹果汁，那么小胡需要重复循环8次上述的榨汁流程，才能榨出8杯苹果汁。如果制作一杯果汁需要5分钟，那么全部制作完毕则需要40分钟。为了提升果汁的制作速度，小胡想出了一个办法。他将榨汁机的数量从1台增加到了8台，这么一来，他就可以从篮子里一次性拿出8个苹果，分别放入8台榨汁机同时榨汁。此时，小胡只需要5分钟就能够制作出8杯苹果汁。**为了制作n杯果汁，非向量化执行的方式是用1台榨汁机重复循环制作n次，而向量化执行的方式是用n台榨汁机只执行1次。**



为了实现向量化执行，需要利用CPU的SIMD指令，SIMD的全称是Single Instruction Multiple Data，即用单条指令操作多条数据。现代计算机系统概念中，它是通过数据并行以提高性能的一种实现方式(其他的还有指令级并行和线程级并行)，它的原理是在CPU寄存器层面实现数据的并行操作。

ClickHouse列式存储除了降低IO和存储的压力之外，还为向量化执行做好了铺垫，除了读取磁盘速度快之外，ClickHouse还利用SSE4.2指令集实现向量化执行，为处理数据提升很大效率。

#### 关系模型与标准SQL查询

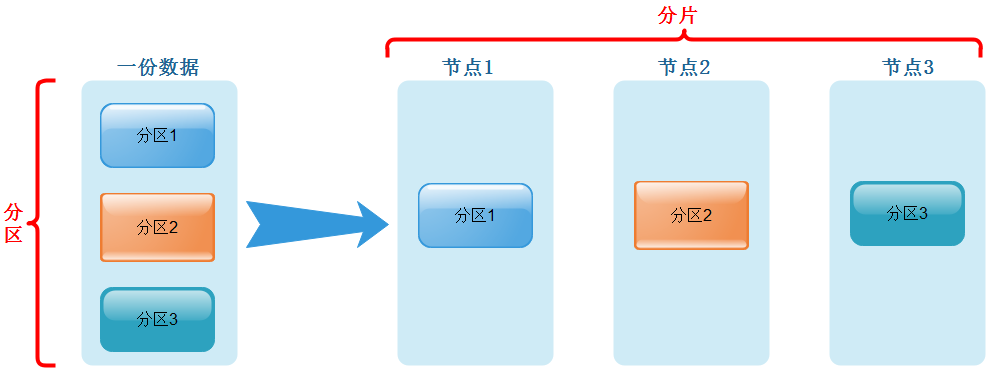
相比HBase和Redis这类NoSQL数据库，ClickHouse使用关系模型描述数据并提供了传统数据库的概念(数据库、表、视图和函数等)。ClickHouse完全使用SQL作为查询语言(支持GROUP BY、ORDER BY、JOIN、IN等大部分标准SQL)，ClickHouse提供了标准协议的SQL查询接口，可以与第三方分析可视化系统无缝集成对接。在SQL解析方面，ClickHouse是大小写敏感，SELECT a 和 SELECT A所代表的语义不同。

#### 多样化的表引擎

与MySQL类似，ClickHouse也将存储部分进行了抽象，把存储引擎作为一层独立的接口。ClickHouse拥有各种表引擎，每种表引擎决定不同的数据存储方式。其中每一种表引擎都有着各自的特点，用户可以根据实际业务场景的要求，选择合适的表引擎使用。将表引擎独立设计的好处是通过特定的表引擎支撑特定的场景，十分灵活，对于简单的场景，可直接使用简单的引擎降低成本，而复杂的场景也有合适的选择。

#### 多线程与分布式

向量化执行是通过数据级并行的方式提升了性能，多线程处理是通过线程级并行的方式实现了性能的提升。相比基于底层硬件实现的向量化执行SIMD，线程级并行通常由更高层次的软件层面控制，目前市面上的服务器都支持多核心多线程处理能力。由于SIMD不适合用于带有较多分支判断的场景，ClickHouse也大量使用了多线程技术以实现提速，以此和向量化执行形成互补。ClickHouse在数据存取方面，既支持分区(纵向扩展，利用多线程原理 )，也支持分片(横向扩展，利用分布式原理)，可以说是将多线程和分布式的技术应用到了极致。



#### 多主架构

HDFS、Spark、HBase和Elasticsearch这类分布式系统，都采用了Master-Slave主从架构，由一个管控节点作为Leader统筹全局。而ClickHouse则采用Multi-Master多主架构，集群中的每个节点角色对等，客户端访问任意一个节点都能得到相同的效果。这种多主的架构有许多优势，例如对等的角色使系统架构变得更加简单，不用再区分主控节点、数据节点和计算节点，集群中的所有节点功能相同。所以它天然规避了单点故障的问题，非常适合用于多数据中心、异地多活的场景。

#### 交互式查询

Hive,SparkSQL,HBase等都支持海量数据的查询场景，都拥有分布式架构，都支持列存、数据分片、计算下推等特性。ClickHouse在设计上吸取了以上技术的优势，例如：Hive、SparkSQL无法保证90%以上的查询在秒级内响应，在大数据量复杂查询下需要至少分钟级的响应时间，而HBase可以对海量数据做到交互式查询，由于不支持标准SQL在对数据做OLAP聚合分析时显得捉襟见肘。ClickHouse吸取以上各个技术的优势，在复杂查询的场景下，它也能够做到极快响应，且无须对数据进行任何预处理加工。

#### 数据分片与分布式查询

数据分片是将数据进行横向切分，这是一种在面对海量数据的场景下，解决存储和查询瓶颈的有效手段，是一种分治思想的体现。ClickHouse支持分片，而分片则依赖集群。每个集群由1到多个分片组成，而每个分片则对应了ClickHouse的1个服务节点。分片的数量上限取决于节点数量（1个分片只能对应1个服务节点）。

ClickHouse拥有高度自动化的分片功能。ClickHouse提供了本地表 （Local Table）与分布式表（Distributed Table）的概念。一张本地表等同于一份数据的分片。而分布式表本身不存储任何数据，它是本地表的访问代理，其作用类似分库中间件。借助分布式表，能够代理访问多个数据分片，从而实现分布式查询。

这种设计类似数据库的分库和分表，十分灵活。例如在业务系统上线的初期，数据体量并不高，此时数据表并不需要多个分片。所以使用单个节点的本地表（单个数据分片）即可满足业务需求，待到业务增长、数据量增大的时候，再通过新增数据分片的方式分流数据，并通过分布式表实现分布式查询。

## ClickHouse安装

Clickhouse官网为：[https://clickhouse.tech/,在官网中可以看到ClickHouse可以基于多种方式安装，rpm安装、tgz安装包安装、docker镜像安装、源码编译安装等。这里我们使用rpm安装包安装。](https://clickhouse.tech/，在官网中可以看到ClickHouse可以基于多种方式安装，rpm安装、tgz安装包安装、docker镜像安装、源码编译安装等。这里我们使用rpm安装包安装。)目前Clickhouse仅支持Linux系统且cpu必须支持SSE4.2指令集，可以通过以下命令查询Linux是否支持：

|  |
| --- |
| **grep -q sse4\_2 /proc/cpuinfo && echo "SSE 4.2 supported" || echo "SSE 4.2 not supported"** |

如果服务器不支持SSE4.2指令集，则不能下载预编译安装包，需要通过源码编译特定版本进行安装。

### **rpm安装包下载**

ClickHouse rpm安装包查询地址为:https://packagecloud.io/Altinity/clickhouse，这里需要在linux中使用wget命令下载对应的clickHouse版本。选择一台服务器创建/software目录并进入此目录，在当前目录下执行如下命令下载ClickHouse需要的rpm安装包，这里只需要下载以下四个rpm安装包即可。

|  |
| --- |
| wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-common-static-20.8.3.18-1.el7.x86\_64.rpm/download.rpm  wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-server-common-20.8.3.18-1.el7.x86\_64.rpm/download.rpm  wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-server-20.8.3.18-1.el7.x86\_64.rpm/download.rpm  wget --content-disposition https://packagecloud.io/Altinity/clickhouse/packages/el/7/clickhouse-client-20.8.3.18-1.el7.x86\_64.rpm/download.rpm |

### **单节点安装**

选择一台服务器，将下载好的clickHouse安装包直接安装即可，安装顺序如下：

|  |
| --- |
| **rpm -ivh clickhouse-common-static-20.8.3.18-1.el7.x86\_64.rpm**  **rpm -ivh clickhouse-server-common-20.8.3.18-1.el7.x86\_64.rpm**  **rpm -ivh clickhouse-server-20.8.3.18-1.el7.x86\_64.rpm**  **rpm -ivh clickhouse-client-20.8.3.18-1.el7.x86\_64.rpm**  这里也可以在当前节点直接执行如下命令，也可以按照依赖关系安装各个rpm包：  **rpm -ivh ./clickhouse-\*.rpm**  *Preparing... ################################# [100%]*  *Updating / installing...*  *1:clickhouse-server-common-20.8.3.1################################# [ 25%]*  *2:clickhouse-common-static-20.8.3.1################################# [ 50%]*  *3:clickhouse-server-20.8.3.18-1.el7################################# [ 75%]*  *Create user clickhouse.clickhouse with datadir /var/lib/clickhouse*  *4:clickhouse-client-20.8.3.18-1.el7################################# [100%]*  *Create user clickhouse.clickhouse with datadir /var/lib/clickhouse* |

#### 目录介绍

安装完成之后会生成如下对应的目录，每个目录的介绍如下：

* /etc/clickhouse-server : 服务端的配置文件目录，包括全局配置config.xml 和用户配置users.xml。
* /var/lib/clickhouse : 默认的数据存储目录，通常会修改，将数据保存到大容量磁盘路径中。
* /var/log/cilckhouse-server : 默认保存日志的目录，通常会修改，将数据保存到大容量磁盘路径中。
* 在/usr/bin下会有可执行文件：

clickhouse:主程序可执行文件

clickhouse-server:一个指向clickhouse可执行文件的软连接，供服务端启动使用。

clickhouse-client:一个指向clickhouse可执行文件的软连接，供客户端启动使用。

#### 启动&停止服务

启动clickhouse-server服务：

|  |
| --- |
| **service clickhouse-server start** |

启动clickhouse服务后可以使用命令行客户端连接到服务：

|  |
| --- |
| #client客户端连接到ch服务  **clickhouse-client**  或者使用命令：  **clickhouse-client --host localhost --port 9000**  *ClickHouse client version 20.8.3.18.*  *Connecting to localhost:9000 as user default.*  *Connected to ClickHouse server version 20.8.3 revision 54438.*  *node1 :)*  #查看9000 端口占用情况  [root@node5 bin]# **yum install net-tools**  [root@node5 bin]# **netstat -tunlp |grep 9000**  #查看当前所有数据库  **show databases;**  *┌─name───────────────────────────┐*  *│ \_temporary\_and\_external\_tables │*  *│ default │*  *│ system │*  *└────────────────────────────────┘*  #查看当前使用的数据库  **select database();**  *┌─database()─┐*  *│ default │*  *└────────────┘*  #退出客户端  **quit;** |

关闭ClickHouse服务：

|  |
| --- |
| **service clickhouse-server stop** |

### **分布式安装**

#### clickhouse分布式安装

Clickhouse支持分布式搭建，搭建步骤如下：

1. **选择三台clickhouse节点，在每台节点上安装clickhouse需要的安装包**

这里选择node1、node2,node3三台节点，分别按照clickhouse单节点安装方式在每台节点上安装clickhouse。

1. **安装zookeeper集群并启动。**

搭建Clickhouse集群时，需要使用Zookeeper去实现集群副本之间的同步，所以这里需要zookeeper集群，zookeeper集群安装后可忽略此步骤。

1. **配置外网可访问**

在每台clickhouse节点中配置/etc/clickhouse-server/config.xml文件第114行<listen\_host>，如下：

|  |
| --- |
| **<listen\_host>::</listen\_host>** |

1. **在每台节点/etc/目录下创建metrika.xml文件，写入以下内容**

在node1、node2、node3节点上/etc/下配置metrika.xml文件：

vim /etc/metrika.xml:

|  |
| --- |
| <yandex>  <clickhouse\_remote\_servers>  <clickhouse\_cluster\_3shards\_1replicas>  <shard>  <internal\_replication>true</internal\_replication>  <replica>  <host>node1</host>  <port>9000</port>  </replica>  </shard>  <shard>  <replica>  <internal\_replication>true</internal\_replication>  <host>node2</host>  <port>9000</port>  </replica>  </shard>  <shard>  <internal\_replication>true</internal\_replication>  <replica>  <host>node3</host>  <port>9000</port>  </replica>  </shard>  </clickhouse\_cluster\_3shards\_1replicas>  </clickhouse\_remote\_servers>    <zookeeper-servers>  <node index="1">  <host>node3</host>  <port>2181</port>  </node>  <node index="2">  <host>node4</host>  <port>2181</port>  </node>  <node index="3">  <host>node5</host>  <port>2181</port>  </node>  </zookeeper-servers>  <macros>  <replica>01</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> |

对以上配置文件中配置项的解释如下:

* clickhouse\_remote\_servers：

clickhouse集群配置标签，固定写法。

* clickhouse\_cluster\_3shards\_1replicas:

配置clickhouse的集群名称，可自由定义名称，注意集群名称中不能包含点号。这里代表集群中有3个分片，每个分片有1个副本。

分片是指包含部分数据的服务器，要读取所有的数据，必须访问所有的分片。

副本是指存储分片备份数据的服务器，要读取所有的数据，访问任意副本上的数据即可。

* shard：

分片，一个clickhouse集群可以分多个分片，每个分片可以存储数据，这里**分片可以理解为clickhouse机器中的每个节点**。这里可以配置一个或者任意多个分片，在每个分片中可以配置一个或任意多个副本，不同分片可配置不同数量的副本。如果只是配置一个分片，这种情况下查询操作应该称为远程查询，而不是分布式查询。

* replica：

每个分片的副本，默认每个分片配置了一个副本。也可以配置多个。如果配置了副本，读取操作可以从每个分片里选择一个可用的副本。如果副本不可用，会依次选择下个副本进行连接。该机制利于系统的可用性。

* internal\_replication：

默认为false,写数据操作会将数据写入所有的副本，设置为true,写操作只会选择一个正常的副本写入数据，数据的同步在后台自动进行。

* zookeeper-servers：

配置的zookeeper集群

* macros：

区分每台clickhouse节点的宏配置，每台clickhouse需要配置不同名称。

* networks：

这里配置ip为“::/0”代表任意IP可以访问，包含IPv4和IPv6。

注意：允许外网访问还需配置/etc/clickhouse-server/config.xml 参照第三步骤。

* clickhouse\_compression：

MergeTree引擎表的数据压缩设置，min\_part\_size：代表数据部分最小大小。min\_part\_size\_ratio：数据部分大小与表大小的比率。method：数据压缩格式。

**注意：需要在每台clickhouse节点上配置metrika.xml文件，并且修改每个节点的 macros配置名称。**

1. **在每台节点上启动clickhouse服务**

首先启动zookeeper集群，然后分别在node1、node2、node3节点上启动clickhouse服务，这里每台节点和单节点启动一样。启动之后，clickhouse集群配置完成。

|  |
| --- |
| **service clickhouse-server start** |

1. **检查集群配置是否完成**

在node1、node2、node3任意一台节点进入clickhouse客户端，查询集群配置：

|  |
| --- |
| #选择三台clickhouse任意一台节点，进入客户端  **clickhouse-client**  #查询集群信息，看到下图所示即代表集群配置成功。  node1 :) **select \* from system.clusters;** |

### **rpm其他方式安装问题**

以上介绍clickhouse安装方式是下载好rpm包之后进行安装，我们也可以自己配置clickhouse的yum源，直接使用yum命令进行安装，不过这个过程是从外网直接下载clickhouse安装包之后自动进行安装。配置如下：

#### 添加官方存储库

选择需要安装clickhouse的节点执行如下命令，添加clickhouse的官方yum源：

|  |
| --- |
| [root@node1 ~]# **yum -y install yum-utils**  [root@node1 ~]# **rpm --import <https://repo.yandex.ru/clickhouse/CLICKHOUSE-KEY.GPG>**  [root@node1 ~]# **yum-config-manager --add-repo <https://repo.yandex.ru/clickhouse/rpm/stable/x86_64>**  *Loaded plugins: fastestmirror*  *adding repo from: https://repo.yandex.ru/clickhouse/rpm/stable/x86\_64*  *[repo.yandex.ru\_clickhouse\_rpm\_stable\_x86\_64]*  *name=added from: https://repo.yandex.ru/clickhouse/rpm/stable/x86\_64*  *baseurl=https://repo.yandex.ru/clickhouse/rpm/stable/x86\_64*  *enabled=1* |

#### 安装clickhouse server和client

|  |
| --- |
| **yum -y install clickhouse-server clickhouse-client** |

经过以上两个步骤即可完成安装，这里安装单机和安装集群都可以使用以上这种模式。

**注意：在Centos7 中采用配置yum 源方式安装ClickHouse启动时使用命令：systemctl start clickhouse-server**

## 客户端命令行参数

我们可以通过clickhouse client来连接启动的clickhouse服务，连接服务时，我们可以指定以下参数，这里指定的参数会覆盖默认值和配置文件中的配置。

|  |  |
| --- | --- |
| **参数** | **解释** |
| --host, -h | 服务端的host名称, 默认是localhost。您可以选择使用host名称或者IPv4或IPv6地址。 |
| --port | 连接的端口，默认值：9000。注意HTTP接口以及TCP原生接口使用的是不同端口。 |
| --user, -u | 用户名。 默认值：default。 |
| --password | 密码。 默认值：空字符串。 |
| --query，-q | 使用非交互模式查询。 |
| --database, -d | 默认当前操作的数据库. 默认值：服务端默认的配置（默认是default）。 |
| --multiline, -m | 如果指定，允许多行语句查询（Enter仅代表换行，不代表查询语句完结）。 |
| --time, -t | 如果指定，**非交互模式下**会打印查询执行的时间到stderr中。 |
| --stacktrace | 如果指定，如果出现异常，会打印堆栈跟踪信息。 |
| --config-file | 配置文件的名称。 |
| --multiquery，-n | 使用非交互模式查询数据时，可以分号隔开多个sql语句。 |

* **--host，-h:**

使用-h指定ip或者host名称时，需要在/etc/clickhouse-server/config.xml配置文件中114行配置：<listen\_host>::</listen\_host> ，代表可以任意ip可访问。配置完成后需要重启当期clickhouse节点生效。

|  |
| --- |
| **clickhouse-client -h node1**  *ClickHouse client version 20.8.3.18.*  *Connecting to node1:9000 as user default.*  *Connected to ClickHouse server version 20.8.3 revision 54438.*  *node1 :)* |

* **--query，-q**

|  |
| --- |
| **clickhouse-client -q "show databases"**  *\_temporary\_and\_external\_tables*  *default*  *system* |

* **--database, -d:**

|  |
| --- |
| **clickhouse-client -d "system" -q "show tables"**  *aggregate\_function\_combinators*  *asynchronous\_metric\_log*  *asynchronous\_metrics*  *build\_options*  *... ....* |

* **--multiline, -m:**

|  |
| --- |
| **clickhouse-client -m**  *ClickHouse client version 20.8.3.18.*  *Connecting to localhost:9000 as user default.*  *Connected to ClickHouse server version 20.8.3 revision 54438.*  node1 :) **select**  :-] **1+1**  :-] **;**  *SELECT 1 + 1*  *┌─plus(1, 1)─┐*  *│ 2 │*  *└────────┘*  *1 rows in set. Elapsed: 0.004 sec.* |

* **--time, -t:**

|  |
| --- |
| **clickhouse-client -t -q "show databases"**  *\_temporary\_and\_external\_tables*  *default*  *system*  ***0.004*** |

* **--stacktrace:**

|  |
| --- |
| **clickhouse-client --stacktrace**  *ClickHouse client version 20.8.3.18.*  *Connecting to localhost:9000 as user default.*  *Connected to ClickHouse server version 20.8.3 revision 54438.*  node1 :) **use aaa;**  *USE aaa*  *Received exception from server (version 20.8.3):*  *Code: 81. DB::Exception: Received from localhost:9000. DB::Exception: Database aaa doesn't exist.* ***Stack trace:***   1. *Poco::Exception::Exception(std::\_\_1 ... ....*   *... ....* |

* **--multiquery，-n**

|  |
| --- |
| [root@node1 ~]# **clickhouse-client -n -q "show databases;use default;"**  *\_temporary\_and\_external\_tables*  *default*  *system* |

## 数据类型

ClickHouse提供了许多数据类型，它们可以划分为基础类型、复合类型和特殊类型。我们可以在system.data\_type\_families表中检查数据类型名称以及是否区分大小写。这个表中存储了ClickHouse支持的所有数据类型。

|  |
| --- |
| **select \* from system.data\_type\_families limit 10;**  *SELECT \**  *FROM system.data\_type\_families*  *LIMIT 10*  *┌─name────────────┬─case\_insensitive─┬─alias\_to─┐*  *│ Polygon │ 0 │ │*  *│ Ring │ 0 │ │*  *│ MultiPolygon │ 0 │ │*  *│ IPv6 │ 0 │ │*  *│ IntervalSecond │ 0 │ │*  *│ IPv4 │ 0 │ │*  *│ UInt32 │ 0 │ │*  *│ IntervalYear │ 0 │ │*  *│ IntervalQuarter │ 0 │ │*  *│ IntervalMonth │ 0 │ │*  *└─────────────────┴──────────────────┴──────────┘*  *10 rows in set. Elapsed: 0.004 sec.* |

下面介绍下常用的数据类型，ClickHouse与Mysql、Hive中常用数据类型的对比图如下：

|  |  |  |
| --- | --- | --- |
| **MySQL** | **Hive** | **ClickHouse(区分大小写)** |
| byte | TINYINT | Int8 |
| short | SMALLINT | Int16 |
| int | INT | Int32 |
| long | BIGINT | Int64 |
| varchar | STRING | String |
| timestamp | TIMESTAMP | DateTime |
| float | FLOAT | Float32 |
| double | DOUBLE | Float64 |
| boolean | BOOLEAN | 无 |

### **Int**

ClickHouse中整形分为Int8、Int16、Int32、Int64来表示整数不同的取值范围，其末尾数字正好代表占用字节的大小（8位=1字节），整形又包含有符号整形和无符号整形，他们写法上的区别为无符号整形前面加“U”表示。

* 有符号整型范围

|  |  |  |
| --- | --- | --- |
| **类型** | **字节** | **范围** |
| Int8 | 1 | [-128:127] |
| Int16 | 2 | [-32768:32767] |
| Int32 | 4 | [-2147483648:2147483647] |
| Int64 | 8 | [-9223372036854775808:9223372036854775807] |

* 无符号整形范围：

|  |  |  |
| --- | --- | --- |
| **类型** | **字节** | **范围** |
| UInt8 | 1 | [0:255] |
| UInt16 | 2 | [0:65535] |
| UInt32 | 4 | [0:4294967295] |
| UInt64 | 8 | [0:18446744073709551615] |

### **Float**

我们建议使用整数方式来存储数据，因为浮点类型数据计算可能导致四舍五入的误差。浮点类型包含单精度浮点数和双精度浮点数。

* 单精度浮点数

|  |  |  |
| --- | --- | --- |
| **类型** | **字节** | **有效精度位数** |
| Float32 | 4 | 7 |

Float32从小数点后第8位起会发生数据溢出。

* 双精度浮点数

|  |  |  |
| --- | --- | --- |
| **类型** | **字节** | **有效精度位数** |
| Float64 | 8 | 16 |

Float64从小数点后第17位起会发生数据溢出。

* 示例
  + toFloat32(...) 用来将字符串转换成Float32类型的函数
  + toFloat64(...) 用来将字符串转换成Float64类型的函数

|  |
| --- |
| #浮点数有可能导致数据误差  node1 :) **select 1-0.9**  *SELECT 1 - 0.9*  *┌───────minus(1, 0.9)─┐*  *│ 0.09999999999999998 │*  *└─────────────────────┘*  *1 rows in set. Elapsed: 0.021 sec.*  #Float32类型，从第8位开始产生溢出，会四舍五入。  node1 :) **select toFloat32(0.123456789);**  *SELECT toFloat32(0.123456789)*  *┌─toFloat32(0.123456789)─┐*  *│ 0.12345679 │*  *└────────────────────────┘*  *1 rows in set. Elapsed: 0.004 sec.*  # Float64类型，从第17为开始产生数据溢出，会四舍五入  node1 :) **select toFloat64(0.12345678901234567890);**  *SELECT toFloat64(0.12345678901234568)*  *┌─toFloat64(0.12345678901234568)─┐*  *│ 0.12345678901234568 │*  *└────────────────────────────────┘*  *1 rows in set. Elapsed: 0.006 sec.* |

### **Decimal**

有符号的定点数，可在加、减和乘法运算过程中保持精度。ClickHouse提供了Decimal32、Decimal64、Decimal128、Decimal256几种精度的定点数，支持几种写法：

* Decimal(P,S)
* Decimal32(S),数据范围：(-1\*10^(9-S),1\*10^(9-S))
* Decimal64(S),数据范围：(-1\*10^(18-S),1\*10^(18-S))
* Decimal128(S),数据范围：(-1\*10^(38-S),1\*10^(38-S))
* Decimal256(S),数据范围：(-1\*10^(76-S),1\*10^(76-S))

其中，P代表精度，决定总位数（整数部分+小数部分），取值范围是1~76。S代表规模，决定小数位数，取值范围是0~P。

根据P值的范围可以有如下对等写法，这里以小数点后2位举例：

|  |  |  |
| --- | --- | --- |
| **P取值** | **原生写法示例** | **等同于** |
| [1:9] | Decimal(9,2) | Decimal32(2) |
| [10:18] | Decimal(18,2) | Decimal64(2) |
| [19:38] | Decimal(38,2) | Decimal128(2) |
| [39:76] | Decimal(76,2) | Decimal256(2) |

另外，Decimal数据在进行四则运算时，精度（总位数）和规模（小数点位数）会发生变化，具体规则如下：

* **精度（总位数）对应规则：**
* Decimal64(S1) 运算符 Decimal32(S2) -> Decimal64(S)
* Decimal128(S1) 运算符 Decimal32(S2) -> Decimal128(S）
* Decimal128(S1) 运算符 Decimal64(S2) -> Decimal128(S)
* Decimal256(S1) 运算符Decimal<32|64|128>(S2) -> Decimal256(S)

两个不同精度的数据进行四则运算时，结果数据的精度以最大精度为准。

* **规模（小数点位数）对应规则：**
* 加法|减法：S=max(S1,S2)，即以两个数据中小数点位数最多的为准。
* 乘法：S=S1+S2(注意：S1精度>=S2精度)，即以两个数据的小数位相加为准。
* 除法：规模以被除数的小数位为准。两数相除，**被除数的小数位数不能小于除数的小数位数，**也就是触发的规模可以理解为与两个数据中小数点位数大的为准。举例：a/b ，a是被除数，与a的规模保持一致。
* 示例：
* toDecimal32(value,S):将字符串value转换为Decimal32类型,小数点后有S位。
* toTypeName(字段)：获取字段的数据类型函数。

|  |
| --- |
| #测试加法,S取两者最大的，P取两者最大的  node1 :) **select**  **toDecimal64(2,3) as x,**  **toTypeName(x) as xtype,**  **toDecimal32(2,2) as y,**  **toTypeName(y) as ytype,**  **x+y as z,**  **toTypeName(z) as ztype;**  结果如下：    #测试减法，S取两者最大的，P取两者最大的。  node1 :) **select**  **toDecimal64(2,3) as x,**  **toTypeName(x) as xtype,**  **toDecimal32(2,2) as y,**  **toTypeName(y) as ytype,**  **x-y as z,**  **toTypeName(z) as ztype;**  结果如下：    #测试乘法，S取两者最大的，P取两者小数位之和。  node1 :) **select**  **toDecimal64(2,3) as x,**  **toTypeName(x) as xtype,**  **toDecimal32(2,2) as y,**  **toTypeName(y) as ytype,**  **x\*y as z,**  **toTypeName(z) as ztype;**  结果如下：    #测试除法，S取两者最大的，P取被除数的小数位数。  node1 :) **select**  **toDecimal64(2,3) as x,**  **toTypeName(x) as xtype,**  **toDecimal32(2,2) as y,**  **toTypeName(y) as ytype,**  **x/y as z,**  **toTypeName(z) as ztype;**    node1 :) **select 1-toDecimal64(0.9,1);**  *SELECT 1 - toDecimal64(0.9, 1)*  *┌─minus(1, toDecimal64(0.9, 1))─┐*  *│ 0.1 │*  *└───────────────────────────────┘* |

### **String**

字符串可以是任意长度的。它可以包含任意的字节集，包含空字节。因此，字符串类型可以代替其他 DBMSs 中的VARCHAR、BLOB、CLOB 等类型。

### **FixedString**

固定长度N的字符串（N必须是严格的正自然数）,一般在明确字符串长度的场景下使用，可以使用下面的语法对列声明为FixedString类型：

|  |
| --- |
| # N表示字符串的长度。  <column\_name> FixedString(N) |

当向ClickHouse中插入数据时,如果字符串包含的字节数少于 N ,将对字符串末尾进行空字节填充。如果字符串包含的字节数大于N,将抛出Too large value for FixedString(N)异常。

当做数据查询时，ClickHouse不会删除字符串末尾的空字节。 如果使用WHERE子句，则须要手动添加空字节以匹配FixedString的值，新版本后期不需要手动添加。

* 示例：
  + toFixedString(value,N)：将字符串转换为N位长度，N不能小于value字符串实际长度。

|  |
| --- |
| #查看字符号串长度  node1 :) **select toFixedString('hello',6) as a,length(a) as alength;**  *SELECT*  *toFixedString('hello', 6) AS a,*  *length(a) AS alength*  *┌─a─────┬─alength─┐*  *│ hello │ 6 │*  *└───────┴─────────┘*  node1 :) **select toFixedString('hello world',6) as a,length(a) as alength;**  *SELECT*  *toFixedString('hello world', 6) AS a,*  *length(a) AS alength*  *Received exception from server (version 20.8.3):*  *Code: 131. DB::Exception: Received from localhost:9000. DB::Exception: String too long for type FixedString(6).* |

### **UUID**

UUID是一种数据库常见的主键类型，在ClickHouse中直接把它作为一种数据类型。UUID共有32位，它的格式为8-4-4-4-12，如果在插入新记录时未指定UUID列值，则UUID值将用0来填充（00000000-0000-0000-0000-000000000000）。

UUID类型不支持算术运算、聚合函数sum和avg。

* 示例：
  + generateUUIDv4()随机生成一个32位的UUID。

|  |
| --- |
| # 使用mydb库  node1 :) **use mydb;**  #创建表t\_uuid,指定x列为UUID类型，表引擎为TinyLog  node1 :) **CREATE TABLE t\_uuid (x UUID, y String) ENGINE=TinyLog**  #向表 t\_uuid中插入一条数据  node1 :) **INSERT INTO t\_uuid SELECT generateUUIDv4(), 'Example 1';**  #向表t\_uuid中插入一条数据，这里不指定UUID的值，默认会生成0来填充  node1 :) **INSERT INTO t\_uuid (y) VALUES ('Example 2')**  #查询结果  node1 :) **select \* from t\_uuid;**  *SELECT \**  *FROM t\_uuid*  *┌────────────────────────────────────x─┬─y─────────┐*  *│ 9c9f82dc-48a0-4749-b46a-cf6a1159c1fe │ Example 1 │*  *│ 00000000-0000-0000-0000-000000000000 │ Example 2 │*  *└──────────────────────────────────────┴───────────┘*  *2 rows in set. Elapsed: 0.003 sec.* |

### **Date**

Date只能精确到天，用两个字节存储，表示从1970-01-01（无符号）到当前的日期值。日期中没有存储时区信息，不能指定时区。

* 示例：
  + now() : 获取当前天日期，返回格式：yyyy-MM-dd HH:mm:ss
  + toDate(value) : 将字符串转成Date，只支持yyyy-MM-dd格式。

|  |
| --- |
| # 创建表t\_date  node1 :) **CREATE TABLE t\_date (x date) ENGINE=TinyLog;**  # 向表中插入两条数据  node1 :) **INSERT INTO t\_date VALUES('2021-06-01'),('2021-07-01');**  # 查询结果  node1 :) **SELECT x,toTypeName(x) FROM t\_date;**  *SELECT*  *x,*  *toTypeName(x)*  *FROM t\_date*  *┌──────────x─┬─toTypeName(x)─┐*  *│ 2021-06-01 │ Date │*  *│ 2021-07-01 │ Date │*  *└────────────┴───────────────┘*  *2 rows in set. Elapsed: 0.003 sec.*  # 获取当前天日期时间及日期转换  node1 :) **select now(),toDate(now()) as d,toTypeName(d) ;**  *SELECT*  *now(),*  *toDate(now()) AS d,*  *toTypeName(d)*  *┌───────────────now()─┬──────────d─┬─toTypeName(toDate(now()))─┐*  *│ 2021-01-25 18:03:00 │ 2021-01-25 │ Date │*  *└─────────────────────┴────────────┴───────────────────────────┘*  *1 rows in set. Elapsed: 0.004 sec.* |

### **DateTime**

DateTime精确到秒，可以指定时区。用四个字节（无符号的）存储Unix时间戳。允许存储与日期类型相同的范围内的值。最小值为0000-00-00 00:00:00，时间戳类型值精确到秒。

时区使用启动客户端或服务器时的系统时区。默认情况下，客户端连接到服务的时候会使用服务端时区。您可以通过启用客户端命令行选项 --use\_client\_time\_zone 来设置使用客户端时区。

* 示例：
  + toDateTime(DateTimeValue) ：将字符串转成DateTime，只支持yyyy-MM-dd HH:MI:SS。
  + toDateTime(DateTimeValue,时区) ：同上，支持将数据转换为对应时区时间。

|  |
| --- |
| # 创建表 t\_datetime  node1 :) **CREATE TABLE t\_datetime(`timestamp` DateTime) ENGINE = TinyLog;**  # 向表中插入一条数据  node1 :) **INSERT INTO t\_datetime Values('2021-06-01 08:00:00');**  # 查询数据  node1 :) **SELECT timestamp,toTypeName(timestamp) as t FROM t\_datetime;**  *SELECT*  *timestamp,*  *toTypeName(timestamp) AS t*  *FROM t\_datetime*  *┌───────────timestamp─┬─t────────┐*  *│ 2021-06-01 08:00:00 │ DateTime │*  *└─────────────────────┴──────────┘*  *1 rows in set. Elapsed: 0.003 sec*  # 转换时区查询  node1 :) **SELECT toDateTime(timestamp, 'Asia/Shanghai') AS column, toTypeName(column) AS x FROM t\_datetime;**  *SELECT*  *toDateTime(timestamp, 'Asia/Shanghai') AS column,*  *toTypeName(column) AS x*  *FROM t\_datetime*  *┌──────────────column─┬─x─────────────────────────┐*  *│ 2021-06-01 08:00:00 │ DateTime('Asia/Shanghai') │*  *└─────────────────────┴───────────────────────────┘*  *1 rows in set. Elapsed: 0.003 sec.* |

### **DateTime64**

DateTime64精确到毫秒和微秒，可以指定时区。在内部，此类型以Int64类型将数据存储。时间刻度的分辨率由precision参数确定。此外，DateTime64 类型可以像存储其他数据列一样存储时区信息，时区会影响 DateTime64 类型的值如何以文本格式显示，以及如何解析以字符串形式指定的时间数据 (‘2020-01-01 05:00:01.000’)。时区信息不存储在表的行中，而是存储在列的元数据中。

语法如下：

|  |
| --- |
| # precision 精度，timezone：时区  DateTime64(precision, [timezone]) |

* 示例：
  + toDateTime64(timeStr,precision):将字符串转成DateTime64，精度为precision。支持yyyy-MM-dd HH:MI:SS.SSS时间格式。
  + toDateTime64(timeStr,precision，timezone):同上，只是可以将时间转换为对应时区时间。

|  |
| --- |
| #创建表  node1 :) **CREATE TABLE dt(`timestamp` DateTime64(3, 'Europe/Moscow'),`event\_id` UInt8) ENGINE = TinyLog**  #插入数据  node1 :) **INSERT INTO dt Values (1546300800000, 1), ('2019-01-01 00:00:00', 2)，(1546300812345, 3)**  #查询数据  node1 :) **select \* from dt;**  *SELECT \**  *FROM dt*  *┌───────────────timestamp─┬─event\_id─┐*  *│ 2019-01-01 03:00:00.000 │ 1 │*  *│ 2019-01-01 00:00:00.000 │ 2 │*  *│ 2019-01-01 03:00:12.345 │ 3 │*  *└─────────────────────────┴──────────┘*  *3 rows in set. Elapsed: 0.003 sec.*  #使用toDateTime64转换时间  node1 :) **select toDateTime64(timestamp,4) as t1,toDateTime64(timestamp,4,'Europe/London') as t2,event\_id from dt;**  *SELECT*  *toDateTime64(timestamp, 4) AS t1,*  *toDateTime64(timestamp, 4, 'Europe/London') AS t2,*  *event\_id*  *FROM dt*  *┌───────────────────────t1─┬───────────────────────t2─┬─event\_id─┐*  *│ 2019-01-01 03:00:00.0000 │ 2019-01-01 00:00:00.0000 │ 1 │*  *│ 2019-01-01 00:00:00.0000 │ 2018-12-31 21:00:00.0000 │ 2 │*  *│ 2019-01-01 03:00:12.3450 │ 2019-01-01 00:00:12.3450 │ 3 │*  *└──────────────────────────┴──────────────────────────┴──────────┘*  *3 rows in set. Elapsed: 0.003 sec.* |

### **布尔类型**

ClickHouse中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。具体参照枚举类型。

### **枚举类型 Enum**

枚举类型通常在定义常量时使用，ClickHouse提供了Enum8和Enum16两种枚举类型。Enum保存'string'=integer的对应关系。在 ClickHouse 中，尽管用户使用的是字符串常量，但所有含有 Enum 数据类型的操作都是按照包含整数的值来执行。这在性能方面比使用 String 数据类型更有效。

Enum8和Enum16分别对应'String'=Int8和'String'=Int16，Enum8类型的每个值范围是-128 ... 127，Enum16类型的每个值范围是-32768 ... 32767，所有的字符串或者数字都必须是不一样的，允许存在空字符串，Enum类型中数字可以是任意顺序，顺序并不重要。

向Enum字段中插入值时，可以插入枚举的字符串值也可以插入枚举对应的Int值，建议插入对应的字符串值，这样避免插入对应的Int值不在Enum枚举集合中再次查询表时报错。定义了枚举类型值之后，不能写入其他值的数据，写入的值不在枚举集合中就会抛出异常。

* 示例：

|  |
| --- |
| #创建一个表，带有Enum类型的列  node1 :) **CREATE TABLE t\_enum(x Enum8('hello' = 1, 'world' = 2)) ENGINE = TinyLog**  #向表中插入数据  node1 :) **insert into t\_enum values('hello'),(2);**  #查询结果  node1 :) **select \* from t\_enum;**  *SELECT \**  *FROM t\_enum*  *┌─x─────┐*  *│ hello │*  *│ world │*  *└───────┘*  *2 rows in set. Elapsed: 0.003 sec.*  #插入不在枚举集合中的值时，抛出异常  node1 :) **insert into t\_enum values ('aa')**  *INSERT INTO t\_enum VALUES*  *Exception on client:*  *Code: 36. DB::Exception: Unknown element 'aa' for type Enum8('hello' = 1, 'world' = 2)*  *Connecting to database mydb at localhost:9000 as user default.*  *Connected to ClickHouse server version 20.8.3 revision 54438.*  #使用枚举类型代替boolean类型。建表，并插入数据  node1 :) **CREATE TABLE t\_enum2(bl Enum8('true' = 1, 'false' = 0)) ENGINE = TinyLog;**  node1 :) **insert into t\_enum2 values(0),(1);**  node1 :) **select \* from t\_enum2;**  *SELECT \**  *FROM t\_enum2*  *┌─bl────┐*  *│ false │*  *│ true │*  *└───────┘*  *2 rows in set. Elapsed: 0.004 sec.*  #查询时可以通过函数将对应的Enum底层数值获取出来  node1 :) **select toInt8(bl) from t\_enum2;**  *SELECT toInt8(bl)*  *FROM t\_enum2*  *┌─toInt8(bl)─┐*  *│ 0 │*  *│ 1 │*  *└────────────┘*  *2 rows in set. Elapsed: 0.004 sec.* |

### **Nullable**

Nullable类型只能与基础数据类型搭配使用，表示某个类型的值可以为NULL,Nullable(Int8)表示可以存储Int8类型的值，没有值时存NULL。使用Nullable需要注意：Nullable类型的字段不能作为索引字段，尽量避免使用Nullable类型，因为字段被定义为Nullable类型后会额外生成[Column].null.bin文件保存Null值，增加开销，比普通列消耗更多的存储空间。

* 示例：

|  |
| --- |
| #创建表，含有Nullable类型的列  node1 :) **CREATE TABLE t\_null(x Int8, y Nullable(Int8)) ENGINE TinyLog;**  #向表 t\_null中插入数据  node1 :) **INSERT INTO t\_null VALUES (1, NULL), (2, 3);**  #查询表t\_null中的数据  node1 :) **select \* from t\_null;**  *SELECT \**  *FROM t\_null*  *┌─x─┬────y─┐*  *│ 1 │ ᴺᵁᴸᴸ │*  *│ 2 │ 3 │*  *└───┴──────┘*  *2 rows in set. Elapsed: 0.007 sec*  #查询表t\_null做简单运算  node1 :) **SELECT x + y FROM t\_null;**  *SELECT x + y*  *FROM t\_null*  *┌─plus(x, y)─┐*  *│ ᴺᵁᴸᴸ │*  *│ 5 │*  *└────────────┘*  *2 rows in set. Elapsed: 0.004 sec.* |

### **数组类型 Array(T)**

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。

数组的定义方式有两种：Array(T),[1,2,3... ...],数组类型里面的元素必须具有相同的数据类型，否则会报异常。另外，需要注意的是，数组元素中如果存在Null值，则元素类型将变为Nullable。

从数组中查询获取值使用 xx[1|2.. ...]，直接使用中括号获取值，下标从1开始。

* 示例：

|  |
| --- |
| #两种方式定义数组  node1 :) **SELECT array(1, 2) AS x, toTypeName(x),['zs','ls','ww'] as y ,toTypeName(y)**  *SELECT*  *[1, 2] AS x,*  *toTypeName(x),*  *['zs', 'ls', 'ww'] AS y,*  *toTypeName(y)*  *┌─x─────┬─toTypeName(array(1, 2))─┬─y────────────────┬─toTypeName(['zs', 'ls', 'ww'])─┐*  *│ [1,2] │ Array(UInt8) │ ['zs','ls','ww'] │ Array(String) │*  *└───────┴─────────────────────────┴──────────────────┴────────────────────────────────┘*  *1 rows in set. Elapsed: 0.004 sec.*  # 数组中有Null，则数组类型为Nullable  node1 :) **SELECT array(1, 2, NULL) AS x, toTypeName(x);**  *SELECT*  *[1, 2, NULL] AS x,*  *toTypeName(x)*  *┌─x──────────┬─toTypeName(array(1, 2, NULL))─┐*  *│ [1,2,NULL] │ Array(Nullable(UInt8)) │*  *└────────────┴───────────────────────────────┘*  *1 rows in set. Elapsed: 0.006 sec.*  #数组类型里面的元素必须具有相同的数据类型，否则会报异常  node1 :) **SELECT array(1, 'a')**  *SELECT [1, 'a']*  *Received exception from server (version 20.8.3):*  *Code: 386. DB::Exception: Received from localhost:9000. DB::Exception: There is no supertype for types UInt8, String because some of them are String/FixedString and some of them are not.*  *0 rows in set. Elapsed: 0.005 sec.*  #创建表，含有array(T)类型字段，并插入数据  node1 :) **create table t\_array(id UInt32,name String,score Array(UInt32)) ENGINE = TinyLog;**  #插入数据，注意：字符串在clickhouse中只能是单引号  node1 :) **insert into t\_array values (1,'zs',array(10,20,30)),(2,'ls',[100,200,300])**  #数组内获取值  node1 :) **select id,name,score[1] from t\_array;**  *SELECT*  *id,*  *name,*  *score[1]*  *FROM t\_array*  *┌─id─┬─name─┬─arrayElement(score, 1)─┐*  *│ 1 │ zs │ 10 │*  *│ 2 │ ls │ 100 │*  *└──┴─── ─┴───────────────┘*  *5 rows in set. Elapsed: 0.004 sec.* |

### **Tuple类型**

元组类型有1~n个元素组成，每个元素允许设置不同的数据类型，且彼此之间不要求兼容。与数组类似，元组也可以使用两种方式定义：tuple(1,'hello',12.34)或者直接写(1,'hello',45.67),元组中可以存储多种数据类型，但是要注意数据类型的顺序。

* 示例：

|  |
| --- |
| #创建元组  node1 :) **SELECT tuple(1,'a') AS x, toTypeName(x),(1,'b','hello') AS y ,toTypeName(y)**  *SELECT*  *(1, 'a') AS x,*  *toTypeName(x),*  *(1, 'b', 'hello') AS y,*  *toTypeName(y)*  *┌─x───────┬─toTypeName(tuple(1, 'a'))─┬─y───────────────┬─toTypeName(tuple(1, 'b', 'hello'))─┐*  *│ (1,'a') │ Tuple(UInt8, String) │ (1,'b','hello') │ Tuple(UInt8, String, String) │*  *└─────────┴───────────────────────────┴─────────────────┴────────────────────────────────────┘*  *1 rows in set. Elapsed: 0.004 sec.*  #建表，含有元组类型  node1 :) **create table t\_tuple(id UInt8,name String,info Tuple(String,UInt8)) engine = TinyLog**  #插入数据  node1 :) **insert into t\_tuple values (1,'zs',tuple('cls1',100)),(2,'ls',('cls2',200))**  #查询数据  node1 :) **select \* from t\_tuple;**  *SELECT \**  *FROM t\_tuple*  *┌─id─┬─name─┬─info─────────┐*  *│ 1 │ zs │ ('cls1',100) │*  *│ 2 │ ls │ ('cls2',200) │*  *└────┴──────┴──────────────┘*  *2 rows in set. Elapsed: 0.003 sec.* |

### **嵌套类型Nested**

ClickHouse支持嵌套数据类型（Nested）,可以为一个表定义一个或者多个嵌套数据类型字段，但是每个嵌套字段只支持一级嵌套，即嵌套字段内不能继续使用嵌套类型。嵌套一般用来表示简单的级联关系，嵌套本质上是一个多维数组，嵌套类型中的每个数组的长度必须相同。目前，Nested类型支持很局限，MergeTree引擎中不支持Nested类型。

* 示例：

|  |
| --- |
| #创建一个表，每个人可以属于多个部门，在不同部门有不同的编号id  node1 :) **create table t\_nested(**  :-] **id UInt8,**  :-] **name String,**  :-] **dept Nested(**  :-] **id UInt8,**  :-] **name String**  :-] **)) engine = TinyLog;**  #查看表t\_nested的表结构  node1 :) **desc t\_nested;**  *DESCRIBE TABLE t\_nested*  *┌─name──────┬─type──────────┬*  *│ id │ UInt8 │*  *│ name │ String │*  *│ dept.id │ Array(UInt8) │*  *│ dept.name │ Array(String) │*  *└───────────┴───────────────┴*  *4 rows in set. Elapsed: 0.003 sec.*  #向表t\_nested中插入数据  node1 :) **insert into t\_nested values (1,'zs',[10,11,12],['dp1','dp2','dp3']),(2,'ls',[100,101],['dp4','dp5'])**  #查询表 t\_nested数据，可以获取嵌套类型中部分字段  node1 :) **select \*,dept.name[1] as first\_dpt from t\_nested;**  *SELECT*  *\*,*  *dept.name[1] AS first\_dpt*  *FROM t\_nested*  *┌─id─┬─name─┬─dept.id────┬─dept.name───────────┬─first\_dpt─┐*  *│ 1 │ zs │ [10,11,12] │ ['dp1','dp2','dp3'] │ dp1 │*  *│ 2 │ ls │ [100,101] │ ['dp4','dp5'] │ dp4 │*  *└────┴──────┴────────────┴─────────────────────┴───────────┘*  *2 rows in set. Elapsed: 0.004 sec.* |

### **Domain**

Domain类型是特定实现的类型，目前支持**IPv4**和**IPv6**两类，本质上他们是对整形和字符串的进一步封装，IPv4类型基于UInt32封装，IPv6基于FixedString(16)封装。

出于便捷性的考量，例如：IPv4类型支持格式检查，格式错误的IP无法被写入。出于性能的考量，IPv4和IPv6相对于String更加紧凑，占用的空间更小，查询性能更快。

在使用Domain时需要注意，虽然表面看起来与String一样，但是Domain类型并不是字符串，也不支持隐式自动转换成字符串，如果需要返回IP的字符串形式，需要调用函数IPv4NumToString()和IPv6NumToString()显式实现。

* 示例：（示例以IPv4为例，IPv6同理）

|  |
| --- |
| #创建表 t\_domain 含有IPv4字段  node1 :) **CREATE TABLE t\_domain(url String, from IPv4) ENGINE = TinyLog;**  #插入数据  node1 :) **INSERT INTO t\_domain(url, from) VALUES ('https://wikipedia.org', '116.253.40.133')('https://clickhouse.tech', '183.247.232.58')('https:/**  **/clickhouse.tech/docs/en/', '116.106.34.242');**  #查看表 t\_domain结果数据  node1 :) **select \* from t\_domain;**  *SELECT \**  *FROM t\_domain*  *┌─url──────────────────────────────┬───────────from─┐*  *│ https://wikipedia.org │ 116.253.40.133 │*  *│ https://clickhouse.tech │ 183.247.232.58 │*  *│ https://clickhouse.tech/docs/en/ │ 116.106.34.242 │*  *└───────────────────────────────────┴────────────────┘*  *3 rows in set. Elapsed: 0.004 sec.*  #插入数据不符合IP格式会报错  node1 :) **INSERT INTO t\_domain(url, from) VALUES ('https://www.baidu.com', '116.253.40')**  *INSERT INTO t\_domain (url, from) VALUES*  *Exception on client:*  *Code: 441. DB::Exception: Invalid IPv4 value.*  *Connecting to database mydb at localhost:9000 as user default.*  *Connected to ClickHouse server version 20.8.3 revision 54438.*  #将表 t\_domain 中from IPv4类型转换成String类型  node1 :) **SELECT from,toTypeName(from) as tp1,toTypeName(s) as tp2, IPv4NumToString(from) as s FROM t\_domain;**  *SELECT*  *from,*  *toTypeName(from) AS tp1,*  *toTypeName(s) AS tp2,*  *IPv4NumToString(from) AS s*  *FROM t\_domain*  *┌───────────from─┬─tp1──┬─tp2────┬─s──────────────┐*  *│ 116.253.40.133 │ IPv4 │ String │ 116.253.40.133 │*  *│ 183.247.232.58 │ IPv4 │ String │ 183.247.232.58 │*  *│ 116.106.34.242 │ IPv4 │ String │ 116.106.34.242 │*  *└────────────────┴──────┴────────┴────────────────┘*  *3 rows in set. Elapsed: 0.003 sec.* |

## ClickHouse数据库引擎

ClickHouse中支持在创建数据库时指定引擎，目前比较常用的两种引擎为默认引擎和MySQL数据库引擎。

### **Ordinary默认数据库引擎**

Ordinary就是ClickHouse中默认引擎，如果不指定数据库引擎创建的就是Ordinary数据库引擎，在这种数据库下面可以使用任意表引擎。创建时需要注意，Ordinary首字母需要大写，不然会抛出异常。

* 示例：

|  |
| --- |
| #在ClickHouse中创建数据库test1,指定数据库引擎为Ordinary  node1 :) **create database test1 engine = Ordinary;**  *CREATE DATABASE test1*  *ENGINE = Ordinary*  *Ok.*  *0 rows in set. Elapsed: 0.013 sec.*  #查看test1数据库建库信息，显示数据库引擎为Ordinary  node1 :) **show create database test1;**  *SHOW CREATE DATABASE test1*  *┌─statement──────────────────────────────┐*  *│ CREATE DATABASE test1*  *ENGINE = Ordinary │*  *└────────────────────────────────────────┘*  *1 rows in set. Elapsed: 0.002 sec.*  #在ClickHouse中创建数据库test2，这里默认不指定任何数据库引擎  node1 :) **create database test2;**  *CREATE DATABASE test2*  *Ok.*  *0 rows in set. Elapsed: 0.003 sec.*  #查看test2数据库建库信息，显示数据库引擎为Ordinary  node1 :) **show create database test2;**  *SHOW CREATE DATABASE test2*  *┌─statement──────────────────────────────┐*  *│ CREATE DATABASE test2*  *ENGINE = Ordinary │*  *└───────────────────────────────────────┘*  *1 rows in set. Elapsed: 0.002 sec.* |

### **MySQL数据库引擎**

MySQL引擎用于将远程的MySQL服务器中的表映射到ClickHouse中，并**允许对表进行INSERT插入和SELECT查询，**方便在ClickHouse与MySQL之间进行数据交换。这里不会将MySQL的数据同步到ClickHouse中，ClickHouse就像一个壳子，可以将MySQL的表映射成ClickHouse表，使用ClickHouse查询MySQL中的数据，在MySQL中进行的CRUD操作，可以同时映射到ClickHouse中。

MySQL数据库引擎会将对其的查询转换为MySQL语法并发送到MySQL服务器中，因此可以执行诸如SHOW TABLES或SHOW CREATE TABLE之类的操作，但是**不允许创建表、修改表、删除数据、重命名操作。**

ClickHouse中创建库使用MySQL引擎语法如下：

|  |
| --- |
| CREATE DATABASE [IF NOT EXISTS] db\_name [ON CLUSTER cluster]  ENGINE = MySQL('host:port', ['database' | database], 'user', 'password') |

以上参数解释：

* host:port — 链接的MySQL地址。
* database — 链接的MySQL数据库。
* user — 链接的MySQL用户。
* password — 链接的MySQL用户密码。

注意：“MySQL”引擎的写法严格区分大小写。

在ClickHouse中使用MySQL引擎建库，将MySQL库中数据映射到ClickHouse中，mysql库中表字段类型与ClickHouse表字段类型的映射如下，这里每种类型在ClickHouse中都支持Nullable,即可空。

|  |  |
| --- | --- |
| **Mysql** | **ClickHouse** |
| UNSIGNED TINYINT | UInt8 |
| TINYINT | Int8 |
| UNSIGNED SMALLINT | UInt16 |
| SMALLINT | Int16 |
| UNSIGNED INT  UNSIGNED MEDIUMINT | UInt32 |
| INT, MEDIUMINT | Int32 |
| UNSIGNED BIGINT | UInt64 |
| BIGINT | Int64 |
| FLOAT | Float32 |
| DOUBLE | Float64 |
| DATE | Date |
| DATETIME, TIMESTAMP | DateTime |
| BINARY | FixedString |

**注意：**

* + *在Mysql中UNSIGNED 表示无符号，就是没有负数。*
  + *TINYINT代表有符号的范围是-128-127，无符号的范围是从0到255的整型数据，占位大小为1字节。*
  + *SMALLINT:一个小整数。有符号的范围是-2^15(-32,768) 到 2^15-1(32,767)的整型数据，无符号的范围是0到65535，占位大小为2个字节。*
  + *MEDIUMINT：一个中等大小整数，有符号的范围是-8388608到8388607，无符号的范围是0到16777215，占位大小为3个字节。*
  + *INT:一个正常大小整数。有符号的范围是-2^31(-2,147,483,648)到2^31-1(2,147,483,647)的整型数据（所有数字），无符号的范围是0到4294967295,占位大小为 4 个字节。*
* 示例：ClickHouse映射MySQL中的表

|  |
| --- |
| **#登录mysql 在mysql中创建test数据库**  **mysql> create database test;**  *Query OK, 1 row affected (0.00 sec)*  #在mysql test库中新建表 mysql\_table  **mysql> use test;**  **mysql> create table mysql\_table(id int ,name varchar(255));**  #向mysql表 mysql\_table中插入两条数据  **mysql> insert into mysql\_table values (1,"zs"),(2,"ls");**  *Query OK, 2 rows affected (0.01 sec)*  *Records: 2 Duplicates: 0 Warnings: 0*  **#在ClickHouse中创建mysql引擎的数据库，与MySQL交换数据**  node1 :) **CREATE DATABASE mysql\_db ENGINE = MySQL('node2:3306', 'test', 'root', '123456');**  *CREATE DATABASE mysql\_db*  *ENGINE = MySQL('node2:3306', 'test', 'root', '123456')*  *Ok.*  *0 rows in set. Elapsed: 0.032 sec.*  #在ClickHouse中使用mysql\_db库，并展示表，看是否映射MySQL中的表  node1 :) **use mysql\_db;**  node1 :) **show tables;**  *SHOW TABLES*  *┌─name────────┐*  *│ mysql\_table │*  *└──────────┘*  *1 rows in set. Elapsed: 0.005 sec.*  #在ClickHouse中查询表mysql\_table  node1 :) **select \* from mysql\_table;**  *SELECT \**  *FROM mysql\_table*  *┌─id─┬─name─┐*  *│ 1 │ zs │*  *│ 2 │ ls │*  *└──┴────┘*  *2 rows in set. Elapsed: 0.012 sec.*  #在ClickHouse中查看表 mysql\_table的描述，发现映射成ClickHouse中的字段类型。  node1 :) **desc mysql\_table;**  *DESCRIBE TABLE mysql\_table*  *┌─name─┬─type────────┬*  *│ id │ Nullable(Int32) │*  *│ name │ Nullable(String) │*  *└────┴───────────┴*  *2 rows in set. Elapsed: 0.005 sec.* |

* 示例：在MySQL对应的表中插入删除数据，对应的在ClickHouse中也能插入和删除的数据

|  |
| --- |
| **#在MySQL中 test库下的mysql\_table中插入和删除一条数据**  mysql> **insert into mysql\_table values (3,"ww");**  *Query OK, 1 row affected (0.01 sec)*  mysql> **delete from mysql\_table where id = 1;**  *Query OK, 1 row affected (0.01 sec)*  **#在ClickHouse中 mysql\_db库下查询表mysql\_table**  node1 :) **select \* from mysql\_table;**  *SELECT \**  *FROM mysql\_table*  *┌─id─┬─name─┐*  *│ 2 │ ls │*  *│ 3 │ ww │*  *└────┴──────┘*  *2 rows in set. Elapsed: 0.005 sec.* |

* 示例：在MySQL对应的库test下创建新的表a，ClickHouse中也可以展示

|  |
| --- |
| **#在MySQL中 test库下创建库a,并插入数据**  mysql> **create table a (id int,name varchar(255),age int);**  *Query OK, 0 rows affected (0.02 sec)*  mysql> **insert into a values (1,"zhangsan",18),(2,"lisi",19);**  *Query OK, 2 rows affected (0.01 sec)*  *Records: 2 Duplicates: 0 Warnings: 0*  **#在ClickHouse中 mysql\_db库下查询表a是否存在，同时查看数据**  node1 :) **show tables;**  *SHOW TABLES*  *┌─name────────┐*  *│ a │*  *│ mysql\_table │*  *└─────────────┘*  *2 rows in set. Elapsed: 0.005 sec.*  node1 :) **select \* from a;**  *SELECT \**  *FROM a*  *┌─id─┬─name─────┬─age─┐*  *│ 1 │ zhangsan │ 18 │*  *│ 2 │ lisi │ 19 │*  *└────┴──────────┴─────┘*  *2 rows in set. Elapsed: 0.025 sec.* |

* 示例：在ClickHouse中向表a中插入数据，可以在msyql中查询到。但是ClickHouse中不支持创建表和删除数据操作。

|  |
| --- |
| **#在ClickHouse中向表a中插入数据**  node1 :) **insert into a values(3,'wangwu',20);**  **#在MySQL中查询表a数据**  mysql> **select \* from a;**  *+------+----------+------+*  *| id | name | age |*  *+------+----------+------+*  *| 1 | zhangsan | 18 |*  *| 2 | lisi | 19 |*  *| 3 | wangwu | 20 |*  *+------+----------+------+*  *3 rows in set (0.00 sec)* |

## ClickHouse表引擎

MySQL的数据表有InnoDB和MyISAM存储引擎，不同的存储引擎提供不同的存储机制、索引方式等功能，也可以称之为表类型。在ClickHouse中也有表引擎。

表引擎在ClickHouse中的作用十分关键，直接决定了**数据如何存储和读取、是否支持并发读写、是否支持index索引、支持的query种类、是否支持主备复制**等。

ClickHouse提供了大约28种表引擎，各有各的用途，比如有Log系列用来做小表数据分析，MergeTree系列用来做大数据量分析，而Integration系列则多用于外表数据集成。再考虑复制表Replicated系列，分布式表Distributed等，纷繁复杂。

ClickHouse表引擎一共分为四个系列，分别是Log系列、MergeTree系列、Integration系列、Special系列。其中包含了两种特殊的表引擎Replicated、Distributed，功能上与其他表引擎正交，根据场景组合使用。

### **Log系列表引擎**

Log系列表引擎功能相对简单，主要用于快速写入小表（1百万行左右的表），然后全部读出的场景，即一次写入，多次查询。Log系列表引擎包含：TinyLog、StripeLog、Log三种引擎。

* **几种Log表引擎的共性是：**
* 数据被顺序append写到本地磁盘上。
* 不支持delete、update修改数据。
* 不支持index（索引）。
* 不支持原子性写。如果某些操作(异常的服务器关闭)中断了写操作，则可能会获得带有损坏数据的表。
* insert会阻塞select操作。当向表中写入数据时，针对这张表的查询会被阻塞，直至写入动作结束。
* **它们彼此之间的区别是：**
* TinyLog：不支持并发读取数据文件，查询性能较差；格式简单，适合用来暂存中间数据。
* StripLog：支持并发读取数据文件，查询性能比TinyLog好；将所有列存储在同一个大文件中，减少了文件个数。
* Log：支持并发读取数据文件，查询性能比TinyLog好；每个列会单独存储在一个独立文件中。

#### TinyLog

TinyLog是Log系列引擎中功能简单、性能较低的引擎。

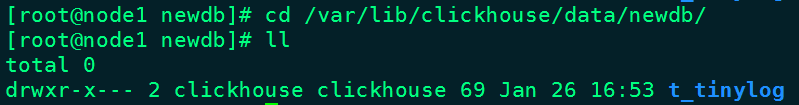
它的存储结构由数据文件和元数据两部分组成。其中，数据文件是按列独立存储的，也就是说每一个列字段都对应一个文件。

由于**TinyLog数据存储不分块**，所以不支持并发数据读取，该引擎适合一次写入，多次读取的场景，对于处理小批量中间表的数据可以使用该引擎，这种引擎会有大量小文件，性能会低。

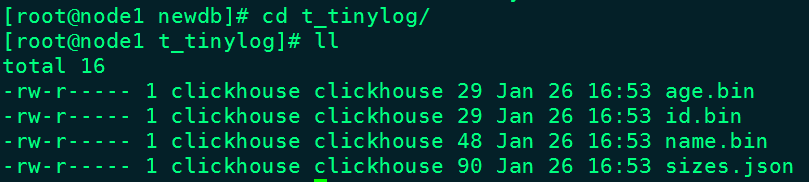
* 示例：

|  |
| --- |
| #在ch中创建库 newdb，并使用  node1 :) **create database newdb;**  node1 :) **use newdb;**  #创建表t\_tinylog 表，使用TinyLog引擎  node1 :) **create table t\_tinylog(id UInt8,name String,age UInt8) engine=TinyLog;**  #向表中插入数据  node1 :) **insert into t\_tinylog values (1,'张三',18),(2,'李四',19),(3,'王五',20);**  #查询表中的数据  node1 :) **select \* from t\_tinylog;**  *SELECT \**  *FROM t\_tinylog*  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *└────┴──────┴─────┘*  *3 rows in set. Elapsed: 0.003 sec.*  #在表中删除一条数据，这里是不支持delete。  node1 :) **delete from t\_tinylog where id = 1;//语句不适合CH**  node1 :) **alter table t\_tinylog delete where id = 1;**  *:Exception: Mutations are not supported by storage TinyLog.* |

当在newdb库中创建表t\_tinylog后，在ClickHouse保存数据的目录/var/lib/clickhouse/data/newdb/下会多一个t\_tinylog目录，如图所示：



在向表t\_tinylog中插入数据后，进入“t\_tinylog”目录，查看目录下的文件，如下图所示：



我们可以发现，表t\_tinylog中的每个列都单独对应一个\*.bin文件，同时还有一个sizes.json文件存储元数据，记录了每个bin文件中数据大小。

#### StripeLog

相比TinyLog而言，**StripeLog数据存储会划分块，每次插入对应一个数据块**，拥有更高的查询性能（拥有.mrk标记文件，支持并行查询）。StripeLog 引擎将所有列存储在一个文件中，使用了更少的文件描述符。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。StripeLog 引擎不支持 ALTER UPDATE 和 ALTER DELETE 操作。

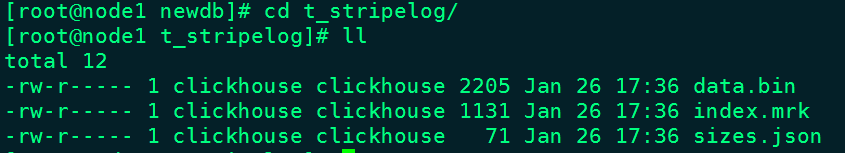
* 示例：

|  |
| --- |
| #在库 newdb中创建表 t\_stripelog，使用StripeLog引擎  node1 :) **create table t\_stripelog(id UInt8,name String,age UInt8) engine = StripeLog;**  #向表t\_stripelog中插入数据，这里插入分多次插入，会将数据插入不同的数据块中  node1 :) **insert into t\_stripelog values (1,'张三',18);**  node1 :) **insert into t\_stripelog values (2,'李四',19);**  #查询表 t\_stripelog数据  node1 :) **select \* from t\_stripelog;**  *SELECT \**  *FROM t\_stripelog*  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *└────┴──────┴─────┘*  *┌─id─┬─name─┬─age─┐*  *│ 2 │ 李四 │ 19 │*  *└────┴──────┴─────┘*  *2 rows in set. Elapsed: 0.003 sec.* |

当在newdb库中创建表 t\_stripelog后，在ClickHouse保存数据的目录/var/lib/clickhouse/data/newdb/下会多一个t\_stripelog目录，如图所示：



在向表t\_stripelog中插入数据后，进入“t\_stripelog”目录，查看目录下的文件，如下图所示：



我们可以发现只有三个文件：

* data.bin:数据文件，所有列字段都写入data.bin文件中。
* index.mrk:数据标记文件，保存了数据在data.bin 文件中的位置信息，即每个插入数据列的offset信息，利用数据标记能够使用多个线程，并行度取data.bin压缩数据，提升查询性能。
* sizes.json:元数据文件，记录了data.bin和index.mrk大小信息。

#### Log

Log引擎表适用于临时数据，一次性写入、测试场景。Log引擎结合了TinyLog表引擎和StripeLog表引擎的长处，是Log系列引擎中性能最高的表引擎。

Log表引擎会将每一列都存在一个文件中，**对于每一次的INSERT操作，会生成数据块**，经测试，**数据块个数与当前节点的core数一致。**

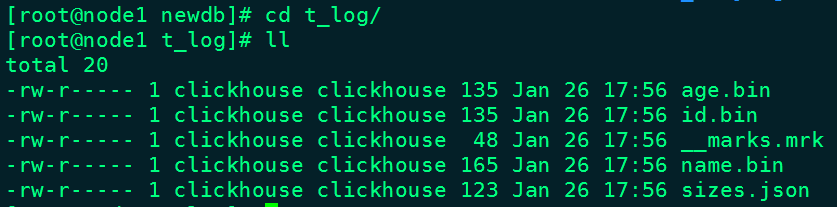
* 示例：

|  |
| --- |
| #在newdb中创建表t\_log 使用Log表引擎  node1 :) **create table t\_log(id UInt8 ,name String ,age UInt8 ) engine = Log;**  #向表 t\_log中插入数据，分多次插入，插入后数据存入数据块  node1 :) **insert into t\_log values (1,'张三',18);**  node1 :) **insert into t\_log values (2,'李四',19);**  node1 :) **insert into t\_log values (3,'王五',20);**  node1 :) **insert into t\_log values (4,'马六',21);**  node1 :) **insert into t\_log values (5,'田七',22);**  #查询表t\_log中的数据  node1 :) **select \* from t\_log;**  *SELECT \**  *FROM t\_log*  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *└────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┐*  *│ 3 │ 王五 │ 20 │*  *│ 4 │ 马六 │ 21 │*  *│ 5 │ 田七 │ 22 │*  *└────┴─────┴─────┘*  *5 rows in set. Elapsed: 0.004 sec.* |

当在newdb库中创建表 t\_log后，在ClickHouse保存数据的目录/var/lib/clickhouse/data/newdb/下会多一个t\_log目录，如图所示：



在向表t\_log中插入数据后，进入“t\_log”目录，查看目录下的文件，如下图所示：



我们发现表t\_log中的每个列都对应一个\*.bin文件。其他两个文件的解释如下：

* \_\_marks.mrk：数据标记，保存了每个列文件中的数据位置信息，利用数据标记能够使用多个线程，并行度取data.bin压缩数据，提升查询性能。
* sizes.json:记录了\*.bin 和\_\_mark.mrk大小的信息。

### **Special系列表引擎**

#### Memory

Memory表引擎直接将数据保存在内存中，ClickHouse中的Memory表引擎具有以下特点:

* Memory 引擎以未压缩的形式将数据存储在 RAM 中，数据完全以读取时获得的形式存储。
* 并发数据访问是同步的，锁范围小，读写操作不会相互阻塞。
* 不支持索引。
* 查询是并行化的，在简单查询上达到最大速率（超过10 GB /秒），在相对较少的行（最多约100,000,000）上有高性能的查询。
* 没有磁盘读取，不需要解压缩或反序列化数据，速度更快（在许多情况下，与 MergeTree 引擎的性能几乎一样高）。
* 重新启动服务器时，表存在，但是表中数据全部清空。
* Memory引擎多用于测试。
* 示例：

|  |
| --- |
| #在 newdb中创建表 t\_memory ，表引擎使用Memory  node1 :) **create table t\_memory(id UInt8 ,name String, age UInt8) engine = Memory;**  *CREATE TABLE t\_memory*  *(*  *`id` UInt8,*  *`name` String,*  *`age` UInt8*  *)*  *ENGINE = Memory*  *Ok.*  *0 rows in set. Elapsed: 0.008 sec.*  #向表 t\_memory中插入数据  node1 :) **insert into t\_memory values (1,'张三',18),(2,'李四',19),(3,'王五',20);**  *INSERT INTO t\_memory VALUES*  *Ok.*  *3 rows in set. Elapsed: 0.004 sec.*  #查询表t\_memory中的数据  node1 :) **select \* from t\_memory;**  *SELECT \**  *FROM t\_memory*  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *└────┴──────┴─────┘*  *3 rows in set. Elapsed: 0.004 sec.*  #重启clickhouse 服务  [root@node1 ~]# **service clickhouse-server restart**  *Stop clickhouse-server service: DONE*  *Start clickhouse-server service: Path to data directory in /etc/clickhouse-server/config.xml: /var/lib/clickhouse/*  *DONE*  #进入 newdb 库，查看表 t\_memory数据，数据为空。  node1 :) **select \* from t\_memory;**  *SELECT \**  *FROM t\_memory*  *Ok.*  *0 rows in set. Elapsed: 0.003 sec.* |

注意：”Memory”表引擎写法固定，不能小写。同时创建好表t\_memory后，在对应的磁盘目录/var/lib/clickhouse/data/newdb下没有“t\_memory”目录，基于内存存储，当重启ClickHouse服务后，表t\_memory存在，但是表中数据全部清空。

#### 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，这里需要多个表的结构相同，并且创建的Merge引擎表的结构也需要和这些表结构相同才能读取。

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如果设置了索引，索引也会被使用。

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：

|  |
| --- |
| Merge(数据库, 正则表达式) |

例如：Merge(hits, '^WatchLog') 表示数据会从 hits 数据库中表名匹配正则 ‘^WatchLog’ 的表中读取。

注意:当选择需要读取的表时，会匹配正则表达式匹配上的表，如果当前Merge表的名称也符合正则表达式匹配表名，这个Merge表本身会自动排除，以避免进入递归死循环，当然也可以创建两个相互无限递归读取对方数据的 Merge 表，但这并没有什么意义。

* 示例：

|  |
| --- |
| #在newdb库中创建表m\_t1，并插入数据  node1 :) **create table m\_t1 (id UInt8 ,name String,age UInt8) engine = TinyLog;**  node1 :) **insert into m\_t1 values (1,'张三',18),(2,'李四',19)**  #在newdb库中创建表m\_t2，并插入数据  node1 :) **create table m\_t2 (id UInt8 ,name String,age UInt8) engine = TinyLog;**  node1 :) **insert into m\_t2 values (3,'王五',20),(4,'马六',21)**  #在newdb库中创建表m\_t3，并插入数据  node1 :) **create table m\_t3 (id UInt8 ,name String,age UInt8) engine = TinyLog;**  node1 :) **insert into m\_t3 values (5,'田七',22),(6,'赵八',23)**  #在newdb库中创建表t\_merge，使用Merge引擎，匹配m开头的表  node1 :) **create table t\_merge (id UInt8,name String,age UInt8) engine = Merge(newdb,'^m');**  #查询 t\_merge表中的数据  node1 :) **select \* from t\_merge;**  *SELECT \**  *FROM t\_merge*  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *└────┴─────┴────┘*  *┌─id─┬─name─┬─age─┐*  *│ 3 │ 王五 │ 20 │*  *│ 4 │ 马六 │ 21 │*  *└────┴─────┴────┘*  *┌─id─┬─name─┬─age─┐*  *│ 5 │ 田七 │ 22 │*  *│ 6 │ 赵八 │ 23 │*  *└────┴─────┴────┘*  *6 rows in set. Elapsed: 0.008 sec.* |

注意：t\_merge表不会在对应的库路径下产生对应的目录结构。

#### Distributed

Distributed是ClickHouse中分布式引擎，之前所有的操作虽然说是在ClickHouse集群中进行的，但是实际上是在node1节点中单独操作的，与node2、node3无关，使用分布式引擎声明的表才可以在其他节点访问与操作。

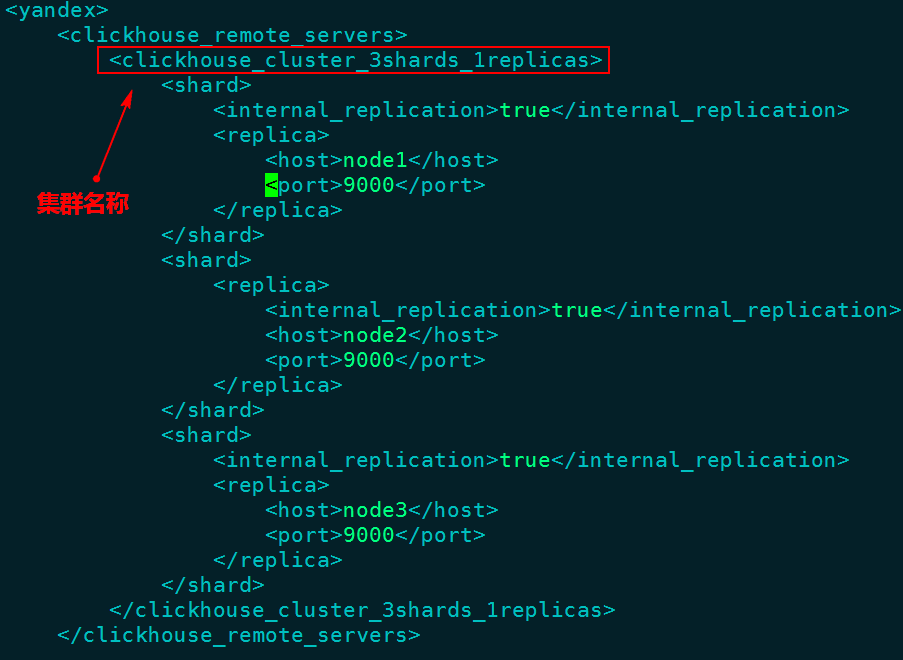
Distributed引擎和Merge引擎类似，本身不存放数据,功能是在不同的server上把多张相同结构的物理表合并为一张逻辑表。

分布式引擎语法：

|  |
| --- |
| Distributed(cluster\_name, database\_name, table\_name[, sharding\_key]) |

对以上语法解释：

* cluster\_name：集群名称，与集群配置中的自定义名称相对应。配置在/etc/metrika.xml文件中，如下图：



* database\_name：数据库名称。
* table\_name：表名称。
* sharding\_key：可选的，用于分片的key值，在数据写入的过程中，分布式表会依据分片key的规则，将数据分布到各个节点的本地表。

注意：创建分布式表是读时检查的机制，也就是说对创建分布式表和本地表的顺序并没有强制要求。

* 示例：

|  |
| --- |
| #在node1、node2、node3节点上启动ClickHouse 服务  [root@node1 ~]# **service clickhouse-server start**  [root@node2 ~]# **service clickhouse-server start**  [root@node3 ~]# **service clickhouse-server start**  #使用默认的default库，在每个节点上创建表 test\_table  node1 :) **create table test\_local (id UInt8,name String) engine= TinyLog**  node2 :) **create table test\_local (id UInt8,name String) engine= TinyLog**  node3 :) **create table test\_local (id UInt8,name String) engine= TinyLog**  #在node1上创建分布式表 t\_distributed，表引擎使用 Distributed 引擎  node1 :) **create table t\_distributed(id UInt8,name String) engine = Distributed(clickhouse\_cluster\_3shards\_1replicas,default,test\_local,id);**  **注意：以上分布式表 t\_distributed 只存在与node1节点的clickhouse中。**  #分别在node1、node2、node3节点上向表test\_local中插入2条数据  node1 :) **insert into test\_local values (1,'张三'),(2,'李四');**  node2 :) **insert into test\_local values (3,'王五'),(4,'马六');**  node3 :) **insert into test\_local values (5,'田七'),(6,'赵八');**  #查询分布式表 t\_distributed 中的数据  node1 :) **select \* from t\_distributed;**  *SELECT \**  *FROM t\_distributed*  *┌─id─┬─name─┐*  *│ 1 │ 张三 │*  *│ 2 │ 李四 │*  *└──┴────┘*  *┌─id─┬─name─┐*  *│ 5 │ 田七 │*  *│ 6 │ 赵八 │*  *└──┴────┘*  *┌─id─┬─name─┐*  *│ 3 │ 王五 │*  *│ 4 │ 马六 │*  *└──┴────┘*  *6 rows in set. Elapsed: 0.010 sec.*  #向分布式表 t\_distributed 中插入一些数据，然后查询 node1、node2、node3节点上的test\_local数据，发现数据已经分布式存储在不同节点上  node1 :) **insert into t\_distributed values (7,'zs'),(8,'ls'),(9,'ww'),(10,'ml'),(11,'tq'),(12,'zb');**  #node1查询本地表 test\_local  node1 :) **select \* from test\_local;**  *SELECT \**  *FROM test\_local*  *┌─id─┬─name─┐*  *│ 1 │ 张三 │*  *│ 2 │ 李四 │*  *│ 9 │ ww │*  *│ 12 │ zb │*  *└───┴────┘*  *4 rows in set. Elapsed: 0.004 sec.*  #node2查询本地表 test\_local  node2 :) **select \* from test\_local;**  *SELECT \**  *FROM test\_local*  *┌─id─┬─name─┐*  *│ 3 │ 王五 │*  *│ 4 │ 马六 │*  *│ 7 │ zs │*  *│ 10 │ ml │*  *└───┴────┘*  *4 rows in set. Elapsed: 0.004 sec.*  #node3查询本地表 test\_local  node3 :) **select \* from test\_local;**  *SELECT \**  *FROM test\_local*  *┌─id─┬─name─┐*  *│ 5 │ 田七 │*  *│ 6 │ 赵八 │*  *│ 8 │ ls │*  *│ 11 │ tq │*  *└───┴────┘*  *4 rows in set. Elapsed: 0.005 sec.* |

以上在node1节点上创建的分布式表t\_distributed 虽然数据是分布式存储在每个clickhouse节点上的，但是只能在node1上查询t\_distributed 表，其他clickhouse节点查询不到此分布式表。如果想要在每台clickhouse节点上都能访问分布式表我们可以指定集群，创建分布式表：

|  |
| --- |
| #创建分布式表 t\_cluster ,引擎使用Distributed 引擎  node1 :) **create table t\_cluster on cluster clickhouse\_cluster\_3shards\_1replicas (id UInt8,name String) engine = Distributed(clickhouse\_cluster\_3shards\_1replicas,default,test\_local,id);**  *CREATE TABLE t\_cluster ON CLUSTER clickhouse\_cluster\_3shards\_1replicas*  *(*  *`id` UInt8,*  *`name` String*  *)*  *ENGINE = Distributed(clickhouse\_cluster\_3shards\_1replicas, default, test\_local, id)*  *┌─host──┬─port─┬─status─┬─error─┬─num\_hosts\_remaining─┬─num\_hosts\_active─┐*  *│ node3 │ 9000 │ 0 │ │ 2 │ 0 │*  *│ node2 │ 9000 │ 0 │ │ 1 │ 0 │*  *│ node1 │ 9000 │ 0 │ │ 0 │ 0 │*  *└───────┴──────┴────────┴───────┴─────────────────────┴──────────────────┘*  *3 rows in set. Elapsed: 0.152 sec.* |

上面的语句中使用了ON CLUSTER分布式DDL（数据库定义语言），这意味着在集群的每个分片节点上，都会创建一张Distributed表，这样便可以从其中任意一端发起对所有分片的读、写请求。

### **MergeTree系列表引擎**

在所有的表引擎中，最为核心的当属MergeTree系列表引擎，这些表引擎拥有最为强大的性能和最广泛的使用场合。对于非MergeTree系列的其他引擎而言，主要用于特殊用途，场景相对有限。而MergeTree系列表引擎是官方主推的存储引擎，有主键索引、数据分区、数据副本、数据采样、删除和修改等功能，支持几乎所有ClickHouse核心功能。

MergeTree系列表引擎包含：MergeTree、ReplacingMergeTree、SummingMergeTree（汇总求和功能）、AggregatingMergeTree（聚合功能）、CollapsingMergeTree（折叠删除功能）、VersionedCollapsingMergeTree（版本折叠功能）引擎，在这些的基础上还可以叠加Replicated和Distributed。

#### MergeTree

MergeTree在写入一批数据时，数据总会以数据片段的形式写入磁盘，且数据片段在磁盘上不可修改。为了避免片段过多，ClickHouse会通过后台线程，定期合并这些数据片段，属于相同分区的数据片段会被合成一个新的片段。这种数据片段往复合并的特点，也正是合并树名称的由来。

**MergeTree作为家族系列最基础的表引擎**，主要有以下特点：

* 存储的数据按照主键排序：创建稀疏索引加快数据查询速度。
* 支持数据分区，可以通过PARTITION BY语句指定分区字段。
* 支持数据副本。
* 支持数据采样。
* MergeTree建表语句：

|  |
| --- |
| 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, ...]** |

* 关于以上建表语句的解释如下：
* ENGINE：ENGINE = MergeTree()，MergeTree引擎没有参数。
* ORDER BY：排序字段。比如ORDER BY (Col1, Col2)，值得注意的是，如果没有使用 PRIMARY KEY 显式的指定主键ORDER BY排序字段自动作为主键。如果不需要排序，则可以使用 ORDER BY tuple() 语法，这样的话，创建的表也就不包含主键。这种情况下，ClickHouse会按照插入的顺序存储数据。**必选项。**
* PARTITION BY：分区字段，例如要按月分区，可以使用表达式 toYYYYMM(date\_column)，这里的date\_column是一个Date类型的列，分区名的格式会是"YYYYMM"。**可选。**
* PRIMARY KEY：指定主键，如果排序字段与主键不一致，可以单独指定主键字段。否则默认主键是排序字段。大部分情况下不需要再专门指定一个 PRIMARY KEY 子句，注意，**在MergeTree中主键并不用于去重，而是用于索引，加快查询速度。可选。**

另外，**如果指定了PRIMARY KEY与排序字段不一致，要保证PRIMARY KEY 指定的主键是ORDER BY 指定字段的前缀**，比如：

|  |
| --- |
| --允许  ... ...  ORDER BY (A,B,C)  PRIMARY KEY A  --报错  ... ...  ORDER BY (A,B,C)  PRIMARY KEY B  *DB::Exception: Primary key must be a prefix of the sorting key* |

这种强制约束保障了即便在两者定义不同的情况下，主键仍然是排序键的前缀，不会出现索引与数据顺序混乱的问题。

* SAMPLE BY：采样字段，如果指定了该字段，那么主键中也必须包含该字段。比如 SAMPLE BY intHash32(UserID) ORDER BY (CounterID, EventDate, intHash32(UserID))。**可选。**
* TTL：数据的存活时间。在MergeTree中，可以为某个列字段或整张表设置TTL。当时间到达时，如果是列字段级别的TTL，则会删除这一列的数据；如果是表级别的TTL，则会删除整张表的数据。**可选。**
* SETTINGS：额外的参数配置。**可选。**

示例：

|  |
| --- |
| #在库 newdb中创建表 t\_mt ,使用MergeTree引擎  node1 :) **create table t\_mt(**  :-] **id UInt8,**  :-]  **name String,**  :-] **age UInt8,**  :-] **birthday Date,**  :-] **location String**  :-] **) engine = MergeTree()**  :-] **order by (id,age)**  :-] **partition by toYYYYMM(birthday);**  #向表中插入数据  node1 :) **insert into t\_mt values (1,'张三',18,'2021-06-01','上海'),**  :-] **(2,'李四',19,'2021-02-10','北京'),**  :-] **(3,'王五',12,'2021-06-01','天津'),**  :-] **(1,'马六',10,'2021-06-18','上海'),**  :-] **(5,'田七',22,'2021-02-09','广州');**  #查看表中的数据  node1 :) **select \* from t\_mt;**  *SELECT \**  *FROM t\_mt*  *┌─id─┬─name─┬─age─┬───birthday─┬─location─┐*  *│ 1 │ 马六 │ 10 │ 2021-06-18 │ 上海 │*  *│ 1 │ 张三 │ 18 │ 2021-06-01 │ 上海 │*  *│ 3 │ 王五 │ 12 │ 2021-06-01 │ 天津 │*  *└───┴──────┴─────┴──────────┴─────────┘*  *┌─id─┬─name─┬─age─┬───birthday─┬─location─┐*  *│ 2 │ 李四 │ 19 │ 2021-02-10 │ 北京 │*  *│ 5 │ 田七 │ 22 │ 2021-02-09 │ 广州 │*  *└────┴─────┴─────┴───────────┴────────┘*  *5 rows in set. Elapsed: 0.006 sec.* |

继续向表 t\_mt中插入以下数据条目：

|  |
| --- |
| #向表t\_mt中继续插入以下数据  node1 :) **insert into t\_mt values (1,'赵八',11,'2021-06-08','北京'),**  :-] **(2,'李九',19,'2021-02-10','天津'),**  :-] **(3,'郑十',12,'2021-07-01','北京');**  #继续查询表 t\_mt中的数据  node1 :) **select \* from t\_mt;**  *SELECT \**  *FROM t\_mt*  *┌─id─┬─name─┬─age─┬───birthday─┬─location─┐*  *│ 1 │ 马六 │ 10 │ 2021-06-18 │ 上海 │*  *│ 1 │ 张三 │ 18 │ 2021-06-01 │ 上海 │*  *│ 3 │ 王五 │ 12 │ 2021-06-01 │ 天津 │*  *└────┴──────┴─────┴────────────┴──────────┘*  *┌─id─┬─name─┬─age─┬───birthday─┬─location─┐*  *│ 2 │ 李四 │ 19 │ 2021-02-10 │ 北京 │*  *│ 5 │ 田七 │ 22 │ 2021-02-09 │ 广州 │*  *└────┴──────┴─────┴────────────┴──────────┘*  *┌─id─┬─name─┬─age─┬───birthday─┬─location─┐*  *│ 1 │ 赵八 │ 11 │ 2021-06-08 │ 北京 │*  *│ 7 │ 郑十 │ 17 │ 2021-06-01 │ 北京 │*  *└────┴──────┴─────┴────────────┴──────────┘*  *┌─id─┬─name─┬─age─┬───birthday─┬─location─┐*  *│ 6 │ 李九 │ 19 │ 2021-02-10 │ 天津 │*  *└────┴──────┴─────┴────────────┴──────────┘*  *8 rows in set. Elapsed: 0.008 sec.* |

可以看到新插入的数据新生成了数据块，实际上这里在底层对应新的分区文件片段，那么为什么新插入的数据没有根据日期和之前的数据放入同一个分区文件呢？MergeTree引擎会在插入数据15分钟左右，将同一个分区的各个分区文件片段合并成一整个分区文件。这里也可以手动执行OPTIMIZE 语句手动触发合并。

* 示例：

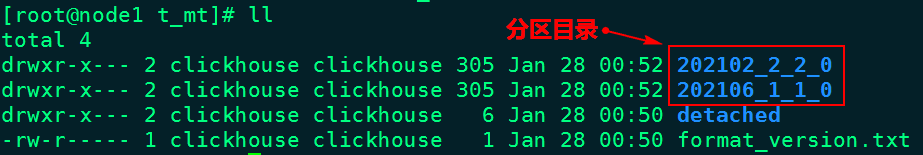
|  |
| --- |
| #手动触发表t\_mt 的分区合并  node1 :) **optimize table t\_mt partition '202102';**  node1 :) **optimize table t\_mt partition '202106';**  **注意：以上optimize 操作，也可以直接写 optimize table t\_mt, 每次执行合并一个分区，如果有多个分区需要执行多次。如果想一次合并所有分区，也可以写成 optimize table t\_mt final;**  #查看表 t\_mt表中的数据，按照相同的分区进行了合并。  node1 :) **select \* from t\_mt;**  *SELECT \**  *FROM t\_mt*  *┌─id─┬─name─┬─age─┬───birthday─┬─location─┐*  *│ 1 │ 马六 │ 10 │ 2021-06-18 │ 上海 │*  *│ 1 │ 赵八 │ 11 │ 2021-06-08 │ 北京 │*  *│ 1 │ 张三 │ 18 │ 2021-06-01 │ 上海 │*  *│ 3 │ 王五 │ 12 │ 2021-06-01 │ 天津 │*  *│ 7 │ 郑十 │ 17 │ 2021-06-01 │ 北京 │*  *└────┴──────┴─────┴────────────┴──────────┘*  *┌─id─┬─name─┬─age─┬───birthday─┬─location─┐*  *│ 2 │ 李四 │ 19 │ 2021-02-10 │ 北京 │*  *│ 5 │ 田七 │ 22 │ 2021-02-09 │ 广州 │*  *│ 6 │ 李九 │ 19 │ 2021-02-10 │ 天津 │*  *└────┴──────┴─────┴────────────┴──────────┘*  *8 rows in set. Elapsed: 0.004 sec.*  **注意：MergeTree引擎表中主键并不用于去重，而是用于索引，加快查询速度。** |

##### MergeTree引擎表目录解析

下面我们介绍下MergeTree引擎表 t\_mt对应到磁盘的数据目录，为了方便从零开始了解，这里我们删除t\_mt表，重新创建t\_mt表，并插入数据，执行命令如下：

|  |
| --- |
| #删除表 t\_mt,重新创建表t\_mt,并加载数据  node1 :) **drop table t\_mt;**  node1 :)**create table t\_mt( id UInt8, name String, age UInt8, birthday Date, location String ) engine = MergeTree() order by (id,age) partition by toYYYYMM(birthday);**  node1 :) **insert into t\_mt values (1,'张三',18,'2021-06-01','上海'), (2,'李四',19,'2021-02-10','北京'), (3,'王五',12,'2021-06-01','天津'), (1,'马六',10,'2021-06-18','上海'), (5,'田七',22,'2021-02-09','广州');** |

以上创建好表t\_mt，当插入数据完成后，在clickhouse节点/var/lib/clickhouse/data/newdb/路径下会生成对应目录“t\_mt”,进入此目录下，可以看到对应的分区目录，如图示：



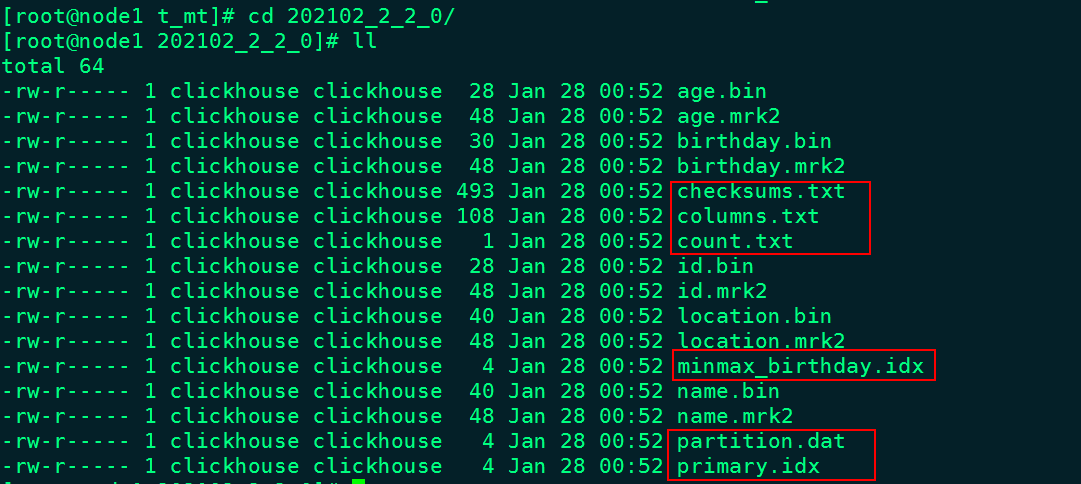
以上分区目录也可以在系统表“system.parts”中查询得到：

|  |
| --- |
| #在系统表 system.part中查询表 t\_mt的分区信息：  node1 :) **select table ,partition ,name ,active from system.parts where table = 't\_mt';** |

以上表各列的解释如下：

* table代表当前表。
* partition是当前表的分区名称。
* name是对应到磁盘上数据所在的分区目录片段。例如“202102\_2\_2\_0”中“202102”是分区名称，“2”是数据块的最小编号，“2”是数据块的最大编号，“0”代表该块在MergeTree中第几次合并得到。
* active代表当前分区片段的状态：1代表激活状态，0代表非激活状态，非激活片段是那些在合并到较大片段之后剩余的源数据片段，损坏的数据片段也表示为非活动状态。非激活片段会在合并后的10分钟左右被删除。

进入到某一个分区目录片段“202102\_2\_2\_0”中，我们可以看到如下目录：



对以上目录的解释如下：

* **checksums.txt**：校验文件，使用二进制格式存储。它保存了余下各类文件(primary. idx、count.txt等)的size大小及size的哈希值，用于快速校验文件的完整性和正确性。
* **columns.txt**： 存储当前分区所有列信息。使用明文格式存储。

|  |
| --- |
| [root@node1 202102\_2\_2\_0]# cat columns.txt  columns format version: 1  5 columns:  `id` UInt8  `name` String  `age` UInt8  `birthday` Date  `location` String |

* **count.txt**：计数文件，使用明文格式存储。用于记录当前数据分区目录下数据的总行数。

|  |
| --- |
| [root@node1 202102\_2\_2\_0]# cat count.txt  2 |

* **primary.idx**：一级索引文件，使用二进制格式存储。用于存放稀疏索引，一张MergeTree表只能声明一次一级索引，即通过ORDER BY或者PRIMARY KEY指定字段。借助稀疏索引，在数据查询的时能够排除主键条件范围之外的数据文件，从而有效减少数据扫描范围，加速查询速度。
* **列.bin**：数据文件，使用压缩格式存储，默认为LZ4压缩格式，用于存储某一列的数据。由于MergeTree采用列式存储，所以每一个列字段都拥有独立的.bin数据文件，并以列字段名称命名。
* **列.mrk2**：列字段标记文件，使用二进制格式存储。标记文件中保存了.bin文件中数据的偏移量信息
* **partition.dat与minmax\_[Column].idx**：如果指定了分区键，则会额外生成partition.dat与minmax索引文件，它们均使用二进制格式存储。partition.dat用于保存当前分区下分区表达式最终生成的值，即分区字段值；而minmax索引用于记录当前分区下分区字段对应原始数据的最小和最大值。比如当使用birthday字段对应的原始数据为2021-02-17、2021-02-23，分区表达式为PARTITION BY toYYYYMM(birthday)，即按月分区。partition.dat中保存的值将会是202102，而minmax索引中保存的值将会是2021-02-17、2021-02-23。

ClickHouse MergeTree引擎表支持分区，索引，修改，并发查询数据，**当查询MergeTree表数据时，首先向primary.idx文件中获取对应的索引，根据索引找到【列.mrk2】文件获取对应的数据块偏移量，然后再根据偏移量从【列.bin】文件中读取块数据。**

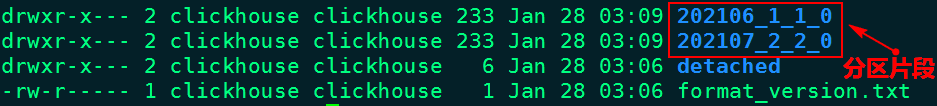
##### MergeTree引擎表设置分区

给表设置分区可以在查询过程中跳过不需要的数据目录，提升查询效率。在ClickHouse中并不是所有的表都支持分区，目前只有MergeTree家族系列的表引擎才支持数据分区。

通过前面的学习，我们知道向MergeTree分区表中每次插入数据时，每次都会生成对应的分区片段，不会立刻合并相同分区的数据，需要等待15分钟左右，ClickHouse会自动合并相同的分区片段，并删除合并之前的源数据片段，当然这里我们也可以手动执行OPTIMIZE 语句手动触发合并分区表中的分区片段。通过下面案例来学习分区表中分区片段合并的规则。

|  |
| --- |
| #创建表 login\_info ,设置MergeTree引擎  node1 :) **create table login\_info(**  :-]  **id UInt8,**  :-] **name String,**  :-] **log\_time Date**  :-] ) **engine = MergeTree()**  :-] **order by (id)**  :-] **partition by toYYYYMM(log\_time);**  #向表 login\_info中插入以下数据  node1 :) **insert into login\_info values (1,'zs','2021-06-01'),**  :-] **(2,'ls','2021-06-01'),**  :-] **(3,'ww','2021-07-01'),**  :-] **(4,'ml','2021-07-01');**  #查看表 login\_info 中数据  node1 :) **select \* from login\_info;**  *SELECT \**  *FROM login\_info*  *┌─id─┬─name─┬───log\_time─┐*  *│ 3 │ ww │ 2021-07-01 │*  *│ 4 │ ml │ 2021-07-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 1 │ zs │ 2021-06-01 │*  *│ 2 │ ls │ 2021-06-01 │*  *└────┴──────┴────────────┘*  *4 rows in set. Elapsed: 0.008 sec.* |

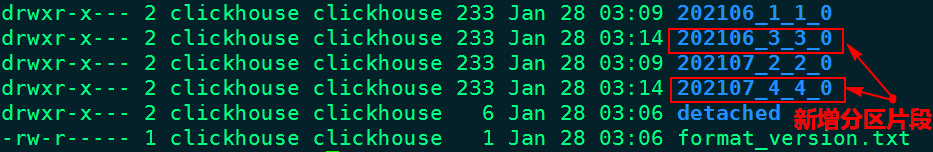
经过以上步骤，在clickhouse节点上查看表login\_info数据目录/var/lib/clickhouse/data/newdb/login\_info，如下图示：



继续向表 login\_info中插入以下数据：

|  |
| --- |
| #继续向表login\_info中插入以下数据  node1 :) **insert into login\_info values (5,'zs1','2021-06-01'),**  :-] **(6,'ls1','2021-06-01'),**  :-] **(7,'ww1','2021-07-01'),**  :-] **(8,'ml1','2021-07-01');**  #查看表 login\_info数据  node1 :) **select \* from login\_info;**  *SELECT \**  *FROM login\_info*  *┌─id─┬─name─┬───log\_time─┐*  *│ 3 │ ww │ 2021-07-01 │*  *│ 4 │ ml │ 2021-07-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 1 │ zs │ 2021-06-01 │*  *│ 2 │ ls │ 2021-06-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 5 │ zs1 │ 2021-06-01 │*  *│ 6 │ ls1 │ 2021-06-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 7 │ ww1 │ 2021-07-01 │*  *│ 8 │ ml1 │ 2021-07-01 │*  *└────┴──────┴────────────┘*  *8 rows in set. Elapsed: 0.006 sec.* |

通过插入数据之后再次查询发现，相同分区的数据展示在不同的数据块中。在clickhouse节点上再次查看表login\_info数据目录/var/lib/clickhouse/data/newdb/login\_info，如下图示：

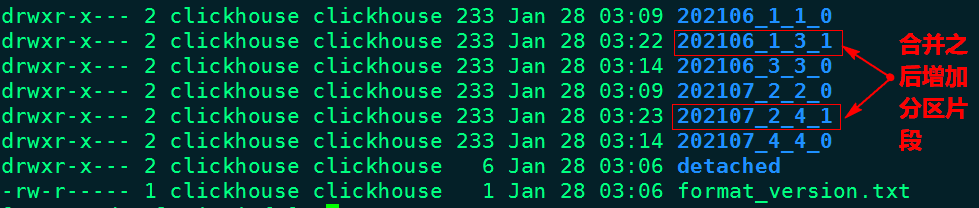


“202106\_3\_3\_0”为例，“202006”为分区，“3”代表数据块的最小编号，“3”代表数据块的最大编号，“0”代表合并的第几次（合并树中块的级别）。

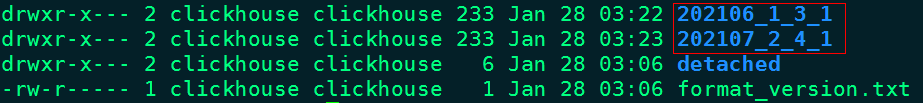
手动执行OPTIMIZE 语句手动触发合并分区表中的分区片段:

|  |
| --- |
| #执行如下命令，手动合并分区片段  node1 :) **optimize table login\_info partition '202106' ;**  node1 :) **optimize table login\_info partition '202107' ;**  #查看表 login\_info中的数据：  node1 :) **select \* from login\_info;**  *SELECT \**  *FROM login\_info*  *┌─id─┬─name─┬───log\_time─┐*  *│ 3 │ ww │ 2021-07-01 │*  *│ 4 │ ml │ 2021-07-01 │*  *│ 7 │ ww1 │ 2021-07-01 │*  *│ 8 │ ml1 │ 2021-07-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 1 │ zs │ 2021-06-01 │*  *│ 2 │ ls │ 2021-06-01 │*  *│ 5 │ zs1 │ 2021-06-01 │*  *│ 6 │ ls1 │ 2021-06-01 │*  *└────┴──────┴────────────┘*  *8 rows in set. Elapsed: 0.006 sec.* |

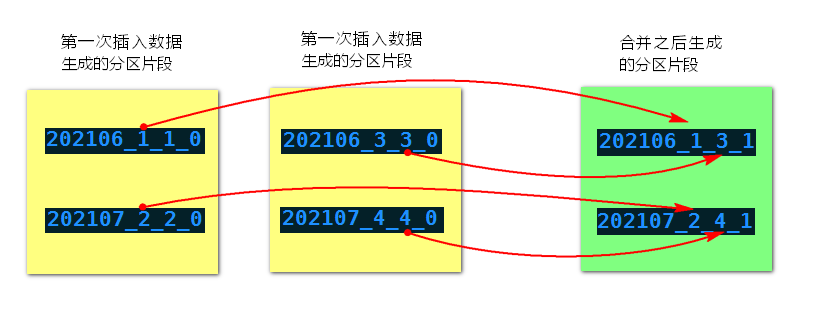
通过合并分区片段之后，在clickhouse节点上再次查看表login\_info数据目录/var/lib/clickhouse/data/newdb/login\_info，如下图示：



经过一段时间再次查询当前目录数据，只剩余合并最后的两个分区片段，如下图所示：



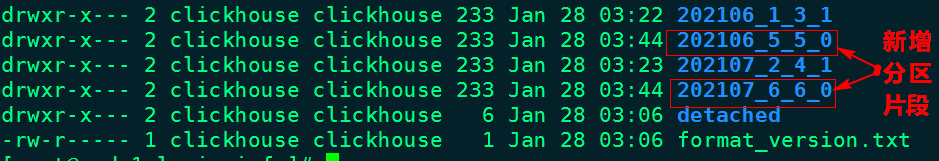
MergeTree分区表合并分区规则如下：获取相同分区片段中最小编号和最大编号，组合成新的分区片段，同时修改合并的次数（合并树中块的级别），合并示意图如下：



继续向表login\_info中插入数据：

|  |
| --- |
| #继续向表login\_info中插入以下数据  node1 :) **insert into login\_info values (9,'zs1','2021-06-01'),**  :-] **(10,'ls1','2021-06-01'),**  :-] **(11,'ww1','2021-07-01'),**  :-] **(12,'ml1','2021-07-01');**  #查看表 login\_info数据  node1 :) **select \* from login\_info;**  *SELECT \**  *FROM login\_info*  *┌─id─┬─name─┬───log\_time─┐*  *│ 3 │ ww │ 2021-07-01 │*  *│ 4 │ ml │ 2021-07-01 │*  *│ 7 │ ww1 │ 2021-07-01 │*  *│ 8 │ ml1 │ 2021-07-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 11 │ ww1 │ 2021-07-01 │*  *│ 12 │ ml1 │ 2021-07-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 1 │ zs │ 2021-06-01 │*  *│ 2 │ ls │ 2021-06-01 │*  *│ 5 │ zs1 │ 2021-06-01 │*  *│ 6 │ ls1 │ 2021-06-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 9 │ zs1 │ 2021-06-01 │*  *│ 10 │ ls1 │ 2021-06-01 │*  *└────┴──────┴────────────┘*  *12 rows in set. Elapsed: 0.006 sec* |

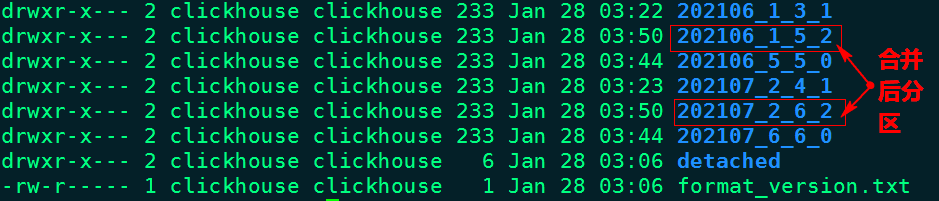
在clickhouse节点上再次查看表login\_info数据目录/var/lib/clickhouse/data/newdb/login\_info，如下图示：



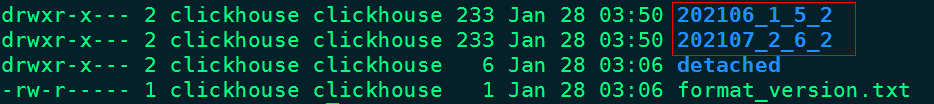
再次执行合并分区命令，合并表login\_info分区片段：

|  |
| --- |
| #执行如下命令，手动合并分区片段  node1 :) **optimize table login\_info partition '202106' ;**  node1 :) **optimize table login\_info partition '202107' ;**  #查看表 login\_info中的数据：  node1 :) **select \* from login\_info;**  *SELECT \**  *FROM login\_info*  *┌─id─┬─name─┬───log\_time─┐*  *│ 3 │ ww │ 2021-07-01 │*  *│ 4 │ ml │ 2021-07-01 │*  *│ 7 │ ww1 │ 2021-07-01 │*  *│ 8 │ ml1 │ 2021-07-01 │*  *│ 11 │ ww1 │ 2021-07-01 │*  *│ 12 │ ml1 │ 2021-07-01 │*  *└────┴──────┴────────────┘*  *┌─id─┬─name─┬───log\_time─┐*  *│ 1 │ zs │ 2021-06-01 │*  *│ 2 │ ls │ 2021-06-01 │*  *│ 5 │ zs1 │ 2021-06-01 │*  *│ 6 │ ls1 │ 2021-06-01 │*  *│ 9 │ zs1 │ 2021-06-01 │*  *│ 10 │ ls1 │ 2021-06-01 │*  *└────┴──────┴────────────┘*  *12 rows in set. Elapsed: 0.008 sec.* |

通过合并分区片段之后，在clickhouse节点上再次查看表login\_info数据目录/var/lib/clickhouse/data/newdb/login\_info，如下图示：



经过一段时间再次查询当前目录数据，只剩余合并最后的两个分区片段，如下图所示：



此外，表设置分区字段时，分区健不仅可以指定成时间列，也可以是表中任意列或者列的表达式。下面案例使用表中的地区列当做分区：

|  |
| --- |
| #创建表 emp\_info ，使用MergeTree分区  node1 :) **create table emp\_info (**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **loc String,**  :-] **salary Decimal32(2)**  :-] **)engine = MergeTree()**  :-] **order by id**  :-] **partition by loc;**  # 向表中插入以下数据  node1 :) **insert into emp\_info values (1,'张三',18,'上海',10.11),**  :-] **(2,'李四',19,'北京',100.123),**  :-] **(3,'王五',20,'上海',200.2),**  :-] **(4,'马六',21,'上海',300.456),**  :-] **(5,'田七',22,'北京',400.78),**  :-] ;  #查看表中的数据，可以观察到所有数据都是按照地区合并在一起。  node1 :) **select \* from emp\_info;**  *SELECT \**  *FROM emp\_info*  *┌─id─┬─name─┬─age─┬─loc──┬─salary─┐*  *│ 2 │ 李四 │ 19 │ 北京 │ 100.12 │*  *│ 5 │ 田七 │ 22 │ 北京 │ 400.78 │*  *└───┴────┴────┴─────┴──────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─salary─┐*  *│ 1 │ 张三 │ 18 │ 上海 │ 10.11 │*  *│ 3 │ 王五 │ 20 │ 上海 │ 200.20 │*  *│ 4 │ 马六 │ 21 │ 上海 │ 300.45 │*  *└───┴────┴────┴─────┴──────┘*  *5 rows in set. Elapsed: 0.006 sec.* |

**注意：如果按照字符串字段来进行分区，在底层/var/lib/clickhouse/data/newdb/目录下对应的表emp\_info中的分区片段名称是使用字符串的hashcode+编码的形式来命名。**

#### ReplacingMergeTree

以上MergeTree不能对相同主键的数据进行去重，ClickHouse提供了ReplacingMergeTree引擎，可以针对同分区内相同主键的数据进行去重，它能够在**合并分区时删除重复的数据**。值得注意的是，ReplacingMergeTree只是在一定程度上解决了数据重复问题，由于自动分区合并机制在后台定时执行，所以**并不能完全保障数据不重复。**ReplacingMergeTree 适用于在后台清除重复的数据以节省空间。

* ReplaceingMergeTree建表语句：

|  |
| --- |
| CREATE TABLE [IF NOT EXISTS] [db.]table\_name [ON CLUSTER cluster]  (  name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],  name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],  ...  ) ENGINE = ReplacingMergeTree([ver])  [PARTITION BY expr]  [ORDER BY expr]  [SAMPLE BY expr]  [SETTINGS name=value, ...] |

以上建表语句的解释如下：

* [ver] ：可选参数，指定列的版本，可以是UInt\*、Date或者DateTime类型的字段作为版本号。该参数决定了数据去重的方式。当没有指定[ver]时，保留最后插入的数据，也就是最新的数据；如果指定了具体的[ver]列，则保留最大版本数据。

**使用ReplacingMergeTree是需要注意以下几点：**

* **如何判断数据重复**

ReplacingMergeTree在去除重复数据时，是以ORDERBY排序键为基准的，而不是PRIMARY KEY。

* **何时删除重复数据**

在执行分区合并时，会触发删除重复数据。optimize的合并操作是在后台执行的，无法预测具体执行时间点，除非是手动执行。

* **不同分区的重复数据不会被去重**

ReplacingMergeTree是以分区为单位删除重复数据的。只有在相同的数据分区内重复的数据才可以被删除，而不同数据分区之间的重复数据依然不能被剔除。

* **数据去重的策略是什么**

如果没有设置[ver]版本号，则保留同一组重复数据中的最新插入的数据；如果设置了[ver]版本号，则保留同一组重复数据中ver字段取值最大的那一行。

* **optimize命令使用**

一般在数据量比较大的情况，尽量不要使用该命令。因为在海量数据场景下，执行optimize要消耗大量时间。

* 示例：
* **测试去重按照Order by 字段进行去重，而不是按照primary 主键字段进行去重。**

|  |
| --- |
| #创建表 t\_replacing\_mt ,使用ReplacingMergeTree引擎  node1 :) **create table t\_replacing\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **gender String**  :-] **) engine = ReplacingMergeTree()**  :-] **order by (id,age)**  :-] **primary key id**  :-] **partition by gender;**  #向表 t\_replacing\_mt 中插入以下数据：  node1 :) **insert into t\_replacing\_mt values (1,'张三',18,'男'),**  :-] **(2,'李四',19,'女'),**  :-] **(3,'王五',20,'男');**  #查询表 t\_replacing\_mt 中的数据：  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└───┴────┴────┴──────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 18 │ 男 │*  *│ 3 │ 王五 │ 20 │ 男 │*  *└───┴────┴────┴──────┘*  #向表 t\_replacing\_mt 中插入id 为1的一行数据  node1 :) **insert into t\_replacing\_mt values (1,'张三',10,'男');**  #查询表 t\_replacing\_mt 数据：  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 10 │ 男 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 18 │ 男 │*  *│ 3 │ 王五 │ 20 │ 男 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└────┴──────┴─────┴────────┘*  #执行 optimize命令手动合并分区数据  node1 :) **optimize table t\_replacing\_mt;**  #查询表 t\_replacing\_mt 数据，发现没有按照primary key 去重。  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  ***│ 1 │ 张三 │ 10 │ 男 │***  ***│ 1 │ 张三 │ 18 │ 男 │***  *│ 3 │ 王五 │ 20 │ 男 │*  *└────┴──────┴─────┴────────┘*  #再次向表 t\_replacing\_mt 插入数据：  node1 :) **insert into t\_replacing\_mt values (1,'张三三',18,'男');**  #查询表 t\_replacing\_mt 数据  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name───┬─age─┬─gender─┐*  *│ 1 │ 张三三 │ 18 │ 男 │*  *└───┴──────┴────┴──────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└───┴────┴────┴──────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 10 │ 男 │*  *│ 1 │ 张三 │ 18 │ 男 │*  *│ 3 │ 王五 │ 20 │ 男 │*  *└───┴─────┴───┴──────┘*  #再次执行 optimize命令手动合并分区数据  node1 :) **optimize table t\_replacing\_mt;**  #查询表 t\_replacing\_mt 数据  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└───┴────┴────┴──────┘*  *┌─id─┬─name───┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 10 │ 男 │*  ***│ 1 │ 张三三 │ 18 │ 男 │***  *│ 3 │ 王五 │ 20 │ 男 │*  *└───┴──────┴────┴─────┘*  **注意：通过以上测试发现ClickHouse ReplacingMergeTree中去除重复数据时，是以ORDERBY排序键为基准的，而不是PRIMARY KEY。** |

* **测试不指定[ver]列时，插入相同排序字段的数据，保留最新一条数据。**

|  |
| --- |
| #删除表 t\_replacing\_mt 重建,使用ReplacingMergeTree引擎  node1 :) **create table t\_replacing\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **gender String**  :-] **) engine = ReplacingMergeTree()**  :-] **order by id**  :-] **primary key id**  :-] **partition by gender;**  #向表 t\_replacing\_mt 中插入以下数据  node1 :) **insert into t\_replacing\_mt values (1,'张三',18,'男'),**  :-] **(2,'李四',19,'女'),**  :-] **(3,'王五',20,'男');**  #查询表 t\_replacing\_mt 中的数据  node1 :) **select \* from t\_replacing\_mt ;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 18 │ 男 │*  *│ 3 │ 王五 │ 20 │ 男 │*  *└────┴──────┴─────┴────────┘*  #向表 t\_replacing\_mt 中插入排序字段相同的一行数据  node1 :) **insert into t\_replacing\_mt values (1,'张三',10,'男');**  #查询表 t\_replacing\_mt 中的数据  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 10 │ 男 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 18 │ 男 │*  *│ 3 │ 王五 │ 20 │ 男 │*  *└────┴──────┴─────┴────────┘*  #执行 optimize命令手动合并分区数据  node1 :) **optimize table t\_replacing\_mt;**  #查询表 t\_replacing\_mt 中的数据  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  ***│ 1 │ 张三 │ 10 │ 男 │***  *│ 3 │ 王五 │ 20 │ 男 │*  *└────┴──────┴─────┴────────┘*  **注意：通过以上测试可以发现，ClickHouse ReplacingMergeTree中不指定[ver]列时，当插入排序字段相同的数据时，保留最新一条数据。** |

* **测试指定[ver]列时，插入相同排序字段的数据，保留当前[ver]列最大值。**

|  |
| --- |
| #删除表 t\_replacing\_mt 重新创建,使用ReplacingMergeTree引擎，指定[ver]  node1 :) create table t\_replacing\_mt(  :-] id UInt8,  :-] name String,  :-] age UInt8,  :-] gender String  :-] ) engine = **ReplacingMergeTree(age)**  :-] **order by id**  :-] **primary key id**  :-] **partition by gender;**  **#向表** t\_replacing\_mt 中插入数据：  node1 :) **insert into t\_replacing\_mt values (1,'张三',18,'男'),**  :-] **(2,'李四',19,'女'),**  :-] **(3,'王五',20,'男');**  #查询表 t\_replacing\_mt中数据：  node1 :) **select \* from t\_replacing\_mt ;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 18 │ 男 │*  *│ 3 │ 王五 │ 20 │ 男 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└────┴──────┴─────┴────────┘*  #向表 t\_replacing\_mt 中插入排序字段相同的一行数据  node1 :) **insert into t\_replacing\_mt values (1,'张三',10,'男');**  #查看表 t\_replacing\_mt中的数据  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 10 │ 男 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 1 │ 张三 │ 18 │ 男 │*  *│ 3 │ 王五 │ 20 │ 男 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└────┴──────┴─────┴────────┘*  #对表 t\_replacing\_mt中的数据执行手动分区合并  node1 :) **optimize table t\_replacing\_mt;**  #查看表 t\_replacing\_mt中的数据  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name─┬─age─┬─gender─┐*  *│ 2 │ 李四 │ 19 │ 女 │*  *└────┴──────┴─────┴────────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  ***│ 1 │ 张三 │ 18 │ 男 │***  *│ 3 │ 王五 │ 20 │ 男 │*  *└────┴──────┴─────┴────────┘*  **注意：通过以上测试可以发现，在ClickHouse中创建ReplacingMergeTree时，如果指定了[ver]列，当存在Order by字段重复时，会保留ver列最大值对应的行。** |

* **测试不同分区中有相同的Order by 字段时，不去重。**

|  |
| --- |
| #删除表 t\_replacing\_mt ，重新创建  node1 :) **create table t\_replacing\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **gender String**  :-] **) engine = ReplacingMergeTree()**  :-] **order by id**  :-] **primary key id**  :-] **partition by gender;**  #向表 t\_replacing\_mt 中插入以下数据：  node1 :) **insert into t\_replacing\_mt values (1,'张三',18,'男'),**  :-] **(2,'李四',19,'女'),**  :-] **(3,'王五',20,'男');**  #再次向表 t\_replacing\_mt 中插入以下数据：  node1 :) **insert into t\_replacing\_mt values (1,'张三三',10,'女');**  #对表 t\_replacing\_mt中的数据执行手动分区合并  node1 :) **optimize table t\_replacing\_mt;**  #查看表中的数据  node1 :) **select \* from t\_replacing\_mt;**  *┌─id─┬─name───┬─age─┬─gender─┐*  ***│ 1 │ 张三三 │ 10 │ 女 │***  *│ 2 │ 李四 │ 19 │ 女 │*  *└───┴──────┴────┴─────┘*  *┌─id─┬─name─┬─age─┬─gender─┐*  ***│ 1 │ 张三 │ 18 │ 男 │***  *│ 3 │ 王五 │ 20 │ 男 │*  *└───┴────┴────┴─────┘*  **注意：通过以上测试可以发现，在ClickHouse中创建ReplacingMergeTree时，不同分区中相同的Order by 字段不会去重。** |

#### SummingMergeTree

该引擎继承了MergeTree引擎，当合并 SummingMergeTree 表的数据片段时，ClickHouse 会把所有具有相同主键的行合并为一行，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，即如果存在重复的数据，会对对这些重复的数据进行合并成一条数据，类似于group by的效果，可以显著减少存储空间并加快数据查询速度。

如果用户**只需要查询数据的汇总结果，不关心明细数据**，并且数据的汇总条件是预先明确的，即GROUP BY的分组字段是确定的，可以使用该表引擎。

* SummingMergeTree建表语句：

|  |
| --- |
| 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, ...]** |

对以上建表语句的解释如下：

* [columns]: 将要被汇总的列,或者多个列，多个列需要写在元组中。可选参数。所选的列必须是数值类型，并且不可位于主键中。如果没有指定 [columns]，ClickHouse 会把所有不在主键中的数值类型的列都进行汇总。

**使用SummingMergeTree注意以下几点：**

* SummingMergeTree是根据什么对两条数据进行合并的

用ORBER BY排序键作为聚合数据的条件Key。即如果排序key是相同的，则会合并成一条数据，并对指定的合并字段进行聚合。

* 仅对分区内的相同排序key的数据行进行合并

以数据分区为单位来聚合数据。当分区合并时，同一数据分区内聚合Key相同的数据会被合并汇总，而不同分区之间的数据则不会被汇总。

* 如果没有指定聚合字段，会怎么聚合

如果没有指定聚合字段，则会按照非主键的数值类型字段进行聚合。

* 对于非汇总字段的数据，该保留哪一条

如果两行数据除了排序字段相同，其他的非聚合字段不相同，那么在聚合发生时，会保留最初的那条数据，新插入的数据对应的那个字段值会被舍弃。

* 示例：
* 测试不指定聚合字段同时测试不同分区内，相同排序key数据不会被合并

|  |
| --- |
| #创建表 t\_summing\_mt ,使用SummingMergeTree表引擎  node1 :) **create table t\_summing\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **loc String,**  :-] **dept String,**  :-] **workdays UInt8,**  :-] **salary Decimal32(2)**  :-] **) engine = SummingMergeTree()**  :-] **order by (id,age)**  :-] **primary key id**  :-] **partition by loc;**  #向表 t\_summing\_mt 中插入以下数据  node1 :) **insert into t\_summing\_mt values (1,'张三',18,'北京','大数据',24,10000),**  :-] **(2,'李四',19,'上海','java',22,8000),**  :-] **(3,'王五',20,'北京','java',26,12000);**  #查看表 t\_summing\_mt 中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 24 │ 10000.00 │*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  #向表 t\_summing\_mt 中插入一条排序键相同的数据  node1 :) **insert into t\_summing\_mt values (1,'马六',18,'北京','前端',27,15000);**  #查看表 t\_summing\_mt中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 24 │ 10000.00 │*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬───salary─┐*  ***│ 1 │ 马六 │ 18 │ 北京 │ 前端 │ 27 │ 15000.00 │***  *└────┴──────┴─────┴──────┴──────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  #手动执行optimize 命令触发合并相同分区数据  node1 :) **optimize table t\_summing\_mt;**  #查看表 t\_summing\_mt 中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  ***│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 51 │ 25000.00 │***  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  **注意：我们可以看到当不指定 聚合字段时，有相同排序字段行进行聚合时，会将数值类型的字段进行聚合合并。**  #继续向表 t\_summing\_mt中插入以下数据：  node1 :) **insert into t\_summing\_mt values (1,'张三',18,'南京','java',18,12000);**  #查看表 t\_summing\_mt中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬───salary─┐*  *│ 1 │ 张三 │ 18 │ 南京 │ java │ 18 │ 12000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 51 │ 25000.00 │*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  #手动指定optimize 命令合并相同排序key的数据  node1 :) **optimize table t\_summing\_mt;**  #查看表 t\_summing\_mt中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬───salary─┐*  ***│ 1 │ 张三 │ 18 │ 南京 │ java │ 18 │ 12000.00 │***  *└────┴──────┴─────┴──────┴──────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  ***│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 51 │ 25000.00 │***  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  **注意：不同分区内相同的排序key的数据不能被合并** |

* 测试指定一个聚合字段

|  |
| --- |
| #删除表 t\_summing\_mt，重新创建表 t\_summing\_mt ，使用SummingMergeTree引擎  node1 :) **create table t\_summing\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **loc String,**  :-] **dept String,**  :-] **workdays UInt8,**  :-] **salary Decimal32(2)**  :-] **) engine = SummingMergeTree(salary)**  :-] **order by (id,age)**  :-] **primary key id**  :-] **partition by loc;**  #向表 t\_summing\_mt 中插入以下数据  node1 :) **insert into t\_summing\_mt values (1,'张三',18,'北京','大数据',24,10000),**  :-] **(2,'李四',19,'上海','java',22,8000),**  :-] **(3,'王五',20,'北京','java',26,12000);**  #查看表 t\_summing\_mt 中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 24 │ 10000.00 │*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  #向表 t\_summing\_mt 中插入一条排序键相同的数据  node1 :) **insert into t\_summing\_mt values (1,'马六',18,'北京','前端',27,15000);**  #查看表 t\_summing\_mt中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 24 │ 10000.00 │*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬───salary─┐*  ***│ 1 │ 马六 │ 18 │ 北京 │ 前端 │ 27 │ 15000.00 │***  *└────┴──────┴─────┴──────┴──────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  #手动执行optimize 命令触发合并相同分区数据  node1 :) **optimize table t\_summing\_mt;**  #查看表 t\_summing\_mt 中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  ***│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 24 │ 25000.00 │***  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  **注意：我们可以看到当指定一个聚合字段时，有相同排序字段行进行聚合时，会按照这个数值字段进行合并，其他的保留最开始一条数据的信息。** |

* 测试指定多个聚合字段

|  |
| --- |
| #删除表 t\_summing\_mt，重新创建表 t\_summing\_mt ，使用SummingMergeTree引擎  node1 :) **create table t\_summing\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **loc String,**  :-] **dept String,**  :-] **workdays UInt8,**  :-] **salary Decimal32(2)**  :-] **) engine = SummingMergeTree((salary,workdays))**  :-] **order by (id,age)**  :-] **primary key id**  :-] **partition by loc;**  #向表 t\_summing\_mt 中插入以下数据  node1 :) **insert into t\_summing\_mt values (1,'张三',18,'北京','大数据',24,10000),**  :-] **(2,'李四',19,'上海','java',22,8000),**  :-] **(3,'王五',20,'北京','java',26,12000);**  #查看表 t\_summing\_mt 中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 24 │ 10000.00 │*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  #向表 t\_summing\_mt 中插入一条排序键相同的数据  node1 :) **insert into t\_summing\_mt values (1,'马六',18,'北京','前端',27,15000);**  #查看表 t\_summing\_mt中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 24 │ 10000.00 │*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬───salary─┐*  ***│ 1 │ 马六 │ 18 │ 北京 │ 前端 │ 27 │ 15000.00 │***  *└────┴──────┴─────┴──────┴──────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  #手动执行optimize 命令触发合并相同分区数据  node1 :) **optimize table t\_summing\_mt;**  #查看表 t\_summing\_mt 中的数据  node1 :) **select \* from t\_summing\_mt;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept───┬─workdays─┬───salary─┐*  ***│ 1 │ 张三 │ 18 │ 北京 │ 大数据 │ 51 │ 25000.00 │***  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ 12000.00 │*  *└────┴──────┴─────┴──────┴────────┴──────────┴──────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬──salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴─────────┘*  **注意：我们可以看到当指定多个聚合字段时，有相同排序字段行进行聚合时，会按照指定的多个数值字段进行合并，其他的保留最开始一条数据的信息。** |

#### AggregatingMergeTree

该表引擎继承自MergeTree，可以使用 AggregatingMergeTree 表来做增量数据统计聚合。如果要按一组规则来合并减少行数，则使用 AggregatingMergeTree 是合适的。AggregatingMergeTree是通过预先定义的聚合函数计算数据并通过二进制的格式存入表内。

与SummingMergeTree的区别在于：SummingMergeTree对非主键列进行sum聚合，而AggregatingMergeTree则可以指定各种聚合函数。对某些字段需要进行聚合时，需要在创建表字段时指定成AggregateFunction类型。

* AggregatingMergeTree建表语句如下：

|  |
| --- |
| CREATE TABLE [IF NOT EXISTS] [db.]table\_name [ON CLUSTER cluster]  (  name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],  name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],  ...  ) ENGINE = **AggregatingMergeTree()**  **[PARTITION BY expr]**  **[ORDER BY expr]**  **[SAMPLE BY expr]**  **[TTL expr]**  **[SETTINGS name=value, ...]** |

* 示例：

|  |
| --- |
| #创建表 t\_aggregating\_mt ，使用AggregatingMergeTree引擎，指定salary字段是聚合字段  node1 :) **create table t\_aggregating\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **loc String,**  :-] **dept String,**  :-] **workdays UInt8,**  :-] **salary AggregateFunction(sum,Decimal32(2))**  :-] **) engine = AggregatingMergeTree()**  :-] **order by (id,age)**  :-] **primary key id**  :-] **partition by loc;** |

* 注意：

对于AggregateFunction类型的列字段，在进行数据的写入和查询时与其他的表引擎有很大区别，在写入数据时，需要调用 \*-State 函数；而在查询数据时，则需要调用相应的 \*-Merge 函数。

对于上面的建表语句而言，需要使用sumState函数进行数据插入。

|  |
| --- |
| #向表 t\_aggregating\_mt 中插入数据，插入方式与之前方式不同  node1 :) **insert into t\_aggregating\_mt select 1,'张三',18,'北京','java',18,sumState(toDecimal32(10000,2));**  node1 :) **insert into t\_aggregating\_mt select 2,'李四',19,'上海','java',22,sumState(toDecimal32(8000,2));**  node1 :) **insert into t\_aggregating\_mt select 3,'王五',20,'北京','java',26,sumState(toDecimal32(12000,2));** |

查询数据时，如果正常语句查询，aggregateFunction类型的列不会正常显示数据，针对以上的数据需要使用sumMerge来展示数据。

|  |
| --- |
| #错误方式查询表 t\_aggregating\_mt 中的数据  node1 :) select \* from t\_aggregating\_mt;  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ java │ 18* ***│ @B │***  *└────┴──────┴─────┴──────┴──────┴──────────┴────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22* ***│ 5 │***  *└────┴──────┴─────┴──────┴──────┴──────────┴────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┐*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26* ***│ O │***  *└────┴──────┴─────┴──────┴──────┴──────────┴────────┘*  #正确方式查询表 t\_aggregating\_mt中的数据,注意需要跟上groupBy  node1 :) **select \* ,sumMerge(salary) from t\_aggregating\_mt group by id,name ,age, loc,dept,workdays,salary ;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┬─sumMerge(salary)─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ java │ 18 │ @B │ 10000.00 │*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ O │ 12000.00 │*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 5 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴────────┴──────────────────┘* |

向表中插入排序字段相同的数据进行分区聚合时，数据按照建表指定的聚合字段进行合并，其他的非聚合字段会保留最初的那条数据，新插入的数据对应的字段值会被舍弃。

|  |
| --- |
| # 向表中插入新的一条数据  node1 :) **insert into t\_aggregating\_mt select 1,'张三三',18,'北京','前端',22,sumState(toDecimal32(5000,2));**  #查询表中的数据，这里为了方便看到分区不合并，直接查询  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┐*  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ O │*  *└────┴──────┴─────┴──────┴──────┴──────────┴────────┘*  *┌─id─┬─name───┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┐*  *│ 1 │ 张三三 │ 18 │ 北京 │ 前端 │ 22 │ ¡ │*  *└────┴────────┴─────┴──────┴──────┴──────────┴────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┐*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 5 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴────────┘*  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┐*  *│ 1 │ 张三 │ 18 │ 北京 │ java │ 18 │ @B │*  *└────┴──────┴─────┴──────┴──────┴──────────┴────────┘*  #使用optimize 命令合并相同分区数据  node1 :) **optimize table t\_aggregating\_mt;**  #再次查询表 t\_aggregating\_mt 表数据，salary 字段已经按照相同分区数据聚合  node1 :) **select \*,sumMerge(salary) from t\_aggregating\_mt group by id,name ,age,loc,dept,workdays,salary;**  *┌─id─┬─name─┬─age─┬─loc──┬─dept─┬─workdays─┬─salary─┬─sumMerge(salary)─┐*  ***│ 1 │ 张三 │ 18 │ 北京 │ java │ 18 │ `ᔠ │ 15000.00 │***  *│ 3 │ 王五 │ 20 │ 北京 │ java │ 26 │ O │ 12000.00 │*  *│ 2 │ 李四 │ 19 │ 上海 │ java │ 22 │ 5 │ 8000.00 │*  *└────┴──────┴─────┴──────┴──────┴──────────┴────────┴──────────────────┘* |

以上方式使用AggregatingMergeTree表引擎比较不方便，更多情况下，我们将AggregatingMergeTree作为物化视图的表引擎与MergeeTree搭配使用。

* 示例：

|  |
| --- |
| #创建表 t\_merge\_base 表，使用MergeTree引擎  node1 :) **create table t\_merge\_base(**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8,**  :-] **loc String,**  :-] **dept String,**  :-] **workdays UInt8,**  :-] **salary Decimal32(2)**  :-] **)engine = MergeTree()**  :-] **order by (id,age)**  :-] **primary key id**  :-] **partition by loc;**  #创建物化视图 view\_aggregating\_mt ,使用AggregatingMergeTree引擎  node1 :) **create materialized view view\_aggregating\_mt**  :-] **engine = AggregatingMergeTree()**  :-] **order by id**  :-] **as select**  :-] **id,**  :-] **name,**  :-] **sumState(salary) as ss**  :-] **from t\_merge\_base**  :-] **group by id ,name ;**  #向表 t\_merge\_base 中插入数据  node1 :) **insert into t\_merge\_base values (1,'张三',18,'北京','大数据',24,10000),**  :-] **(2,'李四',19,'上海','java',22,8000),**  :-] **(3,'王五',20,'北京','java',26,12000);**  #查看 view\_aggregating\_mt视图数据  node1 :) **select \*,sumMerge(ss) from view\_aggregating\_mt group by id,name,ss;**  *┌─id─┬─name─┬─ss─┬─sumMerge(ss)─┐*  *│ 2 │ 李四 │ 5 │ 8000.00 │*  *│ 3 │ 王五 │ O │ 12000.00 │*  *│ 1 │ 张三 │ @B │ 10000.00 │*  *└────┴──────┴────┴──────────────┘*  #继续向表 t\_merge\_base中插入排序键相同的数据  node1 :) **insert into t\_merge\_base values (1,'张三三',18,'北京','前端',22,5000);**  #手动执行optimize 命令，合并物化视图 view\_aggregating\_mt 相同分区数据  node1 :) **optimize table view\_aggregating\_mt;**  #查询视图 view\_aggregating\_mt数据  node1 :) **select \*,sumMerge(ss) from view\_aggregating\_mt group by id,name,ss;**  *┌─id─┬─name─┬─ss─┬─sumMerge(ss)─┐*  *│ 2 │ 李四 │ 5 │ 8000.00 │*  *│ 1 │ 张三 │ `ᔠ│ 15000.00 │*  *│ 3 │ 王五 │ O │ 12000.00 │*  *└────┴──────┴────┴──────────────┘*  **注意：通过普通MergeTree表与AggregatingMergeTree物化视图结合使用，MergeTree中存放原子数据，物化视图中存入聚合结果数据，可以提升数据查询效率。** |

#### CollapsingMergeTree

CollapsingMergeTree就是一种通过以增代删的思路，**支持行级数据修改和删除**的表引擎。它通过定义一个sign标记位字段，记录数据行的状态。如果sign标记为1，则表示这是一行有效的数据；如果sign标记为-1，则表示这行数据需要被删除。当CollapsingMergeTree分区合并时，同一数据分区内，sign标记为1和-1的一组数据会被抵消删除。

每次需要新增数据时，写入一行sign标记为1的数据；需要删除数据时，则写入一行sign标记为-1的数据。此外，只有相同分区内的数据才有可能被折叠。

* CollapsingMergeTree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],  ...  **sign Int8**  ) **ENGINE = CollapsingMergeTree(sign)**  **[PARTITION BY expr]**  **[ORDER BY expr]**  **[SAMPLE BY expr]**  **[SETTINGS name=value, ...]** |

* 存在的问题：

CollapsingMergeTree对于写入数据的顺序有着严格要求，否则导致无法正常折叠。

* 数据折叠保留规则：

在同一个分区内order by 字段相同的数据存在多条，且sign值不同，数据保留规则如下：

1. **如果sign=1和sign=-1的行数相同并且最后一行数据sign=1,则保留第一行sign=-1的行和最后一行sign=1的行。**
2. **如果sign=1的行比sign=-1的行多，则保留最后一条sign=1的行。**
3. **如果sign=-1的行比sign=1的行多，则保留第一条sign=-1的行。**
4. **其他情况，不保留数据。**

* 示例：
* 按照顺序写入需要更新或删除的数据：

|  |
| --- |
| #创建表 t\_collapsing\_mt ,使用CollapsingMergeTree  node1 :) **create table t\_collapsing\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **loc String,**  :-] **login\_times UInt8,**  :-] **total\_dur UInt8,**  :-] **sign Int8**  :-] **)engine = CollapsingMergeTree(sign)**  :-] **order by (id,total\_dur)**  :-] **primary key id**  :-] **partition by loc**  :-] **;**  #向表 t\_collapsing\_mt 中插入以下数据：  node1 :) **insert into t\_collapsing\_mt values(1,'张三','北京',1,30,1),(2,'李四','上海',1,40,1)**  #查看表 t\_collapsing\_mt中的数据  node1 :) **select \* from t\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ 1 │*  *└────┴─────┴──────┴───────────┴──────────┴─────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ 1 │*  *└────┴─────┴──────┴───────────┴─────────┴──────┘*  #向表 t\_collapsing\_mt中继续插入一条数据，删除“张三”数据  node1 :) **insert into t\_collapsing\_mt values(1,'张三','北京',1,30,-1);**  #查询表 t\_collapsing\_mt 中的数据  node1 :) **select \* from t\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ 1 │*  *└────┴─────┴──────┴───────────┴──────────┴─────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ 1 │*  *└───┴─────┴──────┴────────────┴──────────┴─────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ -1 │*  *└────┴─────┴──────┴───────────┴─────────┴─────┘*  #手动触发 optimize 合并相同分区数据  node1 :) **optimize table t\_collapsing\_mt;**  #查询表 t\_collapsing\_mt 中的数据  node1 :) **select \* from t\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ 1 │*  *└───┴─────┴──────┴────────────┴──────────┴─────┘*  #插入以下两条数据，来更新 “李四”数据  node1 :) **insert into t\_collapsing\_mt values(2,'李四','上海',1,40,-1),(2,'李四','上海',2,100,1);**  #查询表 t\_collapsing\_mt 中的数据  node1 :) **select \* from t\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ -1 │*  *│ 2 │ 李四 │ 上海 │ 2 │ 100 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┘*  #手动执行 optimize 触发相同分区合并  node1 :) **optimize table t\_collapsing\_mt;**  #查看表 t\_collapsing\_mt中的数据  node1 :) **select \* from t\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 2 │ 李四 │ 上海 │ 2 │ 100 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┘*  注意：以上功能使用 collapsingMergeTree实现了分区合并。 |

* 乱序写入需要更新或删除的数据：

|  |
| --- |
| #删除表 t\_collapsing\_mt ，重新创建表 t\_collapsing\_mt  这里建表语句与之前一样  #向表 t\_collapsing\_mt 中插入以下数据：  node1 :) **insert into t\_collapsing\_mt values(1,'张三','北京',1,30,-1),(1,'张三','北京',1,30,1),(2,'李四','上海',1,40,1**)  #查询表 t\_collapsing\_mt中的数据  node1 :) **select \* from t\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ -1 │*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┘*  #手动执行 optimize 命令，合并相同分区数据  node1 :) **optimize table t\_collapsing\_mt;**  #查询表 t\_collapsing\_mt表中的数据，数据没有变化  node1 :) **select \* from t\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┐*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ -1 │*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┘*  **注意：当数据插入到表中的顺序标记如果不是1,-1这种顺序时，合并相同分区内的数据不能达到修改和更新效果。** |

如果数据的写入程序是单线程执行的，则能够较好地控制写入顺序；如果需要处理的数据量很大，数据的写入程序通常是多线程执行的，那么此时就不能保障数据的写入顺序了。在这种情况下，CollapsingMergeTree的工作机制就会出现问题。但是可以通过VersionedCollapsingMergeTree的表引擎得到解决。

#### VersionedCollapsingMergeTree

上面提到CollapsingMergeTree表引擎对于数据写入乱序的情况下，不能够实现数据折叠的效果。VersionedCollapsingMergeTree表引擎的作用与CollapsingMergeTree完全相同，它们的不同之处在于，VersionedCollapsingMergeTree对数据的写入顺序没有要求，在同一个分区内，任意顺序的数据都能够完成折叠操作。

VersionedCollapsingMergeTree使用version列来实现乱序情况下的数据折叠，该引擎除了需要指定一个sign标识之外，还需要指定一个UInt\*类型的version版本号。

* VersionedCollapsingMergeTree建表语句：

|  |
| --- |
| CREATE TABLE [IF NOT EXISTS] [db.]table\_name [ON CLUSTER cluster]  (  name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],  name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],  ...  **sign Int8,**  **version UInt8**  ) ENGINE = **VersionedCollapsingMergeTree(sign, version)**  **[PARTITION BY expr]**  **[ORDER BY expr]**  **[SAMPLE BY expr]**  **[SETTINGS name=value, ...]** |

* 示例：

|  |
| --- |
| #创建表 t\_version\_collapsing\_mt ,使用VersionedCollapsingMergeTree引擎  node1 :) **create table t\_version\_collapsing\_mt(**  :-] **id UInt8,**  :-] **name String,**  :-] **loc String,**  :-] **login\_times UInt8,**  :-] **total\_dur UInt8,**  :-] **sign Int8,**  :-] **version UInt8**  :-] **) engine = VersionedCollapsingMergeTree(sign,version)**  :-] **order by (id,total\_dur)**  :-] **primary key id**  :-] **partition by loc;**  #向表 t\_version\_collapsing\_mt 中插入以下数据  node1 :) **insert into table t\_version\_collapsing\_mt values(1,'张三','北京',1,30,-1,1),(2,'李四','上海',1,40,1,2);**  #查询表 t\_version\_collapsing\_mt 中的数据  node1 :) **select \* from t\_version\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┬─version─┐*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ -1 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┴─────────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┬─version─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ 1 │ 2 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┴─────────┘*  #向表 t\_version\_collapsing\_mt中插入以下数据，删除“张三”信息，更新“李四”信息  node1 :) **insert into table t\_version\_collapsing\_mt values(1,'张三','北京',1,30,1,1),(2,'李四','上海',1,40,-1,2),(2,'李四','上海',2,100,1,2);**  #查询表 t\_version\_collapsing\_mt中的数据  node1 :) **select \* from t\_version\_collapsing\_mt ;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┬─version─┐*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ -1 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┴─────────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┬─version─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ 1 │ 2 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┴─────────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┬─version─┐*  *│ 1 │ 张三 │ 北京 │ 1 │ 30 │ 1 │ 1 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┴─────────┘*  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┬─version─┐*  *│ 2 │ 李四 │ 上海 │ 1 │ 40 │ -1 │ 2 │*  *│ 2 │ 李四 │ 上海 │ 2 │ 100 │ 1 │ 2 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┴─────────┘*  #手动执行 optimize 命令，合并相同分区的数据，这里有可能需要执行多次  node1 :) **optimize table t\_version\_collapsing\_mt;**  **注意：如果不想执行多次，也可以执行 optimize table t\_version\_collapsing\_mt final语句**  #查询表 t\_version\_collapsing\_mt 中的数据如下：  node1 :) **select \* from t\_version\_collapsing\_mt;**  *┌─id─┬─name─┬─loc──┬─login\_times─┬─total\_dur─┬─sign─┬─version─┐*  *│ 2 │ 李四 │ 上海 │ 2 │ 100 │ 1 │ 2 │*  *└────┴──────┴──────┴─────────────┴───────────┴──────┴─────────┘* |

### **Integration系列表引擎**

ClickHouse提供了许多与外部系统集成的方法，包括一些表引擎。这些表引擎与其他类型的表引擎类似，可以用于将外部数据导入到ClickHouse中，或者在ClickHouse中直接操作外部数据源。

#### HDFS

HDFS引擎支持ClickHouse 直接读取HDFS中特定格式的数据文件，目前文件格式支持Json,Csv文件等，ClickHouse通过HDFS引擎建立的表，不会在ClickHouse中产生数据，读取的是HDFS中的数据，将HDFS中的数据映射成ClickHouse中的一张表，这样就可以使用SQL操作HDFS中的数据。

ClickHouse并不能够删除HDFS上的数据，当我们在ClickHouse客户端中删除了对应的表，只是删除了表结构，HDFS上的文件并没有被删除，这一点跟Hive的外部表十分相似。

* 语法：

|  |
| --- |
| **ENGINE = HDFS(URI, format)** |

注意：URI是HDFS文件路径，format指定文件格式。HDFS文件路径中文件为多个时，可以指定成some\_file\_?，或者当数据映射的是HDFS多个文件夹下数据时，可以指定somepath/\* 来指定URI

* 其他配置：

由于HDFS配置了HA 模式，有集群名称，所以URI使用mycluster HDFS集群名称时，ClickHouse不识别，这时需要做以下配置：

1. 将hadoop路径下$HADOOP\_HOME/etc/hadoop下的hdfs-site.xml文件复制到/etc/clickhouse-server目录下。
2. 修改/etc/init.d/clickhouse-server 文件，加入一行 “export LIBHDFS3\_CONF=/etc/clickhouse-server/hdfs-site.xml”
3. 重启ClickHouse-server 服务

serveice clickhouse-server restart

当然，这里也可以不做以上配置，在写HDFS URI时，直接写成对应的节点+端口即可。

* 示例：

|  |
| --- |
| #在HDFS路径 hdfs://mycluster/ch/路径下，创建多个csv文件，写入一些数据  c1.csv文件内容:  1,张三,19  2,李四,20  c2.csv文件内容:  3,王五,21  4,马六,22  #创建表 t\_hdfs,使用HDFS引擎  node1 :) **create table t\_hdfs(id UInt8,name String,age UInt8) engine = HDFS('hdfs://mycluster/ch/\*.csv','CSV')**  #查询表 t\_hdfs中的数据  node1 :) **select \* from t\_hdfs;**  *┌─id─┬─name─┬─age─┐*  *│ 3 │ 王五 │ 21 │*  *│ 4 │ 马六 │ 22 │*  *└────┴──────┴─────┘*  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 19 │*  *│ 2 │ 李四 │ 20 │*  *└────┴──────┴─────┘*  **注意：这里表t\_hdfs不会在clickhouse对应的节点路径下创建数据目录，同时这种表映射的是HDFS路径中的csv文件，不能插入数据，t\_hdfs是只读表。**  #创建表 t\_hdfs2 文件 ，使用HDFS引擎  node1 :) **create table t\_hdfs2(id UInt8,name String,age UInt8) engine = HDFS('hdfs://mycluster/chdata','CSV');**  #向表 t\_hdfs2中写入数据  node1 :) **insert into t\_hdfs2 values(5,'田七',23),(6,'赵八',24);**  #查询表t\_hdfs2中的数据  node1 :) **select \* from t\_hdfs2;**  *┌─id─┬─name─┬─age─┐*  *│ 5 │ 田七 │ 23 │*  *│ 6 │ 赵八 │ 24 │*  *└────┴──────┴─────┘*  **注意：t\_hdfs2表没有直接映射已经存在的HDFS文件，这种表允许查询和插入数据。** |

#### MySQL

ClickHouse MySQL数据库引擎可以将MySQL某个库下的表映射到ClickHouse中，使用ClickHouse对数据进行操作。ClickHouse同样支持MySQL表引擎，即映射一张MySQL中的表到ClickHouse中，使用ClickHouse进行数据操作，与MySQL数据库引擎一样，这里映射的表只能做查询和插入操作，不支持删除和更新操作。

* 语法：

|  |
| --- |
| 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],  ...  ) ENGINE = MySQL('host:port', 'database', 'table', 'user', 'password'[, replace\_query, 'on\_duplicate\_clause']); |

* 以上语法的解释如下：
* host:port - MySQL服务器名称和端口
* database - MySQL 数据库。
* table - 映射的MySQL中的表
* user - 登录mysql的用户名
* password - 登录mysql的密码
* replace\_query - 将INSERT INTO 查询是否替换为 REPLACE INTO 的标志,默认为0，不替换。当设置为1时，所有的insert into 语句更改为 replace into 语句。当插入的数据有重复主键数据时，此值为0默认报错，此值为1时，主键相同这条数据，默认替换成新插入的数据。
* on\_duplicate\_clause - 默认不使用。当插入数据主键相同时，可以指定只更新某列的数据为新插入的数据，对应于on duplicate key 后面的语句，其他的值保持不变，需要replace\_query 设置为0。
* 示例：

|  |
| --- |
| #在mysql 中创建一张表 t\_ch,指定id为主键  **CREATE TABLE t\_ch (**  **id INT,**  **NAME VARCHAR (255),**  **age INT,**  **PRIMARY KEY (id)**  **)**  #向表中增加一些数据  **insert into t\_ch values (1,"张三",18),(2,"李四",19),(3,"王五",20)**  #在ClickHouse中创建MySQL引擎表 t\_mysql\_engine  node1 :) **create table t\_mysql\_engine (**  :-]  **id UInt8,**  :-]  **name String,**  :-]  **age UInt8**  :-] **)engine = MySQL('node2:3306','test','t\_ch','root','123456');**  #查询ClickHouse表 t\_mysql\_engine 中的数据：  node1 :) **select \* from t\_mysql\_engine;**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *└────┴──────┴─────┘*  #在ClickHouse中向表 t\_mysql\_engine中插入一条数据  node1 :) insert into t\_mysql\_engine values (4,'马六','21');  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *│ 4 │ 马六 │ 21 │*  *└───┴─────┴───┘*  #在ClickHouse中向表 t\_mysql\_engine中再插入一条数据，这里主键重复，报错。  node1 :) **insert into t\_mysql\_engine values (4,'田七','22');**  *Exception: mysqlxx::BadQuery: Duplicate entry '4' for key*  *'PRIMARY' (node2:3306).*  **注意：在clickhouse 中 t\_mysql\_engine表不会在ClickHouse服务器节点上创建数据目录。** |

* 测试 replace\_query ：

|  |
| --- |
| #在mysql 中删除表 t\_ch,重新创建，指定id为主键  **CREATE TABLE t\_ch (**  **id INT,**  **NAME VARCHAR (255),**  **age INT,**  **PRIMARY KEY (id)**  **)**  #向表中增加一些数据  **insert into t\_ch values (1,"张三",18),(2,"李四",19),(3,"王五",20)**  #在ClickHouse中删除MySQL引擎表 t\_mysql\_engine，重建  node1 :) **create table t\_mysql\_engine (**  :-]  **id UInt8,**  :-]  **name String,**  :-]  **age UInt8**  :-] **)engine = MySQL('node2:3306','test','t\_ch','root','123456',1);**  #查询ClickHouse表 t\_mysql\_engine 中的数据：  node1 :) **select \* from t\_mysql\_engine;**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *└────┴──────┴─────┘*  #在ClickHouse中向表 t\_mysql\_engine中插入一条数据，主键重复。这里由于指定了replace\_query = 1 ,所以当前主键数据会被替换成新插入的数据。  node1 :) **insert into t\_mysql\_engine values (3,'马六','21');**  #查询ClichHouse t\_mysql\_engine表数据  node1 :) **select \* from t\_mysql\_engine;**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 马六 │ 21 │*  *└────┴──────┴─────┘* |

* 测试 on\_duplicate\_clause：

|  |
| --- |
| #在mysql 中删除表 t\_ch,重新创建，指定id为主键  **CREATE TABLE t\_ch (**  **id INT,**  **NAME VARCHAR (255),**  **age INT,**  **PRIMARY KEY (id)**  **)**  #向表中增加一些数据  **insert into t\_ch values (1,"张三",18),(2,"李四",19),(3,"王五",20)**  #在ClickHouse中删除MySQL引擎表 t\_mysql\_engine，重建  node1 :) **create table t\_mysql\_engine (**  :-]  **id UInt8,**  :-]  **name String,**  :-]  **age UInt8**  :-] **)engine = MySQL('node2:3306','test','t\_ch','root','123456',0,'update age = values(age)');**  #查询ClickHouse表 t\_mysql\_engine 中的数据：  node1 :) **select \* from t\_mysql\_engine;**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *└────┴──────┴─────┘*  *#在ClickHouse 中向表 t\_mysql\_engine中插入一条数据*  node1 :) **insert into t\_mysql\_engine values (4,'马六','21');**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *│ 4 │ 马六 │ 21 │*  *└──┴─────┴────┘*  #在ClickHouse中向表 t\_mysql\_engine中插入一条数据，主键重复。  node1 :) **insert into t\_mysql\_engine values (4,'田七','100');**  #查询ClichHouse t\_mysql\_engine表数据  node1 :) **select \* from t\_mysql\_engine;**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  ***│ 4 │ 马六 │ 100 │***  *└────┴──────┴─────┘* |

#### Kafka

ClickHouse中还可以创建表指定为Kafka为表引擎，这样创建出的表可以查询到Kafka中的流数据。对应创建的表不会将数据存入ClickHouse中，这里这张kafka引擎表相当于一个消费者，消费Kafka中的数据，数据被查询过后，就不会再次被查询到。

* 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 = Kafka()  SETTINGS  kafka\_broker\_list = 'host:port',  kafka\_topic\_list = 'topic1,topic2,...',  kafka\_group\_name = 'group\_name',  kafka\_format = 'data\_format'[,] |

* 对以上参数的解释：
* kafka\_broker\_list: 以逗号分隔的Kafka Broker节点列表
* kafka\_topic\_list ： topic列表
* kafka\_group\_name ： kafka消费者组名称
* kafka\_format : Kafka中消息的格式，例如：JSONEachRow、CSV等等，具体参照https://clickhouse.tech/docs/en/interfaces/formats/。这里一般使用JSONEachRow格式数据，需要注意的是，json字段名称需要与创建的Kafka引擎表中字段的名称一样，才能正确的映射数据。
* 示例：

|  |
| --- |
| #创建表 t\_kafka\_consumer ,使用Kafka表引擎  node1 :) **create table t\_kafka\_consumer (**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8**  :-] **) engine = Kafka()**  :-] **settings**  :-] **kafka\_broker\_list='node1:9092,node2:9092,node3:9092',**  :-] **kafka\_topic\_list='ck-topic',**  :-] **kafka\_group\_name='group1',**  :-] **kafka\_format='JSONEachRow';**  #启动kafka,在kafka中创建ck-topic topic,并向此topic中生产以下数据：  创建topic:  **kafka-topics.sh --zookeeper node3:2181,node4:2181,node5:2181 --create --topic ck-topic --partitions 3 --replication-factor 3**  生产数据：  **kafka-console-producer.sh --broker-list node1:9092,node2:9092,node3:9092 --topic ck-topic**  生产数据如下：  **{"id":1,"name":"张三","age":18}**  **{"id":2,"name":"李四","age":19}**  **{"id":3,"name":"王五","age":20}**  **{"id":4,"name":"马六","age":21}**  **{"id":5,"name":"田七","age":22}**  #在ClickHouse中查询表 t\_kafka\_consumer数据，可以看到生产的数据  node1 :) select \* from t\_kafka\_consumer;  *┌─id─┬─name─┬─age─┐*  *│ 2 │ 李四 │ 19 │*  *│ 5 │ 田七 │ 22 │*  *│ 1 │ 张三 │ 18 │*  *│ 4 │ 马六 │ 21 │*  *│ 3 │ 王五 │ 20 │*  *└────┴──────┴─────┘*  **注意：再次查看表 t\_kafka\_consumer数据 ，我们发现读取不到任何数据，这里对应的ClikcHouse中的Kafka引擎表，只是相当于是消费者，消费读取Kafka中的数据，数据被消费完成之后，不能再次查询到对应的数据。** |

以上在ClickHouse中创建的Kafka引擎表 t\_kafka\_consumer 只是一个数据管道，当查询这张表时就是消费Kafka中的数据，数据被消费完成之后，不能再次被读取到。如果想将Kafka中topic中的数据持久化到ClickHouse中，我们可以通过物化视图方式访问Kafka中的数据，可以通过以下三个步骤完成将Kafka中数据持久化到ClickHouse中：

1. 创建Kafka 引擎表,消费kafka中的数据。
2. 再创建一张ClickHouse中普通引擎表，这张表面向终端用户查询使用。这里生产环境中经常创建MergeTree家族引擎表。
3. 创建物化视图，将Kafka引擎表数据实时同步到终端用户查询表中。

示例：

|  |
| --- |
| #在ClickHouse中创建 t\_kafka\_consumer2 表，使用Kafka引擎  node1 :) **create table t\_kafka\_consumer2 (**  :-] **id UInt8,**  :-] **name String,**  :-] **age UInt8**  :-] **) engine = Kafka()**  :-] **settings**  :-] **kafka\_broker\_list='node1:9092,node2:9092,node3:9092',**  :-] **kafka\_topic\_list='ck-topic',**  :-] **kafka\_group\_name='group1',**  :-] **kafka\_format='JSONEachRow';**  #在ClickHouse中创建一张终端用户查询使用的表，使用MergeTree引擎  node1 :) create table t\_kafka\_mt(  :-] id UInt8,  :-] name String,  :-] age UInt8  :-] ) engine = MergeTree()  :-] order by id；  #创建物化视图，同步表t\_kafka\_consumer2数据到t\_kafka\_mt中  node1 :) **create** **materialized view view\_consumer to t\_kafka\_mt**  :-] **as select id,name,age from t\_kafka\_consumer2;**  **注意：物化视图在ClickHouse中也是存储数据的，create materialized view view\_consumer to t\_kafka\_mt 语句是将物化视图view\_consumer中的数据存储到到对应的t\_kafka\_mt 表中，这样同步的目的是如果不想继续同步kafka中的数据，可以直接删除物化视图即可。**  #向Kafka ck-topic中生产以下数据：  生产数据：  **kafka-console-producer.sh --broker-list node1:9092,node2:9092,node3:9092 --topic ck-topic**  生产数据如下：  **{"id":1,"name":"张三","age":18}**  **{"id":2,"name":"李四","age":19}**  **{"id":3,"name":"王五","age":20}**  **{"id":4,"name":"马六","age":21}**  **{"id":5,"name":"田七","age":22}**  #查询表 t\_kafka\_mt中的数据，数据同步完成。  node1 :) **select \* from t\_kafka\_mt;**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *│ 4 │ 马六 │ 21 │*  *│ 5 │ 田七 │ 22 │*  *└────┴──────┴─────┘* |

## ClickHouse SQL语法

### **SQL DDL 操作**

DDL:Data Definition Language,数据库定义语言。在ClickHouse中，DDL语言中修改表结构仅支持Merge表引擎、Distributed表引擎及MergeTree家族的表引擎,SQL 中的库、表、字段严格区分大小写。

#### 创建库

* 创建库基础语法：

|  |
| --- |
| **CREATE DATABASE [IF NOT EXISTS] db\_name [ON CLUSTER cluster] [ENGINE = engine(...)]** |

#### 查看数据库

* 查看数据库语法

|  |
| --- |
| **SHOW DATABASES;** |

#### 删除库

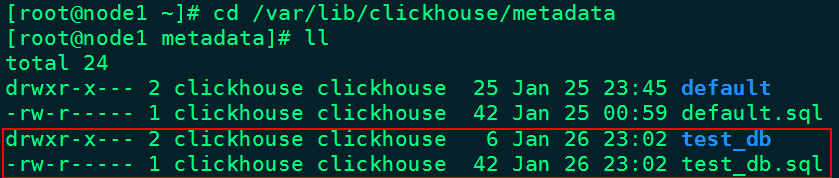
* 删除库基础语法：

|  |
| --- |
| **DROP DATABASE [IF EXISTS] db [ON CLUSTER cluster]** |

* 示例：

|  |
| --- |
| #创建库 test\_db  node1 :) **create database if not exists test\_db;**  *CREATE DATABASE IF NOT EXISTS test\_db*  *Ok.*  *0 rows in set. Elapsed: 0.007 sec.*  #删除库  node1 :) **drop database test\_db;**  *DROP DATABASE test\_db*  *Ok.*  *0 rows in set. Elapsed: 0.003 sec.* |

注意：在创建数据库时，在/var/lib/clickhouse/metadata/目录下会有对应的库目录和库.sql文件，库目录中会存入在当前库下建表的信息，xx.sql文件中存入的是建库的信息。如图：



当删除数据库时，/var/lib/clickhouse/metadata/目录下对应的库目录和xx.sql文件也会被清空。

#### 创建表

创建表的基本语法：

|  |
| --- |
| #第一种  **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**  #第二种  **CREATE TABLE [IF NOT EXISTS] [db.]table\_name AS [db2.]name2 [ENGINE = engine]**  #第三种  **CREATE TABLE [IF NOT EXISTS] [db.]table\_name ENGINE = engine AS SELECT ...** |

注意：以上普通第一种建表语句是直接创建表。第二种创建表方式可以创建一个与db2中name2一样结构的表，也可以指定表引擎，也可以不指定，不指定默认与db2中的name2表引擎一样，不会将表name2中的数据填充到对应的新表中。第三种方式可以指定引擎创建一个与Select 子句的结果相同结构的表，并将Select子句的结果填充它。

* 示例：

|  |
| --- |
| #第一种方式创建表  node1 :) **create table if not exists newdb.t1(**  :-] **id UInt8 default 0 comment '编号',**  :-] **name String default '无姓名' comment '姓名',**  :-] **age UInt8 default 18 comment '年龄'**  :-] **)engine = TinyLog;**  *CREATE TABLE IF NOT EXISTS newdb.t1*  *(*  *`id` UInt8 DEFAULT 0 COMMENT '编号',*  *`name` String DEFAULT '无姓名' COMMENT '姓名',*  *`age` UInt8 DEFAULT 18 COMMENT '年龄'*  *)*  *ENGINE = TinyLog*  *Ok.*  *0 rows in set. Elapsed: 0.004 sec.*  # 第二种方式创建表  node1 :) **create table if not exists t2 engine = Memory as newdb.t\_tinylog;**  *CREATE TABLE IF NOT EXISTS t2 AS newdb.t\_tinylog*  *ENGINE = Memory*  *Ok.*  *0 rows in set. Elapsed: 0.006 sec.*  # 第三种方式创建表  node1 :) **create table if not exists t3 engine = Memory as select \* from newdb.t\_tinylog where id >2;**  *CREATE TABLE IF NOT EXISTS t3*  *ENGINE = Memory AS*  *SELECT \**  *FROM newdb.t\_tinylog*  *WHERE id > 2*  *Ok.*  *0 rows in set. Elapsed: 0.010 sec.*  #查询表t3数据  node1 :) **select \* from t3;**  *SELECT \**  *FROM t3*  *┌─id─┬─name─┬─age─┐*  *│ 3 │ 王五 │ 20 │*  *└────┴──────┴─────┘*  *1 rows in set. Elapsed: 0.004 sec.* |

#### 查看表

* 查看表语法：

|  |
| --- |
| **SHOW TABLES;**  **SHOW TABLES IN default;** |

#### 查看表的定义

* 查看表定义语法：

|  |
| --- |
| **SHOW CREATE TABLE XXX;** |

* 示例：

|  |
| --- |
| #查看表定义  node1 :) **show create table t3;**  *SHOW CREATE TABLE t3*  *┌─statement─────────────────────────────────┐*  *│ CREATE TABLE newdb.t3*  *(*  *`id` UInt8,*  *`Name` String*  *)*  *ENGINE = TinyLog │*  *└───────────────────────────────────────────┘*  *1 rows in set. Elapsed: 0.002 sec.* |

#### 查看表的字段

* 查看表定义语法：

|  |
| --- |
| **DESC XXXX;** |

* 示例：

|  |
| --- |
| #查看表t3的字段  node1 :) **desc t3;**  *DESCRIBE TABLE t3*  *┌─name─┬─type───┬─default\_type─┬─...*  *│ id │ UInt8 │ │ ...*  *│ Name │ String │ │ ...*  *└──────┴────────┴──────────────┴──...*  *2 rows in set. Elapsed: 0.004 sec.* |

#### 删除表

* 删除表的基本语法：

|  |
| --- |
| **DROP [TEMPORARY] TABLE [IF EXISTS] [db.]name [ON CLUSTER cluster]** |

* 示例：

|  |
| --- |
| #删除表  node1 :) **drop table t3;**  *DROP TABLE t3*  *Ok.*  *0 rows in set. Elapsed: 0.003 sec.* |

#### 修改表

* 修改表语法

|  |
| --- |
| **ALTER TABLE [db].name [ON CLUSTER cluster] ADD|DROP|CLEAR|COMMENT|MODIFY COLUMN ...** |

##### 添加列

* 示例：

|  |
| --- |
| #使用default 库，创建表 test1,使用MergeTree引擎  node1 :) **use default;**  node1 :) **create table test1(id UInt8,name String)engine = MergeTree() order by id partition by name;**  *CREATE TABLE test1*  *(*  *`id` UInt8,*  *`name` String,*  *`loc` String*  *)*  *ENGINE = MergeTree()*  *PARTITION BY loc*  *ORDER BY id*  *Ok.*  *0 rows in set. Elapsed: 0.005 sec.Ok.*  #查看表test1表结构  node1 :) **desc test1;**  *DESCRIBE TABLE test1*  *┌─name─┬─type───┬...*  *│ id │ UInt8 │...*  *│ name │ String │...*  *│ loc │ String │...*  *└──────┴────────┴...*  *3 rows in set. Elapsed: 0.004 sec.*  #添加表字段  node1 :) **alter table test1 add column age UInt8;**  #查看表结构，添加字段成功  node1 :) **desc test1;**  *DESCRIBE TABLE test1*  *┌─name─┬─type───┬...*  *│ id │ UInt8 │...*  *│ name │ String │...*  *│ loc │ String │...*  *│ age │ UInt8 │...*  *└─────┴─────┴...*  *4 rows in set. Elapsed: 0.003 sec.* |

##### 删除列

* 示例：

|  |
| --- |
| #删除表test1中的name age字段  node1 :) **alter table test1 drop column age;**  #查看表 test1表结构  node1 :) **desc test1;**  *DESCRIBE TABLE test1*  *┌─name─┬─type───┬...*  *│ id │ UInt8 │...*  *│ name │ String │...*  *│ loc │ String │...*  *└──────┴────────┴...*  *2 rows in set. Elapsed: 0.004 sec.* |

##### 清空列

注意，不能清空排序、主键、分区字段。

* 示例：

|  |
| --- |
| #向表 test1中插入以下几条数据  node1 :) **insert into table test1 values (1,'张三','北京'),(2,'李四','上海'),(3,'王五','北京');**  #查看表中的数据  *┌─id─┬─name─┬─loc──┐*  *│ 1 │ 张三 │ 北京 │*  *│ 3 │ 王五 │ 北京 │*  *└────┴──────┴──────┘*  *┌─id─┬─name─┬─loc──┐*  *│ 2 │ 李四 │ 上海 │*  *└────┴──────┴──────┘*  **#清空 test1 name列在’北京’分区的值**  node1 :) **alter table test1 clear column name in partition '北京';**  #查看表中的数据  node1 :) **select \* from test1;**  *┌─id─┬─name─┬─loc──┐*  *│ 1 │ │ 北京 │*  *│ 3 │ │ 北京 │*  *└────┴──────┴──────┘*  *┌─id─┬─name─┬─loc──┐*  *│ 2 │ 李四 │ 上海 │*  *└────┴──────┴──────┘*  **#清空 test1 name 列下的值**  node1 :) **alter table test1 clear column name;**  #查看表中的数据  node1 :) **select \* from test1;**  *┌─id─┬─name─┬─loc──┐*  *│ 1 │ │ 北京 │*  *│ 3 │ │ 北京 │*  *└───┴────┴─────┘*  *┌─id─┬─name─┬─loc──┐*  *│ 2 │ │ 上海 │*  *└───┴─────┴────┘* |

##### 给列修改注释

* 示例：

|  |
| --- |
| #修改表 test1 name 列的注释  node1 :) **alter table test1 comment column name '姓名';**  #查看表 test1描述  *┌─name─┬─type───┬─default\_type─┬─default\_expression─┬─comment─┬...*  *│ id │ UInt8 │ │ │ │...*  ***│ name │ String │ │ │ 姓名 │...***  *│ loc │ String │ │ │ │...*  *└──────┴────────┴──────────────┴────────────────────┴─────────┴...* |

##### 修改列类型

* 示例：

|  |
| --- |
| #修改表 test1 name列类型为UInt8  node1 :) **alter table test1 modify column name UInt8**  #node1 :) desc test1;  *┌─name─┬─type───┬─default\_type─┬─default\_expression─┬─comment─┬*  *│ id │ UInt8 │ │ │ │*  ***│ name │ UInt8 │ │ │ 姓名 │***  *│ loc │ String │ │ │ │*  *└──────┴────────┴──────────────┴────────────────────┴─────────┴* |

#### 给表重命名

给表重新命名可以作用在任意的表引擎上。

* 给表重命名语法：

|  |
| --- |
| **RENAME TABLE [db11.]name11 TO [db12.]name12, [db21.]name21 TO [db22.]name22, ... [ON CLUSTER cluster]** |

示例：

|  |
| --- |
| #创建库 testdb1  node1 :) **create database testdb1;**  #创建库 testdb2  node1 :) **create database testdb2;**  #使用库testdb1,并创建表 t1  node1 :) **use testdb1;**  node1 :) **create table t1 (id UInt8 ,name String) engine = MergeTree() order by id ;**  #将表 t1 重命名为test1  node1 :) **rename table t1 to test1;**  #将表test1 移动到testdb2库下，并重新命名为t2, testdb1 下没有表了  node1 :) **rename table testdb1.test1 to testdb2.t2;** |

#### **分区表的DDL操作**

ClickHouse中只有MergeTree家族引擎下的表才能分区。这里说的分区表就是MergeTree家族表引擎对应的分区表。

##### 查看分区信息

* 示例：

|  |
| --- |
| #在newdb中创建分区表 t\_partition ,使用MergeTree引擎  node1 :) **create table t\_partition (id UInt8,name String,age UInt8,loc String) engine = MergeTree() order by id partition by loc;**  #向表 t\_partition 中插入以下数据：  node1 :) **insert into t\_partition values (1,'张三',18,'BJ'),(2,'李四',19,'GZ'),(3,'王五',20,'BJ'),(4,'马六',21,'GZ');**  #查询表 t\_partition 的分区信息  node1 :) **select database,table,name,partition from system.parts where table = 't\_partition';**  *┌─database─┬─table───────┬─name───────────────────────────────────┬─partition─┐*  *│ newdb │ t\_partition │ 8700fff36a8bf87b6ea3eedb16d04038\_2\_2\_0 │ GZ │*  *│ newdb │ t\_partition │ e35d0ca9d946a627c9fc98b8f80391ce\_1\_1\_0 │ BJ │*  *└──────────┴─────────────┴────────────────────────────────────────┴───────────┘*  #也可以在ClickHouse节点上查看分区信息，路径为：/var/lib/clickhouse/data/newdb/t\_partition/,信息如下： |

##### 卸载分区

将指定分区的数据移动到 detached 目录。服务器会忽略被分离的数据分区。只有当你使用 ATTACH 时，服务器才会知晓这部分数据。当执行操作以后，可以对 detached 目录的数据进行任意操作，例如删除文件，或者放着不管。

* 卸载分区语法：

|  |
| --- |
| **ALTER TABLE table\_name DETACH PARTITION partition\_expr** |

* 示例：

|  |
| --- |
| #卸载 表 t\_partition 中 ‘BJ’分区数据  node1 :) **alter table t\_partition detach partition 'BJ'**  #查看表 t\_partition中的数据  node1 :) **select \* from t\_partition;**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 2 │ 李四 │ 19 │ GZ │*  *│ 4 │ 马六 │ 21 │ GZ │*  *└────┴──────┴─────┴─────┘*  #查看表 t\_partition 中的分区信息  node1 :) **select database,table,name,partition from system.parts where table = 't\_partition';**  *┌─database─┬─table───────┬─name───────────────────────────────────┬─partition─┐*  *│ newdb │ t\_partition │ 8700fff36a8bf87b6ea3eedb16d04038\_2\_2\_0 │ GZ │*  *└──────────┴─────────────┴────────────────────────────────────────┴───────────┘*  #查看路径/var/lib/clickhouse/data/newdb/t\_partition/detached中数据，发现卸载的对应分区移动到此目录中 |

##### 装载分区

我们可以将已经卸载的分区重新装载到对应的表分区中。这里就是将detached目录中的数据重新移动到对应的表数据目录下。

也可以将卸载的分区数据加载到其他表中，但是这个表需要与原来的表具有相同的表结构及相同的分区字段。

* 装载分区数据语法：

|  |
| --- |
| **ALTER TABLE table\_name ATTACH PARTITION partition\_expr** |

* 示例：

|  |
| --- |
| #将表 t\_partition 对应的 ‘BJ’分区装载回来  node1 :) **alter table t\_partition attach partition 'BJ';**  #查看表 t\_partition 中的数据  node1 :) **select \* from t\_partition;**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 1 │ 张三 │ 18 │ BJ │*  *│ 3 │ 王五 │ 20 │ BJ │*  *└────┴──────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 2 │ 李四 │ 19 │ GZ │*  *│ 4 │ 马六 │ 21 │ GZ │*  *└────┴──────┴─────┴─────┘*  #查看表 t\_partition 分区信息  node1 :) **select database,table,name,partition from system.parts where table = 't\_partition';** |

##### 替换分区

替换分区支持将table1表的分区数据复制到table2表，并替换table2表的已有分区。table1表中分区数据不会被删除，table1和table2表必须要有相同的表结构且分区字段相同。这个操作经常用作数据备份、表数据同步操作。

* 替换分区语法：

|  |
| --- |
| **ALTER TABLE table2 REPLACE PARTITION partition\_expr FROM table1** |

* 示例：

|  |
| --- |
| #创建表 table1 和table2 ，使用MergeTree表引擎，并且两表结构相同  node1 :) **create table table1 (id UInt8,name String,age UInt8,loc String) engine = MergeTree() order by id partition by loc;**  node1 :) **create table table2 (id UInt8,name String,age UInt8,loc String) engine = MergeTree() order by id partition by loc;**  #向table1中插入以下数据  node1 :) **insert into table1 values (1,'张三',18,'BJ'),(2,'李四',19,'GZ'),(3,'王五',20,'BJ'),(4,'马六',21,'GZ');**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 1 │ 张三 │ 18 │ BJ │*  *│ 3 │ 王五 │ 20 │ BJ │*  *└───┴────┴────┴────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 2 │ 李四 │ 19 │ GZ │*  *│ 4 │ 马六 │ 21 │ GZ │*  *└───┴────┴────┴────┘*  #向table2中插入以下数据  node1 :) **insert into table2 values (5,'田七',22,'BJ'),(6,'赵八',23,'GZ'),(7,'李九',24,'BJ'),(8,'郑十',25,'GZ');**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 5 │ 田七 │ 22 │ BJ │*  *│ 7 │ 李九 │ 24 │ BJ │*  *└───┴────┴────┴────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 6 │ 赵八 │ 23 │ GZ │*  *│ 8 │ 郑十 │ 25 │ GZ │*  *└───┴────┴────┴────┘*  #将table1表中’BJ’分区内的数据替换到table2中  node1 :) **alter table table2 replace partition 'BJ' from table1;**  #查看表 table2中的数据  node1 :) **select \* from table2;**  ***┌─id─┬─name─┬─age─┬─loc─┐***  ***│ 1 │ 张三 │ 18 │ BJ │***  ***│ 3 │ 王五 │ 20 │ BJ │***  ***└────┴──────┴─────┴─────┘***  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 6 │ 赵八 │ 23 │ GZ │*  *│ 8 │ 郑十 │ 25 │ GZ │*  *└────┴──────┴─────┴─────┘*  #查看表 table1中的数据，没有变化，不会删除 ‘BJ’ 分区的数据  node1 :) **select \* from table1;**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 1 │ 张三 │ 18 │ BJ │*  *│ 3 │ 王五 │ 20 │ BJ │*  *└────┴──────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 2 │ 李四 │ 19 │ GZ │*  *│ 4 │ 马六 │ 21 │ GZ │*  *└────┴──────┴─────┴─────┘* |

##### 删除分区

ClickHouse中的分区表，可以针对分区表删除某个分区，之后再导入当前分区的数据，以达到数据更新的目的。

执行删除分区命名是直接将对应分区数据删除，不会放入detached目录。该操作会将分区标记为不活跃的，然后在大约10分钟内删除全部数据。

* 删除分区语法：

|  |
| --- |
| **ALTER TABLE table\_name DROP PARTITION partition\_expr** |

* 示例：

|  |
| --- |
| #删除表 t\_partition 中的 'BJ' 分区：  node1 :) **alter table t\_partition drop partition 'BJ';**  #查询 t\_partition 中的分区信息：  node1 :) **select database,table,name,partition from system.parts where table = 't\_partition';**  *┌─database─┬─table───────┬─name───────────────────────────────────┬─partition─┐*  *│ newdb │ t\_partition │ 8700fff36a8bf87b6ea3eedb16d04038\_2\_2\_0 │ GZ │*  *└──────────┴─────────────┴────────────────────────────────────────┴───────────┘* |

##### 移动分区

该操作将 table\_source表的数据分区移动到 table\_dest表，并删除table\_source表的数据。

* 移动分区语法：

|  |
| --- |
| **ALTER TABLE table\_source MOVE PARTITION partition\_expr TO TABLE table\_dest** |

* 示例：

|  |
| --- |
| #创建表 table\_source ,table\_dest, 两表结构相同，都是MergeTree引擎表  node1 :) **create table table\_source (id UInt8,name String,age UInt8,loc String) engine = MergeTree() order by id partition by loc;**  node1 :) **create table table\_dest (id UInt8,name String,age UInt8,loc String) engine = MergeTree() order by id partition by loc;**  #向table\_source 表中插入以下数据  node1 :) **insert into table\_source values (1,'张三',18,'BJ'),(2,'李四',19,'GZ'),(3,'王五',20,'BJ'),(4,'马六',21,'GZ');**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 1 │ 张三 │ 18 │ BJ │*  *│ 3 │ 王五 │ 20 │ BJ │*  *└───┴────┴────┴────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 2 │ 李四 │ 19 │ GZ │*  *│ 4 │ 马六 │ 21 │ GZ │*  *└───┴────┴────┴────┘*  #向table\_dest 表中插入以下数据：  node1 :) **insert into table\_dest values (5,'田七',22,'BJ'),(6,'赵八',23,'GZ'),(7,'李九',24,'BJ'),(8,'郑十',25,'GZ');**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 5 │ 田七 │ 22 │ BJ │*  *│ 7 │ 李九 │ 24 │ BJ │*  *└───┴────┴────┴────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 6 │ 赵八 │ 23 │ GZ │*  *│ 8 │ 郑十 │ 25 │ GZ │*  *└───┴────┴────┴────┘*  #将表 table\_source 中的分区‘BJ’的数据移动到 table\_dest表中  node1 :) **alter table table\_source move partition 'BJ' to table table\_dest;**  #查看表 table\_source中的数据  node1 :) **select \* from table\_source;**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 1 │ 张三 │ 18 │ BJ │*  *│ 3 │ 王五 │ 20 │ BJ │*  *└────┴──────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 2 │ 李四 │ 19 │ GZ │*  *│ 4 │ 马六 │ 21 │ GZ │*  *└────┴──────┴─────┴─────┘*  #查看表 table\_dest中的数据  node1 :) **select \* from table\_dest;**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 6 │ 赵八 │ 23 │ GZ │*  *│ 8 │ 郑十 │ 25 │ GZ │*  *└────┴──────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 5 │ 田七 │ 22 │ BJ │*  *│ 7 │ 李九 │ 24 │ BJ │*  *└────┴──────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 1 │ 张三 │ 18 │ BJ │*  *│ 3 │ 王五 │ 20 │ BJ │*  *└────┴──────┴─────┴─────┘*  #手动执行 optimize 命令，合并table\_dest相同分区数据  node1 :) **optimize table table\_dest;**  #查询表 table\_dest中的数据  node1 :) **select \* from table\_dest;**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 1 │ 张三 │ 18 │ BJ │*  *│ 3 │ 王五 │ 20 │ BJ │*  *│ 5 │ 田七 │ 22 │ BJ │*  *│ 7 │ 李九 │ 24 │ BJ │*  *└────┴──────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 6 │ 赵八 │ 23 │ GZ │*  *│ 8 │ 郑十 │ 25 │ GZ │*  *└────┴──────┴─────┴─────┘*  #查看 table\_source 表中的数据，分区‘BJ’被删除。  node1 :) **select \* from table\_source;**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 2 │ 李四 │ 19 │ GZ │*  *│ 4 │ 马六 │ 21 │ GZ │*  *└────┴──────┴─────┴─────┘* |

##### 重置分区列

重置指定分区的特定列的值，就是将指定分区下某列的数据清空，如果建表时使用了 DEFAULT 语句，该操作会将列的值重置为该默认值。

* 重置分区列语法：

|  |
| --- |
| **ALTER TABLE table\_name CLEAR COLUMN column\_name IN PARTITION partition\_expr** |

* 示例：

|  |
| --- |
| #针对之前的表 table\_dest中的数据进行操作，清空当前表中 ‘BJ’分区中name列  node1 :) **alter table table\_dest clear column name in partition 'BJ';**  #查看表 table\_dest中的数据  node1 :) **select \* from table\_dest;**  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 1 │ │ 18 │ BJ │*  *│ 3 │ │ 20 │ BJ │*  *│ 5 │ │ 22 │ BJ │*  *│ 7 │ │ 24 │ BJ │*  *└────┴──────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┬─loc─┐*  *│ 6 │ 赵八 │ 23 │ GZ │*  *│ 8 │ 郑十 │ 25 │ GZ │*  *└────┴──────┴─────┴─────┘* |

#### **临时表**

ClickHouse支持临时表，临时表具备以下特征：

* 当会话结束或者链接中断时，临时表将随会话一起消失。
* 临时表仅能够使用Memory表引擎，创建临时表时不需要指定表引擎。
* 无法为临时表指定数据库。它是在数据库之外创建的，与会话绑定。
* 如果临时表与另一个表名称相同，那么当在查询时没有显式的指定db的情况下，将优先使用临时表。
* 对于分布式处理，查询中使用的临时表将被传递到远程服务器。
* 创建一个临时表：

|  |
| --- |
| **CREATE TEMPORARY TABLE [IF NOT EXISTS] table\_name [ON CLUSTER cluster]**  **(**  **name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],**  **name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],**  **...**  **)**  **注意：不需要指定表引擎，默认是Memory** |

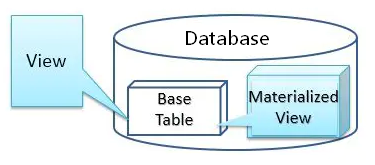
* 示例：

|  |
| --- |
| #查看库 newdb下 表  node1 :) **show tables;**  *SHOW TABLES*  *┌─name────────┐*  *│ t1 │*  *│ t2 │*  *│ t\_log │*  *│ t\_stripelog │*  *│ t\_tinylog │*  *└─────────────┘*  *5 rows in set. Elapsed: 0.004 sec.*  #查询表 t\_log表数据  node1 :) **select \* from t\_log;**  *SELECT \**  *FROM t\_log*  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *└────┴─────┴─────┘*  *┌─id─┬─name─┬─age─┐*  *│ 3 │ 王五 │ 20 │*  *│ 4 │ 马六 │ 21 │*  *│ 5 │ 田七 │ 22 │*  *└────┴─────┴─────┘*  *5 rows in set. Elapsed: 0.004 sec.*  #创建临时表 t\_log ，与当前库下的t\_log同名  node1 :) **create temporary table t\_log(id UInt8 ,name String);**  *CREATE TEMPORARY TABLE t\_log*  *(*  *`id` UInt8,*  *`name` String*  *)*  *Ok.*  *0 rows in set. Elapsed: 0.001 sec.*  #查询表 t\_log的数据与结构，发现没有数据，这里查询的是临时表，结构如下：  node1 :) **desc t\_log;**  *DESCRIBE TABLE t\_log*  *┌─name─┬─type───┬*  *│ id │ UInt8 │*  *│ name │ String │*  *└──────┴────────┴*  *2 rows in set. Elapsed: 0.003 sec.*  #如果想要查询到库newdb下的t\_log需要加上数据库名  node1 :) **select \* from newdb.t\_log;**  #切换库为default，同样还可以查询到表t\_log，说明表不属于任何库  node1 :) **use default;**  node1 :) **desc t\_log;**  *DESCRIBE TABLE t\_log*  *┌─name─┬─type───┬*  *│ id │ UInt8 │*  *│ name │ String │*  *└──────┴────────┴*  *2 rows in set. Elapsed: 0.004 sec.*  #退出客户端之后，重新登录，查询t\_log不存在。  node1 :) **select \* from t\_log;**  *Exception: Received from localhost:9000. DB::Exception: Table default.t\_log doesn't exist..*  #也可以不退出客户端直接删除临时表  node1 :) **drop table t\_log;**  *DROP TABLE t\_log*  *Ok.*  *0 rows in set. Elapsed: 0.001 sec.* |

注意：在大多数情况下，临时表不是手动创建的，而是在使用外部数据进行查询或分布式时创建的,可以使用ENGINE = Memory的表代替临时表。

#### **视图**

ClickHouse中视图分为普通视图和物化视图，两者区别如图所示：



##### 普通视图

普通视图不存储数据，它只是一层select 查询映射，类似于表的别名或者同义词，能简化查询，对原有表的查询性能没有增强的作用，具体性能依赖视图定义的语句，当从视图中查询时，视图只是替换了映射的查询语句。普通视图当基表删除后不可用。

* 创建普通视图语法：

|  |
| --- |
| **CREATE [OR REPLACE] VIEW [IF NOT EXISTS] [db.]table\_name [ON CLUSTER] AS SELECT ...** |

* 示例：

|  |
| --- |
| #在库 newdb中创建表 personinfo  node1 :) **create table personinfo(id UInt8,name String,age UInt8,birthday Date) engine = Log;**  #向表 personinfo中插入如下数据：  node1 :) **insert into personinfo values (1,'张三',18,'2021-06-01');**  node1 :) **insert into personinfo values (2,'李四',19,'2021-06-02');**  node1 :) **insert into personinfo values (3,'王五',20,'2021-06-03');**  node1 :) **insert into personinfo values (4,'马六',21,'2021-06-04');**  node1 :) **insert into personinfo values (5,'田七',22,'2021-06-05');**  #查询表中的数据  node1 :) **select \* from personinfo;**  *SELECT \**  *FROM personinfo*  *┌─id─┬─name─┬─age─┬───birthday─┐*  *│ 1 │ 张三 │ 18 │ 2021-06-01 │*  *│ 2 │ 李四 │ 19 │ 2021-06-02 │*  *└────┴──────┴─────┴────────────┘*  *┌─id─┬─name─┬─age─┬───birthday─┐*  *│ 3 │ 王五 │ 20 │ 2021-06-03 │*  *│ 4 │ 马六 │ 21 │ 2021-06-04 │*  *│ 5 │ 田七 │ 22 │ 2021-06-05 │*  *└────┴──────┴─────┴────────────┘*  *5 rows in set. Elapsed: 0.004 sec.*  #创建视图 person\_view 映射查询子句  node1 :) **create view person\_view as select name,birthday from personinfo;**  *CREATE VIEW person\_view AS*  *SELECT*  *name,*  *birthday*  *FROM personinfo*  *Ok.*  *0 rows in set. Elapsed: 0.009 sec.*  #查询视图person\_view中的数据结果  node1 :) **select \* from person\_view;**  *SELECT \**  *FROM person\_view*  *┌─name─┬───birthday─┐*  *│ 张三 │ 2021-06-01 │*  *│ 李四 │ 2021-06-02 │*  *└──────┴────────────┘*  *┌─name─┬───birthday─┐*  *│ 王五 │ 2021-06-03 │*  *│ 马六 │ 2021-06-04 │*  *│ 田七 │ 2021-06-05 │*  *└──────┴────────────┘*  *5 rows in set. Elapsed: 0.004 sec.*  #删除视图 使用drop即可  node1 :) **drop table person\_view;**  *DROP TABLE person\_view*  *Ok.*  *0 rows in set. Elapsed: 0.002 sec.* |

##### 物化视图

**物化视图是查询结果集的一份持久化存储**，所以它与普通视图完全不同，而非常趋近于表。”查询结果集”的范围很宽泛，可以是基础表中部分数据的一份简单拷贝，也可以是多表join之后产生的结果或其子集，或者原始数据的聚合指标等等。

物化视图创建好之后，若**源表被写入新数据则物化视图也会同步更新**,POPULATE 关键字决定了物化视图的更新策略,若有POPULATE 则在创建视图的过程会将源表已经存在的数据一并导入，类似于 create table ... as,若无POPULATE 则物化视图在创建之后没有数据，只会在创建只有同步之后写入源表的数据，clickhouse 官方并不推荐使用populated，因为在创建物化视图的过程中同时写入的数据不能被插入物化视图。

物化视图是种特殊的数据表，**创建时需要指定引擎**，可以用show tables 查看。另外，物化视图不支持alter 操作。

产生物化视图的过程就叫做“物化”（materialization），广义地讲，物化视图是数据库中的预计算逻辑+显式缓存，典型的**空间换时间**思路，所以用得好的话，它可以避免对基础表的频繁查询并复用结果，从而显著提升查询的性能。

* 物化视图创建语法：

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

* 示例：

|  |
| --- |
| #在库 newdb 中创建物化视图 t\_view1  node1 :) **create materialized view t\_view1 engine = Log as select \* from personinfo;**  #查询 所有表  node1 :) **show tables;**  *SHOW TABLES*  *┌─name───────────┐*  *│ .inner.t\_view1 │*  *│ personinfo*  *└────────────────┘*  *2 rows in set. Elapsed: 0.004 sec.*  #向表 personinfo中插入如下数据：  node1 :) **insert into personinfo values (1,'张三',18,'2021-06-01');**  node1 :) **insert into personinfo values (2,'李四',19,'2021-06-02');**  node1 :) **insert into personinfo values (3,'王五',20,'2021-06-03');**  node1 :) **insert into personinfo values (4,'马六',21,'2021-06-04');**  node1 :) **insert into personinfo values (5,'田七',22,'2021-06-05');**  #查看物化视图 t\_view1数据  node1 :) **select \* from t\_view1;**  *SELECT \**  *FROM t\_view1*  *┌─id─┬─name─┬─age─┬───birthday─┐*  *│ 1 │ 张三 │ 18 │ 2021-06-01 │*  *│ 2 │ 李四 │ 19 │ 2021-06-02 │*  *└────┴──────┴─────┴────────────┘*  *┌─id─┬─name─┬─age─┬───birthday─┐*  *│ 3 │ 王五 │ 20 │ 2021-06-03 │*  *│ 4 │ 马六 │ 21 │ 2021-06-04 │*  *│ 5 │ 田七 │ 22 │ 2021-06-05 │*  *└────┴──────┴─────┴────────────┘*  *5 rows in set. Elapsed: 0.004 sec.*  #创建物化视图 t\_view2  node1 :) **create materialized view t\_view2 engine = Log as select count(name) as cnt from personinfo;**  #向表 personinfo中插入以下数据  node1 :) **insert into personinfo values (6,'赵八',23,'2021-06-06'),(7,'孙九',22,'2021-06-07');**  #查询物化视图表 t\_view2数据,可以看到做了预计算，这里不能一条条插入，不然效果是每条数据都会生成一个结果。  node1 :) **select \* from t\_view2;**  *SELECT \**  *FROM t\_view2*  *┌─cnt─┐*  *│ 2 │*  *└─────┘*  *1 rows in set. Elapsed: 0.004 sec.*  #删除物化视图  node1 :) **drop table t\_view2;**  *DROP TABLE t\_view2*  *Ok.*  *0 rows in set. Elapsed: 0.001 sec.* |

注意：当创建好物化视图t\_view1时，可以进入到/var/lib/clickhouse/data/newdb目录下看到%2Einner%2Et\_view1目录，当物化视图中同步基表数据时，目录中有对应的列文件和元数据记录文件，与普通创建表一样，有目录结构。

### **SQL DML 操作**

DML:Data Manipulation Language,数据操纵语言。ClickHouse中DML语言包含插入、更新、删除数据操作,DML操作仅适用MergeTree引擎,不能针对主键、分区键、排序键进行DML操作，DML操作不支持事务，一旦执行成功会立刻生效。

#### **Insert 插入**

insert 向表中插入数据。

* insert 语法：

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

* 示例：

|  |
| --- |
| #创建表 t\_insert ,这里使用MergeTree引擎  node1 :) **create table t\_insert (id UInt8 ,name String) engine = MergeTree() order by id ;**  #向表 t\_insert 中插入数据  node1 :) **insert into t\_insert values (1,'张三',18),(2,'李四',19);**  #向表 t\_insert 中插入数据  node1 :) **insert into t\_insert select \* from t\_insert;** |

#### **update 更新**

由于ClickHouse针对的是OLAP业务分析，Update操作在ClickHouse中不会经常使用。这种更新效率低下。

* update 更新操作语法：

|  |
| --- |
| **ALTER TABLE [db.]table UPDATE column1 = expr1 [, ...] WHERE filter\_expr** |

* 示例：

|  |
| --- |
| #创建表 t\_update,使用MergeTree引擎  node1 :) **create table t\_update (id UInt8,name String,age UInt8) engine = MergeTree() order by id ;**  #向表 t\_update中插入如下数据  node1 :) **insert into t\_update values (1,'张三',18),(2,'李四',19)**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *└────┴──────┴─────┘*  #更新姓名为 张三的年龄为 22  node1 :) **alter table t\_update update age = 22 where name = '张三';**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 22 │*  *│ 2 │ 李四 │ 19 │*  *└────┴──────┴─────┘* |

#### **delete 删除**

由于ClickHouse针对的是OLAP业务分析，Delete操作与Update操作一样在ClickHouse中不会经常使用。这种删除效率低下。

* delete 删除语法：

|  |
| --- |
| **ALTER TABLE [db.]table [ON CLUSTER cluster] DELETE WHERE filter\_expr** |

* 示例：

|  |
| --- |
| #创建表 t\_delete,使用MergeTree引擎  node1 :) **create table t\_delete (id UInt8,name String,age UInt8) engine = MergeTree() order by id ;**  #向表 t\_delete中插入以下数据  node1 :) **insert into t\_update values (1,'张三',18),(2,'李四',19)**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18│*  *│ 2 │ 李四 │ 19 │*  *└────┴──────┴─────┘*  #删除 张三 此条数据  node1 :) **alter table t\_delete delete where name = '张三';**  *┌─id─┬─name─┬─age─┐*  *│ 2 │ 李四 │ 19 │*  *└────┴──────┴─────┘* |

### **向表中导入导出数据**

ClickHouse中支持多种数据格式数据导入和导出，支持格式有ORC,Parquet,Avro,Protobuf,xml,json,csv等，具体操作参照官网：<https://clickhouse.tech/docs/en/sql-reference/statements/alter/update/。>下面以向表导入导出CSV格式数据为例操作：

* 示例：

|  |
| --- |
| #创建表 t\_csv ，执行引擎为MergeTree  node1 :) **create table t\_csv (id UInt8,name String,age UInt8) engine = MergeTree order by id;**  #在ClickHouse客户端准备数据文件 csvdata 写入以下数据  *vim /root/csvdata*  *1,张三,18*  *2,李四,19*  *3,王五,20*  *4,马六,21*  *5,田七,22*  #导入数据，在ClickHouse-client中执行导入数据命令  [root@node1 ~]# **clickhouse-client --format\_csv\_delimiter="," --query="INSERT INTO newdb.t\_csv FORMAT CSV" < /root/csvdata**  **注意：--format\_csv\_delimiter 指定分隔符**  #进入ClickHouse客户端查看表 t\_csv中的数据  node1 :) **select \* from t\_csv;**  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *│ 3 │ 王五 │ 20 │*  *│ 4 │ 马六 │ 21 │*  *│ 5 │ 田七 │ 22 │*  *└────┴────┴─────┘*  #导出数据，在ClickHouse-client中执行命令，将数据导入到result文件中  [root@node1 ~]# **clickhouse-client --format\_csv\_delimiter="|" --query="select \* from newdb.t\_csv FORMAT CSV" > /root/result**  #查看导出的结果数据  [root@node1 ~]# **cat result**  *1|"张三"|18*  *2|"李四"|19*  *3|"王五"|20*  *4|"马六"|21*  *5|"田七"|22* |

## Java 读写ClickHouse API

Java读取ClickHouse中的数据API 。

1. 首先需要加入maven依赖

|  |
| --- |
| *<!-- 连接ClickHouse需要驱动包-->* <**dependency**>  <**groupId**>ru.yandex.clickhouse</**groupId**>  <**artifactId**>clickhouse-jdbc</**artifactId**>  <**version**>0.2.4</**version**> </**dependency**> |

1. Java 读取ClickHouse单节点表数据

|  |
| --- |
| ClickHouseProperties props = **new** ClickHouseProperties(); props.setUser(**"default"**); props.setPassword(**""**); BalancedClickhouseDataSource dataSource = **new BalancedClickhouseDataSource("jdbc:clickhouse://node1:8123/default", props);** ClickHouseConnection conn = dataSource.getConnection(); ClickHouseStatement statement = conn.createStatement(); ResultSet rs = statement.executeQuery(**"select id,name,age from test"**); **while**(rs.next()){  **int** id = rs.getInt(**"id"**);  String name = rs.getString(**"name"**);  **int** age = rs.getInt(**"age"**);  System.***out***.println(**"id = "**+id+**",name = "**+name +**",age = "**+age); } |

1. Java 读取ClickHouse集群表数据

|  |
| --- |
| ClickHouseProperties props = **new** ClickHouseProperties(); props.setUser(**"default"**); props.setPassword(**""**); BalancedClickhouseDataSource dataSource = **new BalancedClickhouseDataSource("jdbc:clickhouse://node1:8123,node2:8123,node3:8123/default", props);** ClickHouseConnection conn = dataSource.getConnection(); ClickHouseStatement statement = conn.createStatement(); ResultSet rs = statement.executeQuery(**"select id,name from t\_cluster"**); **while**(rs.next()){  **int** id = rs.getInt(**"id"**);  String name = rs.getString(**"name"**);  System.***out***.println(**"id = "**+id+**",name = "**+name ); } |

1. Java向ClickHouse 表中写入数据。

|  |
| --- |
| # API 操作：  ClickHouseProperties props = **new** ClickHouseProperties(); props.setUser(**"default"**); props.setPassword(**""**); BalancedClickhouseDataSource dataSource = **new** BalancedClickhouseDataSource(**"jdbc:clickhouse://node1:8123/default"**, props); ClickHouseConnection conn = dataSource.getConnection(); ClickHouseStatement statement = conn.createStatement(); statement.execute(**"insert into test values (100,'王五',30)"**);//可以拼接批量插入多条  #查询default库下 test表 数据：  node1 :) select \* from test;  *┌──id─┬─name─┬─age─┐*  *│ 100 │ 王五 │ 30 │*  *└─────┴──────┴─────┘*  *┌─id─┬─name─┬─age─┐*  *│ 1 │ 张三 │ 18 │*  *│ 2 │ 李四 │ 19 │*  *└────┴──────┴─────┘* |

## Spark 写入 ClickHouse API

SparkCore写入ClickHouse，可以直接采用写入方式。下面案例是使用SparkSQL将结果存入ClickHouse对应的表中。**在ClickHouse中需要预先创建好对应的结果表。**

1. 导入依赖

|  |
| --- |
| *<!-- 连接ClickHouse需要驱动包-->* <**dependency**>  <**groupId**>ru.yandex.clickhouse</**groupId**>  <**artifactId**>clickhouse-jdbc</**artifactId**>  <**version**>0.2.4</**version**>  ***<!-- 去除与Spark 冲突的包 -->***<**exclusions**>  <**exclusion**>  <**groupId**>com.fasterxml.jackson.core</**groupId**>  <**artifactId**>jackson-databind</**artifactId**>  </**exclusion**>  <**exclusion**>  <**groupId**>net.jpountz.lz4</**groupId**>  <**artifactId**>lz4</**artifactId**>  </**exclusion**> </**exclusions**> </**dependency**>  *<!-- Spark-core -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-core\_2.11</**artifactId**>  <**version**>2.3.1</**version**> </**dependency**> *<!-- SparkSQL -->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-sql\_2.11</**artifactId**>  <**version**>2.3.1</**version**> </**dependency**> *<!-- SparkSQL ON Hive-->* <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-hive\_2.11</**artifactId**>  <**version**>2.3.1</**version**> </**dependency**> |

1. 代码编写：

|  |
| --- |
| **val** session: SparkSession = SparkSession.*builder*().master(**"local"**).appName(**"test"**).getOrCreate() **val** jsonList = *List*[String](  **"{\"id\":1,\"name\":\"张三\",\"age\":18}"**,  **"{\"id\":2,\"name\":\"李四\",\"age\":19}"**,  **"{\"id\":3,\"name\":\"王五\",\"age\":20}"** )  *//将jsonList数据转换成DataSet* **import** session.implicits.\_ **val** ds: Dataset[String] = jsonList.toDS()  **val** df: DataFrame = session.read.json(ds) df.show()  *//将结果写往ClickHouse* **val** url = **"jdbc:clickhouse://node1:8123/default" val** table = **"test" val** properties = **new** Properties() properties.put(**"driver"**, **"ru.yandex.clickhouse.ClickHouseDriver"**) properties.put(**"user"**, **"default"**) properties.put(**"password"**, **""**) properties.put(**"socket\_timeout"**, **"300000"**) df.write.mode(SaveMode.*Append*).option(JDBCOptions.*JDBC\_BATCH\_INSERT\_SIZE*, 100000).jdbc(url, table, properties) |

## Flink 写入 ClickHouse API

可以通过Flink原生JDBC Connector包将Flink结果写入ClickHouse中，Flink在1.11.0版本对其JDBC Connnector进行了重构：

* 重构之前（1.10.x 及之前版本），包名为 flink-jdbc 。
* 重构之后（1.11.x 及之后版本），包名为 flink-connector-jdbc 。

二者对 Flink 中以不同方式写入 ClickHouse Sink 的支持情况如下：

|  |  |  |
| --- | --- | --- |
| **API名称** | **flink-jdbc** | **flink-connector-jdbc** |
| DataStream | 不支持 | 支持 |
| Table API | 支持 | 不支持 |

### **Flink 1.10.x之前版本使用flink-jdbc,只支持Table API**

* 示例:

1. maven中需要导入以下包：

|  |
| --- |
| *<!--添加 Flink Table API 相关的依赖 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner-blink\_2.11</**artifactId**>  <**version**>1.9.1</**version**> </**dependency**> <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-scala-bridge\_2.11</**artifactId**>  <**version**>1.9.1</**version**> </**dependency**> <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-common</**artifactId**>  <**version**>1.9.1</**version**> </**dependency**>  *<!--添加 Flink JDBC 以及 Clickhouse JDBC Driver 相关的依赖 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-jdbc\_2.11</**artifactId**>  <**version**>1.9.1</**version**> </**dependency**> <**dependency**>  <**groupId**>ru.yandex.clickhouse</**groupId**>  <**artifactId**>clickhouse-jdbc</**artifactId**>  <**version**>0.2.4</**version**> </**dependency**> |

1. 代码：

|  |
| --- |
| */\*\*  \* 通过 flink-jdbc API 将 Flink 数据结果写入到ClickHouse中，只支持Table API  \*  \* 注意：  \* 1.由于 ClickHouse 单次插入的延迟比较高，我们需要设置 BatchSize 来批量插入数据，提高性能。  \* 2.在 JDBCAppendTableSink 的实现中，若最后一批数据的数目不足 BatchSize，则不会插入剩余数据。  \*/* **case class** PersonInfo(id:Int,name:String,age:Int)  **object** FlinkWriteToClickHouse1 {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment  //设置并行度为1，后期每个并行度满批次需要的条数时，会插入click中* env.setParallelism(1)  **val** settings: EnvironmentSettings = EnvironmentSettings.*newInstance*().inStreamingMode().useBlinkPlanner().build()  **val** tableEnv: StreamTableEnvironment = StreamTableEnvironment.*create*(env,settings)   *//导入隐式转换* **import** org.apache.flink.streaming.api.scala.\_   *//读取Socket中的数据* **val** sourceDS: DataStream[String] = env.socketTextStream(**"node5"**,9999)  **val** ds: DataStream[PersonInfo] = sourceDS.map(line => {  **val** arr: Array[String] = line.split(**","**)  *PersonInfo*(arr(0).toInt, arr(1), arr(2).toInt)  })   *//将 ds 转换成 table 对象* **import** org.apache.flink.table.api.scala.\_  **val** table: Table = tableEnv.fromDataStream(ds,**'id**,**'name**,**'age**)   *//将table 对象写入ClickHouse中  //需要在ClickHouse中创建表:create table flink\_result(id Int,name String,age Int) engine = MergeTree() order by id;* **val** insertIntoCkSql = **"insert into flink\_result (id,name,age) values (?,?,?)"** *//准备ClickHouse table sink* **val** sink: JDBCAppendTableSink = JDBCAppendTableSink.*builder*()  .setDrivername(**"ru.yandex.clickhouse.ClickHouseDriver"**)  .setDBUrl(**"jdbc:clickhouse://node1:8123/default"**)  .setUsername(**"default"**)  .setPassword(**""**)  .setQuery(insertIntoCkSql)  .setBatchSize(2) *//设置批次量,默认5000条* .setParameterTypes(Types.*INT*, Types.*STRING*, Types.*INT*)  .build()   *//注册ClickHouse table Sink，设置sink 数据的字段及Schema信息* tableEnv.registerTableSink(**"ck-sink"**,  sink.configure(*Array*(**"id"**, **"name"**, **"age"**),*Array*(Types.*INT*, Types.*STRING*, Types.*INT*)))   *//将数据插入到 ClickHouse Sink 中* tableEnv.insertInto(table,**"ck-sink"**)   *//触发以上执行* env.execute(**"Flink Table API to ClickHouse Example"**)   } } |

### **Flink 1.11.x之后版本使用flink-connector-jdbc,只支持DataStream API**

* 示例:

1. 在Maven中导入以下依赖包

|  |
| --- |
| *<!-- Flink1.11 后需要 Flink-client包-->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-clients\_2.11</**artifactId**>  <**version**>1.11.3</**version**> </**dependency**>  *<!--添加 Flink Table API 相关的依赖 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner-blink\_2.11</**artifactId**>  <**version**>1.11.3</**version**> </**dependency**> <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-scala-bridge\_2.11</**artifactId**>  <**version**>1.11.3</**version**> </**dependency**> <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-common</**artifactId**>  <**version**>1.11.3</**version**> </**dependency**>  *<!--添加 Flink JDBC Connector 以及 Clickhouse JDBC Driver 相关的依赖 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-connector-jdbc\_2.11</**artifactId**>  <**version**>1.11.3</**version**> </**dependency**> <**dependency**>  <**groupId**>ru.yandex.clickhouse</**groupId**>  <**artifactId**>clickhouse-jdbc</**artifactId**>  <**version**>0.2.4</**version**> </**dependency**> |

1. 代码

|  |
| --- |
| */\*\*  \* Flink 通过 flink-connector-jdbc 将数据写入ClickHouse ,目前只支持DataStream API  \*/* **object** FlinkWriteToClickHouse2 {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment  //设置并行度为1* env.setParallelism(1)  **import** org.apache.flink.streaming.api.scala.\_   **val** ds: DataStream[String] = env.socketTextStream(**"node5"**,9999)   **val** result: DataStream[(Int, String, Int)] = ds.map(line => {  **val** arr: Array[String] = line.split(**","**)  (arr(0).toInt, arr(1), arr(2).toInt)  })   *//准备向ClickHouse中插入数据的sql* **val** insetIntoCkSql = **"insert into flink\_result (id,name,age) values (?,?,?)"** *//设置ClickHouse Sink* **val** ckSink: SinkFunction[(Int, String, Int)] = JdbcSink.*sink*(  *//插入数据SQL* insetIntoCkSql,   *//设置插入ClickHouse数据的参数* **new** JdbcStatementBuilder[(Int, String, Int)] {  **override def** accept(ps: PreparedStatement, tp: (Int, String, Int)): Unit = {  ps.setInt(1, tp.\_1)  ps.setString(2, tp.\_2)  ps.setInt(3, tp.\_3)  }  },  *//设置批次插入数据* **new** JdbcExecutionOptions.Builder().withBatchSize(5).build(),   *//设置连接ClickHouse的配置* **new** JdbcConnectionOptions.JdbcConnectionOptionsBuilder()  .withDriverName(**"ru.yandex.clickhouse.ClickHouseDriver"**)  .withUrl(**"jdbc:clickhouse://node1:8123/default"**)  .withUsername(**"default"**)  .withUsername(**""**)  .build()  )   *//针对数据加入sink* result.addSink(ckSink)   env.execute(**"Flink DataStream to ClickHouse Example"**)  } } |

## ClickHouse 可视化工具操作

### **tabix**

tabix支持通过浏览器直接连接 ClickHouse，不需要安装其他软件，就可以访问ClickHouse。有两种使用方式，一种是直接浏览器访问配置。另一种是使用ClickHouse内嵌方式。tabix具有以下特点：

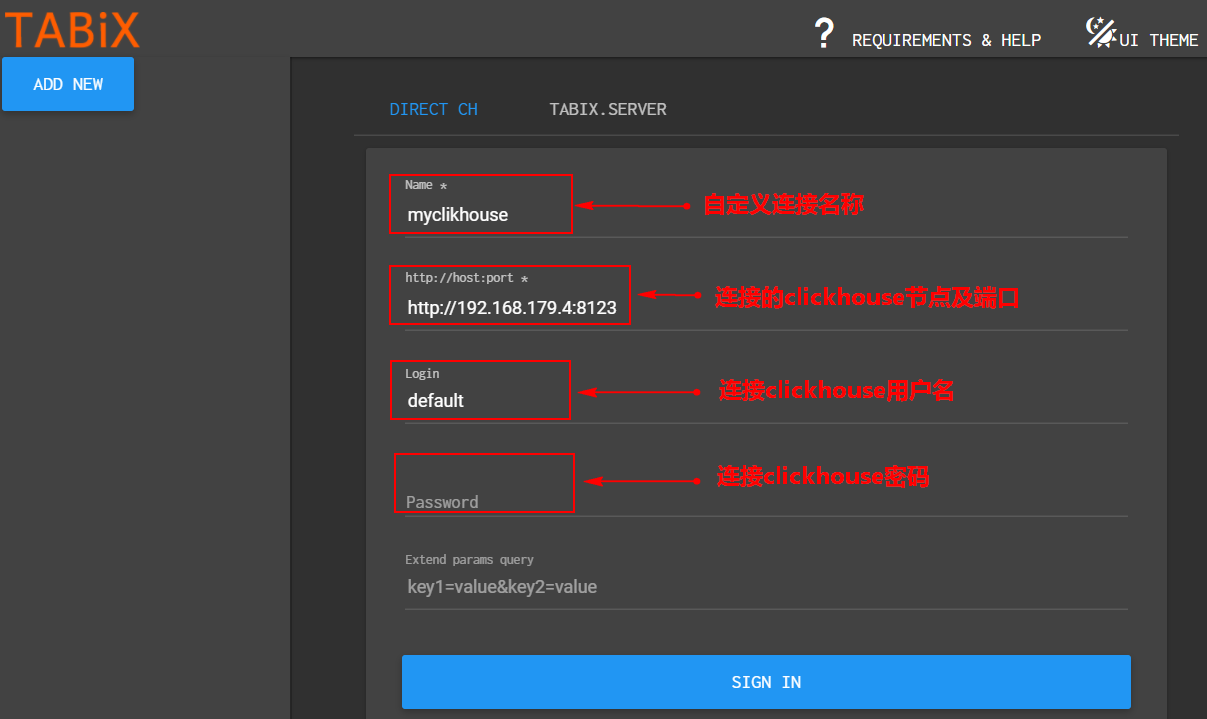
* ⾼亮语法的编辑器。
* ⾃动命令补全。
* 查询命令执⾏的图形分析⼯具。
* 配⾊⽅案选项。

#### **直接浏览器访问**

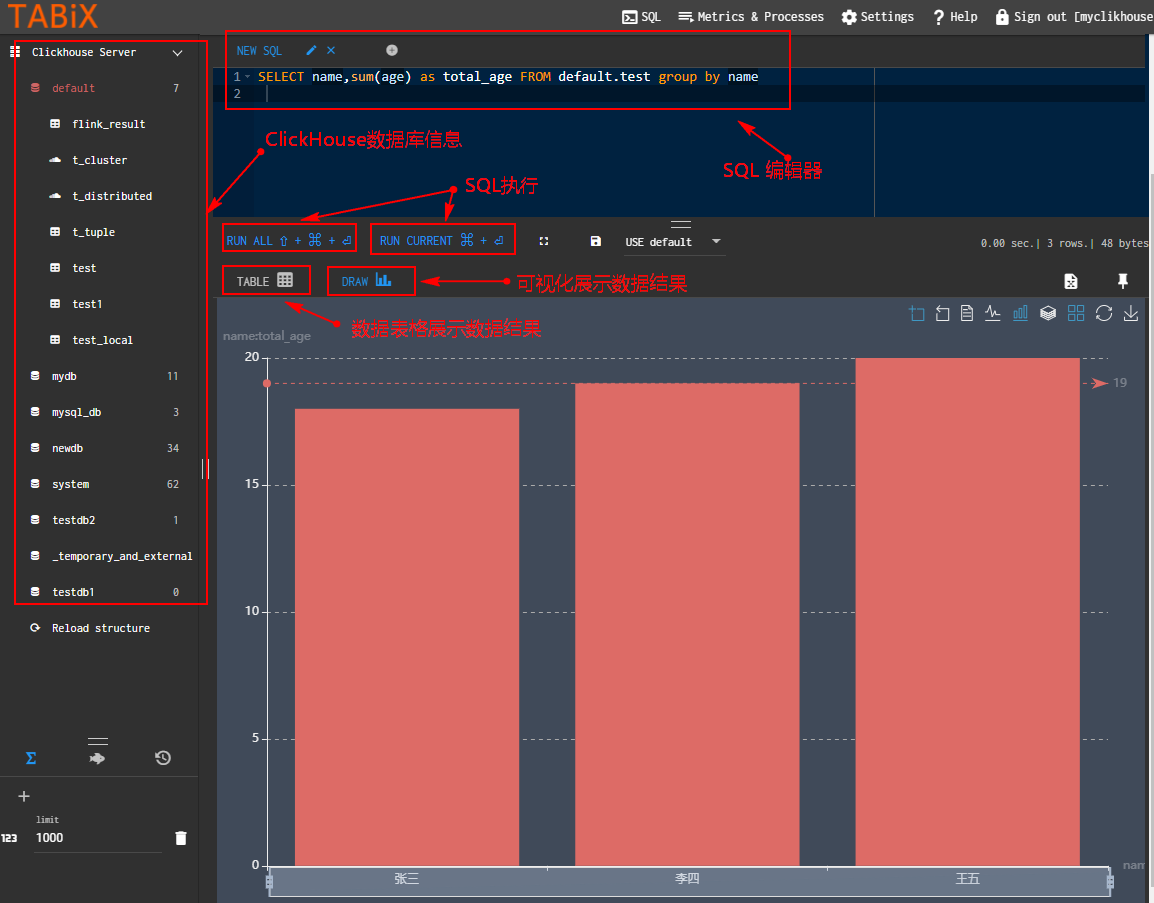
可以直接在浏览器中输入 <http://ui.tabix.io/，配置ClickHouse，通过tabix>来界面化查询ClickHouse。

1. 打开http://ui.tabix.io/，配置ClickHouse

连接ClickHouse的用户名默认为default,这里密码默认为空。访问ClickHouse端口默认为8123。



1. 编写SQL进行查询



#### **ClickHouse内嵌方式**

ClickHouse自带了配置连接tabix，这里通过ClickHouse Server节点访问http://ui.tabix.io/网址进行ClickHouse界面化操作，可以进入ClickHouse Server节点路径/etc/clickhouse-server，配置当前路径下的config.xml文件98行，解开“<http\_server\_default\_response>”标签即可：



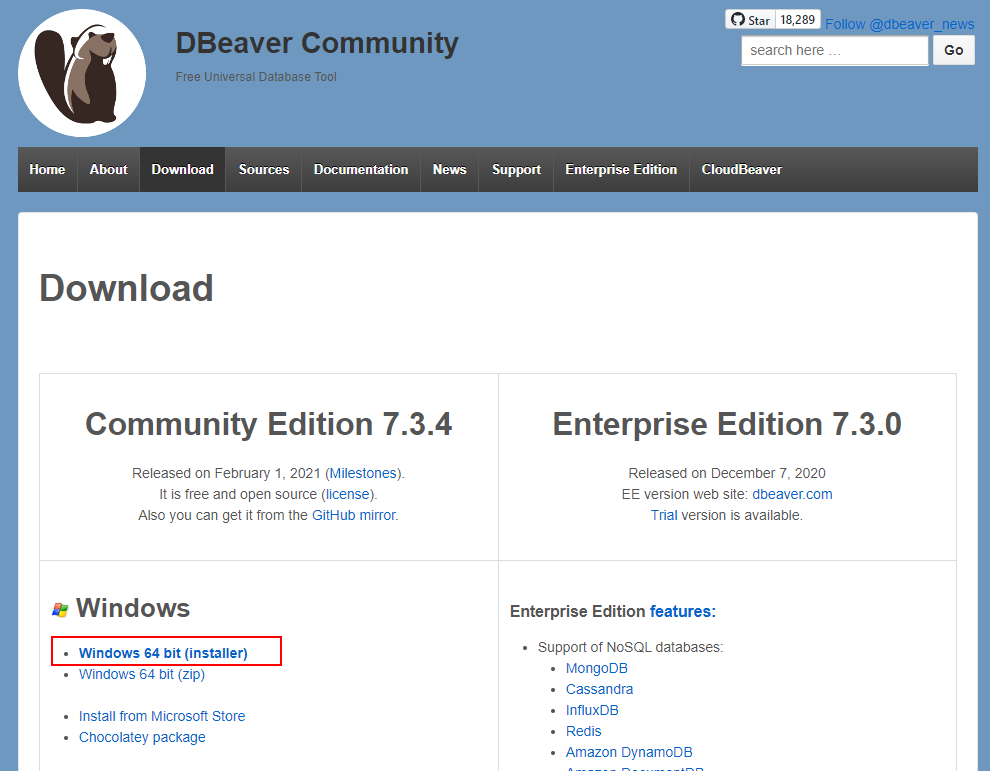
重启ClickHouse Server，可以通过http://node1:8123来访问 tabix,操作同上。

### **DBeaver**

DBeaver是一个SQL客户端和数据库管理工具。对于关系数据库，它使用JDBC API通过JDBC驱动程序与数据库交互。对于其他数据库NoSQL，它使用专有数据库驱动程序。DBeaver支持非常丰富的数据库,开箱即用,支持的数据库产品主要包含：MySQL、MariaDB、PostgreSQL、Microsoft SQL Server、Oracle、DB2、Greenplum、Teradata、PrestoDB、ClickHouse、Vertica、MongoDB、Redshift、Apache Hive、Apache Phoenix、Apache Impala、InfluxDB、Apache Cassandra、Redis、Neo4j、Elasticsearch、Solr等。

#### **DBeaver下载安装**

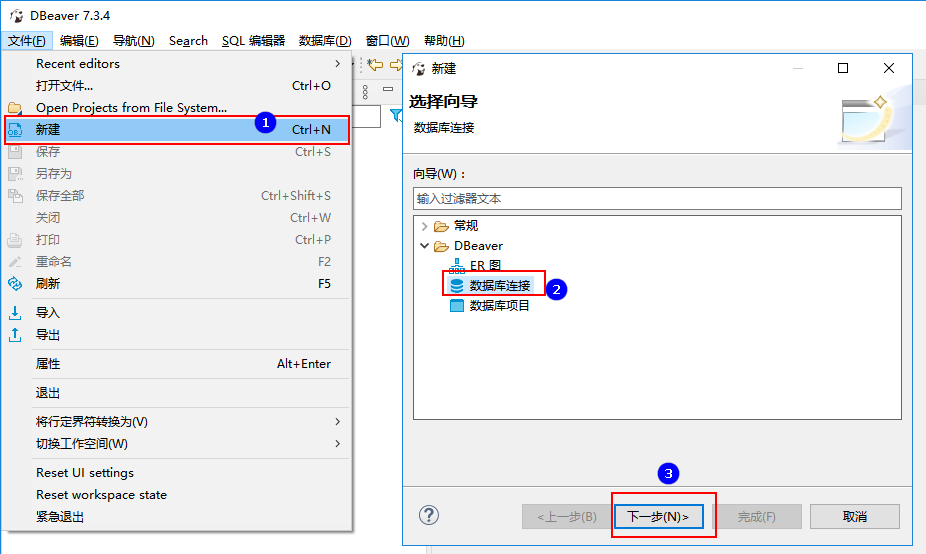
DBeaver支持在Window、MacOS和Linux上安装，这里基于Window安装演示。DBeaver下载地址：<https://dbeaver.io/download/>：

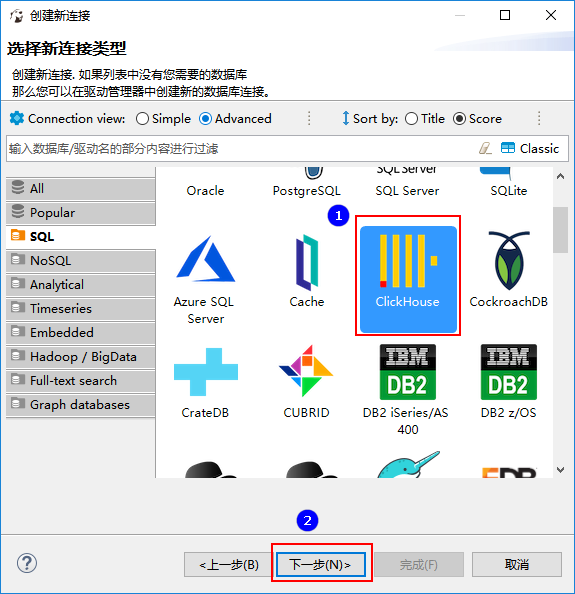


下载好之后直接安装。

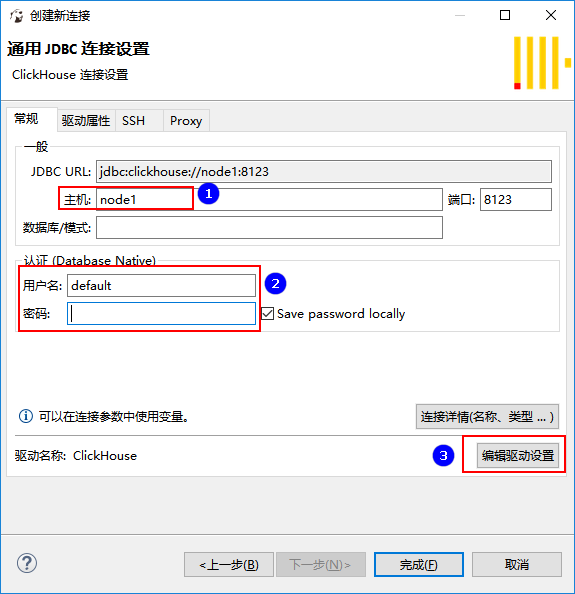
#### **DBeaver连接ClickHouse配置**

1. 打开DBeaver,新建连接，选择ClickHouse:

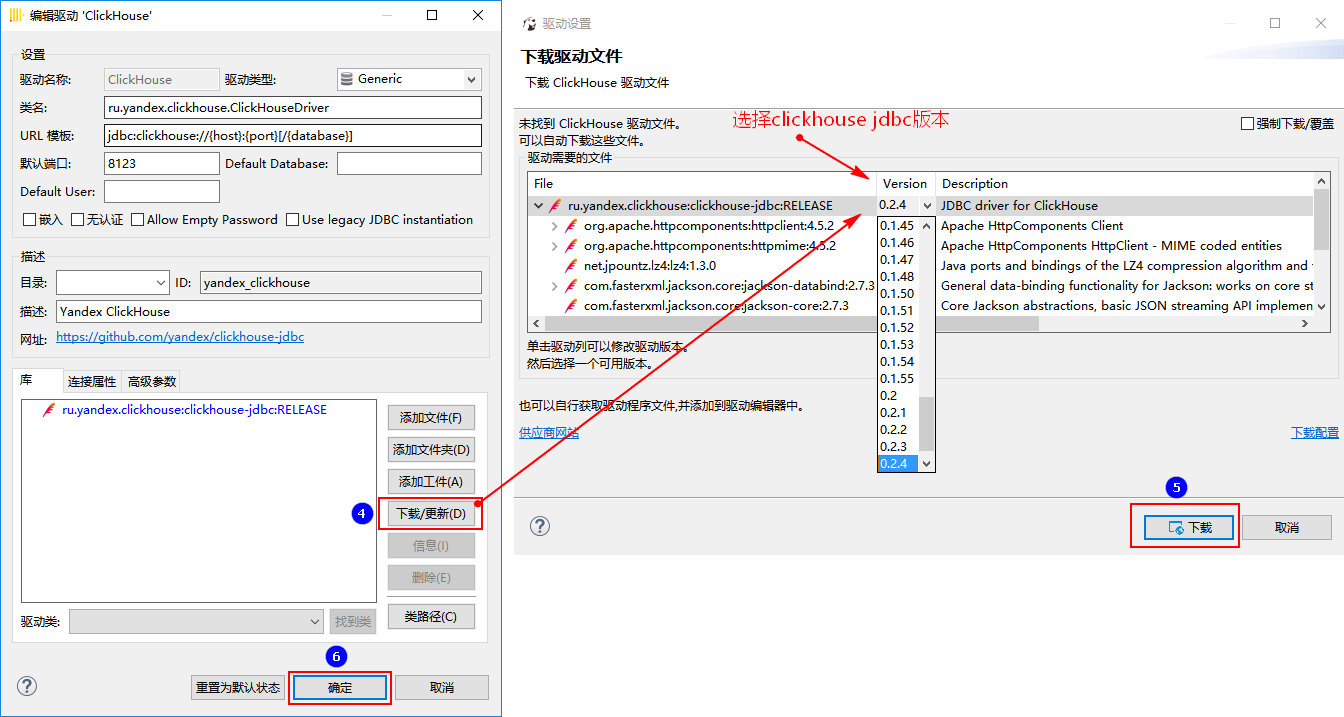




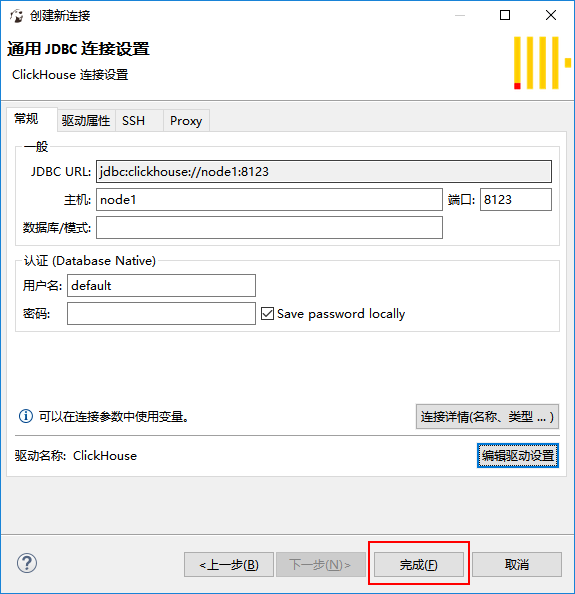
1. 点击“下一步”，设置JDBC连接，配置主机，用户名和密码:



1. 点击“编辑驱动设置”，配置ClickHouse驱动包,下载完成后，点击“确定”。



1. 点击“完成”，完成ClickHouse JDBC配置。



1. 点击“完成”后，可以使用DBeaver操作ClickHouse。

