

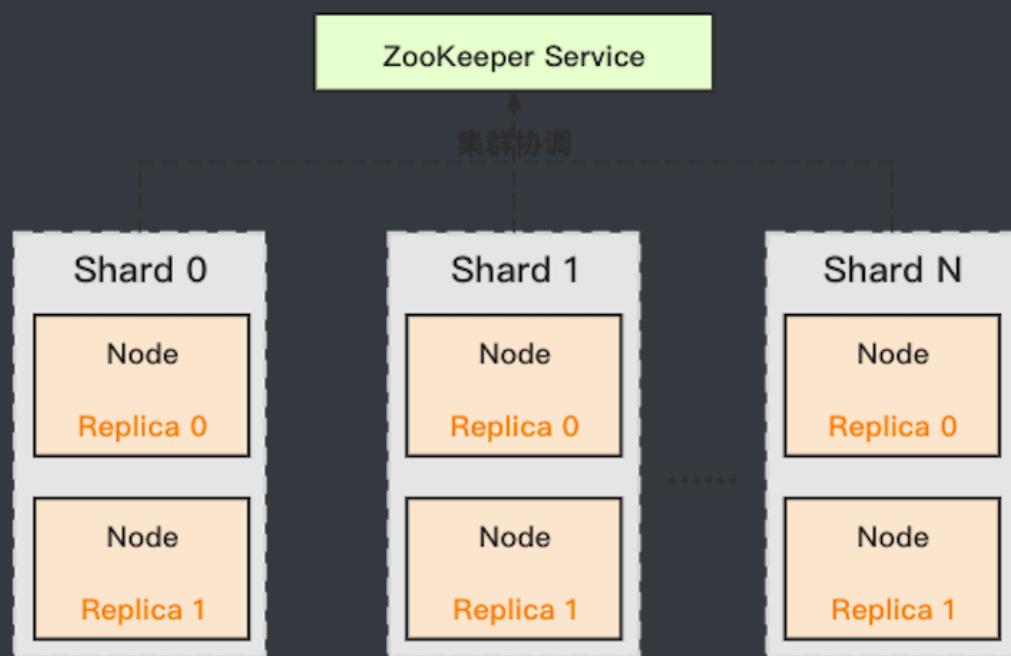
一、简单介绍Clickhouse

ClickHouse是一个面向联机分析处理(OLAP)的开源的面向列式存储的数据库管理系统，简称CK，与Hadoop, Spark相比，ClickHouse很轻量级,由俄罗斯第一大搜索引擎Yandex于2016年6月发布, 开发语言为C++

有以下特点:

- 开源: 开源的列存储数据库管理系统，支持线性扩展，简单方便，高可靠性，
- 速度快: 比Vertica快5倍，比Hive快279倍，比MySQL快800倍,其可处理的数据级别已达到10亿级别
- 功能多: 支持数据统计分析各种场景，支持类SQL查询，异地复制部署

集群架构



ClickHouse 采用典型的分组式的分布式架构，具体集群架构如上图所示：

- Shard: 集群内划分为多个分片或分组 (Shard 0 ... Shard N)，通过 Shard 的线性扩展能力，支持海量数据的分布式存储计算。
- Node: 每个 Shard 内包含一定数量的节点 (Node, 即进程)，同一 Shard 内的节点互为副本，保障数据可靠。ClickHouse 中副本数可按需建设，且逻辑上不同 Shard 内的副本数可不同。
- ZooKeeper Service: 集群所有节点对等，节点间通过 ZooKeeper 服务进行分布式协调。

二、clickhouse 表引擎

1. clickhouse 表引擎和合并树

clickhouse 表引擎

- 多样化的表引擎

目前ClickHouse拥有**合并树、外部存储、内存、文件、接口**和其他6大类20多种表引擎，每一种表引擎都有着各自的特点。

表引擎是ClickHouse的一大特色，正是由表引擎决定了一张数据表的最终特性，数据以何种形式被存储以及数据如何被加载。

- 为什么要设计多种表引擎？

将表引擎独立设计的好处是希望通过特定的表引擎支持特定的场景，提高效率。

换句话说表引擎的多样化，致使我们可以根据实际的业务场景，去选择最合适的表引擎。

- 表引擎中的核心-合并树

在ClickHouse众多的表引擎中，合并树是最为核心的部分，可以说使用ClickHouse就是在使用合并树。

只有合并树才支持主键索引、数据分区、数据副本、TTL等高级特性并拥有最强的性能。

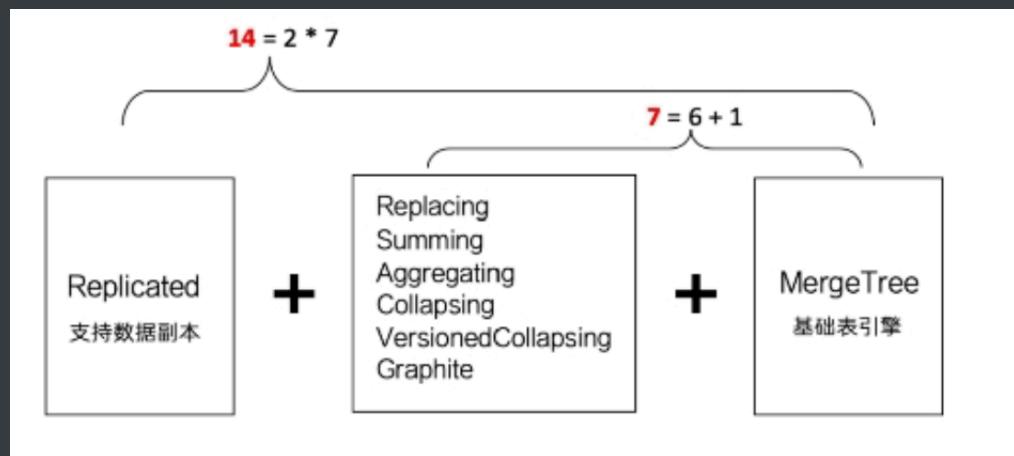
合并树MergeTree

- 为什么合并树是表引擎中的核心？

目前只有合并树系列的表引擎才支持**主键索引、数据分区、数据副本和数据采样**这些特性，也只有该系列的表引擎支持变更的相关操作。

- 合并树系列表引擎都有多少种？

通过最基础的合并树表引擎，向下派生出6个变种表引擎，搭配上Replicated（分布式协同能力）可以说就有14种和合并树相关的表引擎。



- 合并树表引擎系列的根基

虽然合并树表引擎的变种很多，但是作为最基础表引擎，合并树具备了同系列其他表引擎共有的基本特征。

在使用合并树写入一批数据时，数据总会以数据片段的形式写入磁盘，且数据片段不可修改。为了防止数据片段过多，ClickHouse会通过后台线程，定期合并这些数据片段，属于同一个分区的数据片段会被合成一个新的片段。这种数据片段不断合并的特点，就是合并树名称的由来。

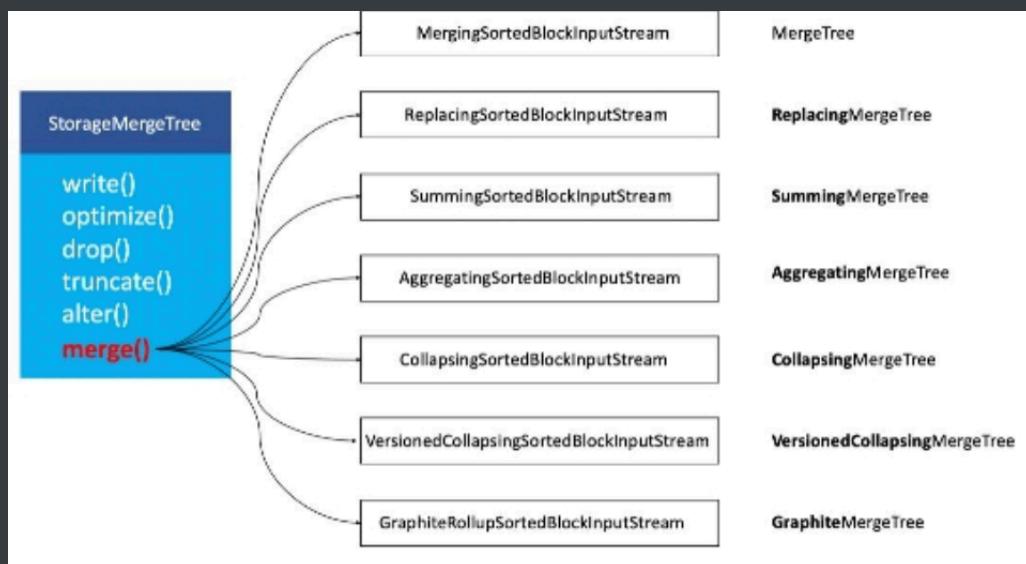
- 合并树系列表引擎的继承关系？

在表引擎底层的实现方法中，在具体的实现逻辑部分，7种合并树表引擎共用一个主体，只有在触发合并动作时，才调用了各自独有的合并逻辑。

从图中可以看到在具体的合并逻辑部分，不同类型的合并树引擎调用了各自的合并逻辑。虽然调用了不同的合并逻辑，但是不同类型的合并树合并逻辑，也都是建立最基础的合并树的 MergingSortedBlockInputStream 上的。

MergingSortedBlockInputStream 的主要作用，是按照ORDER BY的规则保持新分区数据的有序性。

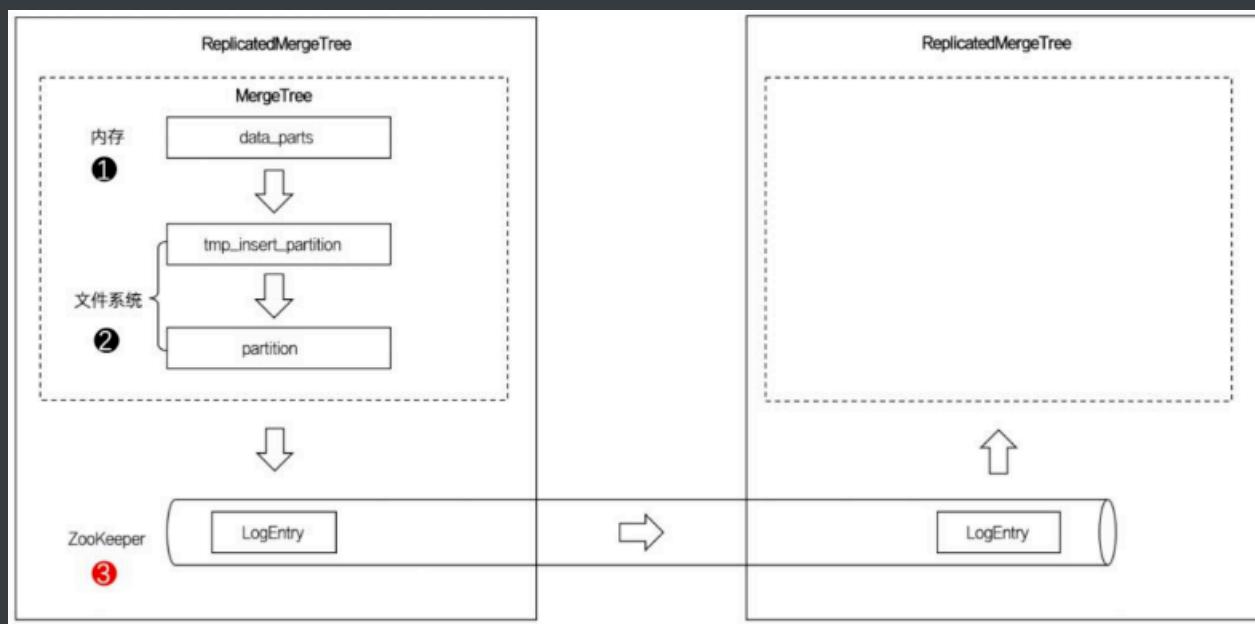
而其他6种变种合并树的合并逻辑，则是在有序的基础之上 "各有所长"，要么是将排序后相邻的重复数据消除、亦或是将它们累加汇总。



Replicated MergeTree

图中的虚线框部分是MergeTree的能力边界，而ReplicatedMergeTree在它的基础之上增加了分布式协同的能力。

借助ZooKeeper的消息日志广播，实现了副本实例之间的数据同步功能。



合并树优点

- 有序存储在压缩时能够获得更好的压缩比
- 对于等值或区间过滤可以更快地确定数据范围
- 可以在merge过程中实现特殊的合并逻辑

创建一个合并树表

```

CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    name1 [type1] [DEFAULT|MATERIALIZED|ALIAS expr1],
    name2 [type2] [DEFAULT|MATERIALIZED|ALIAS expr2],
    ...
    INDEX index_name1 expr1 TYPE type1(...) GRANULARITY value1,
    INDEX index_name2 expr2 TYPE type2(...) GRANULARITY value2
) ENGINE = MergeTree()
[PARTITION BY expr]
[ORDER BY expr]
[PRIMARY KEY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...]

```

- PARTITION BY【选填】：**分区键，用于指定数据以何种标准进行分区。分区键可以是单个列字段、元组形式的多个列字段、列表达式。如果不声明分区键，则ClickHouse会生成一个名为all的分区。合理使用数据分区，可以有效减少查询时数据文件的扫描范围
- ORDER BY【必填】：**排序键，用于指定在一个数据片段内，数据以何种标准排序。默认情况下主键（PRIMARY KEY）与排序键相同。排序键可以是单个列字段（例：ORDER BY CounterID）、元组形式的多个列字段（例：ORDER BY (CounterID,EventDate)）。当使用多个列字段排序时，以ORDER BY (CounterID,EventDate)为例，在单个数据片段内，数据

首先以CounterID排序，相同CounterID的数据再按EventDate排序

- PRIMARY KEY 【选填】：主键，生成一级索引，加速表查询。默认情况下，主键与排序键（ORDER BY）相同，所以通常使用ORDER BY代为指定主键。一般情况下，在单个数据片段内，数据与一级索引以相同的规则升序排序。与其他数据库不同，MergeTree主键允许存在重复数据
- SAMPLE BY 【选填】：抽样表达式，用于声明数据以何种标准进行采样。抽样表达式需要配合SAMPLE子查询使用
- SETTINGS: index_granularity 【选填】：索引粒度，默认值8192。也就是说，默认情况下每隔8192行数据才生成一条索引
- SETTINGS: index_granularity_bytes 【选填】：在19.11版本之前，ClickHouse只支持固定大小的索引间隔（index_granularity）。在新版本中增加了自适应间隔大小的特性，即根据每一批次写入数据的体量大小，动态划分间隔大小。而数据的体量大小，由index_granularity_bytes参数控制，默认10M
- SETTINGS: enable_mixed_granularity_parts 【选填】：设置是否开启自适应索引间隔的功能，默认开启

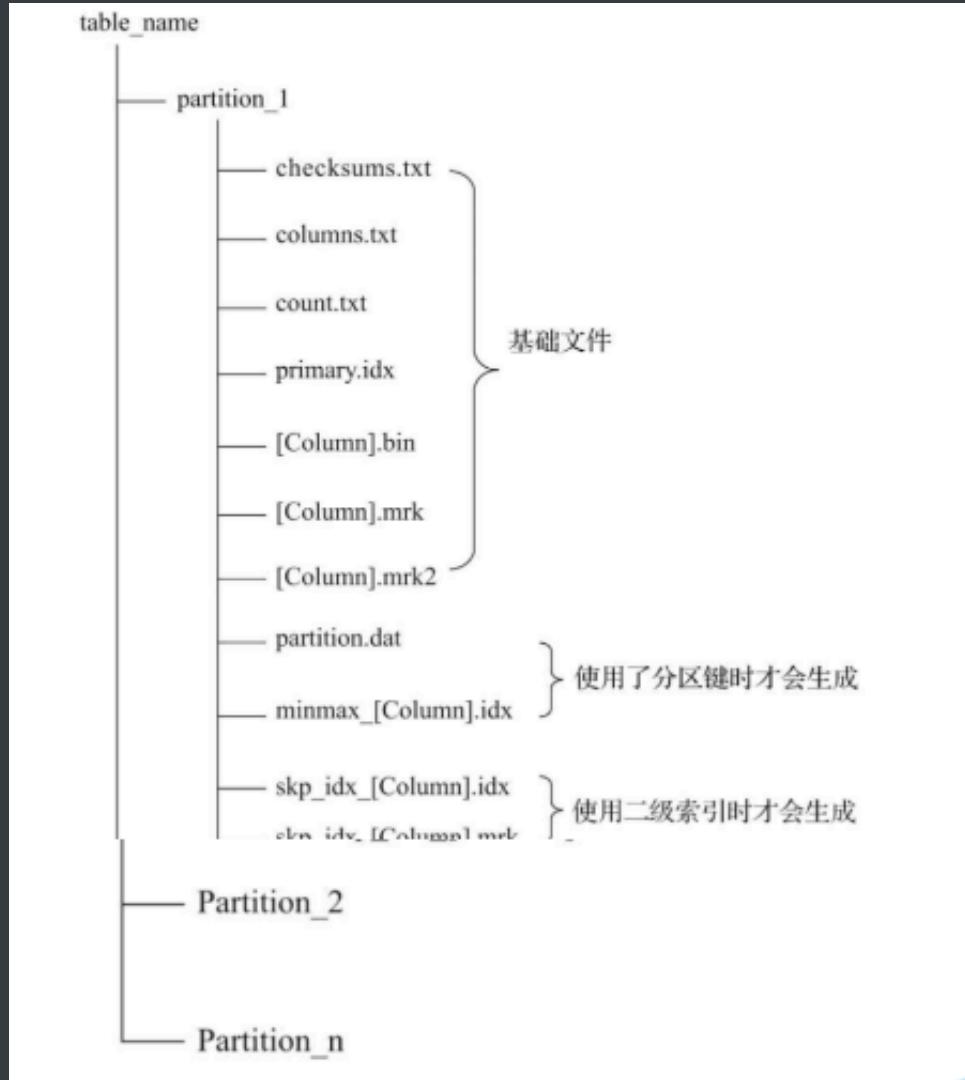
2. 数据分区

数据是以分区目录的形式进行组织存放的，每个分区独立分开存储。借助这种形式，在对MergeTree进行数据查询时，可以有效跳过无用的数据文件，只使用最小的分区目录子集。

合并树数据分区的规则由分区ID决定，而具体到每个数据分区所对应的ID，则是由分区键的取值决定的。

分区键支持使用任何一个或一组字段表达式声明，其业务语义可以是年、月、日或者组织单位等任何一种规则。针对取值数据类型的不同，分区ID的生成逻辑目前拥有四种规则：**不指定分区键、使用整型分区键、使用日期类型分区键、使用其他类型分区键**

分区表目录结构



一张数据表的完整物理结构分为3个层级，依次是数据表目录、分区目录及各分区下具体的数据文件。

- **partition**: 分区目录，相同分区的数据，最终会被合并到同一个分区目录，而不同分区的数据，不会被合并在一起。
- **checksums.txt**: 校验文件
- **columns.txt**: 列信息文件
- **count.txt**: 计数文件
- **[Column].bin**: 数据文件
- **primary.idx**: 一级索引文件
- **[Column].mrk**: 列字段标记文件
- **[Column].mrk2**: 如果使用了自适应大小的索引间隔，则标记文件会以. mrk2命名。它的工作原理和作用与. mrk标记文件相同。
- **partition.dat**与**minmax_[Column].idx**: 分区键额外的索引文件。例如EventTime字段对应的原始数据为2019-05-01、2019-05-05，分区表达式为PARTITION BY toYYYYMM(EventTime)。partition.dat中保存的值将会是2019-05，而minmax索引中保存的值将会是2019-05-01，2019-05-05。

1. **skp_idx_[Column].idx**与**skp_idx_[Column].mrk**: 如果在建表语句中声明了跳数索引，则会

额外生成相应的跳数索引与标记文件，它们同样也使用二进制存储

数据分区规则

MergeTree数据分区的规则由分区ID决定，而具体到每个数据分区所对应的ID，则是由分区键的取值决定的。分区键支持使用任何一个或一组字段表达式声明，其业务语义可以是年、月、日或者组织单位等任何一种规则。针对取值数据类型的不同，分区ID的生成逻辑目前拥有四种规则：

1. 不指定分区键：如果不使用分区键，即不使用PARTITION BY声明任何分区表达式，则分区ID默认取名为all，所有的数据都会被写入这个all分区
2. 使用整型：如果分区键取值属于整型（兼容UInt64，包括有符号整型和无符号整型），且无法转换为日期类型YYYYMMDD格式，则直接按照该整型的字符形式输出，作为分区ID的取值
3. 使用日期类型：如果分区键取值属于日期类型，或者是能够转换为YYYYMMDD格式的整型，则使用按照YYYYMMDD进行格式化后的字符形式输出，并作为分区ID的取值
4. 使用其他类型：如果分区键取值既不属于整型，也不属于日期类型，例如String、Float等，则通过128位Hash算法取其Hash值作为分区ID的取值

分区ID在不同规则下的示例如下图：

类型	样例数据	分区表达式	分区 ID
无分区键		无	all
整型	18,19,20	PARTITION BY Age	分区 1: 18 分区 2: 19 分区 3: 20
	‘A0’， ‘A1’， ‘A2’	PARTITION BY length(Code)	分区 1: 2
日期	2019-05-01, 2019-06-11	PARTITION BY EventTime	分区 1: 20190501 分区 2: 20190611
	2019-05-01, 2019-06-11	PARTITION BY toYYYYMM(EventTime)	分区 1: 201905 分区 2: 201906
其他	‘www.nauu.com’	PARTITION BY URL	分区 1: 15b31467bc77fa1c24ac9380cd8b4033 CSDN @通过的漏风到空

如果通过元组的方式使用多个分区字段，则分区ID依旧是根据上述规则生成的，只是多个ID之间通过-符号依次拼接。例如按照上述表格中的例子，使用两个字段分区：PARTITION BY (length(Code),EventTime)

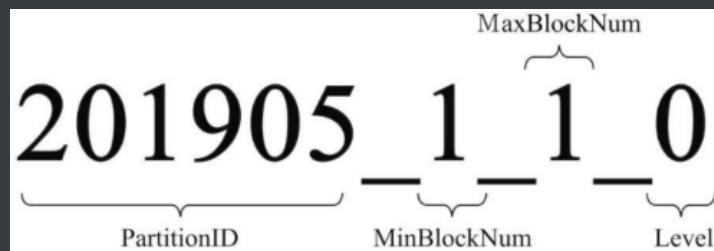
则最终的分区ID会是下面的模样：

2-20190501
2-20190611

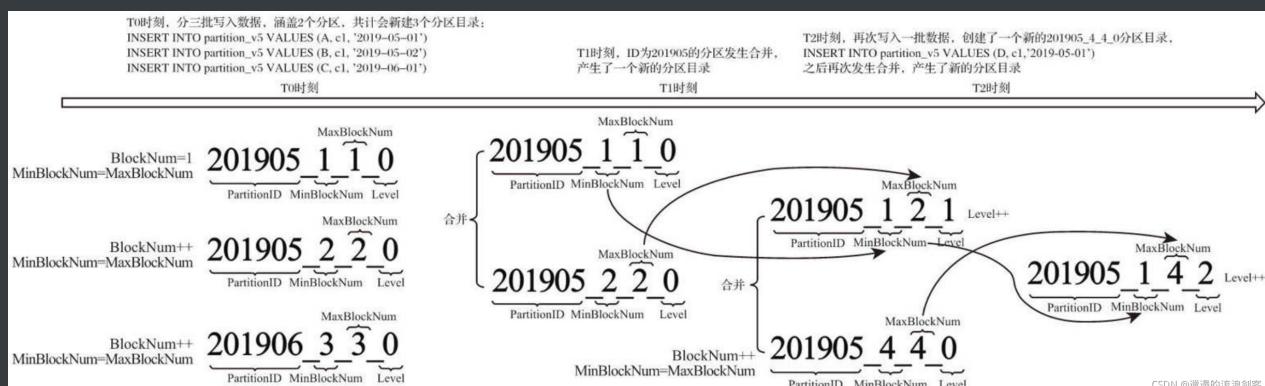
分区目录命名规则

进入数据表所在的磁盘目录后，会发现MergeTree分区目录的完整物理名称并不是只有ID而已，在ID之后还跟着一串奇怪的数字，例如201905_1_1_0。

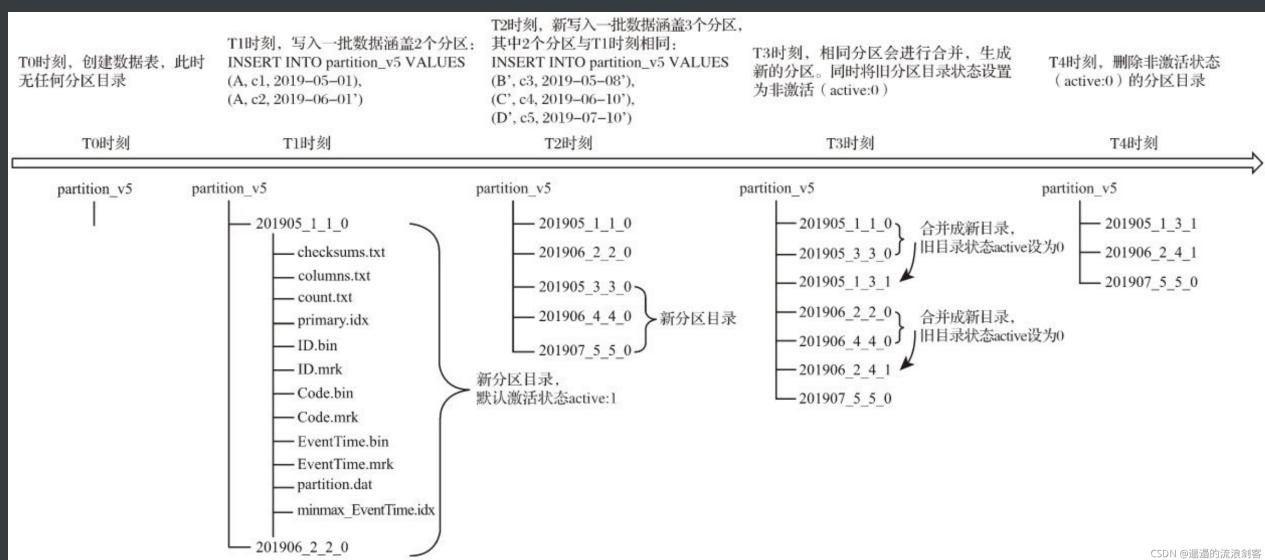
- PartitionID：分区ID
- MinBlockNum和MaxBlockNum：最小数据块编号与最大数据块编号。
- Level：合并的层级



分区目录合并过程



CSDN @遥道的浪浪剑客



CSDN @遥道的浪浪剑客

三、Clickhouse 的索引和数据存储

1. Clickhouse 索引

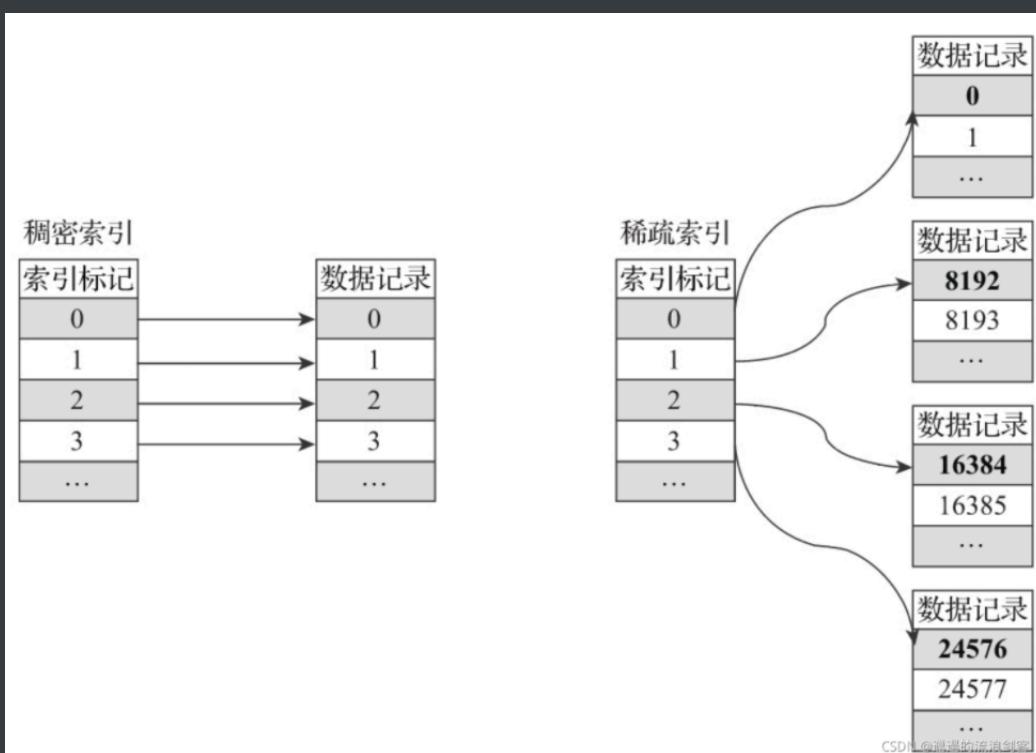
一级索引

合并树的主键使用PRIMARY KEY定义，待主键定义之后，合并树会依据settings中(index_granularity)间隔参数，生成一级索引并保存至primary.idx文件内，索引数据按照PRIMARY KEY排序。PRIMARY KEY可通过ORDER BY指代主键。在此种情形下，PRIMARY KEY与ORDER BY定义相同，所以索引(primary.idx)和数据(.bin)会按照完全相同的规则排序。

primary.idx文件内的一级索引采用稀疏索引实现。

稀疏索引

稀疏索引的优势是显而易见的，它仅需使用少量的索引标记就能够记录大量数据的区间位置信息，且数据量越大优势越为明显。以默认的索引粒度(8192)为例，MergeTree只需要12208行索引标记就能为1亿行数据记录提供索引。由于稀疏索引占用空间小，所以primary.idx内的索引数据常驻内存，取用速度自然极快。

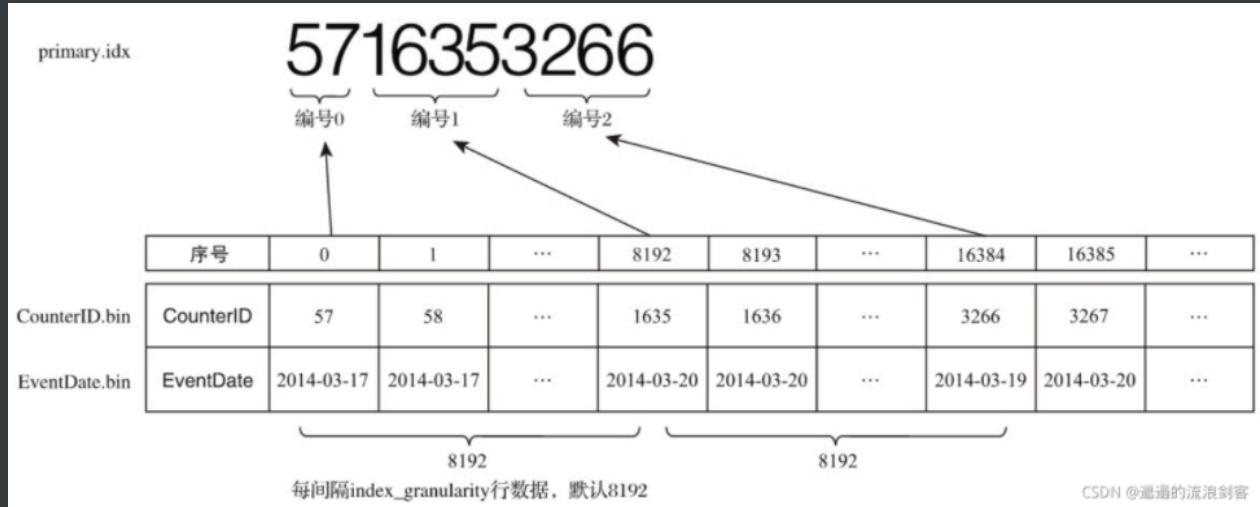


索引数据生成规则

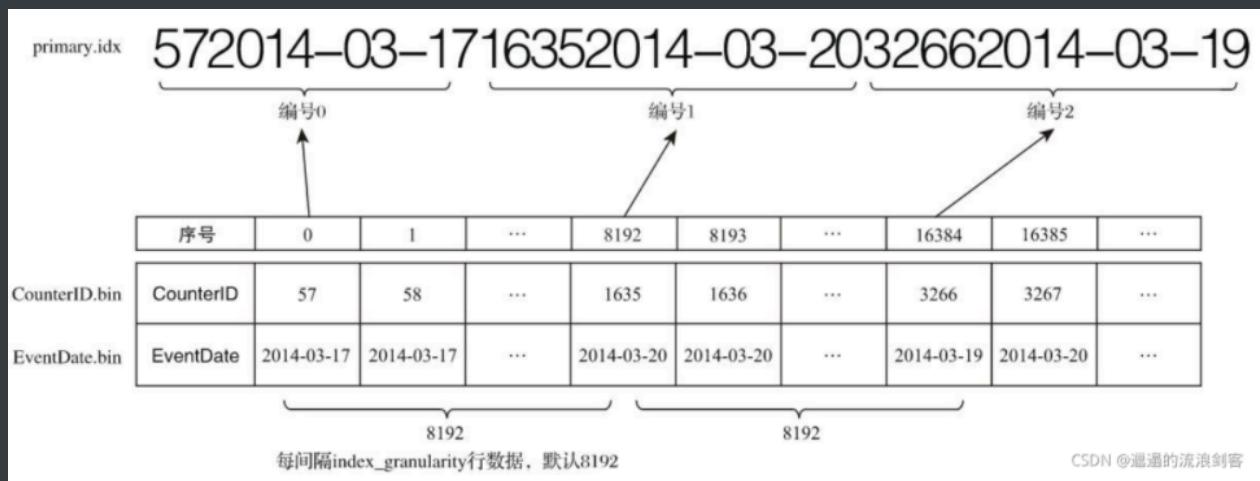
由于是稀疏索引，所以MergeTree需要间隔设定的索引粒度才会生成一条索引记录，其索引值会依据声明的主键字段获取。

如图所示表中使用CounterID作为主键，则每间隔8192行数据就会取一次CounterID的值作为索引值，索引数据最终会被写入primary.idx文件进行保存。

使用counterID作为主键



使用CounterID和EventDate作为主键



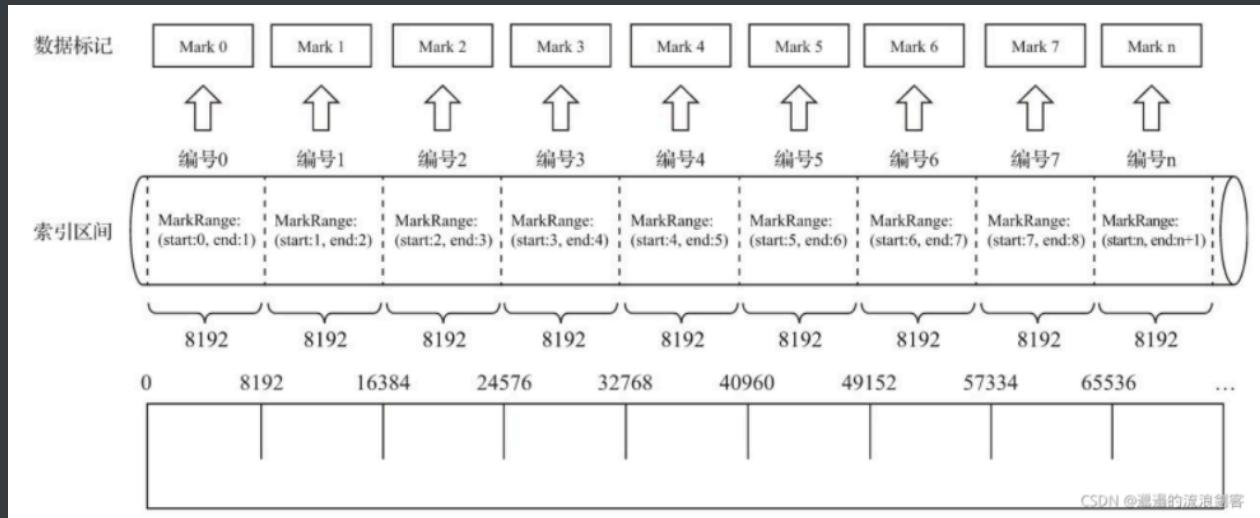
数据标记

如果把MergeTree比作一本书

primary.idx一级索引好比这本书的一级章节目录

.bin文件中的数据好比这本书中的文字

那么数据标记(.mrk)会为一级章节目录和具体的文字之间建立关联，好比目录中的页码。



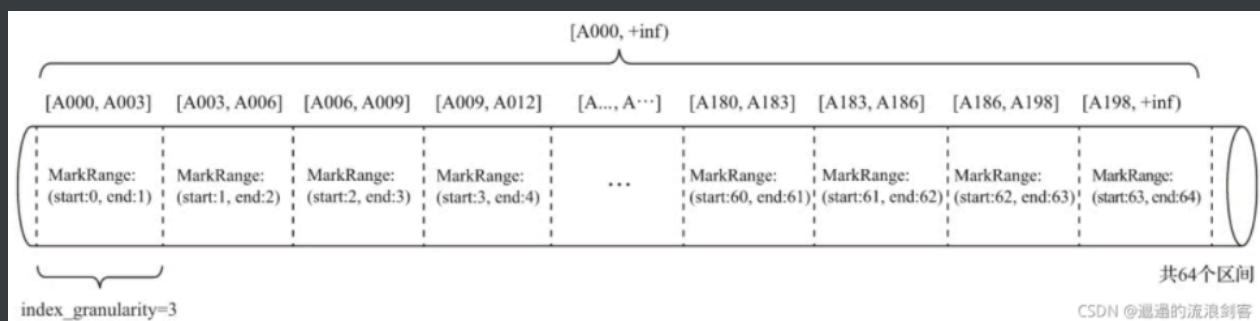
索引查询过程

假设现在有一份测试数据，共192行记录。其中，主键ID为String类型，ID的取值从A000开始，后面依次为A001、A002……直到A192为止。

MergeTree的索引粒度index_granularity=3，根据索引的生成规则，primary.idx文件内的索引数据如下图所示：



根据索引数据，假设MergeTree会将此数据片段划分为 $192/3=64$ 个小的MarkRange，两个相邻MarkRange相距的步长为1。其中，所有MarkRange（整个数据片段）的最大数值区间为[A000, +inf)，如下图所示：



想要根据索引查询数据的过程大体如下：

1、生成查询条件区间

首先，将查询条件转换为条件区间，例如下面的例子

```
WHERE ID = 'A003'  
['A003', 'A003']
```

```
WHERE ID > 'A000'  
('A000', +inf)
```

```
WHERE ID < 'A188'  
(-inf, 'A188')
```

```
WHERE ID LIKE 'A006%'  
['A006', 'A007']
```

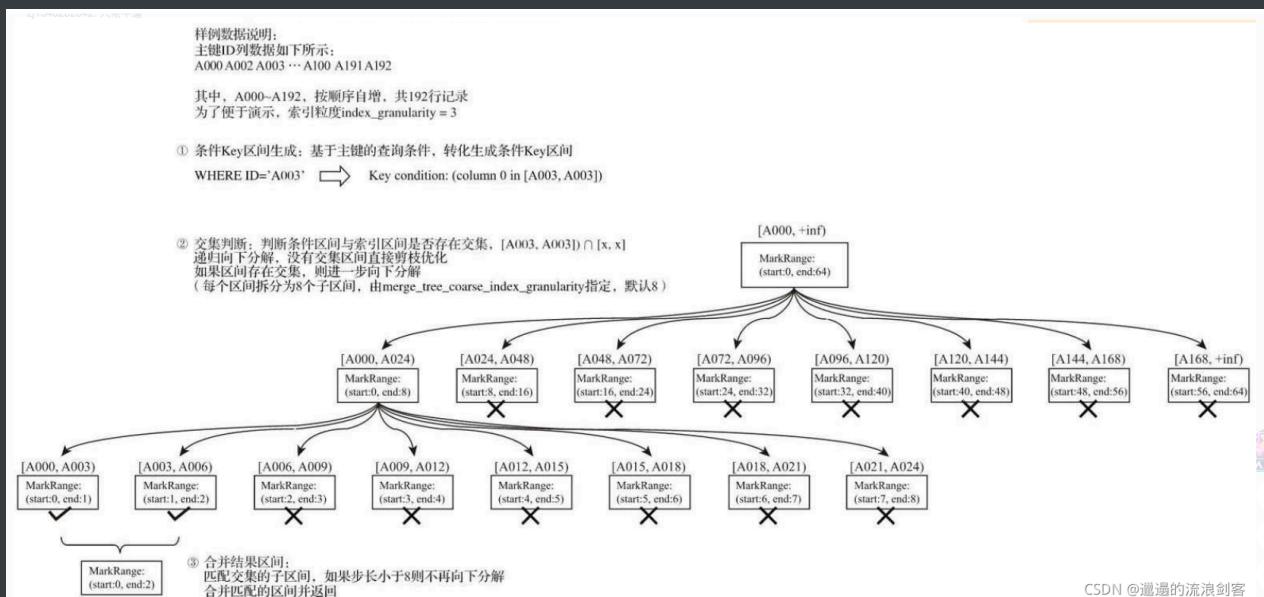
2、递归交集判断

以递归的形式，依次对MarkRange的数值区间与条件区间做交集判断。从最大区间[A000, +inf)开始。如果不存在交集，则直接通过剪枝算法优化此整段MarkRange。

如果存在交集，且MarkRange步长大于8(end-start)，则将此区间进一步拆分成8个子区间
(`merge_tree_coarse_index_granularity`指定，默认值为8)，并重复此规则，继续做递归交集判断，如果存在交集，且MarkRange不可再分解（步长小于8），则记录MarkRange并返回

3、合并MarkRange区间

将最终匹配的MarkRange聚在一起，合并它们的范围



MergeTree通过递归的形式持续向下拆分区间，最终将MarkRange定位到最细的粒度

以上图为例，当查询条件WHERE ID='A003'的时候，最终只需要读取[A000, A003]和[A003, A006]两个区间的数据，它们对应MarkRange(start:0,end:2)范围

跳数索引

除了一级索引之外，MergeTree同样支持跳数索引。由数据的聚合信息构建而成。根据索引类型的不同，其聚合信息的内容也不同。跳数索引的目的与一级索引一样，也是帮助查询时减少数据扫描的范围。跳数索引在默认情况下是关闭的。需要设置

`allow_experimental_data_skipping_indices`（该参数在新版本中已被取消）才能使用：

```
SET allow_experimental_data_skipping_indices = 1
```

与一级索引一样，如果在建表语句中声明了跳数索引，则会额外生成相应的索引与标记文件
（`skp_idx_[Column].idx` 与 `skp_idx_[Column].mrk`）

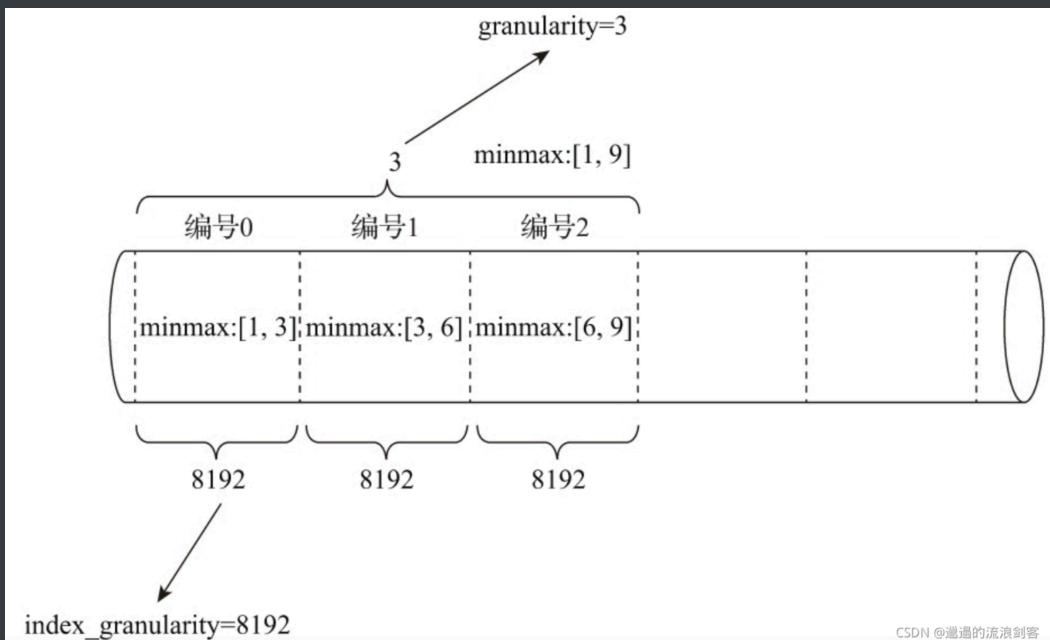
granularity与**index_granularity**的关系

对于跳数索引而言，`index_granularity`定义了数据的粒度，而`granularity`定义了聚合信息汇总的粒度。`granularity`定义了一行跳数索引能够跳过多少个`index_granularity`区间的数据

跳数索引的数据生成规则：首先，按照`index_granularity`粒度间隔将数据划分为n段，总共有[0, n-1]个区间 ($n=\text{total_rows}/\text{index_granularity}$, 向上取整)。

接着，根据索引定义时声明的表达式，从0区间开始，依次按`index_granularity`粒度从数据中获取聚合信息，每次向前移动1步，聚合信息逐步累加。最后，当移动`granularity`次区间时，则汇总并生成一行跳数索引数据

以minmax索引为例，它的聚合信息是在一个`index_granularity`区间内数据的最小和最大极值。以下图为例，假设`index_granularity=8192`且`granularity=3`，则数据会按照`index_granularity`划分为n等份，MergeTree从第0段分区开始，依次获取聚合信息。当获取到第3个分区时
(`granularity=3`)，则汇总并会生成第一行minmax索引（前3段minmax极值汇总后取值为[1, 9]）



跳数索引的类型

1) minmax: minmax索引记录了一段数据内的最小和最大极值，其索引的作用类似分区目录的minmax索引，能够快速跳过无用的数据区间

```
INDEX a ID TYPE minmax GRANULARITY 5
```

上述示例中minmax索引会记录这段数据区间内ID字段的极值。极值的计算涉及每5个index_granularity区间中的数据

2) set: set索引直接记录了声明字段或表达式的取值（唯一值，无重复），其完整形式为set(max_rows)，其中max_rows是一个阈值，表示在一个index_granularity内，索引最多记录的数据行数。如果max_rows=0，则表示无限制

```
INDEX b (length(ID) * 8) TYPE set(100) GRANULARITY 5
```

上述示例中set索引会记录数据中ID的长度*8后的取值。其中，每个index_granularity内最多记录100条

3) ngrambf_v1: ngrambf_v1索引记录的是数据短语的布隆表过滤器，只支持String和FixedString数据类型。ngrambf_v1只能提升in、notIn、like、equals和notEquals查询的性能，其完整形式为

ngrambf_v1(n,size_of_bloom_filter_in_bytes,number_of_hash_functions,random_seed)。具体的含义如下

- n: token长度，依据n的长度将数据切割为token短语
- size_of_bloom_filter_in_bytes: 布隆过滤器的大小
- number_of_hash_functions: 布隆过滤器中使用Hash函数的个数

- random_seed: Hash函数的随机种子

```
INDEX c (ID, Code) TYPE ngrambf_v1(3, 256, 2, 0) GRANULARITY 5
```

例如在上面例子中，ngrambf_v1索引会依照3的粒度将数据切割成短语token，token会经过2个Hash函数映射后再被写入，布隆过滤器大小为256字节

4) tokenbf_v1: tokenbf_v1索引是ngrambf_v1的变种，同样也是一种布隆过滤器索引。tokenbf_v1除了短语token的处理方法外，其他与ngrambf_v1是完全一样的。tokenbf_v1会自动按照非字符的、数字的字符串分割token，具体用法如下所示：

```
INDEX d ID TYPE tokenbf_v1(256, 2, 0) GRANULARITY 5
```

2. 数据存储

数据存储方式

在合并树中数据是按列存储的。

而具体到每个列字段，数据也是独立存储的，每个列字段都拥有一个与之对应的 .bin数据文件。也正是这些 .bin文件，最终承载着数据的物理存储。数据文件以分区目录的形式被组织存放，所以在 .bin文件中只会保存当前分区片段内的这一部分数据。

而对应到存储的具体实现方面，合并树也并不是一股脑地将数据直接写入 .bin文件，而是经历了一番精心设计：

1. 首先，数据是经过压缩的，默认使用LZ4算法；
2. 其次，数据会事先依照ORDER BY的声明排序；
3. 最后，数据是以压缩数据块的形式被组织写入.bin文件中的。

数据快文件头



一个压缩数据块由头信息和压缩数据两部分组成。

头信息固定使用9位字节表示，具体由1个UInt8（1字节）整型和2个UInt32（4字节）整型组成，分别代表使用的压缩算法类型、压缩后的数据大小和压缩前的数据大小。

MergeTree 数据写入方式

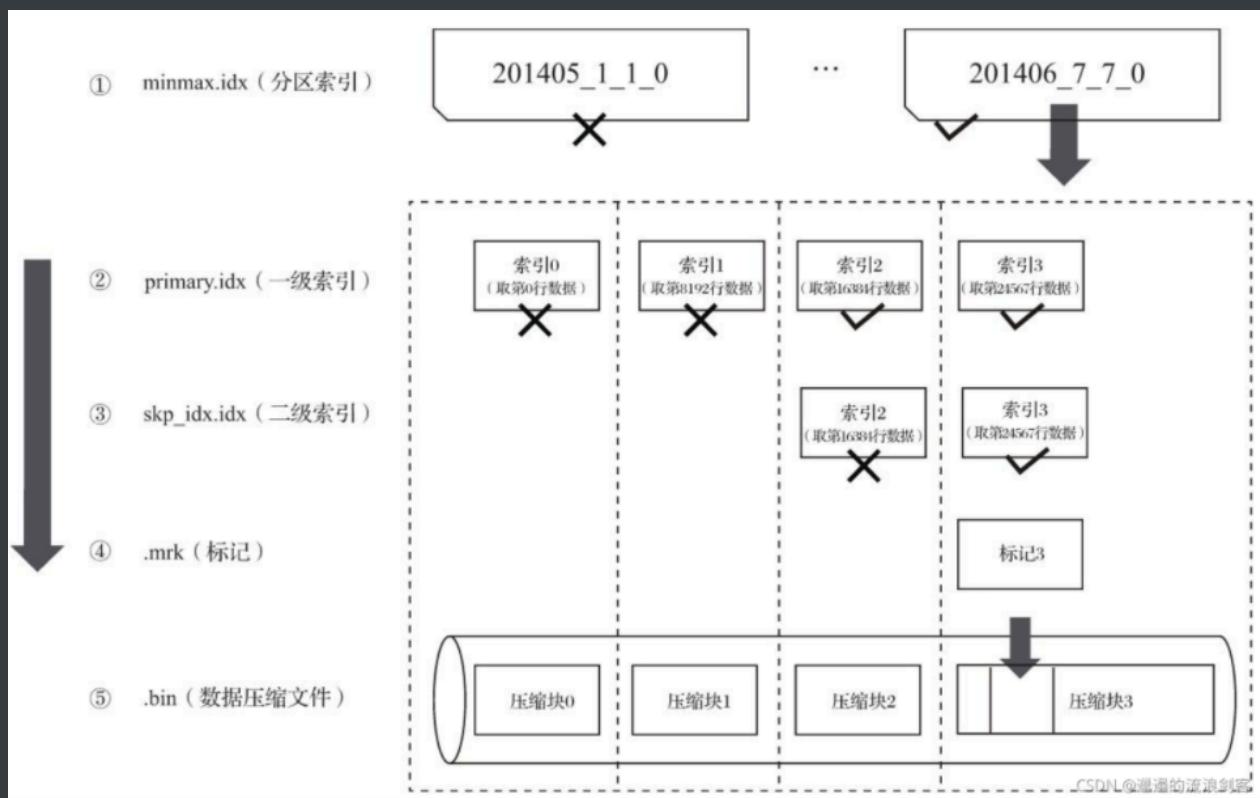
MergeTree在数据具体的写入过程中，会依照索引粒度（默认情况下，每次取8192行），按批次获取数据并进行处理。如果把一批数据的未压缩大小设为size，则整个写入过程遵循以下规则：

1. 单个批次数据 $size < 64KB$ ：如果单个批次数据小于64KB，则继续获取下一批数据，直至累积到 $size \leq 64KB$ 时，生成下一个压缩数据块。
2. 单个批次数据 $64KB < size \leq 1MB$ ：如果单个批次数据大小恰好在64KB与1MB之间，则直接生成下一个压缩数据块。
3. 单个批次数据 $size > 1MB$ ：如果单个批次数据直接超过1MB，则首先按照1MB大小截断并生成下一个压缩数据块。剩余数据继续依照上述规则执行

3. Clickhouse 索引查询、写入过程

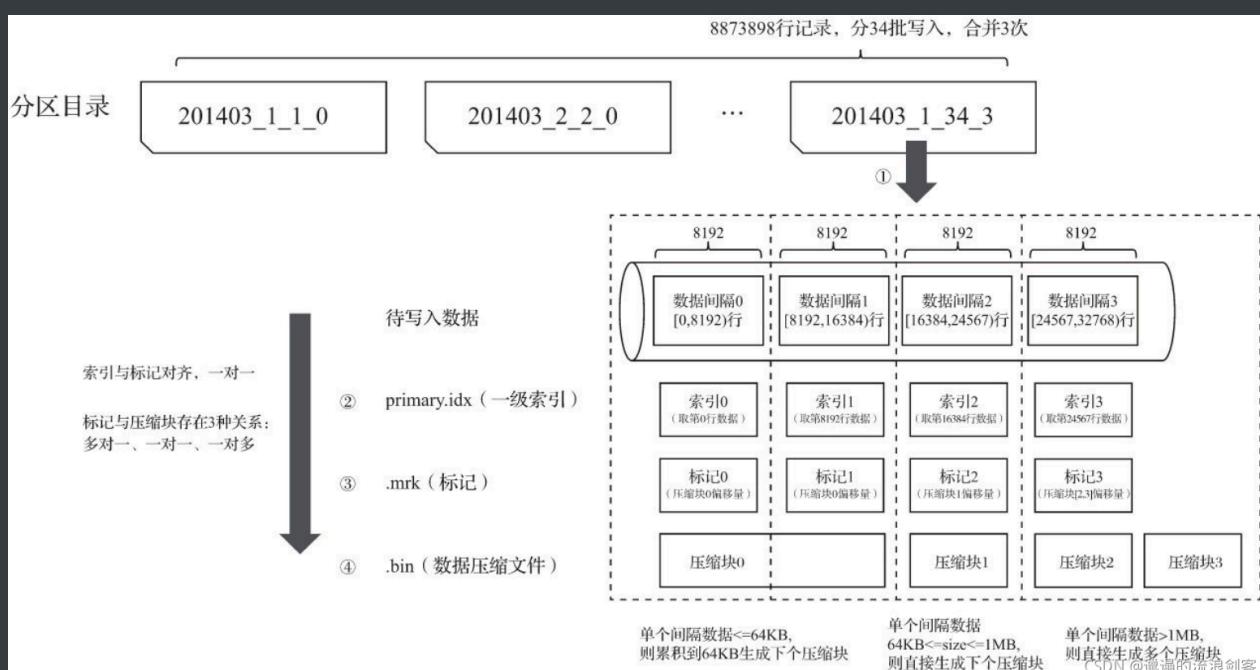
查询过程

数据查询的本质，可以看作一个不断减小数据范围的过程。在最理想的情况下，MergeTree首先可以依次借助分区索引、一级索引和跳数索引，将数据扫描范围缩至最小。然后再借助数据标记，将需要解压与计算的数据范围缩至最小。



写入过程

数据写入的第一步是生成分区目录，伴随着每一批数据的写入，都会生成一个新的分区目录。在后续的某一时刻，属于相同分区的目录会依照规则合并到一起；接着，按照索引粒度，会分别生成primary.idx一级索引、每一个列字段的.mrk数据标记文件和.bin压缩数据文件。



四、本地表与分布式表

1. CK 的几种表分类

- 本地表:

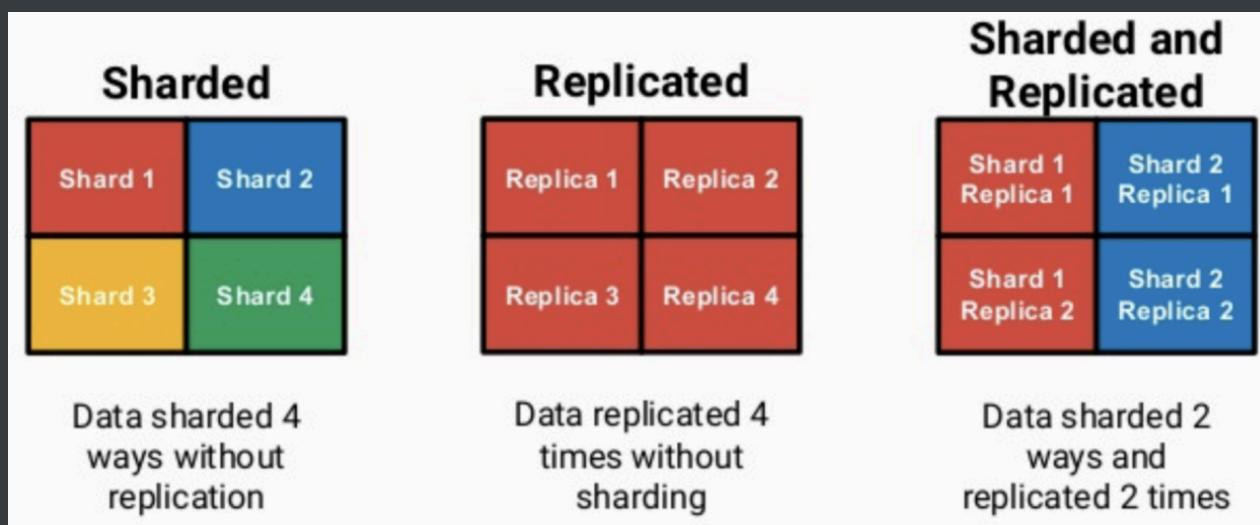
实际存储数据的表

- 分布式表 Distribute Table

一个逻辑上的表, 可以理解为数据库中的视图, 一般查询都查询分布式表. 分布式表引擎会将我们的查询请求路由本地表进行查询, 然后进行汇总最终返回给用户

2. 表分片与副本

- ClickHouse依靠ReplicatedMergeTree引擎族与ZooKeeper实现了复制表机制, 成为其高可用的基础.
- ClickHouse像ElasticSearch一样具有数据分片(shard)的概念, 这也是分布式存储的特点之一, 即通过并行读写提高效率. ClickHouse依靠Distributed引擎实现了分布式表机制, 在所有分片(本地表)上建立视图进行分布式查询.



3. 如和创建副本表

如何创建Replicated Table

不同于HDFS的副本机制(基于集群实现), Clickhouse的副本机制是基于表实现的. 用户在创建每张表的时候, 可以决定该表是否高可用.

```
CREATE TABLE IF NOT EXISTS {local_table}
({columns})
ENGINE =
ReplicatedMergeTree('/clickhouse/tables/#_tenant_id_#/##__appname__#/#
_at_date_#/#{shard}/hits', '{replica}')
partition by toString(_at_date_)
sample by intHash64.toInt64(toDateTime(_at_timestamp_)))
order by (_at_date_, _at_timestamp_,
intHash64.toInt64(toDateTime(_at_timestamp_)))
```

支持副本表的引擎都是ReplicatedMergeTree引擎族, 具体可以查看官网: [Data Replication](#)

ReplicatedMergeTree引擎族接收两个参数:

- ZK中该表相关数据的存储路径, ClickHouse官方建议规范化, 例如:
`/clickhouse/tables/{shard}/{database_name}/{table_name}` .
- 副本名称, 一般用 `{replica}` 即可.

ReplicatedMergeTree引擎族非常依赖于zookeeper, 它在zookeeper中存储了大量的数据:

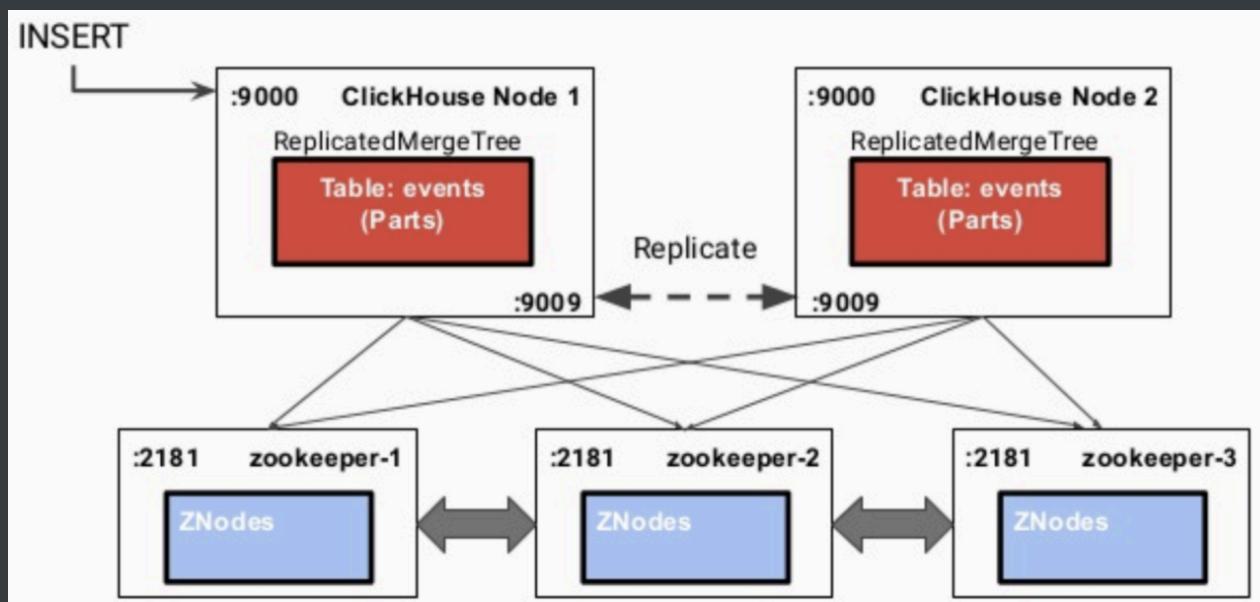
表结构信息、元数据、操作日志、副本状态、数据块校验值、数据part merge过程中的选主信息...

同时, zookeeper又在复制表急之下扮演了三种角色:

元数据存储、日志框架、分布式协调服务

可以说当使用了 ReplicatedMergeTree 时, zookeeper压力特别重, 一定要保证zookeeper集群的高可用和资源.

副本表数据同步的流程



1. 写入到一个节点
2. 通过 interserver HTTP port 端口同步到其他实例上
3. 更新zookeeper集群记录的信息

4. 分布式表与分布式引擎

ClickHouse分布式表的本质并不是一张表, 而是一些本地物理表(分片)的分布式视图,本身并不存储数据. 分布式表建表的引擎为 `Distributed` .

创建分布式表

- 使用`on cluster`语句在集群的某台机器上执行以下代码, 即可在每台机器上创建本地表和分布式表, 其中一张本地表对应着一个数据分片, 分布式表通常以本地表加“_all”命名。它与本地表形成一对多的映射关系, 之后可以通过分布式表代理操作多张本地表。
- 这里有个要注意的点, 就是分布式表的表结构尽量和本地表的结构一致。
如果不一致, 在建表时不会报错, 但在查询或者插入时可能会抛出异常。

```

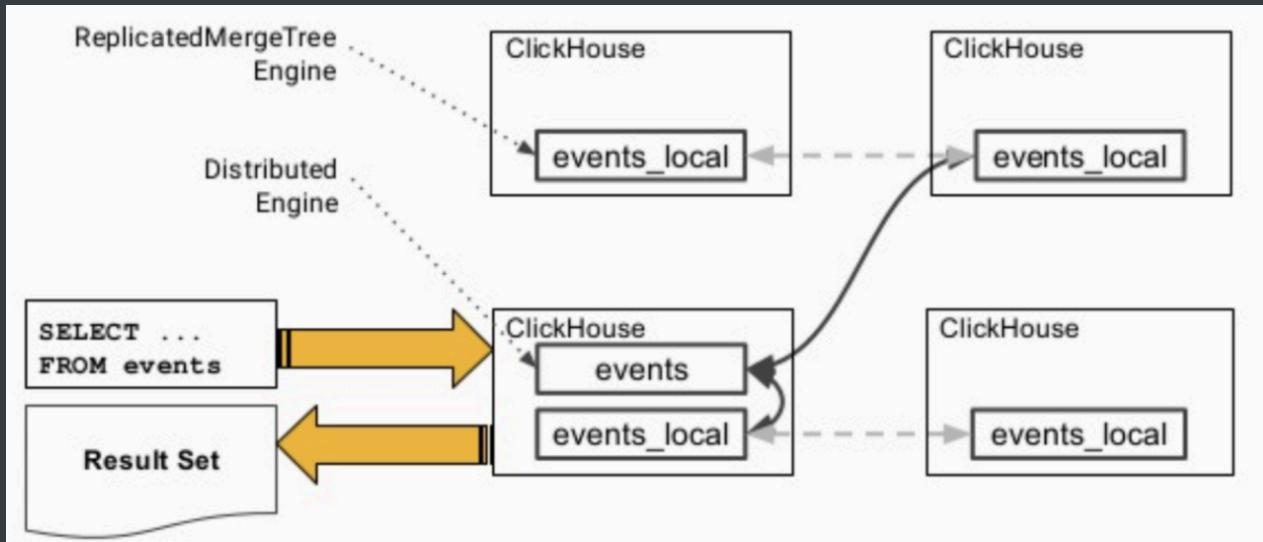
CREATE TABLE IF NOT EXISTS {distributed_table}
as {local_table}
ENGINE = Distributed({cluster}, '{local_database}', '{local_table}',
rand())
    
```

Distributed引擎需要以下几个参数:

- 集群标识符
- 本地表所在的数据库名称
- 本地表名称
- 分片键(sharding key) - 可选

该键与config.xml中配置的分片权重(weight)一同决定写入分布式表时的路由, 即数据最终落到哪个物理表上. 它可以是表中一列的原始数据(如 site_id), 也可以是函数调用的结果, 如上面的SQL语句采用了随机值 rand() . 注意该键要尽量保证数据均匀分布, 另外一个常用的操作是采用区分度较高的列的哈希值, 如 intHash64(user_id) .

数据查询的流程



1. 各个实例之间会交换自己持有的分片的表数据
2. 汇总到同一个实例上返回给用户

5. 大量数据插入最好不要使用分布式表

1. 分布式表接收到数据后会将数据拆分成多个parts, 并转发数据到其它服务器, 会引起服务器间网络流量增加、服务器merge的工作量增加, 导致写入速度变慢, 并且增加了Too many parts的可能性.
2. 数据的一致性问题, 先在分布式表所在的机器进行落盘, 然后异步的发送到本地表所在机器进行存储, 中间没有一致性的校验, 而且在分布式表所在机器时如果机器出现down机, 会存在数据丢失风险.
3. 数据写入默认是异步的, 短时间内可能造成不一致.
4. 对zookeeper的压力比较大.

五、Ck Bitmap和物化视图

1. Bitmap 与宽表

bitmap VS 大宽表

1. 宽表

明细数据, 各个标签作为独立的一列

优点:

- 支持明细数据查询

缺点:

- 数据量过大，数据行数过多，复杂查询条件下会导致查询效率极低；
- 需要支持动态schema，增减标签需要更改表结构，增加或减少数据列；
- 占用较多存储资源

mid	性别	会员类型	...
738291	M	年费	...
874013	F	非会员	...
858021	F	月费	...
...

2. Bitmap 结构

将字段的不同值作为标签，每个标签对应的用户mid数组(bitmap)作为value

优点

- 数据行数减小，查询效率提高
- 支持动态调整schema，不需要调整表结构
- 占用存储资源大大减少

缺点

- 数据明细查询不支持
- 标签的值需要尽量枚举，不支持连续型标签

tag	buids	log_date
24	[1,3,5]	2020-01-01
...

Clickhouse中的bitmap

bitmap在clickhouse中是一种AggregateFunction的数据类型，其构造方法有两种：

1. 通过聚合函数groupBitmap来构造

2. 通过bitmapBuild函数对整形数组进行转换得到

bimap和array类型的转换方式如下

- bitmapBuild : 将array转换为bitmap

```
select bitmapBuild([1,2,3,4,5]) as res, toTypeName(res);
```

```
SELECT
    bitmapBuild([1, 2, 3, 4, 5]) AS res,
    toTypeName(res)
  res | toTypeName(bitmapBuild([1, 2, 3, 4, 5])) |
    | AggregateFunction(groupBitmap, UInt8) |
```

- bitmapToArray : 将bitmap转为array

```
select bitmapToArray(bitmapBuild([1,2,3,4,5])) as res,
       toTypeName(res);
```

```
SELECT
    bitmapToArray(bitmapBuild([1, 2, 3, 4, 5])) AS res,
    toTypeName(res)
  res | toTypeName(bitmapToArray(bitmapBuild([1, 2, 3, 4, 5]))) |
    | [1,2,3,4,5] | Array(UInt8) |
```

2. 物化视图

概述

物化视图与普通视图的区别

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

优缺点

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

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

基本语法

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

```
CREATE [MATERIALIZED] VIEW [IF NOT EXISTS] [db.]table_name  
[TO [db.]name]  
[ENGINE = engine] [POPULATE] AS SELECT ...
```

创建物化视图的限制

1. 必须指定物化视图的 engine 用于数据存储
2. TO [db].[table]语法的时候，不得使用 POPULATE。
3. 查询语句(select) 可以包含下面的子句： DISTINCT, GROUP BY, ORDER BY, LIMIT...
4. 物化视图的 alter 操作有些限制，操作起来不大方便。
5. 若物化视图的定义使用了 TO [db.]name 子语句，则可以将目标表的视图 卸载DETACH 再装载 ATTACH

物化视图的数据更新策略

1. 物化视图创建好之后，若源表被写入新数据则物化视图也会同步更新
2. POPULATE 关键字决定了物化视图的更新策略：
 - 若有 POPULATE 则在创建视图的过程会将源表已经存在的数据一并导入，类似于create table ... as
 - 若无 POPULATE 则物化视图在创建之后没有数据，只会同步创建表之后写入源表的数据
 - clickhouse 官方并不推荐使用 POPULATE，因为在创建物化视图的过程中同时写入的数据不能被插入物化视图。
3. 物化视图不支持同步删除，若源表的数据不存在（删除了）则物化视图的数据仍然保留
4. 物化视图是一种特殊的数据表，可以用 show tables 查看

3. Clickhouse 集成roaringBitmap

BitMap字段类型，该类型扩展自AggregateFunction类型，表示groupBitmap聚合函数的中间状态(位图对象)，字段类型定义：

```
AggregateFunction(groupBitmap, UInt(8|16|32:64))
```

该类型可以通过groupBitmapStae获取，示例：

```
select labelname,labelvalue groupBitmapState(id) As uv from  
label_table group by labelname, labelvalue
```

注：

- groupBitmap(expr): 从无符号整数列进行位图或聚合计算，返回 UInt64 类型的基数，说白了就是根据位图统计不重复元素个数；
- groupBitmapAnd / groupBitmapOr: 计算位图的 AND/OR 的聚合结果，返回 UInt64 类型的基数；
- groupBitmapState: 从无符号整数列进行位图或聚合计算，返回位图对象；
- groupBitmapAndState / groupBitmapOrState: 计算位图的 AND/OR 的聚合结果，返位图对象；

4. Bitmap标签表实例

创建原始表

```
CREATE TABLE IF NOT EXISTS tbl_tag_src ON CLUSTER default_cluster(  
    tagname String,    --标签名称  
    tagvalue String,   --标签值  
    userid UInt64  
)ENGINE =  
ReplicatedMergeTree('/clickhouse/default/tables/{shard}/tbl_tag_src  
,'{replica}')  
PARTITION BY tagname  
ORDER BY tagvalue;
```

创建分布式表

```
CREATE TABLE IF NOT EXISTS default.tbl_tag_src_all ON CLUSTER  
default_cluster  
AS tbl_tag_src  
ENGINE = Distributed(default_cluster, default, tbl_tag_src, rand());
```

建立标签位图表

```
CREATE TABLE IF NOT EXISTS tbl_tag_bitmap ON CLUSTER default_cluster
(
    tagname String,    --标签名称
    tagvalue String,   --标签值
    tagbitmap AggregateFunction(groupBitmap, UInt64)    --userid集合
)
ENGINE =
ReplicatedAggregatingMergeTree('/clickhouse/default/tables/{shard}/tb
l_tag_bitmap ','{replica}')
PARTITION BY tagname
ORDER BY (tagname, tagvalue)
SETTINGS index_granularity = 128;
```

创建位图分布式表(这里最好可以使用物化视图代替)

```
CREATE TABLE IF NOT EXISTS default.tbl_tag_bitmap_all ON CLUSTER
default_cluster
(
    tagname String,    --标签名称
    tagvalue String,   --标签值
    tagbitmap AggregateFunction(groupBitmap, UInt64 )    --userid集合
)
ENGINE = Distributed(default_cluster, default, tbl_tag_bitmap,
rand());
```

将标签原始表的数据到位图表

```
-- 导入数据， 将同一个标签的所有userid使用groupBitmapState函数合并成一个
bitmap
INSERT INTO tbl_tag_bitmap_all
SELECT tagname,tagvalue,groupBitmapState(userid)
FROM tbl_tag_src_all
GROUP BY tagname,tagvalue;
```

标签查询

查询持有贵金属产品，并且性别是男的userid列表：

```

WITH
(
    SELECT tagbitmap FROM tbl_tag_bitmap_all WHERE tagname = '持有
产品' AND tagvalue = '贵金属' LIMIT 1
) AS bitmap1,
(
    SELECT tagbitmap FROM tbl_tag_bitmap_all WHERE tagname = '性
别' AND tagvalue = '男' LIMIT 1
) AS bitmap2
SELECT bitmapToArray(bitmapAnd(bitmap1, bitmap2)) AS res

```

分别统计持有保险的客户中，男性和女性的总人数：

```

---- 查询持有保险的客户中，男性人数：
WITH
(
    SELECT tagbitmap FROM tbl_tag_bitmap_all WHERE tagname = '持有
产品' AND tagvalue = '保险' LIMIT 1
) AS bitmap1,
(
    SELECT tagbitmap FROM tbl_tag_bitmap_all WHERE tagname = '性
别' AND tagvalue = '男' LIMIT 1
) AS bitmap2
SELECT bitmapCardinality(bitmapAnd(bitmap1, bitmap2)) AS res

---- 查询持有保险的客户中，女性人数：
WITH
(
    SELECT tagbitmap FROM tbl_tag_bitmap_all WHERE tagname = '持有
产品' AND tagvalue = '保险' LIMIT 1
) AS bitmap1,
(
    SELECT tagbitmap FROM tbl_tag_bitmap_all WHERE tagname = '性
别' AND tagvalue = '女' LIMIT 1
) AS bitmap2
SELECT bitmapCardinality(bitmapAnd(bitmap1, bitmap2)) AS res

```

六、关于ck的一些总结

1. Clickhouse 数据查询速度为什么这么快

大方向上

- 列式存储与数据压缩

ClickHouse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。

在执行数据查询时，列式存储可以减少数据扫描范围和数据传输时的大小，提高了数据查询的效率。

- 数据分片与分布式查询

ClickHouse通过分片和分布式表机制提供了线性扩展的能力。

分片机制：用来解决单节点的性能瓶颈，通过将数据进行水平切分，将一张表中的数据拆分到多个节点，不同节点之间的数据没有重复，这样就可以通过增加分片对ClickHouse进行线性扩展。

分布式表：在查询分片的数据时，通过分布式表进行查询，分布式表引擎自身不存储任何数据，仅是一层代理，能够自动路由到集群中的各个分片节点获取数据，即分布式表需要和其他数据表一起协同工作。

- mergetree 和 稀疏索引

仅需使用少量的索引标记就能够记录大量数据的区间位置信息，快速缩小数据查询范围

- 向量化执行引擎

ClickHouse利用CPU的SIMD指令实现了向量化执行。SIMD的全称是Single Instruction Multiple Data，即用单条指令操作多条数据，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，相比同类OLAP产品执行效率更高。

细节上

- 极致的算法优化

在 ClickHouse 的底层实现中，经常会面对一些重复的场景，例如字符串子串查询、数组排序、使用 HashTable 等。在每种数据处理场景都尽量选择了最效率的算法；

比如在字符串搜索方面，针对不同的场景，ClickHouse 最终选择了这些算法：对于常量，使用 Volnitsky 算法；对于非常量，使用 CPU 的向量化执行 SIMD，暴力优化；正则匹配使用 re2 和 hyperscan 算法。性能是算法选择的首要考量指标。

- 极致的cpu利用

- ClickHouse将数据划分为多个partition，每个partition再进一步划分为多个index granularity，然后通过多个CPU核心分别处理其中的一部分来实现并行数据处理。
- 在这种设计下，单条Query就能利用整机所有CPU。极致的并行处理能力，极大的降低了查询延时
- 列式存储代替行式存储，不仅可以加快数据扫描和读取效率，可以SIMD指令，还可以充分利用CPU cache 的预读能力，减少了CPU CACHE MISS的概率；

- 利用SIMD指令集；

ClickHouse实现了向量执行引擎（Vectorized execution engine），对内存中的列式数据，一个batch调用一次SIMD指令（而非每一行调用一次），不仅减少了函数调用次数、降低了cache miss，而且可以充分发挥SIMD指令的并行能力，大幅缩短了计算耗时。向量执行引擎，通常能够带来数倍的性能提升。

使用注意事项

1. 列式存储，查询列越少效率越高，不要用*；
2. 由于单表遍历的性能特别高，非常时候简单的条件查询和group by 查询；
3. 由于算法优化不足，对复杂场景的查询，比如联表查询和子查询性能很差，复杂查询场景考虑切换引擎；
4. 选择可以将数据尽量打散的列做为主键，利用稀疏索引去提升查询性能；
5. 大数据量频繁插入，不要插入分布式表，避免分片数据同步造成网络拥堵；
6. 插入数据的batch size 不要过小，由于合并树的合并过程，避免造成数据目录过多合并不过来；