ALIBABA CLOUD

阿里云

云数据库ClickHouse
开发指南

文档版本：20220712

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.数据类型

本文描述云数据库ClickHouse的数据类型。

数据类型列表

ClickHouse完整数据类型介绍，请参考开源官方数据类型介绍。

分类	关键字	数据类型	取值/取值范围
整数类型	`Int8`	Int8	取值范围：`-128 - 127`。
	`Int16`	Int16	取值范围：`-32768 - 32767`。
	`Int32`	Int32	取值范围：`-2147483648 - 2147483647`。
	`Int64`	Int64	取值范围：`-9223372036854775808 - 9223372036854775807`。
浮点类型	`Float32`	单精度浮点数	同C语言Float类型，单精度浮点数在机内占4个字节，用32位二进制描述。
	`Float64`	双精度浮点数	同C语言Double类型，双精度浮点数在机内占8个字节，用64位二进制描述。
Decimal类型	`Decimal`	Decimal	有符号的定点数，可在加、减和乘法运算过程中保持精度。支持几种写法： • Decimal(P, S) • Decimal32(S) • Decimal64(S) • Decimal128(S)
	`String`	字符串	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。

分类	关键字	数据类型	取值/取值范围
字符串类型	FixedString	固定字符串	当数据的长度恰好为N个字节时，FixedString类型是高效的。 在其他情况下，这可能会降低效率。可以有效存储在FixedString类型的列中的值的示例： • 二进制表示的IP地址（IPv6使用FixedString（16）） • 语言代码（ru_RU, en_US ... ） • 货币代码（USD, RUB ... ） • 二进制表示的哈希值（MD5使用FixedString（16），SHA256使用FixedString（32））
时间日期类型	`Date`	日期	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。
	`DateTime`	时间戳	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（没有闰秒）。时区使用启动客户端或服务器时的系统时区。
	`Datetime64`	Datetime64	此类型允许以日期（date）加时间（time）的形式来存储一个时刻的时间值。
布尔型	Boolean	Boolean	ClickHouse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。

分类	关键字	数据类型	取值/取值范围
数组类型	`Array`	Array	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。
元组类型	`Tuple`	Tuple	Tuple(T1, T2, ...)，元组，其中每个元素都有单独的类型，不能在表中存储元组（除了内存表）。它们可以用于临时列分组。在查询中，IN表达式和带特定参数的 lambda 函数可以来对临时列进行分组。
Domain数据类型	`Domain`	Domain	Domain类型是特定实现的类型： IPv4是与UInt32类型保持二进制兼容的Domain类型，用于存储IPv4地址的值。它提供了更为紧凑的二进制存储的同时支持识别可读性更加友好的输入输出格式。 IPv6是与FixedString（16）类型保持二进制兼容的Domain类型，用于存储IPv6地址的值。它提供了更为紧凑的二进制存储的同时支持识别可读性更加友好的输入输出格式。
枚举类型	`Enum8`	Enum8	取值范围：`-128 - 127`。
	`Enum16`	Enum16	取值范围：`-32768 - 32767`。

分类	关键字	数据类型	取值/取值范围
可为空	`Nullable`	Nullable	除非在 ClickHouse 服务器配置中另有说明，否则 NULL 是任何 Nullable 类型的默认值。Nullable 类型字段不能包含在表索引中。
嵌套类型	`nested`	nested	嵌套的数据结构就像单元格内的表格。嵌套数据结构的参数（列名和类型）的指定方式与CREATE TABLE查询中的指定方式相同。每个表行都可以对应于嵌套数据结构中的任意数量的行。

2.SQL语法

目前云数据库ClickHouse SQL语法与开源社区版完全一致。

ClickHouse官网说明：SQL语法介绍

2.1. Create DataBase

本文介绍创建数据库的基本语法。

CREATE DATABASE基本语法如下：

```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER cluster];
```

如果CREATE 语句中存在IF NOT EXISTS 关键字，则当数据库已经存在时，该语句不会创建数据库，且不会返回任何错误。

ON CLUSTER 关键字用于指定集群名称。

示例：

```
CREATE DATABASE db_001 ON CLUSTER default;
```

2.2. Create Table

本文介绍如何在云数据库ClickHouse中创建表。

创建本地表

语法：

```
CREATE TABLE [IF NOT EXISTS] [db.]local_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 = engine_name()
[PARTITION BY expr]
ORDER BY expr
[PRIMARY KEY expr]
[SAMPLE BY expr]
[SETTINGS name=value, ...];
```

参数说明：

参数	说明
db	数据库的名称，默认为当前选择的数据库，本文以default为例。
local_table_name	本地表名。

参数	说明
`ON CLUSTER cluster`	在每一个节点上都创建一个本地表，固定为 `ON CLUSTER default` 。
`name1,name2`	列名。
`type1,type2`	列类型。 说明 云数据库ClickHouse支持的数据类型，请参见数据类型。
`ENGINE = engine_name()`	表引擎类型。 双副本版集群建表时，需要使用MergeTree系列引擎中支持数据复制的Replicated*引擎，否则副本之间不进行数据复制，导致数据查询结果不一致。使用该引擎建表时，参数填写方式如下。 • `ReplicatedMergeTree('/clickhouse/tables/{database}/{table}/{shard}', '{replica}')` ，固定配置，无需修改。 • `ReplicatedMergeTree()` ，等同于 `ReplicatedMergeTree('/clickhouse/tables/{database}/{table}/{shard}', '{replica}')` 。 说明 云数据库ClickHouse支持的表引擎类型，请参见表引擎。
`ORDER BY expr`	排序键，必填项，可以是一组列的元组或任意表达式。
`[DEFAULT|MATERIALIZED|ALIAS expr]`	默认表达式。 • DEFAULT：普通的默认表达式。在字段缺省的情况下，自动生成默认值并填充。 • MATERIALIZED：物化表达式。 • ALIAS：别名表达式。
`GRANULARITY`	索引粒度参数。
`[PARTITION BY expr]`	分区键。一般按照日期分区，也可以使用其他字段或字段表达式。
`[PRIMARY KEY expr]`	主键，默认情况下主键和排序键相同。因此，多数情况下，不需要再专门使用 `PRIMARY KEY` 子句指定主键。
`[SAMPLE BY expr]`	抽样表达式。如果要使用抽样表达式，主键中必须包含这个表达式。
`[SETTINGS name=value, ...]`	影响性能的额外参数。 说明 `SETTINGS` 中支持的参数，请参见SETTINGS配置项。

> **说明** 参数 `ORDER BY`、`GRANULARITY`、`PARTITION BY`、`PRIMARY KEY`、`SAMPLE BY` 和 `[SETTINGS name=value, ...]` 只有MergeTree系列表引擎支持。更多参数说明，请参见CREATE TABLE。

示例：

```
CREATE TABLE local_table ON CLUSTER default
(
    Year UInt16,
    Quarter UInt8,
    Month UInt8,
    DayofMonth UInt8,
    DayOfWeek UInt8,
    FlightDate Date,
    FlightNum String,
    Div5WheelsOff String,
    Div5TailNum String
)ENGINE = MergeTree()
 PARTITION BY toYYYYMM(FlightDate)
 PRIMARY KEY (intHash32(FlightDate))
 ORDER BY (intHash32(FlightDate),FlightNum)
 SAMPLE BY intHash32(FlightDate)
SETTINGS index_granularity= 8192;
```

创建分布式表

分布式表是本地表的集合，它将多个本地表抽象为一张统一的表，对外提供写入、查询功能。当数据写入分布式表时，会被自动分发到集合中的各个本地表中。当查询分布式表时，集合中的各个本地表都会被分别查询，并且把最终结果汇总后返回。您需要先创建本地表，再创建分布式表。

语法：

```
CREATE TABLE [db.]distributed_table_name ON CLUSTER default
 AS db.local_table_name ENGINE = Distributed(cluster, db, local_table_name [, sharding_key]
)
```

参数说明：

参数	说明
`db`	数据库的名称，默认为当前选择的数据库，本文以default为例。
`distributed_table_name`	分布式表名。
`ON CLUSTER cluster`	在每一个节点上都创建一个表，固定为 `ON CLUSTER default` 。
`local_table_name`	已创建的本地表名。

参数	说明
sharding_key	分片表达式，用于决定将数据写到哪个分片。 sharding_key 可以是一个表达式，例如函数 rand() ，也可以是一列，例如 user_id （Integer类型）。

示例：

```
CREATE TABLE distributed_table ON CLUSTER default
 AS default.local_table
ENGINE = Distributed(default, default, local_table, rand());
```

通过复制表结构创建表

您可以通过复制表结构创建与源表具有相同结构的表。

语法：

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name2 ON CLUSTER cluster AS [db.]table_name1 [ENGIN
E = engine_name];
```

参数说明：

参数	说明
db	数据库的名称，默认为当前选择的数据库，本文以default为例。
table_name1	被复制表结构的源表，本文以已创建的本地表local_table为例。
table_name2	新创建的表。
ON CLUSTER cluster	在每一个节点上都创建一个表，固定为 ON CLUSTER default 。
[ENGINE = engine_name]	表引擎类型。如果没有指定表引擎，默认与被复制表结构的表相同。 **说明** 云数据库ClickHouse支持的表引擎类型，请参见表引擎。

示例：

```
create table t2 ON CLUSTER default as default.local_table;
```

通过SELECT语句创建表

使用指定的表引擎创建一个与 SELECT 子句的结果具有相同结构的表，并使用SELECT子句的结果进行填充。

语法：

```
CREATE TABLE [IF NOT EXISTS] [db.]s_table_name ON CLUSTER cluster ENGINE = engine_name() AS
SELECT ...
```

参数说明：

参数	说明
db	数据库的名称，默认为当前选择的数据库，本文以default为例。
s_table_name	通过SELECT语句创建的表。
ON CLUSTER cluster	在每一个节点上都创建一个表，固定为 ON CLUSTER default 。
ENGINE = engine_name()	表引擎类型。 说明 云数据库ClickHouse支持的表引擎类型，请参见表引擎。
SELECT ...	SELECT 子句。

示例：

```
create table t3 ON CLUSTER default ENGINE =MergeTree() order by Year as select * from defau
lt.local_table;
```

参考文档

- 创建表的更多信息，请参见CREATE TABLE。
- 通过复制表结构创建表的更多信息，请参见With a Schema Similar to Other Table。
- 通过SELECT语句创建表的更多信息，请参见From SELECT query。

2.3. Create View

本文介绍如何在云数据库ClickHouse中创建普通视图。

创建视图

语法：

```
CREATE VIEW [IF NOT EXISTS] [db.]view_name [ON CLUSTER cluster] AS SELECT ...
```

参数说明：

参数	说明
db	数据库的名称，默认为当前选择的数据库，本文以default为例。
view_name	视图名。
[ON CLUSTER cluster]	在每一个节点上都创建一个视图，固定为 ON CLUSTER default 。

参数	说明
SELECT ...	SELECT 子句。当数据写入视图中 SELECT 子句所指定的源表时，插入的数据会通过 SELECT 子句查询进行转换并将最终结果插入到视图中。 说明 SELECT 查询可以包含 DISTINCT 、 GROUP BY 、 ORDER BY 和 LIMIT 等，但是相应的转换是在每个插入数据块上独立执行的。

示例：

1. 创建 SELECT 子句指定的源表。

```
create table test ON CLUSTER default (
  id Int32,
  name String
) ENGINE = MergeTree()
    ORDER BY (id);
```

2. 创建基于源表的视图。

```
CREATE VIEW test_view ON CLUSTER default AS SELECT * FROM test;
```

3. 写入数据至源表。

```
insert into test values(1,'a'),(2,'b'),(3,'c');
```

4. 查询视图。

```
SELECT * FROM test_view;
```

查询结果如下。

```
id┃name
━┃━━
 1┃ a
 2┃ b
 3┃ c
```

参考文档

创建视图的更多信息，请参见Create View。

2.4. Create Materialized View

本文介绍如何在云数据库ClickHouse中创建物化视图。

创建物化视图

语法：

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] [db.]Materialized_name [TO[db.]name] [ON CLUSTER c
luster]
ENGINE = engine_name()
ORDER BY expr
[POPULATE]
AS SELECT ...
```

参数说明：

参数	说明
db	数据库的名称，默认为当前选择的数据库，本文以default为例。
Materialized_name	物化视图名。
TO[db.]name	将物化视图的数据写入到新表中。 **说明** 如果需要将物化视图的数据写入新表，不能使用 POPULATE 关键字。
[ON CLUSTER cluster]	在每一个节点上都创建一个物化视图，固定为 ON CLUSTER default 。
ENGINE = engine_name()	表引擎类型，具体请参见表引擎。
[POPULATE]	POPULATE 关键字。如果创建物化视图时指定了 POPULATE 关键字，则在创建时将 SELECT 子句所指定的源表数据插入到物化视图中。不指定 POPULATE 关键字时，物化视图只会包含在物化视图创建后新写入源表的数据。 **说明** 一般不推荐使用 POPULATE 关键字，因为在物化视图创建期间写入源表的数据将不会写入物化视图中。
SELECT ...	SELECT 子句。当数据写入物化视图中 SELECT 子句所指定的源表时，插入的数据会通过 SELECT 子句查询进行转换并将最终结果插入到物化视图中。 **说明** SELECT 查询可以包含 DISTINCT 、 GROUP BY 、 ORDER BY 和 LIMIT 等，但是相应的转换是在每个插入数据块上独立执行的。

示例：

1. 创建 SELECT 子句指定的源表。

```
create table test ON CLUSTER default (
  id Int32,
  name String
) ENGINE = ReplicatedMergeTree()
    ORDER BY (id);
```

2. 写入数据至源表。

```
insert into test values(1,'a'),(2,'b'),(3,'c');
```

3. 创建基于源表的物化视图。

```
CREATE MATERIALIZED VIEW test_view ON CLUSTER default
ENGINE = MergeTree()
ORDER BY (id) AS SELECT * FROM test;
```

4. 查询物化视图，验证未指定 `POPULATE` 关键字时，是否能查询到物化视图创建前写入源表的数据。

```
SELECT * FROM test_view;
```

查询数据为空，说明未指定 `POPULATE` 关键字时，查询不到物化视图创建前写入源表的数据。

5. 写入数据至源表。

```
insert into test values(4,'a'),(5,'b'),(6,'c');
```

6. 查询物化视图。

```
SELECT * FROM test_view;
```

查询结果如下。

```
id|name
—|—
 4| a
 5| b
 6| c
```

参考文档

创建物化视图的更多信息，请参见Create Materialized View。

2.5. INSERT INTO

本文介绍如何用INSERT INTO 语句往表中插入数据。

基本语法

INSERT INTO 语句基本格式如下：

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

对于存在于表结构中但不存在于插入列表中的列，它们将会按照如下方式填充数据：

- 如果存在DEFAULT表达式，根据DEFAULT表达式计算被填充的值。
- 如果没有定义DEFAULT表达式，则填充零或空字符串。

使用SELECT的结果写入

语法结构如下：

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

写入的列与SELECT的列的对应关系是使用位置来进行对应的，它们在SELECT表达式与INSERT中的名称可以是不同的。需要对它们进行对应的类型转换。

除了VALUES格式之外，其他格式中的数据都不允许出现诸如now()，1 + 2等表达式。VALUES格式允许您有限度的使用这些表达式，但是不建议您这么做，因为执行这些表达式很低效。

影响性能的注意事项

在执行INSERT时将会对写入的数据进行一些处理，比如按照主键排序、按照月份对数据进行分区等。如果在您的写入数据中包含多个月份的混合数据时，将会显著地降低INSERT的性能。为了避免这种情况，通常采用以下方式：

- 数据总是以尽量大的batch进行写入，如每次写入100,000行。
- 数据在写入ClickHouse前预先对数据进行分组。

在以下的情况下，性能不会下降：

- 数据总是被实时地写入。
- 写入的数据已经按照时间排序。

2.6. SELECT

本文描述如何使用SELECT语句查询数据。

基本语法

SELECT语句基本格式如下：

```
SELECT [DISTINCT] expr_list
    [FROM [db.]table | (subquery) | table_function] [FINAL]
    [SAMPLE sample_coeff]
    [ARRAY JOIN ...]
    [GLOBAL] ANY|ALL INNER|LEFT JOIN (subquery)|table USING columns_list
    [PREWHERE expr]
    [WHERE expr]
    [GROUP BY expr_list] [WITH TOTALS]
    [HAVING expr]
    [ORDER BY expr_list]
    [LIMIT [n, ]m]
    [UNION ALL ...]
    [INTO OUTFILE filename]
    [FORMAT format]
    [LIMIT n BY columns]
```

所有的子句都是可选的，除了SELECT之后的表达式列表（expr_list）。 下面将选择部分子句进行说明。ClickHouse官网中文文档有更详细说明，请参考查询语法。

简单查询语句示例：

```
SELECT
    OriginCityName,
    DestCityName,
    count(*) AS flights,
    bar(flights, 0, 20000, 40)
FROM ontime_distributed
WHERE Year = 1988
GROUP BY OriginCityName, DestCityName
ORDER BY flights DESC
LIMIT 20;
```

SAMPLE 子句

通过SAMPLE子句用户可以进行近似查询处理，近似查询处理仅能工作在MergeTree*类型的表中，并且在创建表时需要您指定采样表达式。

SAMPLE子句可以使用SAMPLE k来表示，其中k可以是0到1的小数值，或者是一个足够大的正整数值。

当k为0到1的小数时，查询将使用'k'作为百分比选取数据。例如，SAMPLE 0.1查询只会检索数据总量的10%。当k为一个足够大的正整数时，查询将使用'k'作为最大样本数。例如，SAMPLE 10000000查询只会检索最多10,000,000行数据。

示例：

```
SELECT
    Title,
    count() * 10 AS PageViews
FROM hits_distributed
SAMPLE 0.1
WHERE
    CounterID = 34
    AND toDate(EventDate) >= toDate('2013-01-29')
    AND toDate(EventDate) <= toDate('2013-02-04')
    AND NOT DontCountHits
    AND NOT Refresh
    AND Title != ''
GROUP BY Title
ORDER BY PageViews DESC LIMIT 1000;
```

在这个例子中，查询将检索数据总量的0.1（10%）的数据。值得注意的是，查询不会自动校正聚合函数最终的结果，所以为了得到更加精确的结果，需要将count()的结果手动乘以10。

当使用像SAMPLE 10000000这样的方式进行近似查询时，由于没有了任何关于将会处理了哪些数据或聚合函数应该被乘以几的信息，所以这种方式不适合在这种场景下使用。

使用相同的采样率得到的结果总是一致的：如果我们能够看到所有可能存在在表中的数据，那么相同的采样率总是能够得到相同的结果（在建表时使用相同的采样表达式），换句话说，系统在不同的时间，不同的服务器，不同表上总以相同的方式对数据进行采样。

例如，我们可以使用采样的方式获取到与不进行采样相同的用户ID的列表。这将表明，您可以在IN子查询中使用采样，或者使用采样的结果与其他查询进行关联。

ARRAY JOIN 子句

ARRAY JOIN子句可以帮助查询并进行与数组和nested数据类型的连接。它有点类似arrayJoin函数，但它的功能更广泛。

ARRAY JOIN本质上等同于INNER JOIN数组。例如：

```
:) CREATE TABLE arrays_test (s String, arr Array(UInt8)) ENGINE = Memory
CREATE TABLE arrays_test
(
    s String,
    arr Array(UInt8)
) ENGINE = Memory
Ok.
0 rows in set. Elapsed: 0.001 sec.
:) INSERT INTO arrays_test VALUES ('Hello', [1,2]), ('World', [3,4,5]), ('Goodbye', [])
INSERT INTO arrays_test VALUES
Ok.
3 rows in set. Elapsed: 0.001 sec.
:) SELECT * FROM arrays_test
SELECT *
FROM arrays_test
┌─s───────┬─arr─────┐
│ Hello   │ [1,2]   │
│ World   │ [3,4,5] │
│ Goodbye │ []      │
└─────────┴─────────┘
3 rows in set. Elapsed: 0.001 sec.
:) SELECT s, arr FROM arrays_test ARRAY JOIN arr
SELECT s, arr
FROM arrays_test
ARRAY JOIN arr
┌─s─────┬─arr─┐
│ Hello │   1 │
│ Hello │   2 │
│ World │   3 │
│ World │   4 │
│ World │   5 │
└───────┴─────┘
5 rows in set. Elapsed: 0.001 sec.
```

Join 语句Null的处理

JOIN语句中的Null处理，请参考join_use_nulls、Nullable、NULL。

WHERE 子句

如果存在WHERE子句，则在该子句中必须包含一个UInt8类型的表达式。这个表达式通常是一个带有比较和逻辑的表达式。这个表达式将会在所有数据转换前用来过滤数据。

如果在支持索引的数据库表引擎中，这个表达式将被评估是否使用索引。

PREWHERE 子句

这个子句与WHERE子句的意思相同。主要的不同之处在于表数据的读取。当使用PREWHERE时，首先只读取PREWHERE表达式中需要的列。然后再根据PREWHERE执行的结果读取其他需要的列。

如果在过滤条件中有少量不适合索引过滤的列，但是它们又可以提供很强的过滤能力。这时使用PREWHERE是有意义的，因为它将帮助减少数据的读取。

例如：在一个需要提取大量列的查询中为少部分列编写PREWHERE是很有作用的。

> **说明** PREWHERE仅支持*MergeTree系列引擎。在一个查询中可以同时指定PREWHERE和WHERE，在这种情况下，PREWHERE优先于WHERE执行。PREWHERE不适合用于已经存在于索引中的列，因为当列已经存在于索引中的情况下，只有满足索引的数据块才会被读取。如果将'optimize_move_to_prewhere'设置为1，并且在查询中不包含PREWHERE，则系统将自动的把适合PREWHERE表达式的部分从WHERE中抽离到PREWHERE中。

WITH TOTALS 修饰符

如果您指定了WITH TOTALS修饰符，将会在结果中得到一个被额外计算出的行。在这一行中将包含所有key的默认值（零或者空值），以及所有聚合函数对所有被选择数据行的聚合结果。

该行仅在JSON*, TabSeparated*, Pretty*输出格式中与其他行分开输出。

3.集群参数设置

3.1. config.xml参数修改

本文介绍了修改ClickHouse集群config.xml配置文件中全局参数的方法。

背景信息

ClickHouse集群有若干参数，参数默认值通常配置在config.xml和user.xml文件中，您可以通过修改全局参数帮助进行实例优化，其中：

- user.xml配置文件您可以通过set global 命令行进行全局修改，无需重启实例，详情请参见user.xml参数修改。
- Config.xml配置文件中的参数，您可以通过控制台进行在线修改并重启实例使修改后的参数生效。基于修改频率和实例稳定性考虑，您可以通过控制台修改的参数如下表所示：

参数名	默认值	取值范围	是否需要重启	参数说明
keep_alive_timeout	300	>0	是	ClickHouse在关闭连接之前等待传入请求的秒数。
max_connections	4096	>0	是	最大入站连接数。
max_partition_size_to_drop	0	>=0	是	如果MergeTree分区的大小超过 `max_partition_size_to_drop` （以字节为单位），则不能使用DROP语句将其删除。
max_concurrent_queries	100	>0	是	同时处理的最大请求数。
uncompressed_cache_size	1717986918	>0	是	MergeTree表引擎使用的未压缩数据的缓存大小（以字节为单位）。 ClickHouse实例有一个共享缓存，内存按需分配。如果启用了“使用未压缩的缓存”选项，则使用缓存。
timezone	Asia/Shanghai	合法时区信息如： ○ Africa/Abidjan ○ Asia/Shanghai ○ Europe/Moscow	是	实例的时区。
mark_cache_size	5368709120	>=5368709120	是	MergeTree表引擎使用的标记缓存的近似大小（以字节为单位）。

参数名	默认值	取值范围	是否需要重启	参数说明
max_table_size_to_drop	0	>=0	是	如果MergeTree表的大小超过 `max_table_size_to_drop` （以字节为单位），则不能使用DROP语句将其删除。

参数修改

1. 使用阿里云账号登录云数据库ClickHouse控制台。
2. 在页面左上角，选择集群所在地域。
3. 在**集群列表**页面，选择**默认实例列表**或**云原生版实例列表**，单击目标集群ID。
4. 在左侧导航栏，单击**参数配置**，查看参数列表。
5. 单击参数列表中**运行参数值**列中的修改按钮，进入参数编辑状态，您可以根据提示的参考输入范围，输入合法的参数值，并点击**确定**。或单击**取消**退出参数编辑状态。

6. 您可以通过勾选页面右上方的**只显示已修改过的参数**筛选框，进行参数筛选，筛选后的参数列表页面仅展示修改过但未提交的参数。

撤销修改

> **注意** 仅支持撤销未提交的已修改参数。

修改参数后，如果您需要将修改过的参数全部取消修改，请在提交之前，点击参数列表右上角的**撤销**按钮。ClickHouse将会撤销全部已修改未提交的参数内容，显示为修改前的参数值。

提交参数

确认修改完所有参数后，点击页面右上角的**提交参数**按钮，进行参数提交。

> **说明** 如果提交参数中包含需要重启生效的参数，ClickHouse将会在完成参数修改后自动重启集群。请先确认重启目标ClickHouse集群不会对业务产生影响后，在弹出的修改参数窗口中单击**确认**。

确认修改参数后，ClickHouse先进行参数校验，如果参数值设置不合法，将会提示**错误**，请返回修改参数值在参考范围内后重新提交。

3.2. user.xml参数修改

本文介绍如何修改ClickHouse集群user.xml配置文件中的参数。

背景介绍

ClickHouse集群有若干参数，参数默认值通常配置在config.xml和user.xml文件中，您可以通过修改参数帮助进行实例优化，其中：

- config.xml配置文件中的参数，您可以通过控制台进行在线修改并重启实例使修改后的参数生效。详情请参见config.xml参数修改。
- user.xml配置文件您可以通过 `set global` 命令行进行修改，无需重启实例。

注意事项

user.xml配置文件中的参数不能使用clickhouse-client工具修改，您可以通过数据管理DMS控制台连接集群，并执行set语句修改user.xml配置文件中的参数。如何通过数据管理DMS控制台连接集群，请参见DMS连接ClickHouse。更多操作方式，请参见操作方式。

命令格式

```
set global on cluster default key = value;
```

如果value取值是bool型或字符型，value需用单引号括起来。示例：

```
set global on cluster default max_block_size=10000;
```

```
set global on cluster default totals_mode='any';
```

```
set global on cluster default input_format_parallel_parsing='True';
```

> **说明** 目前支持的参数清单，请参见 ClickHouse参数列表。

操作方式

- **通过数据管理DMS控制台执行**

  通过数据管理DMS控制台连接集群，并执行set语句。

- **通过客户端工具执行**

  通过DBeaver等客户端工具连接集群，然后运行设置参数命令，可以将上述set命令当成普通DDL一样运行。

- **通过JDBC连接执行**

  编写Java代码，通过JDBC连接集群，并执行set语句。如何通过JDBC连接集群，请参见通过JDBC方式连接ClickHouse。

4.数据库引擎

默认情况下，ClickHouse使用自己的数据库引擎，该引擎提供可配置的表引擎和所有支持的SQL语法。除此之外，还可以选择使用MySQL数据库引擎。

5.表引擎

云数据库ClickHouse支持的表引擎分为MergeTree、Log、Integrations和Special四个系列。本文主要对这四类表引擎进行概要介绍，并通过示例介绍常用表引擎的功能。

概述

表引擎即表的类型，在云数据库ClickHouse中决定了如何存储和读取数据、是否支持索引、是否支持主备复制等。云数据库ClickHouse支持的表引擎，请参见下表。

系列	描述	表引擎	特点
MergeTree	MergeTree系列引擎适用于高负载任务，支持大数据量的快速写入并进行后续的数据处理，通用程度高且功能强大。 该系列引擎的共同特点是支持数据副本、分区、数据采样等特性。	MergeTree	用于插入极大量的数据到一张表中，数据以数据片段的形式一个接着一个的快速写入，数据片段按照一定的规则进行合并。
		ReplacingMergeTree	用于解决MergeTree表引擎相同主键无法去重的问题，可以删除主键值相同的重复项。
		CollapsingMergeTree	在建表语句中新增标记列 `Sign` ，用于消除ReplacingMergeTree表引擎的如下功能限制。 • 在分布式场景下，相同主键的数据可能被分布到不同节点上，不同分片间可能无法去重。 • 在没有彻底optimize之前，可能无法达到主键去重的效果，比如部分数据已经被去重，而另外一部分数据仍旧有主键重复。 • optimize是后台动作，无法预测具体执行时间点。 • 手动执行optimize在海量数据场景下需要消耗大量时间，无法满足业务即时查询的需求。
		VersionedCollapsingMergeTree	在建表语句中新增 `Version` 列，用于解决CollapsingMergeTree表引擎乱序写入导致无法正常折叠（删除）的问题。
		SummingMergeTree	用于对主键列进行预先聚合，将所有相同主键的行合并为一行，从而大幅度降低存储空间占用，提升聚合计算性能。
		AggregatingMergeTree	预先聚合引擎的一种，用于提升聚合计算的性能，可以指定各种聚合函数。
		GraphiteMergeTree	用于存储Graphite数据并进行汇总，可以减少存储空间，提高Graphite数据的查询效率。

系列	描述	表引擎	特点
Log	Log系列引擎适用于快速写入小表（1百万行左右的表）并读取全部数据的场景。 该系列引擎的共同特点如下。 • 数据被追加写入磁盘中。 • 不支持 `delete` 、 `update` 。 • 不支持索引。 • 不支持原子性写。 • `insert` 会阻塞 `select` 操作。	TinyLog	不支持并发读取数据文件，格式简单，查询性能较差，适用于暂存中间数据。
		StripeLog	支持并发读取数据文件，将所有列存储在同一个大文件中，减少了文件数，查询性能比TinyLog好。
		Log	支持并发读取数据文件，每个列会单独存储在一个独立文件中，查询性能比TinyLog好。
Integrations	Integrations系列引擎适用于将外部数据导入到云数据库ClickHouse中，或者在云数据库ClickHouse中直接使用外部数据源。	Kafka	将Kafka Topic中的数据直接导入到云数据库ClickHouse。
		MySQL	将MySQL作为存储引擎，直接在云数据库ClickHouse中对MySQL表进行 `select` 等操作。
		JDBC	通过指定JDBC连接串读取数据源。
		ODBC	通过指定ODBC连接串读取数据源。
		HDFS	直接读取HDFS上特定格式的数据文件。
Special	Special系列引擎适用于特定的功能场景。	Distributed	本身不存储数据，可以在多个服务器上进行分布式查询。
		MaterializedView	用于创建物化视图。
		Dictionary	将字典数据展示为一个云数据库ClickHouse表。
		Merge	本身不存储数据，可以同时从任意多个其他表中读取数据。
		File	直接将本地文件作为数据存储。
		NULL	写入数据被丢弃，读取数据为空。
		Set	数据总是保存在RAM中。
		Join	数据总是保存在内存中。
		URL	用于管理远程HTTP、HTTPS服务器上的数据。

系列	描述	表引擎	特点
		View	本身不存储数据，仅存储指定的 `SELECT` 查询。
		Memory	数据存储在内存中，重启后会导致数据丢失。查询性能极好，适合于对于数据持久性没有要求的1亿以下的小表。 在云数据库ClickHouse中，通常用来做临时表。
		Buffer	为目标表设置一个内存Buffer，当Buffer达到了一定条件之后会写入到磁盘。

> **说明** 表引擎的更多信息，具体请参见表引擎介绍。

MergeTree

MergeTree表引擎主要用于海量数据分析，支持数据分区、存储有序、主键索引、稀疏索引和数据TTL等。MergeTree表引擎支持云数据库ClickHouse的所有SQL语法，但是部分功能与标准SQL存在差异。

本文以主键为例进行介绍。云数据库ClickHouse的SQL语法中主键用于去重，保持数据唯一，而在MergeTree表引擎中，其主要作用是加速查询，即便在Compaction完成后，主键相同的数据行也仍旧共同存在。

> **说明** MergeTree表引擎的更多信息，具体请参见MergeTree。

示例如下。

1. 创建表test_tbl，主键为 `id` 和 `create_time` ，并且按照主键进行存储排序，按照 `create_time` 进行数据分区。

```
CREATE TABLE test_tbl ON CLUSTER default (
  id UInt16,
  create_time Date,
  comment Nullable(String)
) ENGINE = MergeTree()
   PARTITION BY create_time
     ORDER BY  (id, create_time)
     PRIMARY KEY (id, create_time)
     SETTINGS index_granularity=8192;
```

2. 写入主键重复的数据。

```
insert into test_tbl values(1, '2019-12-13', null);
insert into test_tbl values(1, '2019-12-13', null);
insert into test_tbl values(2, '2019-12-14', null);
insert into test_tbl values(3, '2019-12-15', null);
insert into test_tbl values(3, '2019-12-15', null);
```

3. 查询数据。

```
select * from test_tbl;
```

查询结果如下。

```
id|create_time|comment
—| ———-|——
 1| 2019-12-13| NULL
 1| 2019-12-13| NULL
 2| 2019-12-14| NULL
 3| 2019-12-15| NULL
 3| 2019-12-15| NULL
```

4. 由于MergeTree系列表引擎采用类似LSM Tree的结构，很多存储层处理逻辑直到Compaction期间才会发生，因此需执行optimize语句强制后台Compaction。

```
optimize table test_tbl final;
```

5. 再次查询数据。

```
select * from test_tbl;
```

查询结果如下，主键重复的数据仍存在。

```
id|create_time|comment
—| ———-|——
 1| 2019-12-13| NULL
 1| 2019-12-13| NULL
 2| 2019-12-14| NULL
 3| 2019-12-15| NULL
 3| 2019-12-15| NULL
```

ReplacingMergeTree

为了解决MergeTree表引擎相同主键无法去重的问题，云数据库ClickHouse提供了ReplacingMergeTree表引擎，用于删除主键值相同的重复项。

虽然ReplacingMergeTree表引擎提供了主键去重的能力，但是仍然存在很多限制，因此ReplacingMergeTree表引擎更多被用于确保数据最终被去重，而无法保证查询过程中主键不重复，主要限制如下。

- 在分布式场景下，相同主键的数据可能被分布到不同节点上，不同分片间可能无法去重。
- 在没有彻底optimize之前，可能无法达到主键去重的效果，比如部分数据已经被去重，而另外一部分数据仍旧有主键重复。
- optimize是后台动作，无法预测具体执行时间点。
- 手动执行optimize在海量数据场景下需要消耗大量时间，无法满足业务即时查询的需求。

> **说明** ReplacingMergeTree表引擎的更多信息，具体请参见ReplacingMergeTree。

示例如下。

1. 创建表test_tbl_replacing。

```
CREATE TABLE test_tbl_replacing (
  id UInt16,
  create_time Date,
  comment Nullable(String)
) ENGINE = ReplacingMergeTree()
   PARTITION BY create_time
     ORDER BY  (id, create_time)
     PRIMARY KEY (id, create_time)
     SETTINGS index_granularity=8192;
```

2. 写入主键重复的数据。

```
insert into test_tbl_replacing values(1, '2019-12-13', null);
insert into test_tbl_replacing values(1, '2019-12-13', null);
insert into test_tbl_replacing values(2, '2019-12-14', null);
insert into test_tbl_replacing values(3, '2019-12-15', null);
insert into test_tbl_replacing values(3, '2019-12-15', null);
```

3. 查询数据。

```
select * from test_tbl_replacing;
```

查询结果如下。

```
id|create_time|comment
—| ———-|——
 1| 2019-12-13| NULL
 1| 2019-12-13| NULL
 2| 2019-12-14| NULL
 3| 2019-12-15| NULL
 3| 2019-12-15| NULL
```

4. 由于MergeTree系列表引擎采用类似LSM Tree的结构，很多存储层处理逻辑直到Compaction期间才会发生，因此需执行optimize语句强制后台Compaction。

```
optimize table test_tbl_replacing final;
```

5. 再次查询数据。

```
select * from test_tbl;
```

查询结果如下，主键重复的数据已消除。

```
id|create_time|comment
—| ———-|——
 1| 2019-12-13| NULL
 2| 2019-12-14| NULL
 3| 2019-12-15| NULL
```

CollapsingMergeTree

CollapsingMergeTree表引擎用于消除ReplacingMergeTree表引擎的功能限制。该表引擎要求在建表语句中指定一个标记列Sign，按照Sign的值将行分为两类： `Sign=1` 的行称为状态行，用于新增状态。 `Sign=-1` 的行称为取消行，用于删除状态。

> **说明** CollapsingMergeTree表引擎虽然解决了主键相同数据即时删除的问题，但是状态持续变化且多线程并行写入情况下，状态行与取消行位置可能乱序，导致无法正常折叠（删除）。

后台Compaction时会将主键相同、`Sign`相反的行进行折叠（删除），而尚未进行Compaction的数据，状态行与取消行同时存在。因此为了能够达到主键折叠（删除）的目的，需要业务层进行如下操作。

- 记录原始状态行的值，或者在执行删除状态操作前先查询数据库获取原始状态行的值。

  具体原因：执行删除状态操作时需要写入取消行，而取消行中需要包含与原始状态行主键一样的数据（Sign列除外）。

- 在进行`count()`、`sum(col)`等聚合计算时，可能会存在数据冗余的情况。为了获得正确结果，业务层需要改写SQL，将`count()`、`sum(col)`分别改写为`sum(Sign)`、`sum(col * Sign)`。

  具体原因如下。

  - 后台Compaction时机无法预测，在发起查询时，状态行和取消行可能尚未进行折叠（删除）。
  - 云数据库ClickHouse无法保证主键相同的行落在同一个节点上，不在同一节点上的数据无法进行折叠（删除）。

> **说明** CollapsingMergeTree表引擎的更多信息，具体请参见CollapsingMergeTree。

示例如下。

1. 创建表test_tbl_collapsing。

```
CREATE TABLE test_tbl_collapsing
(
    UserID UInt64,
    PageViews UInt8,
    Duration UInt8,
    Sign Int8
)
ENGINE = CollapsingMergeTree(Sign)
ORDER BY UserID;
```

2. 插入状态行`Sign=1`。

```
INSERT INTO test_tbl_collapsing VALUES (4324182021466249494, 5, 146, 1);
```

> **说明** 如果先插入取消行，再插入状态行，可能会导致位置乱序，即使强制后台Compaction，也无法进行主键折叠（删除）。

3. 插入取消行`Sign=-1`，除`Sign`列外其他值与插入的状态行一致。同时，插入一行相同主键数据的新状态行。

```
INSERT INTO test_tbl_collapsing VALUES (4324182021466249494, 5, 146, -1), (432418202146
6249494, 6, 185, 1);
```

4. 查询数据。

```
SELECT * FROM test_tbl_collapsing;
```

查询结果如下。

```
       UserID       |PageViews|Duration|Sign
 ———————————| ———|————|——
4324182021466249494|  5      | 146    | 1
4324182021466249494|  5      | 146    |-1
4324182021466249494|  6      | 185    | 1
```

5. 如果您需要对指定列进行聚合计算，以 `sum(col)` 为例，为了获得正确结果，需改写SQL语句如下。

```
SELECT
    UserID,
    sum(PageViews * Sign) AS PageViews,
    sum(Duration * Sign) AS Duration
FROM test_tbl_collapsing
GROUP BY UserID
HAVING sum(Sign) > 0;
```

进行聚合计算后，查询结果如下。

```
       UserID       |PageViews|Duration
 ———————————| ———|————
4324182021466249494|  6      | 185
```

6. 由于MergeTree系列表引擎采用类似LSM Tree的结构，很多存储层处理逻辑直到Compaction期间才会发生，因此需执行optimize语句强制后台Compaction。

```
optimize table test_tbl_collapsing final;
```

7. 再次查询数据。

```
SELECT * FROM test_tbl_collapsing;
```

查询结果如下。

```
       UserID       |PageViews|Duration|Sign
 ———————————| ———|————|——
4324182021466249494|  6      | 185    | 1
```

VersionedCollapsingMergeTree

为了解决CollapsingMergeTree表引擎乱序写入导致无法正常折叠（删除）问题，云数据库ClickHouse提供了VersionedCollapsingMergeTree表引擎，在建表语句中新增一列 `Version` ，用于在乱序情况下记录状态行与取消行的对应关系。后台Compaction时会将主键相同、 `Version` 相同、 `Sign` 相反的行折叠（删除）。

与CollapsingMergeTree表引擎类似，在进行 `count()` 、 `sum(col)` 等聚合计算时，业务层需要改写SQL，将 `count()` 、 `sum(col)` 分别改写为 `sum(Sign)` 、 `sum(col * Sign)` 。

> **说明** VersionedCollapsingMergeTree表引擎的更多信息，具体请参见VersionedCollapsingMergeTree。

示例如下。

1. 创建表test_tbl_Versioned。

```
CREATE TABLE test_tbl_Versioned
(
    UserID UInt64,
    PageViews UInt8,
    Duration UInt8,
    Sign Int8,
    Version UInt8
)
ENGINE = VersionedCollapsingMergeTree(Sign, Version)
ORDER BY UserID;
```

2. 插入取消行 `Sign=-1` 。

```
INSERT INTO test_tbl_Versioned VALUES (4324182021466249494, 5, 146, -1, 1);
```

3. 插入状态行 `Sign=1` 、 `Version=1` ，其他列值与插入的取消行一致。同时，插入一行相同主键数据的新状态行。

```
INSERT INTO test_tbl_Versioned VALUES (4324182021466249494, 5, 146, 1, 1),(432418202146
6249494, 6, 185, 1, 2);
```

4. 查询数据。

```
SELECT * FROM test_tbl_Versioned;
```

查询结果如下。

```
       UserID       |PageViews|Duration|Sign|Version
 ———————| ———|————|—|——
4324182021466249494|  5      | 146    | -1 | 1
4324182021466249494|  5      | 146    |  1 | 1
4324182021466249494|  6      | 185    |  1 | 2
```

5. 如果您需要对指定列进行聚合计算，以 `sum(col)` 为例，为了获得正确结果，需改写SQL语句如下。

```
SELECT
    UserID,
    sum(PageViews * Sign) AS PageViews,
    sum(Duration * Sign) AS Duration
FROM test_tbl_Versioned
GROUP BY UserID
HAVING sum(Sign) > 0;
```

进行聚合计算后，查询结果如下。

```
       UserID       |PageViews|Duration
 ———————| ———|————
4324182021466249494|  6      | 185
```

6. 由于MergeTree系列表引擎采用类似LSM Tree的结构，很多存储层处理逻辑直到Compaction期间才会发生，因此需执行optimize语句强制后台Compaction。

```
optimize table test_tbl_Versioned final;
```

7. 再次查询数据。

```
SELECT * FROM test_tbl_Versioned;
```

查询结果如下。

```
       UserID      |PageViews|Duration|Sign|Version
 ──────────| ────|────|──|──
4324182021466249494|  6      | 185    | 1  | 2
```

SummingMergeTree

SummingMergeTree表引擎用于对主键列进行预先聚合，将所有相同主键的行合并为一行，从而大幅度降低存储空间占用，提升聚合计算性能。

使用SummingMergeTree表引擎时，需要注意如下几点。

- 云数据库ClickHouse只在后台Compaction时才会对主键列进行预先聚合，而Compaction的执行时机无法预测，所以可能存在部分数据已经被预先聚合、部分数据尚未被聚合的情况。因此，在执行聚合计算时，仍需要使用 `GROUP BY` 子句。
- 在预先聚合时，云数据库ClickHouse会对主键列之外的其他所有列进行预聚合。如果这些列是可聚合的（比如数值类型），则直接sum求和。如果不可聚合（比如String类型），则会随机选择一个值。
- 通常建议将SummingMergeTree表引擎与MergeTree表引擎组合使用，MergeTree表引擎存储完整的数据，SummingMergeTree表引擎用于存储预先聚合的结果。

> **说明** SummingMergeTree表引擎的更多信息，具体请参见SummingMergeTree。

示例如下。

1. 创建表test_tbl_summing。

```
CREATE TABLE test_tbl_summing
(
    key UInt32,
    value UInt32
)
ENGINE = SummingMergeTree()
ORDER BY key;
```

2. 写入数据。

```
INSERT INTO test_tbl_summing Values(1,1),(1,2),(2,1);
```

3. 查询数据。

```
select * from test_tbl_summing;
```

查询结果如下。

```
key|value
— |──
 1 | 1
 1 | 2
 2 | 1
```

4. 由于MergeTree系列表引擎采用类似LSM Tree的结构，很多存储层处理逻辑直到Compaction期间才会发生，因此需执行optimize语句强制后台Compaction。

```
optimize table test_tbl_summing final;
```

5. 强制后台Compaction后，仍旧需要执行 `GROUP BY` 子句进行聚合计算，再次查询数据。

```
SELECT key, sum(value) FROM test_tbl_summing GROUP BY key;
```

查询结果如下，主键重复的数据已进行了聚合。

```
key|value
—|——
 1 | 3
 2 | 1
```

AggregatingMergeTree

AggregatingMergeTree表引擎也是预先聚合引擎的一种，用于提升聚合计算的性能。与SummingMergeTree表引擎的区别在于，SummingMergeTree表引擎用于对非主键列进行sum聚合，而AggregatingMergeTree表引擎可以指定各种聚合函数。

AggregatingMergeTree的语法比较复杂，需要结合物化视图或云数据库ClickHouse的特殊数据类型AggregateFunction一起使用。

> **说明** AggregatingMergeTree表引擎的更多信息，具体请参见AggregatingMergeTree。

示例如下。

- **结合物化视图使用**

i. 创建明细表visits。

```
CREATE TABLE visits
(
    UserID UInt64,
    CounterID UInt8,
    StartDate Date,
    Sign Int8
)
ENGINE = CollapsingMergeTree(Sign)
ORDER BY UserID;
```

ii. 对明细表visits建立物化视图visits_agg_view，并使用 `sumState` 和 `uniqState` 函数对明细表进行预先聚合。

```
CREATE MATERIALIZED VIEW visits_agg_view
ENGINE = AggregatingMergeTree() PARTITION BY toYYYYMM(StartDate) ORDER BY (CounterID,
StartDate)
AS SELECT
    CounterID,
    StartDate,
    sumState(Sign)    AS Visits,
    uniqState(UserID) AS Users
FROM visits
GROUP BY CounterID, StartDate;
```

iii. 写入数据至明细表visits中。

```
INSERT INTO visits VALUES(0, 0, '2019-11-11', 1);
INSERT INTO visits VALUES(1, 1, '2019-11-12', 1);
```

iv. 使用聚合函数 `sumMerge` 和 `uniqMerge` 对物化视图进行聚合，并查询聚合数据。

```
SELECT
    StartDate,
    sumMerge(Visits) AS Visits,
    uniqMerge(Users) AS Users
FROM visits_agg_view
GROUP BY StartDate
ORDER BY StartDate
```

> **说明** 函数 `sum` 和 `uniq` 不能再使用，否则会出现SQL报错：Illegal type AggregateFunction(sum, Int8) of argument for aggregate function sum...

查询结果如下。

```
StartDate |Visits|Users
──────|──|─
2019-11-11| 1    | 1
2019-11-12| 1    | 1
```

- **结合特殊数据类型AggregateFunction使用**

i. 创建明细表detail_table。

```
CREATE TABLE detail_table
(   CounterID UInt8,
    StartDate Date,
    UserID UInt64
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(StartDate)
ORDER BY (CounterID, StartDate);
```

ii. 写入数据至明细表detail_table中。

```
INSERT INTO detail_table VALUES(0, '2019-11-11', 1);
INSERT INTO detail_table VALUES(1, '2019-11-12', 1);
```

iii. 创建聚合表agg_table，其中 `UserID` 列的类型为AggregateFunction。

```
CREATE TABLE agg_table
(   CounterID UInt8,
    StartDate Date,
    UserID AggregateFunction(uniq, UInt64)
) ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMM(StartDate)
ORDER BY (CounterID, StartDate);
```

iv. 使用聚合函数 `uniqState` 将明细表的数据插入至聚合表中。

```
INSERT INTO agg_table
select CounterID, StartDate, uniqState(UserID)
from detail_table
group by CounterID, StartDate;
```

> **说明** 不能使用 `INSERT INTO agg_table VALUES(1, '2019-11-12', 1);` 语句向聚合表插入数据，否则会出现SQL报错：Cannot convert UInt64 to AggregateFunction(uniq, UInt64)...

v. 使用聚合函数 `uniqMerge` 对聚合表进行聚合，并查询聚合数据。

```
SELECT uniqMerge(UserID) AS state
FROM agg_table
GROUP BY CounterID, StartDate;
```

查询结果如下。

```
state
─────
 1
 1
```

6.MaterializeMySQL引擎

本文介绍如何使用MaterializeMySQL引擎将MySQL（自建MySQL或RDS MySQL）数据同步到云数据库ClickHouse。

背景信息

为了强化实时数仓的能力，云数据库ClickHouse推出了MaterializeMySQL数据库引擎，云数据库ClickHouse服务作为MySQL副本，读取Binlog并执行DDL和DML请求，实现了基于MySQL Binlog机制的业务数据库实时同步功能。

功能优势

云数据库ClickHouse自主研发了如下功能。

- 通过配置同步表名单或排除表名单，实现部分数据库表的同步。
- 修改同步配置项。
- 跳过同步错误。
- 跳过不支持的表结构。
- 控制云数据库ClickHouse表结构 `ORDER BY` 的生成规则。
- 开启Binlog预留功能，避免Binlog清理导致的无法正常同步数据的问题，从而提升MaterializeMySQL引擎的稳定性。
- 控制物化库是否按照分布式模式将MySQL数据同步至云数据库ClickHouse。
- 控制全量同步时每秒最多拉取的行数。

注意事项

- 云数据库ClickHouse的云原生版集群不支持MaterializeMySQL引擎。
- 云数据库ClickHouse的社区兼容版集群仅20.8及以上版本支持MaterializeMySQL引擎。
- 使用MaterializeMySQL引擎查询物化库的表时需要添加FINAL修饰符以确保查询的结果与RDS MySQL一致。
- MaterializeMySQL引擎无法同步以下两种类型的表：无主键表和 `create as select` 语法创建的表。
- 数据源RDS MySQL集群和目标ClickHouse集群必须属于同一个VPC网络。
- 数据源RDS MySQL集群的内核版本是5.6及以上版本，且必须开启全局事务标识（GTID）。
- 数据源RDS MySQL的DDL语句如果不是标准的MySQL语法，有可能导致无法同步。
- 同步RDS MySQL数据时需要将ClickHouse集群所使用的交换机网段添加到RDS MySQL的白名单中。更多信息，请参见设置IP白名单。
- 配置MaterializeMySQL引擎所使用的MySQL账号必须具备MySQL库的RELOAD、REPLICATION SLAVE、REPLICATION CLIENT以及SELECT PRIVILEGE权限。

  > **说明** 您可以通过 `GRANT RELOAD, REPLICATION SLAVE, REPLICATION CLIENT, SELECT ON *.* TO 'your-user-name';` 命令获取MySQL库的相关权限。

- 不能随意修改云数据库ClickHouse中的表结构和数据。
- 云数据库ClickHouse的 `DataTime` 类型数据的时间范围为 `[1970-01-01 00:00:00,2106-02-07 06:28:15]`，如果RDS MySQL中的时间超出当前范围，会导致同步到云数据库ClickHouse的时间值不正确。

使用MaterializeMySQL引擎

为了将MySQL服务器中的表映射到云数据库ClickHouse中，您需要创建MaterializeMySQL引擎的数据库。

语法

您可以在创建数据库时配置数据库引擎类型为MaterializeMySQL并配置相关参数，语法如下。

```
CREATE DATABASE [IF NOT EXISTS] db_name [ON CLUSTER cluster]
ENGINE = MaterializeMySQL('host:port', 'database', 'user', 'password')
[SETTINGS...]
where SETTINGS are:
[ { include_tables | exclude_tables } ]
[ skip_error_count ]
[ skip_unsupported_tables ]
[ order_by_only_primary_key ]
[ enable_binlog_reserved ]
[ shard_model ]
[ rate_limiter_row_count_per_second ]
```

引擎参数说明

参数	说明
host:port	MySQL数据库的URL和端口号。
database	MySQL数据库名称。
user	MySQL数据库账号。
password	MySQL数据库账号的密码。

引擎配置项说明

配置项	类型	说明
include_tables	String	配置同步表名单。配置同步表名单后，则只有表名单内的表会被同步。多个表之间，用","分隔。 模糊匹配规则： • *：替换除/和空字符串以外的任意数量的任何字符。 • ?：替换任意单个字符。 • {N..M}：替换N到M范围内的任何数字，包括两个边界。

配置项	类型	说明
exclude_tables	String	配置排除表名单。排除表名单内的表不进行同步。多个表之间，用","分隔。 模糊匹配规则： • *：替换除/和空字符串以外的任意数量的任何字符。 • ?：替换任意单个字符。 • {N..M}：替换N到M范围内的任何数字，包括两个边界。 **说明** exclude_tables和include_tables两个配置项不能同时使用。
skip_error_count	Int	是否跳过同步错误。取值范围如下： • -9223372036854775808~-1：跳过所有错误。 • 0：默认值，不跳过任何一个错误。 • 1~9223372036854775807：跳过对应数字的错误。
skip_unsupported_tables	Boolean	是否跳过MaterializeMySQL引擎目前无法同步的MySQL表。取值范围如下： • 0：不跳过不支持的表结构，即如果遇到不能同步的表结构，同步会失败。 • 1：默认值，跳过不支持的表结构，即如果遇到不能同步的表结构，跳过并继续进行同步。 **说明** 该参数优先级高于skip_error_count参数，如果设置了skip_unsupported_tables，skip_error_count将不再生效。
order_by_only_primary_key	Boolean	云数据库ClickHouse表结构 `ORDER BY` 的生成规则。取值范围如下： • 0：将MySQL主键和索引都转换为 `ORDER BY` 元组。 • 1：默认值，只将MySQL主键转换为 `ORDER BY` 元组。
enable_binlog_reserved	Int	是否开启Binlog预留功能。取值范围如下： • 0：关闭Binlog预留功能。 • 1：默认值，开启Binlog预留功能。 **说明** 如果开启Binlog预留功能，可能会延迟MySQL数据同步至云数据库ClickHouse。

配置项	类型	说明
shard_model	Boolean	物化库是否按照分布式模式将MySQL数据同步至云数据库ClickHouse。取值范围如下： • 0：默认值，一个MySQL表对应一个云数据库ClickHouse本地表。适用于只有一个节点的云数据库ClickHouse集群。 这种模式下，如果云数据库ClickHouse集群有多个节点，那么MySQL服务器会同步数据到每个节点，导致MySQL服务器压力较大。 • 1：分布式模式，一个MySQL表对应一个云数据库ClickHouse本地表和一个分布式表，其中分布式表和MySQL表同名。查询同步数据时，需要查询分布式表。适用于有多个节点的云数据库ClickHouse集群。
rate_limiter_row_count_per_second	Int	全量同步时每秒最多拉取的行数。取值范围如下： • 0：默认值，不限制拉取的行数。 • 1~9223372036854775807：每秒最多拉取对应数字的行数。例如取值为1，最多拉取1行。

说明

- 如果您配置如上参数时提示参数不支持一类错误，请提交工单升级版本至最新版本。
- MaterializeMySQL引擎所有的配置项都可以进行修改，修改后立即生效。语法如下。

```
ALTER DATABASE <database> MODIFY SETTING
```

示例

```
CREATE DATABASE IF NOT EXISTS db_name ON CLUSTER cluster
ENGINE = MaterializeMySQL('rm-bp1h40w639t19****.mysql.rds.aliyuncs.com:3306', 'database', '
user', 'password')
SETTINGS
include_tables ='a,b,c';
```

使用细则

虚拟字段

使用MaterializeMySQL在云数据库ClickHouse集群上新建ReplacingMergeTree引擎表时，会默认在表中增加两个虚拟字段。

字段	类型	说明
_version	UInt64	事务计数器，记录数据版本信息。

字段	类型	说明
_sign	TypeInt8	删除标记，标记该行是否删除。取值范围如下： • 1：该行未删除。 • -1：该行已删除。 **说明** 带有 `_sign=-1` 的行不会从表中物理删除。

DDL语句转换

MaterializeMySQL不支持直接插入、删除和更新查询，MySQL DDL语句将被转换成相应的ClickHouse DDL语句：

- MySQL INSERT查询被转换为 `INSERT with _sign=1` 。
- MySQL DELETE查询被转换为 `INSERT with _sign=-1` 。
- MySQL UPDATE查询被转换成 `INSERT with _sign=1` 和 `INSERT with _sign=-1` 。

说明
- 如果ClickHouse不能解析某些DDL语句，该语句将被忽略。
- MaterializeMySQL引擎不支持级联UPDATE/DELETE查询。

SELECT查询

- 如果在SELECT查询中没有指定_version，则使用FINAL修饰符，返回_version的最大值对应的数据，即最新版本的数据。
- 如果在SELECT查询中没有指定_sign，则默认使用 `WHERE _sign=1` ，即返回未删除状态 `(_sign=1)` 的数据。

索引转换

- ClickHouse数据库表会自动将MySQL主键和索引子句转换为 `ORDER BY` 元组。
- ClickHouse只有一个物理顺序，由 `ORDER BY` 子句决定。如果需要创建新的物理顺序，请使用物化视图。

字段类型对照表

下表介绍MySQL与云数据库ClickHouse的字段类型对应关系。

MySQL	ClickHouse
TINY	Int8
SHORT	Int16
INT24	Int32
LONG	UInt32
LONG	UInt64

MySQL	ClickHouse
FLOAT	Float32
DOUBLE	Float64
DECIMAL，NEWDECIMAL	Decimal
DATE，NEWDATE	Date
DATETIME，TIMESTAMP	DateTime
DATETIME2，TIMESTAMP2	DateTime64
STRING	String
VARCHAR，VAR_STRING	String
BLOB	String
BIT	UInt64
SET	UInt64
ENUM	Enum16
YEAR	String
TIME	String
GEOMETRY	String
JSON	不支持

同步RDS MySQL数据

1. **创建数据库和表**

   i. 在RDS MySQL中创建数据库和表。

   ```
   CREATE DATABASE cktest;
   CREATE TABLE cktest.test01 (a INT PRIMARY KEY, b INT);
   ```

   ii. 在云数据库ClickHouse中创建同步数据库。

   ```
   CREATE DATABASE mysql ENGINE = MaterializeMySQL('rm-bp1h40w639t19****.mysql.rds.aliyuncs.com:3306', 'cktest', 'user', 'password');
   ```

iii. 在云数据库ClickHouse中查询同步库中的表。

```
SHOW TABLES FROM mysql;
```

结果如下。

```
+--------+
| name   |
+--------+
| test01 |
+--------+
```

2. **插入数据并查询**

i. 在RDS MySQL中插入数据。

```
INSERT INTO cktest.test01 VALUES (1, 10), (2, 20);
```

ii. 在云数据库ClickHouse中查询。

```
SELECT * FROM mysql.test01;
```

结果如下。

```
+----+----+
| a  | b  |
+----+----+
| 1  | 10 |
+----+----+
| 2  | 20 |
+----+----+
```

3. **删除数据，添加列并查询**

i. 在RDS MySQL中删除数据，添加列并更新数据进行查询。

```
DELETE FROM cktest.test01 WHERE a=1;
ALTER TABLE cktest.test01 ADD COLUMN c VARCHAR(16);
UPDATE cktest.test01 SET c='ok!', b=202;
SELECT * FROM test01;
```

结果如下。

```
+---+-----+-----+
| a | b   | c   |
+---+-----+-----+
| 2 | 202 | ok! |
+---+-----+-----+
```

ii. 在云数据库ClickHouse中查询。

```
SELECT * FROM mysql.test01;
```

结果如下。

```
+----+-----+-----+
| a  | b   | c   |
+----+-----+-----+
| 2  | 202 | ok! |
+----+-----+-----+
```

查询同步详情

您可以通过查询系统表system.materialize_mysql，获取使用MaterializeMySQL引擎同步MySQL数据的情况。查询语句如下。

- 集群中只有一个节点。

```
select * from system.materialize_mysql;
```

- 集群中有多个节点。

```
select * from cluster(default, system, materialize_mysql);
```

查询结果如下。

```
+----------+----------------+-------------+------------------+--------------------+
| database |     status     | error_count | lasted_error_msg | sync_failed_tables |
+----------+----------------+-------------+------------------+--------------------+
|  mysql   | increment_sync |      0      |                  |                    |
+----------+----------------+-------------+------------------+--------------------+
```

查询结果字段说明如下。

字段名	说明
database	云数据库ClickHouse中同步的数据库。
status	同步状态，取值说明如下。 • preparing_sync：同步准备阶段。 • full_sync：全量同步阶段。 • increment_sync：增量同步阶段。 • stop_sync_sync：同步停止。
error_count	同步过程中出错的次数。
lasted_error_msg	最近一次同步错误的详情。
sync_failed_tables	同步失败的表。多个表之间，以","分隔。

常见问题

问题1：使用MaterializeMySQL引擎同步MySQL数据时，为什么出现如下报错：`The slave is connecting using CHANGE MASTER TO MASTER_AUTO_POSITION = 1, but the master has purged binary logs containing GTIDs that the slave requires`？

常见原因：MaterializeMySQL引擎停止同步的时间太久，导致MySQL Binlog日志过期被清理掉。

解决方案：删除报错的数据库，重新在云数据库ClickHouse中创建同步的数据库。

问题2：使用MaterializeMySQL引擎同步MySQL数据时，为什么出现表停止同步？为什么系统表system.materialize_mysql中`sync_failed_tables`字段不为空？

常见原因：同步过程中使用了云数据库ClickHouse不支持的MySQL DDL语句。

解决方案：重新同步MySQL数据，具体步骤如下。

1. 删除停止同步的表。

```
drop table <table_name> on cluster default;
```

> **说明** `table_name`为停止同步的表名。如果停止同步的表有分布式表，那么本地表和分布式表都需要删除。

2. 重启同步进程。

```
alter database <database_name> on cluster default MODIFY SETTING skip_unsupported_tables = 1;
```

> **说明** `<database_name>`为云数据库ClickHouse中同步的数据库。

7.ClickHouse资源队列

本文介绍如何使用云数据库ClickHouse的资源队列增强功能。

背景信息

ClickHouse官方开源版目前没有资源队列的功能设计，以下资源队列相关功能是云数据库ClickHouse的增强功能，并且只适配于20.3的内核版本。由于ClickHouse官方开源版有一套完整的用户级别内存隔离机制存在，资源队列功能在新购库中默认不开启，需要使用资源队列功能的用户可通过提交工单自助开启。

使用限制

- 云数据库ClickHouse的云原生版不支持资源队列增强功能。
- 云数据库ClickHouse的社区兼容版仅20.3版本支持资源队列增强功能。

资源队列定义

创建资源队列语法：

```
-- 创建资源队列定义
CREATE RESOURCE QUEUE [IF NOT EXISTS | OR REPLACE] name [ON CLUSTER cluster]
  * {[SET] MEMORY = {number}
  * [, CONCURRENCY = {number}]
  * [, PRIORITY = { LOWEST | LOW | NORMAL | HIGH | HIGHEST }]
  * [, ISOLATE = {number}]
    * }
  * [TO {role [,...] | ALL | ALL EXCEPT role [,...]}]
-- 更改指定资源队列定义
ALTER RESOURCE QUEUE [IF NOT EXISTS | OR REPLACE] name [ON CLUSTER cluster]
  * {[SET] MEMORY = {number}
  * [, CONCURRENCY = {number}]
  * [, PRIORITY = { LOWEST | LOW | NORMAL | HIGH | HIGHEST }]
  * [, ISOLATE = {number}]
    * }
  * [TO {role [,...] | ALL | ALL EXCEPT role [,...]}]
-- 查看指定资源队列定义
SHOW CREATE resource queue name
-- 查看当前用户使用的资源队列定义
SHOW CREATE resource queue current
-- 删除指定资源队列定义
DROP resource queue if exists name
```

定义参数说明：

- memory：声明创建的资源队列内存池大小，如果节点的总内存被已有的资源队列分配完，则创建会失败。
- concurrency：声明创建的资源队列中查询的并发数限制，默认值为*20*，并发限制会对用户发起的查询进行阻塞排队，对系统发起的子查询不会阻塞但是会计入到当前并发统计中。即并发限制为20的资源队列当前同时在跑的系统子查询可以达到25个，但此时用户发起的查询全部会被阻塞，直到系统子查询数降低到20以下。
- priority：声明创建的资源队列优先级，包含cpu调度优先级和内存抢占优先级。
- isolate：声明创建的资源队列内存隔离级别。

- 默认值是*0*，表示没有隔离，当目标资源队列中的内存使用率处于低水位时，更高优先级的资源队列在内存不够用时可以临时抢占目标资源队列的内存。
- 设定为*1*时，表示该资源队列有软隔离，不可被高优先级队列抢占内存。
- 设定为*2*时，表示该资源队列绝对隔离，不可被抢占内存，同时也不会去抢占低优先级队列的内存。

- role：通过 `To role [,...]` 声明，把目标资源队列绑定到多个用户上，对应用户的查询默认会进入到目标资源队列中。当同时有多个资源队列绑定同一用户时，系统会随机选择一个最高优先级的资源队列执行查询。

配置参数说明：

- target_resource_queue：声明目标查询强制路由到指定资源队列中，也可以在用户的profile文件中配置强制用户对应的查询强制路由。
- resource_queue_max_wait_ms：声明查询在资源队列中因为并发限制阻塞排队的最长等待时间，默认10s。

示例：

```
CREATE RESOURCE QUEUE IF NOT EXISTS test_queue ON CLUSTER cluster SET
  MEMORY = 1073741824, CONCURRENCY = 20, ISOLATE = 0, PRIORITY = NORMAL
    TO default;
CREATE RESOURCE QUEUE IF NOT EXISTS anonymous_queue ON CLUSTER cluster SET
  MEMORY = 1073741824, CONCURRENCY = 20, ISOLATE = 1, PRIORITY = LOW;
SHOW CREATE resource queue test_queue;
SHOW CREATE resource queue current;
SELECT count (distinct intDiv(number, 10)) FROM numbers(100000) settings target_resource_qu
eue='anonymous_queue';
DROP resource queue if exists test_queue;
DROP resource queue if exists anonymous_queue;
```

资源队列状态查询

List当前所有的资源队列定义：

```
show resource queues;
```

返回结果说明：

名称	类型	说明
name	String	资源队列名。
concurrency	UInt32	资源队列的并发限制。
memory	UInt64	资源队列的内存池大小。
isolate	UInt8	资源队列的隔离级别。
priority	Enum8（枚举）	资源队列的优先级别。
roles	Array<String>	资源队列绑定的所有相关用户。

List指定资源队列的查询使用状态：

```
show resource queue stat [CURRENT | ALL];
```

返回结果说明：

名称	类型	说明
name	String	资源队列名。
running_query	UInt32	资源队列中正在执行的查询数量。
waiting_query	UInt32	资源队列中正在等待执行的查询数量。
grabbing_query	UInt32	正在临时抢占使用资源队列内存池的查询（不属于该资源队列）数量。
allocated_memory	UInt64	资源队列内存池被正常申请使用的内存大小。
grabbed_memory	UInt64	资源队列内存池被临时抢占使用的内存大小。
free_memory	UInt64	资源队列内存池当前剩余内存大小。

资源队列相关查询异常

并发限制导致查询超时

相关错误码：13005

解决方案：降低客户端的查询并发数。

Bad Query使用内存超过资源队列内存池

相关错误码：241

解决方案：优化查询Plan，或调大资源队列内存池。

查询强制路由的资源队列不存在

相关错误码：13006

解决方案：检查资源队列创建状态。

查询临时抢占申请低优先级资源队列被Kill

相关错误码：394

解决方案：降低资源队列并发数，调整队列优先级。

8.ClickHouse二级索引

本文介绍如何使用云数据库ClickHouse的二级索引增强功能。

背景信息

ClickHouse官方开源版目前没有二级索引的功能设计，以下二级索引相关功能是云数据库ClickHouse的增强功能，并且只适配于20.3和20.8的内核版本。这里的二级索引和官方开源版的索引并不是一个原理，解决的也不是同一类问题，二级索引最常见的使用场景是对根据非排序键的等值条件进行点查加速。

使用限制

- 云数据库ClickHouse的云原生版不支持二级索引增强功能。
- 云数据库ClickHouse的社区兼容版仅20.3和20.8版本支持二级索引增强功能。

二级索引语法

二级索引在创建表时的定义语句示例如下：

```
CREATE TABLE index_test
(
id UInt64,
d DateTime,
x UInt64,
y UInt64,
tag String,
KEY tag_idx tag TYPE range,
KEY d_idx toStartOfHour(d) TYPE range,
KEY combo_idx (toStartOfHour(d), x, y) TYPE range
) ENGINE = MergeTree() ORDER BY id;
```

二级索引相关的修改DDL如下：

```
--删除索引定义
Alter table index_test DROP KEY tag_idx;
--增加索引定义
Alter table index_test ADD KEY tag_idx tag TYPE range;
--清除数据分区下的索引文件
Alter table index_test CLEAR KEY tag_idx in partition partition_expr;
--重新构建数据分区下的索引文件
Alter table index_test MATERIALIZE KEY tag_idx in partition partition_expr;
```

二级索引特性

ClickHouse的二级索引支持多索引列条件交并差检索。总体特点概括如下：

- 多列联合索引 & 表达式索引
- 函数下推
- In Set Clause下推
- 高压缩比 （索引文件大小接近Lucene 8.7）
- 向量化构建 （构建速度对比Lucene 8.7提升4倍）

多列联合索引的目的是减少特定查询pattern下的索引结果归并，针对QPS要求特别高的查询用户可以创建针对性的多列联合索引达到极致的检索性能。而表达式索引主要是方便用户进行自由的检索粒度变换，考虑以下两个典型场景：

- 二级索引中的时间列在搜索条件中，只会以小时粒度进行过滤，这种情况下用户可以对 `toStartOfHour(time)` 表达式创建索引，可以一定程度加速索引构建，同时对time列的时间过滤条件都可以自动转换下推索引。
- 二级索引中的ID列是由UUID构成，UUID几乎是可以保证永久distinct的字符串序列，直接对ID构建索引会导致索引文件太大。这时用户可以使用前缀函数截取UUID来构建索引，如 `prefix8(id)` 是截取8个byte的前缀，对应的还有prefix4和prefix16。prefixUTF4、prefixUTF8、prefixUTF16则是用来截取UTF编码的。

用户对表达式构建索引后，原列上的查询条件也可以正常下推索引，不需要特意改写查询。同样用户对原列构建索引，过滤条件上对原列加了表达式的情况下，优化器也都可以正常下推索引。

In Set Clause下推则是一个关联搜索的典型场景，经常有用户碰到此类场景：user的属性是一张单独的大宽表，user的行为记录又是另一张单独的表，对user的搜索需要先从user行为记录表中聚合过滤出满足条件的user id，再用user ids从属性表中取出明细记录。这种in subquery的场景下，ClickHouse也可以自动下推二级索引进行加速。

二级索引构建性能

以下介绍的是ClickHouse二级索引构建性能和索引压缩率的测试结果，主要对比的是Lucene 8.7的倒排索引和BKD索引。索引构建时间都是基于单线程作业统计的。

场景一：UUID字符串

```
CREATE TABLE string_index_test (
 `C_KEY` String,
 KEY C_KEY_IDX C_KEY Type range
) ...
INSERT INTO string_index_test
select substringUTF8(cast (generateUUIDv4() as String), 1, 16)
from system.numbers limit 100000000;
```

测试结果如下：

测试对象	索引构建耗时	索引文件大小	数据文件大小
ClickHouse	97.2s	1.3G	1.5G
Lucene	487.255s	1.3G	N/A

场景二：枚举字符串

```
CREATE TABLE string_index_test (
 `C_KEY` LowCardinality(String),
 KEY C_KEY_IDX C_KEY Type range
) ...
INSERT INTO string_index_test
select cast((10000000 + rand(number) % 4000) as String)
from system.numbers limit 100000000;
```

测试对象	索引构建耗时	索引文件大小	数据文件大小
ClickHouse	25.5s	187M	193M
Lucene	45.513s	187M	N/A

场景三：随机数值

```
CREATE TABLE long_index_test (
  `C_KEY` UInt64,
  KEY C_KEY_IDX C_KEY Type range
) ...
INSERT INTO long_index_test
select rand(number) % 100000000
from system.numbers limit 100000000;
```

测试对象	索引构建耗时	索引文件大小	数据文件大小
ClickHouse	34.2s	615M	519M
Lucene	81.971s	482M	N/A

场景四：枚举数值

```
CREATE TABLE int_index_test (
  `C_KEY` UInt32,
  KEY C_KEY_IDX C_KEY Type range
) ...
INSERT INTO int_index_test
select rand(number) % 1000
from system.numbers limit 100000000;
```

测试对象	索引构建耗时	索引文件大小	数据文件大小
ClickHouse	12.2s	163M	275M
Lucene	77.999s	184M	N/A

二级索引等值查询性能

测试环境：32core 128G，内存 ecs，1T PL1 ESSD

测试数据：总数据行数13E，表结构如下：

```
CREATE TABLE point_search_test (
 `PRI_KEY` String,
 `SED_KEY` String,
 `INT_0` UInt32,
 `INT_1` UInt32,
 `INT_2` UInt32,
 `INT_3` UInt32,
 `INT_4` UInt32,
 `LONG_0` UInt64,
 `LONG_1` UInt64,
 `LONG_2` UInt64,
 `LONG_3` UInt64,
 `LONG_4` UInt64,
 `STR_0` String,
 `STR_1` String,
 `STR_2` String,
 `STR_3` String,
 `STR_4` String,
 `FIXSTR_0` FixedString(16),
 `FIXSTR_1` FixedString(16),
 `FIXSTR_2` FixedString(16),
 `FIXSTR_3` FixedString(16),
 `FIXSTR_4` FixedString(16),
 KEY SED_KEY_IDX SED_KEY Type range
) ...
```

二级索引等值查询QPS如下：

测试对象	select *	select INT_0	select LONG_0	select STR_0	select FIXSTR_0
冷启动查询QPS	600	3200	3700	3600	3700
预热后查询QPS	4900	27000	27000	26000	26000

二级索引极致性能推荐配置

对二级索引性能有极致要求的用户推荐购买32core 128G内存的ECS规格实例，同时挂载PL2级别的ESSD硬盘。

实例全局参数设置：min_compress_block_size = 4096, max_compress_block_size = 8192。

MergeTree表级别的参数设置：调整min_bytes_for_compact_part参数，优先使用Compact Format。