ClickHouse调研报告

# ClickHouse基本介绍

ClickHouse是一款高速的，列式数据管理系统。主要针对于OLAP（online analysis processing 线上分析处理过程）进行数据的分析和聚合，支持80%的SQL语法，以及多种引擎（库、表都有对应的引擎）。

列式存储与行式存储的区别

行式存储：将数据以一行的形式进行保存，所有相关的数据都在物理意义上的同一行上，能有效的保证数据一致性。

列式存储：将数据以列的形式进行保存，相同的列就可进行合并，减少存储空间。

相关的UI界面地址：ui.tabix.io/

ClickHouse是不支持多目录的，除非自己虚拟化多个卷组

# ClickHouse基本使用场景

## **ClickHouse能够支持OLAP的场景**

首先OLAP的特点如下：

数据库体量非常大，达到数十亿乃至数万亿级别的数据体量。

数据表中包含了大量的列

只有少部分的列被选中参与计算

结果的返回必须是毫秒级或秒级的

## 为什么列式存储更适合OLAP呢

1.行式存储在进行数据统计，数据计算时会先根据条件筛选出相关的所有行数据，在进行一定的聚合。

2.列式存储可以直接针对某一列进行快速聚合，不需要先将数据筛选出来，而直接可以再列进行筛选，达到快速聚合的效果。

## ClickHouse适用场景

1.读多写少的场景，且常用于数据的聚合与计算

2.无需数据更新、删除，更多偏向于数据的插入和查询。

3.对数据的一致性要求较低（非强一致性），无需事务的依赖。

4.查询并发能力相对要求较低，最好稳定在qps<100

5.尽量进行大而宽的表进行查询，最后无需联表。

# ClickHouse优缺点

优点：

为了高效的使用CPU，数据不仅仅按列存储，同时还按向量进行处理；

数据压缩空间大，减少IO；处理单查询高吞吐量每台服务器每秒最多数十亿行；

索引非B树结构，不需要满足最左原则；只要过滤条件在索引列中包含即可；即使在

用的数据不在索引中，由于各种并行处理机制ClickHouse全表扫描的速度也很快；

对函数有一定的优化，减少性能损失。对聚合函数有天然的优势。

写入速度非常快，50-200M/s，对于大量的数据更新非常适用；

缺点：

不支持事务，不支持真正的删除/更新；

不支持高并发，官方建议qps为100，可以通过修改配置文件增加连接数，但是在服务器足够好的情况下；

低版本不支持join操作，高版本支持join但是性能极差，建议使用大宽表，尽可能不要链表；

尽量做1000条以上批量的写入，避免逐行insert或小批量的insert，update，delete操作，因为ClickHouse底层会不断的做异步的数据合并，会影响查询性能，这个在做实时数据写入的时候要尽量避开；

Clickhouse快是因为采用了并行处理机制，即使一个查询，也会用服务器一半的CPU去执行，所以ClickHouse不能支持高并发的使用场景，默认单查询使用CPU核数为服务器核数的一半，安装时会自动识别服务器核数，可以通过配置文件修改该参数。

|  |  |
| --- | --- |
| 参数 | 描述 |
| background\_pool\_size | 后台线程池的大小，merge线程就是在该线程池中执行,该线程池不仅仅是给merge线程用的，默认值是16，允许的前提下建议改成 **cpu 个数的2倍（ 线程数**） |
| background\_schedule\_pool\_size | 执行后台任务（复制表、 kafka流 、DNS 缓存更新） 的线程数。默认 128, 建议改成 cpu 的 2 倍。 |
| background\_distributed\_schedule\_pool\_size | 设置为分布式发送执行后台任务的线程数， 默认 6 ，建议改成 **CPU 个数的 2 倍（线程数）** |
| max\_concurrent\_queries | 最大并发处理的请求数（包含 select，insert等 ），默认值 100 ，推荐 150（不够再加）~ 300 。 |
| max\_thread | 设置单个查询所能使用的最大 CPU 个数，默认是CPU 核数。 |

Clickhouse相关的数据导入方式（类比物化视图）：

全量数据导入：数据导入临时表 -> 导入完成后，将原表改名为tmp1 -> 将临时表改名为正式表 -> 删除原表

增量数据导入： 增量数据导入临时表 -> 将原数据除增量外的也导入临时表 -> 导入完成后，将原表改名为tmp1-> 将临时表改成正式表-> 删除原数据表

# ClickHouse与其他数据库对比

一下是参考ClickHouse官方给出的性能对比图；（没图，有链接）

<https://clickhouse.yandex/benchmark.html>

知乎上搜索到的西能测试对比，只对比关于OLAP类型数据库

<https://zhuanlan.zhihu.com/p/54907288>

# ClickHouse的数据类型（类比Mysql）

## 常用字段类型

|  |  |
| --- | --- |
| ClickHouse | Mysql |
| Int8 | bit |
| Int16 | tinyint |
| Int32 | int |
| Int64 | bigint |
| UInt8 | unsigned bit |
| UInt8 | unsigned tinyint |
| UInt8 | unsigned int |
| UInt8 | unsigned bigint |
| Float32 | float |
| Float64 | double |
| decimal32(s) | decimal(9-s,s) |
| decimal32(s) | decimal(18-s,s) |
| decimal32(s) | decimal(38-s,s) |
| String | varchar |
| FixedString(N) | char(固定长度字符串，会用空字符串进行补全) |
| Date | Date |
| DateTime | DateTime |

## ClickHouse特有字段类型

|  |  |  |
| --- | --- | --- |
| ClickHouse固有类型 | 类型 | 解释 |
| Enum8 | 枚举；String == Int8 | 类似于字符串与整形的对应转换； |
| Enum16 | 枚举；String == Int16 | 类似于字符串与整形的对应转换； |
| Array(T) | 数组；T=数组长度 | 使用一个字段进行对数组的保存，应用不多，而且限定引擎 |
| Tuple(T) | 元组； T=元组长度 | 和数组类似，限定引擎 |

# ClickHouse常用表引擎介绍

## LogFamily

以文件保存在磁盘上，不支持索引，一般都是一些小表会使用

## Special

Memory:

基于内存进行数据存储，宕机或重启后数据全无，但是性能表现极佳。不建议使用

## **MergeTreeFimily**

ClickHouse最常用的引擎家族。具有一系列特点和功能，也有根据场景进行变换的各种类型的存储引擎。

建表语句（除了order by 其他参数均为可选。order by最左测元素推荐为主键）

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],

...

PROJECTION projection\_name\_1 (SELECT <COLUMN LIST EXPR> [GROUP BY] [ORDER BY]),

PROJECTION projection\_name\_2 (SELECT <COLUMN LIST EXPR> [GROUP BY] [ORDER BY])

) ENGINE = MergeTree()

ORDER BY expr

[PARTITION BY expr]

[PRIMARY KEY expr]

[SAMPLE BY expr]

[TTL expr

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

[WHERE conditions]

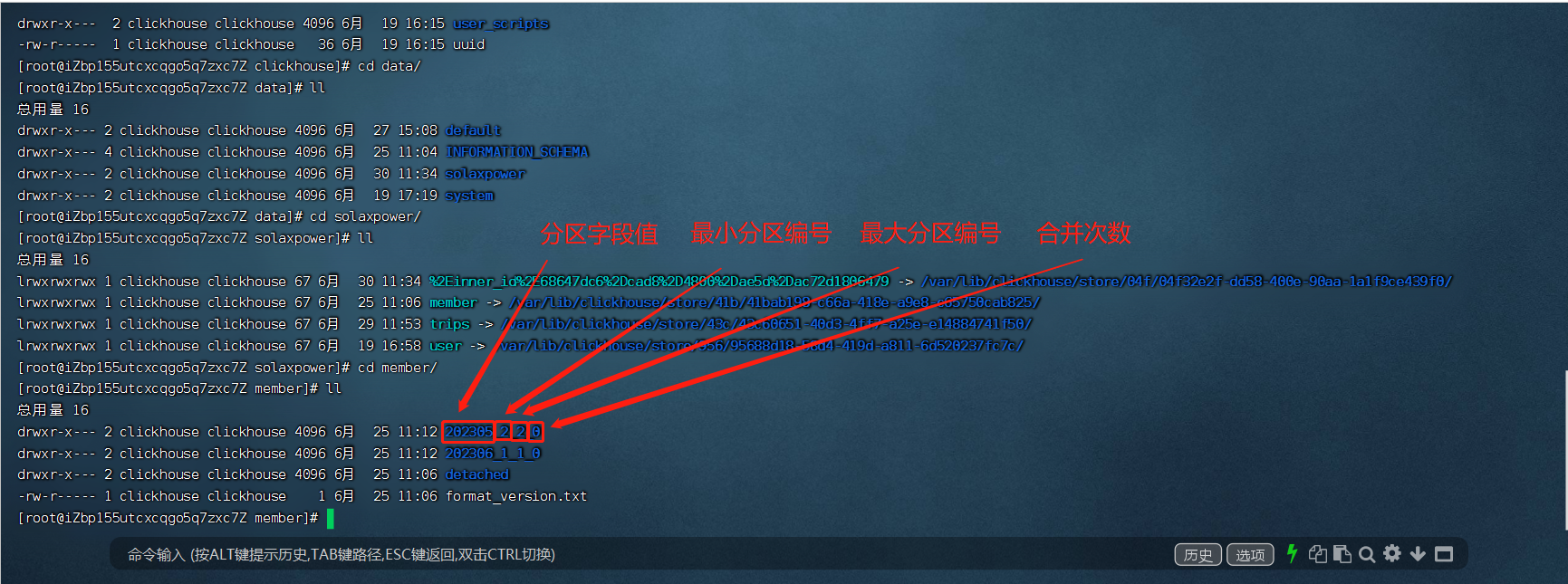
[GROUP BY key\_expr [SET v1 = aggr\_func(v1) [, v2 = aggr\_func(v2) ...]] ] ]

[SETTINGS name=value, ...]

### 数据分区（PARTITION BY）

数据分区是mergeTree家族的重要参数，会根据一定的字段对数据进行分区，从而来提高查询效率。因为在ClickHouse中查询是多线程的，所以会根据分区来进行线程的开启查询。

分区的目录结构如下图：



数据分区编号规则：分区值（分区ID）\_最小分区块编号\_最大分区块编号\_合并层级

分区ID

常规我们会使用Int或者Date进行取分区，假设，我们选取String或Float，则会通过128位Hash算法取其Hash值作为分区id，产生Hash碰撞则为统一分区

最小分区编号

自增，没啥用

最大分区编号

合并分区后最大的编号

合并层级

其实就是合并的次数

分区数据写入合并

CK每次插入都会有一个临时分区，之后10-15分钟进行数据合并，其中会将原有的数据过期，直接使用新的分区数据。

手动触发合并（慎用）optimize table [table\_name] [partition] [partitionID] final

### 主键（PRIMARY KEY）和索引（INDEX）

主键本身自带索引，但是不想关系型数据库会加上唯一约束，本身是一个稀疏索引（根据粒度进行区分，默认粒度为8192）并行查询，启动线程针对每个分区进行查询。

二级索引针对稀疏索引在进行一次聚合，在查找的时候先去查询一次二级索引的区间，在查询稀疏索引的区间达到快速查询的效果。GRANULARITY是对应稀疏索引的颗粒度大小，建议根据业务情况去确定，因为索引本身受一级索引的影响。

### 数据TTL(time to live)

根据一定的周期进行数据清理，适合实时场景，禁止使用key为TTL；

对某一列添加TTL：

TTL [时间字段或时间值] + interval [时间数值] [时间单位]

TTL也可以进行移动，和数据归档。默认情况下为删除例如：

TTL create\_time + interval 1 month to DISK 'a'

### 家族成员

ReplacingMergeTree

* 会根据order by的内容进行去重，去重的时机在合并的过程中去进行去重
* 建表语句的情况下，engine=ReplacingMergeTree(create\_time)，其中create\_time会取值最大的数据当做当前数据。
* 每次插入的时候会进行分区内去重，但是临时分区是不去重的

SummingMergeTree

* 根据分区内进行聚合，且合并分区的时候进行聚合，聚合条件取order by字段内容。非聚合字段取最老一条的数据。
* SummingMergeTree()中写入的字段进行聚合，可以填写多个。假如不填写结果，则会直接对非维度列的数字列进行聚合，非数字列取正序第一个元素

ReplicationMergeTree

* + 副本类型的MergeTree，在建表的时候要根据部署情况来选定对应的分片，节点，然后回通过zookeeper进行数据同步
  + ReplicationMergerTree('/clickhouse/table/{分片}/{表名}',{节点名称})。

# ClickHouse索引

## 一级索引

1. 一级索引是针对所有分区内，order by 最左边的那个元素必须是主键，一级索引为ClickHouse order by 最左边的那个元素按照规则进行稀疏索引。
2. 关于order by()的建立规则，查询频率高，亦或者说高级的列放在前边（ClickHouse的最左原则）。同时，区分度大（基数大）的字段不适合成为索引，比如id之类的，没必要成为索引。

## 二级索引

1. 二级索引是在一级索引的基础上，对一级索引进行了再一次的稀疏索引，放大了一级索引的粒度
2. 二级索引的创建：ALTER TABLE [table\_name] ADD INDEX [index\_name] [feild\_name] TYPE minmax GRANULARITY [间隔区间，是个数字]

# ClickHouse的分区

分区的比例不易太大或太小，需要根据业务场景来进行确定。案例：1亿条数据，每个分区大概在10-30个为最佳。

建表语句：

CREATE TABLE table\_name

(

EventDate DateTime,

CounterID UInt32,

UserID UInt32

) ENGINE = ReplicatedMergeTree('/clickhouse/tables/{分片}/table\_name', '{副本}')

PARTITION BY toYYYYMM(EventDate)

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

# ClickHouseSQL特点

不是所有的引擎都适配查询sql，有部分引擎对sql的关键字有一定的限制

ClickHouse没有相关的删除或更新语句

删除：alter table [table\_name] delete from [table\_name] where [conditional]

更新：alter table [table\_name] update [table\_name] set [feild] = [value] where [conditional]

新特性：Explain

近似于mysql的explain，会根据层级展示语句段执行的先后过程，便于调优

例如 explain syntax [语句]，可以对应的sql转换执行结果

explain plan [语句]，解释执行计划

explain pipeline [语句]，解释详细的执行计划，同时会包括线程使用和核心数使用

# ClickHouse集群

## ClickHouse多主同步

ClickHouse高可用副本需要zookeeper配合实现的。节点1新增了一条数据，会将消息先同步日志写入zk，节点2监听到写入日志会去节点1主动拉取。

注意，主从同步必须使用副本（Replication）的引擎，比如说ReplicationSummingMergeTree

副本表引擎相关的表创建需要两个参数，一个是表路径，一个是副本名称（尽量单个副本使用统一名称），命名规则

ReplicationMergerTree('/clickhouse/table/{分片}/{表名}',{节点名称})

特点：

1. 依赖 zookeeper：在执行 insert 和 alert 查询的时候，ReplicatedMergeTree 需要借助 zookeeper 的分布式协调能力，以实现多个副本之间的数据同步，但是在查询的时候，并不需要使用 zookeeper。
2. 表级别的副本：副本是在表级别定义的，所以每张表的副本配置都可以按照它的实际需求进行个性化，包括副本数量，以及副本在集群内的分布式位置等。
3. 多主架构（Multi Master）：可以在任意一个副本上执行 insert 和 alter 查询，他们的效果是相同的，这些操作会借助 zookeeper 的协调能力被分发至每个副本以本地形式执行。
4. Block 数据块：在执行 insert 命令时，会依据 max\_insert\_block\_size 的大小（默认 1048576 行）将数据切分成若干个 Block数据块。所以数据块是数据写入的基本单元，并且具有写入的原子性和唯一性。
5. 原子性：在数据写入时，一个 Block 块内的数据要么全部写入成功，要么全部失败。
6. 唯一性：在写入一个 Block 数据块的时候，会按照当前的 Block 数据块的数据顺序、数据行和数据大小等指标计算 Hash 信息摘要记录。在此之后，如果某个待写入的 Block 数据块与先前已被写入的 Block 数据块拥有相同的 Hash 摘要，则该 Block 数据块会被忽略，这个设计可以预防由于异常原因引起的 Block 数据块重复写入的问题。

## ClickHouse集群搭建

1. 集群和主从同步的不同在于，多主同步，数据要保持一致。而集群是将一个表的数据切割成好几个分片，每个分片必须在不同的节点上，且每个分片的数据不一定相等。
2. 所有的分片都是基于表去进行分片的，副本也同理，ClickHouse本身是没有真正意义上的集群或者副本的，所有操作都是基于表进行切分的
3. 分片需要依赖一个特殊的表引擎Distributed，该表是不存储任何数据的。
4. ClickHouse集群表插入的过程为，先将数据提交给distribution库，然后通过distribution进行分片同步，其中有一个internal\_replication（内部同步，即副本之间同步）需要开启，减小distribution的同步压力
5. ClickHouse集群表查询的时候会优先选择错误率低的副本，注意是每个分片中错误率比较低的副本；当错误率一致时，会有随机，顺序，加权，host相近4种模式。
6. ClickHouse最好单独部署在一个集群，或者选择使用虚拟化手段进行部署。

集群搭建：集群和分片的定义（3分片，2副本）。

编辑/etc/clickhouse-server/config.xml文件

将这段内容配置在remote\_servers下边

<solax\_power> <!-- 集群名称 -->

<shard><!-- 分片名称 -->

<internal\_replication>true</internal\_replication>

<replica><!-- 副本名称 -->

<host></host>

<port></port>

</replica>

<replica><!-- 副本名称 -->

<host></host>

<port></port>

</replica>

</shard>

<shard>

<replica><!-- 副本名称 -->

<host></host>

<port></port>

</replica>

<replica><!-- 副本名称 -->

<host></host>

<port></port>

</replica>

</shard>

<shard>

<replica><!-- 副本名称 -->

<host></host>

<port></port>

</replica>

<replica><!-- 副本名称 -->

<host></host>

<port></port>

</replica>

</shard>

</solax\_power>

# ClickHouse优化

## **建表注意点**

1. 时间类型能不存String就不要存String（与Hive不太一样），时间最好存对应的时间类型
2. 因为String的转换还需要将字符转换为对应的Date类型
3. 空值存储尽量不要存储Nullable，因为NULL它本身会降低性能，是的资源消耗增加，建议用一些默认值替代。首先可谓NULL的数据会被单独用一个文件存储，其次，有NULL的列是无法被索引的
4. 建表时尽量避免修改index\_guranularity（8192）的基本参数，这个值是用来规定一级索引的区间大小的

## 语句操作点

1. 建表以后可以尝试使用Alter进行表结构的修改，切记，不要随意修改字段类型。假如不需要保存全量的数据，建议使用TTL将其过期
2. 写入和删除不能过于频繁，可以调整最大并发次数。官方建议是tps 2-3（每秒2-3次）。当写入过于频繁，怎会出现Too Many Parts
3. 谓词下推、聚合外推、聚合消除、是指能提前的完成的事会提前完成，无意义的事情会直接取消。例如：select \* from (select \* from trip) where trip\_id = 111112121212会将优化提前，也就是select \* from (select \* from trip where trip\_id = 111112121212) where trip\_id = 111112121212
4. 标量优化，可以对查询sql后边加上setting，例如select trip\_id = 11111 ? 'hello' : 'world' from trip setting optimize\_if\_chain\_to\_mutiif = 1;该语句的setting是指当前设置只针对该次查询生效，其他情况设置不变

## **查询优化点**

1. Prewhere 和 where。Prewhere只针对MergeTree有效，其他情况下还是只能使用where

Prewhere针对于MergeTree来说，先根据对应的条件过滤出列，在进行行的补全，减少了IO的次数。解释一下where的操作过程，先筛选出对应的行，在进行回表，在进行对应列的筛选

1. 字段裁剪

列裁剪：简而言之，拒绝select \*；因为clickhouse是基于列式存储，而且尽可能都是宽表，所以会展示很多无用的字段，同时也会消耗非常多的资源

分区裁剪：简而言之就是，查询的时候规定分区，也就是在where或者prewhere后优先添加partition by的字段，先指定分区

1. order by 和 limit

order by 最后配合limit一起用，避免全表扫描给干挂掉

1. 虚拟列尽量避免

因为每个虚拟列都是单独运算的，数据量大的情况下会使得每一次都去计算。需要进行一定量的取舍。

1. uniqCombined勉强可以替换distinct

uniqCombine可以实现distinct的功能，但是不太精确，但是效率很高。但是distinct比较精确，效率却不怎么样。

1. 物化视图

普通视图只会记录相关的逻辑操作，不记录数据。但是物化视图不仅会记录操作，甚至还有数据。

物化视图太多会引起惊群效应。因为对原表插入数据，物化视图会自主同步数据

视图的创建

create view [视图名称] as [SQL语句]

创建物化视图

create materialized view [IF NOT EXIST] [db.]table\_name [TO [db.]table\_name] [ENGINE = engine] [POPULATE] AS [SQL语句]

TO 是物化视图创建的临时表（默认名称.inner.table\_name），即对临时表重命名

POPULATE 把历史所有数据重新刷新一遍，不推荐使用

1. 注意事项
   1. 查询超市，熔断：比如查询时长超过一定时间后，就拒绝了这次查询，并且将其抛弃
   2. 关闭虚拟内存
   3. 配置join\_use\_null，联表的时候将null替换为默认值，从而提高效率。（null字段存储非常好空间）
   4. .批量写入时先排序。因为分区写入会快很多，不然会反复切换线程，从而到时CPU拉高，效率还低。
   5. 关注CPU，不然“就会爆炸”

# 数据一致性

首先，表引擎ReplacingMergeTree，拒绝手动合并，性能影响非常的大。

group by 进行合并去重，比如建立一个deleted字段，通过group by [字段] having deleted = 0进行区中。

final，通过版本号去规定数据的先后顺序，可以通过final去选择最新的一条。（20.8版本前不要使用，而且final需要对应的引擎）

重复无所谓

总结：用合并树时无法保证强一致性，只能保证最终一致性

上述三点解决方案：手动合并，设计方案+group by，新版本使用final关键字