# 背景

## 现状

### 当前逻辑

TDSQL复用MySQL 的统计信息收集方案和代价评估机制，并针对分布式数据库的特点做了相应改造，主要涵盖三个方面

* 字典存储方式改造：

数据字典存储替换成TDStore 分布式存储，local字典访问替换成远程RPC GetRangeStatistics 访问

* 基于索引的动态采样改造：

替换了索引过滤场景下动态采样(Index Dive)逻辑，引入了QuickCachedRangeInfo 层，动态采样不再通过RPC 访问TDStore。

* 基于TDStore 的统计信息估行与缓存改造：

TDStore 估行采用数据 row\_count = size / avg\_key\_size 的方式估算行数。SQLEngine 通过 RPC 从table\_stats 字典中得到表total count统计信息。total\_count 记录在缓存Rdb\_table\_handler.m\_mtcache\_count 中。

### 面临的问题

TDSQL 的统计信息能力和原生MySQL一致，目前仍然按照单机的方式工作。存储引擎替换成TDStore后，SQLEngine 的基础统计信息存在较多问题。这里把当下发现的问题先进行汇总。后面统一设计修复解决：

* SQLEngine 节点之前的统计信息同步：表级别，列级别，索引级别
* 统计信息和代价估算不准确
* 代价模型丰富：

补充数据shuffle代价估算模型和对应的API

适配TDStore重新评估代价模型的default cost value

* Index Dive 动态索引采样与TDStore 兼容问题
* 列统计信息：直方图缺少混合直方图，NDV 较大的场景下，没有统计Popular Value。能解决数据均匀场景下的预估，当出现大账号等数据倾斜场景，数据预估有有偏差。

**改造后目前存在几个突出问题：**

* **SQLEngine 统计信息缓存更新与同步不及时问题**

当前版本基础统计信息问题很多，例如analyze table 显示 count 估行为0，SQLEngine 之间cache 缓存信息不同步等等。

引入的统计信息的cache 机制目前存两个问题：

1）cache 的失效与更新问题

m\_mtcache\_count缓存默认1小时失效。手动analyze table只能更新数据字典，无法更新m\_mtcache\_count 缓存。引起用户手动更新统计信息失效问题。 调小失效时间又会增大SQLEngine与TDStore的RPC次数，冲击系统。

2）cache在多个SQLEngine间的同步问题

分布式场景下cache需要保持一致性，一个SQLEngine更新了统计信息并udpate本地cache后，其它对等节点的SQLEngine也应该更新统计信息和cache。

SQLEngien cache更新：增加reload语法，SQL engine间广播消息更新cache。

说明：当统计信息过期或者表中更新行数>=10%时会触发重新计算统计信息，计算完成之后并广播。因此，大多数情况下只有一个SQLEngine回去TDStore进行统计信息的计算，其余一般都是用mysql.table\_stats查询计算后的结果。

* **基于索引的代价评估不准**

基于索引的估行是不准的：

下图是tpch scale = 1时，基于主键的代价评估，全表20万数据，p\_partkey 是主键。可以看到range变化的时候（from 100000 to 200000）, 估行没有发生任何变化。猜测可能走了index dive。

文本

描述已自动生成

* **动态采样的准确性问题**

索引过滤动态采样场景下，QuickCachedRangeInfo cache的返回的行数固定（index range scan）。导致在多个谓词过滤条件下，优化器greedy\_search贪心算法做代价竞争时，index range scan永远是best\_access\_path，CBO无法正常工作（选择率优化）。

Index range scan举例：

index range scan计划展示的行数是错误的。下图看到rows总是显示1，实际是19 行。tt2表的id 列是主键列，详细如下

**SQL与Schema：**

explain format=tree select \* from test.tt2 where id < 200;

tt2 | CREATE TABLE `tt2` (

`id` int NOT NULL AUTO\_INCREMENT,

`a` int DEFAULT NULL,

`b` int DEFAULT NULL,

`c` int DEFAULT NULL,

PRIMARY KEY (`id`),

KEY `a` (`a`,`b`)

) ENGINE=ROCKSDB AUTO\_INCREMENT=20501 DEFAULT CHARSET=utf8mb3

图形用户界面, 文本

描述已自动生成

期望结果，像InnoDB 一样展示正确信息：

图形用户界面, 文本

描述已自动生成

**根因分析：**

当access path 是index scan时，mysql会走动态采样的路径。会调用下面API从qck\_rows\_info中拿行：

handler::multi\_range\_read\_info\_const

qck\_rows\_info中使用了cache，这个cache value是错误的，是1（因此代价评估行数恒定为1，这里的缓存更新不及时导致执行计划很多时候不正确），导致index scan的估行行数都是错的。

文本

描述已自动生成

**修复方案：**

方案1：在handler::multi\_range\_read\_info\_const 中检查 index 列是否收集了直方图，如果收集了直方图，不走动态采样逻辑，直接从histogram 拿行。 把直方图的value也可以记录在cache中。

方案2：修正更新 qck\_rows\_info 中的cache 信息。

**验证：**

验证收集直方图后，动态采样不读取cache 信息，从直方图中获取行号基础信息， 可以正确获取行数信息，反映的cost 也更加准确

修改前：

图形用户界面, 文本

描述已自动生成

修改后：

图形用户界面, 文本

描述已自动生成

选择率优化举例：

**现象：**

非索引列的filter估行不跟随谓词条件变化 例如计划显示下面两个谓词的过滤性一致，都是2502，而实际两个谓词的过滤性差别很大

where a > 10 and a < 5330

where a > 1 and a < 2;

文本

描述已自动生成

**期望结果：**

期望优化器估行按照数据均匀分布假设，根据过滤条件进行线性估算。

selectivity =（range\_max - range\_min) / (col\_max\_val - col\_min\_val） \* total\_rows

**复现步骤：**

100% 复现：当查询过滤条件是非索引列时，seletivity永远一致。

**根因分析：**

MySQL的列统计信息只有直方图维度的记录，缺少每一列的min/max/ndv 等统计信息，因此无法使用线性估算的方式通过filter条件动态计算选择率。 看了下代码，当前mysql的实现非常简单，统一使用默认过滤值，所以selectivity 永远不变。

**修复方案：**

给列统计信息增加一个系统表（字典），记录每个列的min(low\_value), max(high\_value)，distinct value (用LLC估算），这样代价估算时可以根绝过滤条件做线性估算了。

说明：当前的实现中，这里是基于max-min的方式去估算distinct value（NDV）的，因为获取distinct count用Query方式很慢。

下面是OB 的列统计信息schema 信息

+-----------+------------------+--------------+--------------+------------------------------+------------------------------+---------+-----------+-------------+----------------------------+-------------+--------------+------------+-------+-------------+-----------+-------+

| OWNER | TABLE\_NAME | COLUMN\_NAME | NUM\_DISTINCT | LOW\_VALUE | HIGH\_VALUE | DENSITY | NUM\_NULLS | NUM\_BUCKETS | LAST\_ANALYZED | SAMPLE\_SIZE | GLOBAL\_STATS | USER\_STATS | NOTES | AVG\_COL\_LEN | HISTOGRAM | SCOPE |

+-----------+------------------+--------------+--------------+------------------------------+------------------------------+---------+-----------+-------------+----------------------------+-------------+--------------+------------+-------+-------------+-----------+-------+

选择率的估算在 Item\_func\_xx::get\_filtering\_effect 中计算，API 中可以拿到 谓词的过滤条件和table->file 数据结构，这样就能够通过handler在open table map 中找到对应table 的统计信息cache 和过滤谓词的range。

Rdb\_table\_handler 成员内部需要增加一个新的map/array, 记录每个列信息的min/max/ndv value

文本

描述已自动生成

* **路径选择与代价评估CBO 存在问题**

基于代价的路径选择：

谓词包含多个过滤条件时，基表的访问路径不准确。下图是tpch scale= 1 场景下，针对两个滤条件的查询计划。

where p\_partkey > 100000 and p\_size = 50

图形用户界面

描述已自动生成

基表的访问路径选择了index scan（p\_partkey>100000 and p\_size=5），这个路径选择明显不优，需要代价评估选择代价最小代价路径。可以看到TSC（TableScan）的代价是25523.79小于index scan 27754.00，TDSQL选择了代价更高的计划。

* **TDStore 的估行不稳定问题**

row\_count跟随sstable的size变动。当sstable发生compaction数据压缩后，size会明显变小。引发在没有数据删除的场景下，优化器估行突然变小，造成估行不稳定，影响代价正确判断。

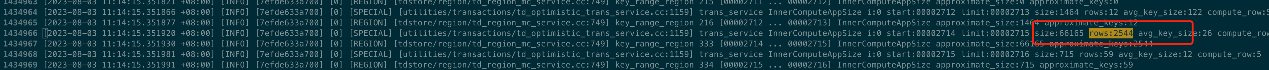
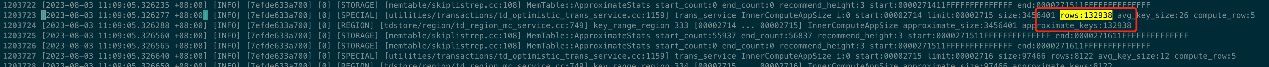
现象：

反复执行explain命令，计划展示rows行号发生跳变，突然从132938减小成rows=2544。

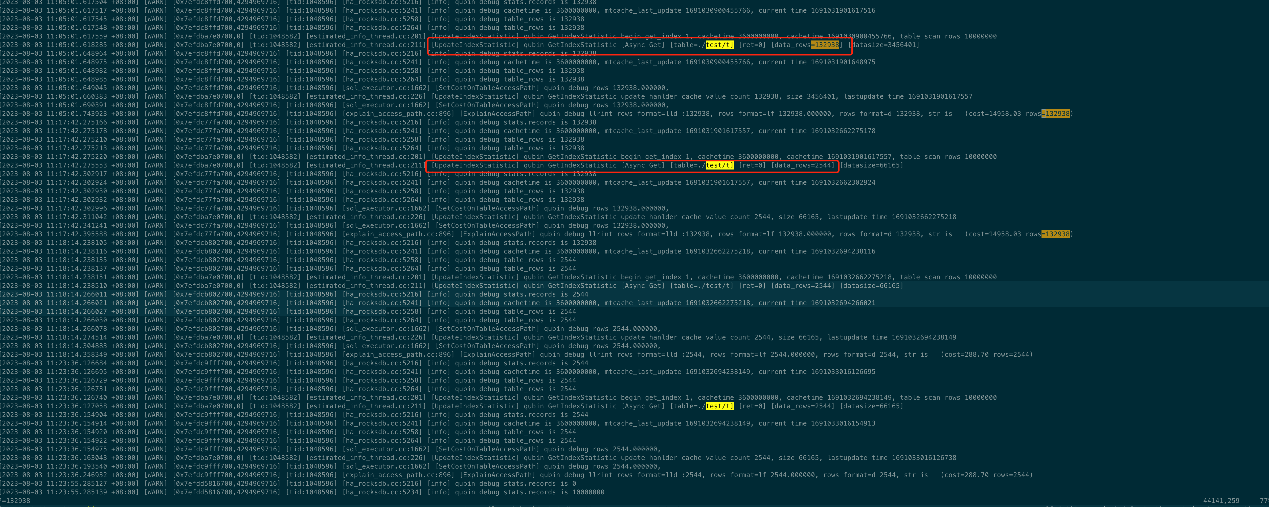
文本

描述已自动生成

增加Debug日志发现API调用TDStoreServiceImpl::GetRangeStatistics ->TDOptimisticTransService::InnerComputeAppSize的内部显示TDStore返回的估行发生了跳变，数据明显减少。日志截图如下：



SQLEngine上层的日志也证明了从存储返回的数据发生了跳变：



复现方法

这个问题是在强制SQLEngine每次explain时强制执行RpcRangeStatsCntl RPC时发现的，原始代码由于会cache数据，下发几率很小，触发RPC的几率很低。不容易遇到这个问题，但是重启几遍sqlengine 就能碰到。

文本

描述已自动生成

目前TDStore估行时依赖的size采样是随机值，value存在跳变现象。这个API需要替换。复现方法已找到，写入部分数据，或者手动触发一个compaction就能复现。

图形用户界面, 网站

描述已自动生成

* **直方图**

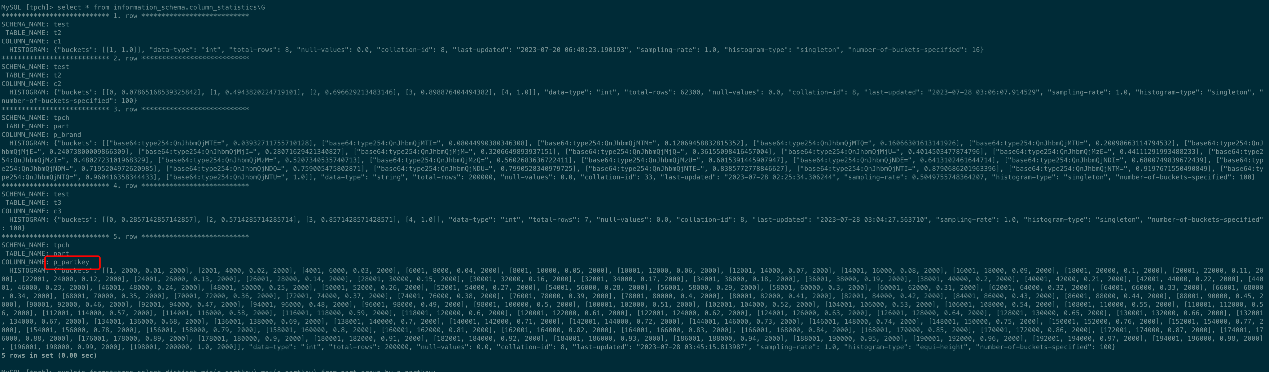
基于索引列查询条件没有使用直方图，查询不准。

1、索引列和非索引列统一采用直方图进行代价估行，解决索引估行不准的问题。

这种方案的一个Bad case 是，直方图的更新不是实时的，目前是手动更新的。需要配合直方图的自动更新，避免直方图更新不及时出现直方图预估不准的问题，否则索引列即使使用了直方图，在更新频繁的场景下依然会出现估行不准引发的计划问题。

关于直方图的更新。

2、Index Dive enhancement：单机MySQl 采用Index Dive 动态采样的方式，实时估行。这种场景避免了对统计信息实时更新的依赖。竞品OceanBase 也提供了类似的方式。TDSQL基于索引列的Index Dive 是不准的，这里可以考虑做个Enhancement：如果是单RG或者查询可以裁剪到一个RG时， 实现基于TDStore 的动态采样估行。

下图主键列已经有了直方图信息，验证了直方图的信息是准确的。 但是基于主键的谓词过滤的计划还是不准确的：  


分布式场景下代价估行不准： 计划不稳定，有时准确使用了直方图，有时没有使用直方图。

实际数据分布：

图形用户界面, 网站

描述已自动生成

当前估行没有使用到直方图： 过滤条件变化，估行不变

文本

描述已自动生成

错误计划：

图形用户界面, 网站

描述已自动生成

正确计划：

图形用户界面, 文本

描述已自动生成

cache 变量信息

文本

中度可信度描述已自动生成

缺少混合直方图： MySQL 的直方图是等高直方图，没有Popular Value List。 可以参照PG 实现引入混合直方图。 TXSQL 目前已经完成了这个工作，这块可以考虑复用作为二期使用。

* **分区表统计信息**

暂时没有验证，待补充

* **代价模型内容较少**

目前代价模型和MySQL 提供的一致，存在指标较少的问题。而且default value 是否匹配TDSQL 匹配暂不可知。

例如MySQL是单机数据库缺少shuffle 相关的代价评估，这个要在代价模型提供基础指标，同时在计算代价的过程中添加进去。 具体指标可以参照OB 的cost\_model 指标。第一期可以只添加network 相关代价。

电脑屏幕的照片上有文字

中度可信度描述已自动生成

文本

描述已自动生成

* **统计信息自动更新**

1、MySQL 基础统计信息自动更新：不及时

innodb\_stats\_auto\_recalc 开启状态下， 表级别更新不及时。 InnoDB 提供统计信息在线更新功能，即表的更新超过10% 自动更新统计信息。 TDSQL 目前无法自动更新。

图形用户界面, 文本

中度可信度描述已自动生成

2、直方图统计信息缺少在线更新： MySQL 不提供直方图的在线更新，TDSQL 未来大概率强依赖直方图，需要提供类似基础统计信息自动收集的功能。

3、直方图cache 信息不同步: 和基础统计信息一样，cache 没有同步。

* **统计信息管理**

除了利用show status xxx 这样的基本的统计信息展示，TDSQL 还需要提供统竞对同计信息的管理功能，这些功能优先级不高。可以作为2期功能增加。

统计信息锁定

统计信息多版本

统计信息自动更新任务状态

按照时间窗口更新自动更新统计信息

除此以外还存在直方图选择率计算不够准确，非动态采样路径下选择率估算没有缩放算法，总是使用默认选择率等问题。

## 改进方案

针对上面的问题，需要重新改造TDSQL的优化器统计信息并重新优化选择率估行方式。

* 统计信息：实时更新统计信息已准确反映统计信息的时效性，并丰富统计信息的种类和管理手段，方便优化器更好的估算代价。
* 优化器选择率计算：非动态采样路径下，在用户没有收集直方图时，不使用默认固定选择率，使用选择率线性缩放方法计算选择率。

由于需要丰富的模块较多，目前准备分阶段实施。本文重点讨论一期设计内容，二期内容放在未来规划章节中

* 一期

优化器选择率改造：估行从基于默认选择率的方式改造成线性估算的评估方式

SQLEngine 统计信息自动更新

SQLEngine 之间缓存一致性

TDStore 估行API重构

Index Range Scan 动态采样估行逻辑优化

基于直方图的统计信息估行

* 二期：

基础统计信息丰富：引入新的列统计信息basic\_column\_statistics，并持久化到字典中。记录每列 min/max/ndv 信息

统计信息管理： 增加统计信息锁定，导入导出，删除功能

在线收集统计信息： AP 导数场景实时收集统计信息不在依赖统计信息自动收集

并行收集统计信息： 增加并行收集能力，提高直方图收集速度

混合直方图： 引入识别高频值的直方图，解决数据分布不均匀场景下的直方图selectivity 准确性问题

动态采样和统计信息估算相结合的估行方案: 基于单region查询的本地主键查询，从存储动态读行。

# 总体设计

为解决统计信息缓存时效性和一致性问题，需要引入统计信息的失效和自动更新机制，并引入同步通知，解决SQLEngine 的统计信息相互独立且不能及时更新的问题。

基本思想是引入Statistic Collector 组件，定期检查rdb\_open\_tables 的用户表全局DML 更新率(DML refresh ratio)是否超过全量数据的阈值（默认10%）。 如果超过阈值触发analyze 命令自动收集统计信息，并更新本地Cache。同时Statistic Collector发送更新命令（analyze table xxx reload)到其它节点让其它节点到DD 字典中重新读取统计信息并更新本地缓存，实现一致性。 为保证DML Refresh Ratio 计数正确性，引入全局dml modify counter 另外一个组件，记录每张表的全局dml 更改记录，并累加写入内部表/字典（minitor\_modified名称待定)，得到全局dml\_modify\_counter。两个组件是两个不同的后台任务，按照不同的周期轮询，整体方案如下图：

图示

描述已自动生成

上面设计主要解决一下几个问题

* 统计信息的自动收集：

统计信息的收集频率和DML 操作频率应当同频，我们给每个表引入一个global\_dml\_refresh\_ratio 概念。 当是全局DML 更新率(DML refresh ratio)到固定阈值时，触发收集统计信息。DML refresh ratio 的计算需要全局级别的DML modify counter 计数，当前SQLEngine 的handler 接口记录了本地的DML modify counter（增/删/改 counter 记录），缺少全局DML Modify Counter计数。 后台任务DML Modify Counter 负责这个功能，它定期把当前节点的DML metric 累加到内部表minitor\_modified中，并把累加数据sync 到本地，实现全局DML metric 在固定的周期内同步，默认一分钟同步一次，参数可调。后台任务Statistics Collector 负责检查DML refresh ratio是否超过阈值和统计信息自动判断。 收集统计信息收集通过执行 analyze table xxx 命令完成。

* 统计信息缓存的失效与更新:

Statistics Collector 也负责统计信息缓存的失效与更新，完成统计信息收集后自动清空缓存，并更新最新值。 清空的方式沿用现在的实现方式，通过HA\_STATUS\_FORCE\_UPDATE\_CACHE\_SYNC 变量控制。另外本地和内部表的dml modify counter 计数也要一并清空。

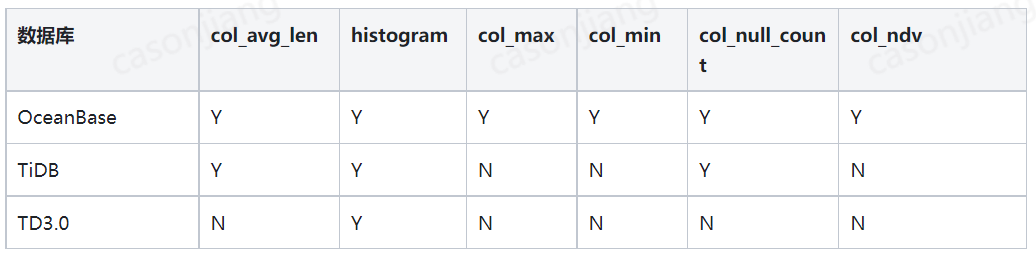
* 统计信息缓存的同步：

SQLEngine 间的缓存同步是另外一个需要考虑的问题：复用现在reload 命令和接口实现同步。 Statistics Collector 在更新本地缓存的同时，需要广播reload 命令到其它节点： analyze table table\_name reload。 reload 操作只负责从(table\_stats/index\_stats/column\_statistics/basic\_column\_statistics) 字典里表里读数据，实现自己节点数据同步。

# 统计信息收集与更新

## 统计信息扩展

TDSQL 的统计信息有表统计信息，列统计信息，索引统计信息三个类别。 其中列统计信息只有直方图信息，缺少列的基础统计信息，优化器无法根据列统计信息做线性代价估算。另外列的平均长度这些基础信息也需要记录，方便优化器根据行长做shuffle 的代价估算，因此需要补充该信息。下表是竞对数据库的列统计信息指标：



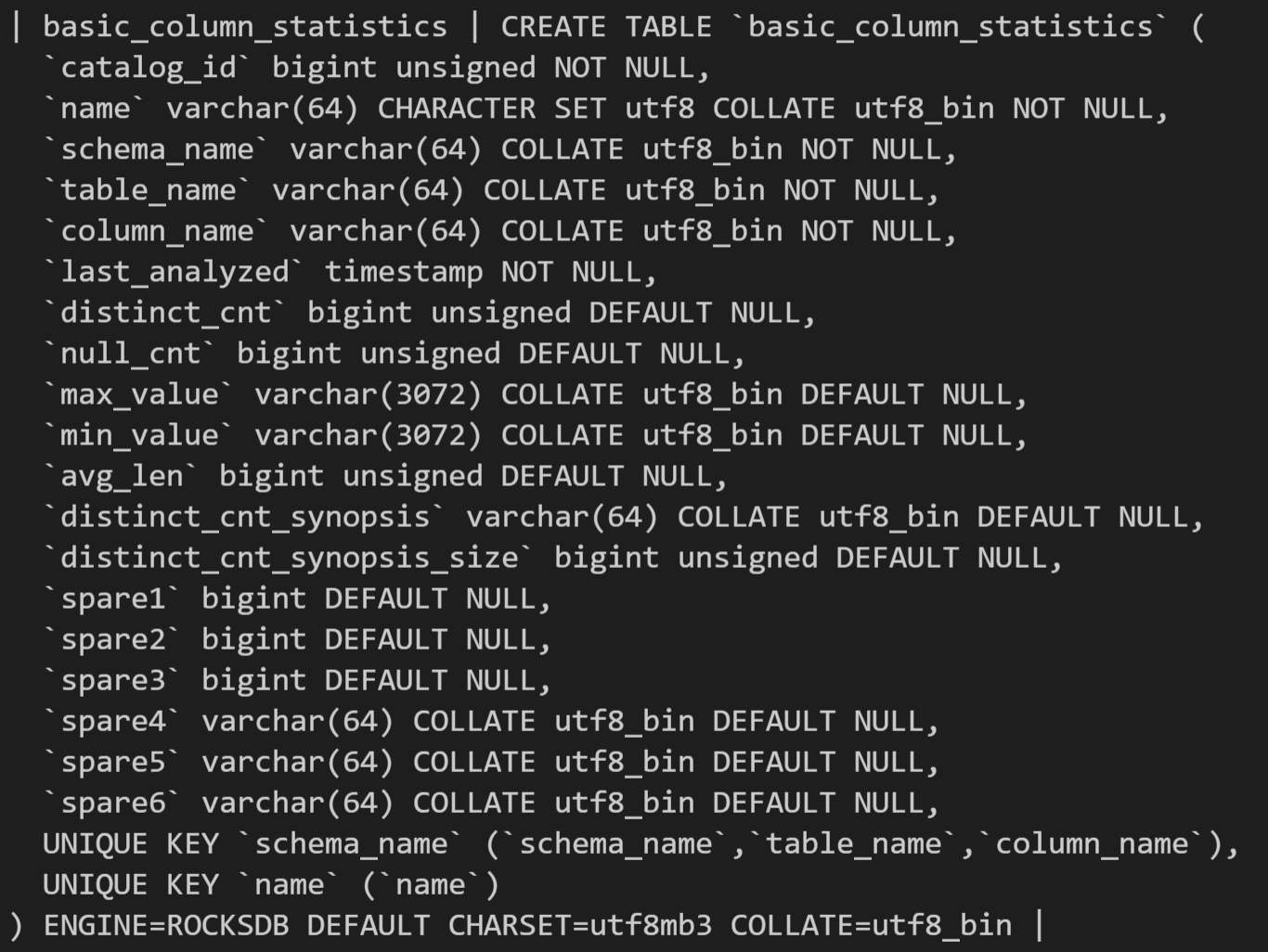
考虑到与MySQL 兼容因素，现有列统计信息字典 column\_statistics 格式保持不变，仍然记录直方图信息。增加新的列统计信息字典和视图 BASIC\_COLUMN\_STATISTICS。

### 基础列统计信息：BASIC\_COLUMN\_STATISTICS

记录每一列的最大，最小值(max/min\_value) 和其二进制编码后的信息(b\_max/min\_value)，null 的个数（null\_count） 和NDV等基础信息。

#### 数据定义

* 字典表 basic\_column\_statistics的schema 信息



| Field | Type | Null | Key | Default | Extra |

| id | bigint(20) | NO | PRI | NULL | |

| catalog\_id | bigint(20) | NO | PRI | NULL |

| name | varchar(4096) | NO | | NULL |

| schema\_name | varchar(4096) | NO | | NULL |

| table\_name | varchar(4096) | NO | | NULL |

| column\_name | varchar(4096) | NO | | NULL |

| last\_analyzed | timestamp(6) | NO | | NULL | |

| distinct\_cnt | bigint(20) | NO | | NULL | |

| null\_cnt | bigint(20) | NO | | NULL | |

| max\_value | varchar(4096) | NO | | NULL | |

| b\_max\_value | varchar(4096) | NO | | NULL | |

| min\_value | varchar(4096) | NO | | NULL | |

| b\_min\_value | varchar(4096) | NO | | NULL | |

| avg\_len | double | NO | | NULL | |

| distinct\_cnt\_synopsis | varchar(4096) | NO | | NULL | |

| distinct\_cnt\_synopsis\_size | bigint(20) | NO | | NULL | |

| spare1 | bigint(20) | YES | | NULL | |

| spare2 | bigint(20) | YES | | NULL | |

| spare3 | bigint(20) | YES | | NULL | |

| spare4 | varchar(4096) | YES | | NULL | |

| spare5 | varchar(4096) | YES | | NULL | |

| spare6 | varchar(4096) | YES | | NULL | |

* BasicColumnStat 数据结构

增加新的数据结构，在SQLEngine 侧记录列基础统计信息，详细如下

struct BasicColumnStat {

uint64\_t column\_id\_;

int64\_t num\_null\_;

int64\_t num\_not\_null\_;

int64\_t num\_distinct\_;

int64\_t avg\_length\_;

int64\_t min\_value;

int64\_t max\_value\_;

char \* b\_min\_value\_; // binary min value for varchar

char \* b\_max\_value\_; // binary max value for varchar

/\*\* last analyzed time \*/

int64\_t last\_analyzed\_;

};

* 每个表 Rdb\_table\_handler 中增加vector 记录所有列信息

struct Rdb\_table\_handler {

uint64\_t m\_mtcache\_count;

uint64\_t m\_mtcache\_size;

uint64\_t m\_mtcache\_last\_update;

+ std::vector<BasicColumnStat> column\_stats\_;

#### 数据收集

**Option 1（首选）：**

* TDStore数据生成

基础列统计信息在TDStore 生成，SQLEngine 只是实时通过RPC 读取更新。需要TDStore 同学分析是否可以提供一个快速计算表的每列统计信息的方法。

例如TDStore 在compaction 时给记录SST 级别的统计信息，并记录每个列的min/max/nulls/ndv 等，收到RPC时，做简单min/max/nulls 归并汇总。 NDV 本期暂不实现

* SQLEngine 数据收集

通过新的RPC RpcBasicStatsCntl 收集列统计信息BasicColumnStat 和表统计信息 table\_rows (表行数） table\_bytes（表大小信息），新的数据结构如下

struct BasicColumnStats {

uint64\_t data\_rows = 0;

uint64\_t data\_size = 0;

std::vector<BasicColumnStat> column\_stats\_;

};

增加新API PackInfo::GetBasicStatitstic ，SQLEngine handler::info 函数中调用，处理逻辑和同现有PackInfo 读TDStore 数据方式一致。

**Option 2（当前采用的方案）**

启动定期任务，执行聚合sql :

Select count(col1),…,count(colN),min(col1),…,

min(colN),max(col1),…,max(colN) … from table ;

把结果写到内部表里。这种方案的问题是聚合计算是CPU 密集操作，在大数据量多表场景下，很难做到列统计信息及时更新。

最优方案为Option1，但是鉴于TDStore的支持工作量比较大，当前手动更新统计信息，暂时采用Option2。

说明：在实际测试发现，这里需要进行修改：

Select min(CONVERT(col USING utf8mb4)),

max(CONVERT(col USING utf8mb4))

from …

主要是在于count非常耗时，且min-max需要进行字符类型的转换。

* **最终结论：**

鉴于 Option1 对TDStore 的改造量较大，经过讨论大家达成一致，基础列信息的收集工作放到二期实现，一期只创建字典表basic\_column\_statistics ，暂不收集这些信息。由于没有基于基础列信息，基于它的优化器选择率工作也一并放入二期，一期可以根据方案2产生基础统计信息，但是考虑到聚合对CPU 的影响，不根据10% 的修改率触发更新。basic\_column\_statistics 的更新时间和直方图的更新时间合并，在maintenance 窗口期内一天触发一次。

#### 字典更新

在analyze table 流程中mysql\_admin\_table 增加update\_basic\_column\_stats()/setup\_basic\_column\_stats\_record() API。流程与现在的update\_table\_stats()一致，把从TDStore 收集的数据写入到字典中。

## 统计信息收集与更新

### 手动收集

通过analyze 命令手动收集统计信息。

#### 分类

命令分为两类：

* 基础统计信息

收集表的行数，平均长度和每一列的min,max,ndv, null 值， 命令如下。 reload 命令只从字典读数据，不收集统计信息，最新版本已经支持该功能。

ANALYZE [NO\_WRITE\_TO\_BINLOG | LOCAL]

TABLE tbl\_name [, tbl\_name] ... 【reload】

* 直方图统计信息

收集指定列正方图信息，直方图的JSON描述形式如下

{

"histogram-type": "xxx",

"last-updated": "2023-08-07 12:49:18.528565",

"sampling-rate": 1.0,

"number-of-buckets-specified": 100}

"buckets":

[

{ "low": 3, "high": 400, "freq": 0.02, "distinct": 350 },

{ "low": 400, "high": 800, "freq": 0.98, "distinct": 375 },

...

]

命令，同样 reload 命令只从字典读数据，不收集统计信息。目前直方图不支持reload 命令，需要增加reload 命令，逻辑与表级别的reload 一致，不在详细描述。

ANALYZE [NO\_WRITE\_TO\_BINLOG | LOCAL]

TABLE tbl\_name

UPDATE HISTOGRAM ON col\_name [, col\_name] ...

[WITH N BUCKETS] 【reload】

#### 缓存失效更新与一致性

收集统计信息后需要强制更新本地统计信息缓存并广播其它节点同步更新统计信息。

* 本地缓存更新

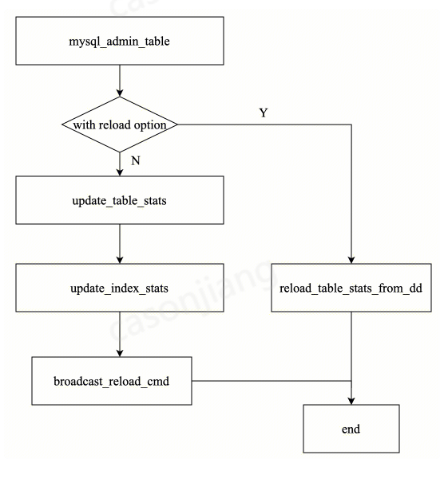
复用当下策略，采用同步更新，通过传递HA\_STATUS\_FORCE\_UPDATE\_CACHE\_SYNC 标记到handler 层通知去TDStore拿最新统计信息。即SQLEngine收到analyze 命令时，分别在基础统计信息路径（mysql\_admin\_table）和直方图路径下（handle\_histogram\_command）标记SYNC 同步更新统计信息选项。handler 层在读取统计信息时根据标记判断是否更新统计信息。如果更新发送RpcBasicStatsCntl RPC 获取最新的table\_rows, table\_size 和BasicColumnStat信息，实时更新缓存。

Note：mysql\_admin\_table 目前已经具备该功能，以上逻辑扩充到直方图统计信路径下（handle\_histogram\_command）

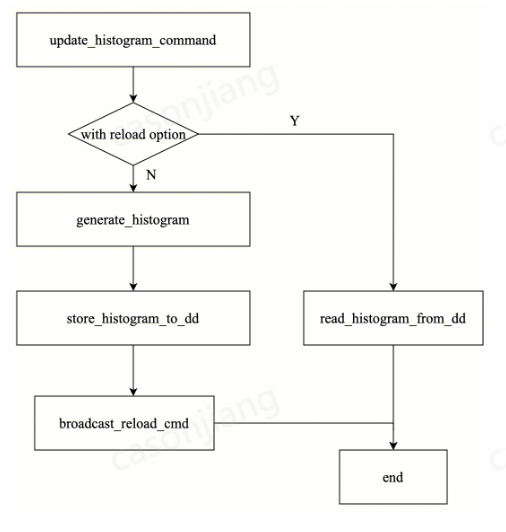
* 远程缓存更新：

在更新完本地节点的缓存后，SQLEngine 需要广播analyze xxx reload命令到其它节点。 reload 命令不收集统计信息只是从TDStore 中加载统计信息到本地缓存，更新缓存的方式和本地缓存更新一致。当前版本已经支持了table\_stats 的reload 更新（API reload\_table\_stats)。 新版本会增加直方图路径（handle\_histogram\_command）下的reload 广播逻辑，构造 analyze table xxx update histogram columnN reload 命令广播到其它节点。其它节点按照同样的方式reload

表级别统计信息更新

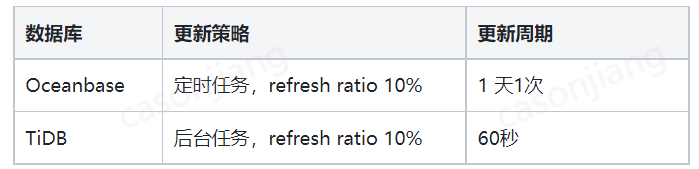


直方图的缓存更新流程如下



### 自动收集更新

自动更新统计信息同样使用analyze 命令更新统计信息，TDSQL收到analyze 命令后的处理逻辑完全一致，只是触发收集的方式不同。目前竞品数据库大都支持自动更新，更新策略见下表：



自动收集通过新引入的后台任务Statistics Collector 完成，该任务定期遍历rdb\_open\_tables 的所有table ，并检查更新率 refresh ratio 是否超过阈值， 如果更新率过高，则发送analyze 命令给SQLEngine完成自动收集。 是否自动收集通过下面两个变量控制：

* stats\_auto\_recalc ：

该变量默认开启，关闭则不收集。

* auto\_reload\_uplimit:

变量更新的判断阈值，默认10%。超过阈值时发送analyze analyze table table\_xxx 命令给自己，否则不发送。

由于收集直方图信息是CPU密集操作，自动收集只收集基本统计信息，不收集直方图信息。直方图的自动收集采用较为简单的单独策略。

#### 基础统计信息自动更新

* 更新率（Refresh Ratio）计算

Refresh Ratio记录一张表DML metric（增/删/改）占比，是表级别的全局信息，用来判定是否触发自动更新。计算公式如下

refresh ratio = (global\_dml\_modify\_counter ) / total\_count

* 全局DML计数：global\_dml\_modify\_counter

更新率计算依赖另外一个全局信息 global\_dml\_modify\_counter，它记录给定时间内的全局增删改计数，为考虑效率，采用异步更新的方式。详细实现如下:

1、global\_dml\_modify\_counter 记录在Rdb\_table\_handler中，每个表维护自己的计数，记录本地的增删改信息 dml\_modify\_counter = delete + insert + update

2、持久化dml\_modify\_counte 到数据字典table\_stats，增加一个新的字段 dml\_mofify\_counter，dml\_modify\_counter 持久化在这个字段里

3、创建DML Modify Counter 后台任务，该任务定期遍历rdb\_open\_tables，读取表对应在table\_stats字典中的dml\_modify\_counter，并把本地的dml\_modify\_counter 累加 stats.dml\_refresh\_counter 中（这里不需要累加，在UPDATE操作的时候直接采用update dml\_modify\_counter=dml\_modify\_counter+global\_dml\_modify\_counter即可）。为避免多节点并发写，累加时读加行锁 (tdsql::SwitchGetDDForUpdateClosure 用guard 加锁)。累加后的结果更新到本地变量dml\_modify\_counter 中，得到全局dml counter 计数。

* 自动更新

创建Statistics Collector 后台任务，定期遍历rdb\_open\_tables 的所有成员，检查 refresh ratio = dml\_modify\_counter / m\_mtcache\_count 是否超过阈值auto\_reload\_uplimit。 超过阈值：触发手动analyze table xxx 命令，同时清空本地dml\_modify\_counter 和字典中的dml\_modify\_counter 字段，保证计数从零开始。不超过阈值： 继续遍历。

#### 直方图自动更新

直方图也需要更新，这里采用类似定时任务的方式，以相对低频的方式更新直方图。 即只允许在时间窗口内更新数据（e.g 每天凌晨1:00 AM - 5:00 AM ）触发直方图更新命令，且一天只更新一次，减少对CPU的冲击。更新逻辑如下：

1、创建Histogram Statistics Collector 后台任务

2、检查时间窗口，时间窗口内继续处理，否则进入sleep

3、从字典column\_statistics 读数据，检查是否有超期未更新表。last\_update 属性24小时以上的为超期未更新数据，创建超期未更新数据表列表，优先高优先级更新这些表的统计信息

4、再次读column\_statistics 字典，上行锁，避免SQLEngine 多节点同时更新。先遍历高优先级更新列表，构造命令 analyze table table\_name update histogram on column，发送给自己更新直方图。在遍历column\_statistics 字典返回数据，判断last\_update 是否在4小时以内，小于4小时的数据执行 analyze table table\_name update histogram on column 命令更新直方图信息。

#### 统计信息采样加速

TiDB有如下的加速方案：

● 快速采样

由于TiDB需要到TiKV上远程读取regon数据，统计信息收集耗时要大于MySQL。为实现加速，TiDB引入快速分析配置项，tidb\_enable\_fast\_analyze。 开启后会随机采样10000行数据来构建统计信息。由于只采样10000行数据，在数据分布不均匀的场景下估行不准，因此默认不开启。

● 并行加速

另外TiDB也支持并发收集统计信息，实现加速。通过参数tidb\_build\_stats\_concurrency控制并发度。

##### 快速采样

###### 背景

当前直方图的自动更新是在后台线程tdsql\_histogram\_statistic\_update进行的，这里基本实现逻辑是进行时间窗口检查（闲时更新），定期执行全量列的更新操作。当前的实现逻辑是全量，即获取column\_statistics中的所有信息，逐一遍历更新：

// Start the scan.

std::unique\_ptr<Raw\_record\_set> rs;

if (table->open\_record\_set(nullptr, rs)) {

return true;

}

综上，当前的实现方案存在如下问题：

1. 全量收集统计信息，虽然设置了时间窗口，但是可能仍然会非常耗时，导致更新失败
2. 没有对统计信息收集过程进行监控，对于耗时卡死的情况，没有处理机制

###### 方案

针对全量更新慢的情况，这里可以采用采样加速的方式，增加一个配置项，每次只更新一部分列的统计信息。

* **基本思路：**

这里需要获取当前的Column\_statistics的所有记录数，然后按照配置的采样率和策略，计算出最终需要获取的record记录数，实现更新。

下一次循环的时候，从当前的这个位置开始继续获取对应比例的更新表信息，逐步实现循环更新。

**采样策略选择以及ratio计算逻辑：**

1. 判断是否为采样加速，如果是需要获取采样的策略以及配置
2. 如果是按照行（RowSample）
3. 采样行数（sample\_value）<总行数（total\_row\_count）

计算采样比例estimate\_percent：

estimate\_percent = ( sample\_value \* 100.0 ) / total\_row\_count

1. 采样行数>总行数，不允许采样
2. 如果是按照百分比（PercentSample）

计算采样比例estimate\_percent：

estimate\_percent = sample\_value \* 1.0

说明：计算出具体的比例之后，根据总行数，可以计算出一个行数，加上一个随机值，就可以将每次执行都随机到某一个区间范围内。

**说明：**

这里是针对后台线程的自动更新，如果是客户端手动更新，则可以使用相关的HINT实现（TODO：参考Oceanbase方案），比如增加DYNAMIC\_SAMPLING(10%)。

在这里是否还需要考虑动态采样过程中的超时问题，如果采样时间过长，则会关闭动态采样。

如果按照行的方式采样，是否需要设置最大支持采样行？

* **改动**

1、新增配置：

double tdsql\_optimizer\_dynamic\_sampling\_ratio; //配置采样的比例

string tdsql\_optimizer\_dynamic\_sampling\_type; //配置采样的策略

参考OceanBase：

enum SampleType

{

RowSample, // 按照行

PercentSample, // 按照比例

InvalidSample

};

说明：这里的采样策略，可以是按照多少行的线性方式（可能会导致当前有一些列完全没有统计信息），也可以按照随机的方式去做（可以避免最坏的情况，但是怎么保证最后随机能够覆盖所有的列）。

2、修改接口：

update\_column\_statistics\_entries // 增加参数，存储当前更新的进度情况

3、新增接口：

check\_update\_ratio() //判断当前的更新率

##### 并行加速

#### 更新周期

##### 更新参数 lease time

统计信息自动收集引入了多个后台任务，而且彼此的更新周期不一致，为避免冗余的控制变量，后台任务的更新周期统一用lease time 控制。lease time 默认3秒钟，每个任务按照固定的倍数与lease time 相乘控制运行周期。详细如下：

更新率计算间隔：20 \* lease\_time

基础统计信息更新间隔：5 \* lease\_time

直方图统计信息更新间隔：200 \* lease\_time

直方图开始更新时间：histogram\_auto\_analyze\_start\_time（默认：01:00 +0000）

直方图结束更新时间：histogram\_auto\_analyze\_end\_time （默认：05:00 +0000）

## 优化器估行与选择率估算

### TDSTORE 统计信息估行

为避免估行跳变，当前估行API TDOptimisticTransService::InnerComputeAppSize也需要改造。

### 优化器估行

#### 动态采样逻辑优化:

由于统计信息已经可以实时反映数据分布，新设计通过统计信息进行选择率估算，不在依赖动态采样。统计信息选取的优先级按照：直方图 > 基础统计信息 > 默认选择率的方式选择。

默认使用统计信息估算，估算方法如下：

1、如果定义了直方图，则通过range start key，end key调用estimate\_selectivity\_by\_histogram从直方图中计算选择率

2、如果没有定义直方图，但列基础统计信息存在，则基于数据均匀分布假设，采用线性估算的方式 计算选择率，

3、如果上面统计信息都没有，通过row\_count \* selectivity 估算函数

除基础统计信息估行外，其余两个估行方式复用MySQL逻辑，不做任何修改。基础统计信息估行依据选择率估算公式实现。

择率估算公式：

selectivity = (range\_max - range\_min) / (col\_max-col\_min)

#### 基础选择率优化

非动态采样路径选择率也需要优化。当前非动态采样路径下，优化器通过 condition->get\_filtering\_effect API，从谓词表达式中计算filter过滤条件的选择率，但选择率是固定的。例如对于等值过滤选择率永远是0.1，这对优化器的代价估算是很不友好的。固定选择率定义如下。

#define COND\_FILTER\_ALLPASS 1.0f

/// Filtering effect for equalities: col1 = col2

#define COND\_FILTER\_EQUALITY 0.1f

/// Filtering effect for inequalities: col1 > col2

#define COND\_FILTER\_INEQUALITY 0.3333f

/// Filtering effect for between: col1 BETWEEN a AND b

#define COND\_FILTER\_BETWEEN 0.1111f

需要根据过滤谓词条件动态计算选择率，而不像现在根据不同的谓词选择固定选择率。现在不同谓词与选择率的对应关系如下

Item\_func\_eq::get\_filtering\_effect <-> COND\_FILTER\_EQUALITY

Item\_func\_between::get\_filtering\_effect <-> COND\_FILTER\_BETWEEN

Item\_func\_gt::get\_filtering\_effect <-> COND\_FILTER\_INEQUALITY

…..

新设计会改造所有item\_func\_xx 的get\_filtering\_effect 实现，加入选择率估算公式，优先跟进选择率公式计算，如果统计信息没有基础统计信息时，使用原来的固定选择率方式兜底。

选择率公式复用动态采样实现的估算公式，依然通过(range\_max - range\_min) / (col\_max-col\_min) 进行线性估算。

## 系统升级与兼容性

### 字典表升级

由于需要给table\_stats增加新的列属性，并引入新的字典表 basic\_column\_stats，因此需要走通用的字典升级流程。 统计信息也同样遵循该升级流程，具体不在这里列举。

### RPC 兼容性

目前表统计信息是通过RPC RpcRangeStatsCntl 从TDStore 获得的。在新引入的RPC GetBasicStatitstic 后，需要保证升级兼容。 当前策略是在rpc\_version 升级到INITIAL\_CLUSTER\_RPC\_VERSION\_V17\_0\_0 前，仍然使用RpcRangeStatsCntl 同TDStore 获取统计信息。 当升级完成后，rpc\_version 升级到INITIAL\_CLUSTER\_RPC\_VERSION\_V17\_2\_0(具体版本可能修改）后，使用新的RPC GetBasicStatitstic 同TDStore 获取数据。

# 统计信息管理

当前的实现中，只做了基础的手动和自动更新，对于统计信息的管理还未进行。具体需要做如下的一些技术规划：

## 删除功能

当前对于直方图是有相关的DROP HISTOGRAM的接口的，但是列基础统计信息没有删除接口，只能在DROP TABLE的时候删除。

## 设置功能

设置功能，即人为的修改表、列的统计信息。在OceanBase中，采用的是存储过程的方式，TDSQL3.0因为只有的编译DEBUG版本下才可以设置系统表，这个在release版本不可行，因此还是需要提供一些HINT或者其他的SQL语法去实现统计信息的设置功能。

## 导入导出功能

在进行数据库迁移时，可以把大表的统计信息导出到指定的用户表一起做数据迁移，而当数据迁移成功后就可以直接将之前保存统计信息的用户表中数据导入到新库对应的统计信息内部表中。因此，这里是需要支持统计信息的导入和导出操作的。

统计信息导出功能主要用于将表的统计信息导出到一张指定的用户表中保存下来，而导入功能主要作用是将指定用户表中的统计信息导入到当前统计信息内部表中，省去重新收集的过程。

## 锁定解锁功能

统计信息的锁定功能主要用于锁死统计信息，防止统计信息更新，以保证统计信息的稳定性。

例如，如果想指定某一张表不收集统计信息，就可以提前上锁，这样计划生成的时候就会采取其他方式获取统计信息（动态采样等）。又如，对于一张数据稳定的表，我们不需要经常去收集它的统计信息，因此就可以提前上锁。

## 历史管理功能

统计信息历史管理功能主要是指统计信息状态发生改变时（统计信息被重新收集、设置、删除、导入、锁定等），会将统计信息发生改变之前的统计信息状态保存到统计信息历史记录表中，统计信息历史记录表可以查询某张表的一个历史统计信息改变情况，同时也能够恢复指定历史版本的统计信息。

## 收集策略配置项管理功能

在统计信息收集过程中，对于一些收集策略需要进行配置。比如收集并发度、采用的采样比例、是否增量收集等。

# 统计信息收集监控诊断

数据库优化器的统计信息收集过程比较耗时，特别是自动统计信息的收集操作。因为收集会涉及当前数据库中所有表，很有可能存在由于某种原因而出现收集失败的表。如果收集失败的表不能及时被诊断和反馈，会导致执行计划生成时无统计信息可用，从而生成烂计划，造成数据库系统运行异常。

# Roadmap

实现上述功能后，TDSQL的统计信息具备了自动更新和同步能力，并且可以根据直方图或者基础统计信息进行选择率计算，功能的丰富程度已经超过MySQL。为了和其它竞对数据库能力对齐，还有部分功能有待丰富。比如

统计信息管理：增加统计信息锁定，导入导出，及历史版本

并行收集统计信息：增加并行收集能力，提高直方图收集速度

在线收集统计信息：AP 导数场景实时收集统计信息不在依赖统计信息自动收集

混合直方图：引入识别高频值的直方图，解决数据分布不均匀场景下的直方图selectivity 准确性问题

单region下动态采样优化：当查询数据只分布在一个region上时，使用动态采样的方式从TDStore 估行，获得更准确的信息。避免统计信息更新不及时影响准确性。

