# 概述

## 概念

Leader：用户连接Session所在的线程，一个并行查询只有一个Leader

Worker：Leader线程会启动多个**辅助线程**来执行**同一个查询**，这些辅助线程就是Worker，多个Worker对应同一个Leader线程，从概念上讲Leader也是一个Worker，在实现中假设为Worker 0

Collector：是Leader上的一个模块负责处理从Worker接收数据并负责管理Worker的生命周期，目前一个Leader里只有一个Collector，将来可以扩展成多个来支持更新复杂的查询计划

说明：这里的Collector相当于MPP中Coordinator。

Parallel Plan：目前是在MySQL查询计划基础上经过并行化改造，生成的可以由多个Worker一起协同执行一个查询的查询计划，目前只支持二阶段并行，所以计划包括Leader和Worker执行的Partial Plan

Partial Plan：Parallel Plan把原来的MySQL串行计划分成了2个部分Leader和Worker的，每一部分我们都称之为Partial Plan，在代码中一般指Worker的计划（因为Leader上的计划，是直接修改的原来MySQL生成的串行计划），每个Worker会从这个计划clone一份来执行。

说明：这里的ParallelPlan属于逻辑概念，类似Presto中的LogicPlan，PartialPlan属于优化后的物理执行计划，类似Presto中的Stage。

channel：数据流转使用的信道，主要包括三种类型：Memory（默认），TCP，bRPC，信道中的结果由exchange负责获取。

exchange：进行行数据的交换，即从多个worker如何获取结果集，主要包括两种类型：FIFO，Sort-Merge（数据有序）。

注：这种并行查询架构与presto的Coordinator-Worker的架构类似。

## 背景

目前Table在TDStore上的存储表数据和index是分开存储的，同一行数据并不一定在同一个节点上，导致二级索引回表时比较慢，如果支持了并行，可以并行操作性能可以线性加速。

使用：注意要想查询走并行，还要注意下面变量的值：

1. max\_parallel\_degree大于0
2. Plan的cost大于parallel\_plan\_cost\_threshold设置的值
3. 表的记录数大于parallel\_scan\_records\_threshold设置的值
4. 评估的并行扫描Range数（Region数）大于parallel\_scan\_ranges\_threshold设置的值

# 用户使用

## 使用限制

* 只支持SELECT
* 只支持单表，并且是普通表和RocksDB引擎表
* 如果表是下列Access方式则不支持：

Index用于Group By（Using index for group-by)

Dynamic Range Scan（Range checked for each record）

Index Merge Scan (index\_merge)

分区表是ref\_or\_null

* 不包含Window函数
* 不包含Rollup
* 相关子查询，包括：

表达式上没有子查询引用

如果在一个子查询内，子查询不引用父查询

* subquery\_to\_derived 打开时的转出的Derived Table. 参考：待增强功能
* SQL中的表达式都Parallel Safe的，详见Item表达式的处理，目前典型的不支持表达式：

自定义函数

包含子查询的表达式，比如：IN、EXISTS

GROUP\_CONCAT

用户变量inline赋值：`@a`:=1

* COUNT(\*) 不走并行，也包括被现在的下推逻辑优化了的，比如：SELECT COUNT(\*) FROM T WHERE COND，下推逻辑已经支持多个并行task处理了

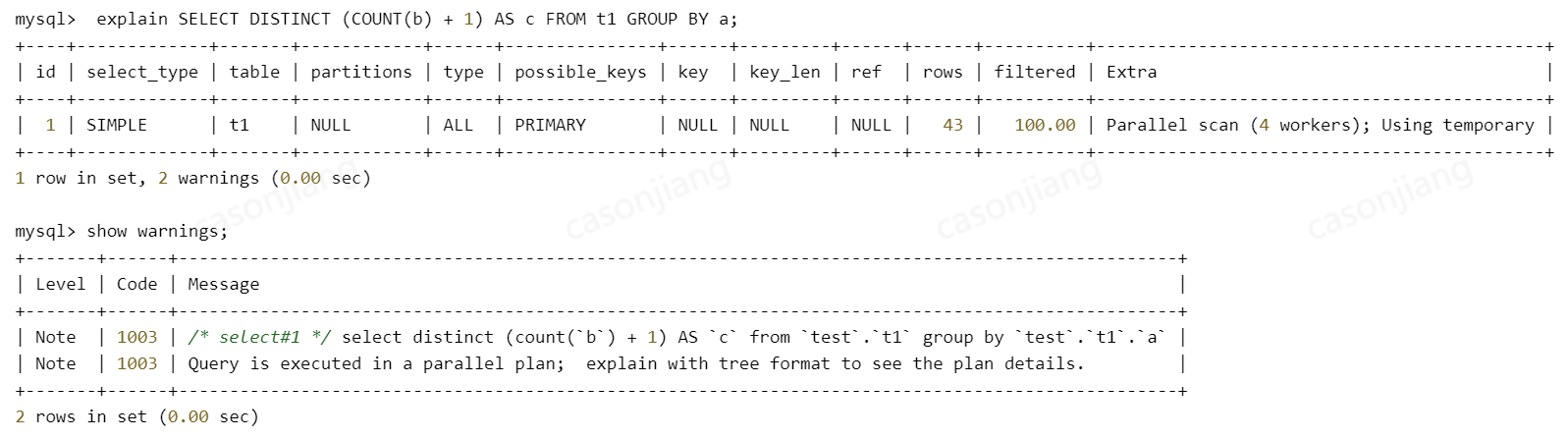
## 使用命令

### Explain命令

下面的SQL：

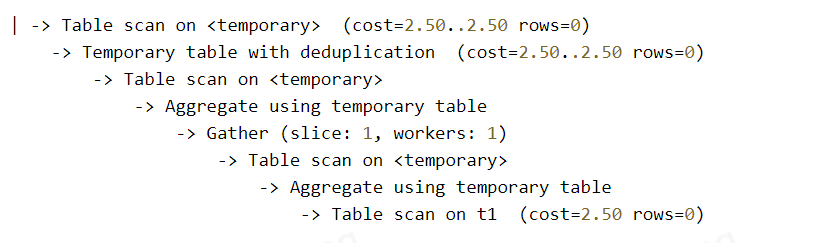
EXPLAIN SELECT DISTINCT (COUNT(b) + 1) AS c FROM t1 GROUP BY a;

* traditional 格式：



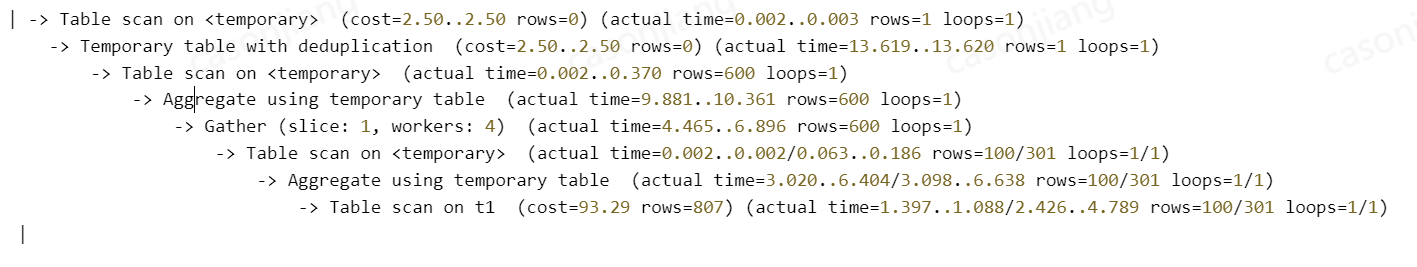
Warning中，提示使用tree格式查看详细的parallel plan。

* tree 格式



Gather (slice: 1, workers: 1)：表示有1个worker，worker发元组给Leader， slice: 1表示它下面的是第1个计划分片（Leader上的计划分片号总是0），为将来多计划分片预留。

* explain analyze：这里Gather之下是在多个worker上执行的，acual 后面：/ 之前的是执行用时最短的worker，/ 之后的是执行用时最长的worker



### PARALLEL hint

可以使用PARALLEL hint指定进行parallel scan的表和它的并行度。

MySQL的Hint可参考：Optimizer Hints，Parallel Hint是一个3个层级的Hint Global、Query Block和Table级别，语法如下：

PARALLEL(num) -- Global级、Query Block级

PARALLEL(@qbname, num) -- Query Block级

PARALLEL(tablename) -- 表级

PARALLEL(tablename, num) -- 表级

NO\_PARALLEL -- Global级、Query Block级

NO\_PARALLEL(@qbname) -- Query Block级

NO\_PARALLEL(tablename) -- 表级

其中，NO\_PARALLEL是禁止并行，tablename和qbname的语法请参考mysql相关文档。

* Global级：当在主查询中指定，没有给定query block name和table name时。这时num将做为整个查询的默认并行度，子查询还可以通过自己的PARALLEL Hint来覆盖它。
* Query Block级：当在非主查询中指定，或者给定了query block name，但没有给定table name时。这时num将做为这个子查询的默认并行度，可以被表级的覆盖。
* Table级：当在Hint中给定了表名时。如果只给表名没给num，会向上（即：Query Block、Table级别、max\_parallel\_degree变量）确定并行度。我们可以支持指定多个表并行，将来的计划中会支持多个表并行。

举例：

EXPLAIN select /\*+ PARALLEL(4) \*/ \* from t1 where a > 4; -- Global 级别

EXPLAIN select /\*+ QB\_NAME(q1) PARALLEL(@q1 4) \*/ \* from t1 where a > 4; -- Query block 级别，只在最顶层查询起作用

explain select \* from t3 where a = (select /\*+ PARALLEL(2) \*/ count(a) from t3); -- Query block 级别，只在子查询中有效

EXPLAIN select /\*+ PARALLEL(t1 4) \*/ from t1 where a > 4; -- Table 级别

EXPLAIN select /\*+ PARALLEL(t3@q2 4) \*/ \* from t3 where a = (select /\*+ QB\_NAME(q2) \*/ count(a) from t3); -- Table 级别

EXPLAIN select /\*+ PARALLEL(@q2 t3 4) \*/ \* from t3 where a = (select /\*+ QB\_NAME(q2) \*/ count(a) from t3); -- Table 级别

注意:

Hint会强制使用或不使用并行，会覆盖 max\_parallel\_degree变量，并且不再检查 cost 和 table records是否满足设置的阈值。

用 PARALLEL hint 与固定查询计划结合可以在不修改语句的情况下，对特定的语句开启并行，如:

call dbms\_admin.statement\_outline\_add\_rule('test', 'select /\*+ PARALLEL(4) \*/ sum(c),b from t1 group by b');

## 变量

### 系统变量



这里可以看到如果要不考虑代价和表大小强制使用并行可以设置：

* parallel\_scan\_records\_threshold = 0
* parallel\_scan\_ranges\_threshold = 1
* parallel\_plan\_cost\_threshold = 0

注意要想查询走并行，还要注意下面变量的值：

* max\_parallel\_degree 大于0
* Plan的cost大于parallel\_plan\_cost\_threshold设置的值
* 表的记录数大于parallel\_scan\_records\_threshold设置的值
* 评估并行扫描range数大于parallel\_scan\_ranges\_threshold设置的值

### Status变量

Session和Global 2个维度，Global表示整个SQLEngine节点的

* PQ\_workers\_created session：累计的并行worker生成数量
* PQ\_rows\_exchanged：累计并行查询worker和leader、worker和worker之间交换row的数量

## Optimizer Trace并行查询信息

在原来MySQL的Optimizer Trace的 join\_optimization 的 steps中加入一个step “parallel\_plan"（位于Optimize最后的”refine\_plan“之后），即路径如下：

{ "steps": [

{

"join\_preparation": { ... }

},

{

"join\_optimization": {

"select#": 1,"

"steps": [

...

{

"refine\_plan": [ ... ]

},

{

"parallel\_plan": {

"steps" [ ... ]

}

}

]

}

}

]

}

正常的 parallel\_plan 对象包括:

* steps[0] = considering，当没有选择Parallel Plan时，会在这里输出原因，如：

"considering": {

"chosen": false,

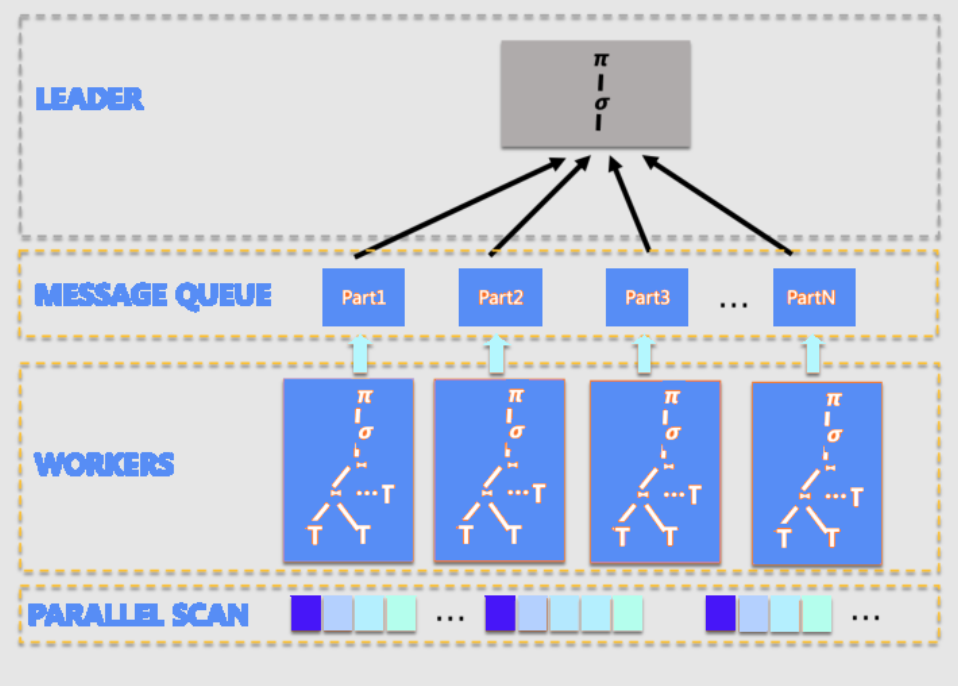
"cause": "not\_single\_table\_or\_just\_const\_tables"

}

* steps[1] = "generating"，用于在Plan生成过程中的trace，他又包含了几个step，与代码逻辑是对应的。
* 将来如果必要可能会加上 steps[2] = "executing"，用来显示执行时的trace。

# 架构

类似于Postgresql，支持二阶段并行：



整个在实现上分为并行优化和并行执行2个阶段，并行优化是生成并行Plan的过程，是在MySQL串行计划基础上进行的，这里将Plan和表达式的clone也归为优化的过程。

当前TDSQL3.0的并行查询**基于AccessPath tree来构建计划**，支持基本的group by和order by查询。

目前MySQL传统计划生成之后会生成AccessPath，分析现在JOIN::optimize代码，官方在使用hypergraph optimizer时是不构造QEP\_TAB的，是否可以在AccessPath生成之后，通过操作AccessPath tree来生成并行Plan？不对QEP\_TAB进行clone？目前看依赖于QEP\_TAB的：

semi-join weedout

dynamic\_index\_range\_scan

**说明：**QEP\_TAB对应的每个Query\_block涉及到的具体“表”的顺序、方法和执行计划。

QEP\_TAB是Query Execution Plan Table的缩写，这里的表Table对象主要包含物化表、临时表、派生表、常量表等。JOIN::optimize()是优化执行器的统一入口，在这里会把一个查询语句块Query\_block最终优化成QEP\_TAB，得到AccessPath。

首先在JOIN::optimize()中生成QEP\_TAB，然后生成AccessPath。

这一步尝试一下。得到的如下好处：

1、避免QEP\_TAB的处理

2、传统MySQL计划在处理Group by、 Order by非常细碎和烦琐，特别是建立执行使用的临时表，期望使用AccessPath可以简化这一过程

3、是否可以避免或减少JOIN的clone处理？

4、支持HyperGraph optimizer

AccessPath仅是MySQL包括查询执行的步骤，完整的查询计划仍然依赖于大部分的传统计划部分，包括Query\_block、JOIN里的部分东西，比如Leaf Tables、Fields。

# 原理

总体上，并行执行框架包括执行计划并行优化和并行执行优化。

## 并行框架入口

在join优化完成后，query expression会根据生成的access path创建iterator，因为之前在原计划中添加collector table的access path，所以此时也会生成CollectorIterator，并行框架的入口就隐藏在CollectorIterator中，在Iterator的执行过程中会自动被调用。当优化器优化完成后，会调用Query\_expression->ExecuteIteratorQuery从root\_iterator开始，迭代执行，依次先迭代执行每个iterator的Init，然后再迭代执行每个Iterator的Read，直至结束。所以并行框架的执行入口就是CollectorIterator::Init，执行过程中会反复调用CollectorIterator::Read。

主要流程：

### CollectorIterator::Init

1、CollectorIterator::Init初始化worker，通知worker开始执行

1）empty\_record

collector表的初始化

2）Collector::Init

初始化collector相关的worker，并通知worker开始执行

### CollectorIterator::Read

2、CollectorIterator::Read接收worker的执行结果，直到全部完成或被中止

- RowExchangeFIFOReader::Read

正常的读取下一行数据，不需要有序

1）ReadOneRow

从下一个channel中读取一行，如果当前channel读完或尚未准备好数据，则按round robin方式切换到下一个。

如果没有任何channel准备好数据，并且也没有出错，则wait在event上，等待数据消息的通知。

1.1）::Receive

根据不同的信道类型，调用不同的Receive接口接收数据

1.2）RowSegmentCodec::SegmentHasData

接收并处理没有group by的count()，将结果保存到collector table的record[0]中。

1.3）RowSegmentCodec::SetSegment

接收结果，将结果保存到collector table的record[0]中，但需要对AGG的结果做一些特殊处理，比如AVG，已经分为sum()+count()，需要特殊处理。

2）AdvanceChannel

如果当前channel数据已经接收完成或数据未准备好（异步接收未结束），切换到下一个channel

## 通信机制

在TDSQL3.0并行查询框架中，行数据的发送通过三种方式实现，分别是：

* memory（row\_channel.cc+message\_queue.cc）
* TCP（row\_channel\_tcp.cc+event\_service.cc）
* bRPC（row\_channel\_brpc.cc+brpc\_stream\_connection.cc）

在类RowChannel中封装了数据收发的一些主要接口，包括：Init、Send、Receive、SendEOF、Close等。

在row\_exchange.cc中，封装了结果集交换（Collector与Worker之间数据交换）读写操作的接口，底层是调用上述三种信道的主要接口（主要是executor.cc调用）。主要接口：

1. CreateRowExchangeReader，在Collector::Init阶段调用，主要是初始化行记录转换的Reader（包括Merge-Sort Reader、FIFO Reader）
2. RowExchange::Init，在LocalWorker::Init阶段调用，用于初始化信道数组m\_channel\_array
3. RowExchangeWriter::Write，构造数据，然后调用不同信道的Send接口发送
4. RowExchangeWriter::WriteEOF

### Memory

在LocalWorker初始化过程中，会根据不同信道类型，生成不同的信道结构，memory row channel是默认的信道，底层通过消息队列实现，主要接口包括：

1. 构造MemRowChannel对象（CreateMemRowChannel）
2. 调用Init接口，初始化message queue和对应的事件event类型
3. 设置对端的event

说明：

Collector::Init

->LocalWorker::Init

->CreateMemRowChannel

->CreateTcpRowChannel

->CreateBrpcStreamRowChannelPair

->RowExchange::Init

->PartialExecutor::Init

### TPC

在并行框架内部，TCP通信底层实现基于epoll实现的。内部通过独立的线程通过epoll\_wait监听，然后循环遍历epoll\_fd，借助互斥锁和条件变量同步事件状态m\_set。

在mysqld服务停止的时候，会调用StopEventService接口执行关闭操作。

### bRPC

## 并行执行计划

**入口函数：**sql/parallel\_query/planner.cc/GenerateParallelPlan

**功能：**并行框架中并行计划生成入口（planner.cc是执行计划相关）

**调用：**sql/sql\_optimizer.cc/JOIN::optimize()

位于JOIN::optimize的create\_access\_path之后，设置plan状态为Plan\_ready之前。

**说明：**这部分生成执行计划的逻辑主要在planner.cc中实现，会调用executor.cc生成Collector，在生成并行的AccessPath时调用rewrite\_access\_path.cc的接口重写AccessPath。

### 执行计划生成过程

AccessPath的处理步骤：

1、在JOIN::optimize生成AccessPath后，调用GenerateParallelPlan->ChooseParallelPlan()，判断是否可以使用并行查询，如果可以生成，则确定并行扫描的表，并生成并行计划对象。

入参是JOIN结构体，大致实现逻辑：

1. 过滤一些不支持的复杂SQL
2. 生成ParallelPlan
3. 将需要并行化的存储到ParallelPlan成员变量m\_partial\_plan（PartialPlan）

说明：

在执行JOIN的Optimizer优化器的时候，处理并行查询的执行计划，生成新的AccessPath，然后构造新的Iterator，最后调用JOIN::exec执行。

JOIN::optimize（sql/sql\_optimizer.cc）

->GenerateParallelPlan（sql/parallel\_query/planner.cc）

->ChooseParallelPlan（planner.cc）

->GetTableParallelScanInfo（planner.cc）

->ChooseParallelDegreeByScanRange（planner.cc）

->ItemRefuseParallel（planner.cc）

->AcquireParallelWorkers（planner.cc）

->ParallelPlan::ParallelPlan（planner.cc）

->ParallelPlan::GenerateAccessPath（planner.cc）

-> ParallelPlan::CreateCollectorAccessPath

-> NewParallelCollectorAccessPath（access\_path.h）

-> AccessPathParallelizer::parallelize\_access\_path（rewrite\_access\_path.cc）

-> AccessPathRewriter::do\_rewrite

-> Collector::CreateMergeSort（executor.cc）

2、然后调用ParallelPlan::Generate生成并行Plan

根据之前已经确定的并行扫描的信息，生成对应的并行计划。

说明：主要参数仍然是JOIN结构体，大致实现逻辑：

1. ParallelPlan::GeneratePartialPlan

生成worker子计划，真正执行时由worker从此计划clone，然后再执行。

说明：首先生成当前的partial query\_block对应的JOIN，克隆基本对象（Field、order by、group by等）持久化到JOIN对象，然后通过ParallelPlan::CreateCollector生成Collector（executor.cc）。

1. ParallelPlan::GenFinalFields

由于原计划已经拆分出worker子计划，那么原计划中一些表列，如投影列就不能映射到基表，因为原计划中可能已经没有基表，这些列表，将来会从worker子计划的执行结果中析取，因此必须对它们做重新映射，也就是映射到收集worker执行结果的collector table中。

1. ParallelPlan::GenerateAccessPath

根据拆分后物理计划生成生成access path，原计划中被拆分的子计划部分被collector table代替。因此，也要生成新的collector access path来代替原有的access path。

说明：这里主要是调用rewrite\_access\_path.cc的接口实现并行AccessPath的重写操作。

**调用：**

JOIN::optimize（sql\_optimizer.cc）

-> ParallelPlan::Generate（planner.cc）

-> ParallelPlan::GeneratePartialPlan（planner.cc）

-> ParallelPlan::GenFinalFields（planner.cc）

-> ParallelPlan::GenerateAccessPath（planner.cc）

3、并行执行计划ParallelPlan生命周期管理

在计划树JOIN执行对应的reset/free等操作时，还需要一并处理对应的ParallelPlan（本质上就是Collector的生命周期管理）。

### Item表达式的处理

支持clone，为Item加入parallel safe属性，有3个值：

Safe，可以并行可在worker里执行

Restricted，只能在Leader执行，比如user function，可能会触发对表的修改

Unsafe，整个查询不能并行，目前没有实现clone的会置为这个值

Item基类默认是Unsafe的。没有实现clone的Item parallel\_safe()应该返回Unsafe，比如UDF相关Item，因为并行在Leader上的计划也依赖于Item clone。

目前的处理：只要包含非Safe的Item，就禁止并行，后面Restricted不会阻止并行，Item会在Leader上执行，包括Agg（Distinct），Window functions等。

除以下列出的特殊的Item外，其它的Item都已经完整支持clone，也包括JSON，GIS等函数。

#### item

通用和基本的Item位于 Item.cc中。

没有实现clone的：

Item\_view\_ref（之前的处理有缺陷，暂时禁掉）

Item\_outer\_ref

Item\_ref\_null\_helper

Item\_trigger\_field (查询计划上不会用到）

Item\_insert\_value（查询计划上不会用到）

#### item\_func

目前基类Item\_func，只是简单的看参数是否safe，也就是默认是safe的。

不支持并行的Item：

Item\_func\_rand 在没有参数和参数是常量时是Restricted，因为这种情况下seed是相同的，worker间会产生相同的结果

Item\_udf\_func Unsafe

Item\_func\_get\_lock Restricted

Item\_func\_release\_lock Restricted

Item\_func\_release\_all\_locks Restricted

Item\_source\_pos\_wait Restricted

Item\_master\_pos\_wait Restricted

Item\_wait\_for\_executed\_gtid\_set Restricted

Item\_master\_gtid\_set\_wait Restricted

Item\_func\_set\_user\_var Unsafe 不支持在查询中设置 user var（这种行为官方已经Obsolete了），get user var是支持的

Item\_func\_match Unsafe Full Text相关

Item\_func\_sp Unsafe（后面可以变为 Restricted）

#### item\_sum

下面没有实现，设置了parallel safe 为 Unsafe：

Item\_sum\_json

Item\_udf\_sum

Item\_func\_group\_concat

#### item\_cmpfunc

下面没有实现，设置了parallel safe 为 Unsafe：

Item\_in\_optimizer

Item\_func\_trig\_cond

Item\_func\_not\_all

Item\_is\_not\_null\_test

Item\_equal（不需要实现，只有在optimize阶段使用）

#### item\_subselect

子查询暂时不支持。都是Unsafe的。

## Worker执行流程

1、在Leader上执行到executor.cc/CollectorIterator::Init()时，会调用Collector::Init()，Collector对象维护Worker数组，在这个函数里会生成Worker对象并启动Worker线程

说明：

CreateIteratorFromAccessPath（executor.cc）

->CollectorIterator::Init（executor.cc）

-> empty\_record（table.h）

-> Collector::Init（executor.cc）

-> CreateLocalWorker（worker.cc）

-> LocalWorker::Init（worker.cc）

-> RowExchange::Init（row\_exchange.cc）

-> CreateRowExchangeReader（row\_exchange.cc）

->Collector::LaunchWorkers（executor.cc）

->LocalWorker::Start（worker.cc）

->RowExchangeFIFOReader::Init（row\_exchange.cc）

2、依次对每个远程worker开始初始化（worker.cc/LocalWorker::Init）

1）根据不同的信道初始化（memory、TCP、bRPC），初始化收发信道：

m\_receiver\_channel：Collector接收Worker结果集使用的信道

m\_sender\_channel：Worker向Collector发送结果集使用的信道

m\_sender\_exchange：Worker进行数据交换的类

2）初始化m\_sender\_exchange的RowChannel数组m\_channel\_array

3）m\_row\_exchange\_writer初始化

调用：

LocalWorker::Init（worker.cc）

->CreateMemRowChannel（row\_channle.cc）

->CreateMemRowChannel::Init

->SetPeerEventForMemChannel

->CreateTcpRowChannel（row\_channel\_tcp.cc）

->CreateTcpRowChannel::Init

->CreateBrpcStreamRowChannelPair（row\_channel\_brpc.cc）

->CreateBrpcStreamRowChannelPair::Init

->RowExchange::Init（row\_exchange.cc）

->PartialExecutor::Init（executor.cc）

-> RowExchangeWriter:: SetExchange（row\_exchange.cc）

在生成Worker对象的同时会生成对应的THD，这里修改了原来THD的Init过程，让它从Leader的THD上复制variables这就完成了session变量的同步。

3、当所有远程worker初始化完成后，初始化collector的m\_receiver\_exchange，输入每个worker的receiver channel，关联所有worker的receiver channel到collector exchange中（即这样建立Collector与Worker的收发关系）。

说明：row\_exchange.cc/RowExchange::Init接口实现。

4、为collector基于receiver exhcange创建CreateRowExchangeReader，得到m\_row\_exchange\_reader对象，用于按行读取远程结果集。

说明：

CollectorIterator::Read接收worker的执行结果，直到全部完成或被中止

- RowExchangeFIFOReader::Read

正常的读取下一行数据，不需要有序

1）ReadOneRow

从下一个channel中读取一行，如果当前channel读完或尚未准备好数据，则按round robin方式切换到下一个。

如果没有任何channel准备好数据，并且也没有出错，则wait在event上，等待数据消息的通知。

1.1）::Receive

根据不同的信道，调用不同接口实现数据接收

1.2）RowSegmentCodec::SegmentHasData

接收并处理没有group by的count()，将结果保存到collector table的record[0]中。

1.3）RowSegmentCodec::SetSegment

接收结果，将结果保存到collector table的record[0]中，但需要对AGG的结果做一些特殊处理，比如AVG，已经分为sum()+count()，需要特殊处理。

2）AdvanceChannel

如果当前channel数据已经接收完成或数据未准备好（异步接收未结束），切换到下一个channel

2.1）RowExchangeMergeSortReader::Read

读取下一行有序的数据，需要做merge sort排序后再输出

- MergeSort::Read # merge sort后输出下一行，采用优先队列方式。

说明：如果结果集需要有序，则创建RowExchangeMergeSortReader，否则创建RowExchangeFIFOReader。

5、Collector::InitParallelScan

6、LaunchWorkers启动Worker

LocalWorker::Start

->launch\_worker\_thread\_handle

-> LocalWorker::ThreadMainEntry

->PartialExecutor::ExecuteQuery

->PartialExecutor::PrepareQueryPlan

-> ::execute

->PartialExecutor::EndQuery

1）调用PartialExecutor::PrepareQueryPlan

在启动后拿到Leader给它的执行计划，这里在代码里抽象成Partial Plan，Clone一个做为自己的执行计划，参见pq::PartialExecutor::PrepareQueryPlan()，这里的执行计划，包括：

* 打开的表
* Query\_expression，对应lex->unit
* Query\_block，对应的 lex->unit->slave
* JOIN，对应 lex->unit->slave->join
* JOIN中比较关键的是对AccessPath的复制，最后会通过AccessPath生成Iterator用于后面执行

在Clone计划的最后，设置QueryResult到我们新建的Query\_result\_to\_collector，由该模块将结果发回Leader

说明：

PartialExecutor::PrepareQueryPlan

->Query\_result\_to\_collector

2）然后调用MySQL本身的执行流程：即：Query\_expression::execute()，来执行查询

3）调用PrintQueryPlanTiming

如果执行是Explain Analyze，那么最后会将Explain Analyze的性能数据保存在Worker对象上，在Leader上会读取这个数据

7、RowExchangeReader::Init

前面构造了对象，这里根据具体Reader的类型，进行初始化：

1. RowExchangeFIFOReader::Init
2. RowExchangeMergeSortReader::Init

8、Leader上在适当的地方会调用Collector::End()，在该函数里会保证所有worker都结束，如果没有结束，会调用Kill来结束Worker线程，然后处理Worker的返回状态，包括收集Worker报的Warning等信息，如果Worker是Error状态那么Leader会向客户端报Error

说明：

Query\_expression::execute（sql\_union.cc）

->Query\_expression::ExecuteIteratorQuery（sql\_union.cc）

->JOIN::join\_free（sql\_select.cc）

->JOIN::end\_parallel\_plan（executor.cc）

->ParallelPlan::EndCollector（planner.cc）

->Collector::End（executor.cc）

->Collector::TerminateWorkers（executor.cc）

->LocalWorker::Terminate（worker.cc）

->Collector::CollectStatusFromWorkers（executor.cc）

## 多表Join支持

目前只支持第1个表做Parallel Scan。

目前限制：Semi Join

1、Loose Scan，主要是Parallel Scan不支持Prefix，没有写出来不在第1表做Loose Scan的查询

2、First Match，进行 First Match扫描的表不能做Parallel Scan，因为如果表上有重复数据，则结果集错误，会多数据。

3、Duplicate Weedout，依赖QEP\_TAB

4、Materialization Lookup/Scan, access path需要重建临时表，需要单独提交

## MDL并行支持

如果不做任何修改，并行的多个worker（这里把Leader看作一个worker），各自去申请MDL，即使是只有Share Read也可能产生死锁，下面是最简单的一个死锁case：

1、Leader拿到t1的S锁（Worker还没有拿锁）

2、在执行并行没启动worker之前，另一个线程的DDL等待拿t1的X锁

3、这时leader启动了worker线程，worker线程会去拿S锁，但拿不到，因为前面等待一个X锁

这时死锁就发生了。另外，worker也会自己去拿Leader没有持有的对象锁，比如在做Time相关函数去拿timezone相关表的表锁，在做GIS函数相关的表锁，这会使问题更复杂，Worker之间会发生死锁。

为了在解决以上问题，在MDL上实现了Group Lock。

### Group Lock

这里有一个前提是：在执行一个Query的并行线程之间没有锁冲突，其中的一个线程拿到锁，可以假定所有线程都拿到。

在MDL中实现Group Lock机制：在一个Group中一个线程拿到锁，另一个线程会无条件拿到，不会做锁等待。

### 实现难点

MDL中有fast path，fast path获取的锁不会出现在锁表上，这使用同组的其他worker拿到已获取的锁很困难。

MDL\_context中，存储自己获取锁时没有锁保护，因为在非Group下，它假定只有自己会访问。

### 实现方案

并行与MDL Lock Group的交互：

1、并行的Leader在启动并行Worker之前，会成为MDL Lock Group的Leader，通过函数Collector::PrepareExecution->MDL\_context::init\_lock\_group()

2、并行的Worker会成为MDL Lock Group的Member，通过函数 PartialExecutor::InitExecThd->MDL\_context::join\_lock\_group()

3、在并行Worker都退出后，并行Leader会调用Collector::Destroy->MDL\_context::deinit\_lock\_group()结束整个Lock Group

MDL Lock Group内的修改：

1、在init\_lock\_group()中，group leader把自己的fast path lock全部 materialize到锁表上，这样其他member只需要访问锁表时就可以拿到本组拿到的所有锁。

2、在Group中的成员拿锁时不使用fast lock，这样同组其他人可以看到。

3、在做是否可以can\_grant时，先遍历锁表，如果本组其它成员已经拿到了锁则，提前返回can\_grant。

这个实现在性能上有损失，后面在并行多时可能会有问题：

1、在并行的查询里禁用fast path，在正常情况下fast path应该是主要路径

2、在判断本组是否已经拿到了锁时，遍历了所有拿到该锁对象的session，可能会有性能问题

### TODO

这里是可能的方案，具体做时可能再考虑。

1、在Leader中维护本组所有成员的锁列表，这个列表需要加rwlock

2、在查看本组是否拿到了锁时只需要访问Leader上的这个列表

3、在死锁检测的中，在本组的任何成员进行wait\_for之前materialize本组的所有的fast path锁。

## Session变量同步

### 变量同步难点

同步的难点：

1、MySQL有大量的Session变量记录在THD::variables中，逐个复制overhead会很重

2、并不是所有的Session变量都直接存储在THD::variables中，比如，Plugin定义的变量，storage engine的变量，比如：innodb\_xxx，通过代码可以看到变量有3种存储方式。

### 实现方案

目前方案：修改THD的init()过程，THD::init()从global variables copy一份到当前THD上，这里对于Worker，改成了从当前的Leader的THD::variables中copy。

目前没有处理plugin定义的变量，SQLEngine目前使用的还都是variable中的变量，我们的临时表也是MyISAM表。这里修改了THD的构造函数改的有点基础，但应该影响不大。

跨节点执行时这里需要再考虑变量跨节点的同步方式。

#### debug变量同步

MySQL中经常使用set debug = 'd,var'称为debug flag的方式进行代码注入测试，mtr中有大量这种测试用例，所以并行需要支持debug flag，来跑这些测试用例。debug变量还有其它作用，比较复杂。

因为debug flag存储在thread local变量中，而标准的debug flag只是本地访问没有并发控制。所以同步的难点还是尽量少的影响正常流程，避免并发问题。

**实现：**

实现了一个RAII类：CSStackClone，该类保存了leader上debug flag的并行查询执行时的一个快照，worker在启动后会复制到自己的线程本地，并行查询cleaup时CSStackClone会析构掉自己保存的快照。

#### debug\_sync变量同步

debug\_sync所有session里的所有信息都存储在THD::debug\_sync\_control中，只需要在worker线程启动前clone该结构即可。

### TODO

# 存储引擎层ParallelScan

这是存储引擎层暴露给Server层的接口，用于并行扫描table或index。

## 类型和结构

加入一个Table\_flags: HA\_CAN\_PARALLEL\_SCAN，支持parallel scan的存储引擎，在ha\_table\_flags()中返回这个flag。

typedef void \* parallel\_scan\_handle\_t;

–只在引擎层使用，传透明指针给Server

struct parallel\_scan\_desc\_t

–用于传给引擎层的参数，定义如下：

struct parallel\_scan\_desc\_t {

// REF\_OR\_NULL is for JT\_REF\_OR\_NULL, there are 2 round scans, first for

// value ref key, second for null ref key. INDEX\_GROUP\_BY is for

// QS\_TYPE\_GROUP\_MIN\_MAX of JT\_RANGE.

enum type\_t { NORMAL, REF\_OR\_NULL, INDEX\_GROUP\_BY } type;

uint keynr; // Key number for parallel scan

key\_range \*min\_key; // Min key of range scan or ref key for ref scan

key\_range \*max\_key; // Max key of range scan, same with min\_key for ref

// scan, null ref key for REF\_OR\_NULL type

uint16\_t key\_used; // Part of keys used by parallel scan

bool is\_asc; // Ascending or descending order

};

结构说明：

type，并行扫描的类型，目前包括：

REF\_OR\_NULL，表示是一个JT\_REF\_OR\_NULL扫描，handler层期望有2个scan，每个是使用不同的 parallel scan range。第1个是ref value，第2个是ref null；

INDEX\_GROUP\_BY，待补充；

NORMAL不属于以上2个类型的。

keynr，索引的id

min\_key、max\_key在做range scan时指定在btree上的扫描范围，减少扫描范围提高性能，对于 普通的Ref Scan min\_key 和 max\_key存相同的值都是Ref key，对于 ref\_or\_null，max\_key存 Null value key，这里 max\_key\_is\_null为true

key\_used，复合索引中，按这个索引列进行worker间的数据的划分，比如：index(a, b, c)，如果指定了1，那么只看第1列进行worker间的数据划分，也就是存储层要保证在a列上在worker间数据是不重复的，否则查询结果可能错误。目前主要用于Group Min Max。

is\_asc，正序还是逆序

enum ha\_extra\_function 新加一个enum：HA\_EXTRA\_TOGGLE\_PARALLEL\_SCAN\_INNER，用于Group min max，在Group min max 扫min/max值，及init时使用。

## 函数

int init\_parallel\_scan(parallel\_scan\_handle\_t \*scan\_handle, ulong \*nranges, parallel\_scan\_desc\_t \*scan\_desc)

– 初始化一个Parallel Scan，返回0为成功，非0表示handler层的错误号，错误号描述参见```handler:print\_error()```，由Leader调用

handle输出参数，表示一个parallel scan，由engine层自己描述

nranges: 输入和输出参数，建议并行扫描分隔的 range 数，输出为实际的range数。为了避免倾斜，range数一般是worker数的100倍，parallel scan以 round robin的方式来处理这些range

scan\_desc为输入参数，见以上结构

int attach\_parallel\_scan(parallel\_scan\_handle\_t scan\_handle)

– attach到一个 Parallel Scan，返回值为handler层错误号。由各个Worker调用

void detach\_parallel\_scan(parallel\_scan\_handle\_t scan\_handle) – 从一个Parallel Scan detach，由Worker调用

void end\_parallel\_scan(parallel\_scan\_handle\_t scan\_handle) – 结束一个Parallel Scan，释放资源，由Leader调用

int restart\_parallel\_scan(parallel\_scan\_handle\_t scan\_handle) - 用于 nest loop 重新扫描内表，引擎层保证每次扫描内表出来的元组顺序是一致的，返回值为handler层错误号。

bool estimate\_parallel\_scan\_ranges(parallel\_scan\_desc\_t \*scan\_desc, ulong \*nranges, ha\_rows \*nrows) - 用评估Parallel Scan分隔的Range数

scan\_desc 与 init\_parallel\_scan相同

nranges: 输入和输出参数，建议并行扫描分隔的 range 数，输出为为评估后的range数

nrows为评估的每个range中的行数

ha\_clone\_consistent\_snapshot(THD \*leader\_thd) – 从Leader 获得一致的read view, 目前有个user调用函数，需要在handler里封一个sql层可调用的函数，引擎层需要实现：clone\_consistent\_snapshot

## Server层Parallel Scan的调用

并行优化在Rewrite AccessPath Tree过程中，会计算出Parallel Scan的Table和对应的索引并记录在PartialPlan上，PartialPlan会被传到所有worker中。

1、Leader在CollectorIterator::Init中，在生成Parallel Worker之前会对每个Parallel Scan Table调用init\_parallel\_scan()接口。

调用关系：

Query\_expression::clone\_from

- CreateIteratorFromAccessPath

- CollectorIterator::Init

- Collector::Init

- Collector::InitParallelScan

- init\_parallel\_scan

2、Worker在根据Leader传给他的PartialPlan生成好自己的的Query Plan后，会对每个需要做Parallel Scan的表调用attach\_parallel\_scan() attach到对应的parallel\_scan\_hander上。

调用：

worker.cc/LocalWorker::ThreadMainEntry

- PartialExecutor::ExecuteQuery

- PartialExecutor::PrepareQueryPlan

- PartialExecutor::AttachTablesParallelScan

- attach\_parallel\_scan

# Roadmap/待增强功能

## 计划处理/基本架构支持

支持基本查询，只支持table scan，支持filter，简单的Item，支持 group by、order by aggregation。

主要完成如下工作：

1、基于白名单，拦住不支持的SQL

2、并行执行线程：Parallel Query Worker

3、Parallel Scan：支持多个线程用同一个Read View做Table Scan（只支持）

4、Worker线程和Leader线程通信通道：基于内存的Message Queue

5、基本的Item clone只支持基本表达式，重点构建基本架构

6、查询计划的其它部分处理包括Query\_block，QEP\_TAB等

7、加入必要的：System variables、Status variables、Optimizer Trace

8、Explain及Explain Analyze

### Converts a subquery to a derived table查询计划修复

Converts a subquery to a derived table 有个bug：转出的Group By的 ORDER::Item并没有指到正确的位置，这导致 access\_path rewrite时 assertion fail，相关代码：

Query\_block::decorrelate\_derived\_scalar\_subquery\_pre()

ORDER \*o = new (thd->mem\_root) PT\_order\_expr(in\_select, ORDER\_ASC);

if (o == nullptr) return true;

o->direction = ORDER\_NOT\_RELEVANT; // ignored by constructur

o->is\_position = false;

o->in\_field\_list = true;

o->used = in\_select->used\_tables();

测试case：

SET optimizer\_switch='subquery\_to\_derived=on';

CREATE TABLE t2 (a INT, b INT);

CREATE TABLE t4 (a INT NOT NULL, b INT NOT NULL);

INSERT INTO t2 VALUES (1, 7), (2, 7), (2,10);

INSERT INTO t4 VALUES (4, 8), (3, 8), (5, 9), (12, 7), (1, 7),

(10, 9), (9, 6), (7, 6), (3, 9), (1, 10);

ANALYZE TABLE t2, t4;

let $query=

SELECT b, MAX(a) AS ma FROM t4

GROUP BY b HAVING ma < (SELECT MAX(t2.a) FROM t2 WHERE t2.b=t4.b);

​

eval $query;

eval EXPLAIN $query;

​

DROP TABLE t2, t4;

目前的方案：是先检测到相关路径禁止并行。

### COUNT(\*) 走并行

### Rollup支持

### group\_concat支持

### Window函数支持

### Group Min Max计划优化

可以在Leader上加上 AccessPath::AGGREGATE 来完成，Gather加上Merge Sort来支持不带prefix的parallel scan，可能使数据在worker之间分配更平均。

issue：https://git.woa.com/tdsql3.0/SQLEngine/issues/1061#note\_90993207

### 完善的GROUP BY、ORDER BY、AGG支持

1、更加细节的查询计划Clone的处理（单个Query Block）

2、其它Parallel Scan Access方法的支持，包括Index Scan，Range Scan、Ref等。

说明：

当前已经支持GROUP BY全下推，涉及Q13。由于目前Parallel Scan的限制，目前只支持通过Unique Key和Primary Key来做分组的，等后面Parallel Scan支持任意prefix再去掉这个限制。

### 多表JOIN及UNION支持

1、支持表JOIN

2、完善的查询计划Clone处理（Query Expression及多个Query Block）

### SUBQUERY支持

待细化

## Worker之间通信

### BLOB接收改成No Wait模式

目前在做主record时Gather和Merge Sort里都会使用no wait模式，当前worker没有数据时会转到下一个，但在开始接收BLOB时改成了wait方式，这里也可以改成nowait模式

### 测试评估Brpc Row Channel实现的iobuf new操作性能影响

详见：https://git.woa.com/tdsql3.0/SQLEngine/issues/1189

### 处理row channel意外断开连接逻辑

comm::RowTxResult加入 UNEXPECTED\_EOF，在目前的逻辑当中可以用它实现assert，将来实现了Remote执行，可以更方便的处理网络异常。

目前看不一定需要：因为Worker异常断开是一定要拿到worker本身的报错信息的，目前`LocalWorker`是通过worker的THD拿到，而Remote worker应该通过control channel拿到，我如果control channel不可用，那应该报”不可用错误“，可以延后到Remote worker支持后再看

### RowChannel::SendEOF支持nowait模式

目前主要是实现Gather算子，worker端SendEOF不强依赖nowait模式，后续实现broadcast，shuffle等需要SendEOF也支持nowait模式

### pq::comm::Event::Wait() 提供 Stage Info

这样在Leader等待Worker退出时或者正在接收worker的row时，可以通过 show processlist 看到状态，方便问题运维定位问题。

## MDL及锁相关

### MDL并行支持完善方案实现

参考：

## MyRocks层

### 执行计划

#### Group min max

https://git.woa.com/tdsql3.0/SQLEngine/issues/1121

#### RefOrNull

https://git.woa.com/tdsql3.0/SQLEngine/issues/1121

### 分区表上的并行查询

https://git.woa.com/tdsql3.0/SQLEngine/issues/1221

目前只支持普通表上的并行查询，并行查询数据结构也是放到ha\_rocksdb里的，为支持分区表上的并行查询，可能需要在ha\_rockspart、ha\_rocksdb都建立相应的并行查询数据结构

### TDStore上实现并行扫描

作用：更细粒度的并行Scan，目前只在rocksdb handler层实现，只到Region粒度，有时粒度太粗了，特别是表多的情况下，对性能影响可能会放大很多

Rescan和Group Index实现：Rescan和Group Index要求Worker之间的数据不能重叠，如果重叠结果是不对的，目前的按Region分无法做到。

另外，一个问题：现在的 init\_parallel\_scan()传入的range是整个查询涉及到的range，如果有多个range，那么传入的range就是多个range的union的结果，在目前的实现中会导致多拿Region列表，将来的实现应该是不和Region绑定的，涉及的Region应该由Worker自己拿。目前的计划层Explain依赖里Region列表来很能分多少个 parallel scan分区，目前的实现太重了，如果分区不与Region绑定这个实现也不对。应该由tdstore来提供一个评估函数来实现。

### 其他

# 技术路线探索

基于MySQL传统计划生成Plan

第1版基于MySQL传统计划，这已经证明是可行的，基于它搭起整个并行架构来，在这版上完成基本的 select sum(a) from t1 where a = 1 group by b order by 1; 这样的查询。

这一版使用老的执行计划，这也是已经证明是可以工作的。即：处理和clone Query\_block、JOIN、QEP\_TAB，然后Leader和Worker走正常的MySQL处理流程，即生成AccessPath，再根据AccessPath生成Iterator。

Leader上的处理流程

1、在make\_join\_readinfo中，选择是否使用并行计划，如果是就生成ParallelPlanner对象

2、在make\_tmp\_tables\_info之前，调用 ParallelPlanner::Generate生成并行计划

3、继续走MySQL现在流程：生成AccessPath，并行路径上在Leader上加入生成Collector AccessPath。

4、最后执行时MySQL根据AccessPath生成 Iterator，在Leader上根据Collector AccessPath生成Collector Iterator，由Collector Iterator负责接收Worker返回的结果。