# 背景

基于Spider架构的分布式系统，理论上是存算分离的架构，通过分布算法将大数据尽可能平均分布到存储节点上，然后由spider节点进行统一管理，提供透明的数据库服务。由于Spider节点本地并不真正存储数据，因此对数据

的读取需要通过网络从数据节点上读取，然后再进行后续的JOIN/Filter/Distinct/Group/Order/Limit等数据库操作，与传统数据库相比，这会带来很大的网络开销，而网络开销与磁盘IO类似，相比CPU计算来说，也是⼀种相当耗时的开销，在整个查询过程中，很容易成为性能瓶颈。因此尽可能的减少网络开销是分布式数据库系统的重点任务之⼀。

由于数据是按分布算法平均分布到各个数据节点，一方面可以利用数据分布的特性，同时对多个数据节点进行查询，通过数据节点间的并行，加快数据查询的速度，达到查询性能的整体提升；另一方面还可以利用数据分布的特

点，裁剪并减少对数据节点的访问，减少需要访问的数据量和网络开销，达到查询性能提升的目的。

为了进一步减少spider节点与数据节点之间的网络开销，在数据节点端尽早的过滤数据，尽早的完成group操作，甚至做部分JOIN操作等，尽可能的减少spider节点与远程数据节点间的数据传输量，是减少网络开销的必然选择。

# 概述

## TDSQL 3.0

* 每个节点可以Access所有数据，但数据其实是分开存储的，如果数据可以本地访问就应该效率更高
* **元组和索引分开存储，元组按主键存储**
* 目前是类似透明的按索引的Key的Range存储在不同Region里
* Server层可以得到Region列表并得到的每个Region的Key范围
* **但在执行过程中Region会分裂，所以Server层在优化时得到的Region不可靠，只能做为参考**

这里在加入Remote worker之后有优化点：尽量让worker的handler层本地读数据可以加速查询，这里不能只考虑单表，多表时的本地访问应该更有效，比如：

explain format=tree select /\*+ no\_bnl(tt4,tt3) \*/ \* from tt4, tt3 where tt4.c > tt3.b and tt3.b > 1 and tt3.b < 100 and tt4.c > 200 and tt4.c < 400

EXPLAIN: -> Gather (slice: 1, workers: 1) (cost=1.87 rows=2)

-> Nested loop inner join (cost=1.87 rows=2)

-> Index range scan on tt4 using c, with index condition: ((tt4.c > 200) and (tt4.c < 400)), with parallel scan ranges: 100 (cost=0.71 rows=1)

-> Index range scan on tt3 using b, with index condition: ((tt4.c > tt3.b) and (tt3.b > 1) and (tt3.b < 100)) (cost=1.16 rows=2)

如果能能让tt3表都是本地读数据效率应该更高？

这里也有不少选项，比如：

* 上面的计划只在对应的tt3数据所在的节点上执行
* 如果分布在多个节点上，可以上tt3来做 parallel scan?

当然也需要考虑tt4表的数据量，所以这里最好的办法还是用cost去找到最优的计划。

## TDSQL 2.5

* 每个节点只存储一部分数据，只能访问本节点数据，如果worker生成不正确可能导致结果错误
* 目前支持的分布方式为与分区表类型相同：按列 hash, range, list
* 查询不用担心底层节点会变化，也可以拿到底层节点信息

2.5属于标准的Sharding系统，已经比较成熟，变化不多：

对于现在还没有实现Repartition的情况下，只有在分布键对上和一边是Sharding表一边是复制表的情况下JOIN才可以下推。

# 架构

关于重构：

* Parallel Scan引入Mode

Round Robin当前使用的方式

Preassign用于有Remote worker的方式，由Planner分配好每个Worker处理的Range

* 引入distribution key/distribution descriptor，优化器用来根据正确性和代价来生成Plan和计算执行节点，比如：对于Sharding存储方式，如果JOIN键不匹配则无法走并行路径
* distibution descriptor 对优化器只是的抽象的对象，只是调用相应的接口完成相关的功能，3.0和2.5有各自的实现
* 引入distribution adapter用来屏蔽3.0和2.5的distribution和worker之间的差异

NeedParallelScan()，2.5本身是Sharding的不需要parallel scan，修改ChooseParallelPlan()，只对NeedParallelScan()返回True的做并行Scan表的处理。

需要处理Sharding的degree的选择：由表的分布信息决定

# 原理

## 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

2.1）DeparsePlan

从物理计划中反写出SQL

2.2）CreateMySQLClientWorker

创建worker对象，每个worker对象对应着一个worker的执行，这里创建MySQLClientWorker对象，用于将反写出的SQL发送给远端数据节点，然后将来CollectorIterator::Read时接收结果

2.3）MySQLClientWorker::Init

worker对象初始化，主要是创建与远程数据节点的session连接，及创建用于接收数据的通道

- MySQLClientQueryExec::Init # 连接远程数据节点，创建session

- MySQLClientChannel::Init # 创建worker端管道服务，主要是建立消息机制，当远程有结果返回时通知collector接收。

- m\_receiver\_exchange.Init # 创建collector端的管道服务，关联所有worker的exchange channel

2.4）CreateRowExchangeReader

创建用于leader从worker读取数据行的对象，collector iterator在Read时调用此接口实现从worker读取数据行。

2.5）InitParallelScan

对并行表做扫描分片，对于spider表，目前没有意义，当前spider表是按其自然分布按节点分片的，不再另行分片。

说明：因为对于TDSQL2.5架构而言，是按照shardkey分片了，并且数据和索引不是分开存储的，这个与TDSQL3.0的架构不同，在TDSQL3.0中是需要对同一个表的数据和索引做并行查询的，但是TDSQL2.5底层采用的是InnoDB存储引擎，这里只需要对单表的partition做并行即可。

2.6）LaunchWorkers

启动并执行worker执行计划。对于spider来说，并不需要启动worker线程，只是依次将worker对象反写的SQL计划发送给各个远端数据节点来执行，此时只需要发送SQL，不需要等待接收结果。

- MySQLClientWorker::Start # 启动worker执行

- MySQLClientQueryExec::Exec # 发送query。

- MySQLClientQueryExec::SendQuery

# 发送query成功后进入RowReading状态。

- RowExchangeFIFOReader::Init

- RowExchangeMergeSortReader::Init

# 为数据接收做准备，分配接收数据用的内存。

### 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（row\_exchange.cc）

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）MySQLClientChannel::Receive

通过session从远程读取一行数据

- MySQLClientQueryExec::ReadNextRow # 读取下一行

- MySQLClientQueryExec::Exec

# 当前状态为RowReading，读取下一行

- MySQLClientQueryExec::ReadRow # fetch下一行

- MySQLClientSync::fetch\_row → mysql\_fetch\_row

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。

## 通信协议

### Memory

### TCP

### MySQL

## 生成并行计划

**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。

在这里把PartialPlan构建好之后，后面就是在执行器中通过CollectorIterator接口实现Init和Read操作，逐一进行火山模型的数据获取。

### 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（构造对象）
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PartialPlan

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。因为原计划已经拆分出子计划，这里的原计划需要增加占位算子Collector（创建临时表），并且子计划输出result table的关联。

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启动

Worker的启动主要流程位于Collector::init（executor.cc）函数中，即在Collector初始化过程中启动，调用关系：

CreateIteratorFromAccessPath（access\_path.cc）

->CollectorIterator::Init（executor.cc）

-> empty\_record（table.h）

-> Collector::Init（executor.cc）

有上述调用关系可知，最终就是在火山模型中的CollectorIterator中调用。

具体如下：

1. 调用DeparsePlan，从物理执行计划反向解析出SQL。

说明：Collector::Init调用，即在初始化Collector的时候，需要从物理计划中反向解析SQL。该函数主要调用plan\_deparser.cc实现反向解析。

这里之所以构造反向SQL，就是为了后面Worker向远端存储节点发送查询QL，这里仅仅使用的是反向SQL，而不是TDSQL3.0中的子计划。

2、为spider创建远程worker对象MySQLClientWorker（mysqlclient\_worker.cc），传入worker plan和用于收集结果的collector table。

说明：Collector与Worker之间采用的是共享内存Memory、TCP的方式通信，Worker与远端DB的通信方式是MySQL协议。这个与TDSQL3.0不同。

3、依次对每个远程worker开始初始化（mysqlclient\_worker.cc /MySQLClientWorker::Init），传入collector用于接收数据的事件event，当worker有数据到来，通过event通知collector接收。

1）调用MySQLClientQueryExec::Init。

远程query执行器对象MySQLClientQueryExec默认会创建一个MySQLClientSpider对象，用于建立到远程数据节点的session，此时初始化连接。

1.1）首先初始化到远程数据节点的session连接，通过传入的spider表，可以得到每个partition及对应的数据节点信息，然后确定与其中的⼀个数据节点建立session连接，保存mysql对象到pollfd，设置开始监听输入信息。

注意：每个worker只建立到一个partition的session连接。

1.2）设置当前stage为已连接，即connected。

2）创建MySQLClientChannel用于接收远程数据的receiver channel，传入远程query执行器MySQLClientQueryExec。

3）调用MySQLClientChannel::Init()，使用之前传入的collector的event，初始化receiver channel，即EventServiceAddFd添加当前远程session连接中的socket fd对象到句柄监听服务中。

3.1）句柄监听服务是一个独立的线程，若未启动，则创建线程并启动，主程序为EventService::mainloop

主线程采用epoll监听句柄消息，如果有，则通过event通知。此event就是之前传入的collector的接收event；

4、当所有远程worker初始化完成后，初始化collector的receiver exchange，输入每个worker的receiver channel，关联所有worker的receiver channel到collector exchange中。

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

5. 为collector基于receiver exhcange创建CreateRowExchangeReader，得到row exchange reader对象，用于按行读取远程结果集。

1）如果结果集需要有序，则创建RowExchangeMergeSortReader

# 现状

## 基于fastpath的优化

控制变量：

tdsql\_spider\_single\_part\_fast\_path

tdsql\_spider\_simple\_query\_fast\_path

接口：single\_table\_select.cc/Query\_block::is\_single\_part\_query

调用：

Sql\_cmd\_select::execute\_inner（single\_table\_select.cc）

-> Sql\_cmd\_dml::is\_single\_part\_dml

-> Query\_block::is\_single\_part\_query

-> Query\_block::all\_condition\_pushed

-> Spider\_condition\_pushdown::check\_cond\_for\_spider\_tables（sql\_spider.cc）

-> Query\_expression::select\_execute\_direct\_pushdown

-> Sql\_cmd\_select::select\_simple\_query

-> Sql\_cmd\_dml::execute\_inner（sql\_select.cc）

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

-> Sql\_cmd\_dml::execute\_single\_part\_after\_optimize（single\_table\_select.cc）

-> Query\_expression::select\_execute\_direct\_pushdown（single\_table\_select.cc）

->accumulate\_statement\_cost

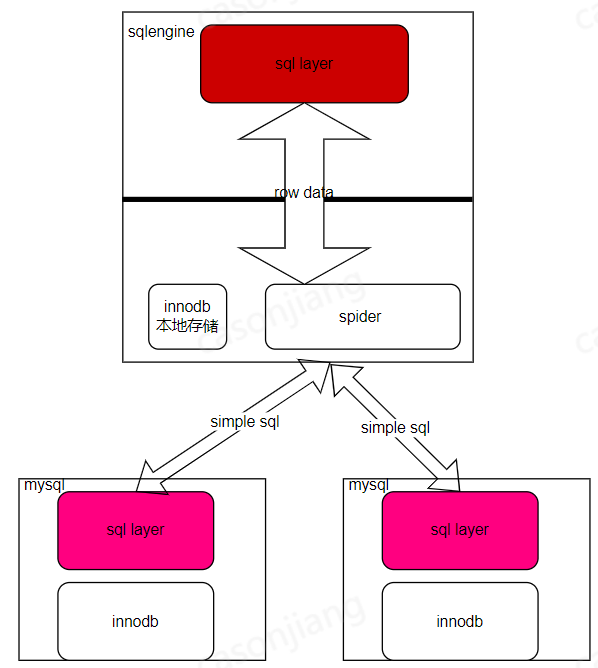
-> Query\_expression::execute（sql\_union.cc）

### 单分片查询/点查

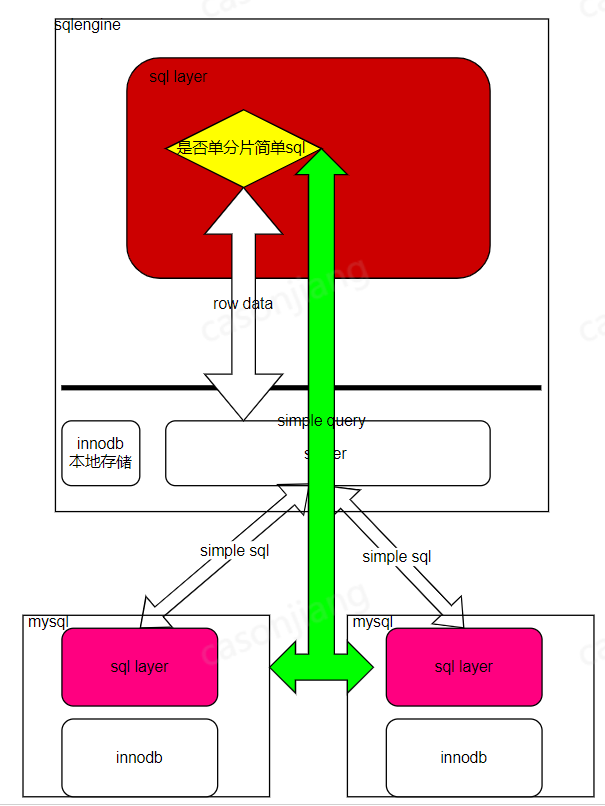
#### 背景

2.5架构下，sql处理是在计算层进行词法，语法解析，优化后再spider handler层将处理row数据转换成简单sql下发到后端存储层执行。

其中对sql的语法词法包括驱动执行在sqlengine和存储mysql节点有重复（标红部分）。



单分片下推就是想by pass掉sqlengine层的词法和执行逻辑, 将简单sql自接转发给后端的mysql。



TDSQL2.0的架构，对于在spider和后端存储重复处理的sql逻辑部分更多的下推，减少重复计算，同时还能减少计算节点网络通信数据量，提升性能。

单分片sql（通过条件减枝判断）因为不涉及多个分片数据在spider层的处理，所以实现最简单。

支持完备的执行计划树下推然后按需在spider节点进行结果集的处理遍历AST找到最大可下推子树添加对应的代理执行节点实现下推，改动比较大。

尽量支持单表的下推，总体还是采用的基于规则（整个计划树）的下推。

#### 原理

一个Spider表的数据分布方式，理论上有很多种：如hash 分布、range分布、list分布、复制分布等，其中复制分布是一种比较特殊的分布方式，指数据在每个数据节点上都是一份全量拷贝，当更新数据时，需要更新所有数据节点；当查询时，只需要访问任何一个数据节点即可。其它类型的分片，每个数据节点只包含一部分数据，并且互相之间没有交集，所有数据节点的数据集合起来才是spider表的全量数据。

如果没有限制条件，通常情况下，对一个spider表的访问需要访问全部的数据节点（除复制分布之外），但如果一个查询中包含某个或某些过滤条件，通过分片的裁剪，查询可能只访问其中一个数据节点的数据，对于这种情况，可以将查询转发到这一个数据节点，然后将这个数据节点返回的数据转发给客户端即可。这种类型的查询，我们可以称之为单分片快速转发查询。如果查询中包含多个表，并且where条件和JOIN条件可以将每个表的访问都限制到同一个数据节点的数据，则整个查询也可以称之单分片查询。

##### 概要设计

希望完全支持单表的下推，其中包括单表sql组成要件：where条件，limit + offset，order by，distinct，聚合函数（agg + group by）。

一般分成存储计算 + 计算层计算，先支持计算层处理逻辑比较简单的。

关键点：

* 基于规则在计算节点拆分重组旁路执行，结果保证和基于spider handler接口完全一样
* 准确控制支持下推的限制条件，之前的sql能力明确化分为能下推和不能下推两个集合执行
* 先提供一个全局参数 single\_table\_pushdown用于控制是否开启这个功能 （是否考虑按规则独立的开关配置？）

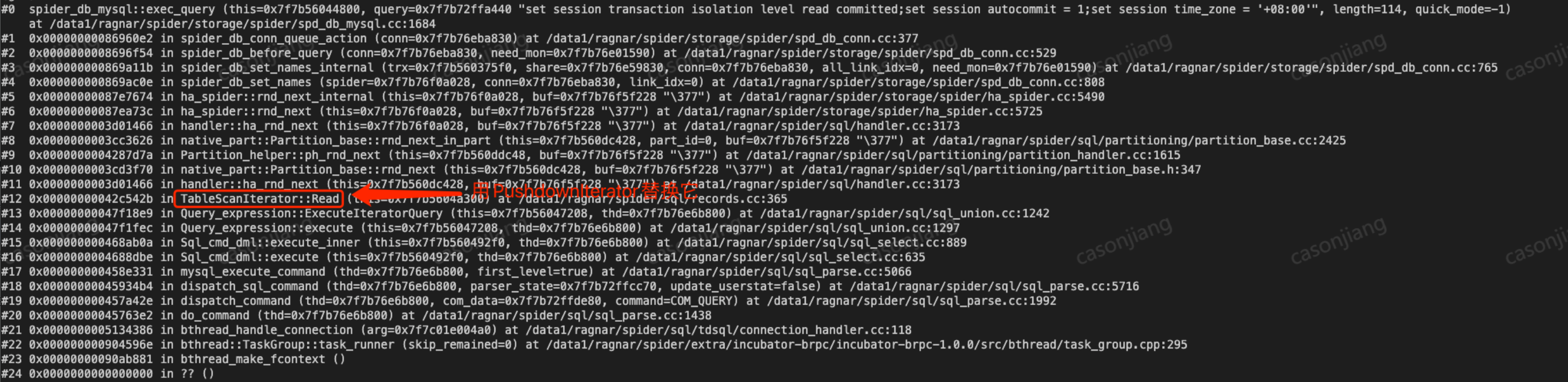
主要分为3大块儿：

1. 条件判断，判断那些sql可以进行单表下推→输出下推类型（AccessPath)→sql层

2、sql改写，为了得到正确的执行结果，根据下推类型改写要下推的sql， 然后下推到存储执行（这儿先串行，后续考虑并行优化？）→ spider handler层

3、合并结果集，根据下推类型决定怎么处理结果集，比如不同的agg，对结果集的处理不同（简单的直接合并就好，有些复杂或者没法处理的先不支持）→sql层，如何利用原有的执行器？

2和3 封装在XxxIterator里方便后续子树下推复用，单分片的查询最好也封装成SingleSetPushdownIterator。



##### 详细设计

主要基于mysql的sql处理逻辑上进行二次开发，因为是基于单表的整树下推，根据每种下推规则实现一个PushdownIterator。

###### 下推条件判断

在sqlengine处理sql流程中，能够越早判断下推越好，但是越早判断能够准确确认是否单分片的信息越少，这儿为了保证逻辑的准确性还是复用mysql原生的词法语法分析逻辑保证完备性，目前实现主要在两个地方进行了判断，一个是语法分析后prepare时，另一个就是在优化器初始阶段，这两个地方分别有分支裁剪的逻辑，单分片sql分区裁剪后只有一个分区的数据。

具体的条件包括：

* 不是explain
* 简单sql (不包括union和多层排序)
* 包含物理表
* 所有的表都是spider的分区表，而且分区裁剪后所有的表都只访问相同的分区

通用单表判断条件：

* 是否打开了single\_table\_pushdown
* 是否为simple select：
* sql command为SELECT
* 引擎为SPIDER（如果只支持这个引擎，assert？)
* 没有子查询，只有单层查询
* 单表查询，没有join
* 没有union
* 没有cte（是否有影响？）

在实践中，并不是所有单分片查询都可以直接转发，还是有一些限制条件，如：

* 包含自定义函数/过程的查询

若自定义函数/过程中包含对另一个表的全量查询，那么直接转发到数据节点执行就会导致数据丢失或错误。

* 包含不定函数的查询

标记为undetermined的函数，这些函数的执行与实例自身或服务器系统环境相关，在不同节点上执行结果不同。

* 若是多表查询，则每个表的分片必须具有相关性

如：对hash分布方式，每个表的shard key必须相同，⽬前2.5只支持这种分布方式。

select

在Query\_expression中添加下推的执行接口execute\_for\_spider，主要就是循环调用handler层的下推查询接口：

ha\_rnd\_next\_spider，顺序按行读取返回的结果返回给前端client

**在ha\_spiderpart和ha\_spider中实现rnd\_next\_spider接口下发原始sql到远端然后读取结果**。

这儿需要处理聚合函数等特殊field的情况，通过临时表接收结果集（不进行agg计算）发送给前端。

select agg(t1.c1)，t2.c2 from t1, t2 where t1.id=t2.id group by t1.c2;

where

目前spider应该已经支持了部分下推，比如基于索引的查询（主要是根据表的索引访问类型，比如ref等反写sql下发），但是很多还不支持。

之前测试tpcc的时候加了如果没有用到索引，就把cond推下去

在get\_quick\_record\_count 中判断的：

#ifdef WITH\_SPIDER\_STORAGE\_ENGINE

if (tab->quick() == nullptr) {

// table scan spider pushdown condition here

assert(tab->table()->file != nullptr);

if (tab->table()->file->ht->db\_type == DB\_TYPE\_SPIDER) {

tab->table()->file->cond\_push(tab->join\_cond() ?

tab->join\_cond() : tab->join()->where\_cond, false);

}

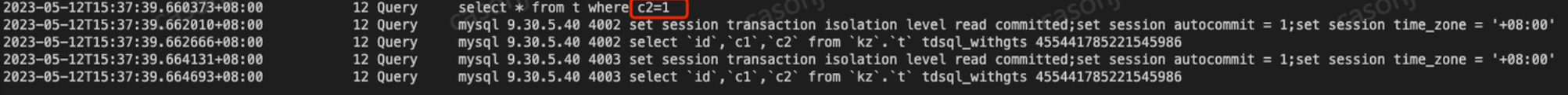
}

#endif

**问题：**

这个判断的地方有些sql没走到，导致有些场景没下推

这个没有过滤某些可能结果有问题的情况，比如时间字段？



* 支持普通条件的尽量下推
* 复用3.0的逻辑，排除特殊条件
* 如何扩展到join，优化后绑定到特定表上的条件（除了特殊情况）都可以下推？

封装并实现一个CondPushdownIterator

* 不支持join，可以像单分片一样直接下推原始client 输入的 sql
* 支持join, 必须走handler的 sql 拼接逻辑

考虑后续子树下推，采用sql拼接的方式生成下推的sql，关注where和having的区别。

limit+offset

3.0 支持了两种：

limit condition push down

limit offset condition pushdown

只是limit 没必要下推？

* limit + offset

PushdownLimitOffsetIterator

* order by + limit
* order by + limit + offset

封装实现OrderbyLimitPushdownIterator/OrderbyLimitoffsetPushdownIterator

* order by + group by

暂时不支持

distinct

* 只支持普通的distinct：

eg: select distinct c, x from t where xx?

spider支持必须带条件的下推：



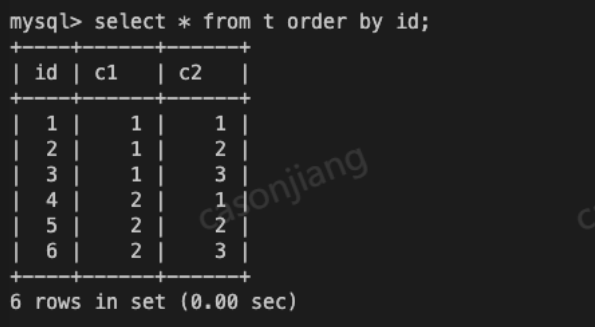
实现 DistinctPushdownIterator

agg

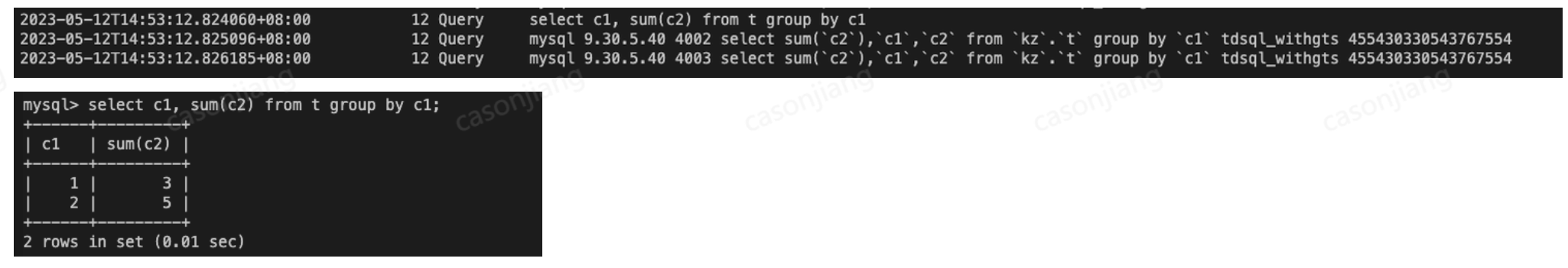
* sum/min/max

理论上直接下推就可以支持的

开启spider的direct\_aggrete结果错误：



sum：

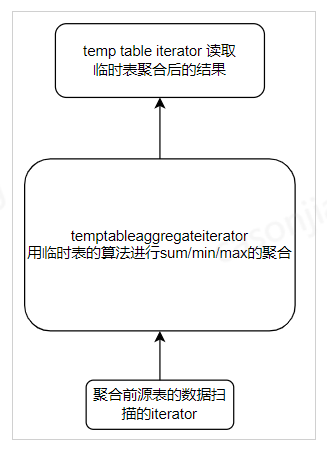


min和max类似

没法用spider的direct\_aggregate，实现SumMinMaxPushdownIterator。

* 基于临时表的聚合实现

在optiimizer前构造一个执行计划（和通过optimizer的一样）



temptable 聚合算法：

select c1, sum(c2) from t group by c1;

1、顺序读取源表的没行，然后通过c1(临时表会建个索引）查找，如果找不到就新写入一行到到临时表

2、找到则聚合逻辑（sum) 更新临时表的c2列

直到全部处理完源表，再按行返回临时表给客户

* count

CountPushdownIterator

从spider引擎返回的数据，count字段需要转换成sum逻辑计算

* avg

AvgPushdownIterator

将avg字段替换成sum字段+count字段, 然后spider引擎返回的数据再重新计算。

insert/update/delete

流程和查询差不多

ha\_spiderpart direct\_pushdown\_xxx 接口 → ha\_spider direct\_pushdown\_xxx

直接下发原始的sql到远端。

##### SQL改写

1、可以直接转发

where 条件下推

order by + limit

distinct

sum/min/max/count

2、等价转换的

avg → sum + count 下发到set

eg: select c1, avg（c2) from t group by c1;

下发

select c1, sum(c2), count(c2) from t group by c1;

##### 结果集合并

###### 简单合并

按照分区（set)串行处理

where 下推→ 直接转发

sum/min/max → 直接返回给上层计算器（复用sum,min,max的计算逻辑）

limit 顺序扫set→ 控制返回的rows

###### 再计算

* 重排序

order by + limit

复用上层filesort排序逻辑, 优化阶段不要选择索引排序

将OrderbyLimitIterator返回的行作为filesort的输入

* 重分组聚合

count → 把上层计算器的逻辑调整为sum

将CountPushdownIterator返回的行作为sum的输入

当然再计算也可以自己实现计算逻辑，个人倾向于调整Iterator复用mysql的。

#### 现状

目前单分片查询采用快速路径的方法，在prepare阶段，优化阶段各有快速路径检测，当发现符合单分片的特点，就可以直接走单分片查询直接转发的路径，主要由spider层进行Query的转发和结果的收集。

目前存在的问题：

1、单分片检测机制相对简单，只判断多个表采用相同的partition id，目前由于只支持hash分布，并且可以保证相同shardkey一定位于相同的数据节点，但如果支持其它分布方式，如range分布，就不一定保证相同shardkey位于相

同的数据节点。

1. 若存在不支持的函数或UDF等，就会导致不支持单分片转发。

### Condition下推

#### 背景

spider是为了解决sharing nothing数据集群的分布式访问问题，在spider节点上创建的spider表，通过partiton的方式可以映射到多个不同的数据库实例上，从而实现数据的水平分布，用户无需感知实际存储数据的数据库实例节点，简称数据节点DataNode，只需要在Spider节点上操作即可完成增删改查，DDL等一系列操作，简化了数据库的维护和使用。

Spider节点上保存有partition的相关信息，还可以通过访问远程DataNode得到Spider表的统计信息（是否足够精确？），因此在Spider节点上就可以进行查询优化，确定每个表的访问方式及JOIN order，但本身并不包括数据，当执行时需要远程访问DataNode上的数据，然后在spider节点上完成Join、Filter、Aggregate、Order、Distinct、Having、limit等。

上述这种执行方式显然有很大的网络开销，所有数据都需要从DataNode上通过网络读取，如果表的数据量很大时，网络开销会非常大，严重影响性能。而实际上，有很多查询，并不是所有操作都必须在spider上来执行，比如一个简单的where条件，只包含spider表的列，如下示例：

t1表是一个spider表：

select \* from t1 where t1.c1 < 100;

显然，如果将spider表t1的数据在远程先做Filter：t1.c1 < 100后，再通过网络读取到spider节点，那么就会减少大量c1>100的数据传输，从而提升执行效率和系统性能。

基于降低网络开销的考虑，所有可能减少网络传输量的操作都可以尽可能的pushdown到数据节点上先执行，然后再将结果返回到spide端。

#### 原理

为尽可能的减少spider节点与远程数据节点之间的通讯，将condition尽早下推到数据节点执行，尽早过滤掉不需要的数据，有可能会大大减少远程数据节点返回的数据量，从而提升整体性能。同时还可以将部分需要耗费CPU的

操作交由远程数据节点来执行，减少spider节点对CPU的消耗。

##### Condition pushdown的条件

并不是所有的filter条件都可以pushdown到datanode端，比如与实例相关的函数，如now()，自定义函数等，在TDSQL3.0中有更详细的不支持类型。另外filter条件中也不能包含其它spider表的列，除非此列是一个复制表的列，因为复制表的数据在每个节点上都存在。

filter条件的pushdown有点类似mysql原生的ICP，将condition下推到存储层。

目前不支持condition pushdown的算子

* udf用户自定义函数，因不确定用户行为
* now(), random, session variables,等不确定结果的function或变量
* group\_concat，不确定结果
* Blob/json要支持吗？

##### Condition pushdown的实现机制

在mysql中，对innodb表有类似icp的机制，对derived table也有类似将condition push的机制。但icp的实现相对简单，能下推的condtion有限，而对derived table的condition的支持相对更适合于spider表的condition pushdown。因此选用与derived table的condition的pushdown的机制作为下推方案。

在原有类Condition\_pushdown上派生新的适用于spider表的类Spider\_condition\_pushdown，实现spider表特有的几个接口：

make\_cond\_for\_spider\_table # 用于检测condition是否可以被pushdown，如果可以全部或部分pushdown，提取可以pushdown的部分，attach到spider的handler上，如有不能push的条件，保留原位。

check\_cond\_for\_spider\_tables # 检测condition是否可以被pushdown，主要用于为fastpath的检测是否可以走fastpath，因为fastpath原来不检测是否可以安全的下推，比如包含udf。

attach\_cond\_to\_spider。# 将可以下推的condition attach到spider表的handler层

extract\_cond\_for\_table。# 提取可以下推的condition，部分condtion需要重建，如and/or。

原计划在prepare阶段实现spider表的condition pushdown的检测和实现，但存在由于提前将部分condition下推，导致在condition优化阶段时，无法进行等值传递和条件推导等优化，如

对条件t1.c1=t2.c1 and t1.c1=10.

如果将t1.c1=10在prepare阶段提前下推，那么在优化阶段就无法推导出条件t2.c1=10，从而会引起join是否可以走fastpath的误解，同时也可能会导致t2表选择错误的access path。

因此spider表的condition pushdown的检测和实现，应该放到condition优化之后，不应影响到优化阶段的fastpath或access path的选择。

##### Condition pushdown的开启参数

tdsql\_pushdown\_condition # on 开启， off 关闭。默认为off。

#### 现状

目前Condition下推只支持在prepare阶段的检测，若符合condition下推的条件，直接将Condition下推到spider的handler层，同时在query中移除此condition。

目前存在的问题：

1、对于优化过程中新产生的condition条件无法下推，如在condition优化过程中推导出来的条件，如t1.c1=10 and t1.c1 = t2.c2. 可以推导出t2.c2=10；

2、由于在prepare阶段将可以下推的condition移除，因此可能会影响condition优化，进而影响到执行计划的选择，如表的扫描方式、JOIN方式等。

## 并行框架现状

并行查询的基本原理在于使用多个不同的worker分别查询数据的某个分片，最后由主线程将所有worker的执行结果合并起来返回给客户端。因此基于spider的分布式系统是一个理想的并行执行场景，每个数据节点可以在自己的数据分片上执行查询，多个数据节点可以并行执行，然后将结果返回给spider节点，由spider节点汇总，然后将结果返回给客户端。

并行框架主要由以下几部分构成：

* 并行计划/下推计划选择
* 并行计划拆分，即主计划与子计划的拆分
* Worker生成及实例化子计划
* 线程间通讯建立或远程实例通讯建立
* 事务一致性同步
* 全局代理锁机制
* 合并worker执行结果
* 执行状态汇总及异常处理
* explain analyze的支持

### 并行计划/下推计划选择

本模块主要用于判断当前查询是否可以产生并行计划，由于查询的某些算子如果由worker并行执行可能会导致产生错误的结果，因此并不是所有操作都可以并行，另外并行的目的是为了提升性能，但并不保证并行一定能提升性能，反而有些场景下并行可能会导致性能变差。此外，产生并行计划还要考虑资源的问题，如在单机上产生过多的worker线程会导致CPU资源的浪费，多节点上可能会导致连接数溢出等。当前的并行计划选择模块还比较简单，主要检测不允许并行的限制条件，也只能产生单表查询的并行计划，并且不能做查询的部分并行。只要查询的任何一个算子不能并行，整个查询都不能并行。

#### 现状

并行计划的限制条件有：

* 未开启并行参数
* 不支持包含增删改中的select查询，只支持selelct查询
* 不支持相关子查询
* 不支持被外层引用的相关子查询
* 不支持由子查询转换的derived table
* 不支持zero结果集的查询
* 不包含select\_count的计划，当其设置后，表不允许被优化掉，在执行阶段通过索引来执行count
* 不支持多表或无表，只支持单表
* 不支持包含window function的查询
* 不支持IN子查询
* 不支持非用户表
* 不支持临时表
* 不支持一些特殊的查询访问方式，如const，index merge，full text，dynamic range scan，range index merge， range fulltext，ROR intersect，ROR union
* 不支持unsafe的item，unsafe指会产生非确定结果集的item，以及用户自定义函数/过程等
* 无法确定并行度的查询，如数据量太小，无法分片等
* 不支持并行扫描的存储引擎
* 超出并行资源限制

目前存在的问题：

* 限制条件强，只有很少⼀部分简单查询可以并行，如要求单表
* 不支持部分并行，如表扫描并行，但后续AGG等算子不并行的情况不支持
* 没有单独针对spider的特点做⼀些限制或优化，如spider表的单分片也可以纳入并行框架；
* 不支持多表
* 不支持复杂查询的多个并行子计划
* 不支持并行子计划的描述，目前子计划只能是原始计划

#### TODO

目前没有针对spider表的特点进行并行计划的生成，对于spider表来说，它天然具备并行的条件，任何spider表都可以支持并行，只是根据裁剪后的分片个数来决定其并行度。因此后续此部分的工作包含：

* 针对spider表的特性调整并行检测和确定并行度
* 并行计划描述的改造，为后续生成部分算子并行计划生成做准备
* 部分算子并行计划的生成。如只支持表的并行扫描，其它算子仍由主线程串行执行
* 单分片并行计划的生成
* 多表相同分片的并行计划的生成
* 根据更多分片方式做并行计划，如包含复制表

### 并行计划拆分

根据并行选择模块输出结果，通常是并行描述符，从原始计划中将子计划拆分出来，原始计划中需要产生新的**占位算子**，替换被拆分出去的子计划，同时也用于关联子计划，将来从执行子计划的worker接收合并执行结果。

#### 现状

目前的计划拆分包括：

* 新的query block的创建，及JOIN对象构建
* 表的添加
* 投影列生成
* GROUP/ORDER等复杂算子的创建
* 输出result table的设置
* 原计划中占位算子collector的创建，并且子计划输出result table的关联

说明：在ParallelPlan::GeneratePartialPlan构建子计划，这里需要在原计划中增加占位算子Collector（临时表），并且子计划输出result table的关联

* 创建并行计划的AccessPath
* 其它设置，如ref引用的重新设置

说明：Plan主要涉及：

planner.cc

#### TODO

针对spider的改进建议：

* 对于spider下推计划，可能不需要真正拆分子计划，而是直接生成下推SQL即可
* 目前由计划反向逆推SQL的实现相对还比较简单，在server层实现。后续完善可以有两种方式：

一是在现有方案中完善

二是借用在spider的handler中的已有实现

TODO：

由于spider节点与数据节点之间最终是通过SQL来交互，因此，拆分并行计划时可以直接生成SQL子计划，无需再clone query tree

### Worker生成及实例化子计划

worker通常有worker线程来实例化，并执行，返回结果。当worker线程创建后，首先要实例化当前worker的子计划，即从并行计划拆分出来的子计划clone到当前worker，然后生成iterator迭代执行。

#### 现状

对于spider来说，目前采用虚拟worker，并不需要产生实际的worker线程，而仍然由主线程对多个远程数据节点分别生成多个worker对象，然后实例化worker子计划，与物理worker不同的是，此时需要再将子计划反向逆推向SQL，通过session连接，发送SQL到远程数据节点。发送过程可以是异步的，目前由于采用MySQL客户端协议，可以异步接收远程的结果到本地缓存，然后再由主线程合并结果集。

说明：Worker主要涉及：

executor.cc

worker.cc

mysqlclient\_worker.cc

spider\_conn\_worker.cc

#### TODO

针对spider的改进建议：

* worker的计划仍是物理计划，后续才转为SQL逻辑计划，可以在拆分并行计划阶段直接生成SQL逻辑计划；
* 依赖于MySQL客户端协议本身的异步，若是某个数据节点接收过程中超时，不能异步接收其它节点数据；
* 与远程数据节点间采用单通道通讯，命令、数据及状态在相同通道中传输，错误处理相对复杂。

TODO：

如果并行计划已经拆分为SQL子计划，那么此时worker相对比较简单。对于spider的worker来说，可以认为只是一个执行远程命令的代理worker，worker本身并不真正执行什么，只是用来获取远程连接，设置session相关变量，同步锁和事务，发送SQL，接收异常和返回结果等。

### 线程间通讯建立或远程实例通讯建立

#### 现状

目前本机主线程与worker线程间通讯采用share memory方式，对于与远程节点的通讯采用mysql客户端协议，通过指定spider表的分片ID获取session连接，然后使用其做为后续与远程数据节点的通讯通道。也就是说，用于spid

er表并行的虚拟worker相当于远程数据节点的⼀个客户端，虚拟worker向远程数据节点发送从并行子计划反向生成的SQL，然后接收返回的结果集到本地缓存。

说明：Collector与Worker之间采用共享内存/TCP通信，Worker与远程存储节点采用MySQL协议。

#### TODO

针对spider的改进建议：

* 主线程的session变量可能会影响到查询的执行，因此哪些session变量应该同步到与worker的session连接中？
* 用户权限检查是否满足要求？session的用户现在应该不会切换到客户端的用户。
* 异常处理机制是否完善？

TODO：远程Session连接

与数据节点之间的session连接是主线程与数据节点之间的通讯通道，由于session的重用，每次获取session后需要根据主线程重设session变量。另外如有mdl锁，或者事务的要求，也必须先同步后才能用于sql执行。

### 事务一致性同步

目前针对spider表的事务一致性功能尚未支持，因为每个数据节点都是⼀个独立的mysql实例，在同一个查询中，必须保证每个数据节点执行的子计划所使用的事务视图是一致的，同时满足相同的事务隔离级别，至少要满足read committed。

#### 现状

在主线程开始执行阶段，获取全局⼀致性快照，然后通过session连接发送到远程数据节点，在数据节点执行子计划时使用此全局⼀致性快照，解决事务可见性问题。

除此之外，还要同步session变量，如事务隔离级别等。

#### TODO

当前在执行SQL时，未使用spider提供的接口spider\_db\_query，不包含全局一致性事务。在spider提供的接口中已经包含获取本地全局一致性视图，然后同步到远程，但在使用中还存在一些问题，需要进一步调试。

### 全局代理锁机制

#### 现状

由于每个worker在执行子计划过程中，可能会加各种mdl锁，对于同一个实例的不同线程来说，本来相互之间加锁是不相关的，但对同⼀个查询的主线程和worker线程来说，如果各自独立加锁，可能就会导致死锁。比如：主线程

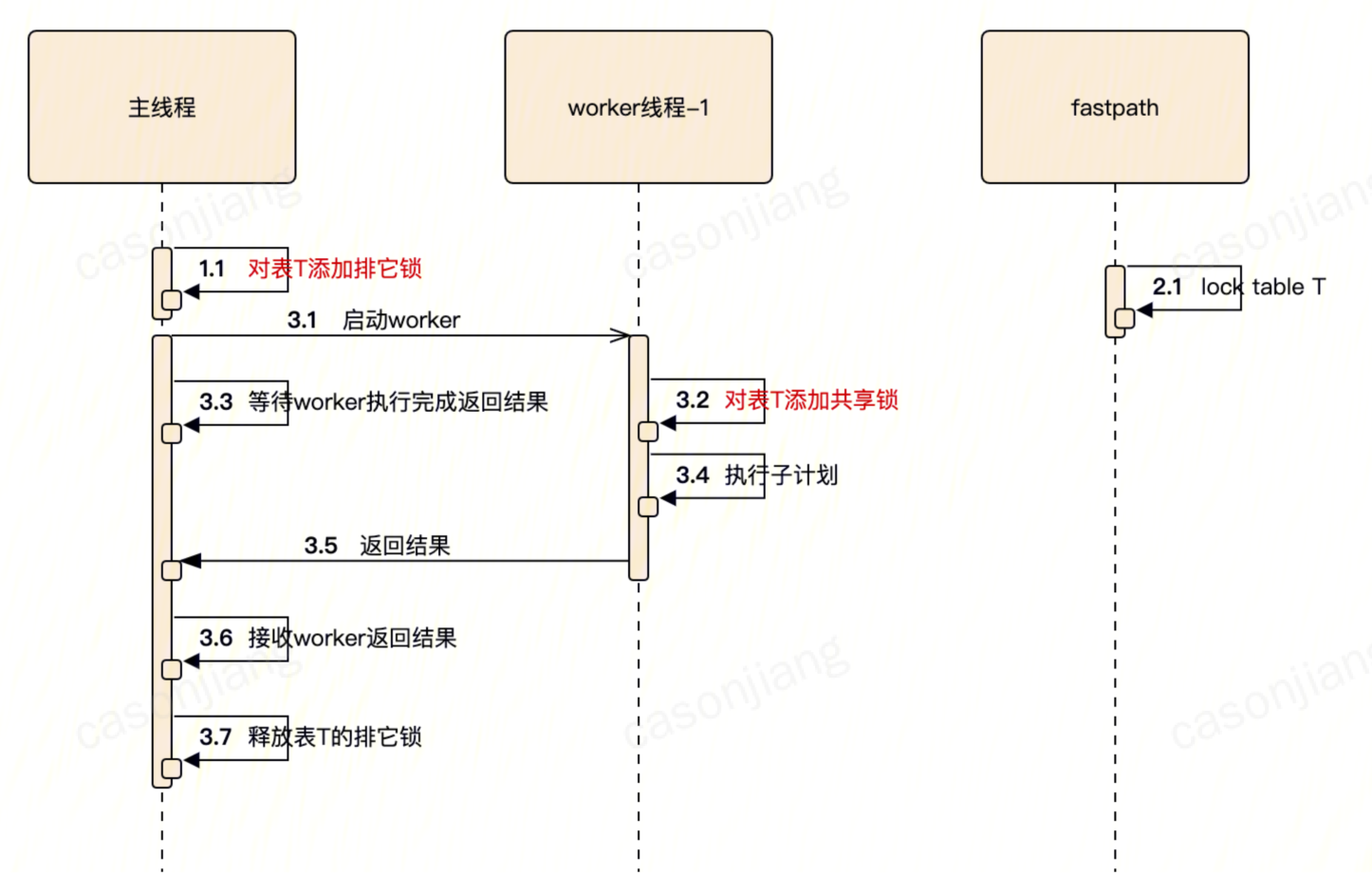
已经对某个表加了排它锁，如果没有并行worker的话，它的后续执行过程中可能会对这个表申请添加共享锁，因为是同一个线程，之前已经持有排它锁，申请共享锁的操作自然就自动通过；但如果有并行worker存在，在并行worker的执行过程中如果对这个表申请添加共享锁，因为不在同⼀个线程，所以就会等待主线程释放排它锁，但主线程又在等待worker执⾏完成返回数据，从而导致死锁。

死锁的场景如下所示：

lock table T write;

select \* from T; # 并⾏执⾏

加锁过程如下图所示：



因此，在本机的上的并行worker，不能简单的只由并行worker自己加锁，目前采用的方式是由主线程代理的机制来避免这种死锁的出现。简单来说：就是并行worker在加锁时，如果发现有主线程，而向主线程申请代理加锁。

对于spider表的数据节点间的并行，则不存在此问题，因为主线程与执行子计划的线程位于不同的实例，封锁对象物理上也不相同，因此不会出现类似上述的死锁问题。

#### TODO

但是对于上述主线程的lock table，应该认为是⼀个session级变量，在建立session的同时，应将其传递到远程数据节点。

### 合并Worker执行结果

#### 现状

合并worker结果集，分为两种：简单合并和有序合并。所谓简单合并，就是把每个worker返回的结果放在一起就可以了，没有额外限制。而有序合并要求就高一点，用于worker返回的结果本身是有序的，而主线程同时又要求整体有序，因此需要在合并结果集时采用merge sort的方式。

目前已经支持持merge sort的有序合并。

说明：结果集合并的操作主要涉及：

row\_exchange.cc

row\_channel.cc

message\_queue.cc

row\_channel\_tcp.cc

row\_segment.cc

merge\_sort.cc

#### TODO

合并Worker执行结果及异常处理：

目前采用轮询的方式依次读取每个worker的数据，只有当前worker读失败后，才会读取下一个worker的数据，存在以下问题：

* 当前在读的worker执行缓慢或网络问题，导致不能及时收到数据，即使有可用的其它worker连接，也无法及时读取。
* 当有远程数据节点执行异常时无法及时发现，主要原因在于数据与命令状态采用相同连接导致，后续看如何快速的传递异常状态给主线程？建议数据与命令/状态的讯道分离。

### 执行状态汇总及异常处理

#### 现状

执行状态汇总指每个查询结束后，会返回一些状态，如examined rows，这些状态可以协助分析执行计划的优劣。在并行执行的情况下，需要汇总每个worker的执行状态。

另外，由于多个worker并行执行，每个worker都可能出现异常，当任何⼀个worker出现异常后，其它worker通常情况下就没有必要再继续执行，必须由主线程通知其它worker终止执行。

#### TODO

合并Worker执行结果及异常处理：

目前采用轮询的方式依次读取每个worker的数据，只有当前worker读失败后，才会读取下一个worker的数据，存在以下问题：

* 当前在读的worker执行缓慢或网络问题，导致不能及时收到数据，即使有可用的其它worker连接，也无法及时读取。
* 当有远程数据节点执行异常时无法及时发现，主要原因在于数据与命令状态采用相同连接导致，后续看如何快速的传递异常状态给主线程？建议数据与命令/状态的讯道分离。

### Explain analyze的支持

explain analyze主要用于性能的分析，可以比较精准的展现执行过程中每个阶段的执行时间，协助分析和定位性能瓶颈。在并行框架中，explain analyze是一个非常有用的功能，还可以定位类似数据倾斜、通讯开销等导致的并行性能问题。

#### 现状

目前worker的explain analyze功能还不支持。

#### TODO

支持explain analyze，有利于性能分析及调优。