Doris PipeLine的设计文档

1.背景

当前Doris的执行引擎是volcano的pull 模型，在单机多核的场景下存在下面的一些问题：

- 无法充分利用多核计算能力，提升查询性能，**多数场景下进行性能调优时需要手动设置并行度**，在生产环境中几乎很难进行设定。
- 单机查询的每个instance对应线程池的一个线程，这会带来额外的两个问题。
    1. 线程池一旦打满。**Doris的查询引擎会进入假性死锁**，对后续的查询无法响应。**同时有一定概率进入逻辑死锁**的情况：比如所有的线程都在执行一个instance的probe任务。
    2. 阻塞的算子会占用线程资源，**而阻塞的线程资源无法让渡给能够调度的instance**，整体资源利用率上不去。
- 阻塞算子依赖操作系统的线程调度机制，**线程切换开销较大（尤其在系统混布的场景中）**
- CPU的资源管理困难，**很难做到更细粒度的资源管理，多查询的混合并发做到合理的资源调度。**
    1. 大查询生成海量instance之后，线程池被打满。小查询几乎得不到调度的机会，**导致混合负载下，小查询的时延较高。**
    2. 在混合部署的Doris集群之间，用户间的cpu资源无法得到好的隔离，目前几乎是处于无管控状态。

2.How To Do

2.1 需要解决的核心问题

阻塞操作的异步化

阻塞算子会带来两个问题：

- **线程切换**： 带来额外的上下文切换开销
- **线程占用**： 阻塞的线程也会挤占单进程的线程资源

阻塞操作

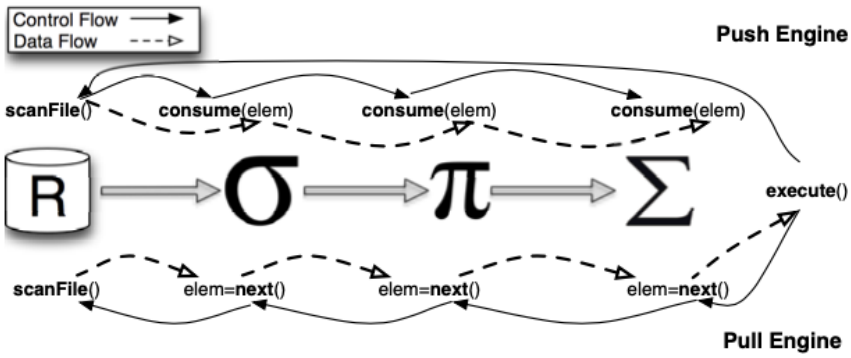
- **物理阻塞**: 包含磁盘I/O，网络I/O，锁等导致线程阻塞的操作
- **逻辑阻塞**：Sort, HashJoin，HashAgg等需要全量物化的算子

CPU的资源管理

混合负载场景中，出现CPU争夺的情况。CPU密集的大查询长时间挤占CPU资源，小查询得到CPU调度困难。

2.2 Pipeline的执行模型

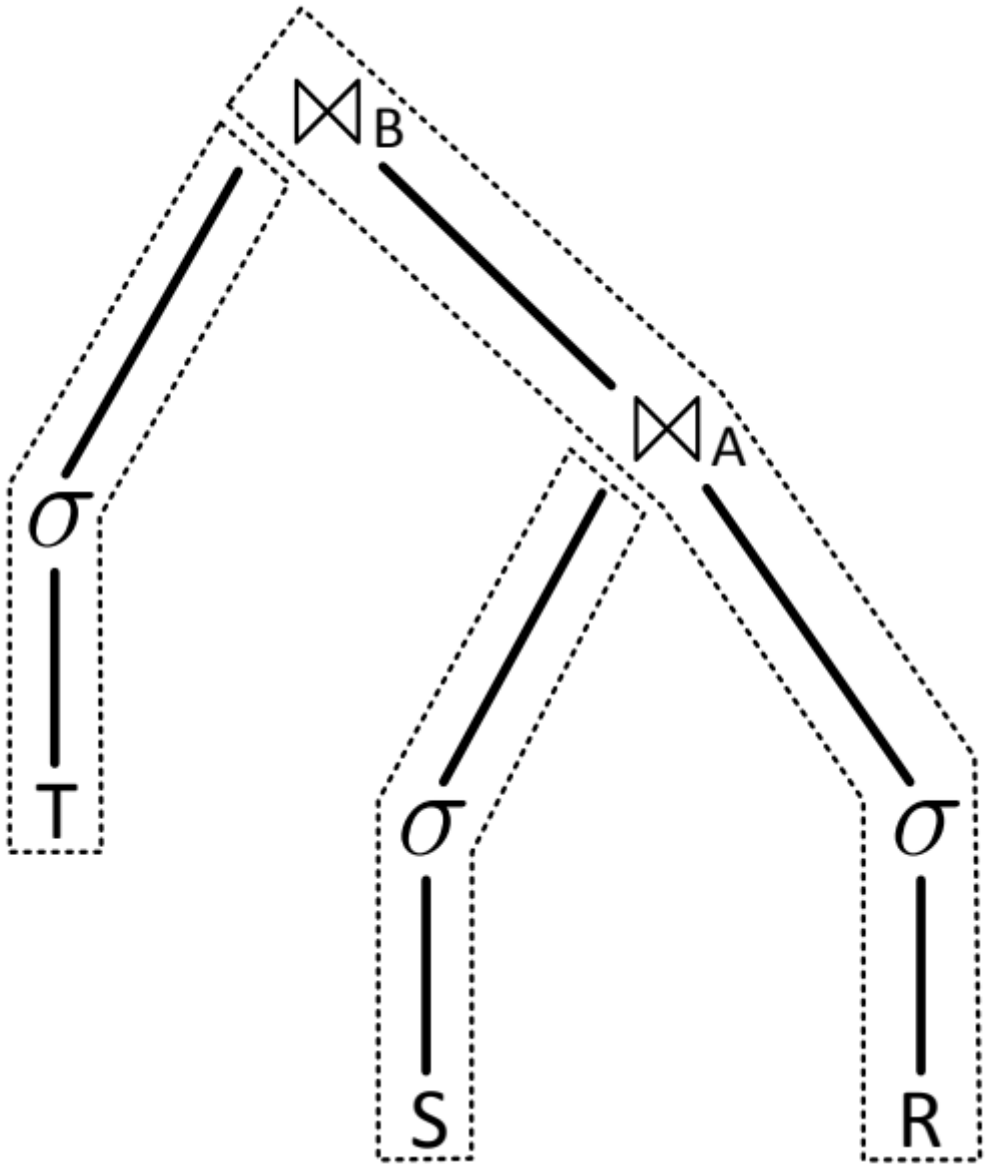
什么是PipeLine执行模型




```
SELECT SUM(R.B)
FROM R
WHERE R.A < 10
```

由上图可以与传统的火山模型模型不同，在Pipeline模型中，是由数据来驱动控制流执行的。具体来说，不同的算子之间数据是通过push数据的方式进行交互。**各个查询执行过程之中的算子被合并和拆分成不同的pipeline task，整个执行引擎使用固定数量的线程池，各个pipeline被调度的逻辑是由数据驱动的。**

举个例子：

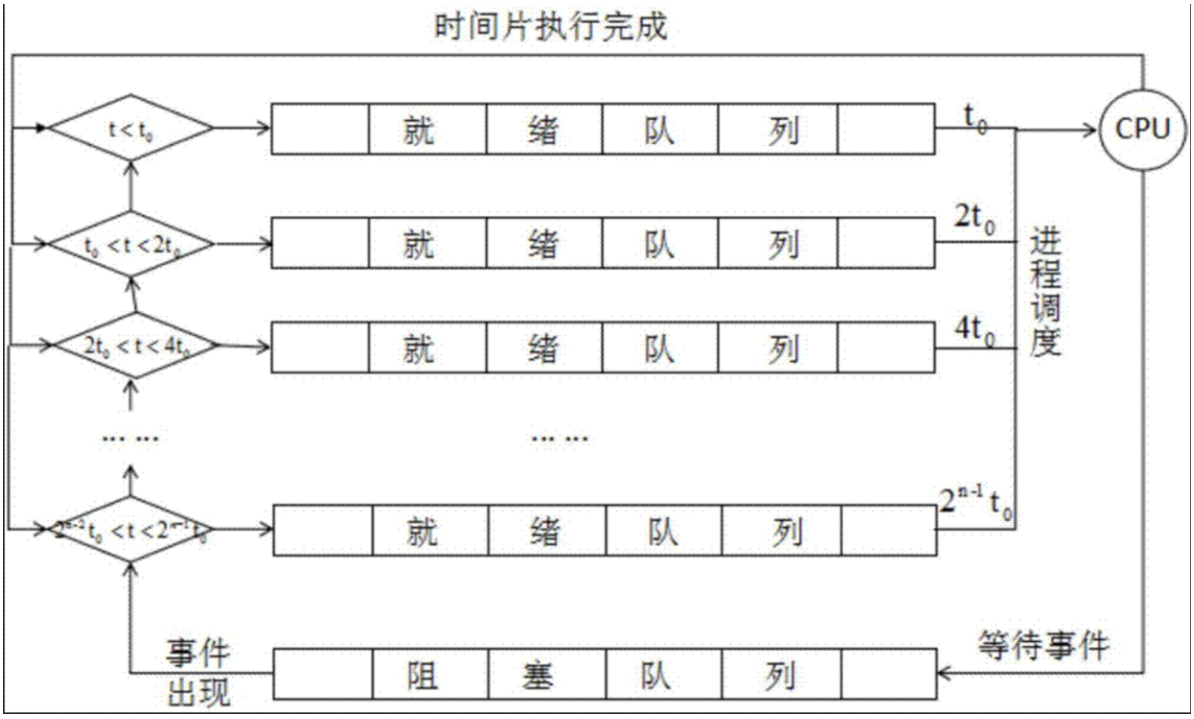


R join S join T，这个执行计划被拆分成3个pipeline，每个PipeLine可以由不同的线程进行调度。Pipeline拆分点便是Join build的阻塞逻辑，由于join算子的特殊，需要Probe阶段需要等待Build 数据的Ready才能进行执行。**每个pipeline是否占有线程资源进行调度，取决于前置的数据是否Ready，所以每个Pipeline的调度核心点是数据进行驱动的，未ready的Pipeline需要主动释放线程资源。**

综上所述，PipeLine的核心思路可以类比**操作系统的分时调度系统**，核心点如下：

- CPU核心是有限的，怎么样让远超CPU数目的多进程在操作性上运行。
- 进程之间是通过怎么样的规则占据CPU的时间片的，什么时候需要出让CPU资源给其他的进程。

这里**CPU资源 = 执行引擎的线程资源**， **进程之间的调度策略 = Pipeline的调度策略**， **进程的优先级 = 查询的资源管理**， 把这些逻辑一一对应起来，就能更好的理解PipeLine执行引擎的精髓了。



2.3 Pipeline怎么解决问题的

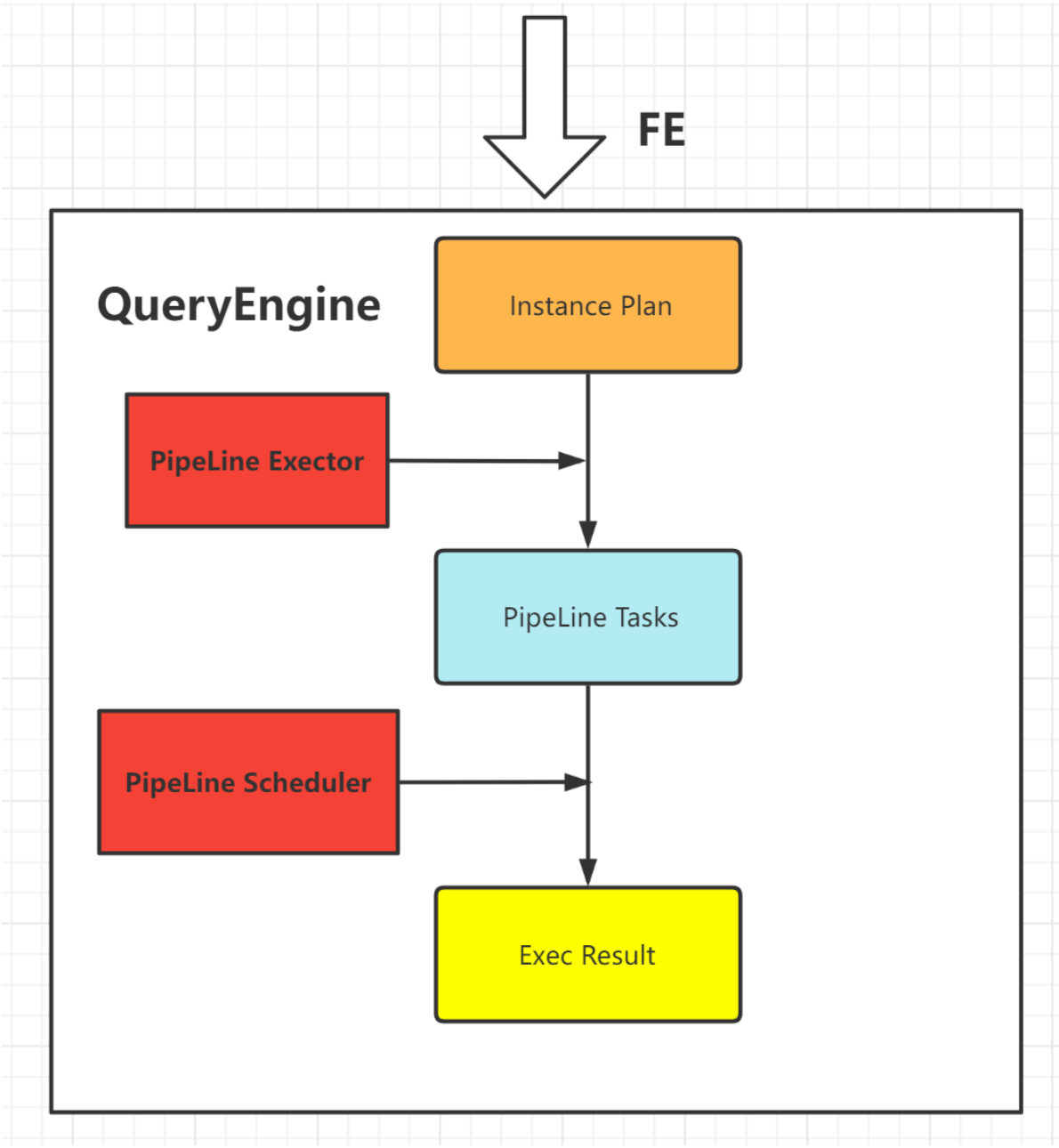
- **阻塞操作的Pipeline拆解**
  阻塞的逻辑被拆分为不同的Pipeline进行调度，而阻塞逻辑之间的pipeline通过数据进行驱动。**解决了pull模型上，阻塞算子占据线程资源，同时线程切换带来的额外开销。**同时不同的Pipeline之间能够实现并发计算，相同Pipeline之间也能实现并发计算。
- **PipeLine的资源管理**
  pipeline调度之后，线程资源 -> pipeline -> 查询。可以通过不同查询的资源占有量，公平的调度不同查询。让混合负载的查询能合理的进行线程资源的共享，过分占用线程资源的pipeline需要主动释放线程资源。

3. 如何对Doris进行PipeLine改造：

Doris上的核心改动点：

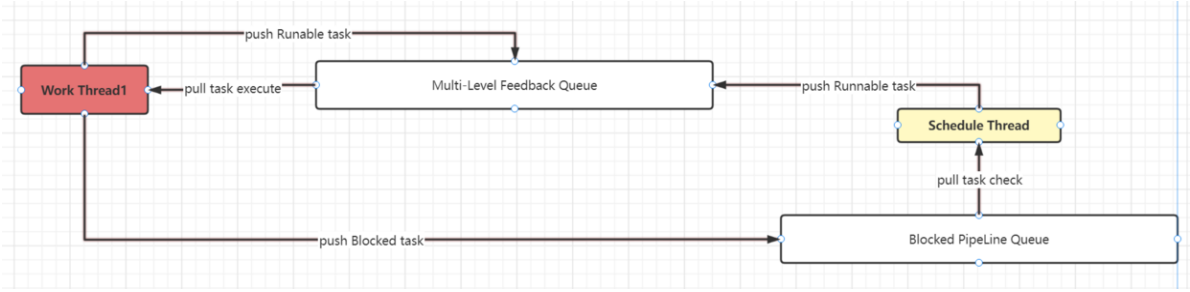
1阶段的改造计划：

整体执行引擎的框架：



这里在原有的QueryEngine基础上添加了两个新的结构模块：

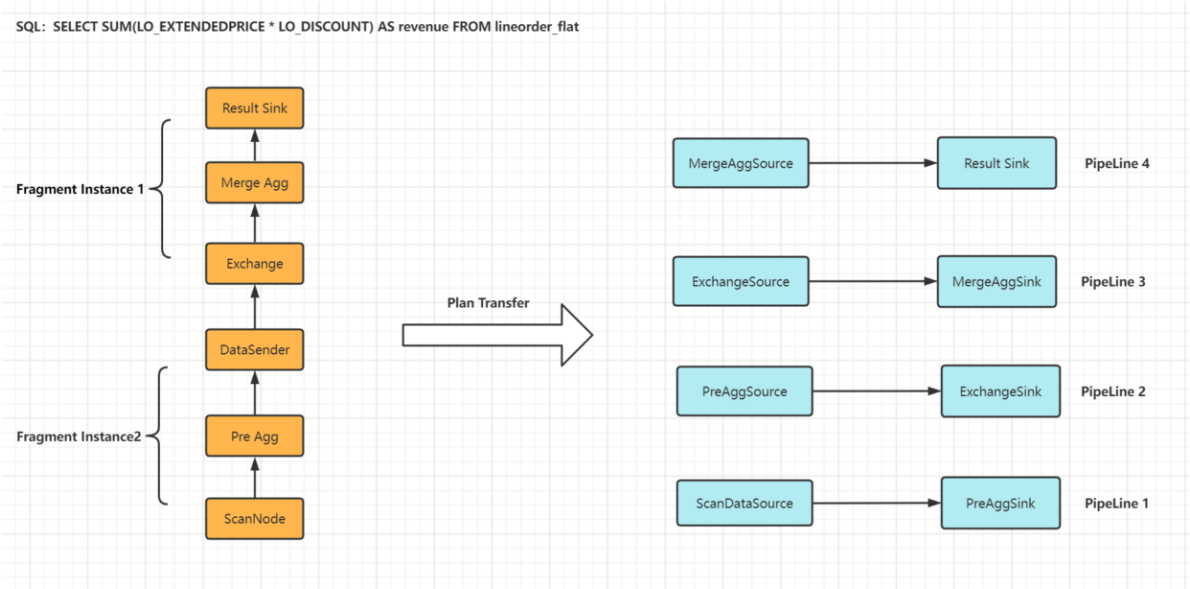
- **Pipeline的调度模块**： 负责PipeLine Tasks的调度和执行



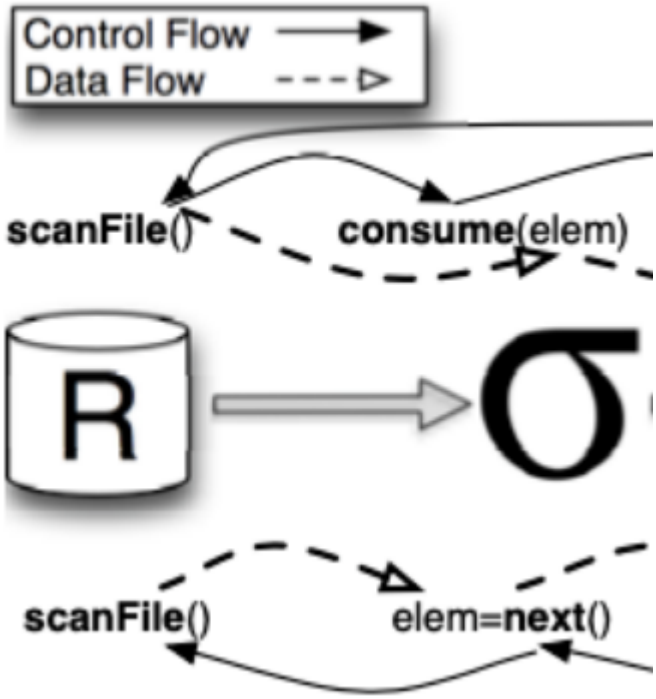
1. 主类: BE.main 调用 **BlockedTaskScheduler**的构造函数
2. 启动**Schedule Thread**，end less for loop 轮询 Blocked Pipeline Queue，进行状态初始化，查询的工作

3. 启动多个**Work Thread**， end less for loop 轮询 自己的任务队列和其他核的任务队列，寻找**Runnable**的Pipeline Task
    - 出让时间片，将Pipeline Task放回任务队列
    - 陷入阻塞，将Pipeline Task放回Blocked Pipeline Queue
    - 关闭已经执行完成的Pipeline Task，调用close释放资源

- **Pipeline的执行模块**： 负责PipeLine Task和原有的QueryPlan的翻译工作，执行计划的PipeLine化和执行逻辑重构



1. 阻塞算子的Pipeline拆解。**把原有的Fragment Instance的架构拆解为多个PipeLine，后续调度模块调度的基本单位便是PipeLineTask。**设计的阻塞算子有：
    - Join Build
    - Sort
    - Agg
    - Scan
    - Exchange
2. 算子的push化改造。（这部分待定，依据实现进度来看是否实现。笔者评估改动量可控）
   核心的点是对于执行Operator的接口进行重构，由原先的pull模式改成push模式。



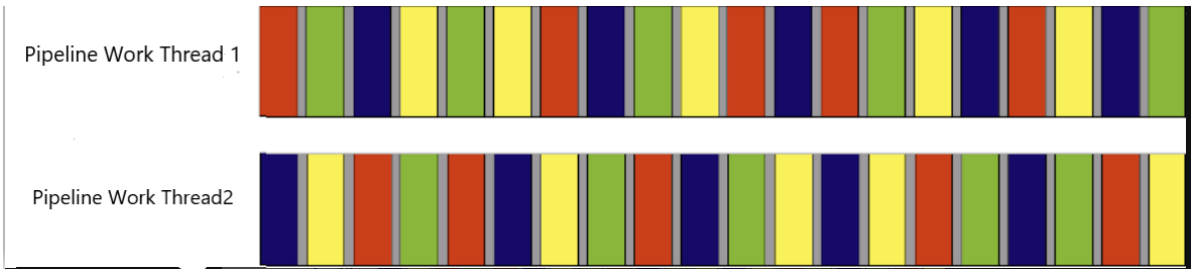
这部分改造只设计Pipeline内部的Operator之间的数据交互方式：**核心优点是能够实现Pipeline Task在任意Operator之间实现中断，出让线程时间片。**

2阶段的改造计划 (本次的改造不涉及)：

- 单pipeline之间的并行化
- 基于用户级别的CPU隔离能力

详细设计

Pipeline的调度模块：



如上图所示，Pipeline上启动固定的Work Thread数目，通常数目为CPU的核数，来执行不同的PipeLineTask。不同的颜色代表不同查询的Pipetask，各个PipeTask可以任意在任何执行线程上进行调度，充分利用多核CPU的并行能力。

1. 任务队列

- 阻塞队列：前置数据，或执行条件没有准备好的Pipeline Task都放置在该队列中。由轮询线程定期轮询，检查是否进入Ready状态，如果Ready则进入调度队列进行调度
- 调度队列：绑定**执行线程**，该队列上的Pipeline Task都是由轮询队列放入的，代表可以被执行的PipelineTask的队列。这是一个多级反馈队列，详细设计思路参考后文

2. 轮询线程

全局**唯一**的线程，轮询各个查询之间的PipeLine Task是否就绪。

- 轮询阻塞队列中的任务，Ready之后 轮询线程将任务 **阻塞队列 -> 调度队列**
- 任务中记录了上一次被调度的线程id，**优先进入该线程id的调度队列**

Code:

```
   while (!_shutdown.load()) {
        {
        auto iter = local_blocked_tasks.begin();
        DateTimeValue now = DateTimeValue::local_time();
        while (iter != local_blocked_tasks.end()) {
            auto* task = *iter;
            auto state = task->get_state();
            if (state == PENDING_FINISH || task->fragment_context()-
>is_canceled()) {
                // should cancel or should finish
                if (task->is_pending_finish()) {
                    iter++;
                } else {
                    _make_task_run(local_blocked_tasks, iter, ready_tasks);
                }
            } else if (task->query_fragments_context()->is_timeout(now)) {
                LOG(WARNING) << "Timeout, query_id="
                             << print_id(task->query_fragments_context()-
>query_id)
                             << ", instance_id="
                             << print_id(task->fragment_context()-
>get_fragment_id());

                task->fragment_context()-
>cancel(PPlanFragmentCancelReason::TIMEOUT);

                if (task->is_pending_finish()) {
                    iter++;
                } else {
                    _make_task_run(local_blocked_tasks, iter, ready_tasks);
                }
            } else if (state == BLOCKED) {
                if (!task->is_blocking()) {
                    task->set_state(RUNNABLE);
                    _make_task_run(local_blocked_tasks, iter, ready_tasks);
                } else {
                    iter++;
                }
            } else {
                // TODO: DCHECK the state
                _make_task_run(local_blocked_tasks, iter, ready_tasks);
            }
        }
```

3. 执行线程

执行引擎启动固定的执行线程进行查询任务的执行，**数目可以由用户配置，默认为CPU的核数。**

- 获取该执行线程本地的调度队列的PipeLine Task进行执行，如果线程绑定的调度队列没有对应的Task，则进行Work Steal，从其他线程的调度队列中拉取Task
- 记录该PipeLine Task的执行时间和处理的Block数目，**超过固定时间片或一定数据量后放回调度队列**，以免大查询饿死其他查询
- PipeLine task执行过程中陷入阻塞，**则将改任务放回阻塞队列之中，交付给轮询线程来后续处理。**

Code:

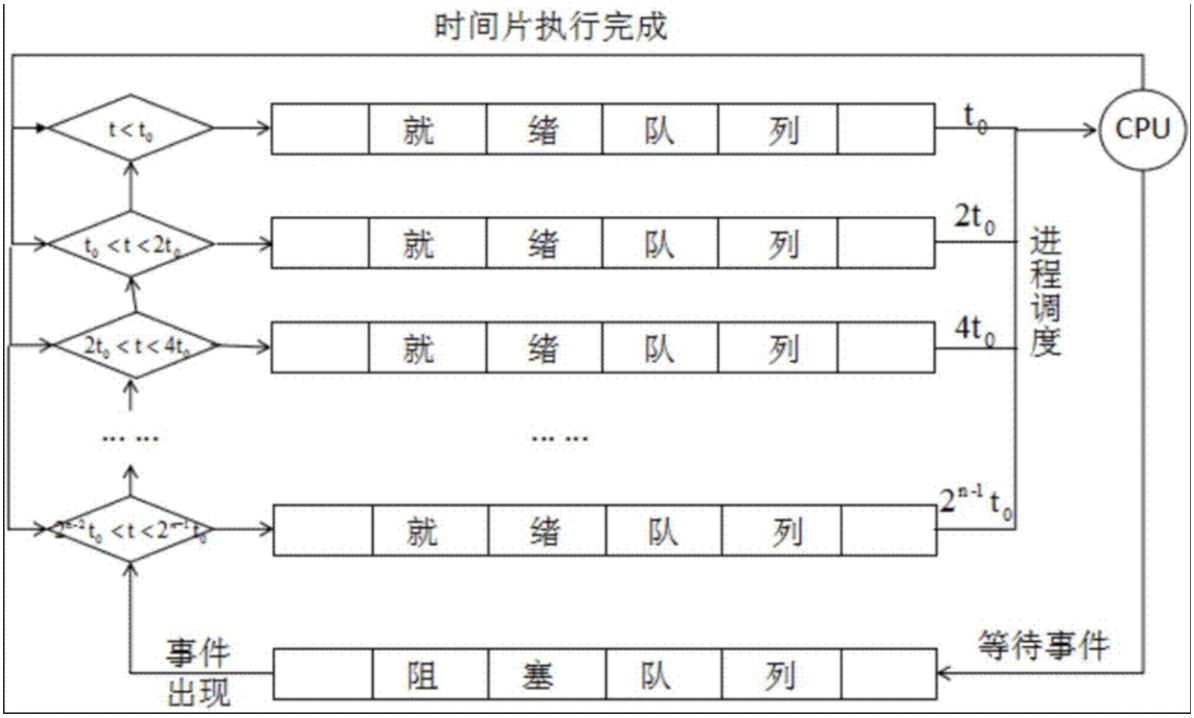
```
while (*marker) {
       // 执行队列里获取PipeLine Task
        auto task = queue->try_take(index);
        if (!task) {
            task = queue->steal_take(index);  // work steal
            if (!task) {
                // TODO: The take is a stock method, rethink the logic
                task = queue->take(index);
                if (!task) {
                    continue;
                }
            }
        }

        // 实际执行对应的task，绑定执行的线程的核心
        auto status = task->execute(&eos); //
        task->set_previous_core_id(index);

        auto pipeline_state = task->get_state();
        switch (pipeline_state) {
        case BLOCKED:
            _blocked_task_scheduler->add_blocked_task(task); // 放置回阻塞队列
            break;
        case RUNNABLE:
            queue->push_back(task, index); // 放置回调度队列
            break;
        default:
            DCHECK(false);
            break;
        }
    }
    LOG(INFO) << "Stop TaskScheduler worker " << index;
```

Pipeline的多级反馈队列：

多级反馈队列：**Multi-Level Feedback Queue**
应用于BSD UNIX系统，Windows NT和其后的Windows系统操作系统的进程调度，由图灵奖获得者Corbato提出。



基于多级反馈队列适配的PipeLine调度规则：

- 新加入的Query的Pipeline任务放在最高优先级队列，level 1
- 每个Query Id在对应优先级队列有一个获取task配额，使用完之后，**该Query相关的Pipeline Task自动下沉到下一级优先级队列**
- **每个优先级队列能固定分配到调度的时间片，避免大查询饿死的问题** (原算法是定期flush到level 1上，也可以考虑这种实现)

```
level 1 50%的cpu时间片
level 2 25%的cpu时间片
leve 3  12%的cpu时间片
.......
```

- 对应level 队列中不存在任务时，**执行线程自动获取下一级优先级队列的PipeTask进行执行**

Code:

```
// Each thread have private muti level queue
class WorkTaskQueue {
public:
    explicit WorkTaskQueue() : _closed(false) {
        double factor = 1;
        for (int i = SUB_QUEUE_LEVEL - 1; i >= 0; --i) {
            _sub_queues[i].set_factor_for_normal(factor);
            factor *= LEVEL_QUEUE_TIME_FACTOR;
        }
    } // 初始化多级反馈队列

   // 提交任务
   void push(PipelineTask* task) {
        size_t level = _compute_level(task);  // 计算任务应该处于的队列
        std::unique_lock<std::mutex> lock(_work_size_mutex);
        _sub_queues[level].push_back(task);
        _total_task_size++;
```

```
        _wait_task.notify_one();
    }

   // 获取任务
   PipelineTask* try_take_unprotected() {
        if (_total_task_size == 0 || _closed) {
            return nullptr;
        }
        double min_consume_time = _sub_queues[0].total_consume_time();
        int idx = 0;
        for (int i = 1; i < SUB_QUEUE_LEVEL; ++i) {
            if (!_sub_queues[i].empty()) {
                double consume_time = _sub_queues[i].total_consume_time(); // 每
个反馈队列有个自己的耗时情况
                if (idx == -1 || consume_time < min_consume_time) {
                    idx = i;
                    min_consume_time = consume_time;
                }
            }
        }
        auto task = _sub_queues[idx].try_take();
        if (task) {
            _total_task_size--;
        }
        return task;
    }
```

PipelineTask的状态转换：

PipeLine的**执行状态定义**：

```
enum PipelineTaskState : uint8_t {
    NOT_READY = 0, // do not prepare
    BLOCKED = 1,   // have some dependencies not finished or some conditions not
met
    BLOCKED_FOR_DEPENDENCY = 2,
    BLOCKED_FOR_SOURCE = 3,
    BLOCKED_FOR_SINK = 4,
    RUNNABLE = 5, // can execute
    PENDING_FINISH = 6, // compute task is over, but still hold resource. like
some scan and sink task
    FINISHED = 7,
    CANCELED = 8
};
```

PipelineTaskState 是被调度的PipeLineTask的状态

- **NOT_READY**：PipeLine的task没有调用**prepare**函数，没有进行准备执行的阶段
- **BLOCKED**：PipeLine的task进行阻塞状态，等待轮询线程的Check
    - 前置的Pipeline没有运行完毕 : **BLOCKED_FOR_DEPENDENCY**
    - SourceOperator的数据不可读，数据没有Ready: **BLOCKED_FOR_SOURCE**
    - SinkOperator的数据不可写: **BLOCKED_FOR_SINK**
- **RUNNABLE** ： PipeLine的task是可执行的，等待执行线程进行执行调度

- **PENDING_FINISH**：PipeLine的task准备结束，等待其他相关Pipeline task结束之后，调用close进行资源回收
- **FINISHED**：已经结束的PipeLine Task，等待Shared_ptr析构释放
- **FINISHED**：已经Cancel的的PipeLineTask，等待Shared_ptr析构释放

PipeLine的**状态机变化规则**

```
                 |-------------------------------------------------------|
                 |---|                  transfer 2    transfer 3       |
transfer 4
                     |-------> BLOCKED ------------|                    |---------
------------------------------> CANCELED
               |------|                             |                    | transfer
5           transfer 6|
 NOT_READY ---| transfer 0                         |-----> RUNNABLE ---|---------
> PENDING_FINISH ------|
               |                                    |          ^        |
              transfer 7|
               |-----------------------------------|          |--------|---------
-----------------------------> FINISHED
                transfer 1                                    transfer 9
transfer 8

 * transfer 0 (NOT_READY -> BLOCKED): this pipeline task has some incomplete
dependencies
 * transfer 1 (NOT_READY -> RUNNABLE): this pipeline task has no incomplete
dependencies
 * transfer 2 (BLOCKED -> RUNNABLE): runnable condition for this pipeline task is
met (e.g. get a new block from rpc)
 * transfer 3 (RUNNABLE -> BLOCKED): runnable condition for this pipeline task is
not met (e.g. sink operator send a block by RPC and wait for a response)
 * transfer 4 (RUNNABLE -> CANCELED): current fragment is cancelled
 * transfer 5 (RUNNABLE -> PENDING_FINISH): this pipeline task completed but wait
for releasing resources hold by itself
 * transfer 6 (PENDING_FINISH -> CANCELED): current fragment is cancelled
 * transfer 7 (PENDING_FINISH -> FINISHED): this pipeline task completed and
resources hold by itself have been released already
 * transfer 8 (RUNNABLE -> FINISHED): this pipeline task completed and no
resource need to be released
 * transfer 9 (RUNNABLE -> RUNNABLE): this pipeline task yields CPU and re-enters
the runnable queue if it is runnable and has occupied CPU for a max time slice
```

执行算子的Pipeline改造

Operator的实现

Operator是Pipeline执行查询的基本单元，可以理解为ExecNode在原有执行框架中的角色。

这部分抽象之后相对简单一些，Operator核心逻辑可以沿用之前的Exec Node的执行逻辑。笔者在ExecNode上抽象出了部分接口，实现代码复用

- **Operator类**

抽象类，不同的ExecNode抽象为对应的Operator。比如**UnionNode -》UnionOperator**，接口基本与ExecNode上的一致

```
class Operator {
public:
    explicit Operator(OperatorTemplate* operator_template);
    virtual ~Operator() = default;

    // After both sink and source need to know the cancel state.
    // do cancel work
    bool is_sink() const;

    bool is_source() const;

    // Should be call after ExecNode is constructed
    virtual Status init(ExecNode* exec_node, RuntimeState* state = nullptr);

    // Only result sink and data stream sink need to impl the virtual function
    virtual Status init(const TDataSink& tsink) { return Status::OK(); };

    // Do prepare some state of Operator
    virtual Status prepare(RuntimeState* state);

    // Like ExecNode when pipeline task first time be scheduled? can't block
    // the pipeline should be open after dependencies is finish
    // Eg a -> c, b-> c, after a, b pipeline finish, c pipeline should call open
    // Now the pipeline only have one task, so the there is no performance
bottleneck for the mechanism??
    // but if one pipeline have multi task to parallel work, need to rethink the
logic
    //
    // Each operator should call open_self() to prepare resource to do data
compute.
    // if ExecNode split to sink and source operator, open_self() should be
called in sink operator
    virtual Status open(RuntimeState* state);

    // Release the resource, should not block the thread
    //
    // Each operator should call close_self() to release resource
    // if ExecNode split to sink and source operator, close_self() should be
called in source operator
    virtual Status close(RuntimeState* state);

    virtual bool can_read() { return false; } // for source

    virtual bool can_write() { return false; } // for sink

    // for pipeline
    virtual Status get_block(RuntimeState* state, vectorized::Block* block, bool*
eos) {
        std::stringstream error_msg;
        error_msg << " has not implements get_block";
        return Status::NotSupported(error_msg.str());
    }
```

- **OperatorBuilder类的实现**

一个抽象类：主要**实现ExecNode -> Operator的工作，并且多个Operator可复用该逻辑和共用的资源，为了后续Pipeline并行做准备**

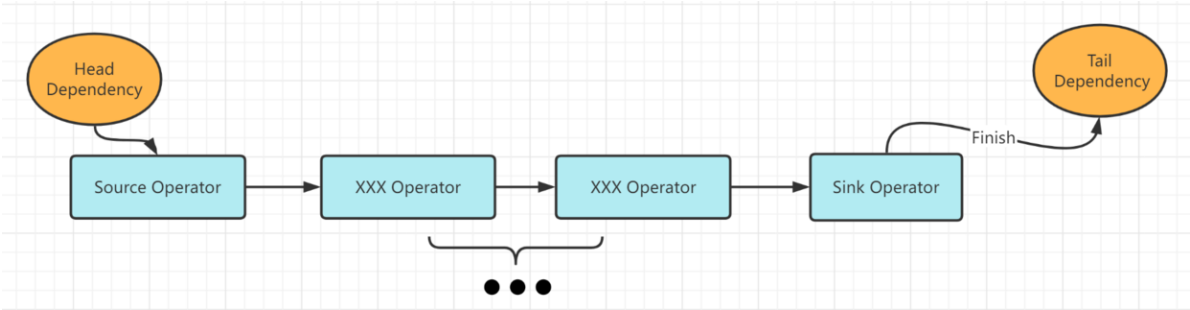
```
class OperatorTemplate {
protected:
    const int32_t _id;
    const std::string _name;
    ExecNode* _related_exec_node;
    std::shared_ptr<RuntimeProfile> _runtime_profile;

    RuntimeState* _state = nullptr;
    bool _is_closed = false;
};
```

PipelineTask的结构：

PipeLine的结构比原本火山模型的执行树简单，它不是多叉结构。整体结构就是一条链：**数据处理链条。**

- **PipeLine的起点Operator为Source Operator：只读**
- **PipeLine的起点Operator为SInk Operator：只写**
- **中间串联的若干个Operator可读可写，流式计算，不会涉及阻塞操作**
- **PipeLine Head依赖：比如Join Probe需要等待前置的Join Build准备好数据**
- **PipeLine Tail依赖：Pipeline执行完成之后要唤醒下游等待的依赖的PipeLine Task**



PipeLine Task可调度的前提：

- Head依赖完成，或没有
- Source Operator可读
- Sink Operator 可写

满足上述条件之后，PipeLine Task可以进入**Runnable状态。**当任意一个Operator生成**EOS**时，或查询被**Cancel**， PipeLineTask结束，并执行Tail依赖的处理。

阻塞算子的Pipeline拆解：

把原有的Fragment Instance的架构拆解为多个PipeLine，后续调度模块调度的基本单位便是PipeLineTask。涉及的阻塞算子有：

- Join Build
- Sort
- Agg
- Scan

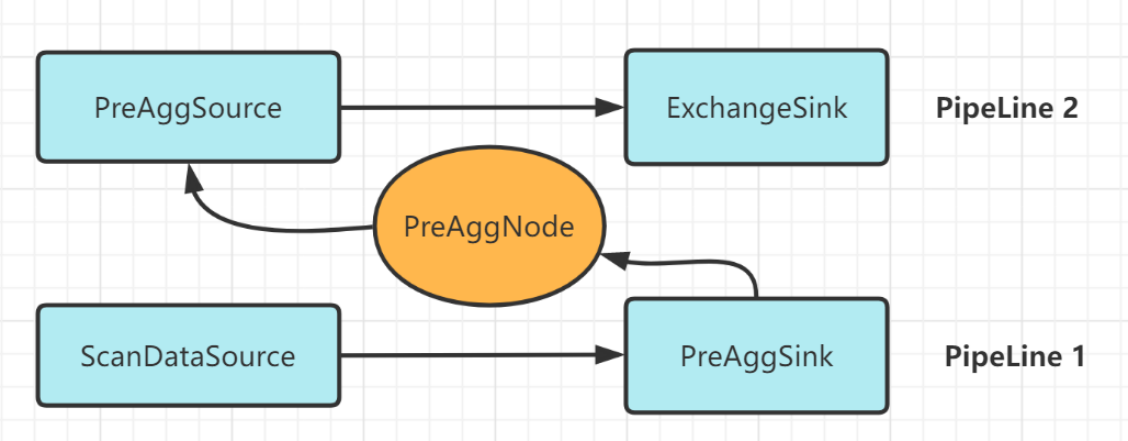
- Exchange

  这些算子需要被拆解成下面的结构：

  两种Operator:

  - is_sink：**数据写入端。**向opertor写入block，如result sink, shuffle sink。**can_write**表示可以向其写入数据，例如对exchange sink operator（由原来的VDataSteamWriter创建）来说，待shuffle写的block未超出限制时，就可以写。
  - is_source:**数据输出端**，如olap scan以及shuffle等。can_read表示该sink可执行，例如对olap scan operator来说，有数据、cancel、没有scan任务时，**can_read**返回true。

  比如AggNode，这里需要拆解为AggSink和AggSource，达到阻塞逻辑和PipeLine隔离的效果。这部分工作沿用上文提及的Operator结构进行实现。



  Sink与Source的中间状态通过原有的PreAggNode，串联起来。

4. POC的情况

原则：**小步快跑，局部验证**.

基于SSB-FLAT的POC验证结果。

配置： 单机48 core

测试方式： 4个线程，并发进行SSB-FALT的查询。

1. 改变单query的instance， 观察线程数和执行时间的变化。
2. 固定单query的instance数目为48， 限定执行线程数目，观察执行时间的变化。

8个instance			1个instance	
0.096	0.091		0.1	0.099
0.029	0.026		0.024	0.024
0.11	0.111		0.135	0.143
0.544	0.546		0.87	0.827
0.45	0.476		0.71	0.711
0.405	0.406		0.623	0.661
0.616	0.631		0.89	0.87
0.471	0.46		0.641	0.628
0.289	0.286		0.349	0.371

0.031	0.032		0.025	0.027
0.663	0.676		1.028	1.026
0.267	0.264		0.364	0.339
0.213	0.205		0.254	0.28
4.184	4.21		6.013	6.006
线程数				
246	48			
限定线程数为48				
非pipeline执行时间			pipeline执行时间	
0.238			0.178	
0.13			0.1	
0.339			0.122	
3.596			1.03	
1.486			0.446	
0.267			0.266	
2.287			0.632	
17.538			0.657	
1.117			0.461	
0.072			0.183	
20.236			0.86	
7.544			0.347	
0.211			0.199	

5. 优缺点

优点

1. 充分利用多核计算能力

能够实现不同pipeline之间的并发计算，来进行多核计算的复用。

相同pipeline之间的并发能力作为第二版PipeLine的功能进行开发。（Poc环境优势巨大，生产环境中性能影响较小）

2. 解决线程池的死锁问题与线程切换的开销

3. 解决线程切换带来的资源开销

4. CPU的资源管理

能够解决大小查询的公平调度问题，多用户之间的资源隔离问题可以利用Pipeline框架解决。更合理的方式是存算分离，在物理资源上进行隔离。

缺点

计算框架的工程复杂度

相比原始的火山模型，Pipeline模型让整个执行引擎的控制流和数据流做了拆解。**这会带来工程复杂度和问题诊断的复杂度的提升**，同时Pipeline之间的阻塞被拆解成逻辑依赖关系了，这里设计的不慎同样会带来概率性的逻辑死锁问题，**这类问题衍生出来的性能和死锁问题都是很难定位的。**

6.参考资料

[1] Leis, Viktor and Boncz, Peter and Kemper, Alfons and Neumann, Thomas, Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age, SIGMOD 2014

[2] Shaikhha, Amir and Dashti, Mohammad and Koch, Christoph, Push versus pull-based loop fusion in query engines, Journal of Functional Programming, Cambridge University Press, 2018

[3] Push-Based Execution in DuckDB, by Mark Raasveldt: slides