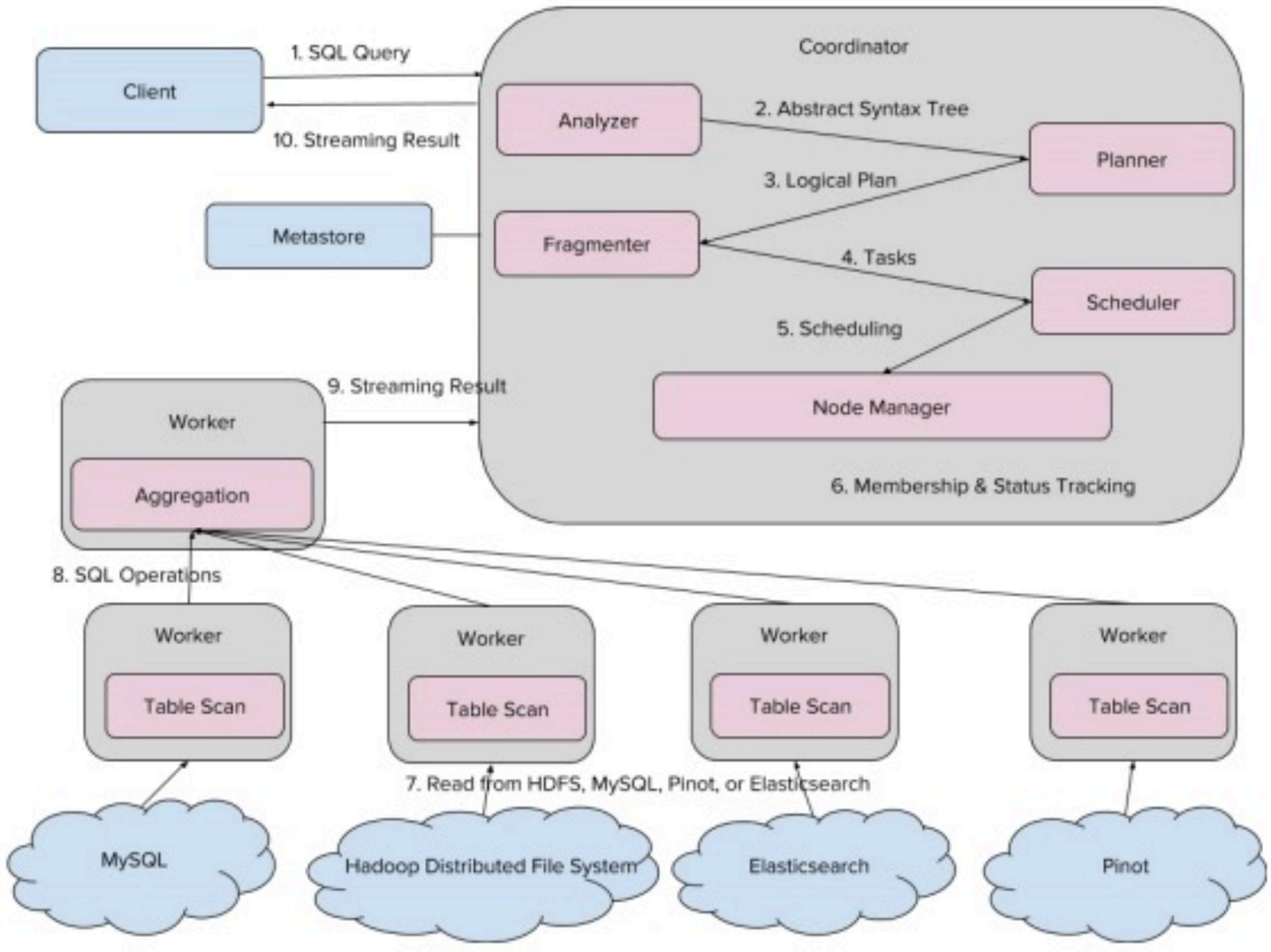
Presto 查询调度原理与实现

宦传建@大数据基础架构组

2019-10

目录



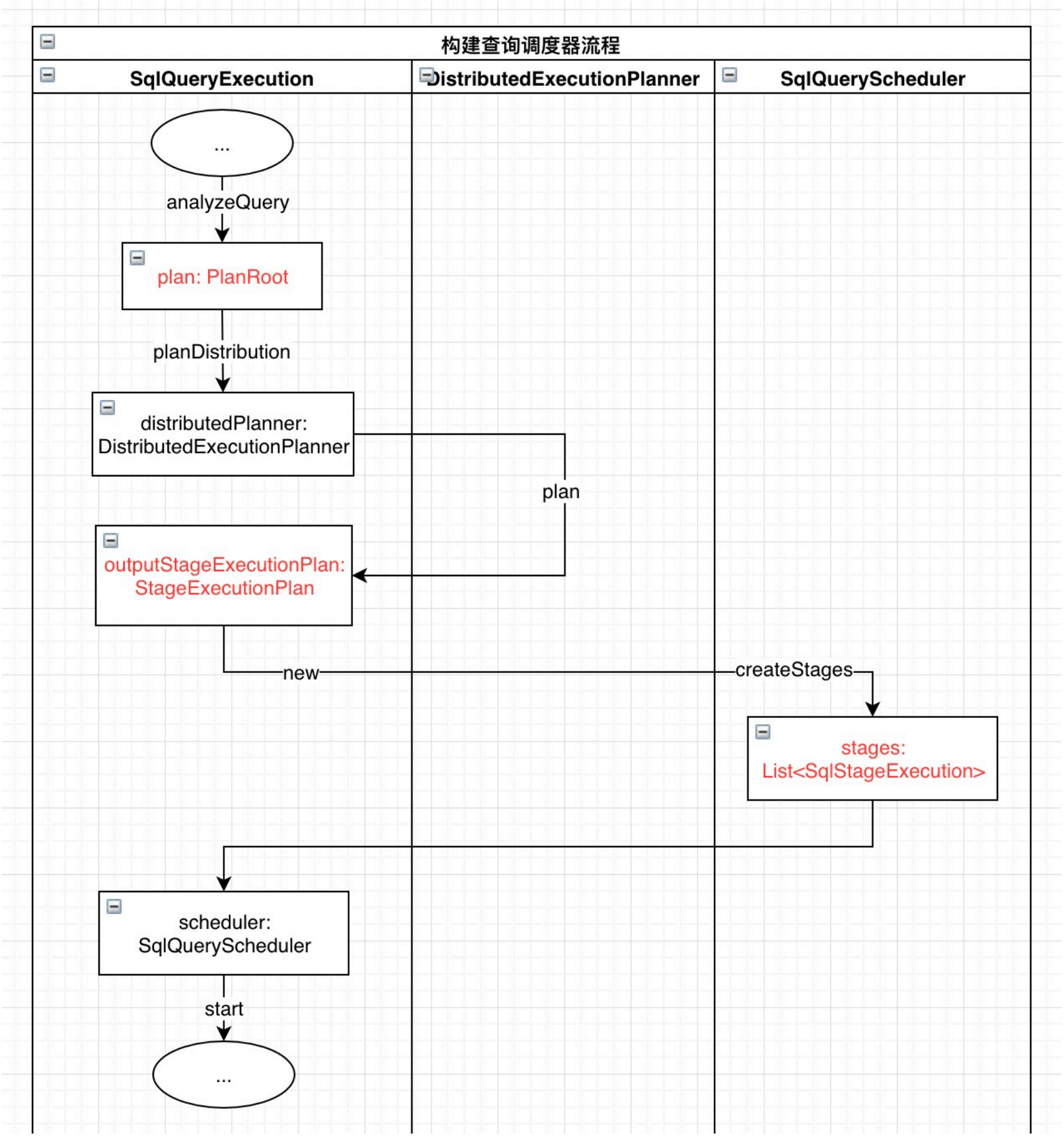
1. 经过词法、语法分析生成 AST (SqlParser)
2. 关联元数据进行语义分析 (Analyzer)
3. 生成逻辑执行计划 (LogicalPlanner)
4. 对生成的 LogicalPlan 进行优化 (PlanOptimizer)
5. 生成分布式执行计划 (PlanFragmenter)
6. **分布式执行计划调度 (SqlQueryScheduler)**
7. Task 执行 。。。

术语

- Stage : 查询执行阶段，类型有：Source, Fix, Single
- Task : Stage 在逻辑上被分为一系列 Task，每个 Task 处理 >= 1 个 Split，用于并行的执行 Stage
- Exchange : 用于 Stage 间交换数据
- OutputBuffer : 用于保存下游 Task 接收的 Split 缓冲区
- Split : 分片，一个分片是一个大的数据集中的小子集，Driver 是作用于一个 Split 上 Operator 的集合，分布式查询计划中 Source Stage 通过 Connector 从数据源读取多个分片，Source Stage 处理完 Split 后丢给后续的 Stage (Fixed, Single)
- Driver : Task 内部的并发单元，一个 Driver 其实就是作用于 Split 的一系列 Operator 的集合，一个 Driver 作用一个 Split
- Operator : 算子，一个 Operator 代表对一个 Split 的一种操作，一个 Operator 每次只会读取一个 Page 对象，相应的，每次也只会产生一个 Page 对象

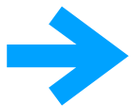
查询调度：根据生成的分布式执行计划，分配 Task 到 Worker 节点并发协同完成整个查询的过程。本质上是分配 Split 到 Worker 上的过程。

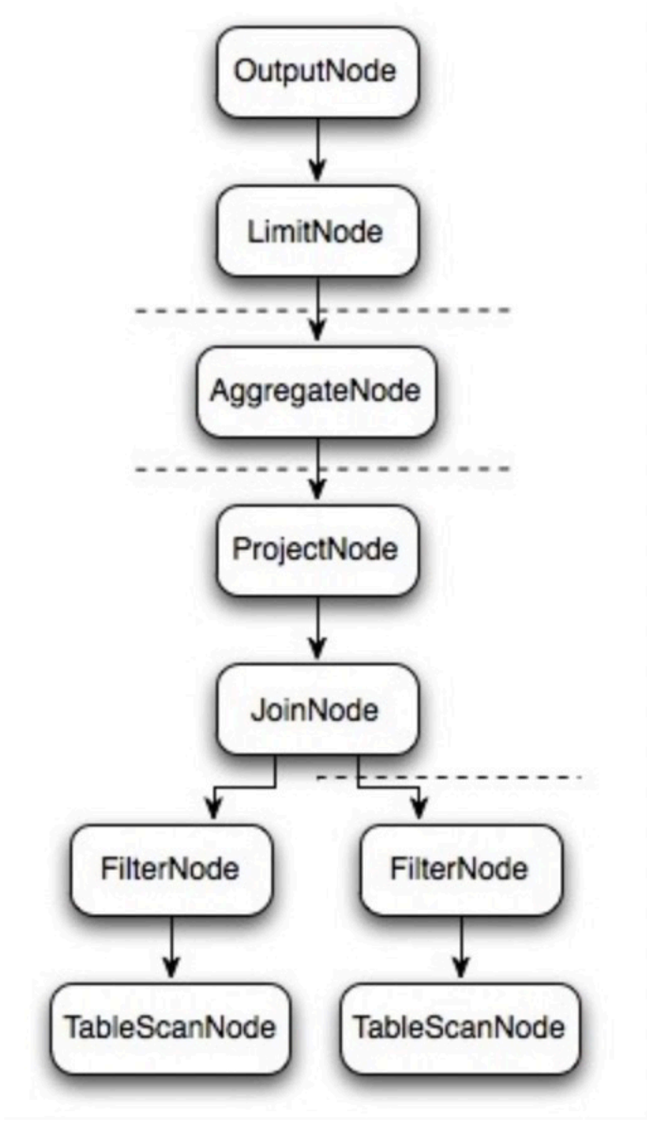
1. 生成查询调度器

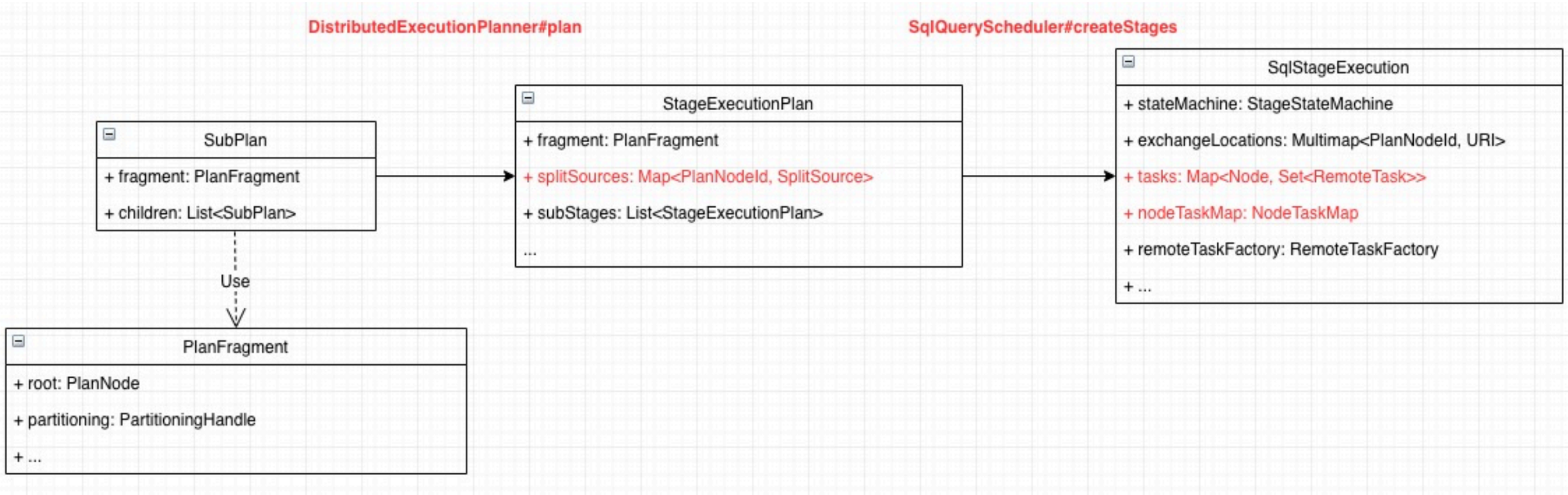


1. 生成逻辑执行计划 SubPlan
2. 创建分布式执行计划生成器 DistributedExecutionPlanner
3. 生成分布式执行计划 StageExecutionPlan
4. 创建查询调度器，包含了 SqlStageExecution，满足调度所需的上下文环境，主要是 SplitSource 和 分配的节点信息
5. 执行查询调度

```sql
select c1.rank, count(*)
from dim.city c1
join dim.city c2 on c1.id = c2.id
where c1.id > 10
group by c1.rank
limit 10;
```







- SplitSource: 针对 hive 有 HiveSplitSource 实现类，Source Stage 阶段会根据 Hive 表元数据信息，启动后台线程来异步加载 HDFS 块信息
- RemoteTask: 分配到 Worker 节点 Task 描述接口
- NodeTaskMap: 维护 Worker 节点与分配的 Task 集合映射关系

初始化 HiveSplitSource 时读 HDFS 数据了么?

SplitSource 加载 - 入口

DistributedExecutionPlanner.java

```
103 @   private StageExecutionPlan doPlan(SubPlan root, Session session, ImmutableList.Builder<SplitSource> allSplitSources)
104     {
105         PlanFragment currentFragment = root.getFragment();
106
107         // get splits for this fragment, this is lazy so split assignments aren't actually calculated here
108         Map<PlanNodeId, SplitSource> splitSources = currentFragment.getRoot().accept(new Visitor(session, currentFragment.getPipelineExecutionStrategy(), allSplitSources),  context: null);
109
110         // create child stages
111         ImmutableList.Builder<StageExecutionPlan> dependencies = ImmutableList.builder();
112         for (SubPlan childPlan : root.getChildren()) {
113             dependencies.add(doPlan(childPlan, session, allSplitSources));
114         }
115
116         return new StageExecutionPlan(
117                 currentFragment,
118                 splitSources,
119                 dependencies.build());
120     }
121
122     private final class Visitor
123             extends PlanVisitor<Map<PlanNodeId, SplitSource>, Void>
124     {
```

内部类，使用 visitor 设计模式

- PlanNode (com.facebook
  - ExplainAnalyzeNode (co
  - SortNode (com.faceboo
  - RemoteSourceNode (co
  - GroupIdNode (com.face
  - DistinctLimitNode (com.
  - SetOperationNode (com
  - MetadataDeleteNode (c
  - TableWriterNode (com.f
  - ValuesNode (com.faceb
  - TableScanNode (com.fa
  - MarkDistinctNode (com.
  - TopNRowNumberNode

TableScanNode.java

```
119
120     @JsonProperty("currentConstraint")
121     public TupleDomain<ColumnHandle> getCurrentConstraint() { return currentConstraint; }
125
126     @Override
127     public List<PlanNode> getSources() { return ImmutableList.of(); }
131
132     @Override
133 @   public <R, C> R accept(PlanVisitor<R, C> visitor, C context)
134     {
135         return visitor.visitTableScan( node: this, context);
136     }
```

```
@Override
public Map<PlanNodeId, SplitSource> visitTableScan(TableScanNode node, Void context)
{
    // get dataSource for table
    SplitSource splitSource = splitManager.getSplits(
            session,
            node.getLayout().get(),
            pipelineExecutionStrategy == GROUPED_EXECUTION ? GROUPED_SCHEDULING : UNGROUPED_SCHEDULING);

    splitSources.add(splitSource);

    return ImmutableMap.of(node.getId(), splitSource);
}
```

遍历逻辑执行计划树 SubPlan 所有节点 PlanNode，TableScanNode 定义了读取表数据的数据结构，通过 visitor 模式，调用 visitTableScan 进入到加载 SplitSource 入口 SplitManager

这里使用 visitor 设计模式有什么好处?

SplitSource 加载 - 具体流程

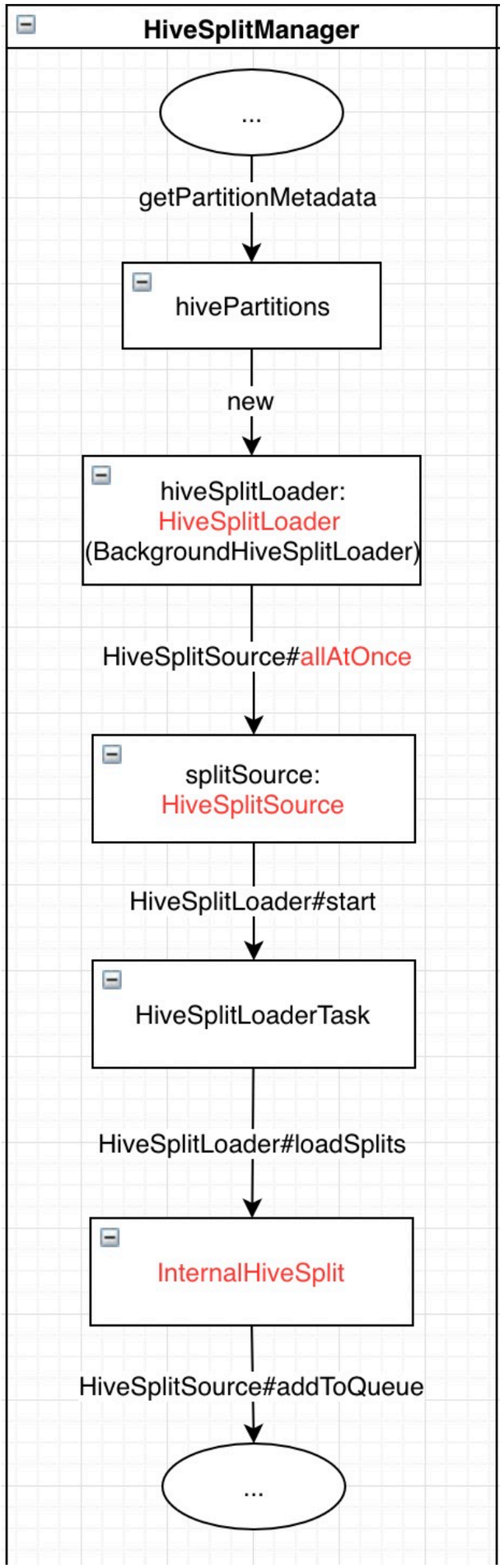
1. 获取 Hive 分区元数据

2. 创建 HiveSplitLoader，实现类 BackgroundHiveSplitLoader

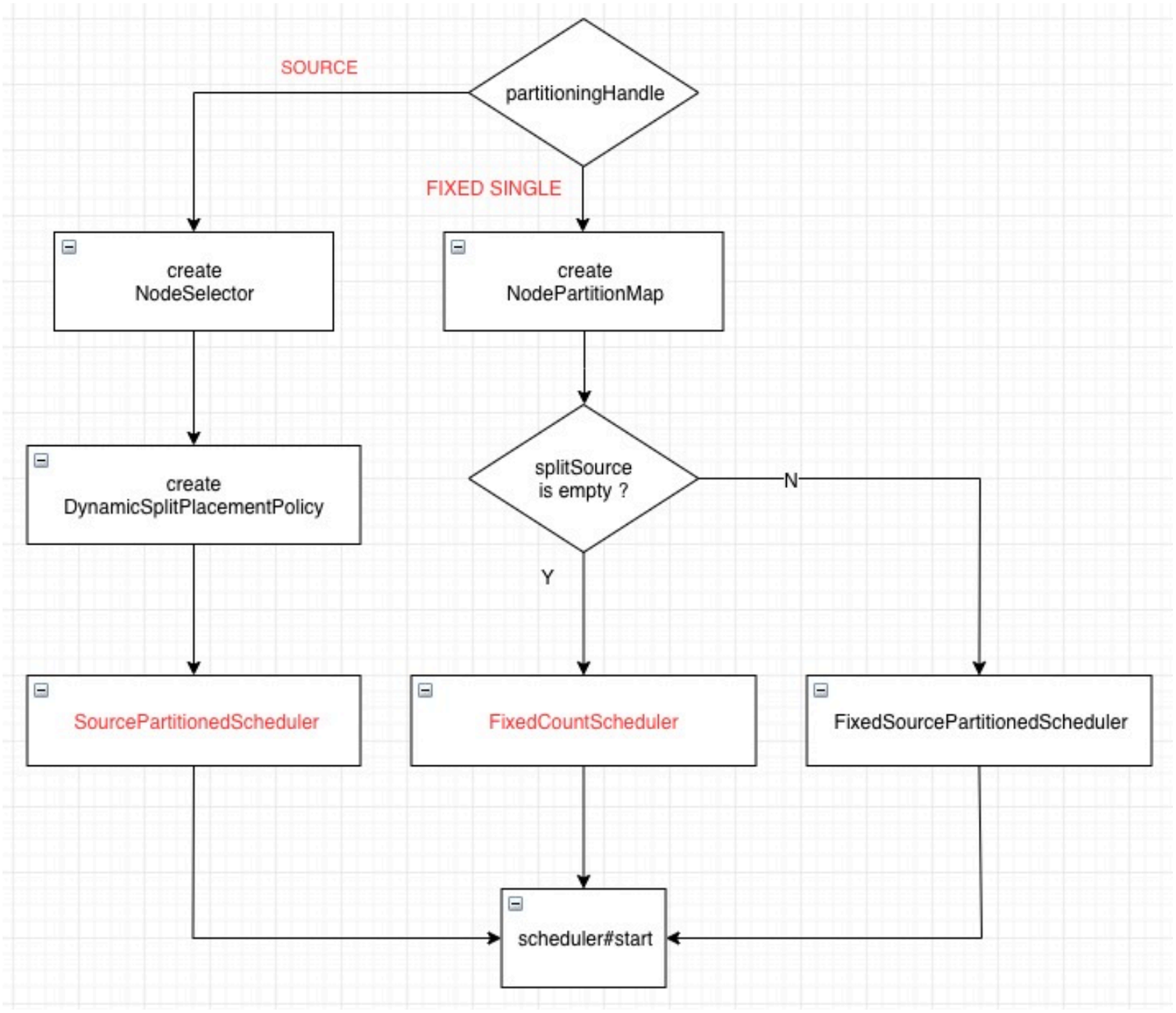
3. 根据调度策略，支持2种创建 HiveSplitSource 方式，这里忽略 HiveSplitSource#bucketed （该模式主要用于 ETL 场景，这里暂不关注）

4. 支持启动多个后台任务 HiveSplitLoaderTask 加载 HDFS 块信息封装到 InternalHiveSplit

5. 创建完成的 InternalHiveSplit 添加到 HiveSplitSource 内部维护的队列



创建 Stage 调度器



1. SOURCE Stage 首先会创建 NodeSelector 和 SplitPlacementPolicy ，提供了不同类型 Stage 分配 Task 的具体算法，是执行调度过程的核心。SOURCE Stage 会创建 SourcePartitionedScheduler
2. FIXED 或者 SINGEL Stage 首先会创建 NodePartitionMap，为每个阶段随机选择固定数量的节点，用于后续的调度，然后创建 FixedCountScheduler

```
NodeSelector
    lockDownNodes(): void
    allNodes(): List<Node>
    selectCurrentNode(): Node
    selectRandomNodes(int): List<Node>
    selectRandomNodes(int, Set<Node>): List<Node>
    computeAssignments(Set<Split>, List<RemoteTask>): SplitPlacementResult
    computeAssignments(Set<Split>, List<RemoteTask>, NodePartitionMap): SplitPlacementResult
```

2. 执行查询调度

FixedCountScheduler - 节点选择

```java
public NodePartitionMap getNodePartitionMap(Session session, NodeScheduler nodeScheduler)
{
    NodeSelector nodeSelector = nodeScheduler.createNodeSelector( connectorId: null);
    List<Node> nodes;
    if (partitioning == SystemPartitioning.COORDINATOR_ONLY) {
        nodes = ImmutableList.of(nodeSelector.selectCurrentNode());
    }
    else if (partitioning == SystemPartitioning.SINGLE) {
        nodes = nodeSelector.selectRandomNodes( limit: 1);
    }
    else if (partitioning == SystemPartitioning.FIXED) {
        nodes = nodeSelector.selectRandomNodes(getHashPartitionCount(session));
    }
```

NodeSelector 会随机选择固定数量的节点用于分配 Task，FIXED Stage 数量由 query.initial-hash-partitions 配置来控制，0.191 版本默认值是 100，SINGLE Stage 固定为 1

NodeScheduler.java

```java
206 @   public static List<Node> selectNodes(int limit, Iterator<Node> candidates)
207     {
208         checkArgument( expression: limit > 0,  errorMessage: "limit must be at least 1");
209         List<Node> selected = new ArrayList<>(limit);
210         while (selected.size() < limit && candidates.hasNext()) {
211             selected.add(candidates.next());
212         }
213
214         return selected;
215     }
216
```

NodeScheduler › selectNodes()

Hierarchy: Callers of selectNodes

Scope: All

- NodeScheduler.selectNodes(int, Iterator<Node>) (com.facebook.presto.execution.scheduler)
  - SimpleNodeSelector.selectRandomNodes(int, Set<Node>) (com.facebook.presto.execution.scheduler)
  - SimpleNodeSelector.computeAssignments(Set<Split>, List<RemoteTask>) (com.facebook.presto.execution.scheduler)
  - TopologyAwareNodeSelector.selectRandomNodes(int, Set<Node>) (com.facebook.presto.execution.scheduler)
    - NodeSelector.selectRandomNodes(int) (com.facebook.presto.execution.scheduler)
      - SystemPartitioningHandle.getNodePartitionMap(Session, NodeScheduler)(2 usages) (com.facebook.presto.sql.planner)
        - NodePartitioningManager.getNodePartitioningMap(Session, PartitioningHandle) (com.facebook.presto.sql.planner)
          - SqlQueryScheduler.SqlQueryScheduler(QueryStateMachine, LocationFactory, StageExecutionPlan, NodePartitioning

FixedCountScheduler - 创建 Task

```java
@Override
public ScheduleResult schedule()
{
    List<RemoteTask> newTasks = partitionToNode.entrySet().stream()
            .map(entry -> taskScheduler.apply(entry.getValue(), entry.getKey()))
            .collect(toImmutableList());

    return new ScheduleResult( finished: true, newTasks,  splitsScheduled: 0);
}
```

SourcePartitionedScheduler - 节点选择入口

TopologyAwareNodeSelector.java

```
129         boolean splitWaitingForAnyNode = false;
130         for (Split split : splits) {
131             if (!split.isRemotelyAccessible()) {
132                 List<Node> candidateNodes = selectExactNodes(nodeMap, split.getAddresses(), includeCoordinator);
133                 if (candidateNodes.isEmpty()) {
134                     log.debug("No nodes available to schedule %s. Available nodes %s", split, nodeMap.getNodesByHost().keys());
135                     throw new PrestoException(NO_NODES_AVAILABLE, "No nodes available to run query");
136                 }
137                 Node chosenNode = bestNodeSplitCount(candidateNodes.iterator(), minCandidates, maxPendingSplitsPerTask, assignmentStats);
138                 if (chosenNode != null) {
139                     assignment.put(chosenNode, split);
140                     assignmentStats.addAssignedSplit(chosenNode);
141                 }
142                 // Exact node set won't matter, if a split is waiting for any node
143                 else if (!splitWaitingForAnyNode) {
144                     blockedExactNodes.addAll(candidateNodes);
145                 }
146                 continue;
147             }
```

TopologyAwareNodeSelector › computeAssignments()

Hierarchy: Callers of computeAssignments

Scope: All

- TopologyAwareNodeSelector.computeAssignments(Set<Split>, List<RemoteTask>) (com.facebook.presto.execution.scheduler)
  - TestNodeScheduler.testScheduleRemote() (com.facebook.presto.execution)
  - BenchmarkNodeScheduler.benchmark(BenchmarkData) (com.facebook.presto.execution)
  - ~~TestNodeScheduler.testBasicAssignment() (com.facebook.presto.execution)~~
  - **DynamicSplitPlacementPolicy.computeAssignments(Set<Split>) (com.facebook.presto.execution.scheduler)**
    - SourcePartitionedScheduler.schedule() (com.facebook.presto.execution.scheduler)
  - TestNodeScheduler.testScheduleLocal() (com.facebook.presto.execution)
  - TestNodeScheduler.testMaxSplitsPerNode() (com.facebook.presto.execution)
  - TestNodeScheduler.testTopologyAwareScheduling() (5 usages) (com.facebook.presto.execution)
  - TestNodeScheduler.testMaxSplitsPerNodePerTask() (com.facebook.presto.execution)

SourcePartitionedScheduler - 节点选择策略

```java
@Nullable
private Node bestNodeSplitCount(Iterator<Node> candidates, int minCandidatesWhenFull, int maxPendingSplitsPerTask, NodeAssignmentStats assignmentStats)
{
    Node bestQueueNotFull = null;
    int min = Integer.MAX_VALUE;
    int fullCandidatesConsidered = 0;

    while (candidates.hasNext() && (fullCandidatesConsidered < minCandidatesWhenFull || bestQueueNotFull == null)) {
        Node node = candidates.next();
        if (assignmentStats.getTotalSplitCount(node) < maxSplitsPerNode) {
            return node;
        }
        fullCandidatesConsidered++;
        int totalSplitCount = assignmentStats.getQueuedSplitCountForStage(node);
        if (totalSplitCount < min && totalSplitCount < maxPendingSplitsPerTask) {
            min = totalSplitCount;
            bestQueueNotFull = node;
        }
    }
    return bestQueueNotFull;
}
```

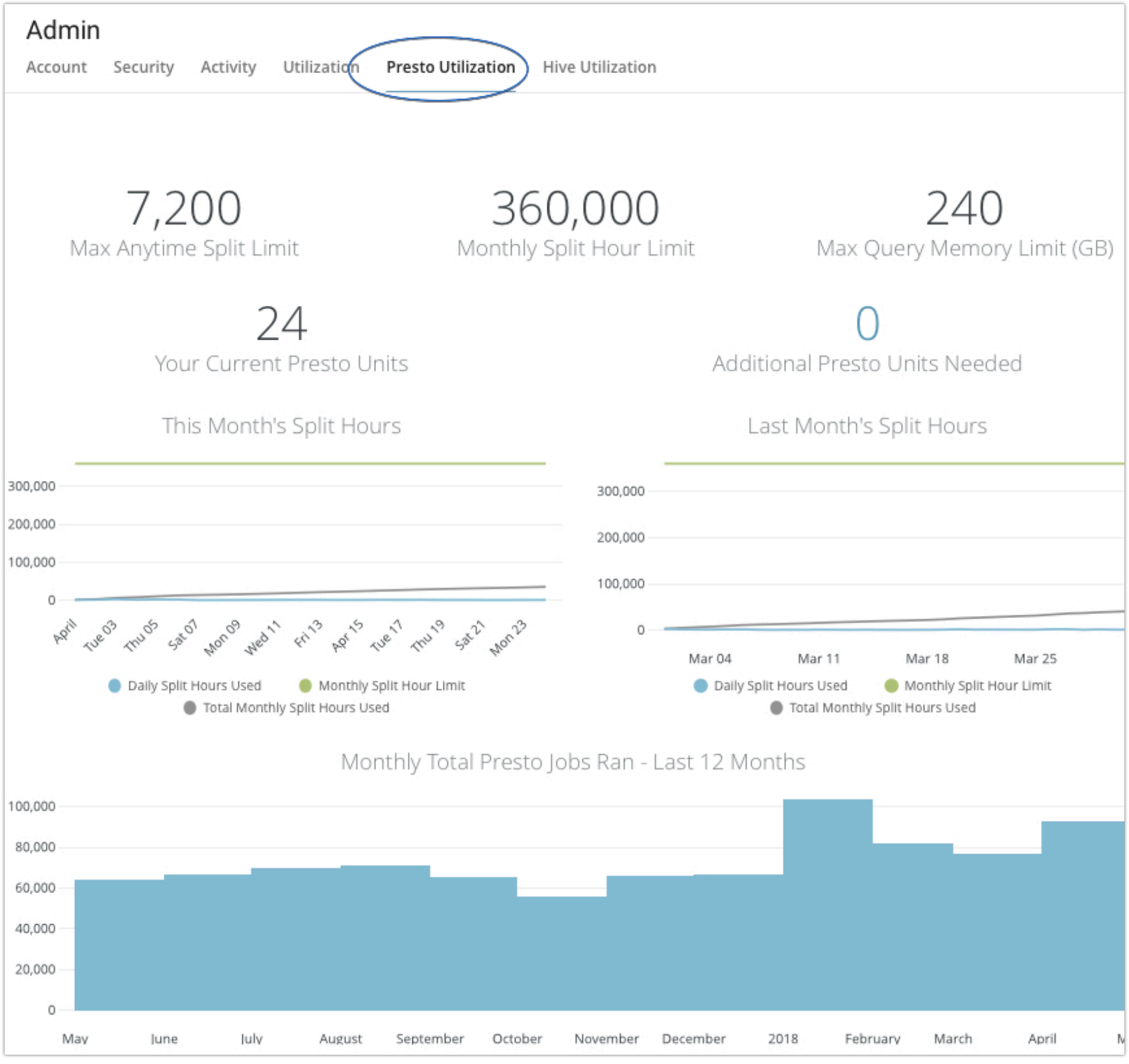
1. 从候选节点列表中随机选择一个，如果该节点已分配的 Split 尚未达到阈值，则选择该节点，该阈值可通过 node-scheduler.max-splits-per-node 配置项调整。
2. 如果第1步选择的节点已分配 Split 达到上限，则选择剩余节点中排队 Split 最少的节点。

SourcePartitionedScheduler - 创建 Task

SourcePartitionedScheduler.java

```
431
432 @      private Set<RemoteTask> assignSplits(Multimap<InternalNode, Split> splitAssignment, Multimap<InternalNode, Lifespan> noMoreSplitsNotification)
433        {
434            ImmutableSet.Builder<RemoteTask> newTasks = ImmutableSet.builder();
435
436            ImmutableSet<InternalNode> nodes = ImmutableSet.<~>builder()
437                    .addAll(splitAssignment.keySet())
438                    .addAll(noMoreSplitsNotification.keySet())
439                    .build();
440            for (InternalNode node : nodes) {
441                // source partitioned tasks can only receive broadcast data; otherwise it would have a different distribution
442                ImmutableMultimap<PlanNodeId, Split> splits = ImmutableMultimap.<~>builder()
443                        .putAll(partitionedNode, splitAssignment.get(node))
444                        .build();
445                ImmutableMultimap.Builder<PlanNodeId, Lifespan> noMoreSplits = ImmutableMultimap.builder();
446                if (noMoreSplitsNotification.containsKey(node)) {
447                    noMoreSplits.putAll(partitionedNode, noMoreSplitsNotification.get(node));
448                }
449                newTasks.addAll(stage.scheduleSplits(
450                        node,
451                        splits,
452                        noMoreSplits.build()));
453            }
454            return newTasks.build();
455        }
456
```

Split - 集群资源抽象单元



这些原理在监控如何体现和帮助定位问题?

GetSplit JMX 监控

SOURCE Stage 构建 HiveSplitSource 时，与 NameNode 交互存在延时，导致 GetSplitTime 抖动

```
where
  t.ds > format_datetime(
    parse_datetime('20191014', 'yyyyMMdd') – interval '14' day,
    'yyyyMMdd'
  )
  and t.ds <= '20191014'
  and type_id = 1
group by
  ds
```

ALL

stageId	状态	stage类型	总Task数	总Driver数	读取数据大小	读取数据行数
2	FINISHED	SOURCE	32	32	47.81kB	13
1	FINISHED	FIXED	8	64	4.09kB	13
0	FINISHED	SINGLE	1	4	1.57kB	13

1. SOURCE Stage 读取 13 个分区数据，存在 32 个 HDFS 块，封装为 32 个 Split，根据前面的调度策略，这里会分配 32 个 Task
2. FIXED Stage 取值 query.initial-hash-partitions=8（默认值 100），因此会创建 8 个 Task
3. SINGLE Stage 固定创建 1 个 Task

数据分布对 SOURCE Stage 的影响

query_table	cnt	partitionsAvg	partitionsMax
st_xml_time_t_leaguesite	78652	392.7076	1262
st_xml_time_overtime_detail	1362	64.616	490

```
show partitions st
```

	partition
1	ds=20181231/overtime_type_id=1/is_taobao=0/time_type_id=1
2	ds=20181231/overtime_type_id=1/is_taobao=0/time_type_id=2
3	ds=20181231/overtime_type_id=1/is_taobao=0/time_type_id=3
4	ds=20181231/overtime_type_id=1/is_taobao=0/time_type_id=4
5	ds=20181231/overtime_type_id=1/is_taobao=0/time_type_id=5
6	ds=20181231/overtime_type_id=1/is_taobao=0/time_type_id=6
7	ds=20181231/overtime_type_id=1/is_taobao=0/time_type_id=7
8	ds=20181231/overtime_type_id=1/is_taobao=1/time_type_id=1
9	ds=20181231/overtime_type_id=1/is_taobao=1/time_type_id=2
10	ds=20181231/overtime_type_id=1/is_taobao=1/time_type_id=3
11	ds=20181231/overtime_type_id=1/is_taobao=1/time_type_id=4
12	ds=20181231/overtime_type_id=1/is_taobao=1/time_type_id=5
13	ds=20181231/overtime_type_id=1/is_taobao=1/time_type_id=6

Hive 表分区设计不合理，分区键过多，设计的某些分区条件实际查询过程中使用频率很低，导致查询分区数过多

日期： 2019-10-12 00:00:00 至 2019-10-12 09:37:01　状态： ALL　queryId： queryId查询　用户： biboard　查询

操作	查询id	用户	状态	查询开始时间	执行时间	生成执行计划耗时	累计阻塞耗时	累计调度耗时	总任务数	总Driver数	读取数据大小	读取数据行数
	2019101...	biboard	FINISHED	2019-10-12 01:38:48	25.52s	74.76ms	2.12m	33.75s	100	459	703.30kB	3710
	2019101...	biboard	FINISHED	2019-10-12 06:10:48	23.96s	45.42ms	1.99m	31.79s	100	459	703.36kB	3710
	2019101...	biboard	FINISHED	2019-10-12 01:38:50	23.49s	35.89ms	1.95m	31.44s	100	459	703.33kB	3710
	2019101...	biboard	FINISHED	2019-10-12 01:38:49	23.42s	35.88ms	1.95m	30.42s	100	459	703.24kB	3710
	2019101...	biboard	FINISHED	2019-10-12 01:30:52	23.17s	41.04ms	26.57m	3.66m	86	165	26.25kB	40
	2019101...	biboard	FINISHED	2019-10-12 06:10:49	22.52s	34.98ms	1.87m	33.98s	100	459	703.34kB	3710
	2019101...	biboard	FINISHED	2019-10-12 06:10:50	22.43s	35.22ms	1.86m	31.51s	100	459	703.37kB	3710
	2019101...	biboard	FINISHED	2019-10-12 02:05:36	21.08s	90.14ms	4.80h	19.82s	110	867	305.72kB	59341
	2019101...	biboard	FINISHED	2019-10-12 06:20:48	19.30s	81.60ms	1.60m	2.25m	100	459	703.28kB	3710
	2019101...	biboard	FINISHED	2019-10-12 02:05:38	18.96s	38.08ms	21.74m	1.85s	11	71	4.13kB	159
	2019101...	biboard	FINISHED	2019-10-12 01:48:48	18.50s	78.32ms	1.53m	44.38s	100	459	703.31kB	3710
	2019101...	biboard	FINISHED	2019-10-12 01:30:54	18.32s	56.82ms	2.20h	57.49s	113	528	11.37kB	119
	2019101...	biboard	FINISHED	2019-10-12 06:20:50	18.07s	38.08ms	1.50m	2.23m	100	459	703.33kB	3710
	2019101...	biboard	FINISHED	2019-10-12 06:20:49	17.88s	36.22ms	1.48m	2.23m	100	459	703.27kB	3710
	2019101...	biboard	FINISHED	2019-10-12 02:05:40	17.43s	60.29ms	20.01m	1.55s	11	71	4.13kB	159
	2019101...	biboard	FINISHED	2019-10-12 02:05:40	17.40s	36.04ms	19.93m	1.58s	11	71	4.13kB	159
	2019101...	biboard	FINISHED	2019-10-12 01:48:50	17.12s	36.75ms	1.42m	22.30s	100	459	703.30kB	3710
	2019101...	biboard	FINISHED	2019-10-12 01:48:49	17.02s	35.96ms	1.41m	22.36s	100	459	703.37kB	3710
	2019101...	biboard	FINISHED	2019-10-12 06:20:42	16.18s	47.64ms	18.28m	28.03s	11	70	5.65kB	1
	2019101...	biboard	FINISHED	2019-10-12 01:30:52	15.91s	57.12ms	36.06m	2.99m	113	528	11.37kB	119
	2019101...	biboard	FINISHED	2019-10-12 06:20:43	15.48s	61.30ms	42.38m	2.39s	20	196	5.65kB	6
	2019101...	biboard	FINISHED	2019-10-12 06:20:43	15.47s	63.24ms	42.32m	2.29s	20	196	5.65kB	6
	2019101...	biboard	FINISHED	2019-10-12 06:10:47	15.32s	54.89ms	33.50m	26.27s	19	134	4.13kB	159

资源占用

```
select
  max(cast(stat_date as timestamp))
from
  st._____day
where
  ds >= '20190101'
```

不合理的 SQL 查询了所有分区，这张表主要是汇总数据，非明细数据，使用分区导致单个 HDFS 块很小，小文件很多。

不同 Stage 的 Task 之间如何交换数据?

Task 如何异步并行下发?

Lazy OutputBuffer

1. 在 AllAtOnceExecution 的调度模式下，虽然 Stage 是自底向上被调度的，但是由于 Task 的异步调度机制存在，某个 Task 被 Worker 执行的时候，其上游的 Task 可能并没有被对应的 Worker 执行
2. 当某个 Task 还未被 Worker 执行时就收到下游 Task 数据读取的请求时，LazyOutputBuffer 会记录下该请求，当该 Task 被执行并生成数据后，再响应对应的请求
3. 随着上游 Task 被调度，通过 StageLinkage 的依赖关系，下游 Task 会被动态分配 split，该 split 中包含了上游 Task 的 Location 信息
4. 可以看出 Presto Stage Scheduler 是完全解耦的，很灵活，任意一个 Task 被下发，并不依赖于其上下游 Stage 的调度。Task 的异步并行下发机制是 Presto 能够达到秒级响应的。

3. QA

谢谢！