Spark SQL 执行计划和优化逻辑

极客时间

金澜涛

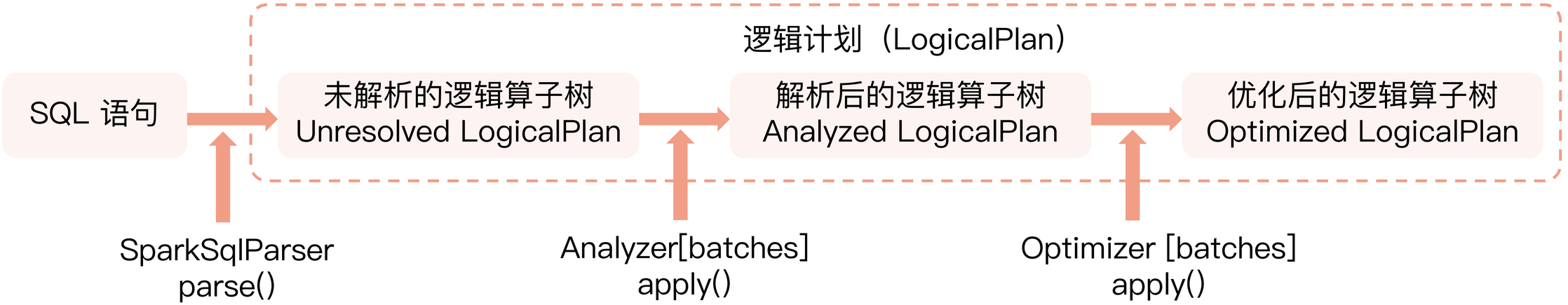
目录

逻辑计划树和优化器

逻辑计划生命周期

三个阶段：

1. 由 SparkSqlParser 中的 AstBuilder 将语法树的各个节点转换为对应逻辑计划节点，组成 Unresolved 的逻辑算子树，不包含数据信息与列信息。
2. Analyzer 结合 Catalog 将一系列规则作用在 Unresolved 的逻辑算子树上，生成 Analyzed 的逻辑算子树。
3. Optimizer 将一系列优化规则应用在逻辑算子树中，确保结果正确的前提下改进低效结构，生成优化后的逻辑算子树。



逻辑计划树的分类

LeafNode

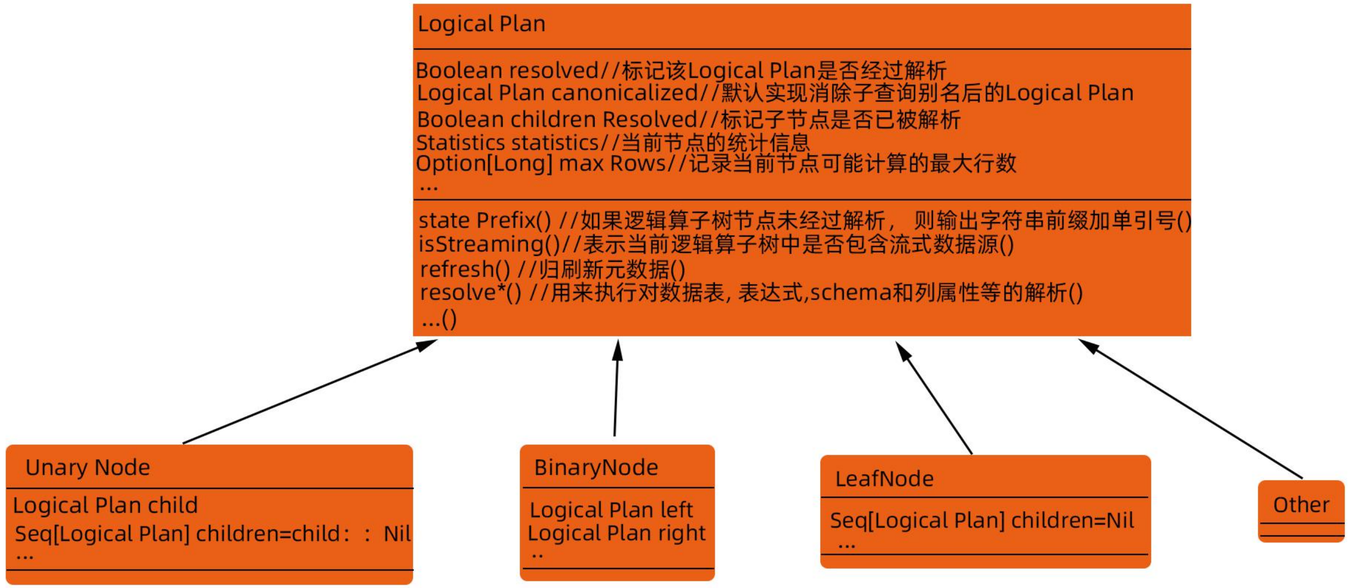
- RunnableCommand

UnaryNode

- RedistributeData
- basicLogicalOperators

BinaryNode

- Join
- CoGroup



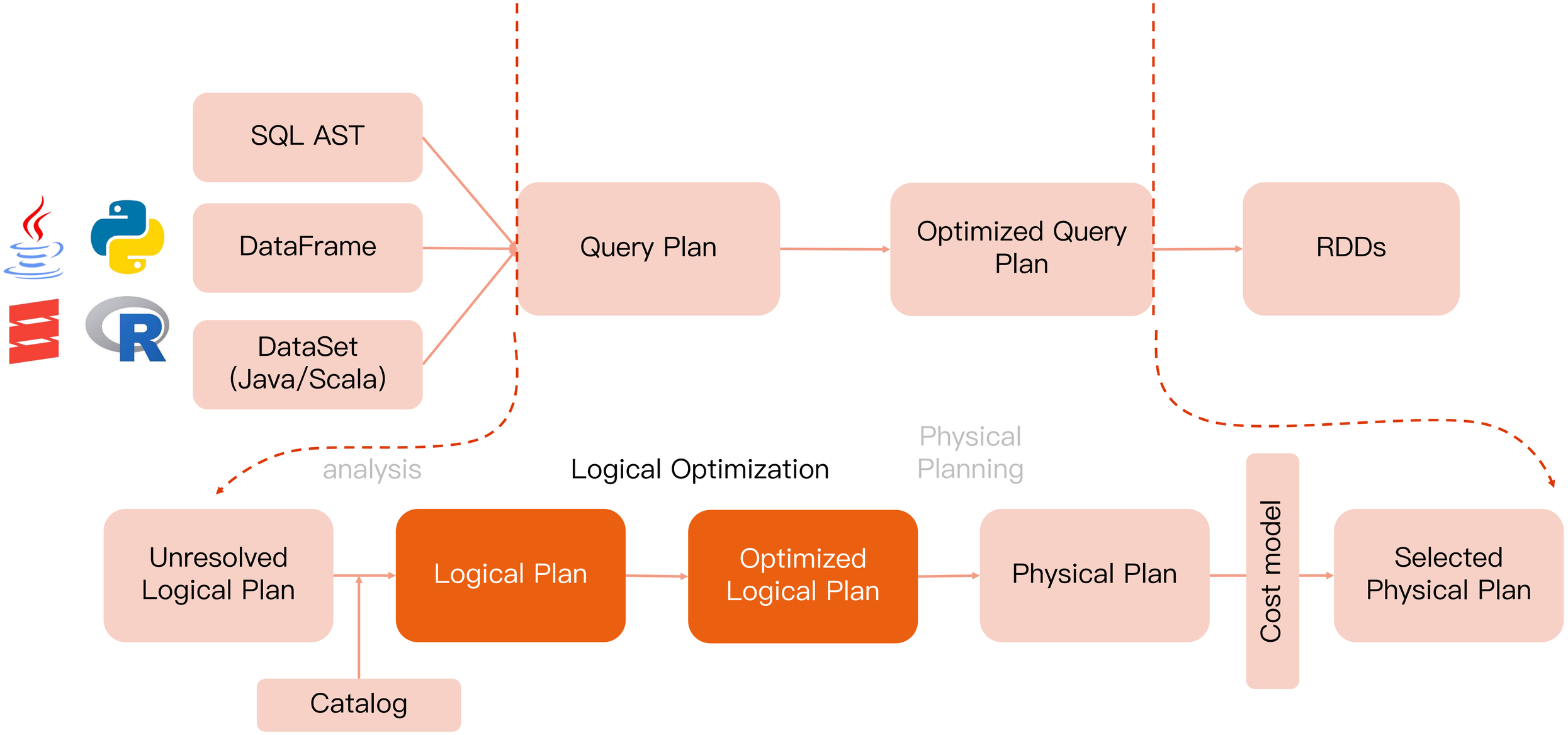
Optimizer

- 优化器是整个 Catalyst 的核心，优化器分为基于规则优化和基于代价优化两种。
- 基于规则的优化策略实际上就是对语法树进行一次遍历，对模式匹配能够满足特定规则的节点进行相应的等价转换。因此，基于规则优化说到底就是一棵树等价地转换为另一棵树。
- SQL 中经典的优化规则有很多，下文结合示例介绍三种比较常见的规则：谓词下推（将过滤尽可能地下沉到数据源端）、常量累加（比如1 + 2事先计算好）和列剪枝（减少读取不必要的列）。

Optimizer

1. Sql text 经过 SqlParser 解析成 Unresolved LogicalPlan
2. Analyzer 模块结合 Catalog 进行绑定，生成 Resolved LogicalPlan
3. Optimizer 模块对 Resolved LogicalPlan 进行优化，生成 Optimized LogicalPlan

Optimizer


SQL AST
DataFrame
DataSet
(Java/Scala)
Query Plan
Optimized Query
Plan
RDDs
analysis
Logical Optimization
Physical
Planning
Unresolved
Logical Plan
Logical Plan
Optimized
Logical Plan
Physical Plan
Cost model
Selected
Physical Plan
Catalog

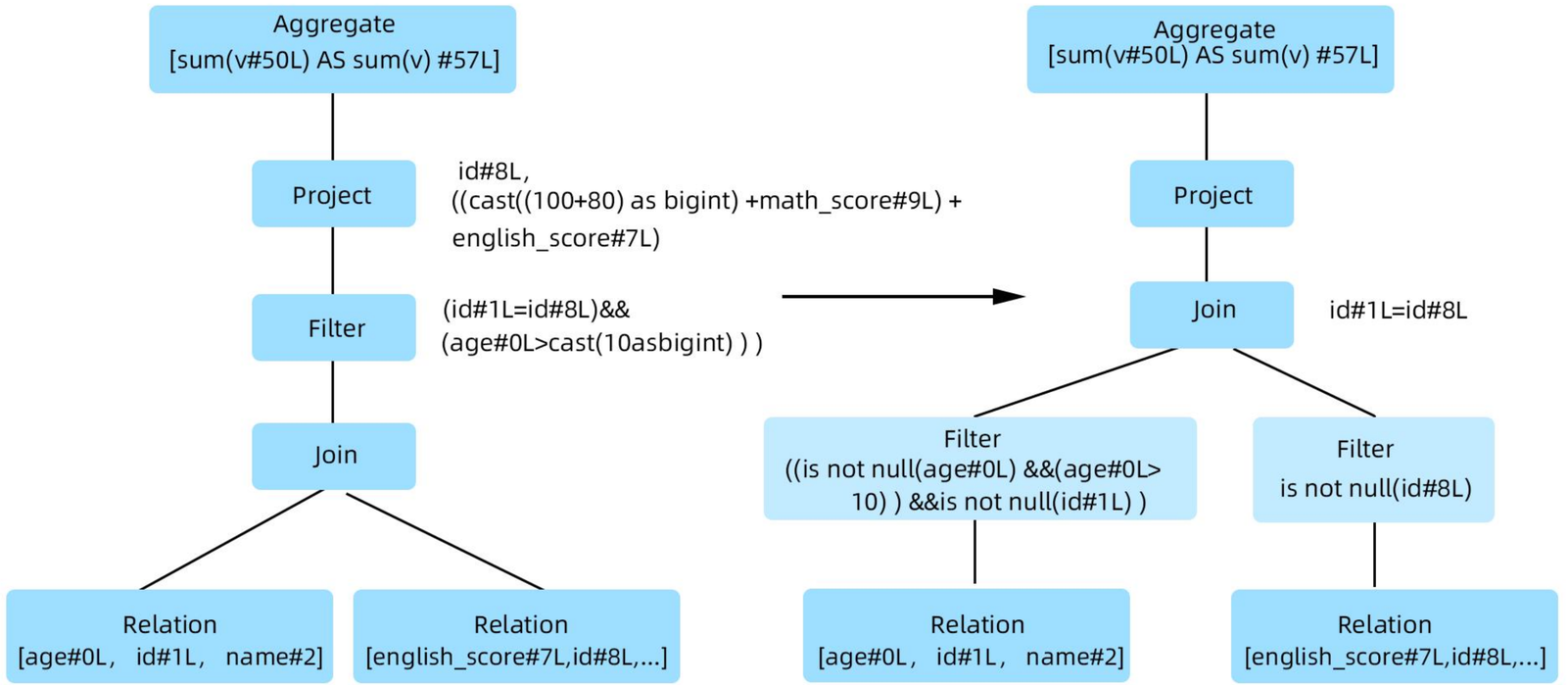
谓词下推（Predicate Pushdown）

谓词下推是由 PushDownPredicate 规则实现，这个过程主要将过滤条件尽可能地下推到底层，最好是数据源。

例如， 语法树中两个表先做 join，之后再使用 age>10 对结果进行过滤。join 算子通常是一个非常耗时的算子，耗时多少一般取决于参与 join 的两个表的大小，如果能够减少参与 join 两表的大小，就可以大大降低 join 算子所需时间。

谓词下推能将过滤条件下推到 join 之前进行，如图中过滤条件 age>0 以及 id!=null 两个条件就分别下推到了 join 之前。这样系统在扫描数据的时候就对数据进行了过滤，参与 join 的数据量将会显著减少，join 耗时必然也会降低。

谓词下推（Predicate Pushdown）

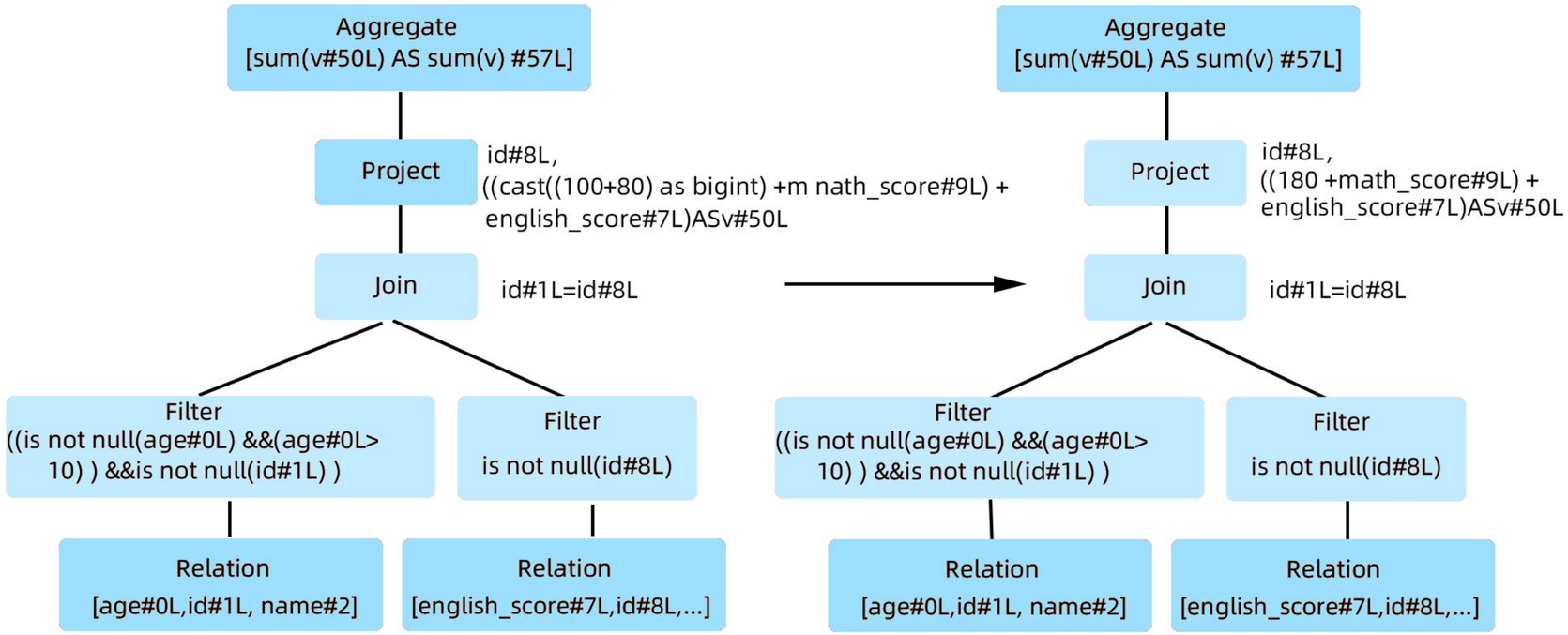


常量累加（Constant Folding）

常量累加其实很简单，就是上文中提到的规则 x+(1+2) –> x+3。

示例如果没有进行优化的话，每一条结果都需要执行一次 100+80 的操作，然后再与变量 math_score 以及 english_score 相加，而优化后就不需要再执行 100+80 操作。

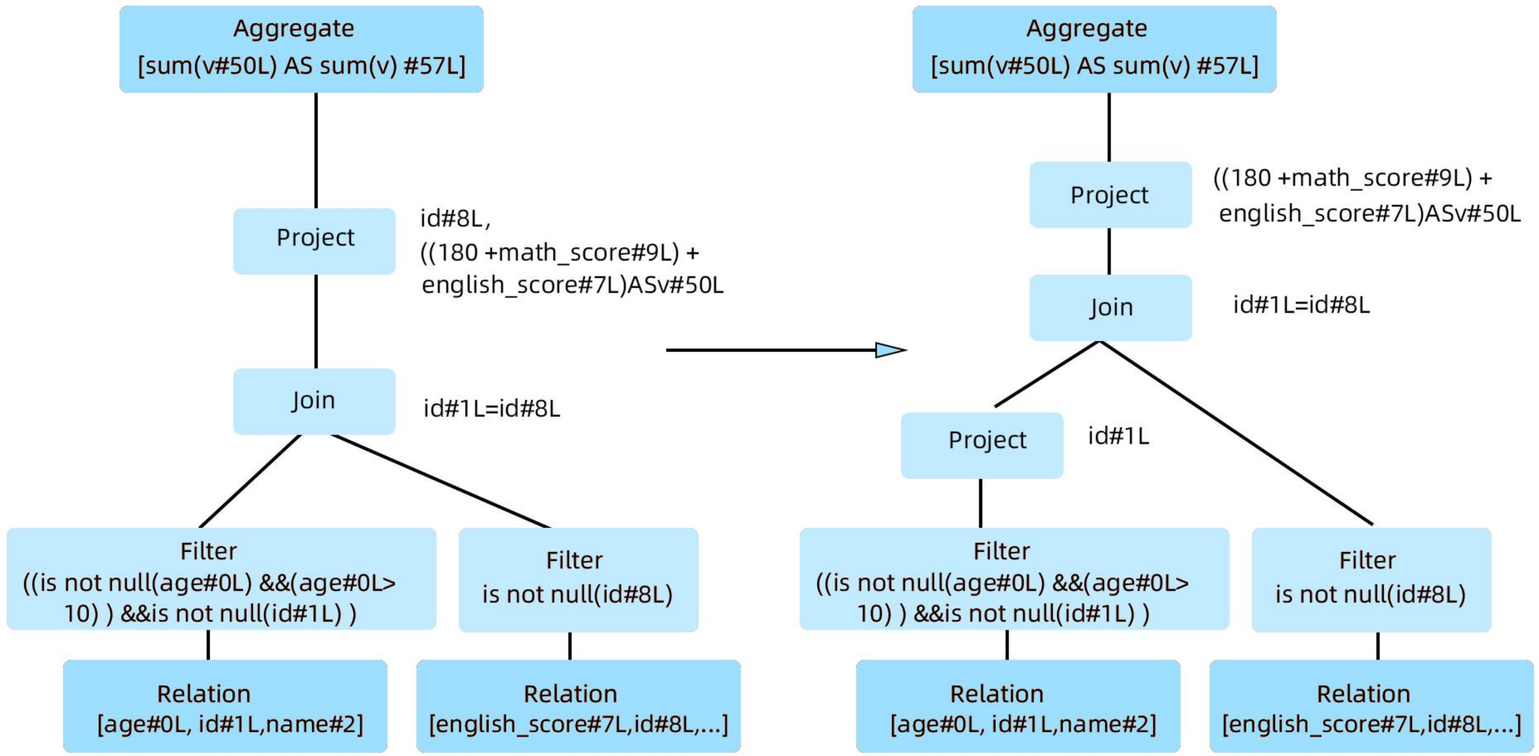
常量累加（Constant Folding）



列剪枝（Column Pruning）

列值裁剪是另一个经典的规则，示例中对于 people 表来说，并不需要扫描它的所有列值，而只需要 id 列，所以在扫描 people 之后需要将其他列进行裁剪，只留下列 id。这个优化一方面大幅度减少了网络、内存数据量消耗，另一方面对于列存格式（Parquet）来说大大提高了扫描效率。

列剪枝（Column Pruning）



Rule 举例

优化规则	优化操作
PushProjectionThroughUnion	列剪裁下推
Reorderjoin	Join 顺序优化
EliminateOuterjoin	OuterJoin 消除
PushPredicateThroughjoin	谓词下推到 Join 算子
PushDownPredicate	谓词下推
LimitPushDown	Limit 算子下推
ColumnPruning	列剪裁
InferFiltersFromConstraints	约束条件提取
CollapseRepurtition	重分区组合
CollapseProject	投影算子组合
CollapseWindow	Window 组合
CombineFthers	过滤条件组合
CombineLimits	Limit 操作组合
CombineUnions	Union 算子组合
NullPropagation	Null 提取
FoldablePropagation	可折叠算子提取
OptimizeIn	In 操作优化
ConstantFolding	常数折叠
ReorderAssociativeOperator	重排序关联算子优化
LikeSimplification	Like 算子简化
BooleanSimplifcation	Boolean 算子简化
SimplifyConditionals	条件简化
RemoveDispensableExpressions	Dispensable 表达式消除
SimplifyBinaryComparison	比较算子简化
PruneFilter	过滤条件剪裁
EliminateSorts	排序算子消除
SimplifyCasts	Cast 算子简化
SimplifyCaseConversionExpressions	case 表达式简化
RewriteCorrelatedScalarSubquery	依赖子查询重写
EliminateSerialization	序列化消除
RemoveAliasOnlyProiect	消除别名

物理计划树和策略器

物理计划树

- 经过 Optimizer 优化后的逻辑计划并不知道如何执行，例如算子节点 Relation（实际上是 LogicalRelation）虽然代表本次查询会从一张确定的表中获取数据，如 marketing.buyers，但是这张表是什么类型的表（Hive 还是 HBase），如何获取（JDBC 还是读 HDFS 文件），数据分布是什么样的（Bucketed 还是 HashDistributed）此时并不清楚。
- 需要将逻辑计划树转换成物理计划树，以获取真实的物理属性。例如，Relation 算子变为 FileSourceScanExec，Join 算子变为 SortMergeJoinExec。
- 一般的，物理算子以 Exec 结尾。

SparkPlan = PhysicalPlan

SparkPlan 实际上就是我们所说的物理计划，它是所有物理计划抽象类。

有了 SparkPlan Tree，才能将其转换成 RDD 的 DAG。

SparkPlan 也有四类：

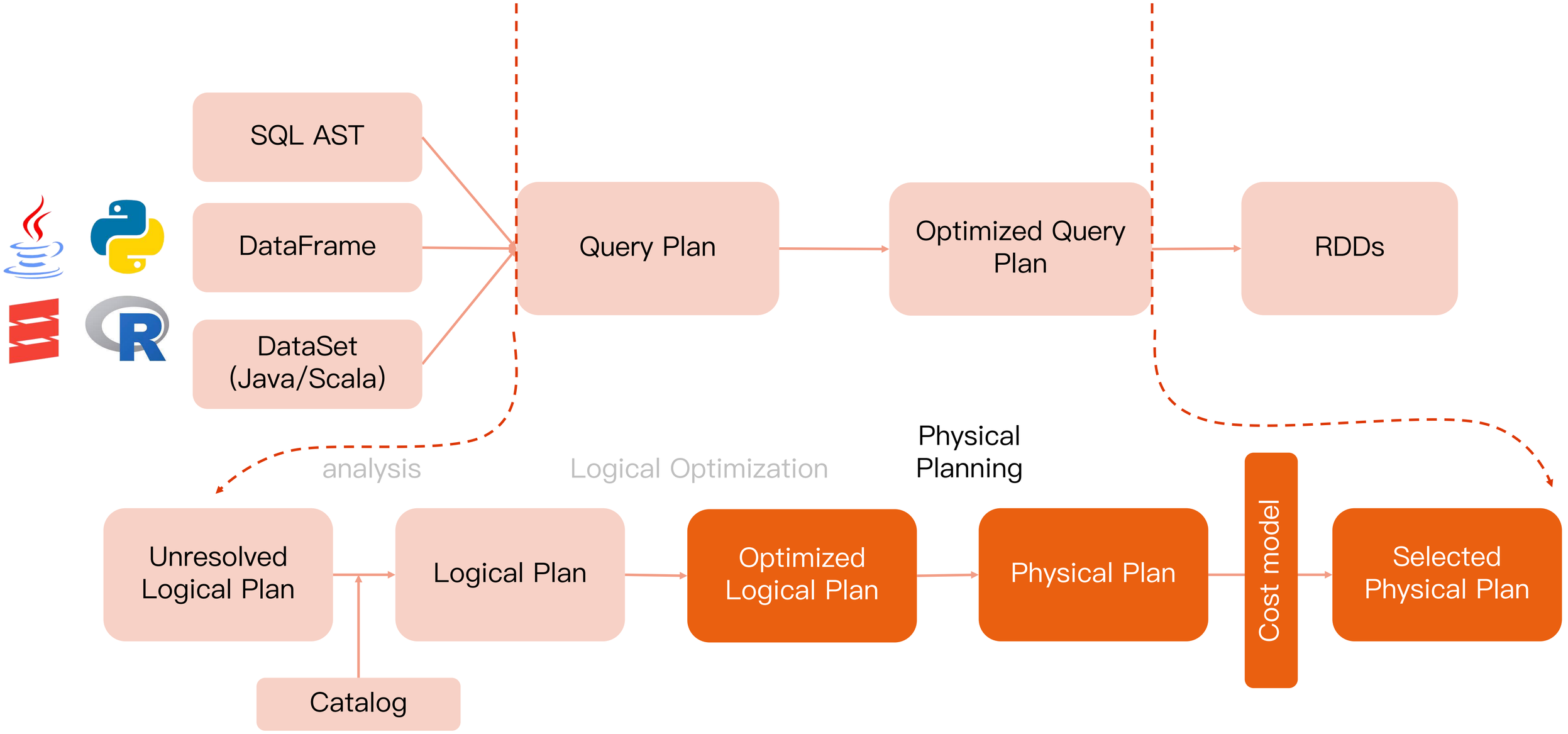
- LeafExecNode 叶子节点 主要和数据源相关，用户创建 RDD。
- UnaryExecNode 一元节点 主要是针对 RDD的转换操作。
- BinaryExecNode 二元节点 join 操作就属于这类。
- 其他类型的节点。

在物理算子树中， LeafExecNode 是创建一个 RDD 开始，遍历 Tree 过程中每个非叶子节点做一次 Transformation，通过 execute 函数转换成新的 RDD，最终会执行 Action 算子把结果返回给用户。

SparkPlanner

1. Sql text 经过 SqlParser 解析成 Unresolved LogicalPlan
2. Analyzer 模块结合 Catalog 进行绑定，生成 Resolved LogicalPlan
3. Optimizer 模块对 resolved LogicalPlan 进行优化,生成 Optimized LogicalPlan
4. SparkPlanner 将 LogicalPlan 转换成 PhysicalPlan（SparkPlan）

SparkPlanner

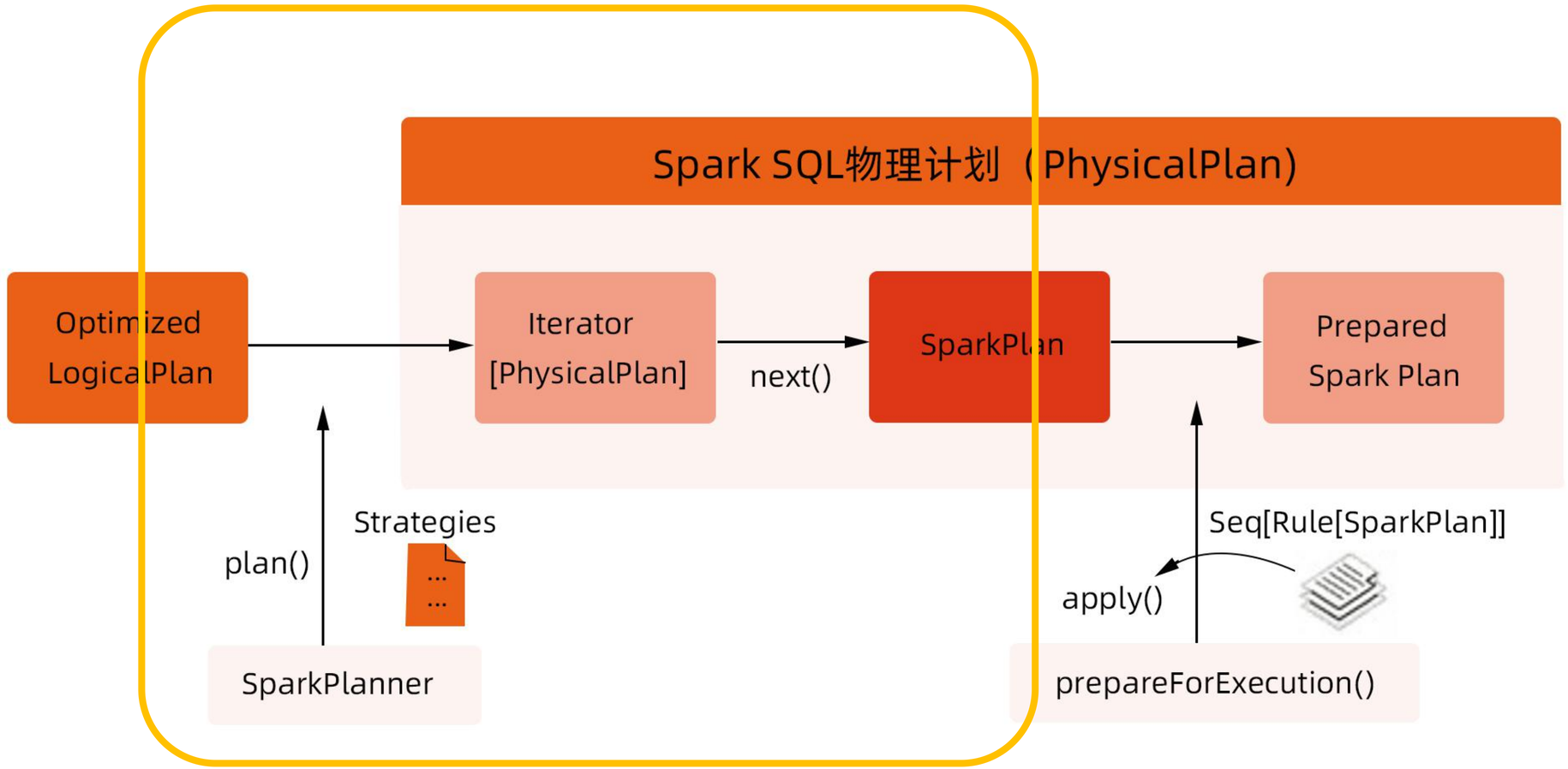


Strategies

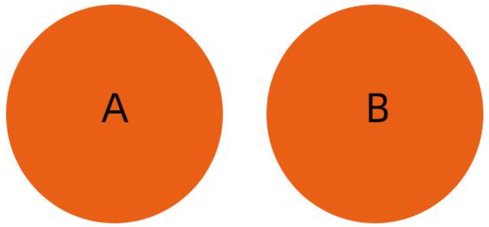
- SparkPlanner 也是类似 Analyzer和Optimizer，使用类似基于规则（Rules 和 Batches）的方式对逻辑计划树进行转换，这里的规则称为策略（Strategy）。
- SparkPlanner 通过在 Optimizer LogicalPlan 树上应用策略（Strategy），从而生成 SparkPlan列表，即 Iterator[PhysicalPlan]。
- SparkPlanner 中定义了一组 Strategy，称为 Strategies，类似生成逻辑执行计划中的 Batches。

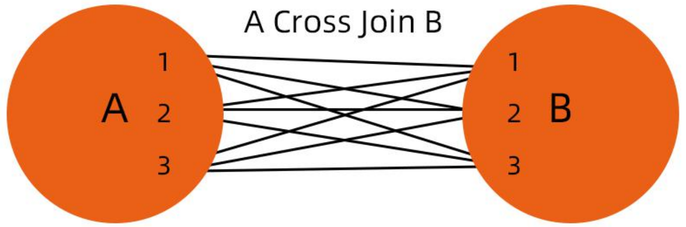
```
override def strategies: Seq[Strategy] =
  experimentalMethods.extraStrategies ++
    extraPlanningStrategies ++ (
    LogicalQueryStageStrategy ::
    PythonEvals ::
    new DataSourceV2Strategy(session) ::
    FileSourceStrategy ::
    DataSourceStrategy ::
    SpecialLimits ::
    Aggregation ::
    Window ::
    JoinSelection ::
    InMemoryScans ::
    BasicOperators ::
    CompactDataSourceTable :: Nil)
```

Strategies

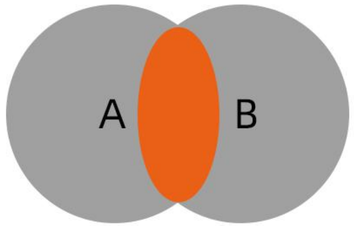

Spark SQL物理计划（PhysicalPlan)
Optimized
LogicalPlan
Iterator
[PhysicalPlan]
next()
SparkPlan
Prepared
Spark Plan
Strategies
plan()
...
...
Seq[Rule[SparkPlan]]
apply()
SparkPlanner
prepareForExecution()

JoinSelection

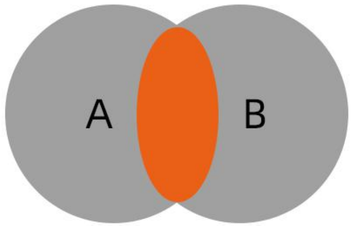




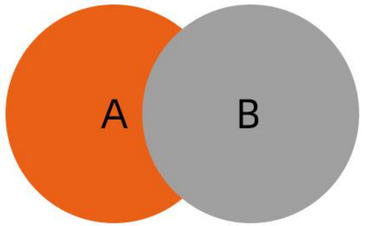
A(Ibber)Join B



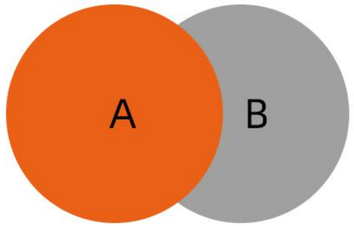
A LeftSemi Join B



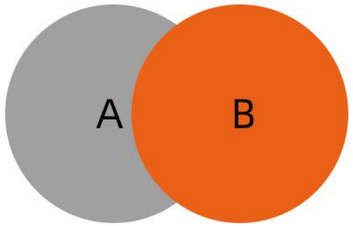
A Left Anti Join B



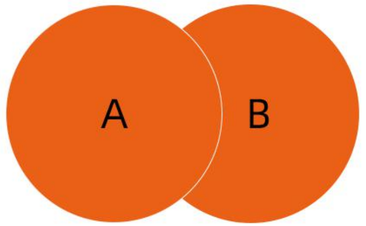
A Left/LeftOuter Join B



A Right/RightOuter Join B



A Full/Outer/FullOuter Join B



build table 的选择

Spark SQL 主要有三种实现 join 的策略，分别是 broadcast hash join、shuffle hash join、sort merge join。对应物理算子就是 BroadcastHashJoinExec、ShuffledHashJoinExec、SortMergeJoinExec。

join 的两边分别是流式表（streamed side）和构建表（build side）。

构建表被作为查找表数据结构，流式表作为顺序遍历的数据结构。通常通过一条条迭代流式表中数据，并在构建表中查找与当前流式表数据 join 键值相同的数据来实现两表的 join。

JoinSelection 的第一步就是 build side 的选择。

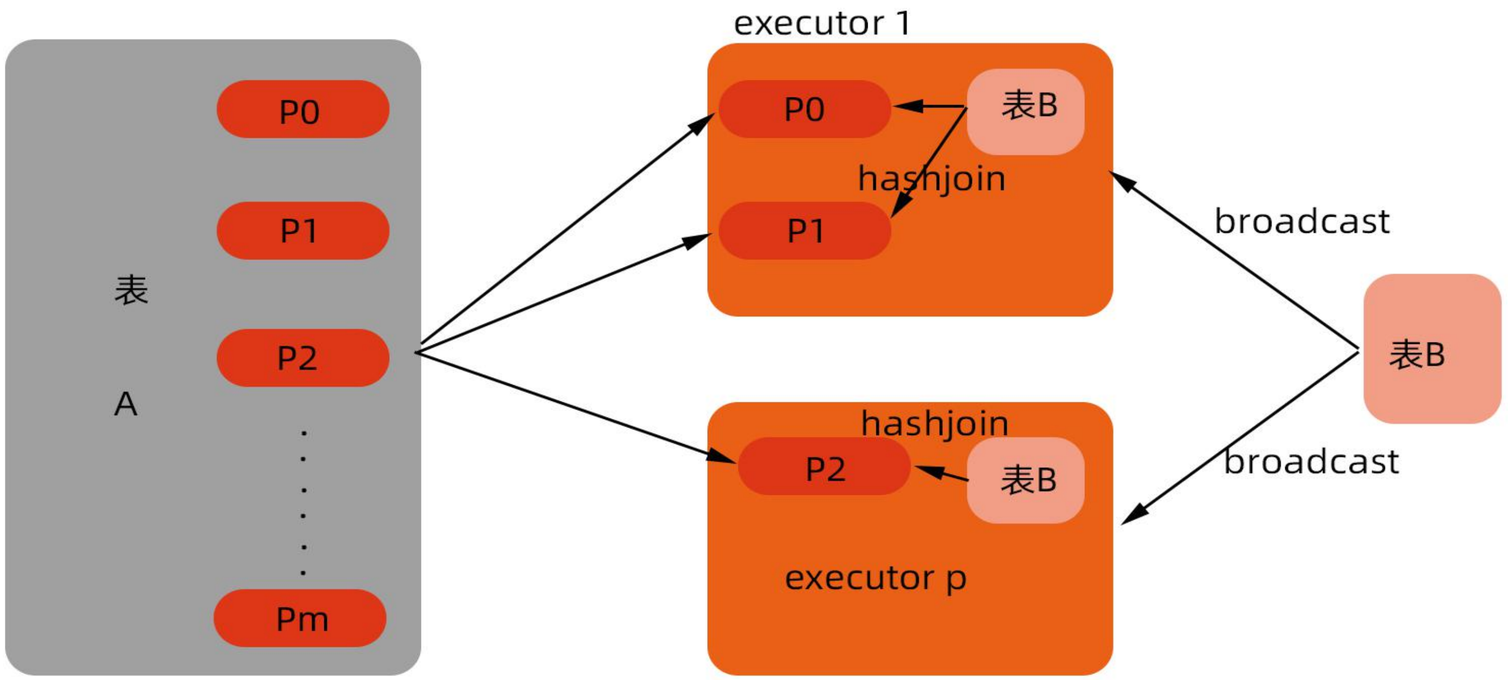
- Hash join 将两表之中较小的那一个构建哈希表，这个小表就是 build table。大表叫做 probe table。
- 当 join 类型为 inner-like（包含 inner join 与 cross join 两种）或 right outer join 时，左表才有可能作为 build table。而在 join 类型为 inner-like 或者 left outer/semi/anti join 时，右表有可能作为 build table。

join 策略的选择

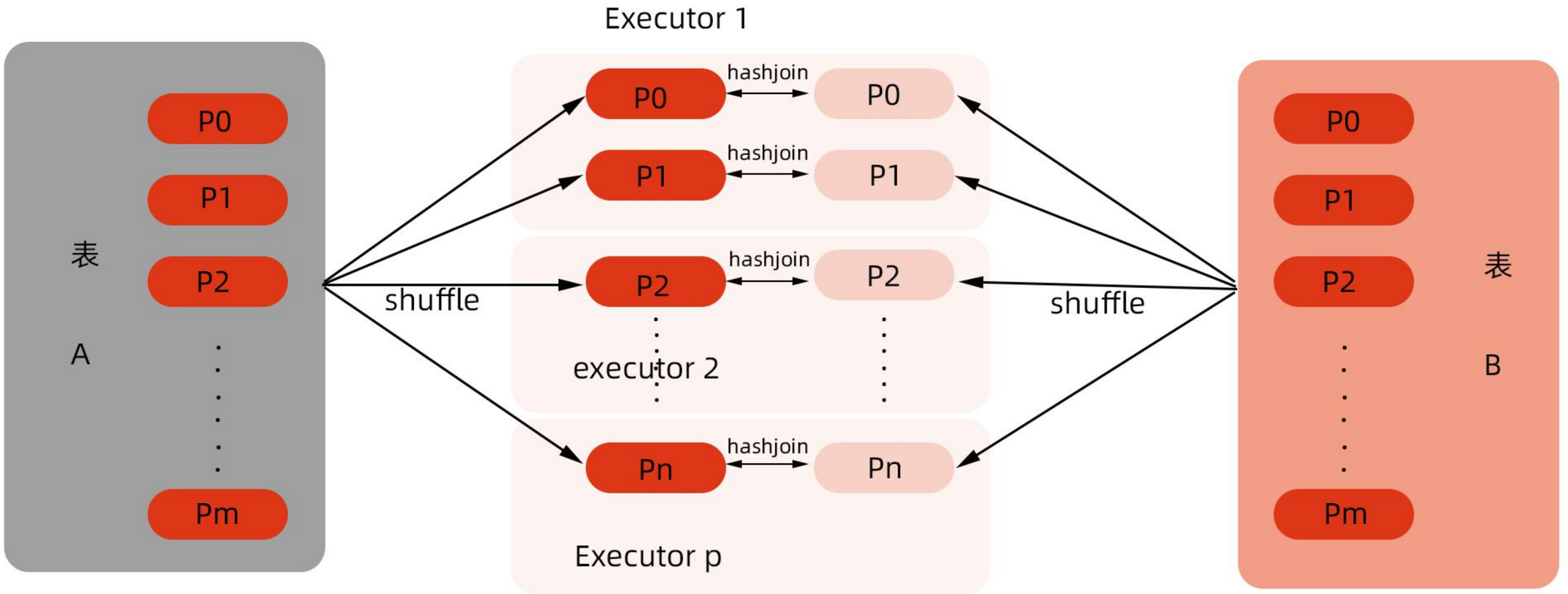
策略的选择会按照效率从高到低的优先级来排：

1. broadcast hash join
    a. 先根据 broadcast hint 来判断。
    b. 其次是广播阈值。
2. hash join
    a. spark.sql.join.preferSortMergeJoin 配置项为 false。
    b. 右表能够作为 build table，构建本地 HashMap（先右后左）。
    c. 右表的数据量比左表小很多（3倍）。
3. sort merge join
    a. 如果上面两种策略都不符合，并且参与 join 的 key 是可以排序的。

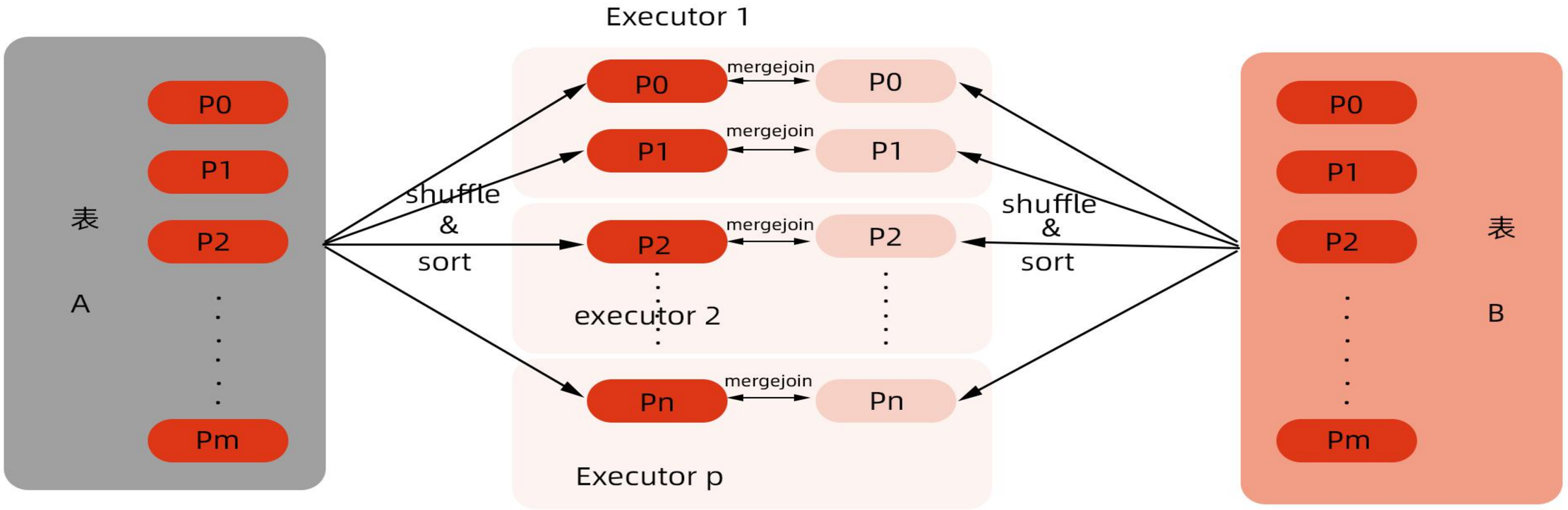
broadcast hash join



shuffled hash join



sort merge join



join 策略的选择（续）

前面提到的策略选择都是在等值连接时，即 join 条件一般为 A.id = B.id，如果是非等值连接，如 join 条件为 A.id > B.id，则只能使用 nest loop join（即二重循环扫描+比对）或者计算两表的笛卡尔积：

broadcast nest Join

- 广播阈值。

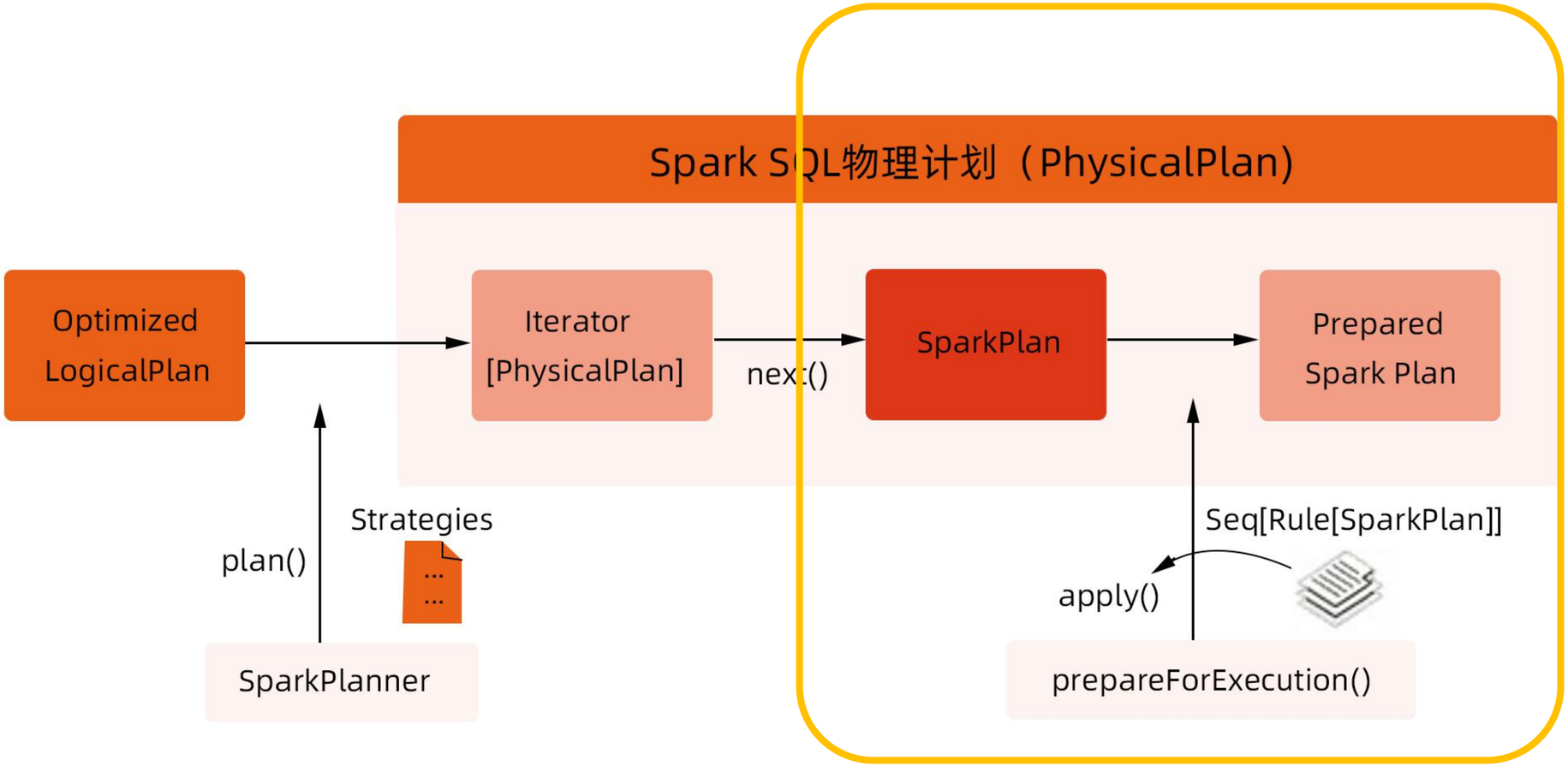
cartesian join

- join 类型是 inner join 或者 cross join。
- 广播阈值。

broadcast nest join

- 如果上面两种策略都不符合，则将较小的表 broadcast。

prepareForExecution



prepareForExecution()

1. SQL text 经过 SqlParser 解析成 Unresolved LogicalPlan。
2. Analyzer 模块结合 Catalog 进行绑定，生成 Resolved LogicalPlan。
3. Optimizer 模块对 resolved LogicalPlan 进行优化，生成 Optimized LogicalPlan。
4. SparkPlanner 将 LogicalPlan 转换成 PhysicalPlan（SparkPlan）。
5. prepareForExecution() 将 PhysicalPlan 转换成可执行物理计划。

prepareForExecution

- 经过 SparkPlanner 和 Strategies，相当于生成了物理计划数组，之后获取第一条便是计算需要的物理计划树了，但是在真正提交之前，还有一步 prepareForExecution。
- prepareForExecution 的主要的目的是为了优化物理计划，使之满足 shuffle 数据分布，数据排序和内部行格式等。

```
// executedPlan should not be used to initialize any SparkPlan. It should be
// only used for execution.
lazy val executedPlan: SparkPlan = {
  // We need to materialize the optimizedPlan here, before tracking the planning phase, to ensure
  // that the optimization time is not counted as part of the planning phase.
  assertOptimized()
  executePhase(QueryPlanningTracker.PLANNING) {
    // clone the plan to avoid sharing the plan instance between different stages like analyzing,
    // optimizing and planning.
    QueryExecution.prepareForExecution(preparations, sparkPlan.clone())
  }
}
```

preparations

```
private[execution] def preparations(
    sparkSession: SparkSession,
    adaptiveExecutionRule: Option[InsertAdaptiveSparkPlan] = None): Seq[Rule[SparkPlan]] = {
  // `AdaptiveSparkPlanExec` is a leaf node. If inserted, all the following rules will be no-op
  // as the original plan is hidden behind `AdaptiveSparkPlanExec`.
  adaptiveExecutionRule.toSeq ++
  Seq(
    PlanDynamicPruningFilters(sparkSession),
    PlanSubqueries(sparkSession),
    EnsureRequirements,
    // `RemoveRedundantSorts` needs to be added after `EnsureRequirements` to guarantee the same
    // number of partitions when instantiating PartitioningCollection.
    RemoveRedundantSorts,
    EnsureRepartitionForWriting,
    EliminateShuffleExec,
    ApplyColumnarRulesAndInsertTransitions(sparkSession.sessionState.columnarRules),
    CollapseCodegenStages(),
    ReuseExchange,
    ReuseSubquery
  )
}
```

EnsureRequirements

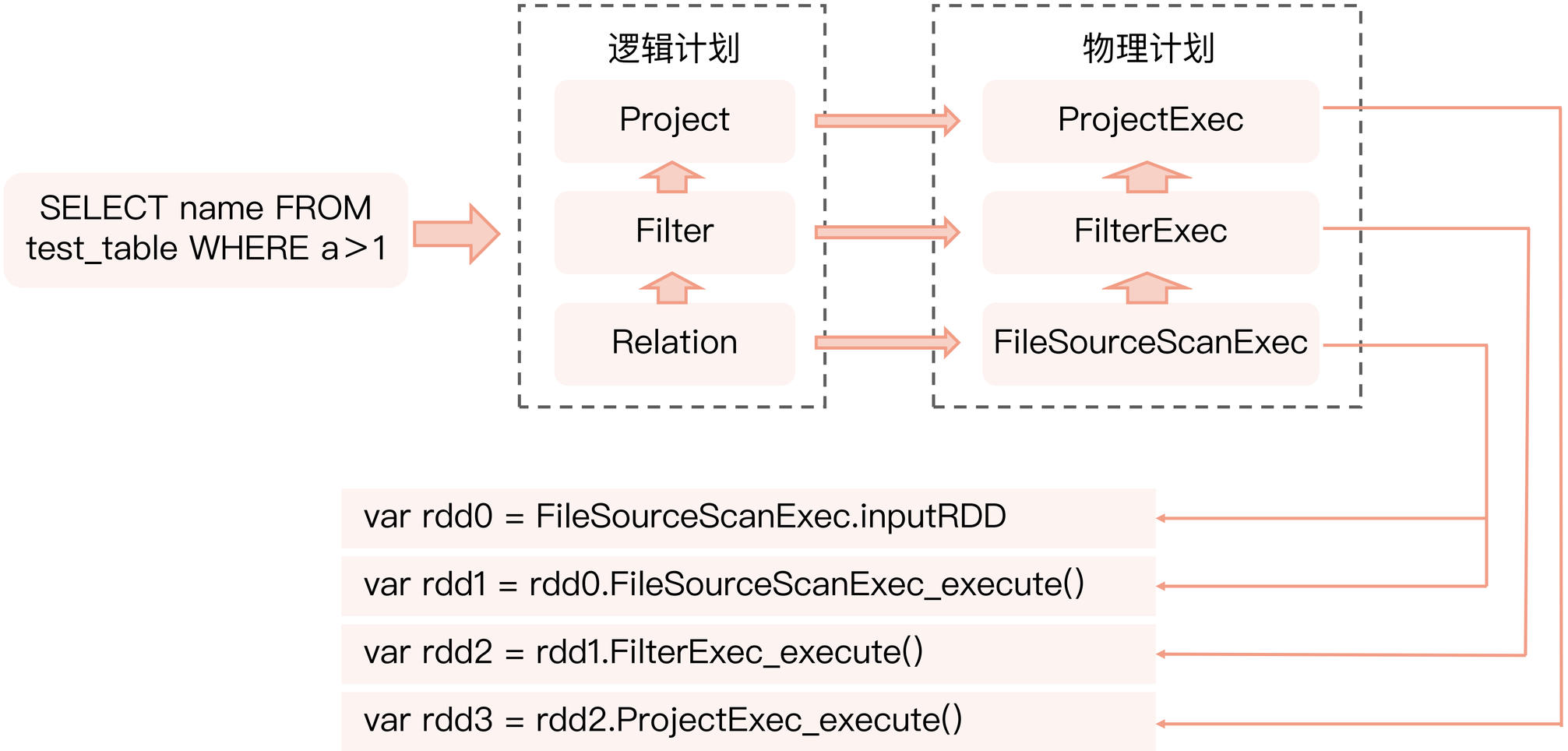
1. 添加 ExChange 节点，遍历子节点，会依次判断子节点的分区方式（partitioning）是否满足所需的数据分布（distribution）。如果不满足，则考虑是否能以广播的形式来满足，如果不行的话就添加 ShuffleExChangeExec 节点，之后会查看所要求的子节点输出（requiredChildDistributions），是否有特殊需求，并且要求有相同的分区数，针对这类对子节点有特殊需求的情况，则会查看每个子节点的输出分区数目，如果匹配不做改变，不然会添加 ShuffleExchangeExec 节点。
2. 查看 requiredChildOrderings 针对排序有特殊需求的添加 SortExec 节点。

EnsureRequirements（源码）

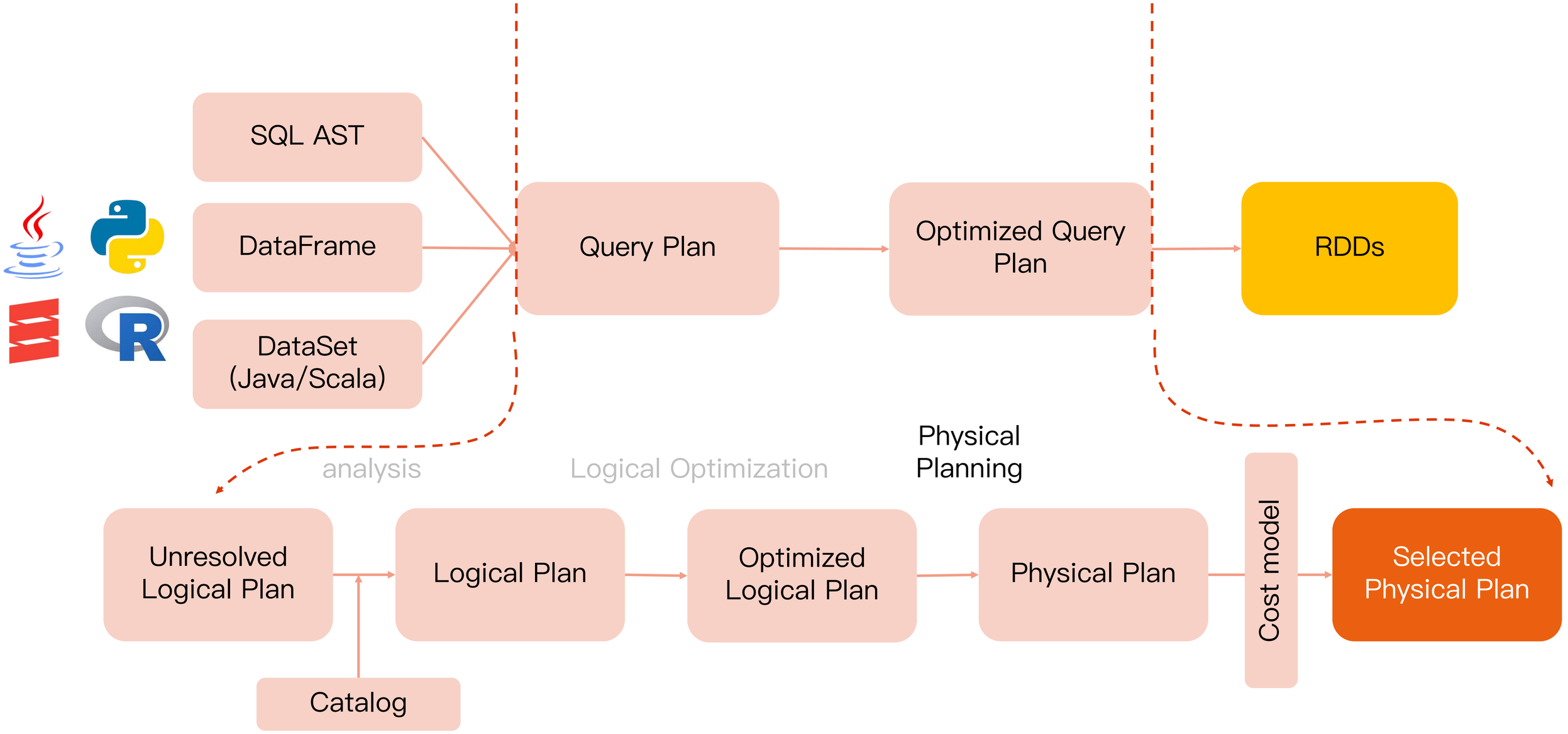
（源码学习）

Execution

Spark Plan 的 Execution 方式均为 调用其 execute() 方法生成 RDD。

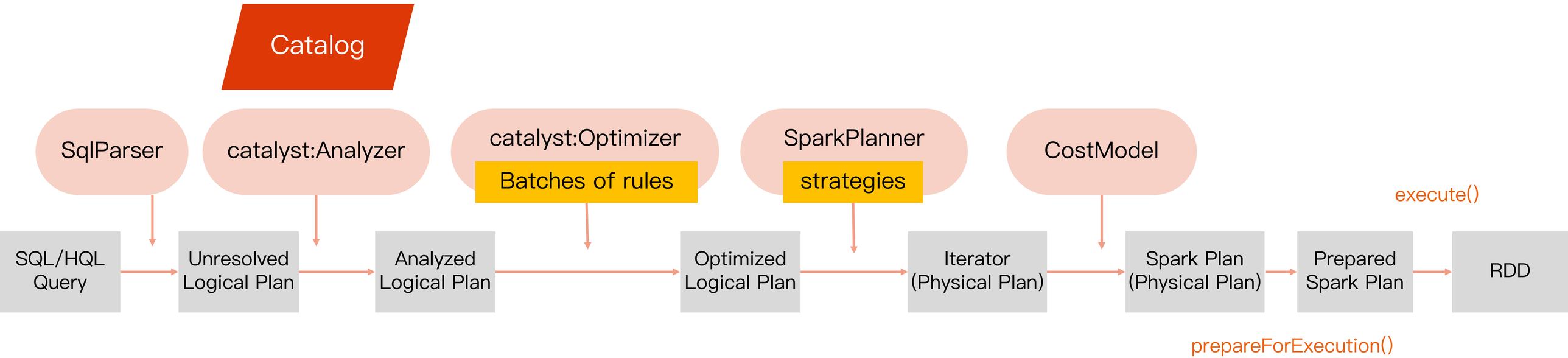


Execution


SQL AST
DataFrame
DataSet
(Java/Scala)
Query Plan
Optimized Query
Plan
RDDs
analysis
Logical Optimization
Physical
Planning
Unresolved
Logical Plan
Logical Plan
Optimized
Logical Plan
Physical Plan
Cost model
Selected
Physical Plan
Catalog

Execute()

1. Sql text 经过 SqlParser 解析成 Unresolved LogicalPlan。
2. Analyzer 模块结合 Catalog 进行绑定，生成 Resolved LogicalPlan。
3. Optimizer 模块对 resolved LogicalPlan 进行优化，生成 Optimized LogicalPlan。
4. SparkPlanner 将 LogicalPlan 转换成 PhysicalPlan（SparkPlan）。
5. prepareForExecution() 将 PhysicalPlan 转换成可执行物理计划。
6. 使用 execute() 执行可执行物理计划。


Catalog
SqlParser
catalyst:Analyzer
catalyst:Optimizer
Batches of rules
SparkPlanner
strategies
CostModel
execute()
SQL/HQL Query
Unresolved Logical Plan
Analyzed Logical Plan
Optimized Logical Plan
Iterator (Physical Plan)
Spark Plan (Physical Plan)
Prepared Spark Plan
RDD
prepareForExecution()

THANKS