Spark SQL 内部原理 RBO

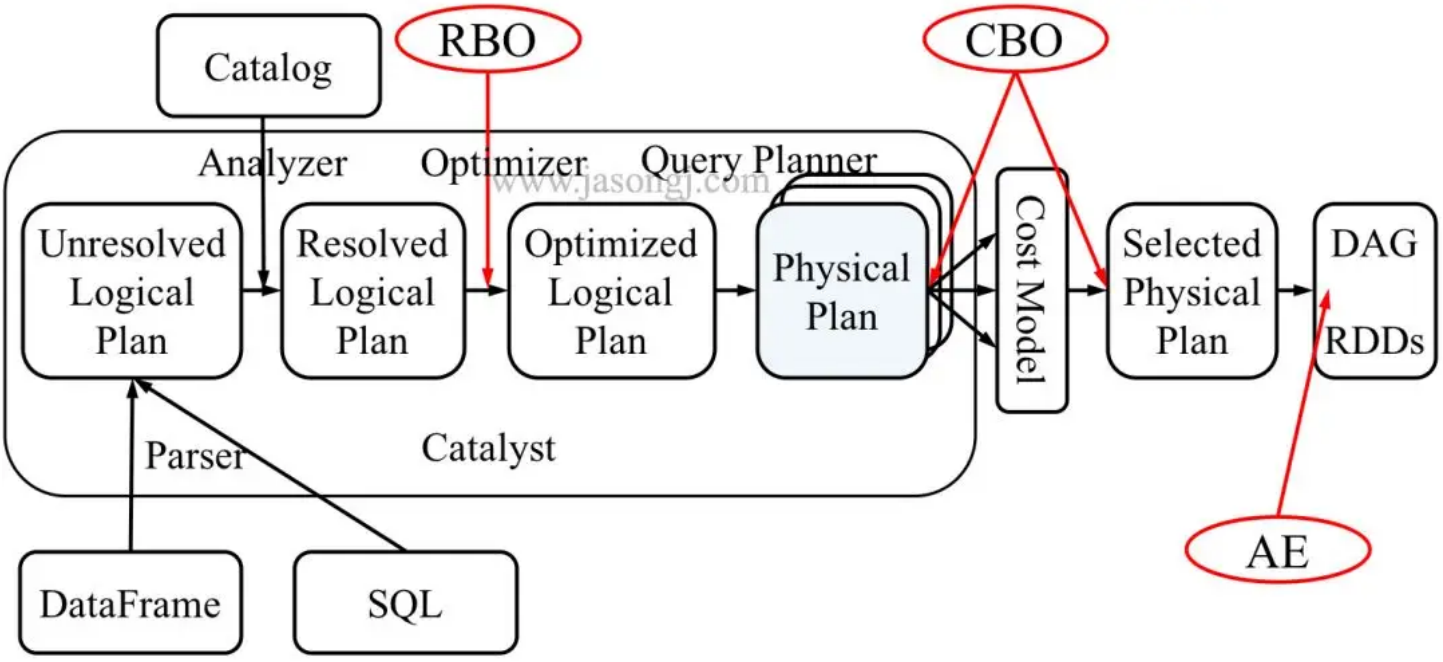
Original 郭俊 Jason Guo 大数据架构 2018年09月09日 19:14

Spark SQL内部原理 RBO

本文结合案例详述了 Spark SQL 的工作原理，包括但不限于 Parser，Analyzer，Optimizer，Rule-based optimization等内容。

Spark SQL 架构。。。。。。。。。。

Spark SQL 的整体架构如下图所示



从上图可见，无论是直接使用 SQL 语句还是使用 DataFrame，都会经过如下步骤转换成 DAG 对 RDD 的操作

- Parser 解析 SQL，生成 Unresolved Logical Plan
- 由 Analyzer 结合 Catalog 信息生成 Resolved Logical Plan
- Optimizer 根据预先定义好的规则对 Resolved Logical Plan 进行优化并生成 Optimized Logical Plan
- Query Planner 将 Optimized Logical Plan 转换成多个 Physical Plan
- CBO 根据 Cost Model 算出每个 Physical Plan 的代价并选取代价最小的 Physical Plan 作为最终的 Physical Plan
- Spark 以 DAG 的方法执行上述 Physical Plan

- 在执行 DAG 的过程中，Adaptive Execution 根据运行时信息动态调整执行计划从而提高执行效率

Parser

Spark SQL 使用 Antlr 进行记法和语法解析，并生成 UnresolvedPlan。

当用户使用 SparkSession.sql(sqlText：String) 提交 SQL 时，SparkSession 最终会调用 SparkSqlParser 的 parsePlan 方法。该方法分两步

- 使用 Antlr 生成的 SqlBaseLexer 对 SQL 进行词法分析，生成 CommonTokenStream
- 使用 Antlr 生成的 SqlBaseParser 进行语法分析，得到 LogicalPlan

现在两张表，分别定义如下

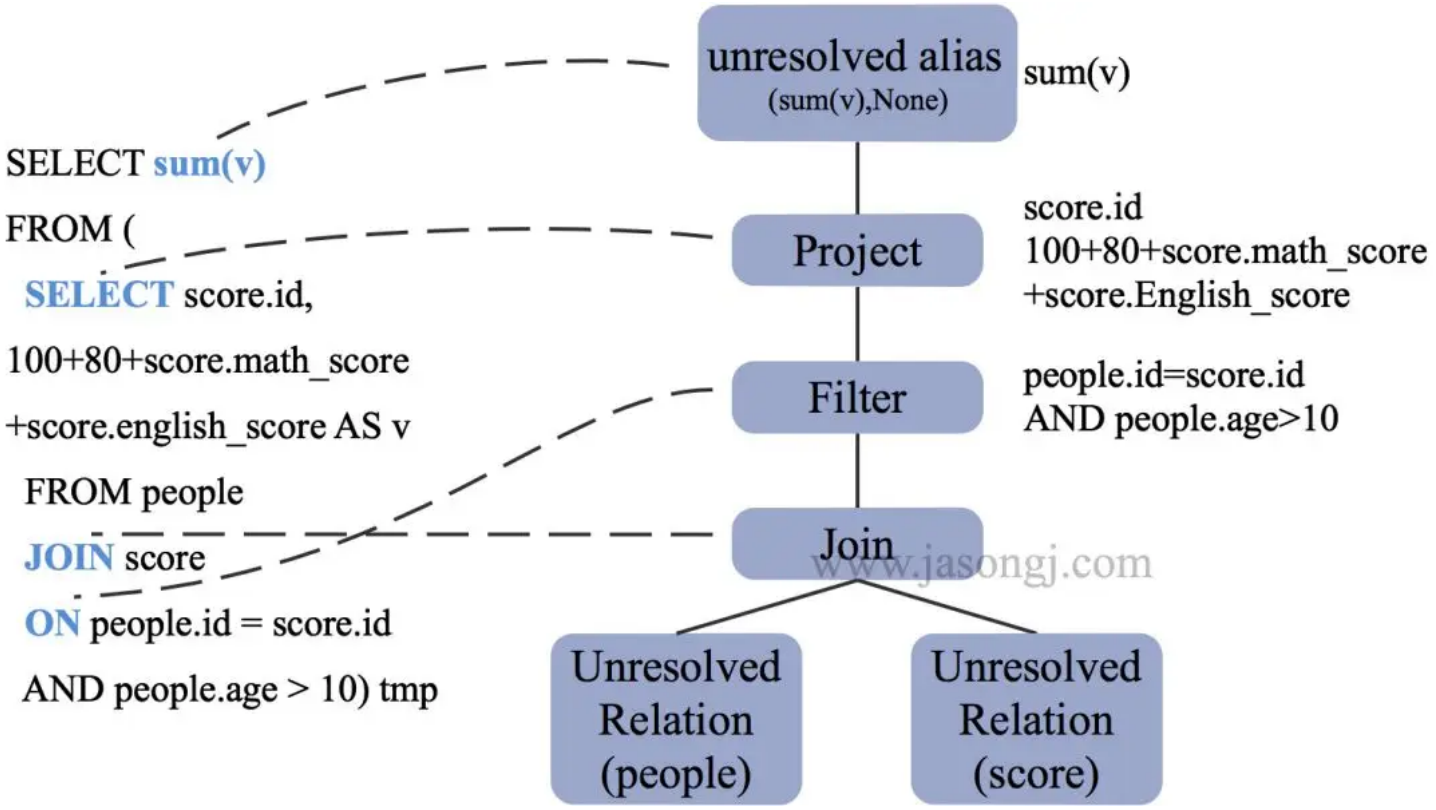
```
CREATE TABLE score (
  id INT,
  math_score INT,
  english_score INT
)
```

```
CREATE TABLE people (
  id INT,
  age INT,
  name INT
)
```

对其进行关联查询如下

```
SELECT sum(v)
FROM (
  SELECT score.id,
    100 + 80 + score.math_score + score.english_score AS v
  FROM people
  JOIN score
  ON people.id = score.id
  AND people.age > 10
) tmp
```

生成的 UnresolvedPlan 如下图所示



从上图可见

- 查询涉及的两张表，被解析成了两个 UnresolvedRelation，也即只知道这们是两张表，却并不知道它们是 EXTERNAL TABLE 还是 MANAGED TABLE，也不知道它们的数据存在哪儿，更不知道它们的表结构如何
- sum(v) 的结果未命名
- Project 部分只知道是选择出了属性，却并不知道这些属性属于哪张表，更不知道其数据类型
- Filter 部分也不知道数据类型

Spark SQL 解析出的 UnresolvedPlan 如下所示

```
== Parsed Logical Plan ==
'Project [unresolvedalias('sum('v), None)]
+- 'SubqueryAlias tmp
   +- 'Project ['score.id, (((100 + 80) + 'score.math_score) + 'score.english_score
      +- 'Filter (('people.id = 'score.id) && ('people.age > 10))
         +- 'Join Inner
            :- 'UnresolvedRelation `people`
            +- 'UnresolvedRelation `score`
```

Analyzer

- 从 Analyzer 的构造方法可见
- Analyzer 持有一个 SessionCatalog 对象的引用
- Analyzer 继承自 RuleExecutor[LogicalPlan]，因此可对 LogicalPlan 进行转换

```
class Analyzer(
    catalog: SessionCatalog,
    conf: SQLConf,
    maxIterations: Int)
  extends RuleExecutor[LogicalPlan] with CheckAnalysis {
```

Analyzer 包含了如下的转换规则

```
lazy val batches: Seq[Batch] = Seq(
    Batch("Hints", fixedPoint,
      new ResolveHints.ResolveBroadcastHints(conf),
      ResolveHints.RemoveAllHints),
    Batch("Simple Sanity Check", Once,
      LookupFunctions),
    Batch("Substitution", fixedPoint,
      CTESubstitution,
      WindowsSubstitution,
      EliminateUnions,
      new SubstituteUnresolvedOrdinals(conf)),
    Batch("Resolution", fixedPoint,
      ResolveTableValuedFunctions ::
      ResolveRelations ::
      ResolveReferences ::
      ResolveCreateNamedStruct ::
      ResolveDeserializer ::
      ResolveNewInstance ::
      ResolveUpCast ::
      ResolveGroupingAnalytics ::
      ResolvePivot ::
      ResolveOrdinalInOrderByAndGroupBy ::
      ResolveAggAliasInGroupBy ::
      ResolveMissingReferences ::
      ExtractGenerator ::
```

```
        ResolveGenerate ::
        ResolveFunctions ::
        ResolveAliases ::
        ResolveSubquery ::
        ResolveSubqueryColumnAliases ::
        ResolveWindowOrder ::
        ResolveWindowFrame ::
        ResolveNaturalAndUsingJoin ::
        ExtractWindowExpressions ::
        GlobalAggregates ::
        ResolveAggregateFunctions ::
        TimeWindowing ::
        ResolveInlineTables(conf) ::
        ResolveTimeZone(conf) ::
        ResolvedUuidExpressions ::
        TypeCoercion.typeCoercionRules(conf) ++
        extendedResolutionRules : _*),
      Batch("Post-Hoc Resolution", Once, postHocResolutionRules: _*),
      Batch("View", Once,
        AliasViewChild(conf)),
      Batch("Nondeterministic", Once,
        PullOutNondeterministic),
      Batch("UDF", Once,
        HandleNullInputsForUDF),
      Batch("FixNullability", Once,
        FixNullability),
      Batch("Subquery", Once,
        UpdateOuterReferences),
      Batch("Cleanup", fixedPoint,
        CleanupAliases)
    )
```

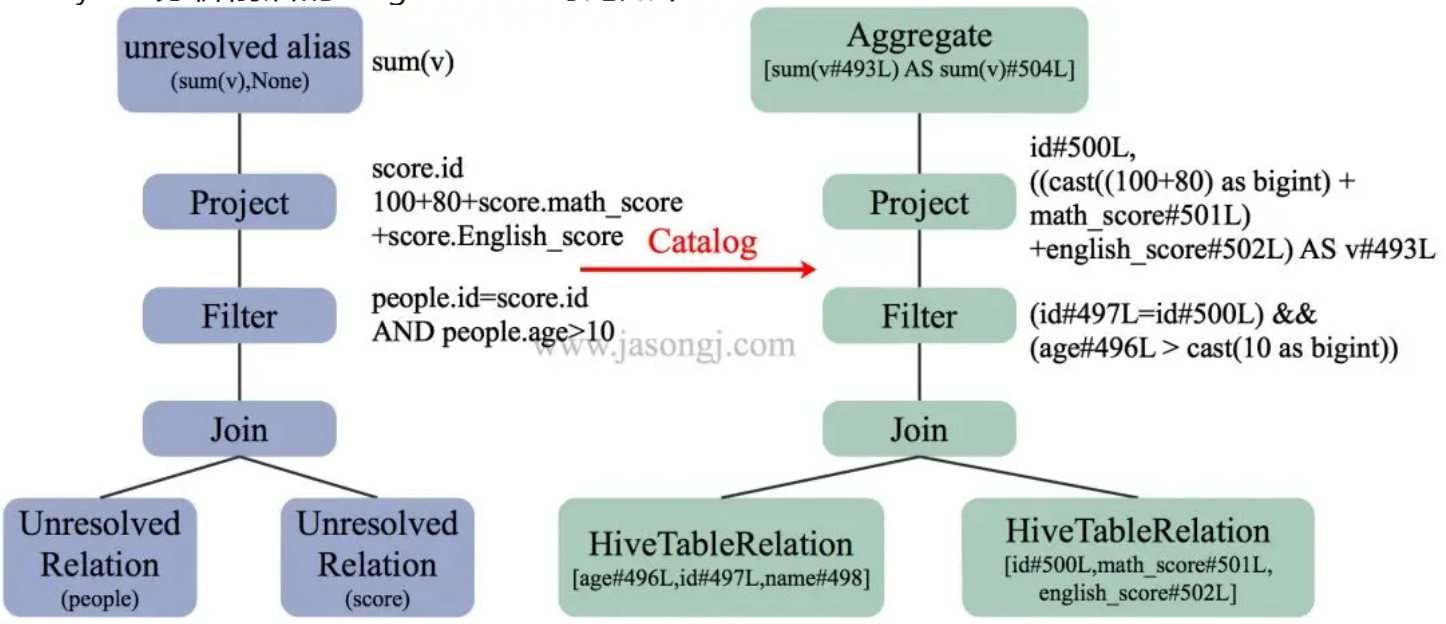
例如，ResolveRelations 用于分析查询用到的 Table 或 View。本例中 UnresolvedRelation (people) 与 UnresolvedRelation (score) 被解析为 HiveTableRelation (json.people) 与 HiveTableRelation (json.score)，并列出其各自包含的字段名。

经 Analyzer 分析后得到的 Resolved Logical Plan 如下所示

```
== Analyzed Logical Plan ==
sum(v): bigint
Aggregate [sum(cast(v#493 as bigint)) AS sum(v)#504L]
```

```
+- SubqueryAlias tmp
   +- Project [id#500, (((100 + 80) + math_score#501) + english_score#502) AS v#493]
      +- Filter ((id#496 = id#500) && (age#497 > 10))
         +- Join Inner
            :- SubqueryAlias people
            :  +- HiveTableRelation `jason`.`people`, org.apache.hadoop.hive.serde2.
            +- SubqueryAlias score
               +- HiveTableRelation `jason`.`score`, org.apache.hadoop.hive.serde2.l
```

Analyzer 分析前后的 LogicalPlan 对比如下



由上图可见，分析后，每张表对应的字段集，字段类型，数据存储位置都已确定。Project 与 Filter 操作的字段类型以及在表中的位置也已确定。

有了这些信息，已经可以直接将该 LogicalPlan 转换为 Physical Plan 进行执行。

但是由于不同用户提交的 SQL 质量不同，直接执行会造成不同用户提交的语义相同的不同 SQL 执行效率差距甚远。换句话说，如果要保证较高的执行效率，用户需要做大量的 SQL 优化，使用体验大大降低。

为了尽可能保证无论用户是否熟悉 SQL 优化，提交的 SQL 质量如何， Spark SQL 都能以较高效率执行，还需在执行前进行 LogicalPlan 优化。

Optimizer

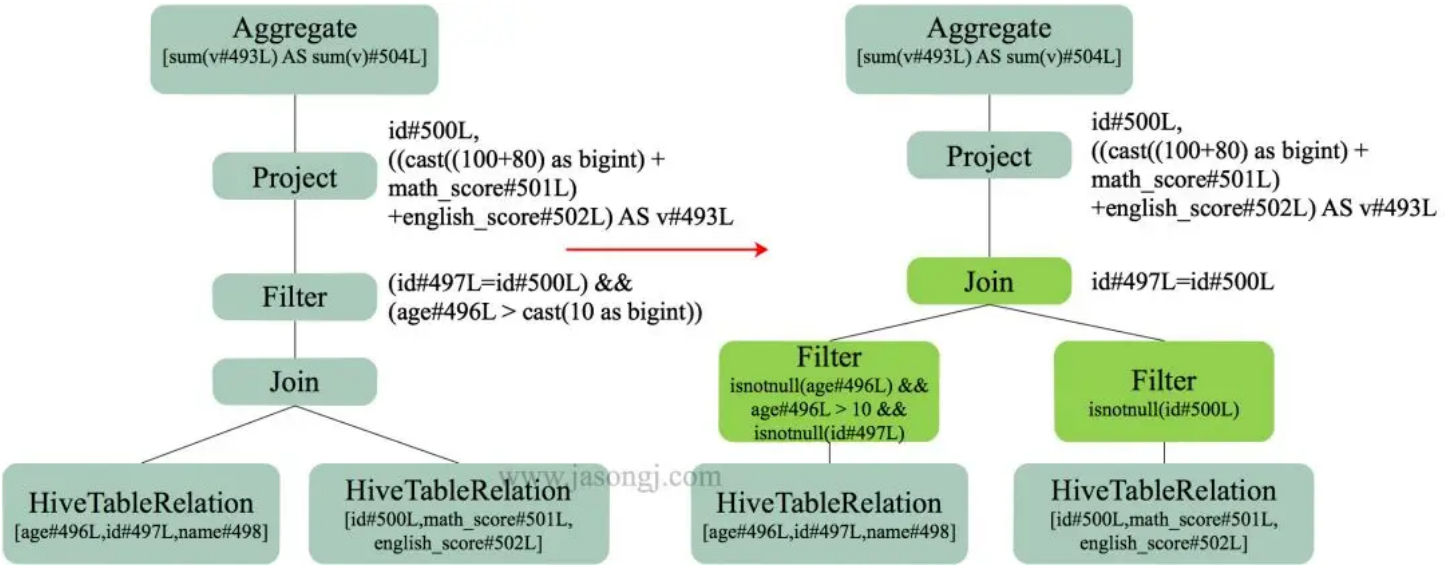
Spark SQL 目前的优化主要是基于规则的优化，即 RBO （ Rule-based optimization ）

- 每个优化以 Rule 的形式存在，每条 Rule 都是对 Analyzed Plan 的等价转换
- RBO 设计良好，易于扩展，新的规则可以非常方便地嵌入进 Optimizer
- RBO 目前已经足够好，但仍然需要更多规则来 cover 更多的场景
- 优化思路主要是减少参与计算的数据量以及计算本身的代价

PushdownPredicate

PushdownPredicate 是最常见的用于减少参与计算的数据量的方法。

前文中直接对两表进行 Join 操作，然后再 进行 Filter 操作。引入 PushdownPredicate 后，可先对两表进行 Filter 再进行 Join，如下图所示。

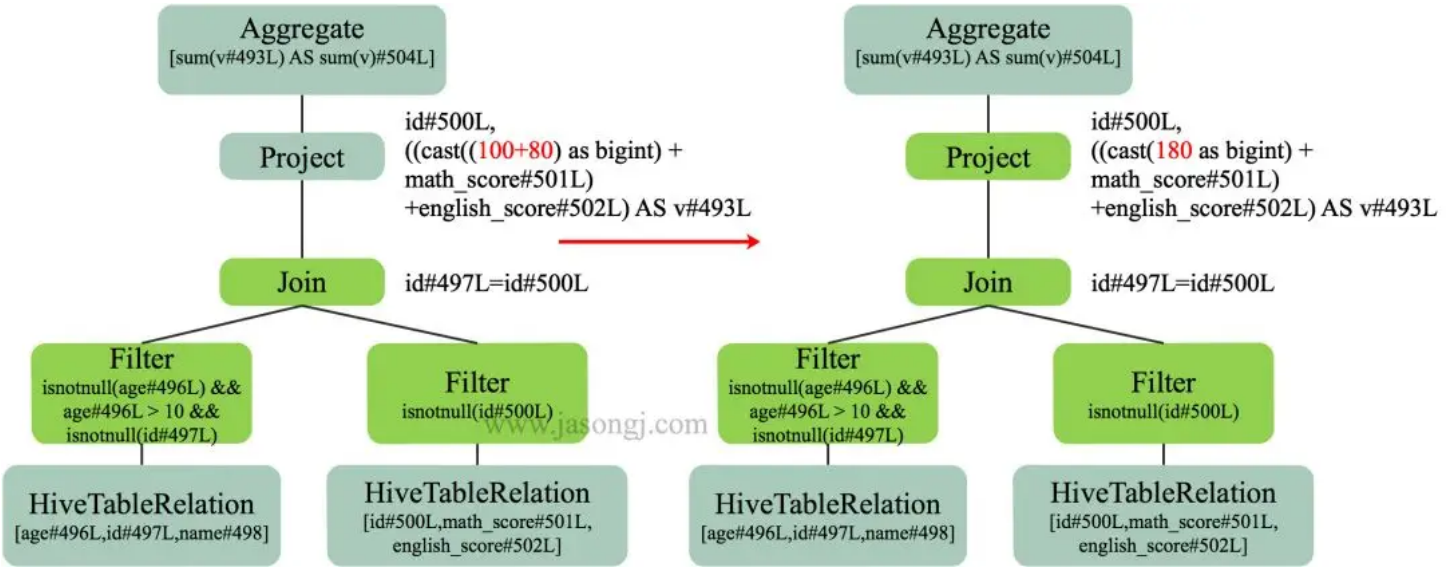


当 Filter 可过滤掉大部分数据时，参与 Join 的数据量大大减少，从而使得 Join 操作速度大大提高。

这里需要说明的是，此处的优化是 LogicalPlan 的优化，从逻辑上保证了将 Filter 下推后由于参与 Join 的数据量变少而提高了性能。另一方面，在物理层面，Filter 下推后，对于支持 Filter 下推的 Storage，并不需要将表的全量数据扫描出来再过滤，而是直接只扫描符合 Filter 条件的数据，从而在物理层面极大减少了扫描表的开销，提高了执行速度。

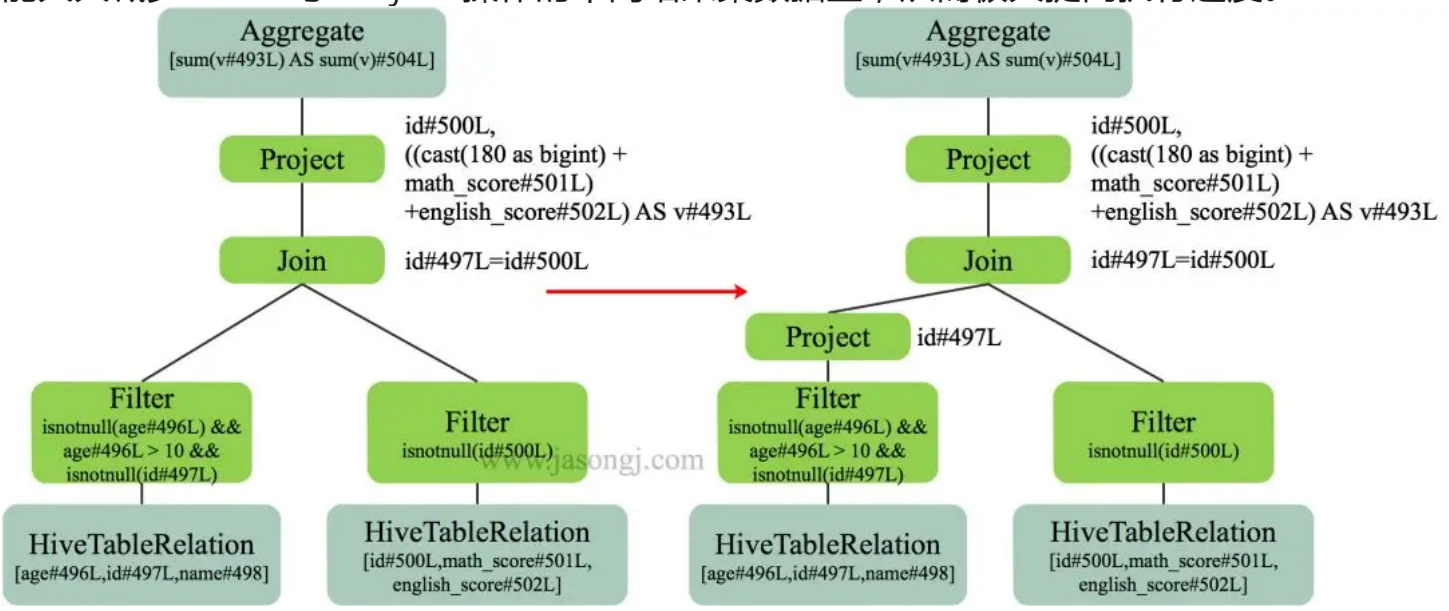
ConstantFolding

本文的 SQL 查询中，Project 部分包含了 *100 + 800 + match_score + english_score* 。如果不进行优化，那如果有一亿条记录，就会计算一亿次 100 + 80，非常浪费资源。因此可通过 ConstantFolding 将这些常量合并，从而减少不必要的计算，提高执行速度。



ColumnPruning

在上图中，Filter 与 Join 操作会保留两边所有字段，然后在 Project 操作中筛选出需要的特定列。如果能将 Project 下推，在扫描表时就只筛选出满足后续操作的最小字段集，则能大大减少 Filter 与 Project 操作的中间结果集数据量，从而极大提高执行速度。



这里需要说明的是，此处的优化是逻辑上的优化。在物理上，Project 下推后，对于列式存储，如 Parquet 和 ORC，可在扫描表时就只扫描需要的列而跳过不需要的列，进一步减少了扫描开销，提高了执行速度。

经过如上优化后的 LogicalPlan 如下

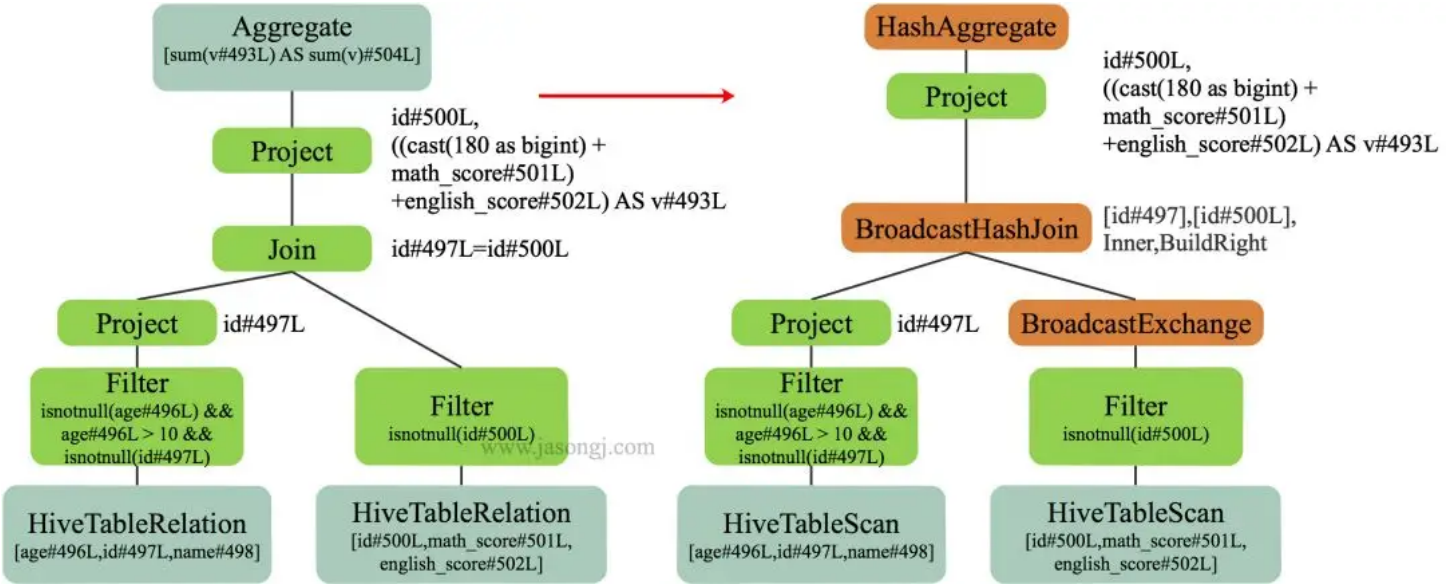
```
== Optimized Logical Plan ==
Aggregate [sum(cast(v#493 as bigint)) AS sum(v)#504L]
+- Project [((180 + math_score#501) + english_score#502) AS v#493]
   +- Join Inner, (id#496 = id#500)
      :- Project [id#496]
      :  +- Filter ((isnotnull(age#497) && (age#497 > 10)) && isnotnull(id#496))
      :     +- HiveTableRelation `jason`.`people`, org.apache.hadoop.hive.serde2.laz
```

```
      +- Filter isnotnull(id#500)
         +- HiveTableRelation `jason`.`score`, org.apache.hadoop.hive.serde2.lazy.La
```

SparkPlanner

得到优化后的 LogicalPlan 后，SparkPlanner 将其转化为 SparkPlan 即物理计划。

本例中由于 score 表数据量较小，Spark 使用了 BroadcastJoin。因此 score 表经过 Filter 后直接使用 BroadcastExchangeExec 将数据广播出去，然后结合广播数据对 people 表使用 BroadcastHashJoinExec 进行 Join。再经过 Project 后使用 HashAggregateExec 进行分组聚合。



总结

至此，一条 SQL 从提交到解析、分析、优化以及执行的完整过程就介绍完毕。

本文介绍的 Optimizer 属于 RBO，实现简单有效。它属于 LogicalPlan 的优化，所有优化均基于 LogicalPlan 本身的特点，未考虑数据本身的特点，也未考虑算子本身的代价。

下文将介绍 CBO，它充分考虑了数据本身的特点（如大小、分布）以及操作算子的特点（中间结果集的分布及大小）及代价，从而更好的选择执行代价最小的物理执行计划，即 SparkPlan。