SparkSQL – 从0到1认识Catalyst

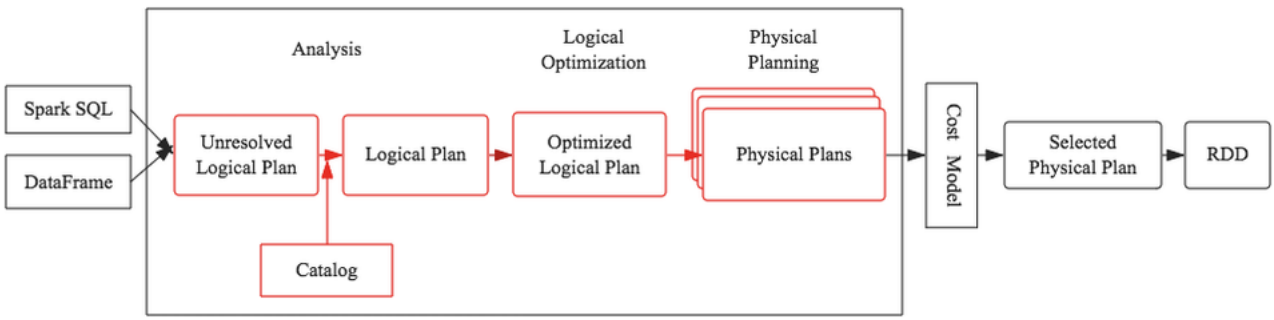
2017年3月1日 (http://hbasefly.com/2017/03/01/sparksql-catalyst/) 范欣欣 (http://hbasefly.com/author/libisthanksgmail-com/) Spark (http://hbasefly.com/category/spark/)

最近想来，大数据相关技术与传统型数据库技术很多都是相互融合、互相借鉴的。传统型数据库强势在于其久经考验的SQL优化器经验，弱势在于分布式领域的高可用性、容错性、扩展性等，假以时日，让其经过一定的改造，比如引入Paxos、raft等，强化自己在分布式领域的能力，相信一定会在大数据系统中占有一席之地。相反，大数据相关技术优势在于其天生的扩展性、可用性、容错性等，但其SQL优化器经验却基本全部来自于传统型数据库，当然，针对列式存储大数据SQL优化器会有一定的优化策略。

本文主要介绍SparkSQL的优化器系统Catalyst，上文讲到其设计思路基本都来自于传统型数据库，而且和大多数当前的大数据SQL处理引擎设计基本相同（Impala、Presto、Hive（Calcite）等），因此通过本文的学习也可以基本了解所有其他SQL处理引擎的工作原理。

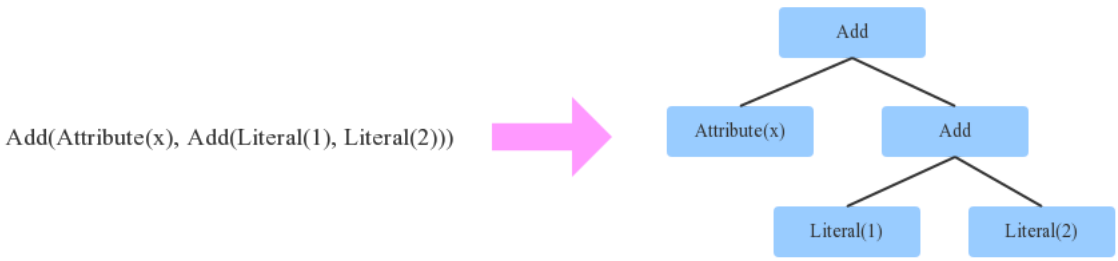
SQL优化器核心执行策略主要分为两个大的方向：基于规则优化（RBO）以及基于代价优化(CBO)，基于规则优化是一种经验式、启发式地优化思路，更多地依靠前辈总结出来的优化规则，简单易行且能够覆盖到大部分优化逻辑，但是对于核心优化算子Join却显得有点力不从心。举个简单的例子，两个表执行Join到底应该使用BroadcastHashJoin还是SortMergeJoin？当前SparkSQL的方式是通过手工设定参数来确定，如果一个表的数据量小于这个值就使用BroadcastHashJoin，但是这种方案显得很不优雅，很不灵活。基于代价优化就是为了解决这类问题，它会针对每个Join评估当前两张表使用每种Join策略的代价，根据代价估算确定一种代价最小的方案。

本文将会重点介绍基于规则的优化策略，后续文章会详细介绍基于代价的优化策略。下图中红色框框部分将是本文的介绍重点：



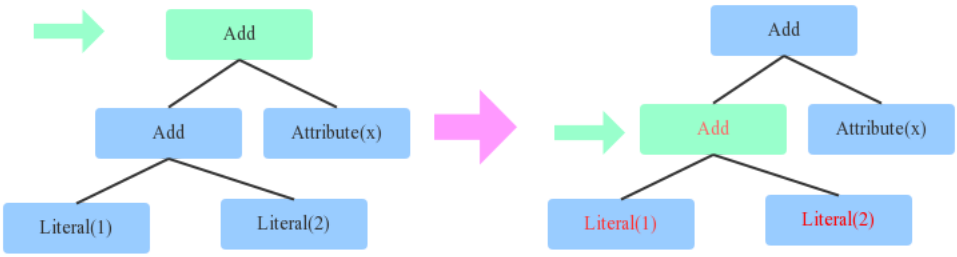
预备知识－Tree&Rule

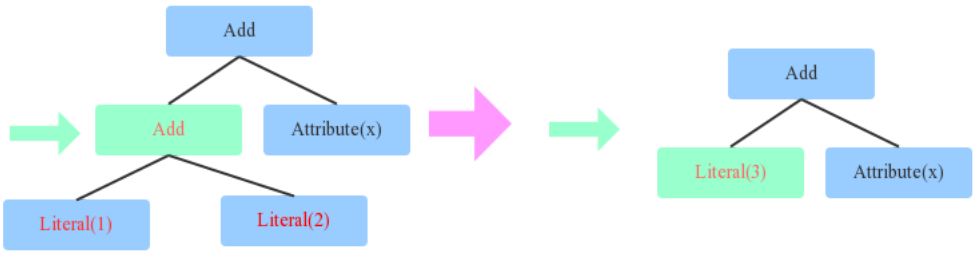
在介绍SQL优化器工作原理之前，有必要首先介绍两个重要的数据结构：Tree和Rule。相信无论对SQL优化器有无了解，都肯定知道SQL语法树这个概念，不错，SQL语法树就是SQL语句通过编译器之后会被解析成一棵树状结构。这棵树会包含很多节点对象，每个节点都拥有特定的数据类型，同时会有0个或多个孩子节点（节点对象在代码中定义为TreeNode对象），下图是个简单的示例：



如上图所示，箭头左边表达式有3种数据类型（Literal表示常量、Attribute表示变量、Add表示动作），表示x+(1+2)。映射到右边树状结构后，每一种数据类型就会变成一个节点。另外，Tree还有一个非常重要的特性，可以通过一定的规则进行等价变换，如下图：

```
expression.transform{
    case Add(Literal(x, IntergeType), Literal(y, IntergeType))  =>  Literal(x+y)
}
```





上图定义了一个等价变换规则(Rule)：两个Integer类型的常量相加可以等价转换为一个Integer常量，这个规则其实很简单，对于上文中提到的表达式x+(1+2)来说就可以转变为x+3。对于程序来讲，如何找到两个Integer常量呢？其实就是简单的二叉树遍历算法，每遍历到一个节点，就模式匹配当前节点为Add、左右子节点是Integer常量的结构，定位到之后将此三个节点替换为一个Literal类型的节点。

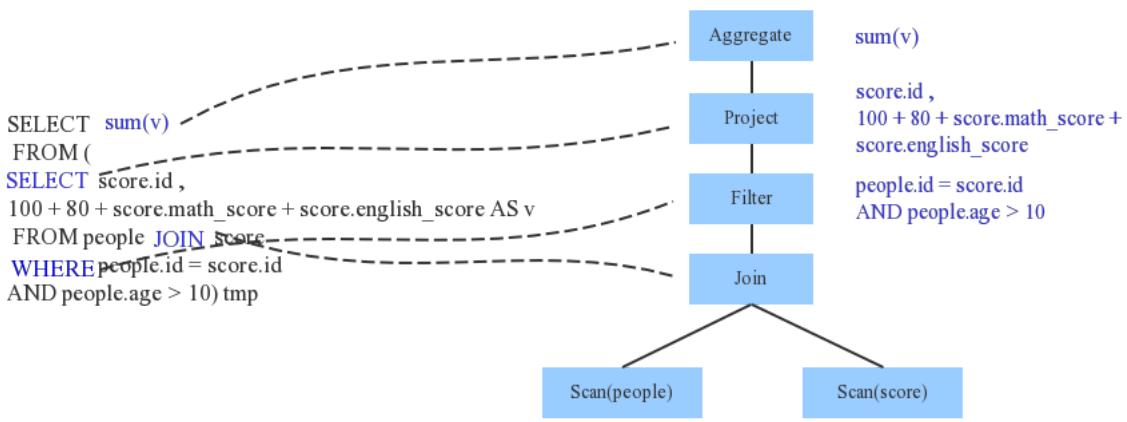
上面用一个最简单的示例来说明等价变换规则以及如何将规则应用于语法树。在任何一个SQL优化器中，通常会定义大量的Rule（后面会讲到），SQL优化器会遍历语法树中每个节点，针对遍历到的节点模式匹配所有给定规则（Rule），如果有匹配成功的，就进行相应转换，如果所有规则都匹配失败，就继续遍历下一个节点。

Catalyst工作流程

任何一个优化器工作原理都大同小异：SQL语句首先通过Parser模块被解析为语法树，此棵树称为Unresolved Logical Plan；Unresolved Logical Plan通过Analyzer模块借助于数据元数据解析为Logical Plan；此时再通过各种基于规则的优化策略进行深入优化，得到Optimized Logical Plan；优化后的逻辑执行计划依然是逻辑的，并不能被Spark系统理解，此时需要将此逻辑执行计划转换为Physical Plan；为了更好的对整个过程进行理解，下文通过一个简单示例进行解释。

Parser

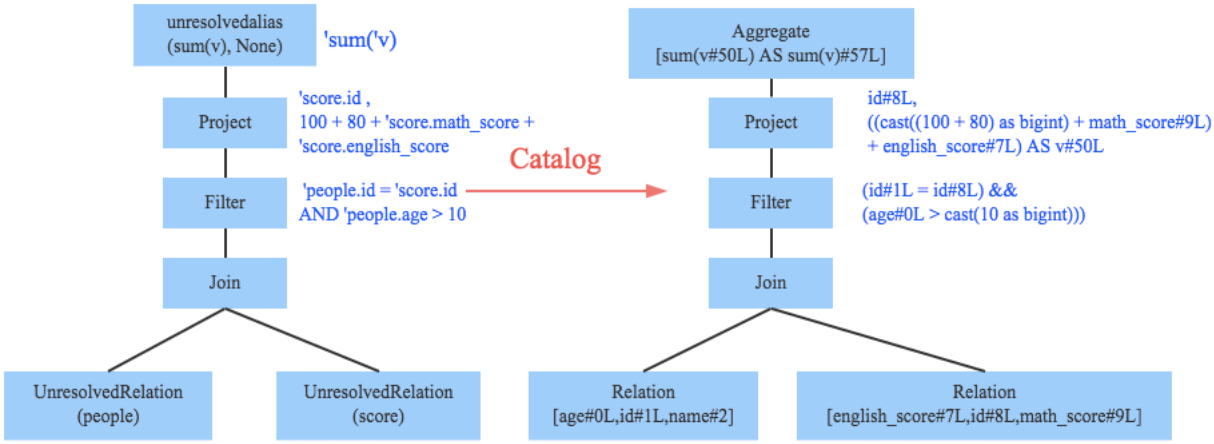
Parser简单来说是将SQL字符串切分成一个一个Token，再根据一定语义规则解析为一棵语法树。Parser模块目前基本都使用第三方类库ANTLR进行实现，比如Hive、 Presto、SparkSQL等。下图是一个示例性的SQL语句（有两张表，其中people表主要存储用户基本信息，score表存储用户的各种成绩），通过Parser解析后的AST语法树如右图所示：



Analyzer

通过解析后的逻辑执行计划基本有了骨架，但是系统并不知道score、sum这些都是些什么鬼，此时需要基本的元数据信息来表达这些词素，最重要的元数据信息主要包括两部分：表的Scheme和基本函数信息，表的scheme主要包括表的基本定义（列名、数据类型）、表的数据格式（Json、Text）、表的物理位置等，基本函数信息主要指类信息。

Analyzer会再次遍历整个语法树，对树上的每个节点进行数据类型绑定以及函数绑定，比如people词素会根据元数据表信息解析为包含age、id以及name三列的表，people.age会被解析为数据类型为int的变量，sum会被解析为特定的聚合函数，如下图所示：



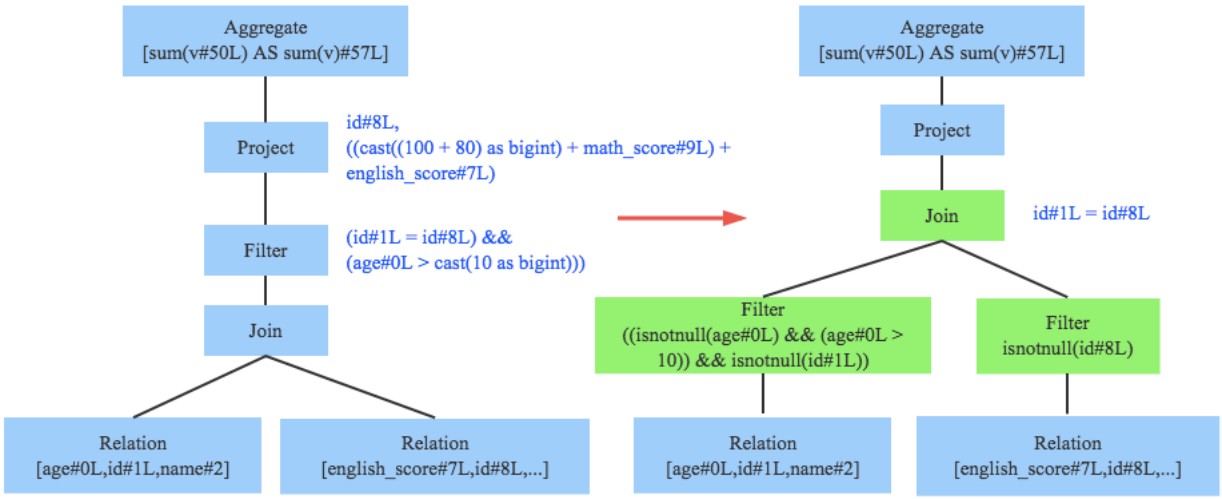

```
16/12/26 13:18:48 DEBUG AnalyzerSResolveReferences: Resolving 'people.id to id#1L
16/12/26 13:18:48 DEBUG AnalyzerSResolveReferences: Resolving 'score.id to id#8L
16/12/26 13:18:48 DEBUG AnalyzerSResolveReferences: Resolving 'people.age to age#0L
16/12/26 13:18:48 DEBUG AnalyzerSResolveReferences: Resolving 'score.id to id#8L
16/12/26 13:18:48 DEBUG AnalyzerSResolveReferences: Resolving 'score.math_score to math_score#9L
16/12/26 13:18:48 DEBUG AnalyzerSResolveReferences: Resolving 'score.english_score to english_score#7L
16/12/26 13:18:48 DEBUG AnalyzerSResolveReferences: Resolving 'v to v#50L
```

SparkSQL中Analyzer定义了各种解析规则，有兴趣深入了解的童鞋可以查看Analyzer类，其中定义了基本的解析规则，如下：

Optimizer

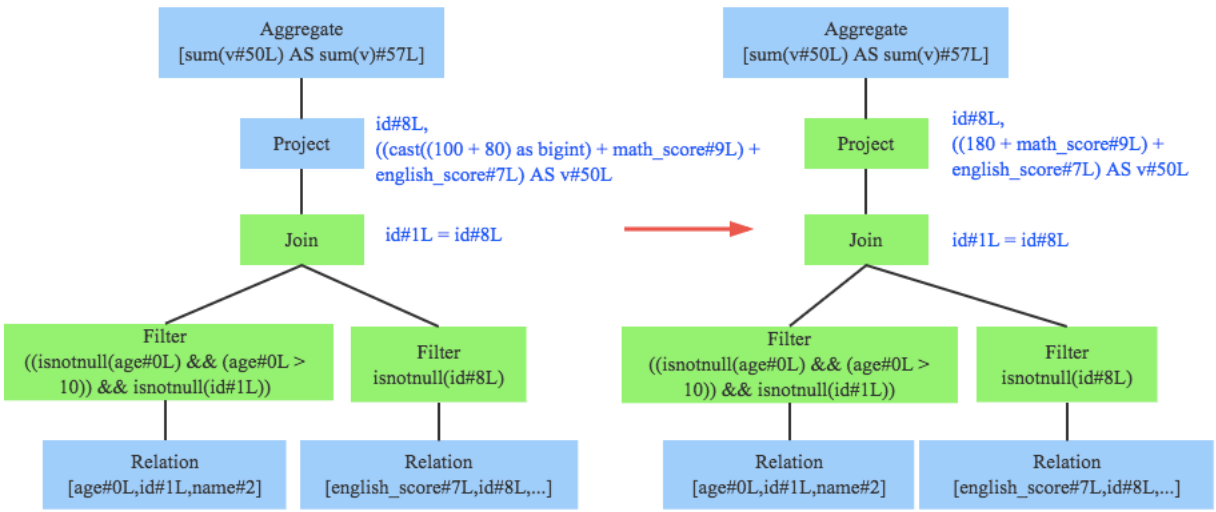
优化器是整个Catalyst的核心，上文提到优化器分为基于规则优化和基于代价优化两种，当前SparkSQL 2.1依然没有很好的支持基于代价优化（下文细讲），此处只介绍基于规则的优化策略，基于规则的优化策略实际上就是对语法树进行一次遍历，模式匹配能够满足特定规则的节点，再进行相应的等价转换。因此，基于规则优化说到底就是一棵树等价地转换为另一棵树。SQL中经典的优化规则有很多，下文结合示例介绍三种比较常见的规则：谓词下推（Predicate Pushdown）、常量累加（Constant Folding）和列值裁剪（Column Pruning）。

Predicate Pushdown

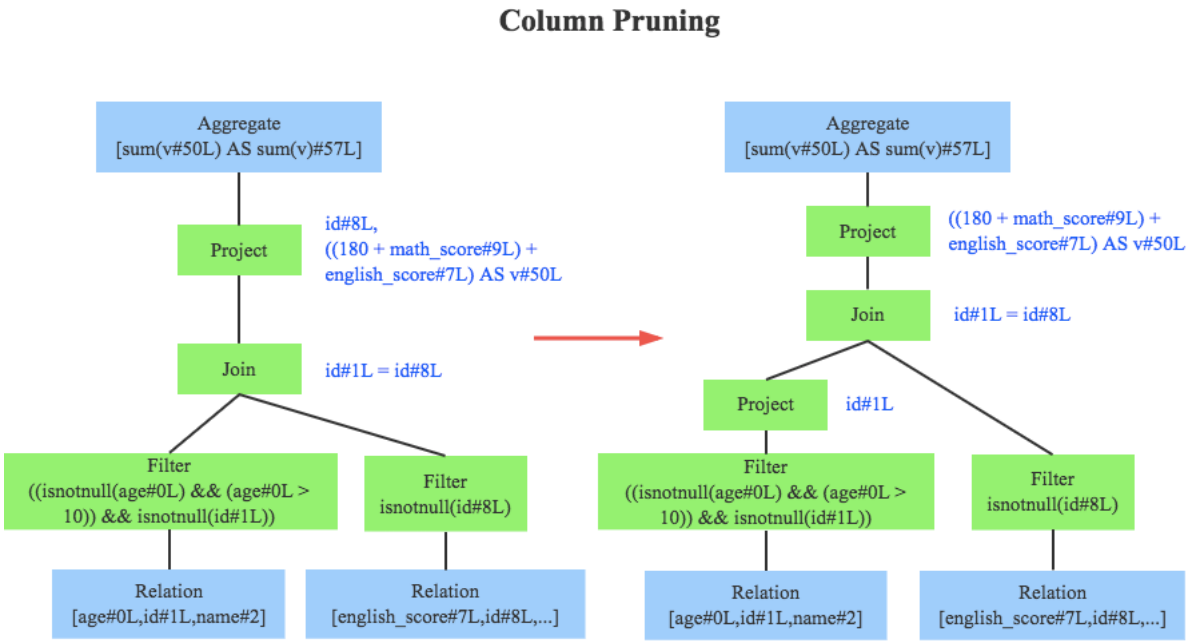


上图左边是经过Analyzer解析后的语法树，语法树中两个表先做join，之后再使用age>10对结果进行过滤。大家知道join算子通常是一个非常耗时的算子，耗时多少一般取决于参与join的两个表的大小，如果能够减少参与join两表的大小，就可以大大降低join算子所需时间。谓词下推就是这样一种功能，它会将过滤操作下推到join之前进行，上图中过滤条件age>0以及id!=null两个条件就分别下推到了join之前。这样，系统在扫描数据的时候就对数据进行了过滤，参与join的数据量将会得到显著的减少，join耗时必然也会降低。

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，虽然是一个很小的改动，但是意义巨大。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。

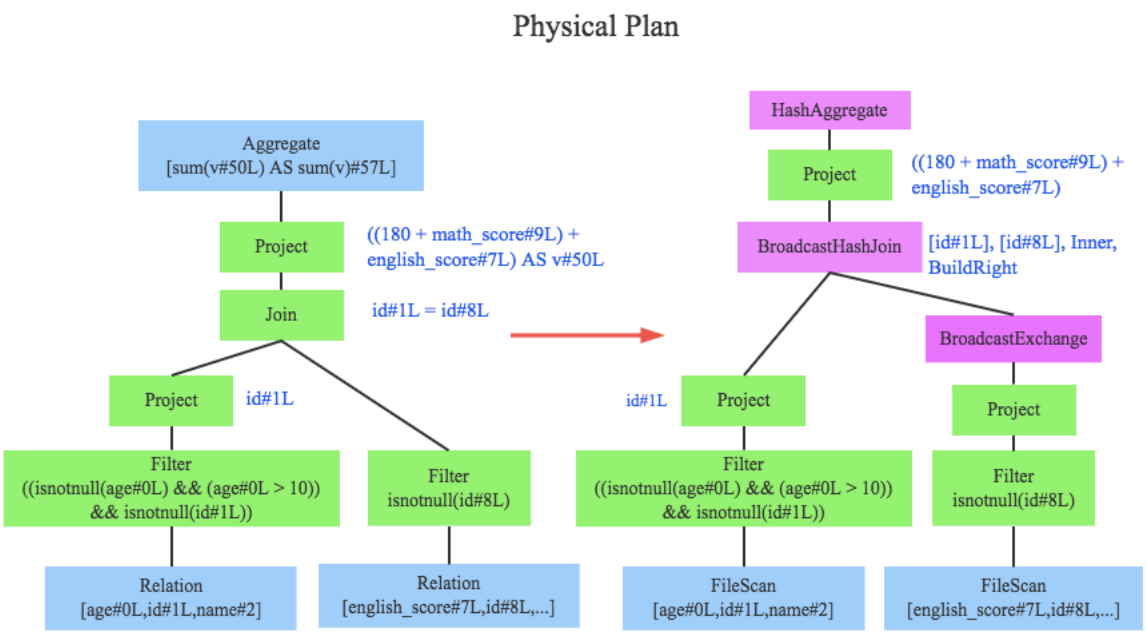


列值裁剪是另一个经典的规则，示例中对于people表来说，并不需要扫描它的所有列值，而只需要列值id，所以在扫描people之后需要将其他列进行裁剪，只留下列id。这个优化一方面大幅度减少了网络、内存数据量消耗，另一方面对于列存数据库（Parquet）来说大大提高了扫描效率。

除此之外，Catalyst还定义了很多其他优化规则，有兴趣深入了解的童鞋可以查看Optimizer类，下图简单的截取一部分规则：

```
Batch("Union", Once,
  CombineUnions) ::
Batch("Subquery", Once,
  OptimizeSubqueries) ::
Batch("Replace Operators", fixedPoint,
  ReplaceIntersectWithSemiJoin,
  ReplaceExceptWithAntiJoin,
  ReplaceDistinctWithAggregate) ::
Batch("Aggregate", fixedPoint,
  RemoveLiteralFromGroupExpressions,
  RemoveRepetitionFromGroupExpressions) ::
Batch("Operator Optimizations", fixedPoint,
  // Operator push down
  PushProjectionThroughUnion,
  ReorderJoin,
  EliminateOuterJoin,
  PushPredicateThroughJoin,
  PushDownPredicate,
  LimitPushDown,
  ColumnPruning,
  InferFiltersFromConstraints,
  // Operator combine
  CollapseRepartition,
  CollapseProject,
  CollapseWindow,
  CombineFilters,
  CombineLimits,
  CombineUnions,
  // Constant folding and strength reduction
  NullPropagation,
  FoldablePropagation,
  OptimizeIn(conf),
  ConstantFolding,
  ReorderAssociativeOperator,
  LikeSimplification,
  BooleanSimplification,
  SimplifyConditionals,
  RemoveDispensableExpressions,
  SimplifyBinaryComparison,
  PruneFilters,
  EliminateSorts,
  SimplifyCasts,
  SimplifyCaseConversionExpressions,
  RewriteCorrelatedScalarSubquery,
  EliminateSerialization,
  RemoveAliasOnlyProject) ::
```


至此，逻辑执行计划已经得到了比较完善的优化，然而，逻辑执行计划依然没办法真正执行，他们只是逻辑上可行，实际上Spark并不知道如何去执行这个东西。比如Join只是一个抽象概念，代表两个表根据相同的id进行合并，然而具体怎么实现这个合并，逻辑执行计划并没有说明。



此时就需要将逻辑执行计划转换为物理执行计划，将逻辑上可行的执行计划变为Spark可以真正执行的计划。比如Join算子，Spark根据不同场景为该算子制定了不同的算法策略，有BroadcastHashJoin、ShuffleHashJoin以及SortMergeJoin等（可以将Join理解为一个接口，BroadcastHashJoin是其中一个具体实现），物理执行计划实际上就是在这些具体实现中挑选一个耗时最小的算法实现，这个过程涉及到基于代价优化策略，后续文章细讲。

SparkSQL执行计划

至此，笔者通过一个简单的示例完整的介绍了Catalyst的整个工作流程，包括Parser阶段、Analyzer阶段、Optimize阶段以及Physical Planning阶段。有同学可能会比较感兴趣Spark环境下如何查看一条具体的SQL的整个过程，在此介绍两种方法：

1. 使用queryExecution方法查看逻辑执行计划，使用explain方法查看物理执行计划，分别如下所示：

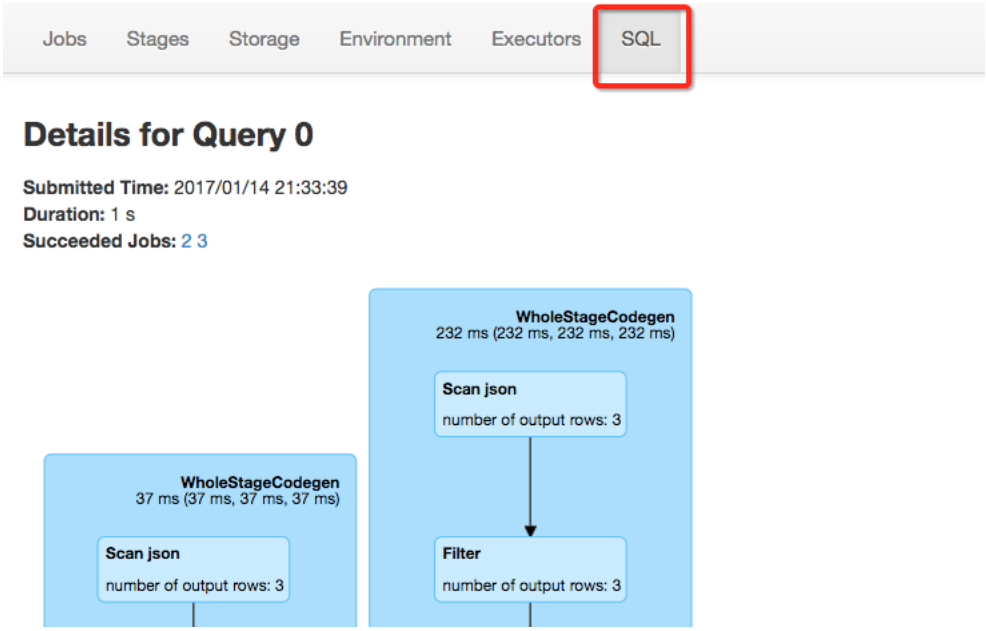
```
scala> spark.sql("select p.name,s.math_score from people p , score s where p.id = s.id and p.id > 1").queryExecution
res2: org.apache.spark.sql.execution.QueryExecution =
== Parsed Logical Plan ==
'Project ['p.name, 's.math_score]
+- 'Filter (('p.id = 's.id) && ('p.id > 1))
   +- 'Join Inner
      :- 'UnresolvedRelation `people`, p
      +- 'UnresolvedRelation `score`, s

== Analyzed Logical Plan ==
name: string, math_score: bigint
Project [name#2, math_score#10L]
+- Filter ((id#1L = id#9L) && (id#1L > cast(1 as bigint)))
   +- Join Inner
      :- SubqueryAlias p
      :  +- SubqueryAlias people
      :     +- Relation[age#0L,id#1L,name#2] json
      +- SubqueryAlias s
         +- SubqueryAlias score
            +- Relation[english_score#8L,id#9L,math_score#10L] json

== Optimized Logical Plan ==
Project [name#2, math_score#10L]
+- Join Inner, (id#1L = id#9L)
   :- Project [id#1L, name#2]
   :  +- Fil...
```

```
scala> spark.sql("select p.name,s.math_score from people p , score s where p.id = s.id and p.id > 1").explain
== Physical Plan ==
*Project [name#2, math_score#10L]
+- *BroadcastHashJoin [id#1L], [id#9L], Inner, BuildRight
   :- *Project [id#1L, name#2]
   :  +- *Filter (isnotnull(id#1L) && (id#1L > 1))
   :     +- *FileScan json [id#1L,name#2] Batched: false, Format: JSON, Location: InMemoryFileIndex[file:/Users/libisthanks/Docum
, PushedFilters: [IsNotNull(id), GreaterThan(id,1)], ReadSchema: struct<id:bigint,name:string>
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, true]))
      +- *Project [id#9L, math_score#10L]
         +- *Filter ((id#9L > 1) && isnotnull(id#9L))
            +- *FileScan json [id#9L,math_score#10L] Batched: false, Format: JSON, Location: InMemoryFileIndex[file:/Users/libist
Filters: [], PushedFilters: [GreaterThan(id,1), IsNotNull(id)], ReadSchema: struct<id:bigint,math_score:bigint>
```

2. 使用Spark WebUI进行查看，如下图所示：



参考文章：

1. Deep Dive into Spark SQL's Catalyst Optimizer：https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html (https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html)

2. A Deep Dive into Spark SQL's Catalyst Optimiser：https://www.youtube.com/watch?v=GDeePbbCz2g&index=4&list=PL-x35fyliRwheCVvliBZNm1VFwltr5b6B (https://www.youtube.com/watch?v=GDeePbbCz2g&index=4&list=PL-x35fyliRwheCVvliBZNm1VFwltr5b6B)

3. Spark SQL: Relational Data Processing in Spark：http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf (http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf)

4. 一个Spark SQL作业的一生：http://www.bitstech.net/2015/12/08/sparksql-introduction/

catalyst (http://hbasefly.com/tag/catalyst/) spark (http://hbasefly.com/tag/spark/) sparksql (http://hbasefly.com/tag/sparksql/)

范欣欣 (HTTP://HBASEFLY.COM/AUTHOR/LIBISTHANKSGMAIL-COM/)
就职于网易杭州研究院后台技术中心数据库技术组，从事HBase开发、运维，对HBase相关技术有浓厚的兴趣。邮箱：libisthanks@gmail.com

在“SparkSQL – 从0到1认识Catalyst”上有 6 条评论

忍冬 (HTTP://SADHEN.COM)
2017年3月5日 下午9:56 (http://hbasefly.com/2017/03/01/sparksql-catalyst/#comment-582)

最后那个链接无法访问，是网易内部的么？

回复 (HTTP://HBASEFLY.COM/2017/03/01/SPARKSQL-CATALYST/?REPLYTOCOM=582#RESPOND)

范欣欣
2017年3月6日 上午9:20 (http://hbasefly.com/2017/03/01/sparksql-catalyst/#comment-584)

链接是网易内部的 不过外面也放出来了 参考：http://www.bitstech.net/2015/12/08/sparksql-introduction/

回复 (HTTP://HBASEFLY.COM/2017/03/01/SPARKSQL-CATALYST/?REPLYTOCOM=584#RESPOND)