一条 SQL 在 Apache Spark 之旅（中）

在 《一条 SQL 在 Apache Spark 之旅（上）》 文章中我们介绍了一条 SQL 在 Apache Spark 之旅的 Parser 和 Analyzer 两个过程，本文接上文继续介绍。

优化逻辑计划阶段 - Optimizer

在前文的绑定逻辑计划阶段对 Unresolved LogicalPlan 进行相关 transform 操作得到了 Analyzed Logical Plan，这个 Analyzed Logical Plan 是可以直接转换成 Physical Plan 然后在 Spark 中执行。但是如果直接这么弄的话，得到的 Physical Plan 很可能不是最优的，因为在实际应用中，很多低效的写法会带来执行效率的问题，需要进一步对Analyzed Logical Plan 进行处理，得到更优的逻辑算子树。于是， 针对 SQL 逻辑算子树的优化器 Optimizer 应运而生。

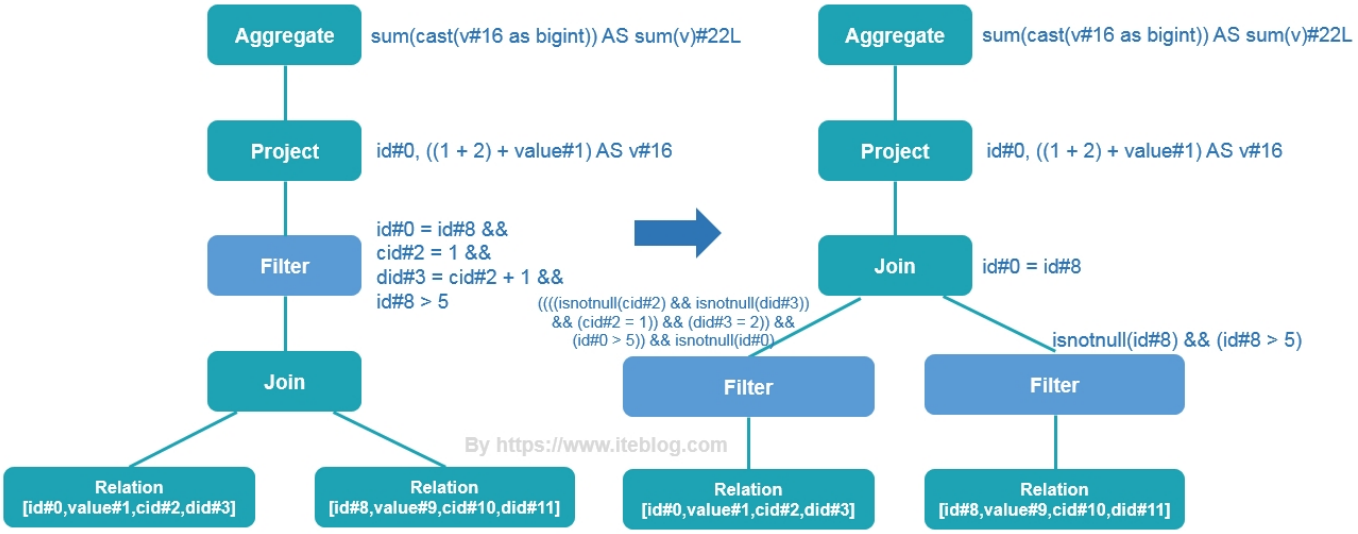
这个阶段的优化器主要是基于规则的（Rule-based Optimizer，简称 RBO），而绝大部分的规则都是启发式规则，也就是基于直观或经验而得出的规则，比如列裁剪（过滤掉查询不需要使用到的列）、谓词下推（将过滤尽可能地下沉到数据源端）、常量累加（比如 1 + 2 这种事先计算好）以及常量替换（比如 SELECT * FROM table WHERE i = 5 AND j = i + 3 可以转换成 SELECT * FROM table WHERE i = 5 AND j = 8）等等。

与前文介绍绑定逻辑计划阶段类似，这个阶段所有的规则也是实现 Rule 抽象类，多个规则组成一个 Batch，多个 Batch 组成一个 batches，同样也是在 RuleExecutor 中进行执行，由于前文已经介绍了 Rule 的执行过程，本节就不再赘述。

那么针对前文的 SQL 语句，这个过程都会执行哪些优化呢？这里按照 Rule 执行顺序一一进行说明。

谓词下推

谓词下推在 Spark SQL 是由 PushDownPredicate 实现的，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。所以针对我们上面介绍的 SQL，使用谓词下推优化得到的逻辑计划如下：

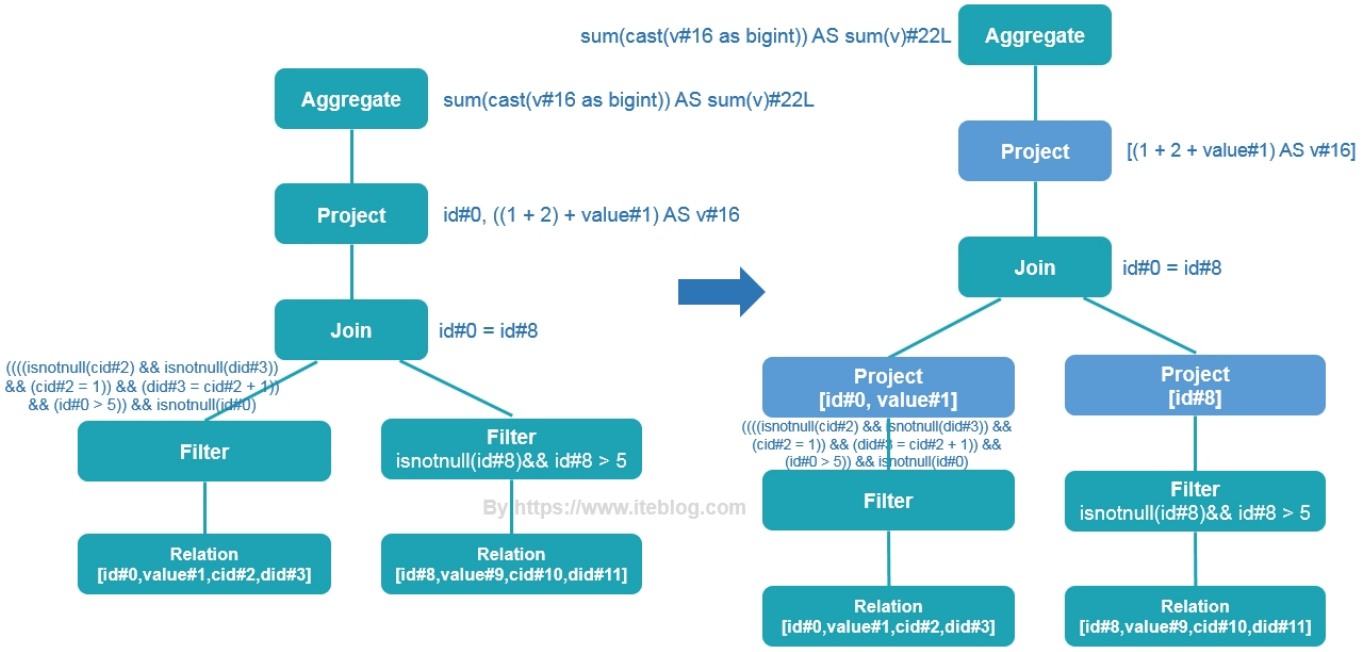


如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

从上图可以看出，谓词下推将 Filter 算子直接下推到 Join
之前了（注意，上图是从下往上看的）。也就是在扫描 t1 表的时候会先使用 ((((isnotnull(cid#2) && isnotnull(did#3)) && (cid#2 = 1)) && (did#3 = 2)) && (id#0 > 50000)) && isnotnull(id#0) 过滤条件过滤出满足条件的数据；同时在扫描 t2 表的时候会先使用 isnotnull(id#8) && (id#8 > 50000) 过滤条件过滤出满足条件的数据。经过这样的操作，可以大大减少 Join
算子处理的数据量，从而加快计算速度。

列裁剪

列裁剪在 Spark SQL 是由 ColumnPruning 实现的。因为我们查询的表可能有很多个字段，但是每次查询我们很大可能不需要扫描出所有的字段，这个时候利用列裁剪可以把那些查询不需要的字段过滤掉，使得扫描的数据量减少。所以针对我们上面介绍的
SQL，使用列裁剪优化得到的逻辑计划如下：

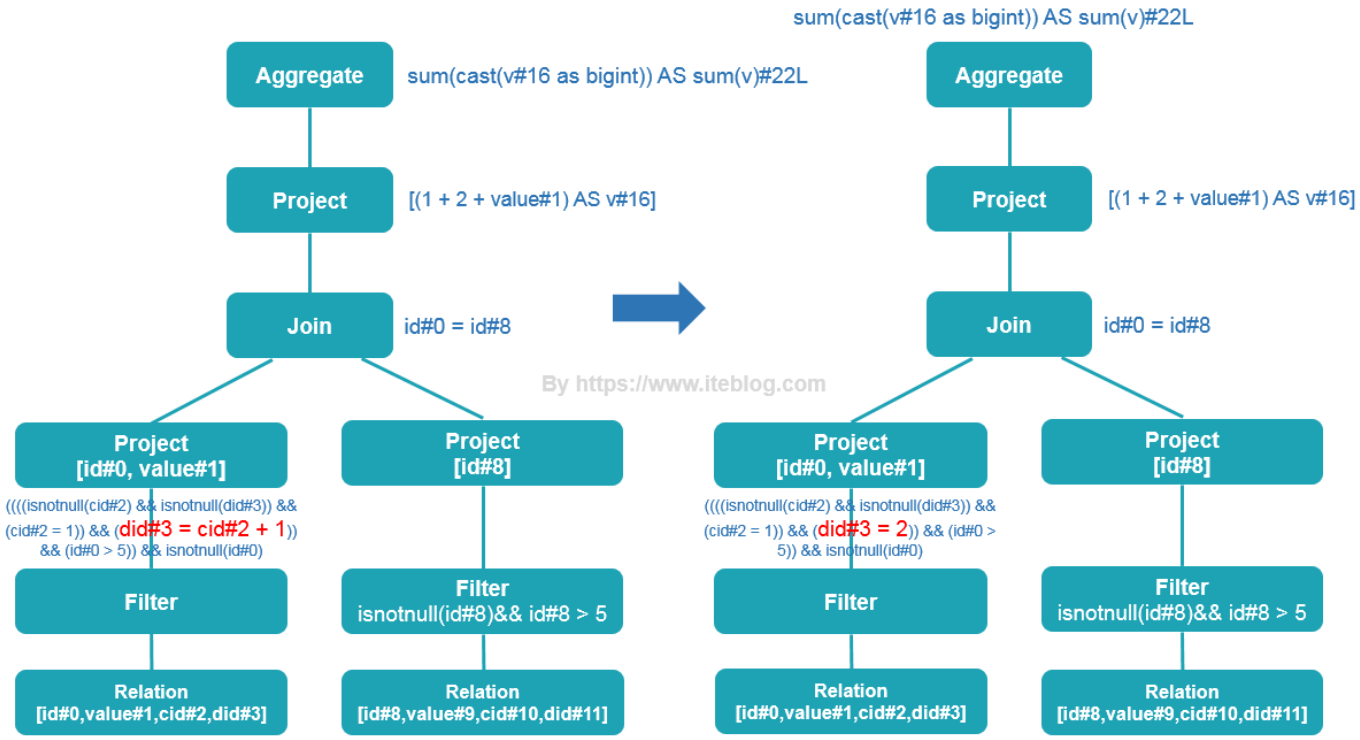


如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

从上图可以看出，经过列裁剪后，t1 表只需要查询 id 和 value 两个字段；t2 表只需要查询 id 字段。这样减少了数据的传输，而且如果底层的文件格式为列存（比如 Parquet），可以大大提高数据的扫描速度的。

常量替换

常量替换在 Spark SQL 是由 ConstantPropagation 实现的。也就是将变量替换成常量，比如 SELECT * FROM table WHERE i = 5 AND j = i + 3 可以转换成 SELECT * FROM table WHERE i = 5 AND j = 8。这个看起来好像没什么的，但是如果扫描的行数非常多可以减少很多的计算时间的开销的。经过这个优化，得到的逻辑计划如下：

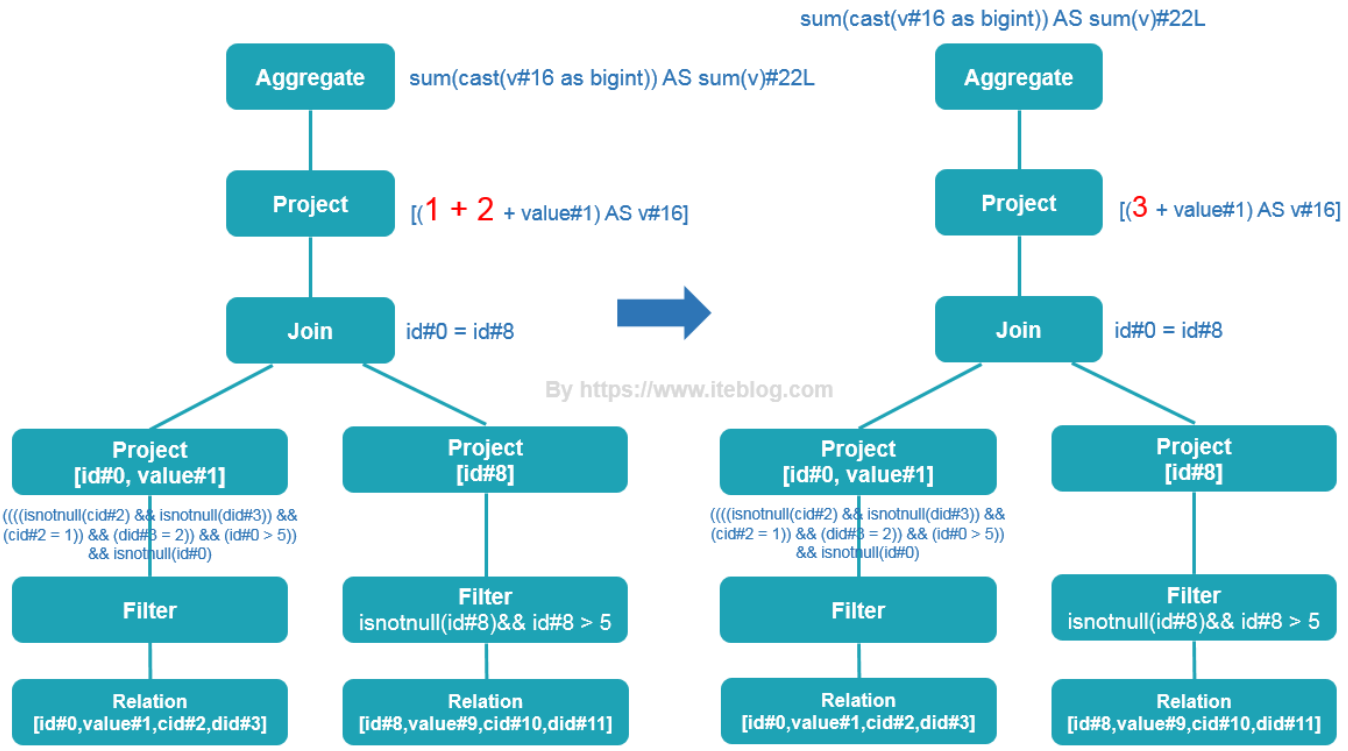


如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

我们的查询中有 t1.cid = 1 AND t1.did = t1.cid + 1 查询语句，从里面可以看出 t1.cid 其实已经是确定的值了，所以我们完全可以使用它计算出 t1.did。

常量累加

常量累加在 Spark SQL 是由 ConstantFolding 实现的。这个和常量替换类似，也是在这个阶段把一些常量表达式事先计算好。这个看起来改动的不大，但是在数据量非常大的时候可以减少大量的计算，减少 CPU 等资源的使用。经过这个优化，得到的逻辑计划如下：

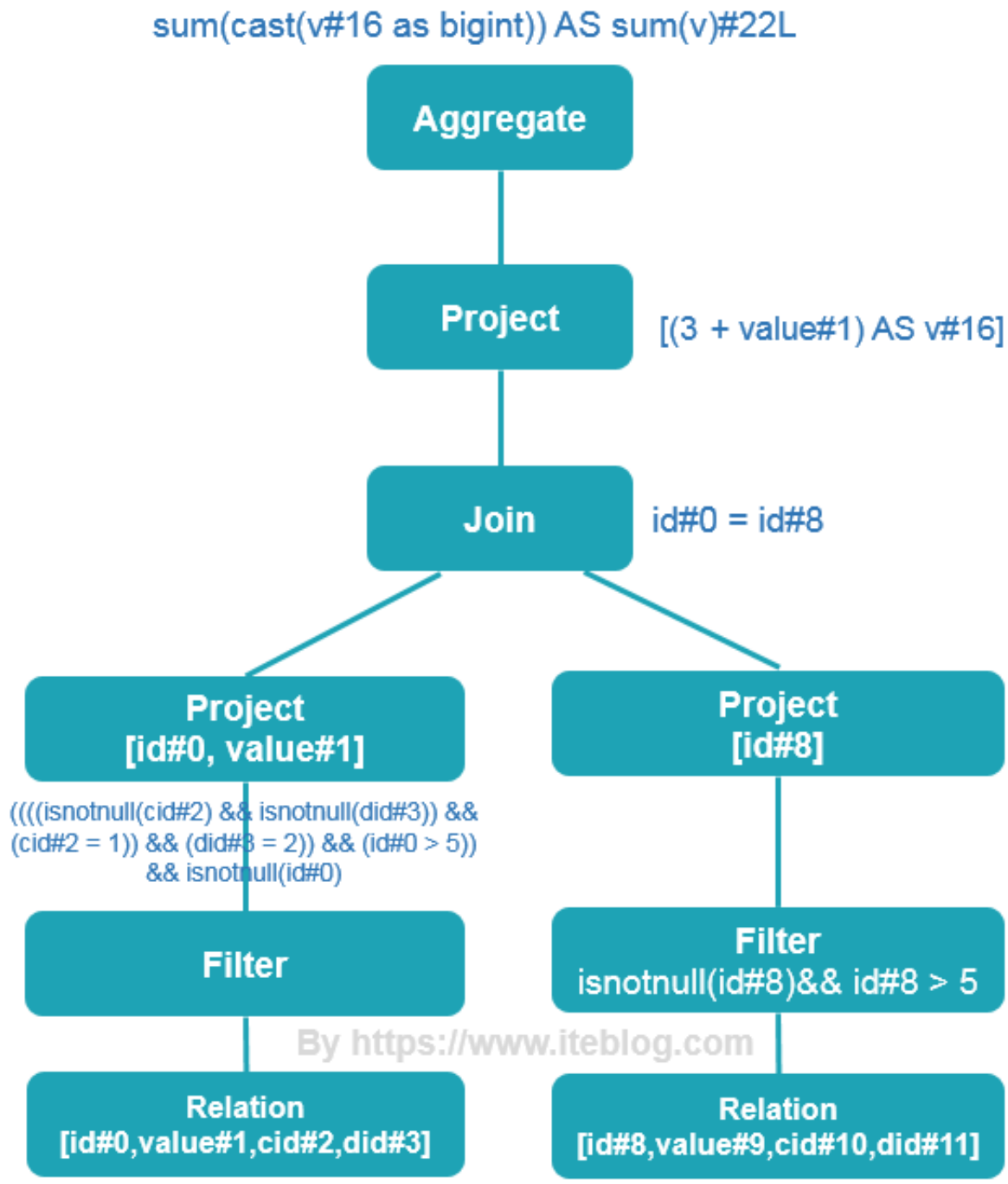


如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

所以经过上面四个步骤的优化之后，得到的优化之后的逻辑计划为：

```
== Optimized Logical Plan ==
Aggregate [sum(cast(v#16 as bigint)) AS sum(v)#22L]
+- Project [(3 + value#1) AS v#16]
   +- Join Inner, (id#0 = id#8)
      :- Project [id#0, value#1]
      :  +- Filter (((((isnotnull(cid#2) && isnotnull(did#3)) && (cid#2 = 1)) && (did#3 = 2)) && (id#0
 > 5)) && isnotnull(id#0))
      :     +- Relation[id#0,value#1,cid#2,did#3] csv
      +- Project [id#8]
         +- Filter (isnotnull(id#8) && (id#8 > 5))
            +- Relation[id#8,value#9,cid#10,did#11] csv
```

对应的图如下：



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

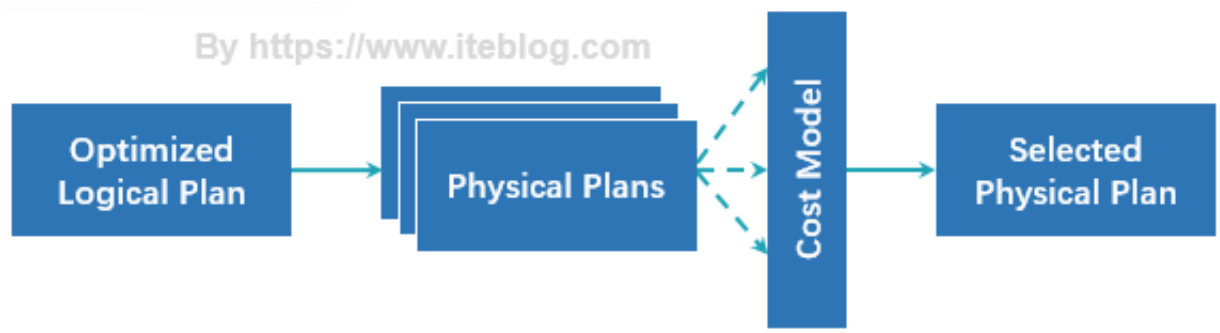
到这里，优化逻辑计划阶段就算完成了。另外，Spark 内置提供了多达70个优化 Rule，详情请参见 这里。

生成可执行的物理计划阶段 - SparkPlanner

前面介绍的逻辑计划在 Spark 里面其实并不能被执行的，为了能够执行这个 SQL，一定需要翻译成物理计划，到这个阶段 Spark 就知道如何执行这个 SQL 了。和前面逻辑计划绑定和优化不一样，这里使用的是策略（Strategy），而且前面介绍的逻辑计划绑定和优化经

过 Transformations 动作之后，树的类型并没有改变，也就是说：Expression 经过 Transformations 之后得到的还是 Transformations ；Logical Plan 经过 Transformations 之后得到的还是 Logical Plan。而到了这个阶段，经过 Transformations 动作之后，树的类型改变了，由 Logical Plan 转换成 Physical Plan 了。

一个逻辑计划（Logical Plan）经过一系列的策略处理之后，得到多个物理计划（Physical Plans），物理计划在 Spark 是由 SparkPlan 实现的。多个物理计划再经过代价模型（Cost Model）得到选择后的物理计划（Selected Physical Plan），整个过程如下所示：



Cost Model 对应的就是基于代价的优化（Cost-based Optimizations，CBO，主要由华为的大佬们实现的，详见 SPARK-16026），核心思想是计算每个物理计划的代价，然后得到最优的物理计划。但是在目前最新版的 Spark 2.4.3，这一部分并没有实现，直接返回多个物理计划列表的第一个作为最优的物理计划，如下：

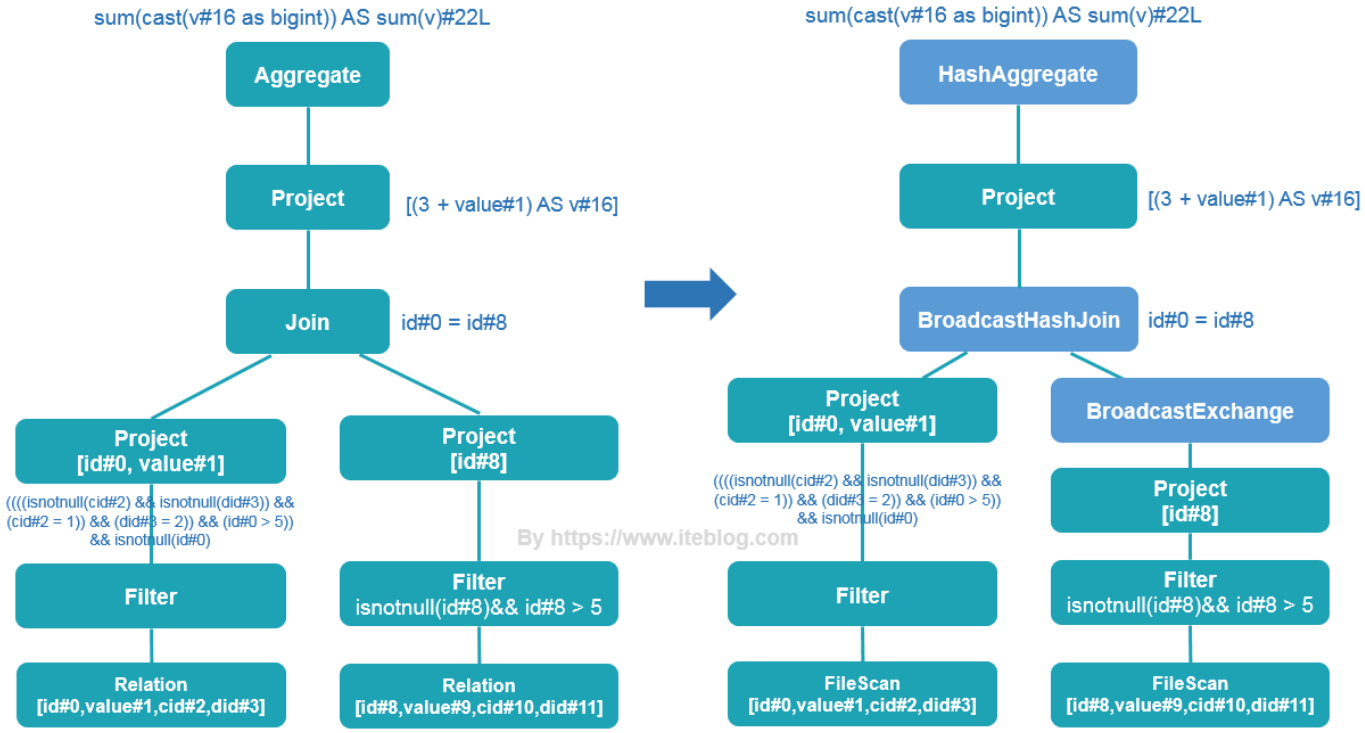
```
lazy val sparkPlan: SparkPlan = {
   SparkSession.setActiveSession(sparkSession)
   // TODO: We use next(), i.e. take the first plan returned by the planner, here for now,
   //       but we will implement to choose the best plan.
   planner.plan(ReturnAnswer(optimizedPlan)).next()
}
```

而 SPARK-16026 引入的 CBO 优化主要是在前面介绍的优化逻辑计划阶段 - Optimizer 阶段进行的，对应的 Rule 为 CostBasedJoinReorder，并且默认是关闭的，需要通过 spark.sql.cbo.enabled 或 spark.sql.cbo.joinReorder.enabled 参数开启。
所以到了这个节点，最后得到的物理计划如下：

```
== Physical Plan ==
*(3) HashAggregate(keys=[], functions=[sum(cast(v#16 as bigint))], output=[sum(v)#22L])
+- Exchange SinglePartition
   +- *(2) HashAggregate(keys=[], functions=[partial_sum(cast(v#16 as bigint))], output=[sum#2
```

```
4L])
   +- *(2) Project [(3 + value#1) AS v#16]
      +- *(2) BroadcastHashJoin [id#0], [id#8], Inner, BuildRight
         :- *(2) Project [id#0, value#1]
         :  +- *(2) Filter (((((isnotnull(cid#2) && isnotnull(did#3)) && (cid#2 = 1)) && (did#3 = 2)) &
& (id#0 > 5)) && isnotnull(id#0))
         :     +- *(2) FileScan csv [id#0,value#1,cid#2,did#3] Batched: false, Format: CSV, Location
: InMemoryFileIndex[file:/iteblog/t1.csv], PartitionFilters: [], PushedFilters: [IsNotNull(cid), IsNo
tNull(did), EqualTo(cid,1), EqualTo(did,2), GreaterThan(id,5), IsNotNull(id)], ReadSchema: struct
<id:int,value:int,cid:int,did:int>
         +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0, int, true] as bigi
nt)))
            +- *(1) Project [id#8]
               +- *(1) Filter (isnotnull(id#8) && (id#8 > 5))
                  +- *(1) FileScan csv [id#8] Batched: false, Format: CSV, Location: InMemoryFileInd
ex[file:/iteblog/t2.csv], PartitionFilters: [], PushedFilters: [IsNotNull(id), GreaterThan(id,5)], Rea
dSchema: struct<id:int>
```

从上面的结果可以看出，物理计划阶段已经知道数据源是从 csv 文件里面读取了，也知道文件的路径，数据类型等。而且在读取文件的时候，直接将过滤条件（PushedFilters）加进去了。同时，这个 Join 变成了 BroadcastHashJoin，也就是将 t2 表的数据 Broadcast 到 t1 表所在的节点。图表示如下：



如果想及时了解Spark、Hadoop或者HBase相关的文章，欢迎关注微信公众号：iteblog_hadoop

到这里， Physical Plan 就完全生成了。由于篇幅的原因，剩余的 SQL 处理我将在下一篇文章进行介绍，包括代码生成（WholeStageCodeGen）以及执行相关的等东西，敬请关注。