Spark SQL之Join实现原理

Join是数据库查询永远绕不开的话题，传统查询SQL技术总体可以分为简单操作（过滤操作-where、排序操作-limit等），聚合操作-groupBy等以及Join操作等。其中Join操作是其中最复杂、代价最大的操作类型，也是OLAP场景中使用相对较多的操作。此外，从业务层面来讲，用户在数仓建设的时候也会涉及Join使用的问题。通常情况下，数据仓库中的表一般会分为"低层次表"和"高层次表"。

所谓"低层次表"，就是数据源导入数仓之后直接生成的表，单表列值较少，一般可以明显归为维度表或者事实表，表和表之间大多存在外健依赖，所以查询起来会遇到大量Join运算，查询效率相对比较差。而"高层次表"是在"低层次表"的基础上加工转换而来，通常做法是使用SQL语句将需要Join的表预先进行合并形成"宽表"，在宽表上的查询因为不需要执行大量Join因而效率相对较高，很明显，宽表缺点是数据会有大量冗余，而且生成相对比较滞后，查询结果可能并不及时。

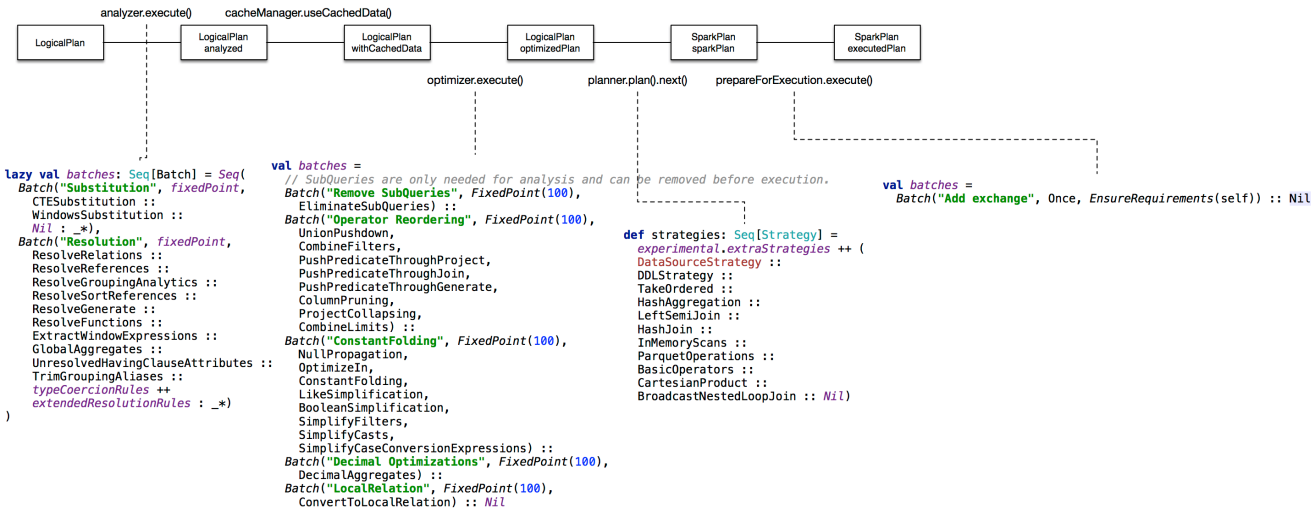
因此，为了获得实效性更高的查询结果，大多数场景还是需要进行复杂的Join操作。Join操作之所以复杂，不仅仅因为通常情况下其时间空间复杂度高，更重要的是它有很多算法，在不同场景下需要选择特定算法才能获得最好的优化效果。

1. Join查询概述

关系代数中，Join一直都是最为复杂的操作。在分布式环境下，Join涉及到大量的数据shuffle操作，因此一直是各个系统的优化重点。当前SparkSQL支持三种Join算法－shuffle hash join、broadcast hash join以及sort merge join。其中前两者归根到底都属于hash join，只不过在hash join之前需要先shuffle还是先broadcast。

本文接下来通过一个简单的例子，从细节层面分析整个Join的实现流程。

```
select * from A join B on A.id = B.id
```



2.文法定义

在ANTLR4的文法文件中，Join表达式相关的定义如下：

```
fromClause : FROM relation (',' relation)* lateralView* ;
relation : relationPrimary joinRelation* ;
joinRelation : (joinType) JOIN right=relationPrimary joinCriteria?
             | NATURAL joinType JOIN right=relationPrimary ;
joinType : INNER? | CROSS | LEFT OUTER? | LEFT SEMI | RIGHT OUTER? | FULL OUTER? | LEFT? ANTI ;
joinCriteria : ON booleanExpression | USING '(' identifier (',' identifier)* ')';
```

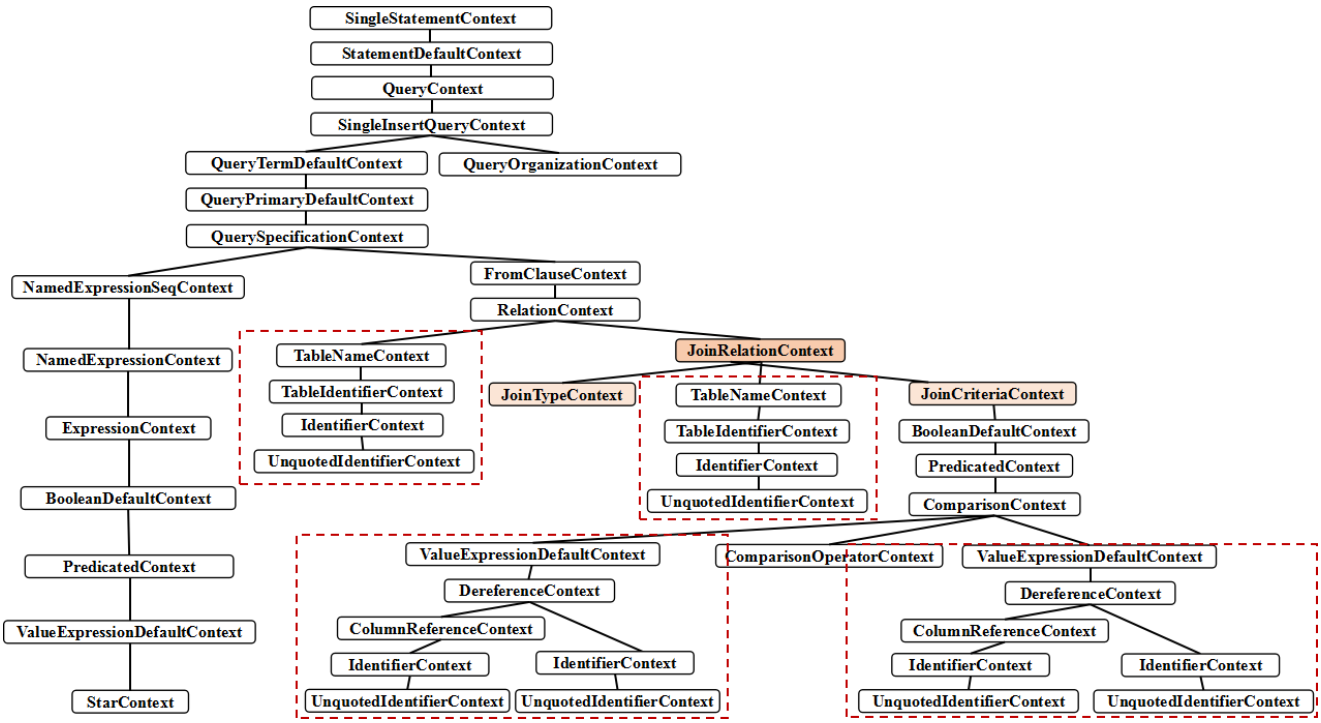
由此可见，Join表达式处于From语句中，是针对数据源的操作。JoinType代表Spark-SQL中目前所支持的Join类型，包括Inner，FullOuter，LeftOuter，RightOuter，LeftSemi，LeftAnti和Cross七种，对应关键字如表所示：

查询关键字	**Join**类型
"inner"	Inner
"outer" | "full" | "fullouter"	FullOuter
"leftouter" | "left"	LeftOuter
"rightouter" | "right"	RightOuter
"leftsemi"	LeftSemi
"leftanti"	LeftAnti
"cross"	Cross

如果需要自定义开发其它类型的Join操作，首先需要修改的就是这部分定义。顾名思义，JoinCriteria代表的是Join操作的条件部分，SQL语句中支持`ON`和`USING`关键字。

3.抽象语法树(AST)

编译器由ANTLR4自动生成，针对该SQL语句生成如下AST：



4.逻辑算子树(Logical Plan)

逻辑算子树的生成包括两步：

(1)生成未解析的逻辑算子树，即Unresolved Logical Plan，主要在Catalyst的AstBuilder完成

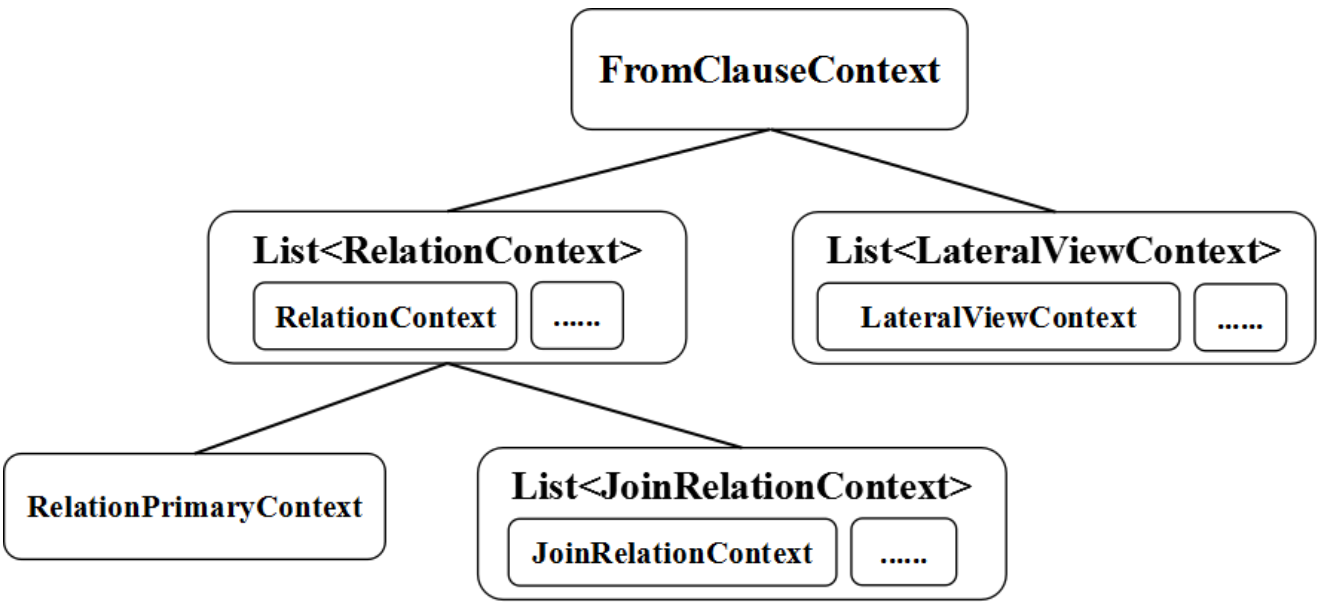
(2)生成解析完毕的逻辑算子树，即Resolved Logical Plan，由Catalyst的Analyzer完成

4.1 生成Unresolved Logical Plan

主要的逻辑在AstBuilder中，核心步骤涉及到以下几个函数

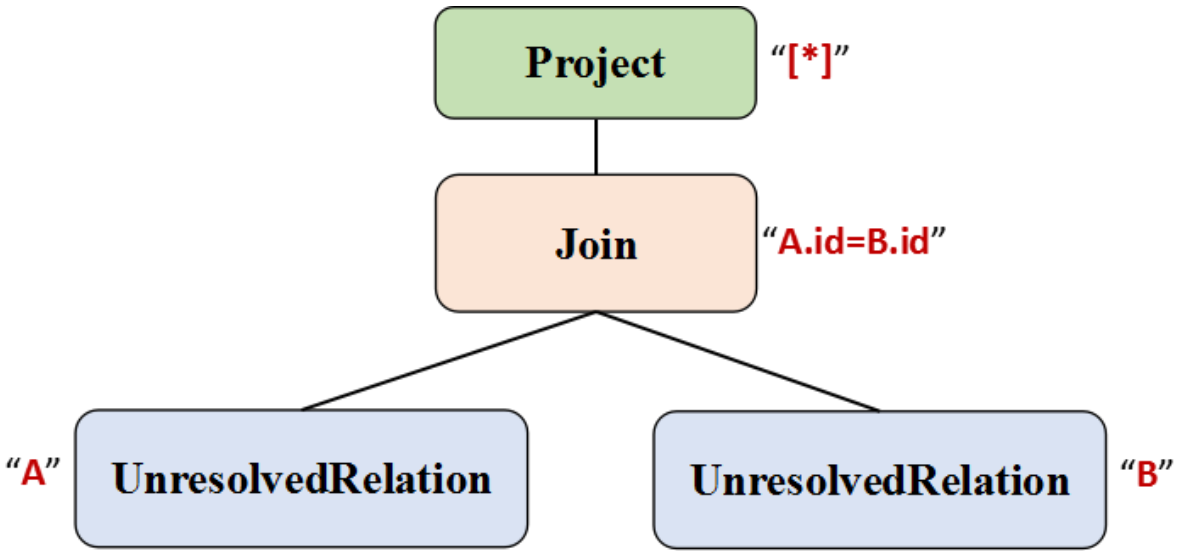
```
override def visitFromClause(ctx: FromClauseContext): LogicalPlan = withOrigin(ctx) {
  val from = ctx.relation.asScala.foldLeft(null: LogicalPlan) { (left, relation) =>
    val right = plan(relation.relationPrimary)
    val join = right.optionalMap(left)(Join(_, _, Inner, None))
    withJoinRelations(join, relation)
  }
  ctx.lateralView.asScala.foldLeft(from)(withGenerate)
}
```

对于from clause中的每个relation以逗号划分，类似于`from relation, relation, ...`；每个relation中通常会有join操作，因此把一个relation中的join操作映射为二叉树。传入join方法`ctx`的用于指明该Join操作的类型，是InnerJoin还是OuterJoin，其中NATURA-JOIN是一种特殊类型的EQUI-JOIN，当两张表的Join keys具有相同名字，并且对应的列数据类型相同。



```
override def visitRelation(ctx: RelationContext): LogicalPlan = withOrigin(ctx) {
  withJoinRelations(plan(ctx.relationPrimary), ctx)
}

private def withJoinRelations(base: LogicalPlan, ctx: RelationContext): LogicalPlan = {
  ctx.joinRelation.asScala.foldLeft(base) { (left, join) =>
    withOrigin(join) {
      val baseJoinType = join.joinType match {
        case null => Inner
        case jt if jt.CROSS != null => Cross
        case jt if jt.FULL != null => FullOuter
        case jt if jt.SEMI != null => LeftSemi
        case jt if jt.ANTI != null => LeftAnti
        case jt if jt.LEFT != null => LeftOuter
        case jt if jt.RIGHT != null => RightOuter
        case _ => Inner
      }

      // Resolve the join type and join condition
      val (joinType, condition) = Option(join.joinCriteria) match {
        case Some(c) if c.USING != null =>
          (UsingJoin(baseJoinType, c.identifier.asScala.map(_.getText)), None)
        case Some(c) if c.booleanExpression != null =>
          (baseJoinType, Option(expression(c.booleanExpression)))
        case None if join.NATURAL != null =>
          if (baseJoinType == Cross) {
            throw new ParseException("NATURAL CROSS JOIN is not supported", ctx)
          }
          (NaturalJoin(baseJoinType), None)
        case None =>
          (baseJoinType, None)
      }
      Join(left, plan(join.right), joinType, condition)
    }
  }
}
```

经过递归调用，该SQL语句最终生成如下的算子树



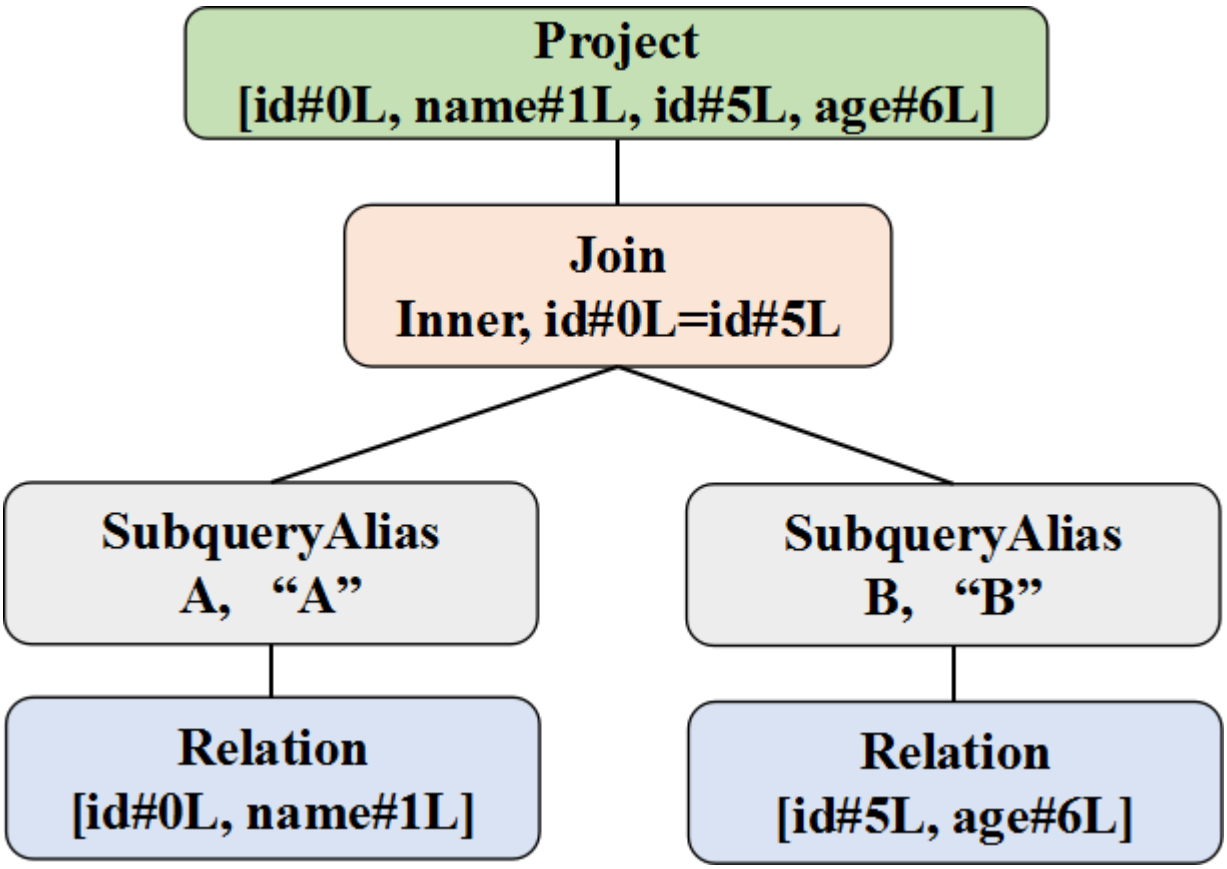
4.2 生成Resolved Logical Plan

Analyzer的规则涉及到resolve Join的规则有：ResolveReferences和ResolveNaturalAndUsingJoin。针对本例，解析流程对应的是ResolveReference中逻辑：

```
/**
 * Generate a new logical plan for the right child with different expression IDs
 * for all conflicting attributes.
 */
private def dedupRight (left: LogicalPlan, right: LogicalPlan): LogicalPlan = {
  val conflictingAttributes = left.outputSet.intersect(right.outputSet)
  ......
}

// ResolveReferences.apply
// If the projection list contains Stars, expand it.
case p: Project if containsStar(p.projectList) =>
  p.copy(projectList = buildExpandedProjectList(p.projectList, p.child))
.....................................
case j @ Join(left, right, _, _) if !j.duplicateResolved =>
  j.copy(right = dedupRight(left, right))
```

根据该逻辑，如果Join操作存在重名的属性(即左右子节点的输出属性名集合有重叠)，那么就调用`dedupRight`方法将右子节点对应的Expression用一个新的Expression ID表示，这样即使出现同名，经过处理之后Expression ID也不相同，因此可以区分。

总而言之，Analyzer所起到的作用就是将一些catalog信息添加到Unresolved Logical Plan中，并对其进行若干的调整，例如加入别名信息(SubqueryAlias)等。经过这一步，最终的Logical Plan算子树如下：

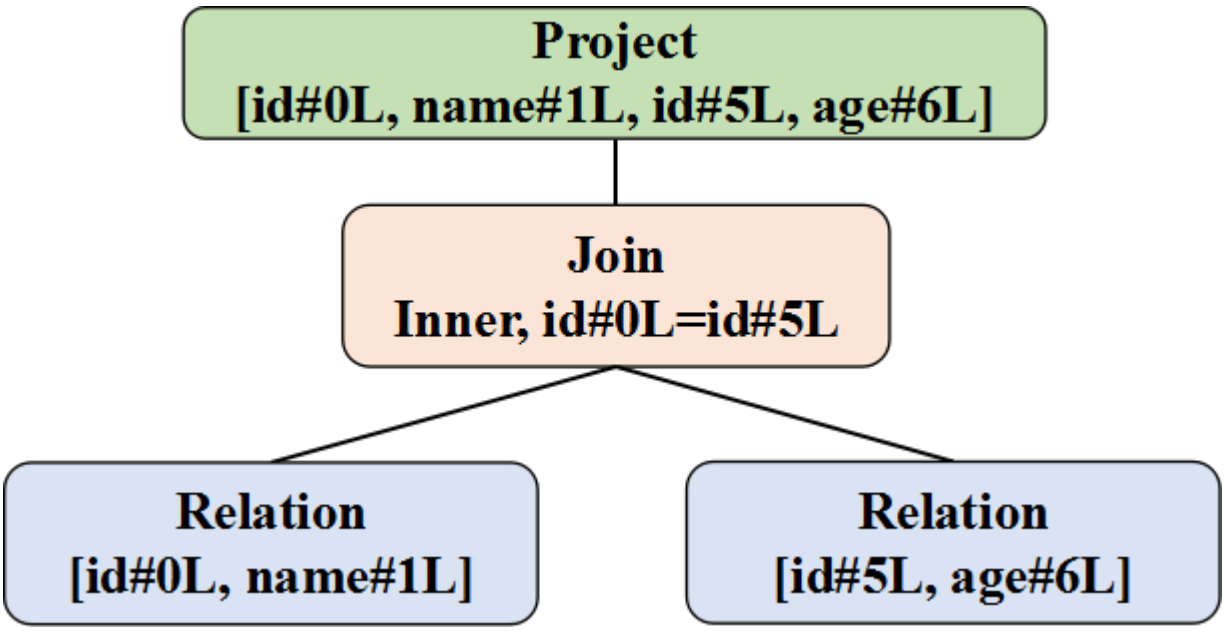


5.优化处理(Optimized Logical Plan)

通过Optimizer来对逻辑算子树进行优化，对于这条语句，起作用的优化规则分别是：

EliminateSubqueryAliases：消除别名

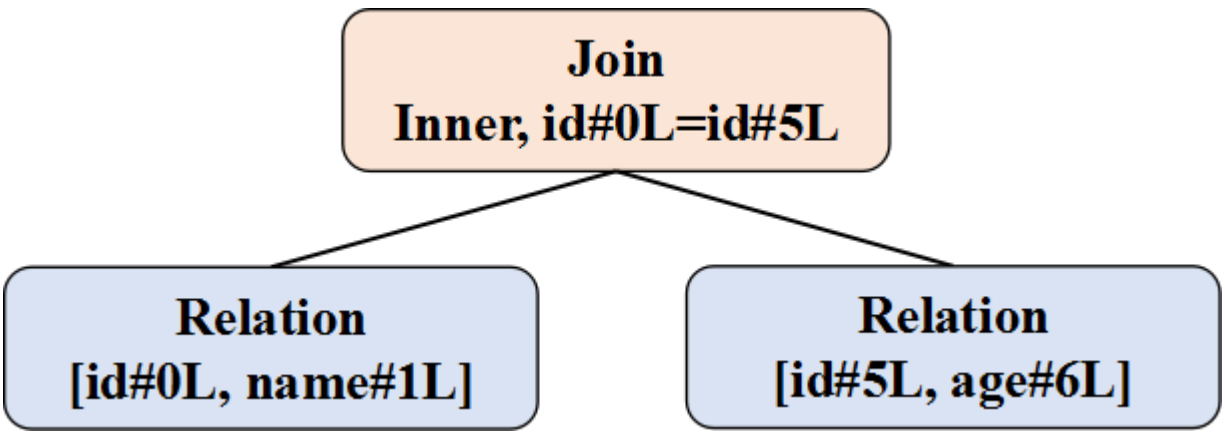
```
/**
 * Removes [[SubqueryAlias]] operators from the plan. Subqueries are only required to provide
 * scoping information for attributes and can be removed once analysis is complete.
 */
object EliminateSubqueryAliases extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transformUp {
    case SubqueryAlias(_, child, _) => child
  }
}
```



ColumnPruning：列剪裁

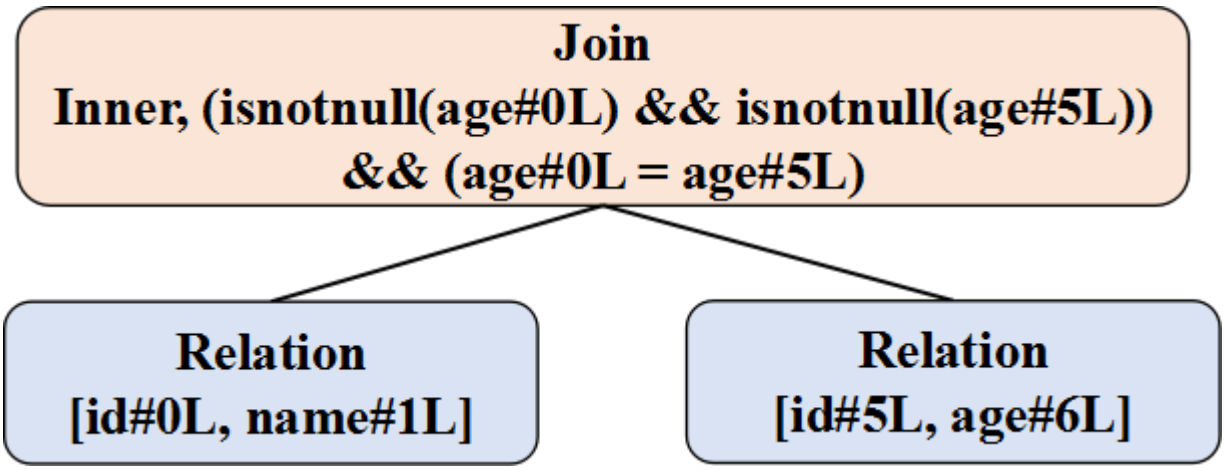
```
object ColumnPruning extends Rule[LogicalPlan] {
  private def sameOutput(output1: Seq[Attribute], output2: Seq[Attribute]): Boolean =
    output1.size == output2.size &&
      output1.zip(output2).forall(pair => pair._1.semanticEquals(pair._2))

  def apply(plan: LogicalPlan): LogicalPlan = removeProjectBeforeFilter(plan transform {
    // Prunes the unused columns from project list of Project/Aggregate/Expand
    case p @ Project(_, p2: Project) if (p2.outputSet -- p.references).nonEmpty =>
      p.copy(child = p2.copy(projectList = p2.projectList.filter(p.references.contains)))
    case p @ Project(_, a: Aggregate) if (a.outputSet -- p.references).nonEmpty =>
      p.copy(
        child = a.copy(aggregateExpressions =
a.aggregateExpressions.filter(p.references.contains)))
    case a @ Project(_, e @ Expand(_, _, grandChild)) if (e.outputSet -- a.references).nonEmpty
=>
      val newOutput = e.output.filter(a.references.contains(_))
      val newProjects = e.projections.map { proj =>
        proj.zip(e.output).filter { case (_, a) =>
          newOutput.contains(a)
        }.unzip._1
      }
      a.copy(child = Expand(newProjects, newOutput, grandChild))

    // Prunes the unused columns from child of `DeserializeToObject`
    case d @ DeserializeToObject(_, _, child) if (child.outputSet -- d.references).nonEmpty =>
      d.copy(child = prunedChild(child, d.references))

    // Prunes the unused columns from child of Aggregate/Expand/Generate
    case a @ Aggregate(_, _, child) if (child.outputSet -- a.references).nonEmpty =>
      a.copy(child = prunedChild(child, a.references))
    case e @ Expand(_, _, child) if (child.outputSet -- e.references).nonEmpty =>
      e.copy(child = prunedChild(child, e.references))
    case g: Generate if !g.join && (g.child.outputSet -- g.references).nonEmpty =>
      g.copy(child = prunedChild(g.child, g.references))

    // Turn off `join` for Generate if no column from it's child is used
    case p @ Project(_, g: Generate)
        if g.join && !g.outer && p.references.subsetOf(g.generatedSet) =>
      p.copy(child = g.copy(join = false))

    // Eliminate unneeded attributes from right side of a Left Existence Join.
    case j @ Join(_, right, LeftExistence(_), _) =>
      j.copy(right = prunedChild(right, j.references))

    // all the columns will be used to compare, so we can't prune them
    case p @ Project(_, _: SetOperation) => p
    case p @ Project(_, _: Distinct) => p
    // Eliminate unneeded attributes from children of Union.
    case p @ Project(_, u: Union) =>
      if ((u.outputSet -- p.references).nonEmpty) {
        val firstChild = u.children.head
        val newOutput = prunedChild(firstChild, p.references).output

        // pruning the columns of all children based on the pruned first child.
```

```
        val newChildren = u.children.map { p =>
          val selected = p.output.zipWithIndex.filter { case (a, i) =>
            newOutput.contains(firstChild.output(i))
          }.map(_._1)
          Project(selected, p)
        }
        p.copy(child = u.withNewChildren(newChildren))
      } else {
        p
      }

    // Prune unnecessary window expressions
    case p @ Project(_, w: Window) if (w.windowOutputSet -- p.references).nonEmpty =>
      p.copy(child = w.copy(
        windowExpressions = w.windowExpressions.filter(p.references.contains)))

    // Eliminate no-op Window
    case w: Window if w.windowExpressions.isEmpty => w.child

    // Eliminate no-op Projects
    case p @ Project(_, child) if sameOutput(child.output, p.output) => child

    // Can't prune the columns on LeafNode
    case p @ Project(_, _: LeafNode) => p

    // for all other logical plans that inherits the output from it's children
    case p @ Project(_, child) =>
      val required = child.references ++ p.references
      if ((child.inputSet -- required).nonEmpty) {
        val newChildren = child.children.map(c => prunedChild(c, required))
        p.copy(child = child.withNewChildren(newChildren))
      } else {
        p
      }
  })
```

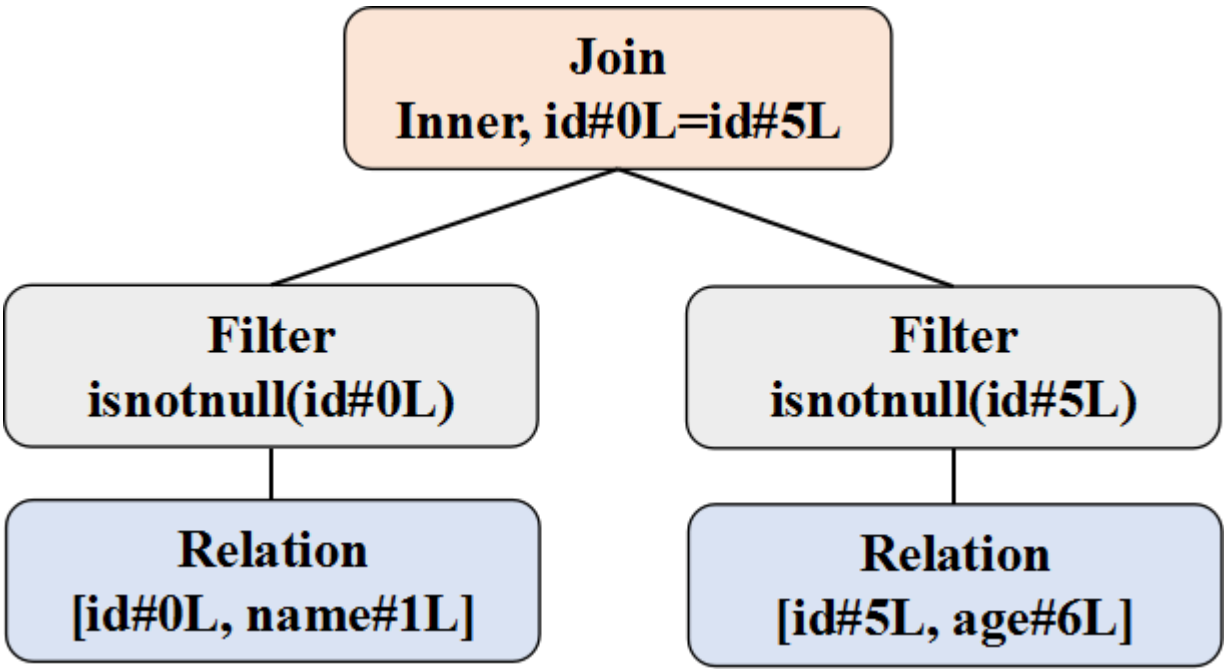


InferFiltersFromConstraints：过滤条件相关

```scala
object InferFiltersFromConstraints extends Rule[LogicalPlan] with PredicateHelper {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case filter @ Filter(condition, child) =>
      val newFilters = filter.constraints --
        (child.constraints ++ splitConjunctivePredicates(condition))
      if (newFilters.nonEmpty) {
        Filter(And(newFilters.reduce(And), condition), child)
      } else {
        filter
      }

    case join @ Join(left, right, joinType, conditionOpt) =>
      // Only consider constraints that can be pushed down completely to either the left or the
      // right child
      val constraints = join.constraints.filter { c =>
        c.references.subsetOf(left.outputSet) || c.references.subsetOf(right.outputSet)
      }
      // Remove those constraints that are already enforced by either the left or the right child
      val additionalConstraints = constraints -- (left.constraints ++ right.constraints)
      val newConditionOpt = conditionOpt match {
        case Some(condition) =>
          val newFilters = additionalConstraints -- splitConjunctivePredicates(condition)
          if (newFilters.nonEmpty) Option(And(newFilters.reduce(And), condition)) else None
        case None =>
          additionalConstraints.reduceOption(And)
      }
      if (newConditionOpt.isDefined) Join(left, right, joinType, newConditionOpt) else join
  }
}
```



PushPredicateThroughJoin：谓词下推



到这一步，逻辑算子树的生成处理已经完毕，下一步开始生成物理算子树，并为开始执行做准备。

6.物理算子树(Physical Plan)

此时可以看到一个`strategies`变量，其形式与之前在`Analyzer`和`Optimizer`里看到的`batches`变量十分相似。除此之外，我们并未看到`SparkPlanner`实现`plan`方法。这并不奇怪，毕竟`Analyzer`和`Optimizer`也没有实现`execute`方法。那我们先去看看`SparkPlanner`的父类`SparkStrategies`：

似乎`SparkStrategies`并未定义任何函数，倒是定义了大量的`Strategy`子类，这些子类都被应用在了`SQLContext#SparkPlanner`中。那么看来，这个类确实是名符其实的`SparkStrategies`。

从LogicalPlan进行分析，Join操作的LogicalPlan有多种类型，主要包含ExtractEquiJoinKeys，Logical.Join类型。从PhysicalPlan中的JoinSelection入手来看。ExtractEquiJoinKeys主要是用于equi-Join，而没有Join key或者Inner-Join的时候会用Logical.Join。以常见的equi-Join为对象，即ExtractEquiJoinKeys，进行分析。

```
object ExtractEquiJoinKeys extends Logging with PredicateHelper {
  /** (joinType, leftKeys, rightKeys, condition, leftChild, rightChild) */
  type ReturnType =
    (JoinType, Seq[Expression], Seq[Expression], Option[Expression], LogicalPlan, LogicalPlan)
  def unapply(plan: LogicalPlan): Option[ReturnType] = plan match {
    case join @ Join(left, right, joinType, condition) =>
      logDebug(s"Considering join on: $condition")
      val predicates = condition.map(splitConjunctivePredicates).getOrElse(Nil)
      val joinKeys = predicates.flatMap {
        case EqualTo(l, r) if l.references.isEmpty || r.references.isEmpty => None
        case EqualTo(l, r) if canEvaluate(l, left) && canEvaluate(r, right) => Some((l, r))
        case EqualTo(l, r) if canEvaluate(l, right) && canEvaluate(r, left) => Some((r, l))
        case EqualNullSafe(l, r) if canEvaluate(l, left) && canEvaluate(r, right) =>
          Some((Coalesce(Seq(l, Literal.default(l.dataType))),
            Coalesce(Seq(r, Literal.default(r.dataType)))))
        case EqualNullSafe(l, r) if canEvaluate(l, right) && canEvaluate(r, left) =>
          Some((Coalesce(Seq(r, Literal.default(r.dataType))),
            Coalesce(Seq(l, Literal.default(l.dataType)))))
        case other => None
      }
      val otherPredicates = predicates.filterNot {
        case EqualTo(l, r) if l.references.isEmpty || r.references.isEmpty => false
        case EqualTo(l, r) =>
          canEvaluate(l, left) && canEvaluate(r, right) ||
            canEvaluate(l, right) && canEvaluate(r, left)
        case other => false
      }
      if (joinKeys.nonEmpty) {
        val (leftKeys, rightKeys) = joinKeys.unzip
        logDebug(s"leftKeys:$leftKeys | rightKeys:$rightKeys")
        Some((joinType, leftKeys, rightKeys, otherPredicates.reduceOption(And), left, right))
      } else {
        None
      }
    case _ => None
  }
}
```

首先就是针对Join操作中的连接条件进行提取，如果是Equi-Join，就将左右子节点的Join Key都提取出来。这里有两种情况：EqualTo和EqualNullSafe，这两者的区别在于对空值是否敏感。EqualTo对空值是敏感的，也就是说对于空值没有额外的处理，而EqualNullSafe情况下的处理逻辑基本和EqualTo一样，但是它会对空值做处理，即赋予相应类型的默认值。那么什么情况下会使用这两种情况呢？实际是用户指定的，条件表达式为“=”或“==”时使用EqualTo，当为“<=>”使用EqualNullSafe。
`otherPredicates`是记录除了EqualTo类型之外的条件表达式，这里主要是除EqualTo之外的表达式（求值），还有就是EqualNullSafe表达式（这里将EqualNullSafe再次加入`otherPredicates`的目的是）。之后生成的结果就是提取出来的Equi-Join Key，并且把其他连接条件也提取出来。

所以`otherPredicates`中的内容基本上可以Shuffle之后在各个数据集上分别处理。

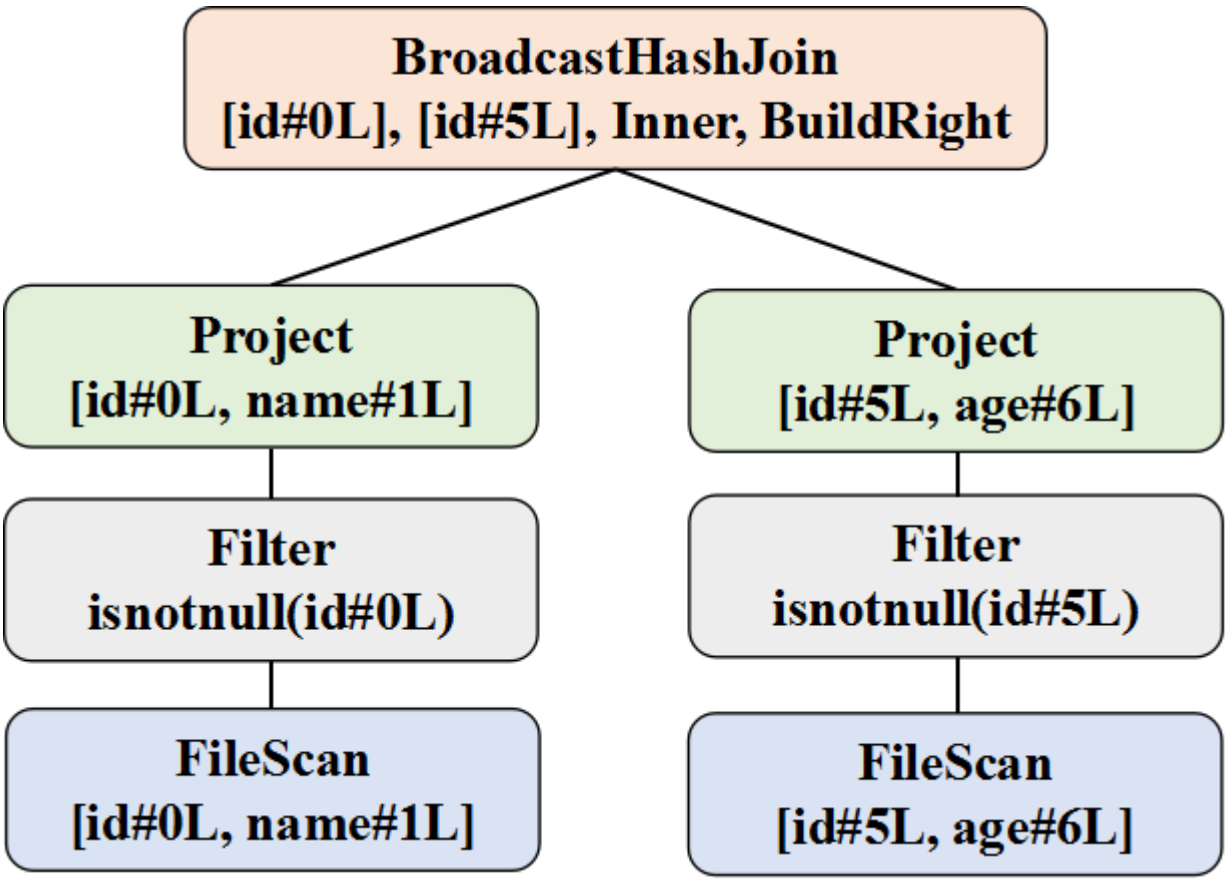
由于在没有特殊设置的情况下会调用SortMergeJoin，所以进入SortMergeJoinExec，传入的参数包括left child和right child，以及对应的join key，还有约束条件。

PhysicalPlan树结点的类型为`SparkPlan`。于是我们查看它的源代码：

注意到，在JoinSelection的apply中，当遇到标记为Join的Logical Plan时，它的做法是先对左右子树分别调用planLater得到结果后， 再构造BroadcastHashJoinExec。而planLater又会调用plan，这意味着每一次调用planLater实际上都是一次递归，这是一个先序遍历。 planLater的实现是this.plan(plan).next()，意味着即使strategies中可应用于传入子树的策略不止一个，返回的Physical Plan数也可能不止一个（注意Strategy的apply函数返回的是个Seq）， 但planLater都只取第一个。

这里就是这个先序遍历开始的地方，同样使用了和`planLater`一样的调用方式，这就证明了我的猜想。这同时说明，尽管Spark可以为同一个Logical Plan生成多个Physical Plan， 但本该在这些Physical Plan中选出最低代价执行计划的功能并未实现。在`LogicalPlan`中我们有看到过疑似要用于cost-based优化的`Statistics`变量，但在Physical Plan这边实际上我们并未见到它的身影， 而且`Statistics`类本身的设计也过于简单（它是一个只包含了一个`BigInt`变量的case class，并未继承任何类）。

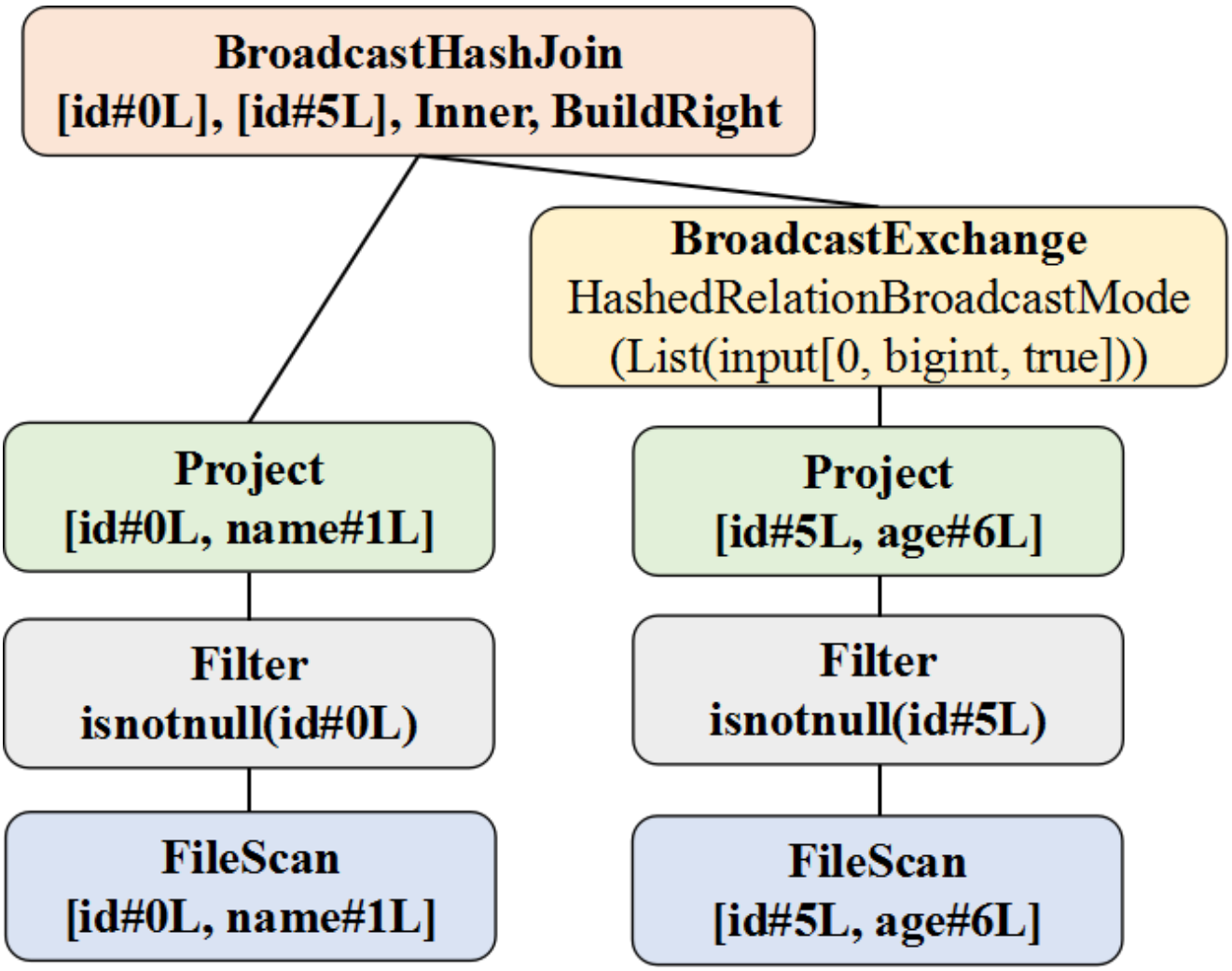
生成的物理算子树如下：



但这毕竟是不能怪SparkSQL的，查询代价受环境的影响很大，比起rule-based优化来说，cost-based太过不稳定，实现起来也复杂很多。不过不管怎么说，SparkSQL仍然留下了可用于实现cost-based优化的接口， 也许有朝一日这个功能真的会实现。

7.执行计划生成

生成的最终执行计划(本质上也是物理执行树)如下：



8.具体执行过程

plan.execute()调用的效果：这时会在driver端，递归的触发物理执行计划的doExecute()方法，这些方法一般都是返回对应的RDD。

9.总结

总体而言，传统数据库单机环境在查询优化时会尽量减少使用Join。然而大数据场景下，数据建模与实际业务意味着表与表之间的关联难以避免，对Join操作的支持与优化力度决定了系统的性能。