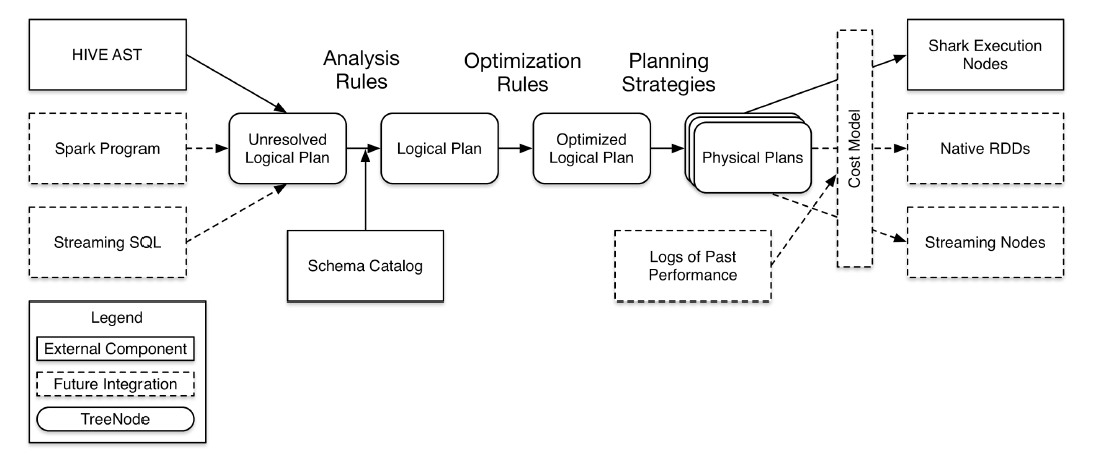
# Architecture



* 把输入的SQL，parser成unresolved logical plan，这一步参考SqlParser的实现
* 把unresolved logical plan转化成resolved logical plan，这一步参考analysis的实现
* 把resolved logical plan转化成optimized logical plan，这一步参考optimize的实现
* 把optimized logical plan转化成physical plan，这一步参考QueryPlanner Strategy的实现

# Source Code Module

## Rule & RuleExecutor

Rule是一个抽象类，拥有一个名字，默认为类名。Rule的实现有很多，渗透在不同的处理过程(analyze, optimize)里。

RuleExecutor是规则执行类，他的下面两个实现会具体讲：

[Analyzer](#_Analyzer)

[Optimizer](#_Optimizer)

RuleExecutor 支持的策略：一次或多次。用来控制converge结束的条件。这里的Strategy名字感觉有点误导人

/\*\*

\* An execution strategy for rules that indicates the maximum number of executions. If the

\* execution reaches fix point (i.e. converge) before maxIterations, it will stop.

\*/

**abstract** **class** Strategy { **def** maxIterations: Int }

/\*\* A strategy that only runs once. \*/

**case** **object** Once **extends** Strategy { **val** maxIterations = 1 }

/\*\* A strategy that runs until fix point or maxIterations times, whichever comes first. \*/

**case** **class** FixedPoint(maxIterations: Int) **extends** Strategy

RuleExecutor的Batch类和batches变量：

/\*\* A batch of rules. \*/

**protected** **case** **class** Batch(name: String, strategy: Strategy, rules: Rule[TreeType]\*)

/\*\* Defines a sequence of rule batches, to be overridden by the implementation. \*/

**protected** **val** batches: Seq[Batch]

每一个batch有多个Rule

batches在apply()里执行的时候，把一个plan丢进来后，用左折叠处理每个batch，最后吐出来一个plan。

converge的条件是达到最大策略次数或者两个TreeNode相等。

/\*\*

\* Executes the batches of rules defined by the subclass. The batches are executed serially

\* using the defined execution strategy. Within each batch, rules are also executed serially.

\*/

**def** apply(plan: TreeType): TreeType = {

**var** curPlan = plan

batches.foreach { batch =>

**var** iteration = 1

**var** lastPlan = curPlan

curPlan = batch.rules.foldLeft(curPlan) { **case** (plan, rule) => rule(plan) }

// Run until fix point (or the max number of iterations as specified in the strategy.

**while** (iteration < batch.strategy.maxIterations && !curPlan.fastEquals(lastPlan)) {

lastPlan = curPlan

curPlan = batch.rules.foldLeft(curPlan) {

**case** (plan, rule) =>

**val** result = rule(plan)

**if** (!result.fastEquals(plan)) {

logger.debug(...)

}

result

}

iteration += 1

}

}

curPlan

}

下面两节具体介绍RuleExecutor的实现

### Analyzer

Analyzer使用于对最初的unresolved logical plan转化成为logical plan。这部分的分析会涵盖整个analysis package。

作用是把未确定的属性和关系，通过Schema信息（来自于Catalog类）和方法注册类来确定下来，这个过程中有三步，第三步会包含许多次的迭代。

/\*\*

\* Provides a logical query plan analyzer, which translates [[UnresolvedAttribute]]s and

\* [[UnresolvedRelation]]s into fully typed objects using information in a schema [[Catalog]] and

\* a [[FunctionRegistry]].

\*/

**class** Analyzer(catalog: Catalog, registry: FunctionRegistry, caseSensitive: Boolean)

**extends** RuleExecutor[LogicalPlan] **with** HiveTypeCoercion {

首先，Catalog类是一个记录表信息的类，专门提供给Analyzer用。

**trait** Catalog {

**def** lookupRelation(

databaseName: Option[String],

tableName: String,

alias: Option[String] = None): LogicalPlan

**def** registerTable(databaseName: Option[String], tableName: String, plan: LogicalPlan): Unit

}

看一个SimpleCatalog的实现，该类在SQLContext里使用，把表名和LogicalPlan存在HashMap里维护起来，生命周期随上下文。提供注册表、删除表、查找表的功能。

**class** SimpleCatalog **extends** Catalog {

**val** tables = **new** mutable.HashMap[String, LogicalPlan]()

**def** registerTable(databaseName: Option[String],tableName: String, plan: LogicalPlan): Unit = {

tables += ((tableName, plan))

}

**def** dropTable(tableName: String) = tables -= tableName

**def** lookupRelation(

databaseName: Option[String],

tableName: String,

alias: Option[String] = None): LogicalPlan = {

**val** table = tables.get(tableName).getOrElse(sys.error(s"Table Not Found: $tableName"))

// If an alias was specified by the lookup, wrap the plan in a subquery so that attributes are

// properly qualified with this alias.

alias.map(a => Subquery(a.toLowerCase, table)).getOrElse(table)

}

}

在查找的时候可以代入一个别名，会把他包装成一个Subquery。Subquery是个简单的case class。

**case** **class** Subquery(alias: String, child: LogicalPlan) **extends** UnaryNode {

**def** output = child.output.map(\_.withQualifiers(alias :: Nil))

**def** references = Set.empty

}

FunctionRegistry类似于Catalog，记录的是函数，在hive package里，处理的是Hive的UDF

**trait** FunctionRegistry {

**def** lookupFunction(name: String, children: Seq[Expression]): Expression

}

FunctionRegistry的实现目前只有一个，如下，如果你要查找方法，就会抛异常。

/\*\*

\* A trivial catalog that returns an error when a function is requested. Used for testing when all

\* functions are already filled in and the analyser needs only to resolve attribute references.

\*/

**object** EmptyFunctionRegistry **extends** FunctionRegistry {

**def** lookupFunction(name: String, children: Seq[Expression]): Expression = {

**throw** **new** UnsupportedOperationException

}

}

回到Analyzer，SQLContext在使用Analyzer前，就是这样生成的：

@transient

**protected**[sql] **lazy** **val** catalog: Catalog = **new** SimpleCatalog

**protected**[sql] **lazy** **val** analyzer: Analyzer =

**new** Analyzer(catalog, EmptyFunctionRegistry, caseSensitive = **true**)

接下来看Catalyst现在的Analyzer作为一个RuleExecutor，已经实现的功能：

**class** Analyzer(catalog: Catalog, registry: FunctionRegistry, caseSensitive: Boolean)

**extends** RuleExecutor[LogicalPlan] **with** HiveTypeCoercion {

// TODO: pass this in as a parameter.

**val** fixedPoint = FixedPoint(100)

**val** batches: Seq[Batch] = Seq(

Batch("MultiInstanceRelations", Once,

NewRelationInstances),

Batch("CaseInsensitiveAttributeReferences", Once,

(**if** (caseSensitive) Nil **else** LowercaseAttributeReferences :: Nil) : \_\*),

Batch("Resolution", fixedPoint,

ResolveReferences ::

ResolveRelations ::

NewRelationInstances ::

ImplicitGenerate ::

StarExpansion ::

ResolveFunctions ::

GlobalAggregates ::

typeCoercionRules :\_\*)

)

#### Batch One

首先是第一个batch里的NewRelationInstance这条Rule，他的作用就是避免一个逻辑计划上同一个实例出现多次，如果出现就生成一个新的plan，保证每个表达式id都唯一。

/\*\*

\* If any MultiInstanceRelation appears more than once in the query plan then the plan is updated so

\* that each instance has unique expression ids for the attributes produced.

\*/

**object** NewRelationInstances **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = {

**val** localRelations = plan collect { **case** l: MultiInstanceRelation => l} // 这一步是搜集所有的多实例关系

**val** multiAppearance = localRelations

.groupBy(identity[MultiInstanceRelation])

.filter { **case** (\_, ls) => ls.size > 1 }

.map(\_.\_1)

.toSet // 这一步是做过滤

plan transform { // 这一步是把原来plan里的多实例关系，凡是出现多个，就变成一个新的单一实例

**case** l: MultiInstanceRelation **if** multiAppearance contains l => l.newInstance

}

}

}

LogicalPlan本身是TreeNode的子类，TreeNode具备collect等一些scala collection操作的能力，这个例子里第一步搜集的过程中体现了collect能力。

#### Batch Two

第二个batch是大小写相关的，如果对大小写不敏感，那么就执行LowercaseAttributeReferences这条Rule，会把所有的属性都变成小写

/\*\*

\* Makes attribute naming case insensitive by turning all UnresolvedAttributes to lowercase.

\*/

**object** LowercaseAttributeReferences **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** UnresolvedRelation(databaseName, name, alias) => // 第一类：未确定的关系

UnresolvedRelation(databaseName, name, alias.map(\_.toLowerCase))

**case** Subquery(alias, child) => Subquery(alias.toLowerCase, child) // 第二类：子查询

**case** q: LogicalPlan => q transformExpressions { // 第三类： 其他类型

**case** s: Star => s.copy(table = s.table.map(\_.toLowerCase)) // 指的是 \* 号

**case** UnresolvedAttribute(name) => UnresolvedAttribute(name.toLowerCase) // 未确定的属性

**case** Alias(c, name) => Alias(c, name.toLowerCase)() // 别名

}

}

}

transform，transformExpressions是TreeNode提供的方法，用于前序遍历树(pre-order)。

从这个处理可以看到logicalPlan里面包含的种类。后续Expression这一块具体还要展开介绍。

Alias的一点说明：

/\*\*

\* Used to assign a new name to a computation.

\* For example the SQL expression "1 + 1 AS a" could be represented as follows:

\* Alias(Add(Literal(1), Literal(1), "a")()

\*

#### Batch Three

Resulotion是第三类batch，定义的结束条件是循环100次。下面依次把各个Rule介绍一遍。

Batch("Resolution", fixedPoint,

ResolveReferences :: // 确定属性

ResolveRelations :: // 确定关系（从catalog里）

NewRelationInstances :: // 去掉同一个实例出现多次的情况

ImplicitGenerate :: // 把包含Generator且只有一条的表达式转化成Generate操作

StarExpansion :: // 扩张 \*

ResolveFunctions :: // 确定方法（从FunctionRegistry里）

GlobalAggregates :: // 把包含Aggregate的表达式转化成Aggregate操作

typeCoercionRules :\_\*) // 来自于HiveTypeCoercion，主要针对Hive语法做强制转换，包含多种规则

用post-order遍历树，把未确定的属性确定下来。如果没有做成功，未确定的属性依然会留下来，留给下一次迭代的时候再确定。

/\*\*

\* Replaces [[UnresolvedAttribute]]s with concrete

\* [[expressions.AttributeReference AttributeReferences]] from a logical plan node's children.

\*/

**object** ResolveReferences **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transformUp {

**case** q: LogicalPlan **if** q.childrenResolved =>

logger.trace(s"Attempting to resolve ${q.simpleString}")

q transformExpressions {

**case** u @ UnresolvedAttribute(name) =>

// Leave unchanged if resolution fails. Hopefully will be resolved next round.

**val** result = q.resolve(name).getOrElse(u)

logger.debug(s"Resolving $u to $result")

result

}

}

}

确定是通过LogicalPlan的resolve方法做的。这个具体在LogicalPlan里介绍。

从catalog里查找关系

/\*\*

\* Replaces [[UnresolvedRelation]]s with concrete relations from the catalog.

\*/

**object** ResolveRelations **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** UnresolvedRelation(databaseName, name, alias) =>

catalog.lookupRelation(databaseName, name, alias)

}

}

Generator是表达式的一种，根据一种input row产生0个或多个rows。

/\*\*

\* When a SELECT clause has only a single expression and that expression is a

\* [[catalyst.expressions.Generator Generator]] we convert the

\* [[catalyst.plans.logical.Project Project]] to a [[catalyst.plans.logical.Generate Generate]].

\*/

**object** ImplicitGenerate **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** Project(Seq(Alias(g: Generator, \_)), child) =>

Generate(g, join = **false**, outer = **false**, None, child)

}

}

这部分也没仔细看。

/\*\*

\* Expands any references to [[Star]] (\*) in project operators.

\*/

**object** StarExpansion **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

// Wait until children are resolved

**case** p: LogicalPlan **if** !p.childrenResolved => p

// If the projection list contains Stars, expand it.

**case** p @ Project(projectList, child) **if** containsStar(projectList) =>

Project(

projectList.flatMap {

**case** s: Star => s.expand(child.output)

**case** o => o :: Nil

},

child)

**case** t: ScriptTransformation **if** containsStar(t.input) =>

t.copy(

input = t.input.flatMap {

**case** s: Star => s.expand(t.child.output)

**case** o => o :: Nil

}

)

// If the aggregate function argument contains Stars, expand it.

**case** a: Aggregate **if** containsStar(a.aggregateExpressions) =>

a.copy(

aggregateExpressions = a.aggregateExpressions.flatMap {

**case** s: Star => s.expand(a.child.output)

**case** o => o :: Nil

}

)

}

/\*\*

\* Returns true if `exprs` contains a [[Star]].

\*/

**protected** **def** containsStar(exprs: Seq[Expression]): Boolean =

exprs.collect { **case** \_: Star => **true** }.nonEmpty

}

确定方法类似确定关系。

/\*\*

\* Replaces [[UnresolvedFunction]]s with concrete [[expressions.Expression Expressions]].

\*/

**object** ResolveFunctions **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** q: LogicalPlan =>

q transformExpressions {

**case** u @ UnresolvedFunction(name, children) **if** u.childrenResolved =>

registry.lookupFunction(name, children)

}

}

}

把包含Aggregate的表达式转化成Aggregate操作

/\*\*

\* Turns projections that contain aggregate expressions into aggregations.

\*/

**object** GlobalAggregates **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** Project(projectList, child) **if** containsAggregates(projectList) =>

Aggregate(Nil, projectList, child)

}

**def** containsAggregates(exprs: Seq[Expression]): Boolean = {

exprs.foreach(\_.foreach {

**case** agg: AggregateExpression => **return** **true**

**case** \_ =>

})

**false**

}

}

换针对Hive语法做强制转换，规则如下

**trait** HiveTypeCoercion {

**val** typeCoercionRules = List(PropagateTypes, ConvertNaNs, WidenTypes, PromoteStrings, BooleanComparisons, BooleanCasts, StringToIntegralCasts, FunctionArgumentConversion)

举个简单的例子来看下表达式的使用和替换：

/\*\*

\* Converts string "NaN"s that are in binary operators with a NaN-able types (Float / Double) \* to the appropriate numeric equivalent.

\*/

**object** ConvertNaNs **extends** Rule[LogicalPlan] {

**val** stringNaN = Literal("NaN", StringType)

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** q: LogicalPlan => q transformExpressions {

// Skip nodes who's children have not been resolved yet.

**case** e **if** !e.childrenResolved => e

/\* Double Conversions \*/

**case** b: BinaryExpression **if** b.left == stringNaN && b.right.dataType == DoubleType =>

b.makeCopy(Array(b.right, Literal(Double.NaN)))

**case** b: BinaryExpression **if** b.left.dataType == DoubleType && b.right == stringNaN =>

b.makeCopy(Array(Literal(Double.NaN), b.left))

**case** b: BinaryExpression **if** b.left == stringNaN && b.right == stringNaN =>

b.makeCopy(Array(Literal(Double.NaN), b.left))

/\* Float Conversions \*/

**case** b: BinaryExpression **if** b.left == stringNaN && b.right.dataType == FloatType =>

b.makeCopy(Array(b.right, Literal(Float.NaN)))

**case** b: BinaryExpression **if** b.left.dataType == FloatType && b.right == stringNaN =>

b.makeCopy(Array(Literal(Float.NaN), b.left))

**case** b: BinaryExpression **if** b.left == stringNaN && b.right == stringNaN =>

b.makeCopy(Array(Literal(Float.NaN), b.left))

}

}

}

### Optimizer

Optimizer用于把analyzed plan转化成为optimized plan。目前Catalyst的optimizer包下就这一个类，SQLContext也是直接使用的这个类。

同样，我们看一下里面包括了哪些处理过程：

**object** Optimizer **extends** RuleExecutor[LogicalPlan] {

**val** batches =

Batch("Subqueries", Once,

EliminateSubqueries) ::

Batch("ConstantFolding", Once,

ConstantFolding,

BooleanSimplification,

SimplifyCasts) ::

Batch("Filter Pushdown", Once,

EliminateSubqueries,

CombineFilters,

PushPredicateThroughProject,

PushPredicateThroughInnerJoin) :: Nil

}

#### Batch One

和子查询相关的一批规则，包含一条消除子查询的规则：EliminateSubqueries

/\*\*

\* Removes [[catalyst.plans.logical.Subquery Subquery]] operators from the plan. Subqueries are

\* only required to provide scoping information for attributes and can be removed once analysis is

\* complete.

\*/

**object** EliminateSubqueries **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** Subquery(\_, child) => child // 处理方式是凡是带child的，都用child替换自己

}

}

注释提到，过了analysis这一步之后，子查询就可以移除了。

#### Batch Two

第二批规则，常量折叠。

Batch("ConstantFolding", Once,

ConstantFolding, // 常量折叠

BooleanSimplification, // 提早短路掉布尔表达式

SimplifyCasts) // 去掉多余的Cast操作

/\*\*

\* Replaces [[catalyst.expressions.Expression Expressions]] that can be statically evaluated with

\* equivalent [[catalyst.expressions.Literal Literal]] values.

\*/

**object** ConstantFolding **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** q: LogicalPlan => q transformExpressionsDown {

// Skip redundant folding of literals.

**case** l: Literal => l

**case** e **if** e.foldable => Literal(e.apply(**null**), e.dataType)

}

}

}

这里不得不提一下foldable字段在Expression类里的定义：

/\*\*

\* Returns true when an expression is a candidate for static evaluation before the query is

\* executed.

\*

\* The following conditions are used to determine suitability for constant folding:

\* - A [[expressions.Coalesce Coalesce]] is foldable if all of its children are foldable

\* - A [[expressions.BinaryExpression BinaryExpression]] is foldable if its both left and right

\* child are foldable

\* - A [[expressions.Not Not]], [[expressions.IsNull IsNull]], or

\* [[expressions.IsNotNull IsNotNull]] is foldable if its child is foldable.

\* - A [[expressions.Literal]] is foldable.

\* - A [[expressions.Cast Cast]] or [[expressions.UnaryMinus UnaryMinus]] is foldable if its

\* child is foldable.

\*/

// TODO: Supporting more foldable expressions. For example, deterministic Hive UDFs.

**def** foldable: Boolean = **false**

好像只有Literal表达式是foldable的，其余表达式必须表达式中每个元素都满足foldable。

第二种规则也好理解，简化布尔表达式。也就是早早地给表达式做一个短路判断。

/\*\*

\* Simplifies boolean expressions where the answer can be determined without evaluating both sides.

\* Note that this rule can eliminate expressions that might otherwise have been evaluated and thus

\* is only safe when evaluations of expressions does not result in side effects.

\*/

**object** BooleanSimplification **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** q: LogicalPlan => q transformExpressionsUp {

**case** and @ And(left, right) =>

(left, right) **match** {

**case** (Literal(**true**, BooleanType), r) => r

**case** (l, Literal(**true**, BooleanType)) => l

**case** (Literal(**false**, BooleanType), \_) => Literal(**false**)

**case** (\_, Literal(**false**, BooleanType)) => Literal(**false**)

**case** (\_, \_) => and

}

**case** or @ Or(left, right) =>

(left, right) **match** {

**case** (Literal(**true**, BooleanType), \_) => Literal(**true**)

**case** (\_, Literal(**true**, BooleanType)) => Literal(**true**)

**case** (Literal(**false**, BooleanType), r) => r

**case** (l, Literal(**false**, BooleanType)) => l

**case** (\_, \_) => or

}

}

}

}

把Cast操作全部移走。

/\*\*

\* Removes [[catalyst.expressions.Cast Casts]] that are unnecessary because the input is already

\* the correct type.

\*/

**object** SimplifyCasts **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpressions {

**case** Cast(e, dataType) **if** e.dataType == dataType => e

}

}

#### Batch Three

一批 过滤下推 规则，

Batch("Filter Pushdown", Once,

EliminateSubqueries, // 消除子查询

CombineFilters, // 过滤操作取合集

PushPredicateThroughProject, // 为映射操作下推谓词

PushPredicateThroughInnerJoin) // 为inner join下推谓词

把相邻的过滤操作取 (AND)

/\*\*

\* Combines two adjacent [[catalyst.plans.logical.Filter Filter]] operators into one, merging the

\* conditions into one conjunctive predicate.

\*/

**object** CombineFilters **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** ff @ Filter(fc, nf @ Filter(nc, grandChild)) => Filter(And(nc, fc), grandChild)

}

}

在映射操作里下推谓词

/\*\*

\* Pushes [[catalyst.plans.logical.Filter Filter]] operators through

\* [[catalyst.plans.logical.Project Project]] operators, in-lining any

\* [[catalyst.expressions.Alias Aliases]] that were defined in the projection.

\*

\* This heuristic is valid assuming the expression evaluation cost is minimal.

\*/

**object** PushPredicateThroughProject **extends** Rule[LogicalPlan] {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** filter @ Filter(condition, project @ Project(fields, grandChild)) =>

**val** sourceAliases = fields.collect { **case** a @ Alias(c, \_) =>

(a.toAttribute: Attribute) -> c

}.toMap

project.copy(child = filter.copy(

replaceAlias(condition, sourceAliases),

grandChild))

}

//

**def** replaceAlias(condition: Expression, sourceAliases: Map[Attribute, Expression]): Expression = {

condition transform {

**case** a: AttributeReference => sourceAliases.getOrElse(a, a)

}

}

}

在inner join操作里下推谓词

/\*\*

\* Pushes down [[catalyst.plans.logical.Filter Filter]] operators where the `condition` can be

\* evaluated using only the attributes of the left or right side of an inner join. Other

\* [[catalyst.plans.logical.Filter Filter]] conditions are moved into the `condition` of the

\* [[catalyst.plans.logical.Join Join]].

\*/

**object** PushPredicateThroughInnerJoin **extends** Rule[LogicalPlan] **with** PredicateHelper {

**def** apply(plan: LogicalPlan): LogicalPlan = plan transform {

**case** f @ Filter(filterCondition, Join(left, right, Inner, joinCondition)) =>

**val** allConditions =

splitConjunctivePredicates(filterCondition) ++

joinCondition.map(splitConjunctivePredicates).getOrElse(Nil)

// Split the predicates into those that can be evaluated on the left, right, and those that

// must be evaluated after the join.

**val** (rightConditions, leftOrJoinConditions) =

allConditions.partition(\_.references subsetOf right.outputSet)

**val** (leftConditions, joinConditions) =

leftOrJoinConditions.partition(\_.references subsetOf left.outputSet)

// Build the new left and right side, optionally with the pushed down filters.

**val** newLeft = leftConditions.reduceLeftOption(And).map(Filter(\_, left)).getOrElse(left)

**val** newRight = rightConditions.reduceLeftOption(And).map(Filter(\_, right)).getOrElse(right)

Join(newLeft, newRight, Inner, joinConditions.reduceLeftOption(And))

}

}

### SQL Context

/\*\*

\* Prepares a planned SparkPlan for execution by binding references to specific ordinals, and

\* inserting shuffle operations as needed.

\*/

@transient

**protected**[sql] **val** prepareForExecution = **new** RuleExecutor[SparkPlan] {

**val** batches =

Batch("Add exchange", Once, AddExchange) ::

Batch("Prepare Expressions", Once, **new** BindReferences[SparkPlan]) :: Nil

}

以上就是Rule包，及RuleExecutor在各处的实现。其中Analyze和Optimize是Catalyst目前提供的，SQL组件直接拿来使用。

## Tree Node

TreeNode Library支持的三个特性：

· Scala collection like methods (foreach, map, flatMap, collect, etc)

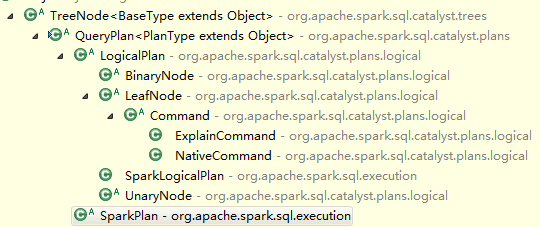
· transform accepts a partial function that is used to generate a new tree.

· debugging support pretty printing, easy splicing of trees, etc.

Collection操作能力

偏函数

继承结构



全局唯一id

**object** TreeNode {

**private** **val** currentId = **new** java.util.concurrent.atomic.AtomicLong

**protected** **def** nextId() = currentId.getAndIncrement()

}

几种节点

/\*\*

\* A [[TreeNode]] that has two children, [[left]] and [[right]].

\*/

**trait** BinaryNode[BaseType <: TreeNode[BaseType]] {

**def** left: BaseType

**def** right: BaseType

**def** children = Seq(left, right)

}

/\*\*

\* A [[TreeNode]] with no children.

\*/

**trait** LeafNode[BaseType <: TreeNode[BaseType]] {

**def** children = Nil

}

/\*\*

\* A [[TreeNode]] with a single [[child]].

\*/

**trait** UnaryNode[BaseType <: TreeNode[BaseType]] {

**def** child: BaseType

**def** children = child :: Nil

}

每个node唯一id，导致在比较的时候，不同分支上长得一样结构的node也不相同，比较如下：

**def** sameInstance(other: TreeNode[\_]): Boolean = {

**this**.id == other.id

}

**def** fastEquals(other: TreeNode[\_]): Boolean = {

sameInstance(other) || **this** == other

}

foreach的时候，先做自己，再把孩子们做一遍

**def** foreach(f: BaseType => Unit): Unit = {

f(**this**)

children.foreach(\_.foreach(f))

}

map的时候是按前序对每个节点都做一次处理

**def** map[A](f: BaseType => A): Seq[A] = {

**val** ret = **new** collection.mutable.ArrayBuffer[A]()

foreach(ret += f(\_))

ret

}

其他的很多变化都类似，接收的是函数或偏函数，把他们作用到匹配的节点上去执行

变化总共有这些，按类别分：

map, flatMap, collect,

mapChildren, withNewChildren,

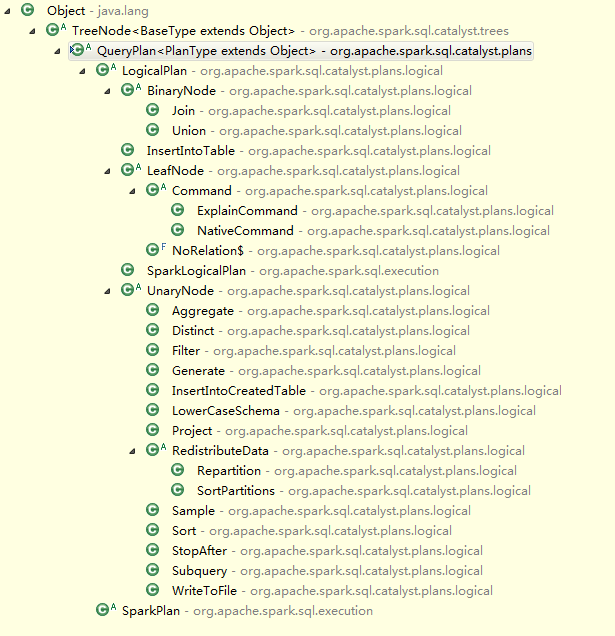
transform, transformDown, transformChildrenDown 前序

       transformUp,  transformChildrenUp   后序

基本上就这些，其实就是提供对这棵树及其子节点的顺序遍历和处理能力

## Plan

QueryPlan的继承结构



QueryPlan提供了三个东西，

* 其一是定义了output，是对外输出的一个属性序列

**def** output: Seq[Attribute]

* 其二是借用TreeNode的那套transform方法，实现了一套transformExpression方法，用途是把partial function遍历到各个子节点上。
* 其三是一个expressions方法，返回Seq[expression]，用于搜集本query里所有的表达式。

QueryPlan在Catalyst里的实现是LogicalPlan，在SQL组件里的实现是SparkPlan，前者主要要被处理、分析和优化，后者是真正被处理执行的，下面简单介绍两者。

### Logical Plan

在QueryPlan上增加的几个属性：

1. references 用于生成output属性列表的参考属性列表

**def** references: Set[Attribute]

1. **lazy** **val** inputSet: Set[Attribute] = children.flatMap(\_.output).toSet
2. 自己及children是否resolved
3. resolve方法，重要，看起来费劲

**def** resolve(name: String): Option[NamedExpression] = {

**val** parts = name.split("\\.")

// Collect all attributes that are output by this nodes children where either the first part

// matches the name or where the first part matches the scope and the second part matches the

// name. Return these matches along with any remaining parts, which represent dotted access to

// struct fields.

**val** options = children.flatMap(\_.output).flatMap { option =>

// If the first part of the desired name matches a qualifier for this possible match, drop it.

**val** remainingParts = **if** (option.qualifiers contains parts.head) parts.drop(1) **else** parts

**if** (option.name == remainingParts.head) (option, remainingParts.tail.toList) :: Nil **else** Nil

}

options.distinct **match** {

**case** (a, Nil) :: Nil => Some(a) // One match, no nested fields, use it.

// One match, but we also need to extract the requested nested field.

**case** (a, nestedFields) :: Nil =>

a.dataType **match** {

**case** StructType(fields) =>

Some(Alias(nestedFields.foldLeft(a: Expression)(GetField), nestedFields.last)())

**case** \_ => None // Don't know how to resolve these field references

}

**case** Nil => None // No matches.

**case** ambiguousReferences =>

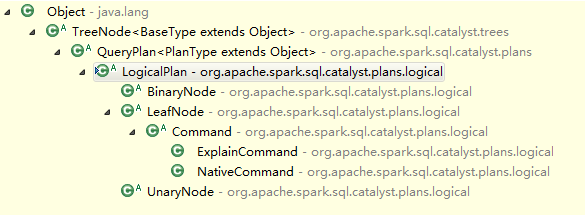
**throw** **new** TreeNodeException(

**this**, s"Ambiguous references to $name: ${ambiguousReferences.mkString(",")}")

}

}

LogicalPlan继承结构



三种Node都继承了LogicalPlan，同时实现的是对应TreeNode节点的接口来区分类别。

三种抽象子类：

/\*\*

\* A logical plan node with no children.

\*/

**abstract** **class** LeafNode **extends** LogicalPlan **with** trees.LeafNode[LogicalPlan] {

self: Product =>

// Leaf nodes by definition cannot reference any input attributes.

**def** references = Set.empty

}

/\*\*

\* A logical plan node with single child.

\*/

**abstract** **class** UnaryNode **extends** LogicalPlan **with** trees.UnaryNode[LogicalPlan] {

self: Product =>

}

/\*\*

\* A logical plan node with a left and right child.

\*/

**abstract** **class** BinaryNode **extends** LogicalPlan **with** trees.BinaryNode[LogicalPlan] {

self: Product =>

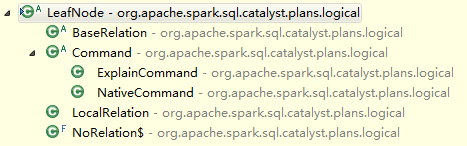
}

#### LogicalPlan impl

分别看LogicalPlan的三种Node的实现：LeafNode，UnaryNode，BinaryNode

##### LeafNode

LeafNode的实现



sql.hive包里，hive表元数据关系的实现



**abstract** **class** BaseRelation **extends** LeafNode {

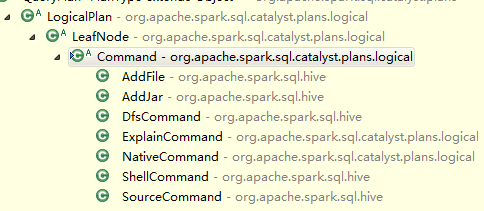
self: Product =>

**def** tableName: String

**def** isPartitioned: Boolean = **false**

}

Hive一些命令 case class



/\*\*

\* A logical node that represents a non-query command to be executed by the system. For example,

\* commands can be used by parsers to represent DDL operations.

\*/

**abstract** **class** Command **extends** LeafNode {

self: Product =>

**def** output = Seq.empty

}

/\*\*

\* Returned for commands supported by a given parser, but not catalyst. In general these are DDL

\* commands that are passed directly to another system.

\*/

**case** **class** NativeCommand(cmd: String) **extends** Command

/\*\*

\* Returned by a parser when the users only wants to see what query plan would be executed, without

\* actually performing the execution.

\*/

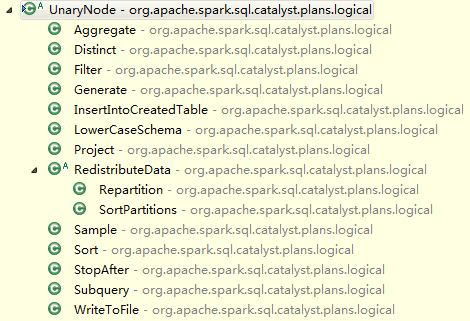
**case** **class** ExplainCommand(plan: LogicalPlan) **extends** Command

**case** **object** NoRelation **extends** LeafNode {

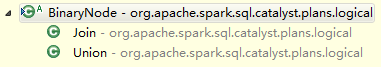
**def** output = Nil

}

##### UnaryNode



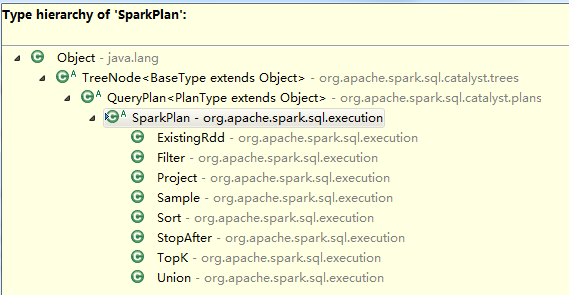
##### BinaryNode



### Spark Plan

SQL组件里的实现

SparkPlan类继承结构如下图：



在SQL模块的execution package的basicOperator类里，有许多SparkPlan的实现，包括

Project，Filter，Sample，Union，StopAfter，TopK，Sort，ExsitingRdd

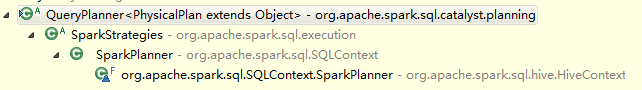
这些实现和Catalyst的basicOperator类里有很多重了，区别在于，SparkPlan是QueryPlan的实现，同logical plan不同的是，SparkPlan会被Spark实现的Strategy真正执行，所以SQL模块里的basicOperator内的这些case class，比Catalyst多了execute()方法

具体Spark策略的实现参考下一小节。

## Planning

### QueryPlanner

QueryPlanner的职责是把逻辑执行计划转化成为物理执行计划，具备一系列Strategy的实现。



**abstract** **class** QueryPlanner[PhysicalPlan <: TreeNode[PhysicalPlan]] {

/\*\* A list of execution strategies that can be used by the planner \*/

**def** strategies: Seq[Strategy]

/\*\*

\* Given a [[plans.logical.LogicalPlan LogicalPlan]], returns a list of `PhysicalPlan`s that can

\* be used for execution. If this strategy does not apply to the give logical operation then an

\* empty list should be returned.

\*/

**abstract** **protected** **class** Strategy **extends** Logging {

**def** apply(plan: LogicalPlan): Seq[PhysicalPlan]

}

/\*\*

\* Returns a placeholder for a physical plan that executes `plan`. This placeholder will be

\* filled in automatically by the QueryPlanner using the other execution strategies that are

\* available.

\*/

**protected** **def** planLater(plan: LogicalPlan) = apply(plan).next()

**def** apply(plan: LogicalPlan): Iterator[PhysicalPlan] = {

// Obviously a lot to do here still...

**val** iter = strategies.view.flatMap(\_(plan)).toIterator

assert(iter.hasNext, s"No plan for $plan")

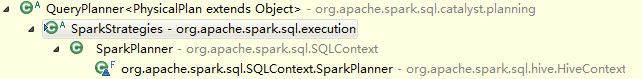
iter

}

}

### QueryPlanner impl

目前的实现是SparkStrategies



在SQLContext里的使用是SparkPlanner：

**protected**[sql] **class** SparkPlanner **extends** SparkStrategies {

**val** sparkContext = self.sparkContext

**val** strategies: Seq[Strategy] =

TopK ::

PartialAggregation ::

SparkEquiInnerJoin ::

BasicOperators ::

CartesianProduct ::

BroadcastNestedLoopJoin :: Nil

}

在HiveContext里的使用是带了hive策略的SparkPlanner：

**val** hivePlanner = **new** SparkPlanner **with** HiveStrategies {

**val** hiveContext = self

**override** **val** strategies: Seq[Strategy] = Seq(

TopK,

ColumnPrunings,

PartitionPrunings,

HiveTableScans,

DataSinks,

Scripts,

PartialAggregation,

SparkEquiInnerJoin,

BasicOperators,

CartesianProduct,

BroadcastNestedLoopJoin

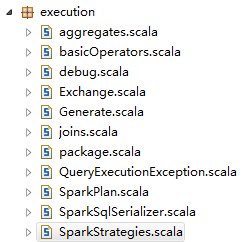
)

}

### Strategy impl

#### Spark Strategy

基本上对应了execution包里的各个类里的实现



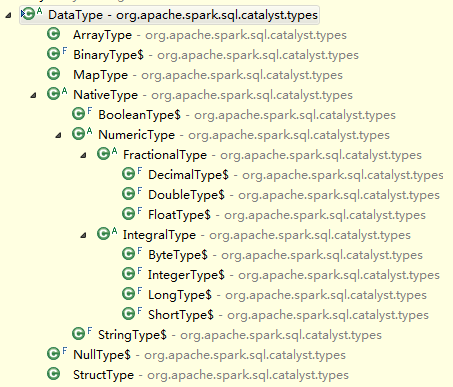
#### Hive Strategy

Hive策略是在Spark策略的基础上附加的一些策略

## Expression

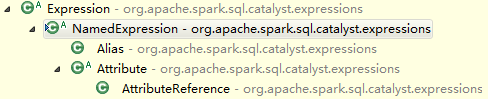
Expression几个属性：

1. 带DataType



并且自带一些inline方法帮助一些dataType的转换

1. 带reference，reference是Seq[Attribute]，Attribute是NamedExpression子类。



NamedExpression是

1. foldable 可折叠的参数描述如下，即静态可以直接执行的表达式

/\*\*

\* Returns true when an expression is a candidate for static evaluation before the query is

\* executed.

\*

\* The following conditions are used to determine suitability for constant folding:

\* - A [[expressions.Coalesce Coalesce]] is foldable if all of its children are foldable

\* - A [[expressions.BinaryExpression BinaryExpression]] is foldable if its both left and right

\* child are foldable

\* - A [[expressions.Not Not]], [[expressions.IsNull IsNull]], or

\* [[expressions.IsNotNull IsNotNull]] is foldable if its child is foldable.

\* - A [[expressions.Literal]] is foldable.

\* - A [[expressions.Cast Cast]] or [[expressions.UnaryMinus UnaryMinus]] is foldable if its

\* child is foldable.

\*/

// TODO: Supporting more foldable expressions. For example, deterministic Hive UDFs.

def foldable: Boolean = false

Expression里只有Literal可折叠，Literal是LeafExpression，根据dataType生成不同类型表达式

**object** Literal {

**def** apply(v: Any): Literal = v **match** {

**case** i: Int => Literal(i, IntegerType)

**case** l: Long => Literal(l, LongType)

**case** d: Double => Literal(d, DoubleType)

**case** f: Float => Literal(f, FloatType)

**case** b: Byte => Literal(b, ByteType)

**case** s: Short => Literal(s, ShortType)

**case** s: String => Literal(s, StringType)

**case** b: Boolean => Literal(b, BooleanType)

**case** **null** => Literal(**null**, NullType)

}

}

**case** **class** Literal(value: Any, dataType: DataType) **extends** LeafExpression {

**override** **def** foldable = **true**

**def** nullable = value == **null**

**def** references = Set.empty

**override** **def** toString = **if** (value != **null**) value.toString **else** "null"

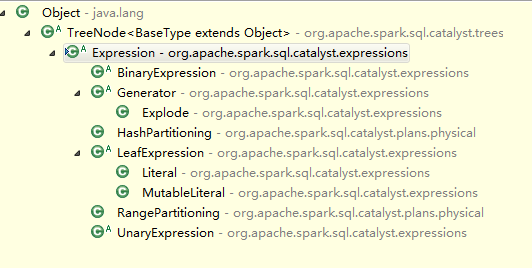
**type** EvaluatedType = Any

**override** **def** apply(input: Row):Any = value // 执行这个叶子表达式的话就是返回value值

}

1. resolved 具体关心children是否都resolved。childeren是TreeNode里的概念，在TreeNode里是一个Seq[BaseType]，而BaseType是TreeNode[T]里的范型。在Expression这里，即TreeNode[Expression]，BaseType就是Expression。

Expression继承结构



抽象子类如下：

**abstract** **class** BinaryExpression **extends** Expression **with** trees.BinaryNode[Expression] {

self: Product =>

**def** symbol: String

**override** **def** foldable = left.foldable && right.foldable

**def** references = left.references ++ right.references

**override** **def** toString = s"($left $symbol $right)"

}

**abstract** **class** LeafExpression **extends** Expression **with** trees.LeafNode[Expression] {

self: Product =>

}

**abstract** **class** UnaryExpression **extends** Expression **with** trees.UnaryNode[Expression] {

self: Product =>

**def** references = child.references

}

### Expression impl

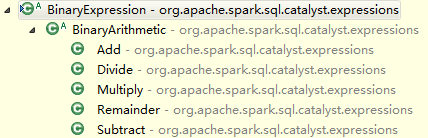
#### LeafExpression



#### UnaryExpression

#### BinaryExpression

##### Arithmetic

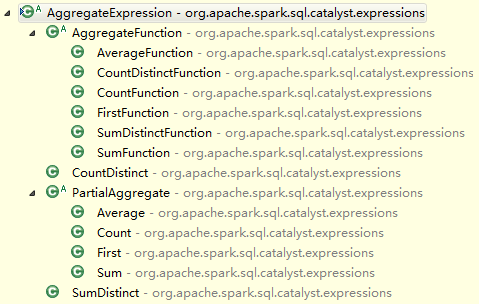


#### Generator

#### Range Partitioning

#### Hash Partitioning

#### Aggregate Expression



## Schema RDD

SchemaRDD是一个RDD[Row]，Row在Catalyst对应的是Table里的一行，定义是

**trait** Row **extends** Seq[Any] **with** Serializable

SchemaRDD就两部分实现，还有几个SQLContext的方法调用

一是RDD的Function的实现

// =========================================================================================

// RDD functions: Copy the interal row representation so we present immutable data to users.

// =========================================================================================

**override** **def** compute(split: Partition, context: TaskContext): Iterator[Row] =

firstParent[Row].compute(split, context).map(\_.copy())

**override** **def** getPartitions: Array[Partition] = firstParent[Row].partitions

**override** **protected** **def** getDependencies: Seq[Dependency[\_]] =

List(**new** OneToOneDependency(queryExecution.toRdd)) // 该SchemaRDD与优化后的RDD是窄依赖

二是DSL function的实现

**def** select(exprs: NamedExpression\*): SchemaRDD =

**new** SchemaRDD(sqlContext, Project(exprs, logicalPlan))

每次DSL的操作会转化成为新的SchemaRDD，

SchemaRDD的DSL操作与Catalyst组件提供的操作的对应关系为

|  |  |  |
| --- | --- | --- |
| SchemaRDD  DSL Operator | Catalyst  basicOperators | LogicalPlan  Sub Node |
| select | Project | UnaryNode |
| where | Filter | UnaryNode |
| join | Join | BinaryNode |
| orderBy | Sort | UnaryNode |
| groupBy | Aggregate | UnaryNode |
| subquery | Subquery | UnaryNode |
| unionAll | Union | UnaryNode |
| sample | Sample | UnaryNode |
| generate | Generate | UnaryNode |
| insertInto | InsertIntoTable | UnaryNode |
|  |  |  |
|  |  |  |
|  |  |  |

接下来具体看每个Operator对应的参数、表达式。

### DSL Operators

output和reference是什么？

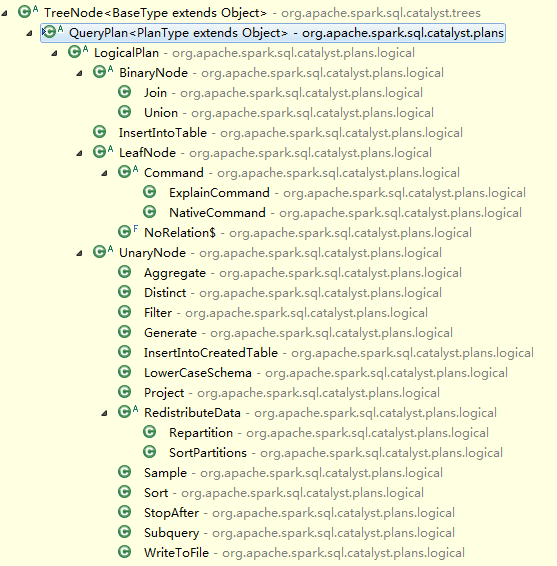
输出的结果属性列表；该类依赖的属性集合

表达式表达了原语之后，怎么被真正执行的？

变成物理执行计划之后，toRdd

DSL Operator的实现都依赖Catalyst的basicOperator，basicOperator里的操作都是LogicalPlan的继承类，主要分两类，一元UnaryNode和二元BinaryNode操作。而UnaryNode和BinaryNode都是TreeNode的实现，TreeNode里还有一种就是LeafNode。

basicOperator的各种实现都是case class，都是LogicalPlan，不具备execute能力



#### select

语句

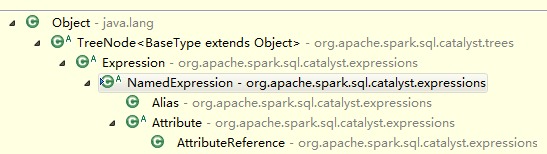
*\* schemaRDD.select('a, 'b + 'c, 'd as 'aliasedName)*

实现

**def** select(exprs: NamedExpression\*): SchemaRDD =

**new** SchemaRDD(sqlContext, Project(exprs, logicalPlan))

select传入的是NamedExpression，具体继承结构如下



代命名的表达式有两类，别名或属性，对应的是Alias和属性实现类AttributeReference，代码略。

Project这个操作代码如下，

**case** **class** Project(projectList: Seq[NamedExpression], child: LogicalPlan) **extends** UnaryNode {

**def** output = projectList.map(\_.toAttribute)

**def** references = projectList.flatMap(\_.references).toSet

}

output对应的是属性列表，references是所有表达式各个reference的集合。references在Expression里的概念是**def** references: Set[Attribute]

#### where

语句

*\* schemaRDD.where('a === 'b)*

*\* schemaRDD.where('a === 1)*

*\* schemaRDD.where('a + 'b > 10)*

实现

**def** where(condition: Expression): SchemaRDD =

**new** SchemaRDD(sqlContext, Filter(condition, logicalPlan))

Filter操作

**case** **class** Filter(condition: Expression, child: LogicalPlan) **extends** UnaryNode {

**def** output = child.output

**def** references = condition.references

}

#### join

实现

**def** join(

otherPlan: SchemaRDD,

joinType: JoinType = Inner,

condition: Option[Expression] = None): SchemaRDD =

**new** SchemaRDD(sqlContext, Join(logicalPlan, otherPlan.logicalPlan, joinType, condition))

joinType分为Inner，LeftOuter，RightOuter，FullOuter，这里默认是inner。实际使用的时候输入otherPlan。

Join操作如下

**case** **class** Join(

left: LogicalPlan,

right: LogicalPlan,

joinType: JoinType,

condition: Option[Expression]) **extends** BinaryNode {

**def** references = condition.map(\_.references).getOrElse(Set.empty)

**def** output = left.output ++ right.output

}

left对应原logicalPlan，right对应otherPlan，type是inner，condition是null。实际的操作都不在这。

#### orderBy

语句

*\* schemaRDD.orderBy('a)*

*\* schemaRDD.orderBy('a, 'b)*

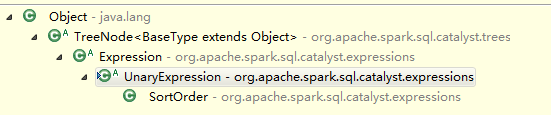
*\* schemaRDD.orderBy('a.asc, 'b.desc)*

实现

**def** orderBy(sortExprs: SortOrder\*): SchemaRDD =

**new** SchemaRDD(sqlContext, Sort(sortExprs, logicalPlan))

SortOrder是一元表达式的一种，类继承关系如下



Sort操作如下

**case** **class** Sort(order: Seq[SortOrder], child: LogicalPlan) **extends** UnaryNode {

**def** output = child.output

**def** references = order.flatMap(\_.references).toSet

}

#### groupBy

语句

*\* schemaRDD.groupBy('year)(Sum('sales) as 'totalSales)*

实现

**def** groupBy(groupingExprs: Expression\*)(aggregateExprs: Expression\*): SchemaRDD = {

**val** aliasedExprs = aggregateExprs.map {

**case** ne: NamedExpression => ne

**case** e => Alias(e, e.toString)()

}

**new** SchemaRDD(sqlContext, Aggregate(groupingExprs, aliasedExprs, logicalPlan))

}

Aggregate操作如下:

**case** **class** Aggregate(

groupingExpressions: Seq[Expression],

aggregateExpressions: Seq[NamedExpression],

child: LogicalPlan)

**extends** UnaryNode {

**def** output = aggregateExpressions.map(\_.toAttribute)

**def** references = child.references

}

#### subquery

语句，适用于给表或结果集起别名

*\* val x = schemaRDD.where('a === 1).subquery('x)*

*\* val y = schemaRDD.where('a === 2).subquery('y)*

*\* x.join(y).where("x.a".attr === "y.a".attr),*

实现

**def** subquery(alias: Symbol) =

**new** SchemaRDD(sqlContext, Subquery(alias.name, logicalPlan))

Subquery操作

**case** **class** Subquery(alias: String, child: LogicalPlan) **extends** UnaryNode {

**def** output = child.output.map(\_.withQualifiers(alias :: Nil))

**def** references = Set.empty

}

#### unionAll

实现，有ALL可能是因为它是保留重名字段的

**def** unionAll(otherPlan: SchemaRDD) =

**new** SchemaRDD(sqlContext, Union(logicalPlan, otherPlan.logicalPlan))

Union操作

**case** **class** Union(left: LogicalPlan, right: LogicalPlan) **extends** BinaryNode {

// TODO: These aren't really the same attributes as nullability etc might change.

**def** output = left.output

**override** **lazy** **val** resolved =

childrenResolved &&

!left.output.zip(right.output).exists { **case** (l,r) => l.dataType != r.dataType }

**def** references = Set.empty

}

#### sample

实现

**def** sample(

fraction: Double,

withReplacement: Boolean = **true**,

seed: Int = (math.random \* 1000).toInt) =

**new** SchemaRDD(sqlContext, Sample(fraction, withReplacement, seed, logicalPlan))

Sample操作

**case** **class** Sample(fraction: Double, withReplacement: Boolean, seed: Int, child: LogicalPlan)

**extends** UnaryNode {

**def** output = child.output

**def** references = Set.empty

}

#### generate

实现及初始化参数描述

\* **@param** generator A table generating function. The API for such functions is likely to change

\* in future releases

\* **@param** join when set to true, each output row of the generator is joined with the input row

\* that produced it.

\* **@param** outer when set to true, at least one row will be produced for each input row, similar to

\* an `OUTER JOIN` in SQL. When no output rows are produced by the generator for a

\* given row, a single row will be output, with `NULL` values for each of the

\* generated columns.

\* **@param** alias an optional alias that can be used as qualif for the attributes that are produced

\* by this generate operation.

**def** generate(

generator: Generator,

join: Boolean = **false**,

outer: Boolean = **false**,

alias: Option[String] = None) =

**new** SchemaRDD(sqlContext, Generate(generator, join, outer, None, logicalPlan))

Generate操作

**case** **class** Generate(

generator: Generator,

join: Boolean,

outer: Boolean,

alias: Option[String],

child: LogicalPlan)

**extends** UnaryNode {

**protected** **def** generatorOutput =

alias

.map(a => generator.output.map(\_.withQualifiers(a :: Nil)))

.getOrElse(generator.output)

**def** output =

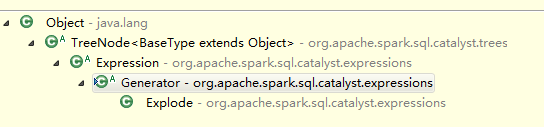
**if** (join) child.output ++ generatorOutput **else** generatorOutput

**def** references =

**if** (join) child.outputSet **else** generator.references

}

Generator类继承结构：



该类的作用是根据一个输入row，产生0个或多个row。所以上面使用的时候，generate的output是需要特殊生成的，比如Explode实现类的makeOutput()方法如下：

// TODO: Move this pattern into Generator.

**protected** **def** makeOutput() =

**if** (attributeNames.size == elementTypes.size) {

attributeNames.zip(elementTypes).map {

**case** (n, t) => AttributeReference(n, t, nullable = **true**)()

}

} **else** {

elementTypes.zipWithIndex.map {

**case** (t, i) => AttributeReference(s"c\_$i", t, nullable = **true**)()

}

}

#### insertInto

实现

\* Adds the rows from this RDD to the specified table.

**def** insertInto(tableName: String, overwrite: Boolean = **false**) =

**new** SchemaRDD(

sqlContext,

InsertIntoTable(UnresolvedRelation(None, tableName), Map.empty, logicalPlan, overwrite))

InsertIntoTable操作

**case** **class** InsertIntoTable(

table: LogicalPlan,

partition: Map[String, Option[String]],

child: LogicalPlan,

overwrite: Boolean)

**extends** LogicalPlan {

// The table being inserted into is a child for the purposes of transformations.

**def** children = table :: child :: Nil

**def** references = Set.empty

**def** output = child.output

**override** **lazy** **val** resolved = childrenResolved && child.output.zip(table.output).forall {

**case** (childAttr, tableAttr) => childAttr.dataType == tableAttr.dataType

}

}

## SQL Parser

除了之前的sql parser外，

在sql.hive package下面多了一个HiveQl

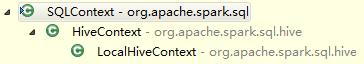
## Columner

## HIVE

### Hive Context

HiveContext是Spark SQL执行引擎之一，将hive数据结合到Spark环境中，读取的配置在hive-site.xml里指定。

继承关系



HiveContext里的sql parser使用的是HiveQl，

执行hql的时候，runHive方法接收cmd，且设置了最大返回行数

**protected** **def** runHive(cmd: String, maxRows: Int = 1000): Seq[String]

调用的方法是hive里的类，返回结果存在java的ArrayList里

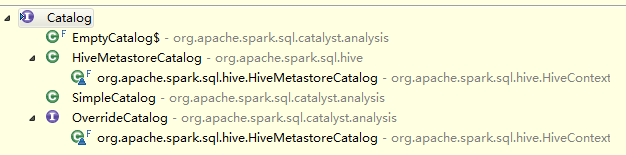
错误日志会记录在outputBuffer里，用于打印输出

逻辑执行计划的几个步骤仍然类似SqlContext，因为QueryExecution也继承了过来

**abstract** **class** QueryExecution **extends** **super**.QueryExecution {

区别在于使用的实例不一样，且toRdd操作逻辑不一样

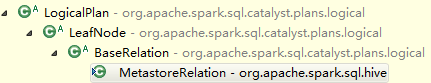
### Hive Catalog



使用HiveMetastoreCatalog存表信息

HiveMetastoreCatalog内，通过HiveContext的hiveconf，创建了hive client，所以可以进行getTable，getPartition，createTable操作

HiveMetastoreCatalog内的MetastoreRelation，继承结构如下



通过hive的接口创建了Table，Partition，TableDesc，并带一个隐式转换HiveMetastoreTypes类，因为在把Schema里的Field转成Attribute的过程中，借助HiveMetastoreTypes的toDataType把Catalyst支持的DataType parse成hive支持的类型

### Hive QL

### Hive UDF

**object** HiveFunctionRegistry

**extends** analysis.FunctionRegistry **with** HiveFunctionFactory **with** HiveInspectors {

继承FunctionRegistry，实现的是lookupFunction方法

HiveFunctionFactory主要做反射的事情，以及把hive的类型转化成为catalyst type

包括

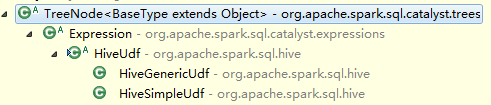
**def** getFunctionInfo(name: String) = FunctionRegistry.getFunctionInfo(name)

**def** getFunctionClass(name: String) = getFunctionInfo(name).getFunctionClass

**def** createFunction[UDFType](name: String) =

getFunctionClass(name).newInstance.asInstanceOf[UDFType]

HiveInspectors是Catalyst DataType和Hive ObjectInspector的转化



Java类到Catalyst dataType的转化

**def** javaClassToDataType(clz: Class[\_]): DataType = clz **match**

### Hive Strategy

Hive planner & strategy

**val** hivePlanner = **new** SparkPlanner **with** HiveStrategies {

**val** hiveContext = self

**override** **val** strategies: Seq[Strategy] = Seq(

TopK,

ColumnPrunings,

PartitionPrunings,

HiveTableScans,

DataSinks,

Scripts,

PartialAggregation,

SparkEquiInnerJoin,

BasicOperators,

CartesianProduct,

BroadcastNestedLoopJoin

)

}

# Summary