Spark SQL

Spark SQL是Apache Spark大数据框架的一部分，主要用于结构化数据处理和对Spark数据执行类SQL的查询。通过Spark SQL，可以针对不同格式的数据执行ETL操作（如JSON,Parquest,数据库，也可以从Hive中读取数据），然后完成特定的查询操作。如下图所说：



Spark SQL将SQL类型的查询语言整合到Spark的核心RDD概念里，这样可以应用于多种任务，包括流处理、批处理、机器学习等。

# 一.简述

下面是一段简短的Spark SQL程序：

1. *scala> val spark = SparkSession*
   1. *.builder()*
   2. *.appName("Spark SQL basic example")*
   3. *.config("spark.some.config.option", "some-value")*
   4. *.getOrCreate()*
2. *scala> val df = spark.read().json("examples/src/main/resources/people.json");*
3. *scala> df.createOrReplaceTempView("people")*
4. *scala> val sqlDF = spark.sql("SELECT \* from people")*
5. *scala> sqlDF.show()*
6. *+----+-------+*
7. *| age| name|*
8. *+----+-------+*
9. *|null|Michael|*
10. *| 30| Andy|*
11. *| 19| Justin|*
12. *+----+-------+*

程序前5行生成SparkSession，为Spark SQL执行的上下文环境。程序2,3两句加载数据源注册table，第4句是SQL入口，是sql函数，返回一个DataFrame，这一步是Lazy。调用show 的action执行时，sql才会执行。

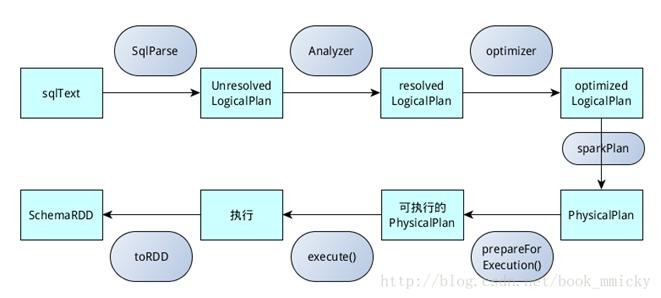
1. *val sqlContext = new org.apache.spark.sql.SQLContext(sc)*

# 二．Spark SQL的执行过程

SparkSQL有两个分支SQLContext和HiveContext

## 2.1 SQLContext

SQLContext只支持SQL语法解析器（SQL-92语法），运行过程如下图：

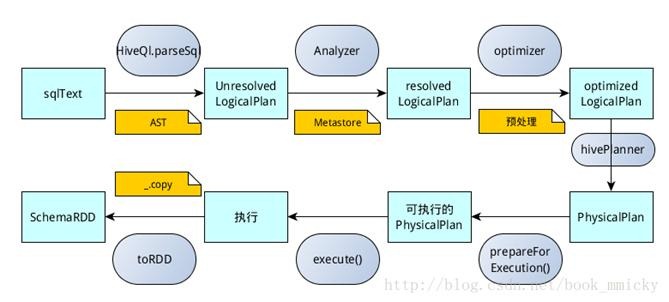


1. SQL语句经过SQLParse解析成Unresolved LogicalPlan
2. 使用analyzer结合数据字典（catalog）进行绑定，生成resolved LogicalPlan
3. 使用optimizer对resolved LogicalPlan进行优化，生成optimized LogicalPlan
4. 使用SparkPlan对LogicalPlan转换成PhysicalPlan
5. 使用prepareForExecution()将PhysicalPlan转换成可执行物理计划
6. 使用execute执行可执行物理计划
7. 生成SchemaRDD

整个过程中涉及到多个SparkSQL组件，如SqlParse,Analyzer,Optimizer及SparkPlan等

## 2.2 HiveContext

HiveContext现在支持SQL语法解析器和HiveSQL语法解析器，默认HiveSQL，用户可以通过配置切换成SQL语法解释器来运行HiveSQL不支持语法，执行过程如下：



1. SQL语句经过HiveQL的parseSQL解析成Unresolved LogicalPlan，在这个过程中对hiveSQL语句使用getAST获取AST树，然后进行解析
2. 使用analyzer结合数据Hive元数据Metastore(新的catalog)进行绑定，生成resolved

Logical Plan

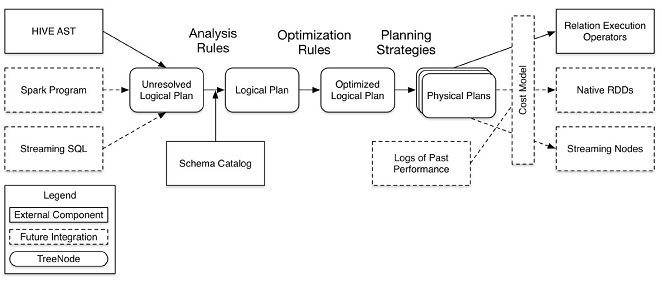
1. 使用Optimizer对resolved LogicalPlan进行优化，生成Optimized LogicalPlan，优化前使用了ExtractPythonUdfs进行预处理
2. 使用HivePlanner将LogicalPlan转换成可执行物理计划
3. 使用execute执行物理计划
4. 执行后，使用map(\_.copy)将结果导入SchemaRDD

## 2.3 SparkSQL运行架构

SparkSQL总体上由四个模块组成：core、catalyst、hive和hive-ThriftServer

1. Core处理数据的输入输出，从不同数据源获取数据（RDD,Parquet及Json等），将结果输出成SchemaRDD
2. Catalyst处理查询语句的整个处理过程，包括解析、绑定、优化和物理计划等，可以将Catalyst作为查询引擎
3. Hive模块对Hive数据处理
4. Hive-ThiftServer提供CLI和JDBC/ODBC接口

在这四个模块中，Catalyst处于最核心的部分作为Spark SQL的查询引擎，下图是其设计图：

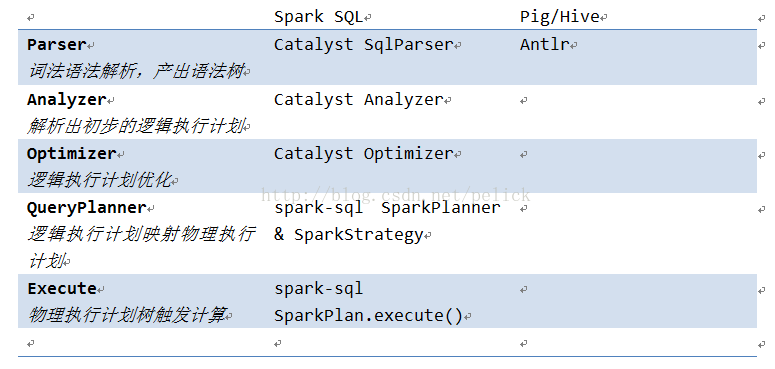


Catalyst主要实现的组件有：

1. SqlParse，完成sql语句的语法解析功能，目前只提供一个简单的sql解析器
2. Analyzer，主要完成绑定工作，将不同来源的Unresolved LogicalPlan和数据元数据（Hive MetaStore,Schema Catalog）进行绑定，生成resolved Logical Plan
3. Planner将Logincal Plan转换成Physical Plan
4. CostModel，主要根据过去的性能统计数据，选择最优的物理执行计划

# 三．SparkSQL的核心流程

在前部分已经介绍了Spark SQL的执行流程，各组件的层次结构如下表：



在Spark SQL中SQLContext是SQL执行的上下文对象，可以用于创建DataFrame对象，并执行SQL查询。在SQLContext中使用Catalog保存查询的表、逻辑计划关系等元数据，其相当于数据字典。其类图如下所示：



根据类图分析，Catalog不仅可以操作Spark SQL，还可以直接操作Hive元数据。Catalog可以通过SparkSession获取，使用如下所示：

*scala> import org.apache.spark.sql.SparkSession*

*scala> val sparkSession = SparkSession.builder.appName("Spark SQL Basic Eample").getOrCreate()*

*scala> val catalog = sparkSession.catalog*

一旦创建好了Catalog，就可以查询元数据中的数据库，Catalog返回的结果全部是DataSet

*scala> catalog.listDatabases().select("name").show(false)*

*+-------+*

*|name |*

*+-------+*

*|default|*

*+-------+*

在Spark 2.0以后，使用createTempView来注册DataFrame，使用方法如下：

*scala> val df = sparkSession.read.json("examples/src/main/resources/people.json")*

*scala> df.createOrReplaceTempView("people")*

*scala> catalog.listTables().select("name").show(false)*

*+------+*

*|name |*

*+------+*

*|people|*

People表就是使用createOrReplaceTempView注册的临时表。可以使用Catalog提供的API来查看某个表是否缓存：

*scala> println(catalog.isCached("people"))*

*false*

*scala> df.cache()*

*res9: df.type = [age: bigint, name: string]*

*scala> println(catalog.isCached("people"))*

*true*

删除注册好的view，从元数据中删除表：

scala> catalog.dropTempView("people")

*scala> catalog.listTables().select("name").show(false)*

*+----+*

*|name|*

通过Catalog，还可以用于查看它操作UDF，下面显示SparkSession已经注册好的函数，包括Spark内置的函数：

*scala> catalog.listFunctions().select("name","className","isTemporary").show(20,false)*

*+---------------------+-----------------------------------------------------------------------+-----------+*

*|name |className |isTemporary|*

*+---------------------+-----------------------------------------------------------------------+-----------+*

*|! |org.apache.spark.sql.catalyst.expressions.Not |true*

*|% |org.apache.spark.sql.catalyst.expressions.Remainder |true*

*|& |org.apache.spark.sql.catalyst.expressions.BitwiseAnd |true*

*|\* |org.apache.spark.sql.catalyst.expressions.Multiply |true*

*|+ |org.apache.spark.sql.catalyst.expressions.Add |true*

*|- |org.apache.spark.sql.catalyst.expressions.Subtract |true |*

*|/ |org.apache.spark.sql.catalyst.expressions.Divide |true |*

*|< |org.apache.spark.sql.catalyst.expressions.LessThan |true |*

*|<= |org.apache.spark.sql.catalyst.expressions.LessThanOrEqual |true |*

*|<=> |org.apache.spark.sql.catalyst.expressions.EqualNullSafe |true |*

*|= |org.apache.spark.sql.catalyst.expressions.EqualTo |true |*

*|== |org.apache.spark.sql.catalyst.expressions.EqualTo |true |*

*|> |org.apache.spark.sql.catalyst.expressions.GreaterThan |true |*

*|>= |org.apache.spark.sql.catalyst.expressions.GreaterThanOrEqual |true |*

*|^ |org.apache.spark.sql.catalyst.expressions.BitwiseXor |true |*

*|abs |org.apache.spark.sql.catalyst.expressions.Abs |true |*

*|acos |org.apache.spark.sql.catalyst.expressions.Acos |true |*

*|add\_months |org.apache.spark.sql.catalyst.expressions.AddMonths |true |*

*|and |org.apache.spark.sql.catalyst.expressions.And |true |*

*|approx\_count\_distinct|org.apache.spark.sql.catalyst.expressions.aggregate.HyperLogLogPlusPlus|true |*

*+---------------------+-----------------------------------------------------------------------+-----------+*

*only showing top 20 rows*

## 3.1 SqlParser

SqlParser根据传入的SQL对语法进行分词，构成语法树，返回一个logical Plan，主要的核心组件是SqlParser（SQL语言解析器），用Scala实现的Parser将解析的结果封装成Catalyst TreeNode，流程图如下：



SQL经过SQL Parser解析生成Unresolved Logical Plan(包含UnresolvedRelation,

UnresolvedFunction和UnresolvedAttribute)。源代码中一般使用如下：

*SparkSession:: def sql(sqlText: String) : DataFrame =*

*{DataSet.ofRows(self, sessionState.sqlParser.parsePlan(sqlText));}*

当调用sql(“select \* from people”)时，实际上是new了一个DataFrame，当new DataFrame时，

在SQLContext中定义了SQL解析方法parseSQL，该方法调用接口ParseInterface定义的parsePlan方法:

*override def parsePlan(sqlText: String): LogicalPlan = parse(sqlText) { parser =>*

*astBuilder.visitSingleStatement(parser.singleStatement()) match {*

*case plan: LogicalPlan => plan*

*case \_ =>*

*val position = Origin(None, None)*

*throw new ParseException(Option(sqlText), "Unsupported SQL statement", position, position)}}*

具体的执行是AbstractSqlParser抽象类中的parse方法进行实现，解析完毕后生成语法树，然后通过AstBuilder解析生成表达式、逻辑计划或者表标识等对象。相关类图如下：



在AbstractSqlParse的parse方法中，先实例化词法解析器SqlBaseLexer和语法解析器SqlBaseParser，然后尝试使用Antrl较快的解析模式SLL，如果解析失败，则会尝试使用普通解析模式LL，解析完毕后返回接卸结果，具体的代码如下：

*protected def parse[T](command: String)(toResult: SqlBaseParser => T): T = {*

*val lexer = new SqlBaseLexer(new ANTLRNoCaseStringStream(command))*

*lexer.removeErrorListeners()*

*lexer.addErrorListener(ParseErrorListener)*

*val tokenStream = new CommonTokenStream(lexer)*

*val parser = new SqlBaseParser(tokenStream)*

*parser.addParseListener(PostProcessor)*

*parser.removeErrorListeners()*

*parser.addErrorListener(ParseErrorListener)*

*try {*

*try {*

*// first, try parsing with potentially faster SLL mode*

*parser.getInterpreter.setPredictionMode(PredictionMode.SLL)*

*toResult(parser)*

*}*

*catch {*

*case e: ParseCancellationException =>*

*// if we fail, parse with LL mode*

*tokenStream.reset() // rewind input stream*

*parser.reset()*

*// Try Again.*

*parser.getInterpreter.setPredictionMode(PredictionMode.LL)*

*toResult(parser)*

*}*

*}*

*catch {*

*case e: ParseException if e.command.isDefined =>*

*throw e*

*case e: ParseException =>*

*throw e.withCommand(command)*

*case e: AnalysisException =>*

*val position = Origin(e.line, e.startPosition)*

*throw new ParseException(Option(command), e.message, position, position)*

*}*

*}*

*}*

SqlParser的核心是词法解析器SqlBaseLexer和SqlBaseParser，这两个类通过Antrl定义自动生成而来的，其定位文件为SqlBase.g4。使用案例如下，LogicalPlan为标红部分：

*scala> val query = sqlContext.sql("select name, age from (select \* from employee where provinceId=1) a where a.age >= 20 and a.age < 40")*

*17/10/10 17:34:34 INFO ParseDriver: Parsing command: select name, age from (select \* from employee where provinceId=1) a where a.age >= 20 and a.age < 40*

*17/10/10 17:34:34 INFO ParseDriver: Parse Completed*

*query: org.apache.spark.sql.DataFrame = [name: string, age: int]*

*scala> query.queryExecution.logical*

*res1: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =*

*'Project [unresolvedalias('name),unresolvedalias('age)]*

*+- 'Filter (('a.age >= 20) && ('a.age < 40))*

*+- 'Subquery a*

*+- 'Project [unresolvedalias(\*)]*

*+- 'Filter ('provinceId = 1)*

*+- 'UnresolvedRelation `employee`, None*

## 3.2 Analyzer

SQL语句经过Antlr和Spark SQL的Catalyst框架的SplParser，经过解析生成Unresolved Logical Plan。Analyzer位于Catalyst的analysis package下，主要职责使用Analysis Rules结合SessionCatalog元数据，将Sql Parser Unresolved 的Logical Plan进行解析，生成Resolved Logical Plan。如下图所示：



下面是相关的名词解释：

1. FixedPoint，相当于迭代次数的上限
2. Batch，批次，这个对象是由一系列的Rule组成，采用了一个策略（迭代几次的别名，例如Once）
3. Rule，理解为一种规则，这种规则会应用到Logical Plan，从而将UnResolved转变为Resolved

*abstract class Rule[TreeType <: TreeNode[\_]] extends Logging {*

*/\*\* Name for this rule, automatically inferred based on class name. \*/*

*val ruleName: String = {*

*val className = getClass.getName*

*if (className endsWith "$") className.dropRight(1) else className*

*}*

*def apply(plan: TreeType): TreeType*

*}*

1. Strategy，最大执行次数，如果执行次数在最大迭代次数之前就达到fix point，策略就会停止，不再应用

在Analyzer中定义了FixedPoint和Seq[Batch]两个变量，其中FixedPoint为迭代次数的上限，而Seq[Batch]为所定义需要执行批处理的序列，每个批处理由一系列Rule和策略所组成。策略一般分为Once和FixePoint，源码如下：

*class Analyzer(catalog: SessionCatalog,conf: SQLConf,maxIterations: Int)*

*extends RuleExecutor[LogicalPlan] with CheckAnalysis {*

*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) ::*

*TypeCoercion.typeCoercionRules ++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))*

*......}*

Analyzer解析主要是根据这些Batch定义的Rule对Unresolved Logical Plan进行解析，而Analyzer本身并没有定义执行Rule的方法，需要调用父类RuleExecutor.execute来执行这些Rule，如下图所示：



在execute方法中，执行这些Rule为串行操作，迭代这些Rule处理未绑定逻辑计划直到FxiedPoint次数或迭代前后两次的Resolved LogicalPlan相同才停止操作。生成的结果如下所示：

*scala> query.queryExecution.analyzed*

*res18: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =*

*Project [name#153, age#154]*

*+- Filter ((age#154 >= 20) && (age#154 < 40))*

*+- SubqueryAlias a*

*+- Project [name#153, age#154, province#155]*

*+- Filter (province#155 = beijing)*

*+- SubqueryAlias employee*

*+- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(assertnotnull(input[0, Employee, true])).name, true) AS name#153, assertnotnull(assertnotnull(input[0, Employee, true])).age AS age#154, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(assertnotnull(input[0, Employee, true])).province, true) AS province#155]*

*+- ExternalRDD [obj#152]*

LogicalPlan中的表，字段均替换为Catalog中表的元数据。

在Spark SQL中Rule都定义在Analyzer.scala的内部类中， 在Batches里面定义了以下几种Batch：

1. **Hints**
2. **Simple Sanity Check**
3. **Substitution**
4. **Resolution，包含的规则如下：**

* ResolveRelations

SqlParser将SQL语句parse以后，比如select \* from src，这个src表parse后就是一个UnresolvedRelation节点。ResolveRelations调用Catalog（维护tablename与LogicalPlan）的HashMap结果，从中找到当前表的结构，解析出表的字段，如UnresolveRelations会得到一个tableWithQualifiers(即表和字段)。

* ResolveReferences

将Sql parser解析出的UnresolveAttributed全部转换成对应的AttributedReferences，调用LogicPlan的resolve方法，将属性转为NamedExepression

* ResolveFunctions

和ResolveReferences差不多，主要对udf进行resolve，在FunctionRegistry中查找这些UDF

* GlobalAggregate

全局的聚合，如果遇到Project，就会返回一个Aggregate

1. **Post-Hoc Resolution**
2. **View**
3. **Nondeterministic**
4. **UDF**
5. **FixNullability**
6. **SubQuery**
7. **Cleaup**

上述的Batch是将不同的Rule进行分类，每种类别采用不同的策略进行Resolve。

## 3.3 Optimizer

Optimizer的主要职责是将Analyzer给Resolved的Logical Plan根据不同的优化策略Batch，对语法树进行优化，优化Logical Plan以及表达式Expression，得到的结果是转换成物理执行计划的前置，如下图：



其实现和处理方式和Analyzer类似，在该类中定义了一系列的Rule并同样继承于RuleExecutor，利用这些Rule对LogicalPlan和Expression进行迭代处理，从而达到树的节点进行合并和优化（类图见Analyzer）。

其中主要的优化策略总结起来就是合并，列裁剪，过滤器下推等几大类，其代码如下：

*abstract class Optimizer(sessionCatalog: SessionCatalog)*

*extends RuleExecutor[LogicalPlan] {*

*def batches: Seq[Batch] = {*

*Batch("Eliminate Distinct", Once, EliminateDistinct) ::*

*// Technically some of the rules in Finish Analysis are not optimizer rules and belong more*

*// in the analyzer, because they are needed for correctness (e.g. ComputeCurrentTime).*

*// However, because we also use the analyzer to canonicalized queries (for view definition),*

*// we do not eliminate subqueries or compute current time in the analyzer.*

*Batch("Finish Analysis", Once,EliminateSubqueryAliases,EliminateView,ReplaceExpressions,*

*ComputeCurrentTime,GetCurrentDatabase(sessionCatalog), RewriteDistinctAggregates,*

*ReplaceDeduplicateWithAggregate) ::*

*//////////////////////////////////////////////////////////////////////////////////////////*

*// Optimizer rules start here*

*//////////////////////////////////////////////////////////////////////////////////////////*

*// - Do the first call of CombineUnions before starting the major Optimizer rules,*

*// since it can reduce the number of iteration and the other rules could add/move*

*// extra operators between two adjacent Union operators.*

*// - Call CombineUnions again in Batch("Operator Optimizations"),*

*// since the other rules might make two separate Unions operators adjacent.*

*Batch("Union", Once,CombineUnions) ::*

*Batch("Pullup Correlated Expressions", Once,PullupCorrelatedPredicates) ::*

*Batch("Subquery", Once,OptimizeSubqueries) ::*

*Batch("Replace Operators", fixedPoint,ReplaceIntersectWithSemiJoin,*

*ReplaceExceptWithAntiJoin,ReplaceDistinctWithAggregate) ::*

*Batch("Aggregate", fixedPoint,RemoveLiteralFromGroupExpressions,*

*RemoveRepetitionFromGroupExpressions) ::*

*Batch("Operator Optimizations", fixedPoint, Seq(*

*// Operator push down，操作下推*

*PushProjectionThroughUnion,ReorderJoin,EliminateOuterJoin,InferFiltersFromConstraints,*

*BooleanSimplification,PushPredicateThroughJoin,PushDownPredicate,LimitPushDown,*

*ColumnPruning,*

*// Operator combine，操作合并*

*CollapseRepartition,CollapseProject,CollapseWindow,CombineFilters,CombineLimits,*

*CombineUnions,*

*// Constant folding and strength reduction，常量合并和维数降低*

*NullPropagation,*

*ConstantPropagation,FoldablePropagation,OptimizeIn,ConstantFolding,*

*ReorderAssociativeOperator,LikeSimplification,BooleanSimplification,*

*SimplifyConditionals,RemoveDispensableExpressions,SimplifyBinaryComparison,*

*PruneFilters,EliminateSorts,SimplifyCasts,SimplifyCaseConversionExpressions,*

*RewriteCorrelatedScalarSubquery,EliminateSerialization,RemoveRedundantAliases,*

*RemoveRedundantProject,SimplifyCreateStructOps,SimplifyCreateArrayOps,*

*SimplifyCreateMapOps,CombineConcats) ++extendedOperatorOptimizationRules: \_\*) ::*

*Batch("Join Reorder", Once,CostBasedJoinReorder) ::*

*Batch("Decimal Optimizations", fixedPoint,DecimalAggregates) ::*

*Batch("Object Expressions Optimization", fixedPoint,EliminateMapObjects,*

*CombineTypedFilters) ::*

*Batch("LocalRelation", fixedPoint,ConvertToLocalRelation,PropagateEmptyRelation) ::*

*Batch("Check Cartesian Products", Once,CheckCartesianProducts) ::*

*Batch("OptimizeCodegen", Once,OptimizeCodegen) ::*

*Batch("RewriteSubquery", Once,RewritePredicateSubquery,CollapseProject) :: Nil*

*}*

Optimizer的优化策略不仅对已绑定的逻辑计划进行优化，而且对逻辑计划中的Expression进行优化。其原理就是遍历树，然后应用优化Rule，对逻辑计划的处理是先序遍历（pre-order），而对Expression的处理是后序遍历。上述案例中优化的LogicalPlan如下：

*scala> query.queryExecution.optimizedPlan*

*res1: org.apache.spark.sql.catalyst.plans.logical.LogicalPlan =*

*Project [name#4, age#5]*

*+- Filter (((isnotnull(province#6) && (province#6 = beijing)) && (age#5 >= 20)) && (age#5 < 40))*

*+- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, Employee, true]).name, true) AS name#4, assertnotnull(input[0, Employee, true]).age AS age#5, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, Employee, true]).province, true) AS province#6]*

*+- ExternalRDD [obj#3]*

其中Expression相关类主要用到了references和outputSet，References主要是Logical Plan或者Expression节点所依赖的那些Expression，而OutputSet是Logical Plan所有的Attributed的输出，如Aggregate是一个Logical Plan，它的reference就是group by表达式和aggreagate的表达式的并集去重：

*case class Aggregate(*

*groupingExpressions: Seq[Expression],*

*aggregateExpressions: Seq[NamedExpression],*

*child: LogicalPlan)*

*extends UnaryNode {*

*override lazy val resolved: Boolean = {*

*val hasWindowExpressions = aggregateExpressions.exists ( \_.collect {*

*case window: WindowExpression => window*

*}.nonEmpty*

*)*

*!expressions.exists(!\_.resolved) && childrenResolved && !hasWindowExpressions*

*}*

*override def output: Seq[Attribute] = aggregateExpressions.map(\_.toAttribute)*

*override def maxRows: Option[Long] = child.maxRows*

*override def validConstraints: Set[Expression] = {*

*val nonAgg = aggregateExpressions.filter(\_.find(\_.isInstanceOf[AggregateExpression]).isEmpty)*

*child.constraints.union(getAliasedConstraints(nonAgg))*

*}*

*}*

相关类图如下：



在Spark SQL中Optimizer Rule都定义在Optimizer.scala的内部类中， 在Batches里面定义了以下几种Batch：

1. **Eliminate Distinct**
2. **Finish Analysis**
3. **Union**
4. **Pullup Correlated Expression**
5. **Subquery**
6. **Replace Operators**
7. **Aggregate**
8. **Operator Optimizations，**包含的规则如下：

* CombineLimits

如果出现2个Limit，则将这两个Limit合并成一个，这个要求Limit是另外一个Limit的child，例如：val query = sql("select \* from (select \* from temp\_shengli limit 100)a limit 10 ")

子查询Limit 100，但是外层查询是limit 10，那么在子查询中不必查询这么多

* ConstantFolding

常量合并是属于Expression优化的一种，可以直接计算的常量，例如：

 val query = sql("select 1+2+3+4 from temp\_shengli")

* CombineFilters

合并两个相邻的Filter，和上述的两个Combine Limit差不多，合并两个节点，可以减少树的深度从而减少重复执行过滤的代价，示例如下：

val query = sql("select key from (select key from temp\_shengli where key >100)a where key > 80 ")

1. Join Reorder
2. Decimal Optimizations
3. Object Expressions Optimization
4. LocalRelation
5. Check Cartesian Products
6. OptinizeCodegen
7. RewriteSubquery

## 3.4 SparkPlanner

经过上述步骤，Catalyst中的SqlParser,Analyzer和Optimizer后，生成优化后的LogicalPlan，之后Catalyst使用SparkPlanner将LogicalPlan转换为Physical Plan。在QueryExecution类代码中，代用SparkPlanner.plan方法对优化的LogicalPlan进行处理，代码如下：

*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()*

*}*

返回一个Iterator[PhysicalPlan]。流程如下所示：



类图如下所示：



QueryPlanner是SparkPlanner的基类，定义了核心的对象及方法，如Strategy,planLater和apply等。其中SparkStrategies继承自QueryPlanner中定义的GenericStratey，在SparkPlanner中通过改写父类QueryPlanner中的Strategies策略变量，在该变量中定义了转变物理计划所执行的策略，如下所示：

*def strategies: Seq[Strategy] =*

*experimentalMethods.extraStrategies ++*

*extraPlanningStrategies ++ (*

*DataSourceV2Strategy ::*

*FileSourceStrategy ::*

*DataSourceStrategy(conf) ::*

*SpecialLimits ::*

*Aggregation ::*

*JoinSelection ::*

*InMemoryScans ::*

*BasicOperators :: Nil)*

QueryPlanner，该类定义了策略的抽象类GenericStrategy，并实现了plan方法，其代码如下所示：

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

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

*def strategies: Seq[GenericStrategy[PhysicalPlan]]*

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

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

*// Collect physical plan candidates.*

*val candidates = strategies.iterator.flatMap(\_(plan))*

*// The candidates may contain placeholders marked as [[planLater]],*

*// so try to replace them by their child plans.*

*val plans = candidates.flatMap { candidate =>*

*val placeholders = collectPlaceholders(candidate)*

*if (placeholders.isEmpty) {*

*// Take the candidate as is because it does not contain placeholders.*

*Iterator(candidate)*

*} else {*

*// Plan the logical plan marked as [[planLater]] and replace the placeholders.*

*placeholders.iterator.foldLeft(Iterator(candidate)) {*

*case (candidatesWithPlaceholders, (placeholder, logicalPlan)) =>*

*// Plan the logical plan for the placeholder.*

*val childPlans = this.plan(logicalPlan)*

*candidatesWithPlaceholders.flatMap { candidateWithPlaceholders =>*

*childPlans.map { childPlan =>*

*// Replace the placeholder by the child plan*

*candidateWithPlaceholders.transformUp {*

*case p if p == placeholder => childPlan*

*}*

*}*

*}*

*}*

*}*

*}*

*val pruned = prunePlans(plans)*

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

*pruned }*

在该步骤中，通过执行query.queryExecution.sparkPlan命令可以生成可执行的PhysicalPlan，如下所示：

*scala> query.queryExecution.sparkPlan*

*res3: org.apache.spark.sql.execution.SparkPlan =*

*Project [name#4, age#5]*

*+- Filter (((isnotnull(province#6) && (province#6 = beijing)) && (age#5 >= 20)) && (age#5 < 40))*

*+- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, Employee, true]).name, true) AS name#4, assertnotnull(input[0, Employee, true]).age AS age#5, staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, Employee, true]).province, true) AS province#6]*

*+- Scan ExternalRDDScan[obj#3]*

## 3.5 SparkPlan的执行

SparkPlan是Catalyst中经过所有Strategies apply的最终的物理执行计划的抽象类，用来执行Spark Job，代码如下：

*lazy val executedPlan: SparkPlan = prepareForExecution(sparkPlan)*

prepareForExecution其实就是一个RuleExecutor[SparkPlan]，Rule就是SparkPlan：

*protected def prepareForExecution(plan: SparkPlan): SparkPlan = {*

*preparations.foldLeft(plan) { case (sp, rule) => rule.apply(sp)*

*}*

SparkPlan继承自QueryPlan[SparkPlan]，其中定义了partition,requiredChildDistribution以及Spark SQL启动执行的execute方法，继承关系如下图：



SparkPlanner通过prepareExecution，调用RuleExecutor的execute方法对前面生成的PhysicalPlan 使用Rule进行匹配，最终生成一个SparkPlan。

*/\*\* A sequence of rules that will be applied in order to the physical plan before execution. \*/*

*protected def preparations: Seq[Rule[SparkPlan]] = Seq(*

*python.ExtractPythonUDFs,*

*PlanSubqueries(sparkSession),*

*new ReorderJoinPredicates,*

*EnsureRequirements(sparkSession.sessionState.conf),*

*CollapseCodegenStages(sparkSession.sessionState.conf),*

*ReuseExchange(sparkSession.sessionState.conf),*

*ReuseSubquery(sparkSession.sessionState.conf))*

SparkPlan执行execute方法后，返回RDD，之后运行的Spark作业对该RDD进行操作。其中在QueryExecution类中，调用execute方法的代码如下：

*/\*\* Internal version of the RDD. Avoids copies and has no schema \*/*

*lazy val toRdd: RDD[InternalRow] = executedPlan.execute()*

# 四．SparkSQL 进阶

## 4.1 TreeNode Library

TreeNode Library是Catalyst的核心类库，语法树的构建都是由TreeNode组成。TreeNode本身是一个BaseType<:TreeNode[BaseType]>的类型，并且实现了Product这个trait，这样就可以存放异构的元素。

TreeNode有三种形态：BinaryNode,UnaryNode及LeafNode，在Catalyst中，这些Node都是继承自Logical Plan，也就是说每个TreeNode节点就是一个Logical Plan(包含Expression)，继承关系类图如下所示：



1. BinaryNode，二元节点，既有左右孩子的二叉节点，实现如下：

*abstract class BinaryNode extends LogicalPlan {*

*def left: LogicalPlan*

*def right: LogicalPlan*

*override final def children: Seq[LogicalPlan] = Seq(left, right)*

*}*

节点的定义简单，左孩子和右孩子都是LogicalPlan，children是一个Seq(left,right)。常用的二元节点是Join

1. UnaryNode,一元节点，即只有一个孩子的节点

*abstract class UnaryNode extends LogicalPlan {*

*def child: LogicalPlan*

*override final def children: Seq[LogicalPlan] = child :: Nil*

*}*

常用的一元节点有：Project,Subquery,Filter和Limit等

1. LeafNode，叶子节点，没有孩子节点的节点

*abstract class LeafNode extends LogicalPlan {*

*override final def children: Seq[LogicalPlan] = Nil*

*override def producedAttributes: AttributeSet = outputSet*

*/\*\* Leaf nodes that can survive analysis must define their own statistics. \*/*

*def computeStats(): Statistics = throw new UnsupportedOperationException*

*}*

提示常用的叶子节点主要包括Unresolved Relation及RDD上的操作。TreeNode中fastEquals，判断实例是否相同：

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

*this.eq(other) || this == other*

*}*

核心方法transform，接收一个PartialFunction（Batch中的Rule），将Rule迭代应用到该节点的所有子节点，最后返回这个节点的副本，如果rule没有对一个节点进行PartialFunction操作，就返回这个节点本身，例如：

*object GlobalAggregates extends Rule[LogicalPlan] {*

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

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

*Aggregate(Nil, projectList, child)*

*}*

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

*// Collect all Windowed Aggregate Expressions.*

*val windowedAggExprs = exprs.flatMap { expr =>*

*expr.collect {*

*case WindowExpression(ae: AggregateExpression, \_) => ae*

*}*

*}.toSet*

*// Find the first Aggregate Expression that is not Windowed.*

*exprs.exists(\_.collectFirst {*

*case ae: AggregateExpression if !windowedAggExprs.contains(ae) => ae*

*}.isDefined)*

*}*

*}*

transform真正调用的是transformDown，用先序遍历来对子节点进行递归的Rule应用，如果在对当前节点应用rule成功，修改后的节点afterRule，来对其children节点进行Rule的应用：

*def transformDown(rule: PartialFunction[BaseType, BaseType]): BaseType = {*

*val afterRule = CurrentOrigin.withOrigin(origin) {*

*rule.applyOrElse(this, identity[BaseType])*

*}*

*// Check if unchanged and then possibly return old copy to avoid gc churn.*

*if (this fastEquals afterRule) {*

*mapChildren(\_.transformDown(rule))*

*} else {*

*afterRule.mapChildren(\_.transformDown(rule))*

*}*

*}*

最重要的方法为afterRule.mapChildren，对children节点进行递归调用PartialFunction，调用了makeCopy来生成节点。

根据第三部分的示例介绍TreeNode实例，看下SparkSQL的整体树的transform：

*scala> val query = sql("SELECT \* FROM (SELECT \* FROM src) a join (select \* from src)b on a.key=b.key")*

第一步生成UnResolve Logical Plan如下：

*Project [\*]*

*Join Inner, Some(('a.key = 'b.key))*

*Subquery a*

*Project [\*]*

*UnresolvedRelation None, src, None*

*Subquery b*

*Project [\*]*

*UnresolvedRelation None, src, None*

转换成执行树（绿色UnaryNode，红色BinaryNode，蓝色LeafNode），如下图所示：



第二步是Analyzer用Batch Rules对Unresolved Logical Plan Tree进行rule apply，这里用ResolveSubQuery，将SubQuery消除掉，Batch(Resolution将Attribute和Relation解析)，Analyzed Logical Plan Tree，如下图：



第三步，Catalyst使用Optimizer，对SQL查询进行优化，查询树如下：

*Project [key#0,value#1,key#2,value#3]*

*Join Inner, Some((key#0 = key#2))*

*MetastoreRelation default, src, None*

*MetastoreRelation default, src, None*

生成的树如下：



最后一步是最终生成的物理执行计划，涉及到Hive的TableScan，涉及到了HashJoin操作，还涉及到了Exchange（包括Shuffle和Partition操作）：

*Project [key#0:0,value#1:1,key#2:2,value#3:3]*

*HashJoin [key#0], [key#2], BuildRight*

*Exchange (HashPartitioning [key#0:0], 150)*

*HiveTableScan [key#0,value#1], (MetastoreRelation default, src, None), None*

*Exchange (HashPartitioning [key#2:0], 150)*

*HiveTableScan [key#2,value#3], (MetastoreRelation default, src, None), None*

生成的物理执行树如图：



## 4.2 PhysicalPlan到RDD的具体实现

SparkPlanner将Optimized Logical Plan转换成Physical Plan后，需要将Physical Plan实现toRDD操作。在QueryExecution中的代码如下：

*lazy val toRdd: RDD[InternalRow] = executedPlan.execute()*

Spark Plan基本上包含4中操作类型，即BasicOperator基本类型，还包括Join,Aggregate和Sort等稍复杂的操作。

### 4.2.1 BasicOperator[BasicPhysicalOperations]

1. ProjectExec

Project大致含义是传入一系列表达式Seq[NamedExpression]，给定输入的Row，经过Convert操作，生成新的Row，代码如下：

*case class ProjectExec(projectList: Seq[NamedExpression], child: SparkPlan)*

*extends UnaryExecNode with CodegenSupport {*

*protected override def doExecute(): RDD[InternalRow] = {*

*child.execute().mapPartitionsWithIndexInternal { (index, iter) =>*

*val project = UnsafeProjection.create(projectList, child.output,*

*subexpressionEliminationEnabled)*

*project.initialize(index)*

*iter.map(project)*

*}*

*}*

*}*

1. FilterExec

Filter的具体实现是根据传入的condition进行对input row的eval计算，最后返回一个Boolean类型，返回true，则这个分区的这条数据会保存下来，否则过来掉：

*protected override def doExecute(): RDD[InternalRow] = {*

*val numOutputRows = longMetric("numOutputRows")*

*child.execute().mapPartitionsWithIndexInternal { (index, iter) =>*

*val predicate = newPredicate(condition, child.output)*

*predicate.initialize(0)*

*iter.filter { row =>*

*val r = predicate.eval(row)*

*if (r) numOutputRows += 1*

*r*

*}*

*}*

1. SampleExec

Sample取样操作其实是调用child.execute的结果后，返回一个RDD，对这个RDD调用其sample函数

*protected override def doExecute(): RDD[InternalRow] = {*

*if (withReplacement) {*

*// Disable gap sampling since the gap sampling method buffers two rows internally,*

*// requiring us to copy the row, which is more expensive than the random number generator.*

*new PartitionwiseSampledRDD[InternalRow, InternalRow](*

*child.execute(),*

*new PoissonSampler[InternalRow](upperBound - lowerBound, useGapSamplingIfPossible = false),*

*preservesPartitioning = true,*

*seed)*

*} else {*

*child.execute().randomSampleWithRange(lowerBound, upperBound, seed)*

*}*

*}*

1. UnionExec

Union操作支持多个子查询的Union，传入的child是一个Seq[SparkPlan]，execute的方式的实现是对其所有的children，select查询的结果集合RDD。通过调用SparkContext的union方法，将所有子查询的结果合并：

*case class UnionExec(children: Seq[SparkPlan]) extends SparkPlan {*

*override def output: Seq[Attribute] =*

*children.map(\_.output).transpose.map(attrs =>*

*attrs.head.withNullability(attrs.exists(\_.nullable)))*

*protected override def doExecute(): RDD[InternalRow] =*

*sparkContext.union(children.map(\_.execute()))*

*}*

1. BasicLimitExec

Limit操作在RDD原生API也有，即take

*trait BaseLimitExec extends UnaryExecNode with CodegenSupport {*

*val limit: Int*

*override def output: Seq[Attribute] = child.output*

*protected override def doExecute(): RDD[InternalRow] = child.execute().mapPartitions { iter =>*

*iter.take(limit)*

*}*

*}*

1. TakeOrdered

TakeOrdered是经过排序后的limit N，一般是用在sort by操作符后的limit，可以理解为TopN操作符，execute如下：

*protected override def doExecute(): RDD[InternalRow] = {*

*val ord = new LazilyGeneratedOrdering(sortOrder, child.output)*

*val localTopK: RDD[InternalRow] = {*

*child.execute().map(\_.copy()).mapPartitions { iter =>*

*org.apache.spark.util.collection.Utils.takeOrdered(iter, limit)(ord)*

*}*

*}*

*val shuffled = new ShuffledRowRDD(*

*ShuffleExchange.prepareShuffleDependency(*

*localTopK, child.output, SinglePartition, serializer))*

*shuffled.mapPartitions { iter =>*

*val topK = org.apache.spark.util.collection.Utils.takeOrdered(iter.map(\_.copy()), limit)(ord)*

*if (projectList != child.output) {*

*val proj = UnsafeProjection.create(projectList, child.output)*

*topK.map(r => proj(r))*

*} else {*

*topK*

*}*

*}*

1. SortExec

Sort是通过RowOrdering这个类来实现排序，child.execute对每个分区进行map,每个分区根据RowOrdering的order进行排序，生成一个新的有序集合，也是通过Spark RDD的sorted方法来实现：

*protected override def doExecute(): RDD[InternalRow] = {*

*val peakMemory = longMetric("peakMemory")*

*val spillSize = longMetric("spillSize")*

*val sortTime = longMetric("sortTime")*

*child.execute().mapPartitionsInternal { iter =>*

*val sorter = createSorter()*

*val metrics = TaskContext.get().taskMetrics()*

*// Remember spill data size of this task before execute this operator so that we can*

*// figure out how many bytes we spilled for this operator.*

*val spillSizeBefore = metrics.memoryBytesSpilled*

*val sortedIterator = sorter.sort(iter.asInstanceOf[Iterator[UnsafeRow]])*

*sortTime += sorter.getSortTimeNanos / 1000000*

*peakMemory += sorter.getPeakMemoryUsage*

*spillSize += metrics.memoryBytesSpilled - spillSizeBefore*

*metrics.incPeakExecutionMemory(sorter.getPeakMemoryUsage)*

*sortedIterator*

*}*

*}*

### 4.2.2 Join Related Operators

Join的操作主要包括BroadHashJoinExec,ShuffledHashJoinExec等均实现了HashJoin，主要类图：



具体的不再介绍。

参考文献：

http://www.myexception.cn/sql/1711425.html

http://blog.csdn.net/column/details/sparksql.html

http://blog.csdn.net/oopsoom/article/details/37658021

http://www.jianshu.com/p/0aa4b1caac2e

https://www.iteblog.com/archives/1701.html

附录：

1. SparkSqlParser的使用

*scala> import org.apache.spark.sql.internal.SQLConf*

*scala> val sqlConf = new SQLConf()*

*scala> val sparkSqlParser = new SparkSqlParser(sqlConf)*

*scala> val logicalPlan = sparkSqlParser.parsePlan("select \* from people")*

*scala> logicalPlan.prettyJson*

*res5: String =*

*[ {*

*"class" : "org.apache.spark.sql.catalyst.plans.logical.Project",*

*"num-children" : 1,*

*"projectList" : [ [ {*

*"class" : "org.apache.spark.sql.catalyst.analysis.UnresolvedStar",*

*"num-children" : 0*

*} ] ],*

*"child" : 0*

*}, {*

*"class" : "org.apache.spark.sql.catalyst.analysis.UnresolvedRelation",*

*"num-children" : 0,*

*"tableIdentifier" : {*

*"product-class" : "org.apache.spark.sql.catalyst.TableIdentifier",*

*"table" : "people" }*

*} ]*