SparkSQL核心概念及流程梳理

`[Stage-1]`

RDD & DataFrame & DataSet

1、RDD

- RDD是编译时类型安全的，编译时就能检查出类型错误
- RDD采用面向对象的编程风格，直接通过类名点的方式操作数据

2、DataFrame

- DataFrame引入了schema和off-heap
- schema : RDD每一行的数据, 结构都是一样的. 这个结构就存储在schema中. Spark通过schame就能够读懂数据, 因此在通信和IO时就只需要序列化和反序列化数据, 而结构的部分就可以省略了
- off-heap : 意味着JVM堆以外的内存, 这些内存直接受操作系统管理（而不是JVM）。Spark能够以二进制的形式序列化数据(不包括结构)到off-heap中, 当要操作数据时, 就直接操作off-heap内存. 由于Spark理解schema, 所以知道该如何操作.
- 不是编译类型安全的，不是面向对象的编程风格。

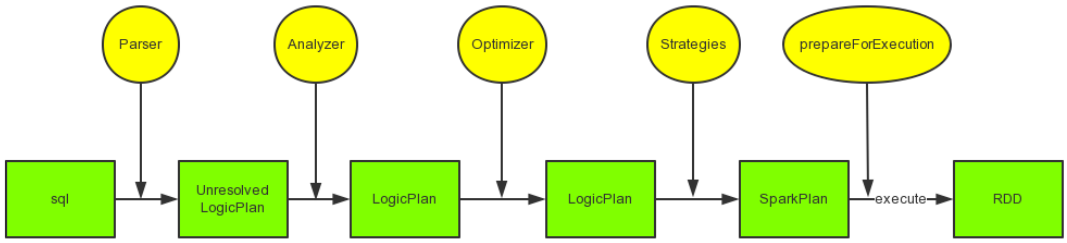
3、DataSet

- DataSet结合了RDD和DataFrame的优点, 并带来的一个新的概念Encoder
- 当序列化数据时, Encoder产生字节码与off-heap进行交互, 能够达到按需访问数据的效果, 而不用反序列化整个对象. Spark还没有提供自定义Encoder的API, 但是未来会加入
- 从1.6.X版本向2.X版本的迁移程序无需任何修改就直接用上了DataSet，因为DataFrame被声明为DataSet[Row]
- DataFrame/DataSet/SQL共享同一套优化和执行引擎
- sql.functions提供了100+种本地方法，实现更复杂的列格式
- Encoder在对象个数据源之间构建桥梁

Spark-SQL运行整体流程梳理

```
scala> val df = sql("select * from spark_test2 where a=2")
df: org.apache.spark.sql.DataFrame = [a: int, b: string]

scala> df.printSchema
root
 |-- a: integer (nullable = true)
 |-- b: string (nullable = true)


scala> Display all 644 possibilities? (y or n)

scala> df.queryExecution
res5: org.apache.spark.sql.execution.QueryExecution =
== Parsed Logical Plan ==
'Project [*]
+- 'Filter ('a = 2)
   +- 'UnresolvedRelation `spark_test2`

== Analyzed Logical Plan ==
a: int, b: string
Project [a#29, b#30]
+- Filter (a#29 = 2)
   +- MetastoreRelation bigdata, spark_test2

== Optimized Logical Plan ==
Filter (isnotnull(a#29) && (a#29 = 2))
+- MetastoreRelation bigdata, spark_test2

== Physical Plan ==
*Filter (isnotnull(a#29) && (a#29 = 2))
+- HiveTableScan [a#29, b#30], MetastoreRelation bigdata, spark_test2
```

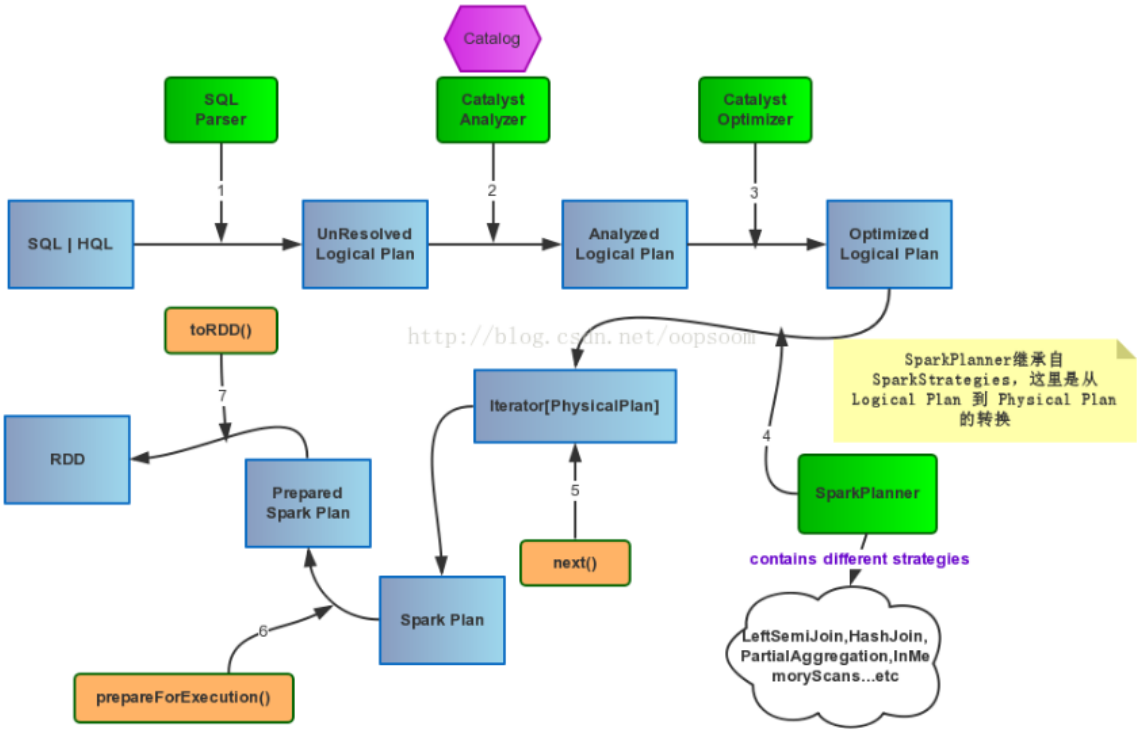
从图中可以看到，df.queryExecution输出了整体Spark SQL的运行流程。整体来看，一个SQL语句要执行需要经过下列步骤：



1) 通过 `SparkSqlParser` （继承自 `AbstractSqlParser` ）来把sql语句转换成 `Unresolved LogicPlan` ；
2) 通过Analyzer把LogicPlan当中的Unresolved的内容给解析成resolved的，这里面包括表名、函数、字段、别名等。
3) 通过Optimizer过滤掉一些垃圾的sql语句。
4) 通过Strategies把逻辑计划转换成可以具体执行的物理计划，具体的类有SparkStrategies和HiveStrategies。
5) 在执行前用prepareForExecution方法先检查一下。
6) 先序遍历，调用执行计划树的execute方法

代码流转详细流程：

Spark SQL 执行流程图



1) sql or hql >>
2) sql parser(parse)生成 unresolved logical plan >>
3) analyzer(analysis)生成analyzed logical plan >>
4) optimizer(optimize)生成optimized logical plan >>
5) spark planner(use strategies to plan)生成physical plan >>
6) 采用不同Strategies生成spark plan >>
7) spark plan(prepare) prepared spark plan >>
8) call toRDD `execute()函数调用` 执行sql生成RDD

Parse SQL with ANTLR

当执行 `spark.sql("select * from test where id=1")` 时，其实是调用 `DataSet.ofRows` new了一个DataSet出来：

```
def sql(sqlText: String): DataFrame = {
  Dataset.ofRows(self, sessionState.sqlParser.parsePlan(sqlText))
}
```

```
def ofRows(sparkSession: SparkSession, logicalPlan: LogicalPlan): DataFrame = {
  val qe = sparkSession.sessionState.executePlan(logicalPlan)
  qe.assertAnalyzed()
  new Dataset[Row](sparkSession, qe, RowEncoder(qe.analyzed.schema))
}
```

这个DataSet需要的LogicalPlan是有 `parsePlan(sqlText)` 而来，这个 `LogicalPlan` 其实是一个 `unresolved logical plan` ，还没有经过 `analyzer` 和 `optimizer` 。 `parsePlan` 如下：

```
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)
  }
```

这里调用了 `ANTLR` 库来对SQL语句解析形成 `Ast树` 。

`unresolved logical plan` 形成后，传入 `ofRows` 方法的 `executePlan` ，开始分析优化SQL语句。

```
def executePlan(plan: LogicalPlan): QueryExecution = new QueryExecution(sparkSession, plan)
```

SQL语句的分析优化的执行流程关键链条在 `QueryExecution` 里面，其中包含了 `analyzer` 、 `optimizer` 和 `LogicalPlan to physical plan` 的过程。

Spark Analyzer

`Analyzer` 会遍历整个语法树，对树上的每个节点进行数据类型绑定以及函数绑定。从Analyzer类的注释来看，它会使用Catalog和FunctionRegistry将UnresolvedAttribute和UnresolvedRelation转换为catalyst里全类型的对象。
`Analyzer` 一些概念的理解：

- FixedPoint：相当于迭代次数的上限；
- Batch: Rule组成的规则组，采取一种策略，这里策略可以简单理解为迭代几次：

```
/** A batch of rules. */
protected case class Batch(name: String, strategy: Strategy, rules: Rule[TreeType]*)
```

- 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；
- Strategy：规则执行的最大迭代次数；
- RuleExecutor：执行Rule的执行环境，它会将包含了一系列的Rule的Batch进行执行；看代码执行过程是一个循环，每个batch下的rules都对当前的plan进行作用，迭代执行，直到达到Fix Point或者最大迭代次数；

```
batches.foreach { batch =>
  val batchStartPlan = curPlan
  var iteration = 1
  var lastPlan = curPlan
  var continue = true

  // Run until fix point (or the max number of iterations as specified in the strategy.
  while (continue) {
    curPlan = batch.rules.foldLeft(curPlan) {
      case (plan, rule) =>
        val startTime = System.nanoTime()
        val result = rule(plan)
        val runTime = System.nanoTime() - startTime
        RuleExecutor.timeMap.addAndGet(rule.ruleName, runTime)

        if (!result.fastEquals(plan)) {
          logTrace(
            s"""
              |=== Applying Rule ${rule.ruleName} ===
              |${sideBySide(plan.treeString, result.treeString).mkString("\n")}
            """.stripMargin)
        }

        result
    }
```

ResolveRelations

看一个简单的rule `ResolveRelations` ，其主要用来解析表（列）基本数据类型信息。

流程是：假如sql语句为 `select * from test` 或者 `INSERT INTO test...`，`ResolveRelations` 会从Catalog中寻找表 `test` 是否存在；假如我们是直接在文件上面运行SQL，像 `"select *from parquet./path/to/query"` ， `parquet` 这个数据库和 `/path/to/query` 这个表都是不存在的，那么这条语句会作为 `UnresolvedRelation` 继续保留，后面再来解析。

Spark Optimizer

对于SQL语句的优化策略包括基于规则优化（RBO）和基于代价优化（CBO）两种，基于规则的优化策略实际上就是对语法树进行一次遍历，模式匹配能够满足特定规则的节点。

Optimizer的工作方式与Analyzer类似，因为它们都继承自RuleExecutor[LogicalPlan]，都是执行一系列的Batch操作。

`Spark Optimizer` 比较常用的规则有谓词下推（ `PredicatePushdown` ）、常量累加（ `ConstantFolding` ）和列值裁剪（ `ColumnPruning` ）。下面针对Spark2.1版本 Optimizer 实现的规则进行功能的梳理，目的是对Optimizer提供的优化点做下全局的了解：

- combinedUnions：将两个相邻的Union合并成一个；
- ReplaceIntersectWithSemiJoin：将 `Intersect` 操作符替换成 `left-semi [[Join]]` 操作符，如：

```
SELECT a1, a2 FROM Tab1 INTERSECT SELECT b1, b2 FROM Tab2
==>  SELECT DISTINCT a1, a2 FROM Tab1 LEFT SEMI JOIN Tab2 ON a1<=>b1
 AND a2<=>b2
```

- ReplaceExceptWithAntiJoin：将 `[[Except]]` 操作符替换为 `left-anti [[Join]]` 操作符，如：

```
SELECT a1, a2 FROM Tab1 EXCEPT SELECT b1, b2 FROM Tab2
==>  SELECT DISTINCT a1, a2 FROM Tab1 LEFT ANTI JOIN Tab2 ON a1<=>b1
 AND a2<=>b2
```

- ReplaceDistinctWithAggregate：将 `Distinct` 操作符替换成 `Aggregate` 操作符，如：

```
SELECT DISTINCT f1, f2 FROM t  ==>  SELECT f1, f2 FROM t GROUP BY
f1, f2
```

- RemoveLiteralFromGroupExpressions：从 `Group` 表达式中删除常量，如：

```
SELECT a, b FROM spark_test GROUP BY a, b, 1, (2 + 3 ) ==>
SELECT a, b FROM spark_test GROUP BY a,b
```

- RemoveRepetitionFromGroupExpressions：实现 `Group` 表达式去重，如：

```
SELECT a, b FROM spark_test GROUP BY  a, b, a, b ==>
SELECT A, B FROM spark_test GROUP BY a, b
```

- PushProjectionThroughUnion：将fliter/select操作推至Union的每个元素，实现先过滤再连接，如：

```
val testUnion = Union(testRelation :: testRelation2 :: testRelation3
 :: Nil)
testUnion.where('a === 1) ==>
Union(testRelation.where('a === 1) ::
    testRelation2.where('d === 1) ::
    testRelation3.where('g === 1) :: Nil)
```

- EliminateOuterJoin：外连接消除，对满足外连接消除条件的sql进行优化，包括全外连接转内连接，全外连接转右连接，全外连接转左连接，左连接转内连接，右连接转内连接。
- PushDownPredicate，谓词下推，它实现将过滤操作下推到join操作之前进行，减少参与join的数据量，如：

```
SELECT * FROM Table1 AS a JOIN Table2 AS b ON a.IDcol = b.IDcol wher
e b.ColumnA="abc"
==>
SELECT * FROM Table1 AS a JOIN (SELECT * FROM Table2 where
ColumnA="abc")AS b ON a.IDcol = b.IDcol
```

。。。

Spark Plan

优化后的逻辑计划根据不同的策略转化为物理计划Spark Plan。
执行LogicPlan到PhysicalPlan的主体SparkPlanner的集成关系是： `SparkPlanner <-- SparkStrategies <-- QueryPlanner`
SparkPlanner.plan(OptimizedLogicPlan)，会返回一个 `Iterator[PhysicalPlan]` 的迭代器。
SparkStrategies包含了一系列的Strategies，这些Strategies会应用到一个Logical Plan，生成对应的Physical Plan

Spark Plan是Catalyst里经过所有Strategies apply 的最终的物理执行计划的抽象类，它只是用来执行spark job的。在执行该计划之前，经过 `prepareForExecution` 应用一系列 `Rule` 对SparkPlan再做一次优化，主要应用的 `Rule[SparkPlan]` 有：

```
protected def preparations: Seq[Rule[SparkPlan]] = Seq(
  python.ExtractPythonUDFs,
  PlanSubqueries(sparkSession),
  EnsureRequirements(sparkSession.sessionState.conf),
  CollapseCodegenStages(sparkSession.sessionState.conf),
  ReuseExchange(sparkSession.sessionState.conf),
  ReuseSubquery(sparkSession.sessionState.conf))
```

SparkStrategies实现了逻辑计划到物理计划的转换规则，现在实现的规则有：

```
def strategies: Seq[Strategy] =
    extraStrategies ++ (
    FileSourceStrategy ::
    DataSourceStrategy ::
    DDLStrategy ::
    SpecialLimits ::
    Aggregation ::
    JoinSelection ::
    InMemoryScans ::
    BasicOperators :: Nil)
```

用户可以通过实现 `ExperimentalMethods` 这个类来实现自定义规则。

通过执行 `SparkPlan.execute` 返回 `RDD[InternalRow]`