**Spark SQL**

|  |  |
| --- | --- |
| **审核人** |  |
| **重要性** | 中 |
| **紧迫性** | 中 |
| **拟制人** | 张包峰 |
| **提交日期** |  |

**张包峰**

(版权所有,翻版必究)

**修改记录**

|  |  |  |
| --- | --- | --- |
| **更新时间** | **变更内容** | **变更理由** |
| 2014.7.14 |  | V1.0.1 |
| 2014.7.21 | 更正Spark SQL Hive支持的描述，  增加spark-hive工程描述 |  |
|  |  |  |
|  |  |  |

[**张包峰** 1](#_Toc393716497)

[1.1 Catalyst 4](#_Toc393716498)

[1.1.1 TreeNode体系 5](#_Toc393716499)

[1.1.2 执行计划表示模型 6](#_Toc393716500)

[1.1.3 Expression体系 9](#_Toc393716501)

[1.1.4 Rules体系 9](#_Toc393716502)

[1.2 Hive相关 10](#_Toc393716503)

[1.2.1 Hive支持方式 10](#_Toc393716504)

[1.2.2 Hive on Spark 11](#_Toc393716505)

[1.3 spark-hive工程 12](#_Toc393716506)

[1.3.1 解析过程 12](#_Toc393716507)

[1.3.2 Analyze过程 13](#_Toc393716508)

[1.3.3 Optimizer过程 14](#_Toc393716509)

[1.3.4 Planner及执行过程 14](#_Toc393716510)

[1.4 SQL Core 15](#_Toc393716511)

[1.4.1 生成SchemaRDD 15](#_Toc393716512)

[1.4.2 层次结构 16](#_Toc393716513)

[1.4.3 查询流程 17](#_Toc393716514)

[1.4.4 其他特性 19](#_Toc393716515)

[1.5 存在问题 22](#_Toc393716516)

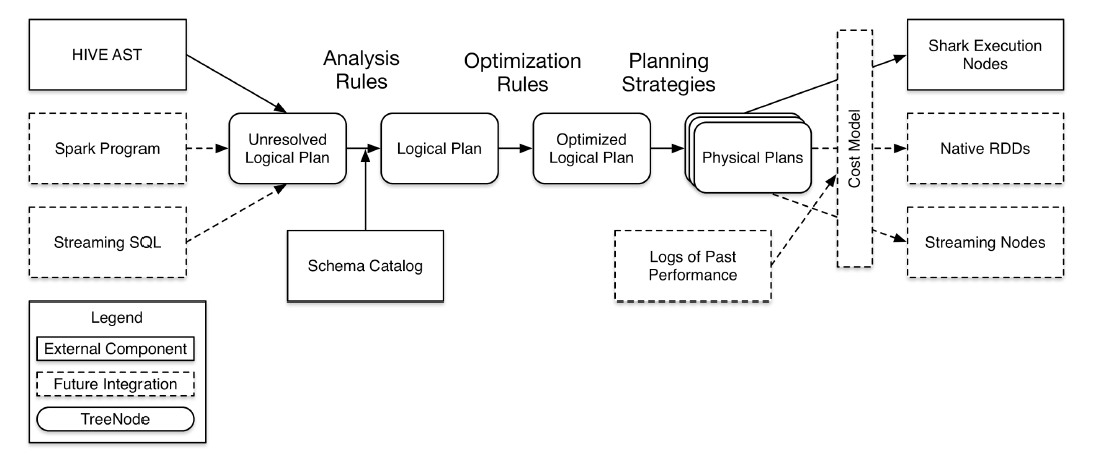
[1.6 例子 22](#_Toc393716517)

## Catalyst

Catalyst是与Spark解耦的一个独立库，是一个impl-free的执行计划的生成和优化框架。

目前与Spark Core还是耦合的，对此user邮件组里有人对此提出疑问，见[mail](http://apache-spark-user-list.1001560.n3.nabble.com/Catalyst-dependency-on-Spark-Core-td9581.html)。

以下是Catalyst较早时候的架构图，展示的是代码结构和处理流程。



*Catalyst架构图*

**Catalyst定位**

其他系统如果想基于Spark做一些类sql、标准sql甚至其他查询语言的查询，需要基于Catalyst提供的解析器、执行计划树结构、逻辑执行计划的处理规则体系等类体系来实现执行计划的解析、生成、优化、映射工作。

对应上图中，主要是左侧的TreeNode lib及中间三次转化过程中涉及到的类结构都是Catalyst提供的。至于右侧物理执行计划映射生成过程，物理执行计划基于成本的优化模型，具体物理算子的执行都由系统自己实现。

**Catalyst现状**

在解析器方面提供的是一个简单的scala写的sql parser，支持语义有限，而且应该是标准sql的。

在规则方面，提供的优化规则是比较基础的(和Pig/Hive比没有那么丰富)，不过一些优化规则其实是要涉及到具体物理算子的，所以部分规则需要在系统方那自己制定和实现(如spark-sql里的SparkStrategy)。

Catalyst也有自己的一套数据类型。

下面介绍Catalyst里几套重要的类结构。

### TreeNode体系

TreeNode是Catalyst执行计划表示的数据结构，是一个树结构，具备一些scala collection的操作能力和树遍历能力。这棵树一直在内存里维护，不会dump到磁盘以某种格式的文件存在，且无论在映射逻辑执行计划阶段还是优化逻辑执行计划阶段，树的修改是以替换已有节点的方式进行的。

TreeNode，内部带一个children: Seq[BaseType]表示孩子节点，具备foreach、map、collect等针对节点操作的方法，以及transformDown(默认，前序遍历)、transformUp这样的遍历树上节点，对匹配节点实施变化的方法。

提供UnaryNode, BinaryNode, LeafNode三种trait，即非叶子节点允许有一个或两个子节点。

TreeNode提供的是范型。

TreeNode有两个子类继承体系，QueryPlan和Expression。QueryPlan下面是逻辑和物理执行计划两个体系，前者在Catalyst里有详细实现，后者需要在系统自己实现。Expression是表达式体系，后面章节都会展开介绍。

 *TreeNode及其子类类图*

Tree的transformation实现：

传入PartialFunction[TreeType, TreeType]，如果与操作符匹配，则节点会被结果替换掉，否则节点不会变动。整个过程是对children递归执行的。

### 执行计划表示模型

#### 逻辑执行计划

QueryPlan继承自TreeNode，内部带一个output: Seq[Attribute],具备transformExpressionDown、transformExpressionUp方法。

在Catalyst中，QueryPlan的主要子类体系是LogicalPlan，即**逻辑执行计划**表示。其物理执行计划表示由使用方实现(spark-sql项目中)。

LogicalPlan继承自QueryPlan，内部带一个reference: Set[Attribute]，主要方法为resolve(name: String): Option[NamedeExpression]，用于分析生成对应的NamedExpression。

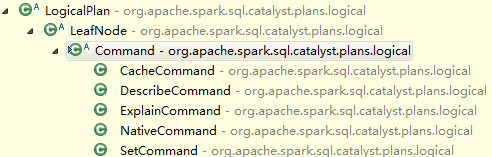
LogicalPlan有许多具体子类，也分为UnaryNode, BinaryNode, LeafNode三类，具体在org.apache.spark.sql.catalyst.plans.logical路径下。



*逻辑执行计划继承体系*

**逻辑执行计划实现**

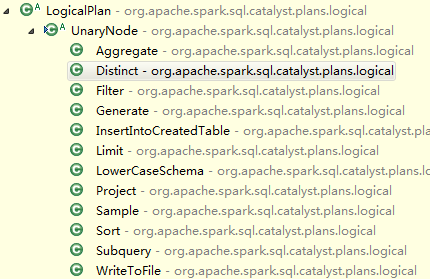
LeafNode主要子类是Command体系：



*Command子类*

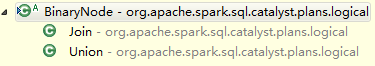
各command的语义可以从子类名字看出，代表的是系统可以执行的non-query命令，如DDL。

UnaryNode的子类：



*UnaryNode子类*

BinaryNode的子类：



*BinaryNode子类*

#### 物理执行计划

另一方面，**物理执行计划**节点在具体系统里实现，比如spark-sql工程里的SparkPlan继承体系。

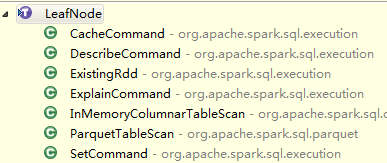


*物理执行计划继承体系*

**物理执行计划实现**

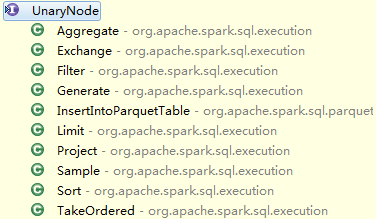
每个子类都要实现execute()方法，大致有以下实现子类(不全)。

LeadNode的子类：



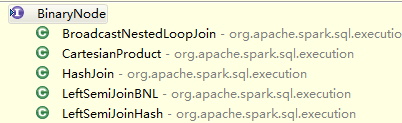
*LeafNode子类*

UnaryNode的子类：



*UnaryNode子类*

BinaryNode的子类：



*BinaryNode子类*

提到物理执行计划，还要提一下Catalyst提供的**分区表示模型**。

待说明。

#### 执行计划映射

Catalyst还提供了一个QueryPlanner[Physical <: TreeNode[PhysicalPlan]]抽象类，需要子类制定一批strategies: Seq[Strategy]，其apply方法也是类似根据制定的具体策略来把逻辑执行计划算子映射成物理执行计划算子。由于物理执行计划的节点是在具体系统里实现的，所以QueryPlanner及里面的strategies也需要在具体系统里实现。



*QueryPlanner及其子类*

在spark-sql项目中，SparkStrategies继承了QueryPlanner[SparkPlan]，内部制定了LeftSemiJoin, HashJoin, PartialAggregation, BroadcastNestedLoopJoin, CartesianProduct等几种策略，每种策略接受的都是一个LogicalPlan，生成的是Seq[SparkPlan]，每个SparkPlan理解为具体RDD的算子操作。

比如在BasicOperators这个Strategy里，以match-case匹配的方式处理了很多基本算子(可以一对一直接映射成RDD算子)，如下：

**case** logical.Project(projectList, child) =>

execution.Project(projectList, planLater(child)) :: Nil

**case** logical.Filter(condition, child) =>

execution.Filter(condition, planLater(child)) :: Nil

**case** logical.Aggregate(group, agg, child) =>

execution.Aggregate(partial = **false**, group, agg, planLater(child))(sqlContext) :: Nil

**case** logical.Sample(fraction, withReplacement, seed, child) =>

execution.Sample(fraction, withReplacement, seed, planLater(child)) :: Nil

### Expression体系

Expression，即表达式，指不需要执行引擎计算，而可以直接计算或处理的节点，包括Cast操作，Projection操作，四则运算，逻辑操作符运算等。

### Rules体系

凡是需要处理执行计划树(Analyze过程，Optimize过程，SparkStrategy过程)，实施规则匹配和节点处理的，都需要继承RuleExecutor[TreeType]抽象类。

RuleExecutor内部提供了一个Seq[Batch]，里面定义的是该RuleExecutor的处理步骤。每个Batch代表着一套规则，配备一个策略，该策略说明了迭代次数(一次还是多次)。

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

Rule[TreeType <: TreeNode[\_]]是一个抽象类，子类需要复写apply(plan: TreeType)方法来制定处理逻辑。

RuleExecutor的apply(plan: TreeType): TreeType方法会按照batches顺序和batch内的Rules顺序，对传入的plan里的节点迭代处理，处理逻辑为由具体Rule子类实现。

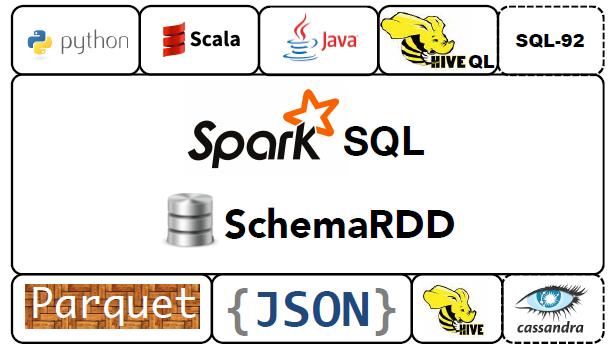
## Hive相关

### Hive支持方式

Spark SQL对hive的支持是单独的spark-hive项目，对Hive的支持包括HQL查询、hive metaStore信息、hive SerDes、hive UDFs/UDAFs/ UDTFs，类似Shark。

只有在HiveContext下通过hive api获得的数据集，才可以使用hql进行查询，其hql的解析依赖的是org.apache.hadoop.hive.ql.parse.ParseDriver类的parse方法，生成Hive AST。

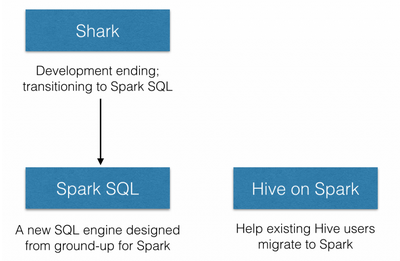
实际上sql和hql，并不是一起支持的。可以理解为hql是独立支持的，能被hql查询的数据集读取自hive。下图中的parquet、json等其他文件支持只发生在sql环境下(SQLContext)。



*Spark SQL数据支持情况*

### Hive on Spark

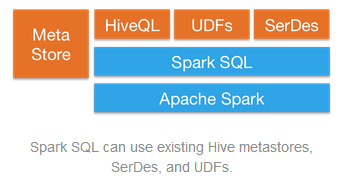
Hive官方提出了[Hive on Spark的JIRA](https://issues.apache.org/jira/browse/HIVE-7292)。Shark结束之后，拆分为两个方向：



*After Shark aborted*

**[更正]**

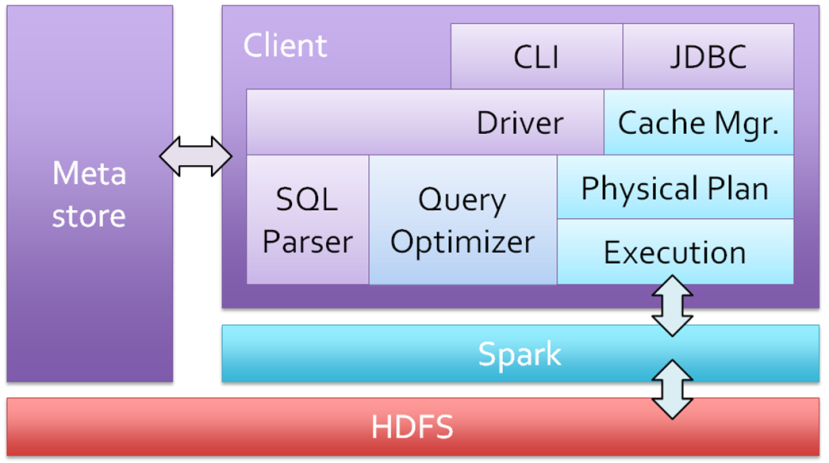
Spark SQL里现在对Hive的支持，体现在复用了Hive的meta store数据、hql解析、UDFs、SerDes，在执行DDL和某些简单命令的时候，调的是hive客户端。hql翻译前会处理一些与query主体无关的set, cache, add file等命令，然后调用ParserDriver翻译hql，并把AST转换成Catalyst的LogicalPlan，后续优化、物理执行计划翻译及执行过程，与Sql一样使用的是Catalyst提供的内容，执行引擎是Spark。在整个结合过程中，ASTNode映射成LogicalPlan是重点。



*Hive in Spark SQL*

而Hive社区的Hive on Spark会怎样实现，具体参考jira里的[设计文档](https://issues.apache.org/jira/secure/attachment/12652517/Hive-on-Spark.pdf)。

与Shark对比，



*Shark Modules*

Shark多依赖了Hive的执行计划相关模块以及CLI。CLI和JDBC部分是Spark SQL后续打算支持的。Shark额外提供的对Table数据行转列、序列化、压缩存内存的模块，也被拿到了Spark Sql的sql工程里。

以上说明了Shark与Spark SQL Hive的区别，对Shark这个项目继承性的理解。

而Spark SQL Hive与Hive社区 Hive on Spark的区别需要具体参考jira里的设计文档，我也还没有读过。

## spark-hive工程

|  |  |  |
| --- | --- | --- |
|  | Spark SQL | Hive Support |
| Parser  *词法语法解析，产出语法树* | Catalyst SqlParser | HiveQl.parseSql |
| Analyzer  *解析出初步的逻辑执行计划* | Catalyst Analyzer | Hive MetaStore &  Hive UDFs |
| Optimizer  *逻辑执行计划优化* | Catalyst Optimizer | Catalyst Optimizer |
| QueryPlanner  *逻辑执行计划映射物理执行计划* | spark-sql SparkPlanner & SparkStrategy | HivePlanner |
| Execute  *物理执行计划树触发计算* | spark-sql SparkPlan.execute() | SparkPlan.execute() |
|  |  |  |

### 解析过程

HiveQl.parseSql()把hql解析成logicalPlan。解析过程，提取出以下command进行非Catalyst执行计划的处理，包括：

* set key=value
* cache table
* uncache table
* add jar
* add file
* dfs
* source

然后由Hive的ParseDriver把hql解析成AST，得到ASTNode，

**def** getAst(sql: String): ASTNode = ParseUtils.findRootNonNullToken((**new** ParseDriver).parse(sql))

把Node转化为Catalyst的LogicalPlan，转化逻辑较复杂，也是Spark sql对hql支持的最关键部分。详见HiveQl.nodeToPlan(node: Node): LogicalPlan方法。

*大致转换逻辑包括：*

*处理TOK\_EXPLAIN和TOK\_DESCTABLE*

*处理TOK\_CREATETABLE，包括创建表时候一系列表的设置TOK\_XXX*

*处理TOK\_QUERY，包括TOK\_SELECT，TOK\_WHERE，TOK\_GROUPBY，TOK\_HAVING，TOK\_SORTEDBY，TOK\_LIMIT等等，对FROM后面跟的语句进行nodeToRelation处理。*

*处理TOK\_UNION*

对Hive AST树结构和表示不熟悉，所以此处略过。

### Analyze过程

#### metadata交互

Catalog类为HiveMetastoreCatalog，通过hive的conf生成client(org.apache.hadoop.hive.ql.metadata.Hive，用于与MetaStore通信，获得metadata以及进行DDL操作)，catalog的lookupRelation方法里面，client.getTable()得到表信息，client.getAddPartitionsForPruner()得到分区信息。

#### udf相关

FunctionRegistry类为HiveFunctionRegistry，根据方法名，通过hive的相关类去查询该方法，检查是否具有该方法，是UDF，还是UDAF(aggregation)，或是UDTF(table)。这里只做已有udf的查询，不做新方法的include。

与Catalyst的Expression对应继承关系如下：



*UDF与Catalyst Expression体系*

#### Inspector相关

HiveInspectors提供了几个映射数据类型和ObjectInspetor子类的方法，包括PrimitiveObjectInspector,ListObjectInspector,MapObjectInspector,StructObjectInspector四种

### Optimizer过程

在做优化前，会尝试对之前生成的逻辑执行计划进行create tabl操作，因为执行的hql可能是“CREATE TABLE XXX”，这部分处理在HiveMetastoreCatalog的CreateTables单例里，继承了Rule[LogicalPlan]。

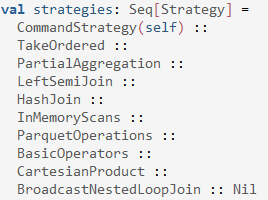
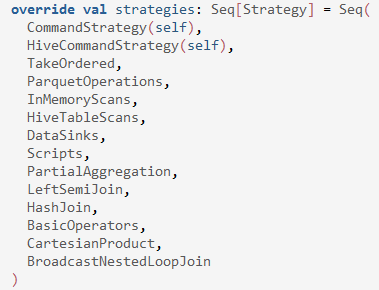
以及PreInsertionCasts处理，也是HiveMetastoreCatalog里的单例，继承了Rule[LogicalPlan]。

之后的optimizer过程同SQLContext里，用的是同一个Catalyst提供的Optimizer类。

### Planner及执行过程

HiveContext继承自SQLContext，其QueryExecution也继承自SQLContext的QueryExecution。后续执行计划优化、物理执行计划翻译、处理及执行过程同SQL的处理逻辑是一致的。

翻译物理执行计划的时候，hive planner里制定了些特定的策略，与SparkPlanner稍有不同。



*HivePlanner与SparkPlanner对比*

多了Scripts，DataSinks，HiveTableScans和HiveCommandStrategy四种处理物理执行计划的策略(见HiveStrategies)。

1. Scripts，用于处理那种hive命令行执行脚本的情况。实现方式是使用ProcessBuilder新起一个JVM进程的方式，用”/bin/bash –c scripts”的方式执行脚本并获取输出流数据，转化为Catalyst Row数据格式。
2. DataSinks，用于把数据写入到Hive表的情况。里面涉及到一些hive读写的数据格式转化、序列化、读配置等工作，最后通过SparkContext的runJob接口，提交作业。
3. HiveTableScans，用于对hive table进行扫描，支持使用谓词的分区裁剪(Partition pruning predicates are detected and applied)。
4. HiveCommandStrategy，用于执行native command和describe command。我理解是这种命令是直接调hive客户端单机执行的，因为可能只与meta data打交道。

toRDD: RDD[Row]处理也有少许区别，返回RDD[Row]的时候，对每个元素做了一次拷贝。

### CLI

spark的hive-thriftserver项目

对比Hive CLI和Spark SQL CLI的实现

#### Hive CLI

org.apache.hive.service.server.HiveServer2类，启动方式：

**try** {

ServerOptionsProcessor oproc = **new** **ServerOptionsProcessor**("hiveserver2");

**if** (!oproc.**process**(args)) {

***LOG***.fatal("Error starting HiveServer2 with given arguments");

System.*exit*(-1);

}

HiveConf hiveConf = **new** **HiveConf**();

HiveServer2 server = **new** **HiveServer2**();

server.**init**(hiveConf);

server.**start**();

} **catch** (Throwable t) {

***LOG***.fatal("Error starting HiveServer2", t);

System.*exit*(-1);

}

HiveServer2继承CompositeService类，CompositeService类内部维护一个serviceList，能够加入、删除、启动、停止不同的服务。HiveServer2在init(hiveConf)的时候，会加入CLIService和ThriftCLIService两个Service。根据传输模式，如果是http或https的话，就使用ThriftHttpCLIService，否则使用ThriftBinaryCLIService。无论是哪个ThriftCLIService，都传入了CLIService的引用，thrift只是一个封装。

加入了这些服务后，把服务都启动起来。

CLIService也继承自CompositeService，CLIService 在init的时候会加入SessionManager服务，并且根据hiveConf，从 hadoop shims里得到UGI里的serverUsername。

SessionManager管理hive连接的开启、关闭等管理功能，已有的连接会维护在一个HashMap里，value为HiveSession类，里面大致是用户名、密码、hive配置等info。

所以CLIService里几乎所有的事情都是委托给SessionManager做的。

SessionManager内主要是OperationManager这个服务，是最重要的和执行逻辑有关的类，下面会具体说。

另外，关于ThriftCLIService，有两个实现子类，子类只复写了run()方法，设置thrift server相关的网络连接，其他对CLIService的调用逻辑都在父类ThriftCLIService本身里面。



实际上，ThriftCLIService里很多事情也是委托给CLIService做的。

那么上面大致是Hive CLI、Thrift server启动的流程，以及几个主要类的相互关系。

#### Spark SQL CLI

根据上面Hive CLI的逻辑，看看Spark SQL的CLI是怎么做的。

Spark里的HiveThriftServer2(这个类名看起来有点奇怪)继承了Hive的HiveServer2，并且复写了init方法，其初始化的时候加入的是SparkSQLCLIService和ThriftBinaryCLIService两个服务。前者继承了Hive的CLIService，有一些不同的逻辑；后者直接使用的是Hive的类，但传入的是SparkSQLCLIService的引用。

SparkSQLEnv单例维护了HiveContext和SparkContext两个spark sql/hive的上下文类，是让master使用的。

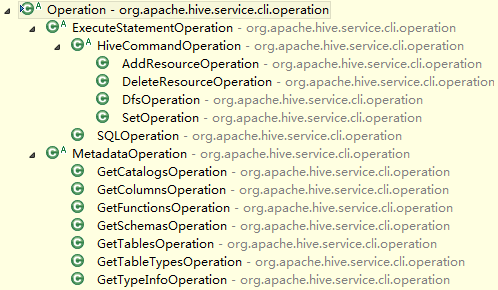
SparkSQLCLIService内部，类似Hive的CLIService，有一个SparkSQLSessionManager，继承自Hive的SessionManager。也有得到serverUsername的逻辑，代码和CLIService是一样的。

SparkSQLSessionManager复写了init这个方法，里面有Spark自己的SparkSQLOperationManager服务，继承自Hive的OperationManager类。

可能上面这几个类有点看晕了，本质上都是一些封装而已，没什么大的区别。真正重要的是SparkSQLOperationManager这个类里面，定义了如何使用Spark SQL来处理query操作。

**SparkSQLOperationManager关键逻辑**

Hive的CLI Operation父类有如下的子类继承体系，代表hive cli会处理的不同操作类型：



上半部分ExecuteStatementOperation子类体系是实际和查询相关的操作，下半部分是一些元数据读取操作。SparkSQLOperationManager实际改写的就是ExecuteStatementOperation子类的执行逻辑，而元数据相关的操作还是沿用hive本来的处理逻辑。

原本hive的ExecuteStatementOperation处理逻辑是这样的：

**public** **static** ExecuteStatementOperation newExecuteStatementOperation(

HiveSession parentSession, String statement, Map<String, String> confOverlay, **boolean** runAsync) {

String[] tokens = statement.**trim**().**split**("\\s+");

String command = tokens[0].**toLowerCase**();

**if** ("set".**equals**(command)) {

**return** **new** **SetOperation**(parentSession, statement, confOverlay);

} **else** **if** ("dfs".**equals**(command)) {

**return** **new** **DfsOperation**(parentSession, statement, confOverlay);

} **else** **if** ("add".**equals**(command)) {

**return** **new** **AddResourceOperation**(parentSession, statement, confOverlay);

} **else** **if** ("delete".**equals**(command)) {

**return** **new** **DeleteResourceOperation**(parentSession, statement, confOverlay);

} **else** {

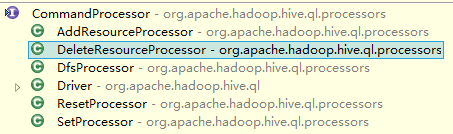
**return** **new** **SQLOperation**(parentSession, statement, confOverlay, runAsync);

}

}

ExecuteStatementOperation也分两部分，HiveCommandOperation和SQLOperation。

不同的ExecuteStatementOperation子类最终由对应的CommandProcessor子类来完成操作请求。



那Spark是如何改写ExecuteStatementOperation的执行逻辑的呢？

最核心的逻辑如下：

**def** run(): Unit = {

logInfo(s"Running query '$statement'")

setState(OperationState.RUNNING)

**try** {

result = hiveContext.sql(statement)

logDebug(result.queryExecution.toString())

**val** groupId = round(random \* 1000000).toString

hiveContext.sparkContext.setJobGroup(groupId, statement)

iter = result.queryExecution.toRdd.toLocalIterator

dataTypes = result.queryExecution.analyzed.output.map(\_.dataType).toArray

setHasResultSet(**true**)

} **catch** {

// Actually do need to catch Throwable as some failures don't inherit from Exception and

// HiveServer will silently swallow them.

**case** e: Throwable =>

logError("Error executing query:",e)

**throw** **new** HiveSQLException(e.toString)

}

setState(OperationState.FINISHED)

}

statement是一个String，即query本身，调用HiveContext的sql()方法，返回的是一个SchemaRDD。HiveContext的这段逻辑如下：

**override** **def** sql(sqlText: String): SchemaRDD = {

// TODO: Create a framework for registering parsers instead of just hardcoding if statements.

**if** (dialect == "sql") {

**super**.sql(sqlText)

} **else** **if** (dialect == "hiveql") {

**new** SchemaRDD(**this**, HiveQl.parseSql(sqlText))

} **else** {

sys.error(s"Unsupported SQL dialect: $dialect. Try 'sql' or 'hiveql'")

}

}

调完sql()后返回的是一个带被解析过了的基础逻辑计划的SchemaRDD。后续，

logDebug(result.queryExecution.toString())

这一步触发了逻辑执行计划的进一步分析、优化和变成物理执行计划的几个过程。之后，

result.queryExecution.toRdd

toRdd这步是触发计算并返回结果。这几个逻辑在之前Spark SQL源码分析的文章里都提到过。

除了上面这部分，还有一些schema转化、数据类型转化的逻辑，是因为Catalyst这边，有自己的数据行表示方法，也有自己的dataType，而且schema这块呢，在生成SchemaRDD的时候也转化过一次。所以在返回执行结果的时候，需要有转换回Hive的TableSchema、FieldSchema的逻辑。

以上说明了Spark SQL是如何把query的执行转换到Spark SQL里的。

#### 总结

基本上Spark SQL在CLI这块的实现很靠近Hive Service项目里的CLI模块，主要类继承体系、执行逻辑差不多都一样。Spark SQL修改的关键逻辑在CLIService内的SessionManager内的OperationManager里，将非元数据查询操作的query丢给了Spark SQL的Hive工程里的HiveContext.sql()来完成，通过返回的SchemaRDD，来进一步得到结果数据、得到中间执行计划的Schema信息。

## SQL Core

Spark SQL的核心是把已有的RDD，带上Schema信息，然后注册成类似sql里的”Table”，对其进行sql查询。这里面主要分两部分，一是生成SchemaRDD，二是执行查询。

### 生成SchemaRDD

如果是spark-hive项目，那么读取metadata信息作为Schema、读取hdfs上数据的过程交给Hive完成，然后根据这俩部分生成SchemaRDD，在HiveContext下进行hql()查询。

对于Spark SQL来说，

数据方面，RDD可以来自任何已有的RDD，也可以来自支持的第三方格式，如json file、parquet file。

SQLContext下会把带case class的RDD隐式转化为SchemaRDD

**implicit** **def** createSchemaRDD[A <: Product: TypeTag](rdd: RDD[A]) =

**new** SchemaRDD(**this**,

SparkLogicalPlan(ExistingRdd.fromProductRdd(rdd)))

ExsitingRdd单例里会反射出case class的attributes，并把RDD的数据转化成Catalyst的GenericRow，最后返回RDD[Row]，即一个SchemaRDD。这里的具体转化逻辑可以参考ExsitingRdd的productToRowRdd和convertToCatalyst方法。

之后可以进行SchemaRDD提供的注册table操作、针对Schema复写的部分RDD转化操作、DSL操作、saveAs操作等等。

**Row和GenericRow是Catalyst里的行表示模型**

Row用Seq[Any]来表示values，GenericRow是Row的子类，用数组表示values。Row支持数据类型包括Int, Long, Double, Float, Boolean, Short, Byte, String。支持按序数(ordinal)读取某一个列的值。读取前需要做isNullAt(i: Int)的判断。

各自都有Mutable类，提供setXXX(i: int, value: Any)修改某序数上的值。

### 层次结构

|  |  |  |
| --- | --- | --- |
|  | Spark SQL | Pig/Hive |
| Parser  *词法语法解析，产出语法树* | Catalyst SqlParser | Antlr |
| Analyzer  *解析出初步的逻辑执行计划* | Catalyst Analyzer |  |
| Optimizer  *逻辑执行计划优化* | Catalyst Optimizer |  |
| QueryPlanner  *逻辑执行计划映射物理执行计划* | spark-sql SparkPlanner & SparkStrategy |  |
| Execute  *物理执行计划树触发计算* | spark-sql SparkPlan.execute() |  |
|  |  |  |

下图大致对比了Pig，Spark SQL，Shark在实现层次上的区别，仅做参考。

*层次对比图(1)*

*层次对比图(2)*

### 查询流程

SQLContext里对sql的一个解析和执行流程：

1. 第一步parseSql(sql: String)，simple sql parser做词法语法解析，生成LogicalPlan。
2. 第二步analyzer(logicalPlan)，把做完词法语法解析的执行计划进行初步分析和映射，

目前SQLContext内的Analyzer由Catalyst提供，定义如下：

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

catalog为SimpleCatalog，catalog是用来注册table和查询relation的。

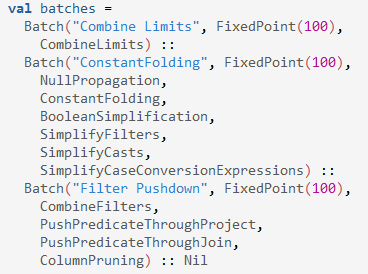
而这里的FunctionRegistry不支持lookupFunction方法，所以该analyzer**不支持Function注册**，即UDF。

Analyzer内定义了几批规则：

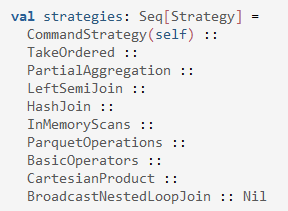


1. 从第二步得到的是初步的logicalPlan，接下来第三步是optimizer(plan)。

Optimizer里面也是定义了几批规则，会按序对执行计划进行优化操作。



1. 优化后的执行计划，还要丢给SparkPlanner处理，里面定义了一些策略，目的是根据逻辑执行计划树生成最后可以执行的物理执行计划树，即得到SparkPlan。



1. 在最终真正执行物理执行计划前，最后还要进行两次规则，SQLContext里定义这个过程叫prepareForExecution，这个步骤是额外增加的，直接new RuleExecutor[SparkPlan]进行的。



1. 最后调用SparkPlan的execute()执行计算。这个execute()在每种SparkPlan的实现里定义，一般都会递归调用children的execute()方法，所以会触发整棵Tree的计算。

### 其他特性

#### 内存列存储

SQLContext下cache/uncache table的时候会调用列存储模块。

该模块借鉴自Shark，目的是当把表数据cache在内存的时候做行转列操作，以便压缩。

**实现类**

InMemoryColumnarTableScan类是SparkPlan LeafNode的实现，即是一个物理执行计划。传入一个SparkPlan(确认了的物理执行计)和一个属性序列，内部包含一个行转列、触发计算并cache的过程(且是lazy的)。

ColumnBuilder针对不同的数据类型(boolean, byte, double, float, int, long, short, string)由不同的子类把数据写到ByteBuffer里，即包装Row的每个field，生成Columns。与其对应的ColumnAccessor是访问column，将其转回Row。

CompressibleColumnBuilder和CompressibleColumnAccessor是带压缩的行列转换builder，其ByteBuffer内部存储结构如下

*\* .--------------------------- Column type ID (4 bytes)*

*\* | .----------------------- Null count N (4 bytes)*

*\* | | .------------------- Null positions (4 x N bytes, empty if null count is zero)*

*\* | | | .------------- Compression scheme ID (4 bytes)*

*\* | | | | .--------- Compressed non-null elements*

*\* V V V V V*

*\* +---+---+-----+---+---------+*

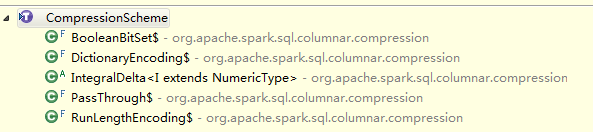
*\* | | | ... | | ... ... |*

*\* +---+---+-----+---+---------+*

*\* \-----------/ \-----------/*

*\* header body*

CompressionScheme子类是不同的压缩实现



*CompressionSchema子类*

都是scala实现的，未借助第三方库。不同的实现，指定了支持的column data类型。在build()的时候，会比较每种压缩，选择压缩率最小的（若仍大于0.8就不压缩了）。

这里的估算逻辑，来自子类实现的gatherCompressibilityStats方法。

**Cache逻辑**

cache之前，需要先把本次cache的table的物理执行计划生成出来。

在cache这个过程里，InMemoryColumnarTableScan并没有触发执行，但是生成了以InMemoryColumnarTableScan为物理执行计划的SparkLogicalPlan，并存成table的plan。

其实在cache的时候，首先去catalog里寻找这个table的信息和table的执行计划，然后会进行执行（执行到物理执行计划生成），然后把这个table再放回catalog里维护起来，这个时候的执行计划已经是最终要执行的物理执行计划了。但是此时Columner模块相关的转换等操作都是没有触发的。

真正的触发还是在execute()的时候，同其他SparkPlan的execute()方法触发场景是一样的。

**Uncache逻辑**

UncacheTable的时候，除了删除catalog里的table信息之外，还调用了InMemoryColumnarTableScan的cacheColumnBuffers方法，得到RDD集合，并进行了unpersist()操作。cacheColumnBuffers主要做了把RDD每个partition里的ROW的每个Field存到了ColumnBuilder内。

#### UDF（暂不支持）

如前面对SQLContext里Analyzer的分析，其FunctionRegistry没有实现lookupFunction。

在spark-hive项目里，HiveContext里是实现了FunctionRegistry这个trait的，其实现为HiveFunctionRegistry，实现逻辑见org.apache.spark.sql.hive.hiveUdfs

#### Parquet支持

<http://parquet.io/>

Specific Docs and Codes:

<https://github.com/apache/incubator-parquet-format>

<https://github.com/apache/incubator-parquet-mr>

<http://www.slideshare.net/julienledem/parquet-hadoop-summit-2013>

#### JSON支持

SQLContext下，增加了jsonFile的读取方法，而且目前看，代码里实现的是hadoop textfile的读取，也就是这份json文件应该是在HDFS上的。具体这份json文件的载入，InputFormat是TextInputFormat，key class是LongWritable，value class是Text，最后得到的是value部分的那段String内容，即RDD[String]。

除了jsonFile，还支持jsonRDD，例子：

<http://spark.apache.org/docs/latest/sql-programming-guide.html#json-datasets>

读取json文件之后，转换成SchemaRDD。JsonRDD.inferSchema(RDD[String])里有详细的解析json和映射出schema的过程，最后得到该json的LogicalPlan。

Json的解析使用的是FasterXML/jackson-databind库，[GitHub地址](https://github.com/FasterXML/jackson-databind)，[wiki](http://wiki.fasterxml.com/JacksonHome)

把数据映射成Map[String, Any]

Json的支持丰富了Spark SQL数据接入场景。

#### JDBC支持

[Jdbc support branch](https://github.com/apache/spark/tree/branch-1.0-jdbc) is under going

#### SQL92

Spark SQL目前的SQL语法支持情况见SqlParser类。目标是支持SQL92？？

1. 基本应用上，sql server 和oracle都遵循[sql 92语法标准](http://owen.sj.ca.us/rkowen/howto/sql92F.html)。

2. 实际应用中大家都会超出以上标准，使用各家数据库厂商都提供的丰富的自定义标准函数库和语法。

3. 微软sql server的sql 扩展叫T-SQL(Transcate SQL).

4. Oracle 的sql 扩展叫PL-SQL.

## 存在问题

Json file转化会比String的转化快？

待整理

<http://apache-spark-developers-list.1001551.n3.nabble.com/sparkSQL-thread-safe-td7263.html>

<http://apache-spark-user-list.1001560.n3.nabble.com/Supported-SQL-syntax-in-Spark-SQL-td9538.html>

## 例子

select distinct st.sno, sname

from student st join sc sc

on (st.sno = sc.sno)

where sc.cno in ('c001', 'c002', 'c003')

and sc.sno<>'s001'

**Logical**

Distinct

Project ['st.sno,'sname]

Filter ('sc.cno IN (c001,c002,c003) && NOT ('sc.sno = s001))

Join Inner, Some(('st.sno = 'sc.sno))

UnresolvedRelation None, student, Some(st)

UnresolvedRelation None, sc, Some(sc)

**Analyzed**

Distinct

Project [sno#0,sname#1]

Filter (cno#10 IN (c001,c002,c003) && NOT (sno#9 = s001))

Join Inner, Some((sno#0 = sno#9))

SparkLogicalPlan (ExistingRdd [sno#0,sname#1,sage#2,ssex#3], MapPartitionsRDD[2] at mapPartitions at basicOperators.scala:219)

SparkLogicalPlan (ExistingRdd [sno#9,cno#10,score#11], MapPartitionsRDD[14] at mapPartitions at basicOperators.scala:219)

**Optimized**

Distinct

Project [sno#0,sname#1]

Join Inner, Some((sno#0 = sno#9))

Project [sno#0,sname#1]

SparkLogicalPlan (ExistingRdd [sno#0,sname#1,sage#2,ssex#3], MapPartitionsRDD[2] at mapPartitions at basicOperators.scala:219)

Project [sno#9]

Filter (cno#10 IN (c001,c002,c003) && NOT (sno#9 = s001))

SparkLogicalPlan (ExistingRdd [sno#9,cno#10,score#11], MapPartitionsRDD[14] at mapPartitions at basicOperators.scala:219)

**Physical**

Aggregate false, [sno#0,sname#1], [sno#0,sname#1]

Project [sno#0,sname#1]

HashJoin [sno#0], [sno#9], BuildRight

Project [sno#0,sname#1]

ExistingRdd [sno#0,sname#1,sage#2,ssex#3], MapPartitionsRDD[2] at mapPartitions at basicOperators.scala:219

Project [sno#9]

Filter (cno#10 IN (c001,c002,c003) && NOT (sno#9 = s001))

ExistingRdd [sno#9,cno#10,score#11], MapPartitionsRDD[14] at mapPartitions at basicOperators.scala:219