**spark SQL初步认识**

spark SQL是spark的一个模块，主要用于进行结构化数据的处理。它提供的最核心的编程抽象就是DataFrame。

DataFrame：它可以根据很多源进行构建，包括：结构化的数据文件，hive中的表，外部的关系型数据库，以及RDD

**创建DataFrame**

数据文件students.json

{"id":1, "name":"leo", "age":18}

{"id":2, "name":"jack", "age":19}

{"id":3, "name":"marry", "age":17}

spark-shell里创建DataFrame

//将文件上传到hdfs目录下

hadoop@master:~/wujiadong$ hadoop fs -put students.json /student/2016113012/spark

//启动spark shell

hadoop@slave01:~$ spark-shell

//导入SQLContext

scala> import org.apache.spark.sql.SQLContext

import org.apache.spark.sql.SQLContext

//声明一个SQLContext的对象，以便对数据进行操作

scala> val sql = new SQLContext(sc)

warning: there was one deprecation warning; re-run with -deprecation for details

sql: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@27acd9a7

//读取数据

scala> val students = sql.read.json("hdfs://master:9000/student/2016113012/spark/students.json")

students: org.apache.spark.sql.DataFrame = [age: bigint, id: bigint ... 1 more field]

//显示数据

scala> students.show

+---+---+-----+

|age| id| name|

+---+---+-----+

| 18| 1| leo|

| 19| 2| jack|

| 17| 3|marry|

+---+---+-----+

**DataFrame常用操作**

scala> students.show

+---+---+-----+

|age| id| name|

+---+---+-----+

| 18| 1| leo|

| 19| 2| jack|

| 17| 3|marry|

+---+---+-----+

scala> students.printSchema

root

|-- age: long (nullable = true)

|-- id: long (nullable = true)

|-- name: string (nullable = true)

scala> students.select("name").show

+-----+

| name|

+-----+

| leo|

| jack|

|marry|

+-----+

scala> students.select(students("name"),students("age")+1).show

+-----+---------+

| name|(age + 1)|

+-----+---------+

| leo| 19|

| jack| 20|

|marry| 18|

+-----+---------+

scala> students.filter(students("age")>18).show

+---+---+----+

|age| id|name|

+---+---+----+

| 19| 2|jack|

+---+---+----+

scala> students.groupBy("age").count().show

+---+-----+

|age|count|

+---+-----+

| 19| 1|

| 17| 1|

| 18| 1|

+---+-----+

**两种方式将RDD转换成DataFrame**

**1）基于反射方式**

package wujiadong\_sparkSQL

import org.apache.spark.sql.SQLContext

import org.apache.spark.{SparkConf, SparkContext}

/\*\*

\* Created by Administrator on 2017/3/5.

\*/

object RDDDataFrameReflection {

def main(args: Array[String]): Unit = {

val conf = new SparkConf().setAppName("rdddatafromareflection")

val sc = new SparkContext(conf)

val sqlContext = new SQLContext(sc)

val fileRDD = sc.textFile("hdfs://master:9000/student/2016113012/data/students.txt")

val lineRDD = fileRDD.map(line => line.split(","))

//将RDD和case class关联

val studentsRDD = lineRDD.map(x => Students(x(0).toInt,x(1),x(2).toInt))

//在scala中使用反射方式，进行rdd到dataframe的转换，需要手动导入一个隐式转换

import sqlContext.implicits.\_

val studentsDF = studentsRDD.toDF()

//注册表

studentsDF.registerTempTable("t\_students")

val df = sqlContext.sql("select \* from t\_students")

df.rdd.foreach(row => println(row(0)+","+row(1)+","+row(2)))

df.rdd.saveAsTextFile("hdfs://master:9000/student/2016113012/data/out")

}

}

//放到外面

case class Students(id:Int,name:String,age:Int)

运行结果

hadoop@master:~/wujiadong$ spark-submit --class wujiadong\_sparkSQL.RDDDataFrameReflection --executor-memory 500m --total-executor-cores 2 /home/hadoop/wujiadong/wujiadong.spark.jar

17/03/05 22:46:45 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

17/03/05 22:46:48 INFO Slf4jLogger: Slf4jLogger started

17/03/05 22:46:48 INFO Remoting: Starting remoting

17/03/05 22:46:49 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@192.168.1.131:34921]

17/03/05 22:46:49 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.

17/03/05 22:46:51 WARN MetricsSystem: Using default name DAGScheduler for source because spark.app.id is not set.

17/03/05 22:47:00 INFO FileInputFormat: Total input paths to process : 1

17/03/05 22:47:07 INFO deprecation: mapred.tip.id is deprecated. Instead, use mapreduce.task.id

17/03/05 22:47:07 INFO deprecation: mapred.task.id is deprecated. Instead, use mapreduce.task.attempt.id

17/03/05 22:47:07 INFO deprecation: mapred.task.is.map is deprecated. Instead, use mapreduce.task.ismap

17/03/05 22:47:07 INFO deprecation: mapred.task.partition is deprecated. Instead, use mapreduce.task.partition

17/03/05 22:47:07 INFO deprecation: mapred.job.id is deprecated. Instead, use mapreduce.job.id

1,leo,17

2,marry,17

3,jack,18

4,tom,19

17/03/05 22:47:10 INFO FileOutputCommitter: Saved output of task 'attempt\_201703052247\_0001\_m\_000000\_1' to hdfs://master:9000/student/2016113012/data/out/\_temporary/0/task\_201703052247\_0001\_m\_000000

**2）编程接口方式**

package wujiadong\_sparkSQL

import org.apache.spark.sql.types.\_

import org.apache.spark.sql.{Row, SQLContext}

import org.apache.spark.{SparkConf, SparkContext}

/\*\*

\* Created by Administrator on 2017/3/5.

\*/

object RDDDataFrameBianchen {

def main(args: Array[String]): Unit = {

val conf = new SparkConf().setAppName("RDDDataFrameBianchen")

val sc = new SparkContext(conf)

val sqlContext = new SQLContext(sc)

//指定地址创建rdd

val studentsRDD = sc.textFile("hdfs://master:9000/student/2016113012/data/students.txt").map(\_.split(","))

//将rdd映射到rowRDD

val RowRDD = studentsRDD.map(x => Row(x(0).toInt,x(1),x(2).toInt))

//以编程方式动态构造元素据

val schema = StructType(

List(

StructField("id",IntegerType,true),

StructField("name",StringType,true),

StructField("age",IntegerType,true)

)

)

//将schema信息映射到rowRDD

val studentsDF = sqlContext.createDataFrame(RowRDD,schema)

//注册表

studentsDF.registerTempTable("t\_students")

val df = sqlContext.sql("select \* from t\_students order by age")

df.rdd.collect().foreach(row => println(row))

}

}

运行结果

hadoop@master:~/wujiadong$ spark-submit --class wujiadong\_sparkSQL.RDDDataFrameBianchen --executor-memory 500m --total-executor-cores 2 /home/hadoop/wujiadong/wujiadong.spark.jar

17/03/06 11:07:25 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

17/03/06 11:07:27 INFO Slf4jLogger: Slf4jLogger started

17/03/06 11:07:27 INFO Remoting: Starting remoting

17/03/06 11:07:28 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@192.168.1.131:49756]

17/03/06 11:07:32 WARN MetricsSystem: Using default name DAGScheduler for source because spark.app.id is not set.

17/03/06 11:07:38 INFO FileInputFormat: Total input paths to process : 1

17/03/06 11:07:44 INFO deprecation: mapred.tip.id is deprecated. Instead, use mapreduce.task.id

17/03/06 11:07:44 INFO deprecation: mapred.task.id is deprecated. Instead, use mapreduce.task.attempt.id

17/03/06 11:07:44 INFO deprecation: mapred.task.is.map is deprecated. Instead, use mapreduce.task.ismap

17/03/06 11:07:44 INFO deprecation: mapred.task.partition is deprecated. Instead, use mapreduce.task.partition

17/03/06 11:07:44 INFO deprecation: mapred.job.id is deprecated. Instead, use mapreduce.job.id

[1,leo,17]

[2,marry,17]

[3,jack,18]

[4,tom,19]

17/03/06 11:07:47 INFO RemoteActorRefProvider$RemotingTerminator: Shutting down remote daemon.

17/03/06 11:07:47 INFO RemoteActorRefProvider$RemotingTerminator: Remote daemon shut down; proceeding with flushing remote transports.

17/03/06 11:07:47 INFO RemoteActorRefProvider$RemotingTerminator: Remoting shut down.

**DataFrame与RDD**

1）在spark中，DataFrame是一种以RDD为基础的分布式数据集，类似于传统数据库中的二维表格

2）DataFrame与RDD的主要区别就是，前者带有schema元信息，即DataFrame所表示的二维表数据集的每一列都带有名称和类型

参考资料  
<http://9269309.blog.51cto.com/9259309/1851673>

参考资料  
<http://blog.csdn.net/ronaldo4511/article/details/53406069>

参考资料  
<http://spark.apache.org/docs/latest/sql-programming-guide.html#overview>

**整理对Spark SQL的理解**

2014-07-22  [openlog](http://www.360doc.com/userhome/2459)  摘自 [csdn博客](http://blog.csdn.net/pelick/article/details/37809681)  阅 5228  转 7

[转藏到我的图书馆](javascript:void(0);)

[微信](javascript:void(0);)分享：

|  |
| --- |
| * [登录](https://passport.csdn.net/account/login?ref=toolbar)|[注册](https://passport.csdn.net/account/register?ref=toolbar)  [张包峰的博客](http://blog.csdn.net/pelick)大数据处理 | 分布式计算 | 数据仓库  * [http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_1.gif目录视图](http://blog.csdn.net/pelick?viewmode=contents) * [http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_2.gif摘要视图](http://blog.csdn.net/pelick?viewmode=list) * [http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_3.gif订阅](http://blog.csdn.net/pelick/rss/list)   [有奖征资源，博文分享有内涵](http://blog.csdn.net/blogdevteam/article/details/25387451)       [6月推荐文章汇总](http://blog.csdn.net/blogdevteam/article/details/37602759)      [CSDN博文大赛初赛晋级名单公布](http://blog.csdn.net/blogdevteam/article/details/37874361) [整理对Spark SQL的理解](http://blog.csdn.net/pelick/article/details/37809681) 分类： [Spark](http://blog.csdn.net/zbf8441372/article/category/1556747) 2014-07-15 10:18 2382人阅读 [评论](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)(2) [收藏](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml) [举报](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)  [Spark](http://www.csdn.net/tag/spark)[Catalyst](http://www.csdn.net/tag/catalyst)[执行流程](http://www.csdn.net/tag/%e6%89%a7%e8%a1%8c%e6%b5%81%e7%a8%8b)[代码结构](http://www.csdn.net/tag/%e4%bb%a3%e7%a0%81%e7%bb%93%e6%9e%84)[实现理解](http://www.csdn.net/tag/%e5%ae%9e%e7%8e%b0%e7%90%86%e8%a7%a3)  目录[(?)[+]](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml) 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。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_4  **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TreeNodelib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。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_5  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。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_6  **逻辑执行计划实现**  LeafNode主要子类是Command体系：  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_7  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：  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_8  BinaryNode的子类：  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_9 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。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_10  **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：  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_11  UnaryNode的子类：  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_12  BinaryNode的子类：  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_13  提到物理执行计划，还要提一下Catalyst提供的**分区表示模型**。   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。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_14  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：  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. case logical.Project(projectList, child) => 2. execution.Project(projectList, planLater(child)) :: Nil 3. case logical.Filter(condition, child) => 4. execution.Filter(condition, planLater(child)) :: Nil 5. case logical.Aggregate(group, agg, child) => 6. execution.Aggregate(partial = false, group, agg, planLater(child))(sqlContext) :: Nil 7. case logical.Sample(fraction, withReplacement, seed, child) => 8. 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。  具体可以参考org.apache.spark.sql.expressionspackage下的类。   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)。  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. 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查询的数据集必须读取自hive api。下图中的parquet、json等其他文件支持只发生在sql环境下(SQLContext)。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_15 Hive on Spark Hive官方提出了[Hive onSpark的JIRA](https://issues.apache.org/jira/browse/HIVE-7292)。Shark结束之后，拆分为两个方向：  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_16  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, addfile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。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_17  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，  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_18  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 http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_19 解析过程 HiveQl.parseSql()把hql解析成logicalPlan。解析过程，提取出一些command，包括：  2  set key=value  2  cache table  2  uncache table  2  add jar  2  add file  2  dfs  2  source  然后由Hive的ParseDriver把hql解析成AST，得到ASTNode，  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. 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Sparksql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：  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_20 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createtabl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。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_21  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。   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SchemaRD，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  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. implicit def createSchemaRDD[A <: Product: TypeTag](rdd: RDD[A]) = 2. new SchemaRDD(this, 3. 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 http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_22  下图大致对比了Pig，Spark SQL，Shark在实现层次上的区别，仅做参考。  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_23  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_24 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：  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. val batches: Seq[Batch] = Seq( 2. Batch("MultiInstanceRelations", Once, 3. NewRelationInstances), 4. Batch("CaseInsensitiveAttributeReferences", Once, 5. (if (caseSensitive) Nil else LowercaseAttributeReferences :: Nil) : \_\*), 6. Batch("Resolution", fixedPoint, 7. ResolveReferences :: 8. ResolveRelations :: 9. NewRelationInstances :: 10. ImplicitGenerate :: 11. StarExpansion :: 12. ResolveFunctions :: 13. GlobalAggregates :: 14. typeCoercionRules :\_\*), 15. Batch("Check Analysis", Once, 16. CheckResolution), 17. Batch("AnalysisOperators", fixedPoint, 18. EliminateAnalysisOperators) 19. )   3.  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。  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. val batches = 2. Batch("Combine Limits", FixedPoint(100), 3. CombineLimits) :: 4. Batch("ConstantFolding", FixedPoint(100), 5. NullPropagation, 6. ConstantFolding, 7. LikeSimplification, 8. BooleanSimplification, 9. SimplifyFilters, 10. SimplifyCasts, 11. SimplifyCaseConversionExpressions) :: 12. Batch("Filter Pushdown", FixedPoint(100), 13. CombineFilters, 14. PushPredicateThroughProject, 15. PushPredicateThroughJoin, 16. ColumnPruning) :: Nil   4.  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。  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. val strategies: Seq[Strategy] = 2. CommandStrategy(self) :: 3. TakeOrdered :: 4. PartialAggregation :: 5. LeftSemiJoin :: 6. HashJoin :: 7. InMemoryScans :: 8. ParquetOperations :: 9. BasicOperators :: 10. CartesianProduct :: 11. BroadcastNestedLoopJoin :: Nil   5.  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。  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. val batches = 2. Batch("Add exchange", Once, AddExchange(self)) :: 3. Batch("Prepare Expressions", Once, new BindReferences[SparkPlan]) :: Nil   6.  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  **[java]** [view plaincopy?](http://www.360doc.com/content/14/0722/22/2459_396377522.shtml)[在CODE上查看代码片](https://code.csdn.net/snippets/426745)   1. \*    .--------------------------- Column type ID (4 bytes) 2. \*    |   .----------------------- Null count N (4 bytes) 3. \*    |   |   .------------------- Null positions (4 x N bytes, empty if null count is zero) 4. \*    |   |   |     .------------- Compression scheme ID (4 bytes) 5. \*    |   |   |     |   .--------- Compressed non-null elements 6. \*    V   V   V     V   V 7. \*   +---+---+-----+---+---------+ 8. \*   |   |   | ... |   | ... ... | 9. \*  +---+---+-----+---+---------+ 10. \*  \-----------/ \-----------/ 11. \*       header         body   CompressionScheme子类是不同的压缩实现  http://image76.360doc.com/DownloadImg/2014/07/2222/43641653_25  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 待整理  <http://parquet.io/>    Specific Docs and Codes:  [https://github.com/apache/incubator-parquet-format](https://github.com/apache/incubator-parquet-format" \t "_blank)  <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 大家可以跟进社区邮件列表，后续待整理。  <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>   总结 以上整理了对Spark SQL各个模块的实现情况，代码结构，执行流程以及自己对Spark SQL的理解。  理解有偏差的地方欢迎交流讨论 :)  全文完 :) |

# [基于spark1.3.1的spark-sql实战－01](http://blog.csdn.net/stark_summer/article/details/45825177)

标签： [spark](http://www.csdn.net/tag/spark)[sql](http://www.csdn.net/tag/sql)[relational database](http://www.csdn.net/tag/relational%20database)[商业智能](http://www.csdn.net/tag/%e5%95%86%e4%b8%9a%e6%99%ba%e8%83%bd)[structure](http://www.csdn.net/tag/structure)

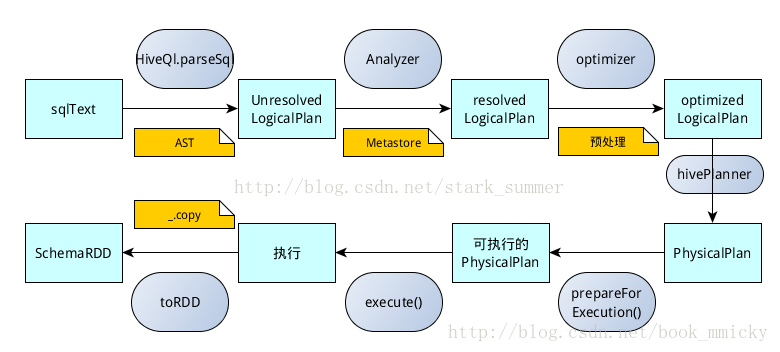
2015-05-19 13:43 22254人阅读 [评论](http://blog.csdn.net/stark_summer/article/details/45825177#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/stark_summer/article/details/45825177#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

spark（61） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

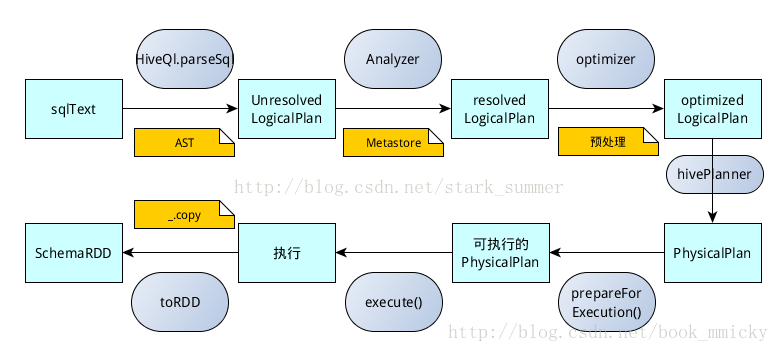
版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/stark_summer/article/details/45825177)



sqlContext总的一个过程如下图所示：

1. SQL语句经过SqlParse解析成UnresolvedLogicalPlan；
2. 使用analyzer结合数据数据字典（catalog）进行绑定，生成resolvedLogicalPlan；
3. 使用optimizer对resolvedLogicalPlan进行优化，生成optimizedLogicalPlan；
4. 使用SparkPlan将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。



1. SQL语句经过HiveQl.parseSql解析成Unresolved LogicalPlan，在这个解析过程中对hiveql语句使用getAst()获取AST树，然后再进行解析；
2. 使用analyzer结合数据hive源数据Metastore（新的catalog）进行绑定，生成resolved LogicalPlan；
3. 使用optimizer对resolved LogicalPlan进行优化，生成optimized LogicalPlan，优化前使用了ExtractPythonUdfs(catalog.PreInsertionCasts(catalog.CreateTables(analyzed)))进行预处理；
4. 使用hivePlanner将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. 执行后，使用map(\_.copy)将结果导入SchemaRDD。

[**Spark**](http://lib.csdn.net/base/spark) sql 三个核心部分：

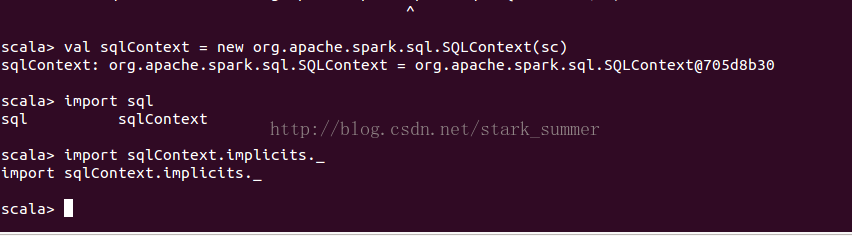
1. 可以加载各种结构化数据源(e.g., JSON, [**Hive**](http://lib.csdn.net/base/hive), and Parquet).  
2.  可以让你通过SQL ,spark 内部程序或者外部工具，通过标准的[**数据库**](http://lib.csdn.net/base/mysql)连接(JDBC/ODBC)连接spark，比如一个商业智能的工具Tableau

3.当你通过使用spark程序，spark sql 提供丰富又智能的SQL或者 regular [**Python**](http://lib.csdn.net/base/python)/[**Java**](http://lib.csdn.net/base/javase)/[**Scala**](http://lib.csdn.net/base/scala) code,包括 join RDDS ,SQL tables ，使用SQL自定义用户函数

# DataFrames

A DataFrame isa distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood.DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

**SQLContext**:



除了SQLContext之外 ，还有HiveContext 来创建,HiveContext包含是SQLContext的，功能比SQLContext更强大，可以操作HiveQL还可以定义UDF，在spark1.3.1以后版本更推荐使用HiveContext,但是需要依赖Hive jar包

## Creating DataFrames

拥有SQLContext就可以创建DataFrames from an [existing RDD](http://spark.apache.org/docs/latest/sql-programming-guide.html#interoperating-with-rdds), from a Hive table, or from [data sources](http://spark.apache.org/docs/latest/sql-programming-guide.html#data-sources).

由于Hadoop使用了 lzo压缩方式，所以也需要在spark指定Hadoop Lzo的jar包，否则会报错”Compression codec com.hadoop.compression.lzo.LzoCodec not found.“

在spark\_home/conf/spark\_env.sh目录增加如下配置：

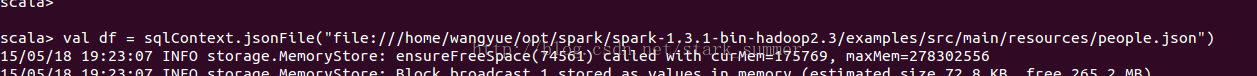
export SPARK\_LIBRARY\_PATH=$SPARK\_LIBRARY\_PATH:/home/wangyue/opt/hadoop/hadoop-2.3.0-cdh5.1.0/lib/native/Linux-amd64-64/\*:/home/wangyue/opt/hadoop/hadoop-2.3.0-cdh5.1.0/share/hadoop/common/hadoop-lzo-0.4.15-cdh5.1.0.jar  
export SPARK\_CLASSPATH=$SPARK\_CLASSPATH:/home/wangyue/opt/hadoop/hadoop-2.3.0-cdh5.1.0/share/hadoop/common/hadoop-lzo-0.4.15-cdh5.1.0.jar

重启spark集群后：

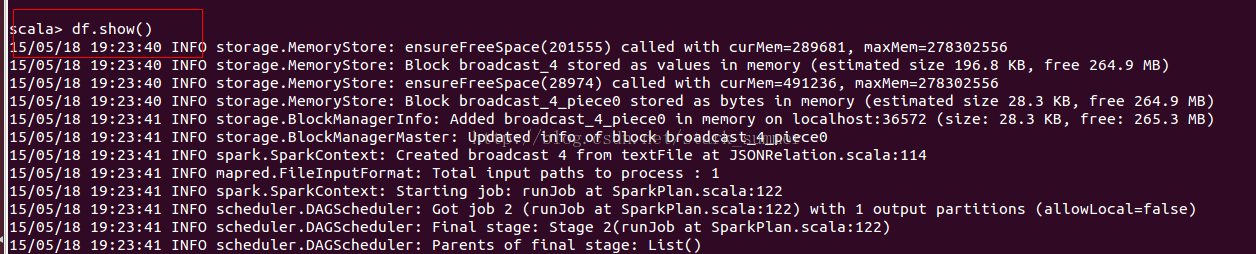


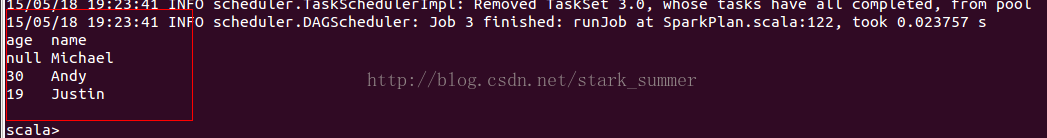
已经可以在classpath中看到lzo jar

加载Json文件：

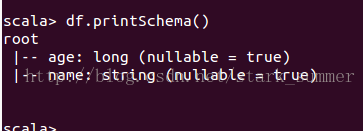


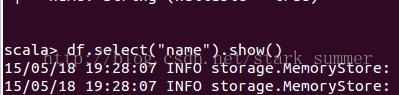
通过show方法查询dataframe数据

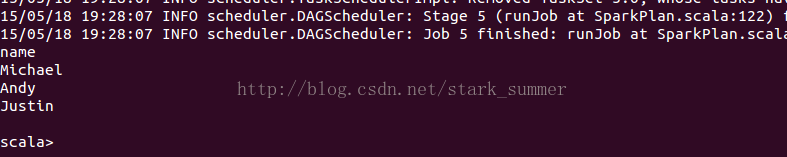




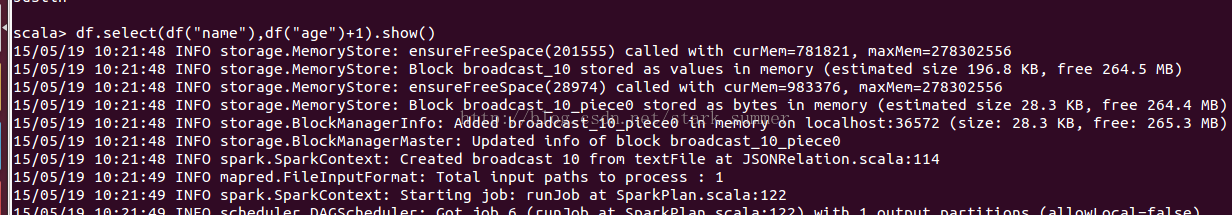
## DataFrame Operations

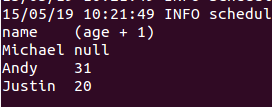




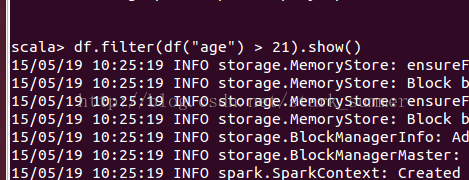


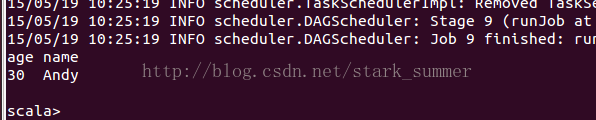
// Select everybody, but increment the age by 1





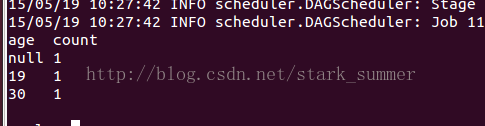
// Select people older than 21





// Count people by age

http://img.blog.csdn.net/20150519102655559?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvc3Rhcmtfc3VtbWVy/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center



## Running SQL Queries Programmatically

The sql function on a SQLContext enables applications to run SQL queries programmatically and returns the result as a DataFrame.

**val** sqlContext **=** ... *// An existing SQLContext*

**val** df **=** sqlContext.sql("SELECT \* FROM table")

目前1.3.1版本后 可以通过SQLContext 运行 SQL程序，然后返回DataFrame格式的结果

目前有两种方式将RDD 转成DataFrame

### 1. Inferring the Schema Using Reflection

*// sc is an existing SparkContext.*

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

*// this is used to implicitly convert an RDD to a DataFrame.*

**import** **sqlContext.implicits.\_**

*// Define the schema using a case class.*

*// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,*

*// you can use custom classes that implement the Product interface.*

**case** **class** **Person**(name**:** String, age**:** Int)

*// Create an RDD of Person objects and register it as a table.*

**val** people **=** sc.textFile("file:///home/wangyue/opt/spark/spark-1.3.1-bin-hadoop2.3/examples/src/main/resources/people.txt").map(**\_**.split(",")).map(p **=>** **Person**(p(0), p(1).trim.toInt)).toDF()

people.registerTempTable("people")

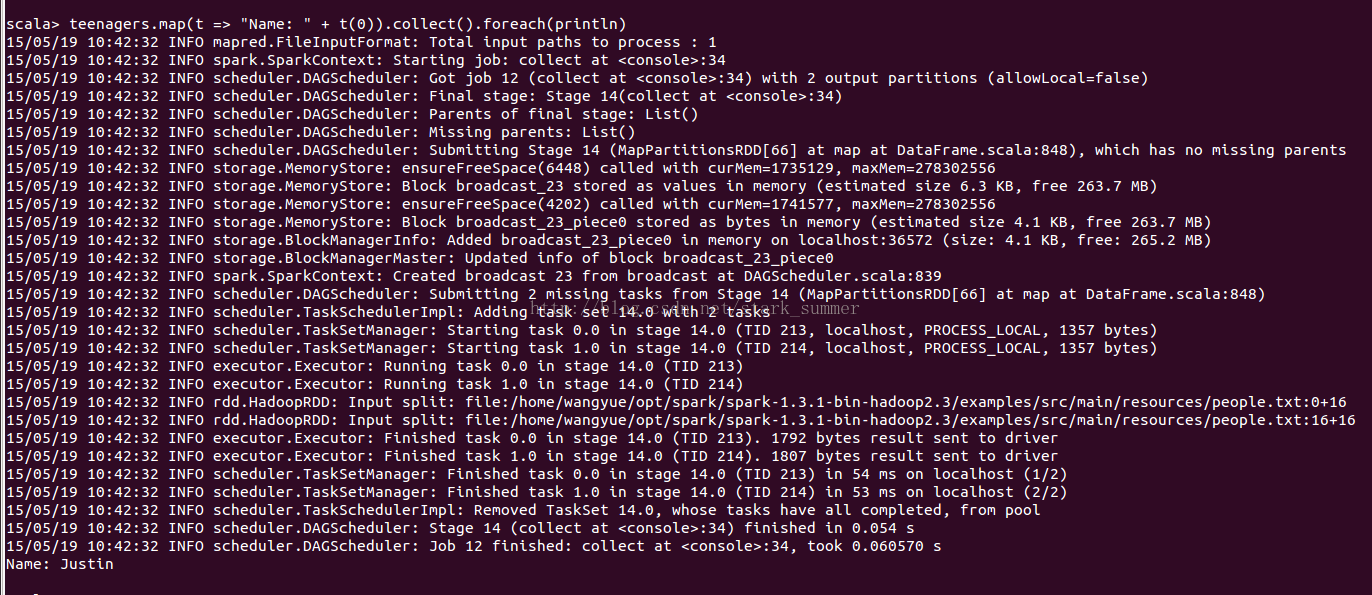
*// SQL statements can be run by using the sql methods provided by sqlContext.*

**val** teenagers **=** sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

*// The results of SQL queries are DataFrames and support all the normal RDD operations.*

*// The columns of a row in the result can be accessed by ordinal.*

teenagers.map(t **=>** "Name: " + t(0)).collect().foreach(println)



### 2. Programmatically Specifying the Schema

When case classes cannot be defined ahead of time (for example, the structure of records is encoded in a string, or a text dataset will be parsed and fields will be projected differently for different users), a DataFrame can be created programmatically with three steps.

1. Create an RDD of Rows from the original RDD;
2. Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
3. Apply the schema to the RDD of Rows via createDataFrame method provided by SQLContext.

For example:

*// sc is an existing SparkContext.*

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

*// Create an RDD*

**val** people **=** sc.textFile("file:///home/wangyue/opt/spark/spark-1.3.1-bin-hadoop2.3/examples/src/main/resources/people.txt")

*// The schema is encoded in a string*

**val** schemaString **=** "name age"

*// Import Row.*

**import** **org.apache.spark.sql.Row**;

*// Import Spark SQL data types*

**import** **org.apache.spark.sql.types.**{**StructType**,**StructField**,**StringType**};

*// Generate the schema based on the string of schema*

**val** schema **=**

**StructType**(

schemaString.split(" ").map(fieldName **=>** **StructField**(fieldName, **StringType**, **true**)))

*// Convert records of the RDD (people) to Rows.*

**val** rowRDD **=** people.map(**\_**.split(",")).map(p **=>** **Row**(p(0), p(1).trim))

*// Apply the schema to the RDD.*

**val** peopleDataFrame **=** sqlContext.createDataFrame(rowRDD, schema)

*// Register the DataFrames as a table.*

peopleDataFrame.registerTempTable("people")

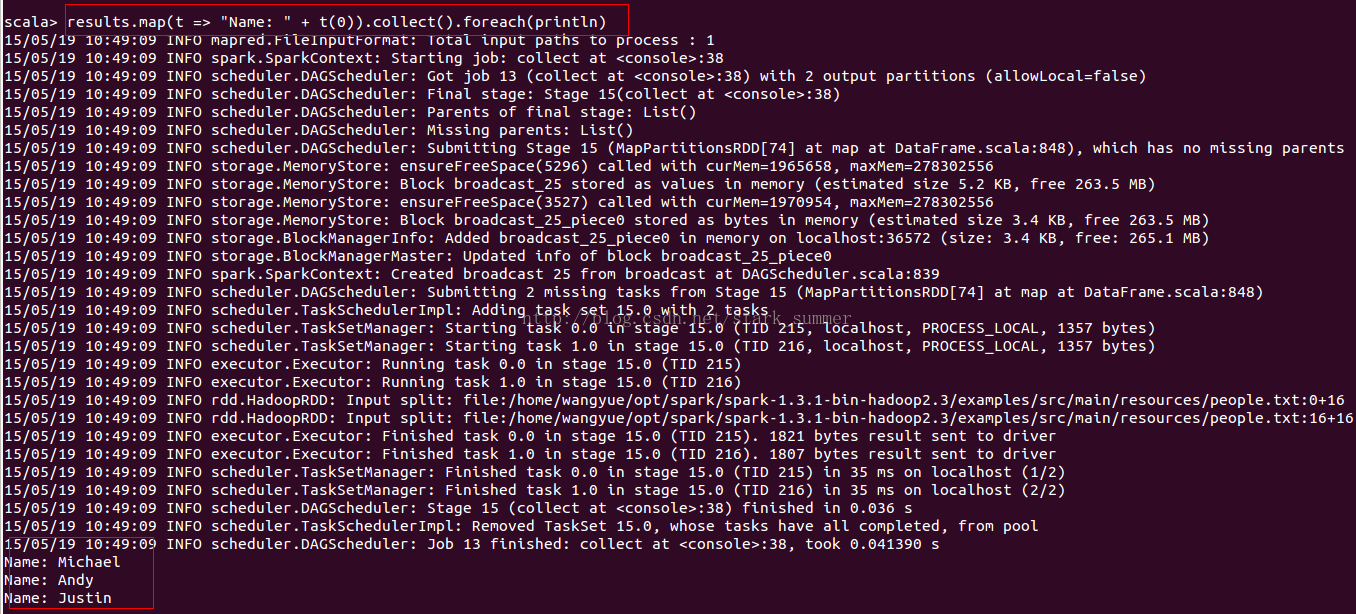
*// SQL statements can be run by using the sql methods provided by sqlContext.*

**val** results **=** sqlContext.sql("SELECT name FROM people")

*// The results of SQL queries are DataFrames and support all the normal RDD operations.*

*// The columns of a row in the result can be accessed by ordinal.*

results.map(t **=>** "Name: " + t(0)).collect().foreach(println)



# Data Sources

## 1. Generic Load/Save Functions

**yal** df **=** sqlContext.load("people.parquet")

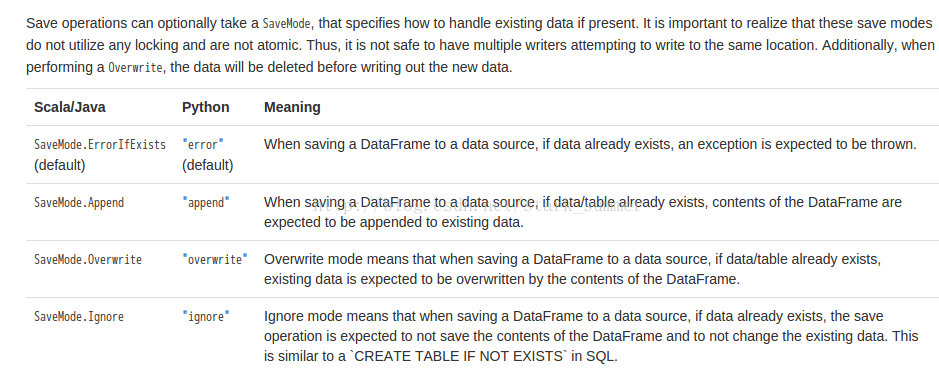
df.select("name", "age").save("namesAndAges.parquet")

### 2. Manually Specifying Options

**val** df **=** sqlContext.load("people.json", "json")

df.select("name", "age").save("namesAndAges.parquet", "parquet")

### Save Modes



### Saving to Persistent Tables

在HiveContext 下，DataFrame 会使用saveAsTable命令会将数据等信息保存到HiveMetastore中，这样即使重启启动spark sql还能活取到HiveMetastore中的数据

在SQLContext下，DataFrame 会使用saveAsTable命令会将数据等信息保存到managed table中，但这些数据通过metastore控制，当表执行drop会删除metastore中数据

## Parquet Files

### 1. Loading Data Programmatically

*// sqlContext from the previous example is used in this example.*

*// This is used to implicitly convert an RDD to a DataFrame.*

**import** **sqlContext.implicits.\_**

**val** people **= sc.textFile("file:///home/wangyue/opt/spark/spark-1.3.1-bin-hadoop2.3/examples/src/main/resources/people.txt").map(\_.split(",")).map(p => Person(p(0), p(1).trim.toInt)).toDF()**

... *// An RDD of case class objects, from the previous example.*

*// The RDD is implicitly converted to a DataFrame by implicits, allowing it to be stored using Parquet.*

people.saveAsParquetFile("people.parquet")

*// Read in the parquet file created above. Parquet files are self-describing so the schema is preserved.*

*// The result of loading a Parquet file is also a DataFrame.*

**val** parquetFile **=** sqlContext.parquetFile("people.parquet")

*//Parquet files can also be registered as tables and then used in SQL statements.*

parquetFile.registerTempTable("parquetFile")

**val** teenagers **=** sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")

*// sqlContext from the previous example is used in this example.*

*// This is used to implicitly convert an RDD to a DataFrame.*

**import** **sqlContext.implicits.\_**

**val** people **= sc.textFile("file:///home/wangyue/opt/spark/spark-1.3.1-bin-hadoop2.3/examples/src/main/resources/people.txt").map(\_.split(",")).map(p => Person(p(0), p(1).trim.toInt)).toDF()**

*// The RDD is implicitly converted to a DataFrame by implicits, allowing it to be stored using Parquet.*

people.saveAsParquetFile("people.parquet")

*// Read in the parquet file created above. Parquet files are self-describing so the schema is preserved.*

*// The result of loading a Parquet file is also a DataFrame.*

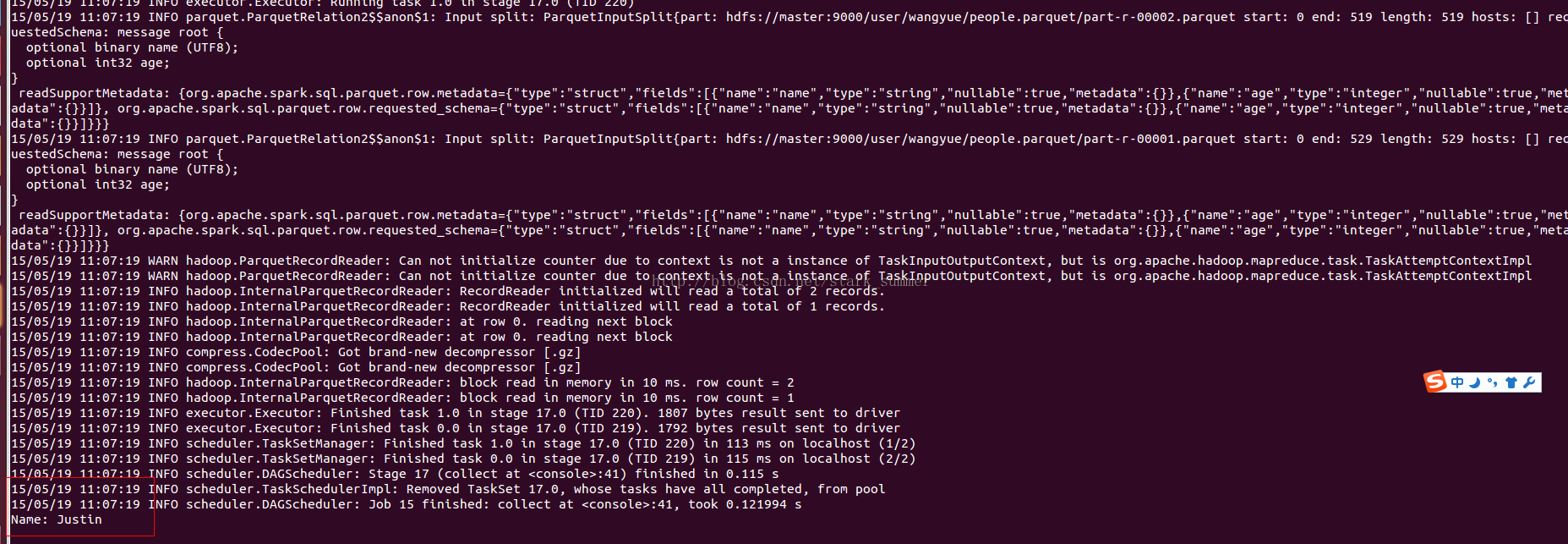
**val** parquetFile **=** sqlContext.parquetFile("people.parquet")

*//Parquet files can also be registered as tables and then used in SQL statements.*

parquetFile.registerTempTable("parquetFile")

**val** teenagers **=** sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")

teenagers.map(t **=>** "Name: " + t(0)).collect().foreach(println)



### Schema merging

*/ sqlContext from the previous example is used in this example.*

*// This is used to implicitly convert an RDD to a DataFrame.*

**import** **sqlContext.implicits.\_**

*// Create a simple DataFrame, stored into a partition directory*

**val** df1 **=** sc.makeRDD(1 to 5).map(i **=>** (i, i \* 2)).toDF("single", "double")

df1.saveAsParquetFile("data/test\_table/key=1")

*// Create another DataFrame in a new partition directory,*

*// adding a new column and dropping an existing column*

**val** df2 **=** sc.makeRDD(6 to 10).map(i **=>** (i, i \* 3)).toDF("single", "triple")

df2.saveAsParquetFile("data/test\_table/key=2")

*// Read the partitioned table*

**val** df3 **=** sqlContext.parquetFile("data/test\_table")

df3.printSchema()

*// The final schema consists of all 3 columns in the Parquet files together*

*// with the partiioning column appeared in the partition directory paths.*

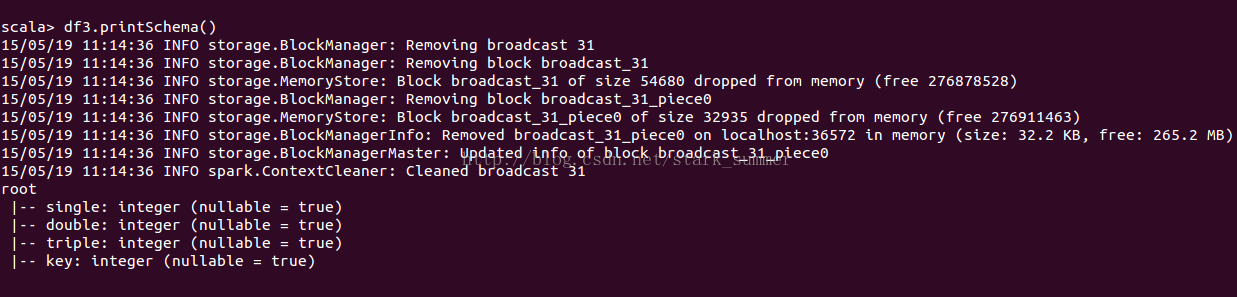
*// root*

*// |-- single: int (nullable = true)*

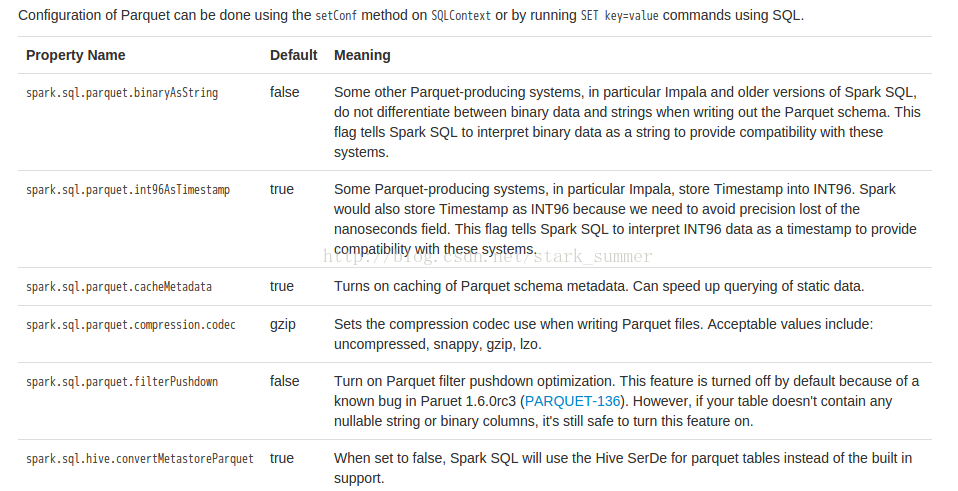
*// |-- double: int (nullable = true)*

*// |-- triple: int (nullable = true)*

*// |-- key : int (nullable = true)*



### Configuration



## JSON Datasets

Spark SQL can automatically infer the schema of a JSON dataset and load it as a DataFrame. This conversion can be done using one of two methods in a SQLContext:

* jsonFile - loads data from a directory of JSON files where each line of the files is a JSON object.
* jsonRDD - loads data from an existing RDD where each element of the RDD is a string containing a JSON object.

Note that the file that is offered as jsonFile is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object. As a consequence, a regular multi-line JSON file will most often fail.

*// sc is an existing SparkContext.*

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

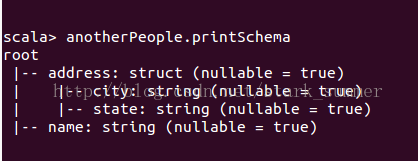
*// A JSON dataset is pointed to by path.*

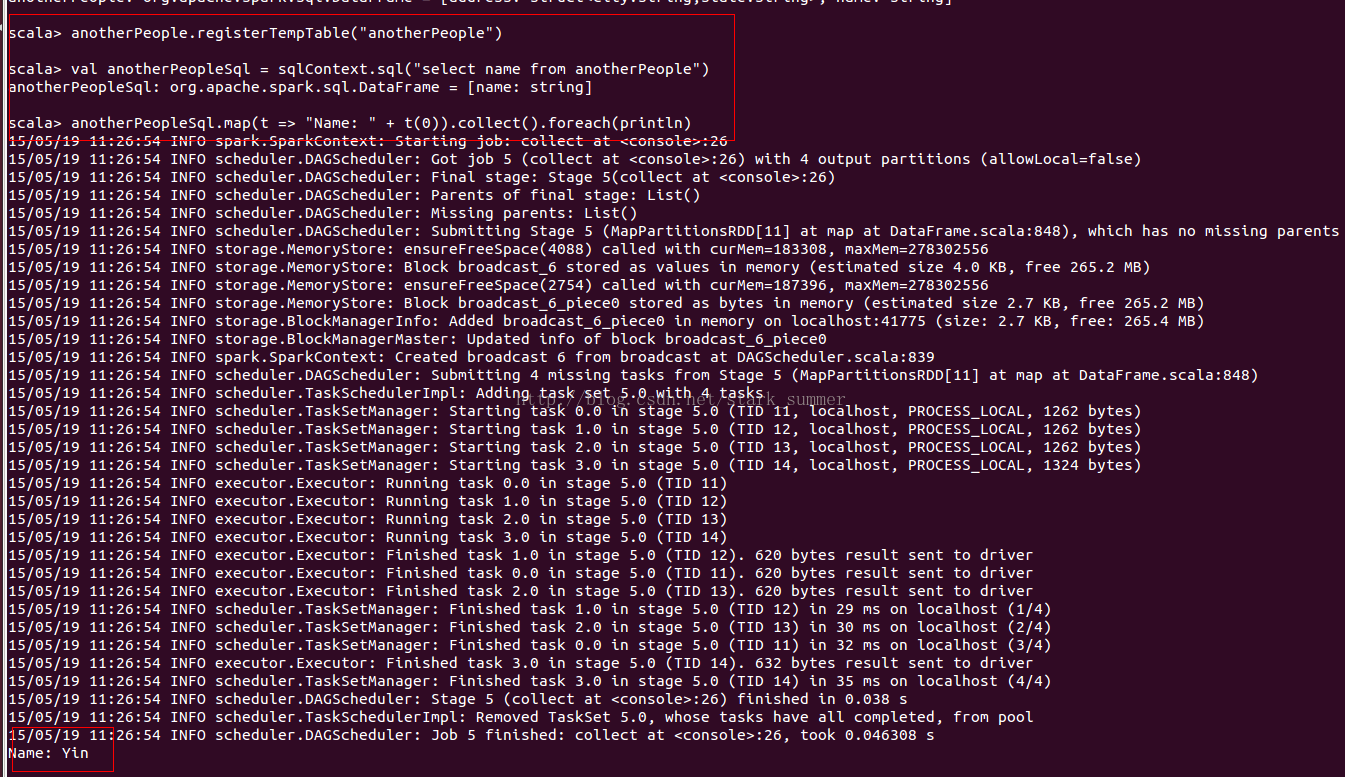
*// The path can be either a single text file or a directory storing text files.*

**val** path **=** "

file:///home/wangyue/opt/spark/spark-1.3.1-bin-hadoop2.3/

examples/src/main/resources/people.json"*// Create a DataFrame from the file(s) pointed to by path***val** people **=** sqlContext.jsonFile(path)*// The inferred schema can be visualized using the printSchema() method.*people.printSchema()*// root// |-- age: integer (nullable = true)// |-- name: string (nullable = true)// Register this DataFrame as a table.*people.registerTempTable("people")*// SQL statements can be run by using the sql methods provided by sqlContext.***val** teenagers **=** sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")*// Alternatively, a DataFrame can be created for a JSON dataset represented by// an RDD[String] storing one JSON object per string.***val** anotherPeopleRDD **=** sc.parallelize( """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: **Nil**)**val** anotherPeople **=** sqlContext.jsonRDD(anotherPeopleRDD)





全部查询：

scala> val anotherPeopleSql = sqlContext.sql("select name,address.city  from anotherPeople")

scala> anotherPeopleSql.map(t => "Name: " + t(0)+ " city:"+t(1)).collect().foreach(println)

