

写在开始：

这是我翻译的《Learning Spark》的第 9 章“Spark SQL”的中文版。供大家学习交流使用。源版权归属于 O'REILLY。本翻译不得用于商业用途。

在 [github](#) 上我找到了该书中文版的第 2 章，在 [51CTO](#) 上我找到了该书中文版的第 3-8 章。于是我就抽空翻译了第 9 章，以后的章节也将陆续的放出。

目录

第 9 章 spark SQL	2
连接 spark SQL （linking with Spark SQL）	3
在程序中使用 Spark SQL （Using Spark SQL in Applications）	5

初始化 Spark SQL （Initializing Spark SQL）	5
基本查询实例（Basic Query Example）	6
SchemaRDDs.....	7
使用 Row objects (Working with Row objects)	8
缓存(Caching).....	9
加载和储存数据（Loading and Saving Data）	10
Apache Hive.....	10
Parquet	11
JSON.....	12
From RDDs	14
JDBC/ODBC server	15
使用 Beeline （working with Beeline）	18
长生命周期的表和查询（Long-Lived Tables and Queries）	19
用户自定义函数（User-Defined Functions）	19
Spark SQL UDFs.....	19
Hive UDFs	20
Spark SQL 性能（Spark SQL 性能）	21
性能调谐选项（Performance Tuning Options）	21
结论（Conclusion）	22

P161

第 9 章 spark SQL

这一章来介绍 Spark SQL，Spark SQL 是 spark 用来处理结构化和半结构化数据的接口。结构化数据是指拥有概要说明的数据——也就是说，一个每个字段都有明确意义的记录的集合。当你处理这种数据的时候，Spark SQL 提供了更加容易和高效的加载和查询这些数据的方法。尤其在以下三种能力上 spark 表现突出（详见图 9-1）：

1. 它可以从各种数据源加载数据（比如 json、hive 和 parquet 等）
2. 它可以让你通过 sql 查询数据。你既可以通过 spark 编程中进行查询；也可以使用拓展工具通过标准的数据库连接器（JDBC/ODBC）来连接 spark SQL，比如说通过业务职能工具 Tableau。

3. 在 spark 编程中, spark sql 在 SQL 和普通的 python/java/scala 语言间提供了丰富的整合方式。例如, 你可以将 RDDs 和 SQL 表进行 join 操作、在 SQL 中使用自己定制的方法。除此之外, 还有很多更简单的方法去做很多的工作。

为了实现这些功能, spark SQL 提供了一种特别的 RDD, 叫做 SchemaRDD。一个 SchemaRDD 是一个包含多行对象的 RDD, 每一行代表一条记录。一个 SchemaRDD 当然也知道它包含的行的格式(等信息)。虽然 SchemaRdds 看起来像普通的 RDDs, 但是它的内部以高效的方式存储数据、保持图表的优势。并且, 它们提供了一些普通的 RDDs 没有的新操作, 比如说运行 sql 查询。SchemaRDDs 可以从外部数据源、查询的结果或者普通的 RDDs 等途径来创建。

P162

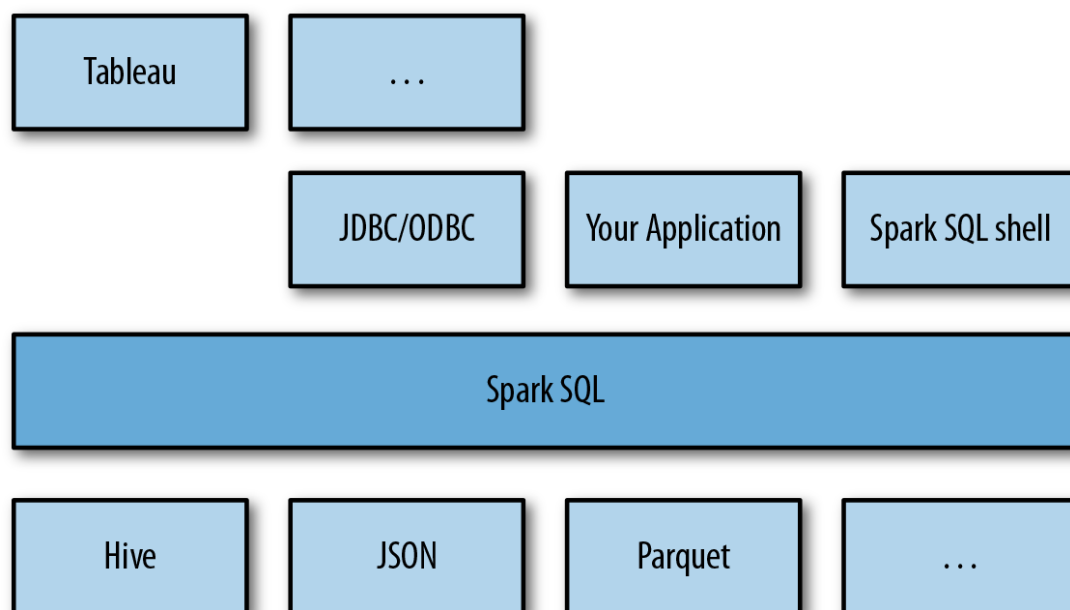


图 9-1

在这一章节中, 我们首先演示如何在内部普通的 spark 编程中使用 schemaRDDs 去加载和查询结构化数据。然后, 我们讲解 Spark SQL JDBC server, 它是一个共享的服务, 可以让你通过 SQL shells 或者其他诸如 Tableau 可视化工具连接它, 并在其上运行 Spark SQL。最后, 我们再来谈一些它的先进的特征。Spark SQL 是一个新的组件, 它将在 spark 1.3 和将来的版本中持续进化, 所以请关注最新版本的 spark SQL 和 schemaRDDs 的文档。

通章, 我们将使用 spark SQL 去探究 json 格式的 tweets 文件。如果你手上没有这样的 tweets 文件, 你可以使用 [Databricks reference application \(http://bit.ly/1yC32hs\)](http://bit.ly/1yC32hs) 去下载一些, 或者使用书中的 git repo (git 资源) files/testtweet.json。

连接 spark SQL (linking with Spark SQL)

我们的应用程序代码需要包含 spark SQL 依赖库文件在内的一些 spark 依赖库文件。这样可以让 spark core 免去依赖大量的其他附件包 (packages) 就可以对应用程序代码进行编译。

Spark SQL 编译时对于 hadoop SQL 引擎 (hive) 的依赖可有有无。支持 hive 功能的 spark SQL 允

许我们访问hive的表, UDFs (user-defined functions), SerDes (serialization and deserialization formats), 和 Hive query language(HiveQL). 需要特别注意的是, 我们虽然需要hive的库文件, 但是并不意味着我们需要真的安装hive。通常, 最好在编译spark SQL时加入对hive的支持去访问这些功能。如果你下载的是二进制的spark, 那么它应该已经包含了对hive的支持。如果你从源码编译出spark, 请运行`sbt/sbt -Phive` 进行装配。

P163

如果你遇到无法排除和屏蔽的与hive冲突的依赖, 你也可以将hive排除在外, 让程序编译和连接到spark SQL。在这种情况下, 你连接到一个单独的maven的artifact.

使用java和scala时, 在example 9-1中展示连接到支持hive的spark SQL的maven坐标。

Example 9-1. Maven coordinates for Spark SQL with Hive support

```
groupId = org.apache.spark
artifactId = spark-hive_2.10
version = 1.2.0
```

如果你不能包含以上的hive依赖, 请使用artifact ID `spark-sql_2.10`替代 `spark-hive_2.10`。

对于python来说, 没有什么spark库需要依赖。

在做程序设计时, 根据是否需要支持hive, 我们有两条程序入口。在需要hive功能支持时, 我们使用HiveContext作为查询入口, 用于访问hiveQL和其他依赖hive的功能。更基础的SQLContext则提供了除了hive功能支持之外的Spark SQL的功能的子集。这种分离是为那些所实现的功能与hive依赖相冲突的用户而准备的。使用HiveContext时不需要真实的安装hive。

HiveQL是spark SQL使用的的查询语言。已经有很多HiveQL的文档, 包括programming hive (<http://shop.oreilly.com/product/0636920023555.do>) 和hive language Manual (<http://bit.ly/1yC3goM>)。在spark 1.0和 spark 1.1 中, Spark SQL 是基于hive 0.12的, 而spark 1.2也还是支持hive 0.13的。如果你已经懂得标准SQL, 使用HiveQL会非常的亲切。



Spark SQL是spark中较新和更新较快的组件。Hive版本的兼容集在将来可能会变动。所有具体细节请关注最新的文档。

最后, 去连接一个真实存在的hive安装, 你必须拷贝hive-site.xml 文件到spark的配置目录 (`$SPARK_HOME/conf`)。如果你没有一个真实安装的hive, Spark SQL也照样能运行。

P164

注意, 如果你没有真实的安装一个hive, spark SQL将会创建一个自己的hive 元数据(metadata DB)在你的程序工作路径中, 叫做metastore_DB。此外, 如果你打算用HiveQL的CREATE TABLE 请求(不是使用CREATE EXTERNAL TABLE), 这些文件将被放入你的默认文件系统(如果你在classpath中配置了hdfs-site.xml,则在HDFS中, 否则可能是你的本机文件系统)中的

/user/hive/warehouse路径中。

在程序中使用 Spark SQL （Using Spark SQL in Applications）

最有力的使用spark SQL的方法是在spark程序中。这样使用，我们可以将SQL和普通的程序代码联合起来使用，完成加载和查询数据等操作。这种方法支持python、java或scala语言编程。

在这种方式下使用spark SQL的话，我们构建基于SparkContext的HiveContext（或者SQLContext，如果想使用简装版的话）。这个context为操作spark SQL的数据提供了额外的操作。使用HiveContext时，我们可以创建SchemaRDDs，它象征了我们的结构化数据，并提供了基于SQL的操作和普通的RDD操作（比如map()）。

初始化 Spark SQL （Initializing Spark SQL）

作为Spark SQL的使用开端，我们需要导入一些东西，如example 9-2所示：

```
Example 9-2. Scala SQL imports
// Import Spark SQL
import org.apache.spark.sql.hive.HiveContext
// Or if you can't have the hive dependencies
import org.apache.spark.sql.SQLContext
```

Scala用户应该注意我们没有导入HiveContext._，就像我们使用SparkContext一样，我们使用隐式转换。这种隐式转换根据所请求的类型信息转换RDDs到专门的Spark SQL的RDDs（从而提供程序操作）。相反，一旦我们构造了一个HiveContext，我们便可以用代码导入隐式转换。代码如example 9-3。

Java和Python的导入操作将分别在example 9-4和example 9-5中介绍。

```
Example 9-3. Scala SQL implicits
// Create a Spark SQL HiveContext
val hiveCtx = ...
// Import the implicit conversions
import hiveCtx._
```

Page 165

```
Example 9-4. Java SQL imports
// Import Spark SQL
import org.apache.spark.sql.hive.HiveContext;
// Or if you can't have the hive dependencies
import org.apache.spark.sql.SQLContext;
// Import the JavaSchemaRDD
```

```
import org.apache.spark.sql.SchemaRDD;  
import org.apache.spark.sql.Row;
```

Example 9-5. Python SQL imports

```
# Import Spark SQL  
from pyspark.sql import HiveContext, Row  
# Or if you can't include the hive requirements  
from pyspark.sql import SQLContext, Row
```

一旦我们导入这些引用，我们需要去创建一个HiveContext，或者创建一个SQLContext（如果不能引入hive依赖的话）（参考example 9-6到example 9-8）。这两种类型都获取一个SparkContext去运行。

Example 9-6. Constructing a SQL context in Scala

```
val sc = new SparkContext(...)  
val hiveCtx = new HiveContext(sc)
```

Example 9-7. Constructing a SQL context in Java

```
JavaSparkContext ctx = new JavaSparkContext(...);  
SQLContext sqlCtx = new HiveContext(ctx);
```

Example 9-8. Constructing a SQL context in Python

```
hiveCtx = HiveContext(sc)
```

现在，我们有了一个HiveContext或者SQLContext，我们已经准备好去加载我们的数据并查询它了。

基本查询实例（Basic Query Example）

针对一张表进行查询操作，我们在HiveContext或SQLContext上使用sql()方法。我们需要做的第一件事是告诉Spark SQL有哪些数据需要被查询。在这种情况下，我们需要从json格式文件中加载一些Twitter数据，并用“temporary table”命名它，以便我们可以用sql去查询它（详细的细节将在第170页“Loading and Saving Data”中阐述）。然后我们可以通过“retweetCount”来select顶部的tweets。请参看example 9-9至example 9-11。

P 166

Example 9-9. Loading and querying tweets in Scala

```
val input = hiveCtx.jsonFile(inputFile)  
// Register the input schema RDD  
input.registerTempTable("tweets")  
// Select tweets based on the retweetCount  
val topTweets = hiveCtx.sql("SELECT text, retweetCount FROM  
tweets ORDER BY retweetCount LIMIT 10")
```

Example 9-10. Loading and querying tweets in Java

```
SchemaRDD input = hiveCtx.jsonFile(inputFile);  
// Register the input schema RDD  
input.registerTempTable("tweets");  
// Select tweets based on the retweetCount  
SchemaRDD topTweets = hiveCtx.sql("SELECT text, retweetCount FROM  
tweets ORDER BY retweetCount LIMIT 10");
```

Example 9-11. Loading and querying tweets in Python

```
input = hiveCtx.jsonFile(inputFile)  
# Register the input schema RDD  
input.registerTempTable("tweets")  
# Select tweets based on the retweetCount  
topTweets = hiveCtx.sql("""SELECT text, retweetCount FROM  
tweets ORDER BY retweetCount LIMIT 10""")
```



如果你已经安装了一个hive，并且拷贝了hive-site.xml到\$SPARK_HOME/conf文件夹下，你也可以运行hiveCtx.sql去查询已经存在的hive tables。

SchemaRDDs

加载数据和查询操作都会返回SchemaRDDs，SchemaRDDs有点像传统数据库中的表。在遮罩下，一个SchemaRDDs由一系列行对象组成，这些行对象附带了每列类型的附加信息。行对象仅仅由基本类型（如integer 和 String）组成的array封装而成。我们将在下面的部分中详细讨论他们的细节。

需要注意一点：在将来的spark版本中，SchemaRDD有可能被更名成DataFrame。至本书出版时，这个更名还在讨论中。

SchemaRDD也是普通的RDDs，所以你可以用已有的RDD变换如map()或filter()等来处理它。然而，他们也提供一些附加功能。其中最重要的，你可以将SchemaRDD注册成临时表，并通过HiveContext.sql或SQLContext.sql来查询它。正如example9-9至example9-11所示，你可以通过SchemaRDD的registerTempTable()方法来完成。

P 167



临时表只能在HiveContext或SQLContext局部使用，当程序退出，临时表就会被消除。

SchemaRDDs能够存储多种基本类型，从结构体到这些结构体的队列。他们使用HiveQL语法进行类型定义。Table 9-1列出了所支持的类型。

Table 9-1. Types stored by SchemaRDDs

Spark SQL/HiveQL type	Scala type	Java type	Python
TINYINT	Byte	Byte/byte	int/Long (in range of -128 to 127)
SMALLINT	Short	Short/short	int/Long (in range of -32768 to 32767)
INT	Int	Int/int	int or long
BIGINT	Long	Long/long	long
FLOAT	Float	Float/float	float
DOUBLE	Double	Double/double	float
DECIMAL	Scala.math.BigDecimal	Java.math.BigDecimal	decimal.Decimal
STRING	String	String	string
BINARY	Array[Byte]	byte[]	bytearray
BOOLEAN	Boolean	Boolean/boolean	bool
TIMESTAMP	java.sql.Timestamp	java.sql.Timestamp	datetime.datetime
ARRAY<DATA_TYPE>	Seq	List	list, tuple, or array
MAP<KEY_TYPE, VAL_TYPE>	Map	Map	dict
STRUCT<COL1: COL1_TYPE, ...>	Row	Row	Row

P 168

最后一种类型，结构，仅代表Spark SQL中的其他行。所有的类型都可以相互嵌套；比如说，你可以将某种结构放入array类型中，或者放入map类型中。

使用 Row objects (Working with Row objects)

行对象在SchemaRDDs中代表记录，是简单的包含字段的定长Array。在Scala/Java中，行对象有一系列的getter方法去获取给定index的字段内容。标准的getter，get（在scala中为apply），返回我们指定列号和类型的数据。对于Boolean、Byte、Double、Float、Int、Long、Short和String，有个getType()方法，能够返回那种类型的数据。比如，getString(0)将会把字段0的内容作为String类型返回。正如你在example 9-12和example 9-13所看到的。

Example 9-12. Accessing the text column (also first column) in the topTweets SchemaRDD in Scala

```
val topTweetText = topTweets.map(row => row.getString(0))
```

Example 9-13. Accessing the text column (also first column) in the topTweets

SchemaRDD in Java

```
JavaRDD<String> topTweetText = topTweets.toJavaRDD().map(new Function<Row, String>() {  
    public String call(Row row) {  
        return row.getString(0);  
    }  
});
```

在python中，由于没有明确的类型，行对象有点不同。我们仅仅通过`row[i]`就可以访问第*i*个元素。并且，Python可以通过`row.column_name`的形式用名称去访问字段。这一点请参看 example 9-14。如果你不确定列名称，我们将在“JSON” on page 172中说明输出表信息的内容。

Example 9-14. Accessing the text column in the topTweets SchemaRDD in Python

```
topTweetText = topTweets.map(lambda row: row.text)
```

Page169

缓存(Caching)

在Spark SQL中，缓存方式有点不同。由于我们已经知道了每一列的类型，spark可以更加高效的存储数据。我们应该使用`hiveCtx.cacheTable("tableName")`去高效的使用内存去缓存，而不是缓存所有对象。Spark SQL表示的数据是以列式的格式存储在内存中。这种缓存的表只存在于驱动程序的生命周期中，所以如果驱动程序退出，我们就需要重新缓存我们的数据。通过使用RDDs，当我们希望运行多个task或者查询在相同的数据上时，我们将缓存这些用到的表。



在spark1.2中，RDDs的普通`cache()`执行的结果与执行`cacheTable()`的结果相同。

你也可以使用HiveQL/SQL声明去缓存表。简单的运行`CACHE tableName` 或者 `UNCACHE TABLE tableName`就可以进行缓存或不缓存表的操作。在使用命令行客户端去操作JDBC server时常使用这种方法。

如图9-2所示，缓存SchemaRDDs在Spark application UI中显示的很像其他的RDDs。

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
PhysicalRDD [contributorsDs#0,createdAt#1,currentUserRetweetId#2,hashtagEntities#3,id#4L,inReplyToScreenName#5,inReplyToStatusId#6L,inReplyToUserId#7L,isFavorited#8,isPossiblySensitive#9,isTruncated#10,mediaEntities#11,retweetCount#12,retweetedStatus#13,source#14,text#15,urlEntities#16,user#17,userMentionEntities#18], MappedRDD[5] at map at JsonRDD.scala:43	Memory Deserialized 1x Replicated	2	100%	1061.7 KB	0.0 B	0.0 B

Figure 9-2. Spark SQL UI of SchemaRDD

我们将在第180页“Spark SQL Performance”中更详细的讨论Spark SQL的缓存表现。

Page 170

加载和储存数据（Loading and Saving Data）

不需要复杂的操作，Spark SQL就可以支持从沙箱（box）外部的各种类型的数据源中获取行对象。这些数据源包括Hive tables、JSON以及Parquet 文件。更进一步，如果你只是用SQL从这些数据源中获取一个子集的字内容，spark SQL可以很聪明的仅仅扫描这些子集的字的数据。而幼稚的SparkContext.hadoopFile则会扫描全部的数据。

除了这些数据源，你也可以在程序中通过制定表结构信息将普通的RDDs转换成SchemaRDDs。这样，即使底层数据是python或java对象，也可以很容易的执行SQL查询。通常，SQL查询在你一次操作大量数据时将更加简洁（比如说你想一步就计算出平均年龄、最大年龄和统计不同的user IDs）。此外，你可以简单的将这些RDDs与其他的Spark SQL数据源进行连接操作。在这部分，我们将涉及到用这种使用RDDs的方法处理外部资源。

Apache Hive

当从hive加载数据，Spark SQL支持hive支持的所有格式，包括text files、RCFiles、ORC、Parquet、Avro和Protocol Buffers。

Spark SQL连接一个已安装好的hive时，需要提供一个hive configuration。你可以把你的hive-site.xml文件拷贝到Spark的./conf/路径中。如果你只是研究，在没有hive-site.xml时，一个本地的hive metastore可以被使用，然后我们可以很轻易的导入数据到hive table中并执行查询操作。

在example 9-15至example 9-17中说明查询一个hive table。我们的example hive table有两列，key（一个integer）和value（一个string）。我们将在这一章的后续部分讲解如何创建这样一张表。

Example 9-15. Hive load in Python

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT key, value FROM mytable")
keys = rows.map(lambda row: row[0])
```

Example 9-16. Hive load in Scala

```
import org.apache.spark.sql.hive.HiveContext
val hiveCtx = new HiveContext(sc)
val rows = hiveCtx.sql("SELECT key, value FROM mytable")
val keys = rows.map(row => row.getInt(0))
```

Page 171

Example 9-17. Hive load in Java

```
import org.apache.spark.sql.hive.HiveContext;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SchemaRDD;
HiveContext hiveCtx = new HiveContext(sc);
SchemaRDD rows = hiveCtx.sql("SELECT key, value FROM mytable");
JavaRDD<Integer> keys = rdd.toJavaRDD().map(new Function<Row, Integer>() {
    public Integer call(Row row) { return row.getInt(0); }
});
```

Parquet

Parquet是一种能用高效的嵌套存储字段方式存储数据的列式存储格式。它通常被hadoop生态圈的工具所使用，且支持spark SQL中的所有数据类型。Spark SQL支持直接从Parquet文件中读取或写入的方法。

首先，如example 9-18所示，你可以使用HiveContext.parquetFile或者SQLContext.parquetFile方法去加载数据。

```
# Load some data in from a Parquet file with field's name and favouriteAnimal
rows = hiveCtx.parquetFile(parquetFile)
names = rows.map(lambda row: row.name)
print "Everyone"
print names.collect()
```

你也可以将Parquet 文件注册成一个Spark SQL 临时表，并对它进行查询。Example9-19是

example 9-18（导入数据）的后续操作。

Example 9-19. Parquet query in Python

```
# Find the panda lovers
tbl = rows.registerTempTable("people")
pandaFriends = hiveCtx.sql("SELECT name FROM people WHERE favouriteAnimal = \"panda\"")
print "Panda friends"
print pandaFriends.map(lambda row: row.name).collect()
```

最后，你可以通过`saveAsParquet()`将SchemaRDD的内容存储为Parquet，如example 9-20所示。

Page 172

Example 9-20. Parquet file save in Python

```
pandaFriends.saveAsParquetFile("hdfs://...")
```

JSON

如果你有一个json文件中的内容适合某种schema，Spark SQL能够通过扫描文件推断schema并且让你能够通过名字来访问字段（example 9-21）。如果你曾经发现你要开始处理一个庞大目录结构的JSON记录，Spark SQL的schema推断是一种有效的方式去处理数据而不需要多写任何特殊的加载代码。

正如example 9-22至example 9-24所示，要加载JSON文件，我们所要做的就是调用hiveContext的`jsonFile()`方法。如果你好奇你数据的推断schema是什么样子的，你可以在结果SchemaRDD上调用`printSchema`（example 9-25）。

Example 9-21. Input records

```
{"name": "Holden"}
{"name": "Sparky The Bear", "lovesPandas": true, "knows": {"friends": ["holden"]}}
```

Example 9-22. Loading JSON with Spark SQL in Python

```
input = hiveCtx.jsonFile(inputFile)
```

Example 9-23. Loading JSON with Spark SQL in Scala

```
val input = hiveCtx.jsonFile(inputFile)
```

Example 9-24. Loading JSON with Spark SQL in Java

```
SchemaRDD input = hiveCtx.jsonFile(jsonFile);
```

Example 9-25. Resulting schema from printSchema()

```
root
|-- knows: struct (nullable = true)
|   |-- friends: array (nullable = true)
|   |   |-- element: string (containsNull = false)
|-- lovesPandas: boolean (nullable = true)
|-- name: string (nullable = true)
```

你也可以查看一些为tweets而生成的schema，如example 9-26所示。

Page 173

Example 9-26. Partial schema of tweets

```
root
|-- contributorsIDs: array (nullable = true)
| |-- element: string (containsNull = false)
|-- createdAt: string (nullable = true)
|-- currentUserRetweetId: integer (nullable = true)
|-- hashtagEntities: array (nullable = true)
| |-- element: struct (containsNull = false)
| | |-- end: integer (nullable = true)
| | |-- start: integer (nullable = true)
| | |-- text: string (nullable = true)
|-- id: long (nullable = true)
|-- inReplyToScreenName: string (nullable = true)
|-- inReplyToStatusId: long (nullable = true)
|-- inReplyToUserId: long (nullable = true)
|-- isFavorited: boolean (nullable = true)
|-- isPossiblySensitive: boolean (nullable = true)
|-- isTruncated: boolean (nullable = true)
|-- mediaEntities: array (nullable = true)
| |-- element: struct (containsNull = false)
| | |-- displayURL: string (nullable = true)
| | |-- end: integer (nullable = true)
| | |-- expandedURL: string (nullable = true)
| | |-- id: long (nullable = true)
| | |-- mediaURL: string (nullable = true)
| | |-- mediaURLHttps: string (nullable = true)
| | |-- sizes: struct (nullable = true)
| | | |-- 0: struct (nullable = true)
| | | | |-- height: integer (nullable = true)
| | | | |-- resize: integer (nullable = true)
| | | | |-- width: integer (nullable = true)
| | | |-- 1: struct (nullable = true)
| | | | |-- height: integer (nullable = true)
| | | | |-- resize: integer (nullable = true)
| | | | |-- width: integer (nullable = true)
| | | |-- 2: struct (nullable = true)
| | | | |-- height: integer (nullable = true)
| | | | |-- resize: integer (nullable = true)
| | | | |-- width: integer (nullable = true)
| | | |-- 3: struct (nullable = true)
```

```

| | | | |-- height: integer (nullable = true)
| | | | |-- resize: integer (nullable = true)
| | | | |-- width: integer (nullable = true)
| | |-- start: integer (nullable = true)
| | |-- type: string (nullable = true)
| | |-- url: string (nullable = true)
|-- retweetCount: integer (nullable = true)
...

```

当你看到这些schemas, 一个自然的问题是如何访问这些嵌套字段和array字段。连同在Python, 当我们定义了一张表, 我们可以通过.(点号)来标示嵌套来访问嵌套元素 (例如 `toplevel.nextlevel`)。

Page 174

你可以在sql中通过指定索引[element]来访问array元素, 如example 9-27所示。

Example 9-27. SQL query nested and array elements

```
select hashtagEntities[0].text from tweets LIMIT 1;
```

From RDDs

除此之外, 我们还可以从RDD创建一个SchemaRDD去加载数据。在scala中, RDDs在某些情况下会被隐私转换成SchemaRDDs。

在python中, 我们先创新一个包含行对象的RDD, 然后再调用inferSchema()方法, 如example 9-28 所示。

Example 9-28. Creating a SchemaRDD using Row and named tuple in Python

```

happyPeopleRDD = sc.parallelize([Row(name="holden", favouriteBeverage="coffee")])
happyPeopleSchemaRDD = hiveCtx.inferSchema(happyPeopleRDD)
happyPeopleSchemaRDD.registerTempTable("happy_people")

```

在scala中, 老朋友隐式转换为我们保持Schema的引用 (Example 9-29)

Example 9-29. Creating a SchemaRDD from case class in Scala

```

case class HappyPerson(handle: String, favouriteBeverage: String)
...
// Create a person and turn it into a Schema RDD
val happyPeopleRDD = sc.parallelize(List(HappyPerson("holden", "coffee")))
// Note: there is an implicit conversion
// that is equivalent to sqlCtx.createSchemaRDD(happyPeopleRDD)
happyPeopleRDD.registerTempTable("happy_people")

```

在java中, 我们能够调用applySchema()将一个包含序列化的class (该class拥有public的getters和setters) 的RDD转换成SchemaRDD。如example 9-30所示。

Example 9-30. Creating a SchemaRDD from a JavaBean in Java

```

class HappyPerson implements Serializable {
    private String name;
    private String favouriteBeverage;
    public HappyPerson() {}
    public HappyPerson(String n, String b) {
        name = n; favouriteBeverage = b;
    }
    public String getName() { return name; }
    public void setName(String n) { name = n; }
    public String getFavouriteBeverage() { return favouriteBeverage; }
    public void setFavouriteBeverage(String b) { favouriteBeverage = b; }
};

...
ArrayList<HappyPerson> peopleList = new ArrayList<HappyPerson>();
peopleList.add(new HappyPerson("holden", "coffee"));
JavaRDD<HappyPerson> happyPeopleRDD = sc.parallelize(peopleList);
SchemaRDD happyPeopleSchemaRDD = hiveCtx.applySchema(happyPeopleRDD,
HappyPerson.class);
happyPeopleSchemaRDD.registerTempTable("happy_people");

```

Page 175

JDBC/ODBC server

Spark SQL 也提供了JDBC的连接性，它支持用商业智能工具（BI）连接到spark集群，也可以让多用户分享spark集群。JDBC server作为一个单独的spark引擎运作，并可被多个客户端分享。任何客户端都可以在内存中缓存表、查询它们，等等等等。并且集群的资源 and 被缓存的数据也是被多个客户端共享的。

Spark SQL的JDBC server正如hive中的hiveServer2，由于它使用Thrift连接协议，所以也被称为“Thrift server”。注意，要运行JDBC server，需要编译时支持了hive的spark（但并不表示你需要实实在在的部署hive。译者注）

可以在spark的目录中用sbin/start-thriftserver.sh来启动服务（example 9-31）。这个脚本列出了很多spark-submit选项（在page122，这里直接列出）

Table 7-2. Common flags for *spark-submit*

Flag	Explanation
<code>--master</code>	Indicates the cluster manager to connect to. The options for this flag are described in Table 7-1.
<code>--deploy-mode</code>	Whether to launch the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster"). In client mode <code>spark-submit</code> will run your driver on the same machine where <code>spark-submit</code> is itself being invoked. In cluster mode, the driver will be shipped to execute on a worker node in the cluster. The default is client mode.
<code>--class</code>	The "main" class of your application if you're running a Java or Scala program.
<code>--name</code>	A human-readable name for your application. This will be displayed in Spark's web UI.
<code>--jars</code>	A list of JAR files to upload and place on the classpath of your application. If your application depends on a small number of third-party JARs, you can add them here.
<code>--files</code>	A list of files to be placed in the working directory of your application. This can be used for data files that you want to distribute to each node.
<code>--py-files</code>	A list of files to be added to the PYTHONPATH of your application. This can contain <i>.py</i> , <i>.egg</i> , or <i>.zip</i> files.
<code>--executor-memory</code>	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 megabytes) or "15g" (15 gigabytes).
<code>--driver-memory</code>	The amount of memory to use for the driver process, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 megabytes) or "15g" (15 gigabytes).

Table 7-1. Possible values for the `--master` flag in *spark-submit*

Value	Explanation
<code>spark://host:port</code>	Connect to a Spark Standalone cluster at the specified port. By default Spark Standalone masters use port 7077.
<code>mesos://host:port</code>	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050.
<code>yarn</code>	Connect to a YARN cluster. When running on YARN you'll need to set the <code>HADOOP_CONF_DIR</code> environment variable to point the location of your Hadoop configuration directory, which contains information about the cluster.
<code>local</code>	Run in local mode with a single core.
<code>local[N]</code>	Run in local mode with N cores.
<code>local[*]</code>	Run in local mode and use as many cores as the machine has.

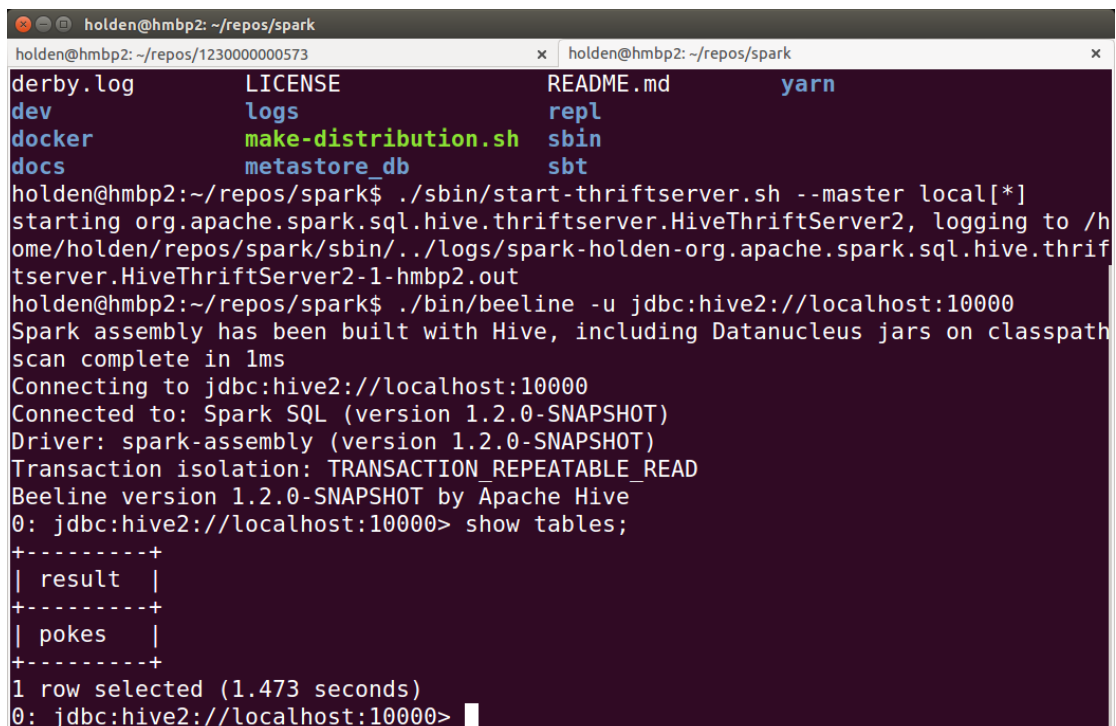
默认情况下，它监听localhost:10000.但我们可以用下面的方式改变这些配置。
如设置环境变量(HIVE_SERVER2_THRIFT_PORT 和 HIVE_SERVER2_THRIFT_BIND_HOST)，或者

改变hive配置项(hive.server2.thrift.port 和 hive.server2.thrift.bind.host)。你也可以在命令行中指定hive选项 --hiveconf property=value。

Example 9-31. Launching the JDBC server
./sbin/start-thriftserver.sh --master sparkMaster

Spark也提供了一个beeline客户端去连接JDBC服务，如example 9-32 和figure 9-3所示。这是一个简单的让你可以在服务器上运行命令的SQL shell。

Example 9-32. Connecting to the JDBC server with Beeline
holden@hmbp2:~/repos/spark\$./bin/beeline -u jdbc:hive2://localhost:10000
Spark assembly has been built with Hive, including Datanucleus jars on classpath
scan complete in 1ms
Connecting to jdbc:hive2://localhost:10000
Connected to: Spark SQL (version 1.2.0-SNAPSHOT)
Driver: spark-assembly (version 1.2.0-SNAPSHOT)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 1.2.0-SNAPSHOT by Apache Hive
0: jdbc:hive2://localhost:10000> show tables;
+-----+
| result |
+-----+
| pokes |
+-----+
1 row selected (1.182 seconds)
0: jdbc:hive2://localhost:10000>



```
holden@hmbp2: ~/repos/spark
holden@hmbp2: ~/repos/1230000000573 x | holden@hmbp2: ~/repos/spark x
derby.log      LICENSE      README.md      yarn
dev            logs          repl
docker         make-distribution.sh  sbin
docs          metastore_db  sbt
holden@hmbp2:~/repos/spark$ ./sbin/start-thriftserver.sh --master local[*]
starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /home/holden/repos/spark/sbin/../logs/spark-holden-org.apache.spark.sql.hive.thriftserver.HiveThriftServer2-1-hmbp2.out
holden@hmbp2:~/repos/spark$ ./bin/beeline -u jdbc:hive2://localhost:10000
Spark assembly has been built with Hive, including Datanucleus jars on classpath
scan complete in 1ms
Connecting to jdbc:hive2://localhost:10000
Connected to: Spark SQL (version 1.2.0-SNAPSHOT)
Driver: spark-assembly (version 1.2.0-SNAPSHOT)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 1.2.0-SNAPSHOT by Apache Hive
0: jdbc:hive2://localhost:10000> show tables;
+-----+
| result |
+-----+
| pokes  |
+-----+
1 row selected (1.473 seconds)
0: jdbc:hive2://localhost:10000>
```

Figure 9-3. Launching the JDBC server and connecting a Beeline client



JDBC server启动后，会把所有的输出重定向到logfile。如果你在访问JDBC server的过程中遇到问题，请在logfile中获取详细的错误信息。

还有很多的外部工具可以通过ODBC连接到Spark SQL。Spark SQL ODBC由Simba (<http://www.simba.com/>) 开发，并且可以从各种spark供应商（如，Databricks Cloud, Datastax, 和 MapR）处获取。它一般被诸如Microstrategy或者Tableau这样的商业智能工具所使用。查看你的工具如何去连接spark SQL。此外，因为spark SQL和hive使用相同的语言和server，这些工具能够连接hive的话，通常也就能够连接spark SQL。

P177

使用 Beeline（working with Beeline）

在Beeline客户端中，你可以使用标准的HiveQL去创建、罗列和查询表。你可以在HiveQL语言手册 (<http://bit.ly/1yC3goM>) 中找到详细的信息。这里，我们只演示一小部分共通操作。

首先，我们要使用本地数据中创建一个表，使用CREATE TABLE命令创建表，然后再LOAD DATA。Hive很轻易的支持加载那些含有固定分隔符的文件，如CSV。如example 9-33

Example 9-33. Load table

```
> CREATE TABLE IF NOT EXISTS mytable (key INT, value STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
> LOAD DATA LOCAL INPATH 'learning-spark-examples/files/int_string.csv'
INTO TABLE mytable;
```

要罗列表，你可以使用SHOW TABLES声明（example 9-34）。你也可以用DESCRIBE tablename查看每个表的表结构。

Example 9-34. Show tables

```
> SHOW TABLES;
mytable
Time taken: 0.052 seconds
```

如果你要缓存表，请使用CACHE tablename，之后你可以使用UNCACHE tableName来释放它们。注意，正像先前说明的那样，这些缓存是被该JDBC server的所有客户端共享的。

最后，Beeline可以很容易的查看查询计划（步骤）。你可以在给定的查询上运行EXPLAIN去查看查询计划（步骤），如example 9-35所示。

Example 9-35. Spark SQL shell EXPLAIN

```
spark-sql> EXPLAIN SELECT * FROM mytable where key = 1;
```

```
== Physical Plan ==
Filter (key#16 = 1)
HiveTableScan [key#16,value#17], (MetastoreRelation default, mytable, None), None
Time taken: 0.551 seconds
```

在这个具体的查询计划中，Spark SQL在紧挨着运行HiveTableScan前，运行了一个filter操作。
P178

在这里，你也可以写SQL去查询数据。Beeline shell可以很迅速的在多用户共享的缓存表中进行数据检索。

长生命周期的表和查询（Long-Lived Tables and Queries）

Spark SQL's server的一个长处是我们可以多个程序中共享缓存的表。自从JDBC Thrift server成了一个独立的驱动程序后（这便成为了可能）。正如先前所示，你只要先注册表，再运行CACHE命名去缓存它。



Standalone Spark SQL Shell

除了JDBC server，你也可以通过./bin/spark-sql去在spark SQL上运行一个单进程的简单shell。如果已经在conf/hive-site.xml中设置了Hive metastore，这个shell可以连接到它；如果没有设置，则会在本地创建一个（metastore）。这个在本地开发中很有用。

但是在一个共享的集群中，你应该使用能让多个用户通过Beeline去连接的JDBC server。

用户自定义函数（User-Defined Functions）

User-defined functions, 或者 UDFs, 让你在SQL中使用自定义的函数（这些函数可以是Python、Java和Scala编写的）。在一个团体中这是一种非常受欢迎的方式去在SQL中暴露定制功能，这样大家就可以直接使用它而不需要重复的编码。Spark SQL让UDFs的编写尤其的简单。它既支持自己的UDF接口，也支持已经存在的Apache Hive UDFs。

Spark SQL UDFs

在你的程序语言中，Spark SQL提供了一种内置的方法去方便的通过传递一个函数去注册UDFs。在Scala和Python中，我们可以使用编程语言中的本地函数和lambda表达式，而在java中，我们只要适当的去扩展适当的UDF class。我们的UDFs可以处理各种类型，且我们可以返回一个与输入类型不同类型的返回值。

在Python和java中，我们需要指定一种在table 9-1（page 167）列出的SchemaRDD的(存储数据的)类型。在java中，可以在org.apache.spark.sql.api.java.DataType中找到这些类型，而在Python中，我们需要import the DataType。

在example 9-36和example 9-37中，使用了一个很简单的UDF去计算字符串的长度，我们可以

用这个UDF去找出我们所使用的tweets的长度。

P179

Example 9-36. Python string length UDF

```
# Make a UDF to tell us how long some text is
hiveCtx.registerFunction("strLenPython", lambda x: len(x), IntegerType())
lengthSchemaRDD = hiveCtx.sql("SELECT strLenPython('text') FROM tweets LIMIT 10")
```

Example 9-37. Scala string length UDF

```
registerFunction("strLenScala", (_, String).length)
val tweetLength = hiveCtx.sql("SELECT strLenScala('tweet') FROM tweets LIMIT 10")
```

在java定义UDFs，需要一些额外的导入项。我们extend特殊的class来为RDD自定义操作。取决于输入参数数的个数，我们extend[n]，如example 9-38 和example 9-39所示。

Example 9-38. Java UDF imports

```
// Import UDF function class and DataTypes
// Note: these import paths may change in a future release
import org.apache.spark.sql.api.java.UDF1;
import org.apache.spark.sql.types.DataTypes;
```

Example 9-39. Java string length UDF

```
hiveCtx.udf().register("stringLengthJava", new UDF1<String, Integer>() {
    @Override
    public Integer call(String str) throws Exception {
        return str.length();
    }
}, DataTypes.IntegerType);
SchemaRDD tweetLength = hiveCtx.sql(
    "SELECT stringLengthJava('text') FROM tweets LIMIT 10");
List<Row> lengths = tweetLength.collect();
for (Row row : result) {
    System.out.println(row.get(0));
}
```

Hive UDFs

Spark SQL也可以使用已经存在的Hive UDFs。标准的Hive UDFs已经被包含在内。如果你要使用自定义的UDF，请确保在你的应用中已经包含了容纳了UDF的JARs。如果你运行JDBC server，注意我们可以加入—jars这个命令行标志。开发Hive UDFs超出了本书的讲解范围，这里我们只介绍如何使用已有的Hive UDFs。

P180

要使用Hive UDF，需要我们使用HiveContext而不是普通的SQLContext。启用一个Hive UDF，简单的调用hiveCtx.sql("CREATE TEMPORARY FUNCTION name AS class.function")就可以了。

Spark SQL 性能（Spark SQL 性能）

正如在引言中所提到的，spark SQL 是更高级的查询语言，且额外的类型信息让spark SQL更加高效。

Spark SQL不仅仅是面向那些熟悉SQL的人。Spark SQL可以非常简单的去实现条件聚合操作，如计算多列的和（如examp 9-40所示）而不需要构造我们在Chapter 6中讨论的特殊对象。

Example 9-40. Spark SQL multiple sums
SELECT SUM(user.favouritesCount), **SUM**(retweetCount), **user.id** **FROM** tweets
GROUP BY user.id

Spark SQL能够使用类型知识去更有效的表示数据。当缓存数据，Spark SQL使用内存内分列存储。这个不仅让缓存时更省空间，并且如果随后的查询只涉及到部分数据，Spark SQL可以最小量的读取数据。

谓词下推（Predicate push-down）让spark SQL将一些我们的查询“下移”到引擎。通常，当我们在spark中只想读取部分数据，标准的做法是保持整个数据集然后在上面执行一个filter操作。然而，在Spark SQL中，如果底层的数据存储支持key范围的局部查询或者其他什么限定，spark SQL能够在我们的查询中将这限定下移到数据存储（层面），从而可能减少数据的读取量。

（打个比方，假设有一张有10个key的表，现在只需要按条件查询其中2个key的内容，谓词下推（Predicate push-down）机制可以只把要用的两个key的列读入内存，而其他的内容被无视。译者注）

性能调谐选项（Performance Tuning Options）

在spark SQL中有很多不同的性能调谐选项；它们罗列在Table 9-2中

Option	Default	Usage
spark.sql.codegen	false	When true, Spark SQL will compile each query to Java bytecode on the fly. This can improve performance for large queries, but codegen can slow down very short queries.
spark.sql.inMemoryColumnarStorage.compressed	false	Compress the in-memory columnar storage automatically. 自动压缩内存中的列式存储
spark.sql.inMemoryColumnarStorage.batch	1000	The batch size for columnar caching.

Size		Larger values may cause out-of-memory problems
spark.sql.parquet.compression.codec	snappy	Which compression codec to use. Possible options include uncompressed, snappy, gzip, and lzo.

我们可以在JDBC连接器、Beeline Shell中使用set命令去设置这些性能选项和其他选项，如example 9-41所示。

Example 9-41. Beeline command for enabling codegen

```
beeline> set spark.sql.codegen=true;
SET spark.sql.codegen=true
spark.sql.codegen=true
Time taken: 1.196 seconds
```

在传统的spark SQL应用中，我们可以在Spark configuration中设置这些属性，如example 9-42所示。

Example 9-42. Scala code for enabling codegen

```
conf.set("spark.sql.codegen", "true")
```

Page 181

有几个选项需要特别注意。第一个是spark.sql.codegen，这个选项使Spark SQL在运行查询前编译每个查询操作到java字节码。Codegen可以让长查询或者频繁的重复查询变得相当的迅速。因为它创建优化的代码去运行它们。但是，在一个特别短（1-2秒）的临时查询中，因为每次查询都要运行一下编译，从而可能导致额外的开销。CodeGen仍然在测试中，但是我们推荐在进行大数据量查询或相同的查询被一遍又一遍的执行时使用它。

Page 182

你可能需要调整的第二个选项是spark.sql.inMemoryColumnarStorage.batchSize。当缓存SchemaRDD时，Spark SQL按照这个选项（默认：1000）分批次在RDD中分组聚合记录，并压缩每批。小尺寸batch导致低压缩，但另一方面，大尺寸batch也会导致问题发生，因为太大的batch有可能太大了而无法在内存中被建立。如果你表中的行太大（比如，每行有几百个字段，或者某个String字段太长，如网页内容），你应该缩小batch size去避免out-of-memory错误。如果是这样，默认的batch size可能很好，因为当你超过1000条记录时会有额外的压缩来消减返回量（as there are diminishing returns for extra compression when you go beyond 1,000 records.）

结论（Conclusion）

关于Spark SQL，我们已经知道了如何使用spark去处理结构化和半结构化数据。此外再提醒一下，请记住我们先前的操作（从chapter3 到 chapter6）也可以用在SchemaRDDs上，Spark Sqi支持这些。在很多途径，可以很方便的在代码中结合使用SQL（因为它们有表达复杂逻辑的能力）（且很简洁）。当你这样使用Spark SQL，你会从引擎中得到很多优化去处理表格。