# Spark SQL中对Json支持的详细介绍

在这篇文章中，我将介绍一下Spark SQL对Json的支持，这个特性是Databricks的开发者们的努力结果，它的目的就是在Spark中使得查询和创建JSON数据变得非常地简单。随着WEB和手机应用的流行，JSON格式的数据已经是WEB Service API之间通信以及数据的长期保存的事实上的标准格式了。但是使用现有的工具，用户常常需要开发出复杂的程序来读写分析系统中的JSON数据集。而Spark SQL中对JSON数据的支持极大地简化了使用JSON数据的终端的相关工作，Spark SQL对JSON数据的支持是从1.1版本开始发布，并且在Spark 1.2版本中进行了加强。

## 现有Json工具实践

在实践中，用户往往在处理现代分析系统中JSON格式的数据中遇到各种各样的困难。如果用户需要将数据集写成JSON格式的话，他们需要编写复杂的逻辑程序来转换他们的数据集到JSON格式中。如果需要读取或者查询JSON数据集，他们通常需要预先定义好数据结构并用它来转换JSON数据。在这种情况下，用户必须等待这些数据处理完成之后，才能够使用他们生成的JSON数据。无论是在写或者是读，预先定义和维护这些模式往往使得ETL工作变得非常地繁重！并且可能消除掉JSON这种半结构化(semi-structured)的数据格式的好处。如果用户想消费新的数据，他们不得不在创建外部表的时候定义好相关的模式，并使用自定义的JSON serialization/deserialization依赖库，或者是在查询JSON数据的时候使用UDF函数。

作为一个例子，如果有下面的一些JSON数据模式

```
{"name":"Yin", "address":{"city":"Columbus","state":"Ohio"}}
{"name":"Michael", "address":{"city":null, "state":"California"}}
```

在类似于Hive的系统中，这些JSON对象往往作为一个值储存到单个的列中，如果需要访问这个数据，我们需要使用UDF来抽取出我们需要的数据。在下面的SQL查询例子中，外层的字段(name和address)被抽取出来，嵌套在内层的address字段也被进一步的抽取出来：

```
/**
 * User: 过往记忆
 * Date: 15-02-04
 * Time: 上午07:30
 * bolg:
 * 本文地址：/archives/1260
 * 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 * 过往记忆博客微信公共帐号：iteblog_hadoop
 */

SELECT
  v1.name, v2.city, v2.state
```

```
FROM people
  LATERAL VIEW json_tuple(people.jsonObject, 'name', 'address') v1
    as name, address
  LATERAL VIEW json_tuple(v1.address, 'city', 'state') v2
    as city, state;
```

## Spark SQL中对JSON的支持

Spark SQL提供了内置的语法来查询这些JSON数据，并且在读写过程中自动地推断出JSON数据的模式。Spark SQL可以解析出JSON数据中嵌套的字段，并且允许用户直接访问这些字段，而不需要任何显示的转换操作。上面的查询语句如果使用Spark SQL的话，可以这样来写：

```
SELECT name, age, address.city, address.state FROM people
```

## 在Spark SQL中加载和保存JSON数据集

为了能够在Spark SQL中查询到JSON数据集，唯一需要注意的地方就是指定这些JSON数据存储的位置。这些数据集的模式是直接可以推断出来，并且内置就有相关的语法支持，不需要用户显示的定义。在编程中使用API中，我们可以使用SQLContext提供的jsonFile和jsonRDD方法。使用这两个方法，我们可以利用提供的JSON数据集来创建SchemaRDD
对象。并且你可以将SchemaRDD 注册成表。下面是一个很好的例子：

```
/**
 * User: 过往记忆
 * Date: 15-02-04
 * Time: 上午07:30
 * bolg:
 * 本文地址：/archives/1260
 * 过往记忆博客，专注于hadoop、hive、spark、shark、flume的技术博客，大量的干货
 * 过往记忆博客微信公共帐号：iteblog_hadoop
 */
// Create a SQLContext (sc is an existing SparkContext)
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// Suppose that you have a text file called people with the following content:
// {"name":"Yin", "address":{"city":"Columbus","state":"Ohio"}}
// {"name":"Michael", "address":{"city":null, "state":"California"}}
// Create a SchemaRDD for the JSON dataset.
val people = sqlContext.jsonFile("[the path to file people]")
// Register the created SchemaRDD as a temporary table.
people.registerTempTable("people")
```

当然，我们也可以使用纯的SQL语句来创建JSON数据集。例如

```
CREATE TEMPORARY TABLE people
USING org.apache.spark.sql.json
OPTIONS (path '[the path to the JSON dataset]')
```

在上面的例子中，因为我们没有显示地定义模式，Spark SQL能够自动地扫描这些JSON数据集，从而推断出相关的模式。如果一个字段是JSON对象或者数组，Spark SQL将使用STRUCT 类型或者ARRAY类型来代表这些字段。即使JSON数是半结构化的数据，并且不同的元素肯恩好拥有不同的模式，但是Spark SQL仍然可以解决这些问题。如果你想知道JSON数据集的模式，你可以通过使用返回来的SchemaRDD
对象中提供的printSchema()函数来打印出相应的模式，或者你也可以在SQL中使用DESCRIBE [table name] 。例如上面的people数据集的模式可以通过people.printSchema() 打印出：

```
root
 |-- address: struct (nullable = true)
 |    |-- city: string (nullable = true)
 |    |-- state: string (nullable = true)
 |-- name: string (nullable = true)
```

当然，用户在利用 jsonFile 或
jsonRDD创建表的时候也可以显示的指定一个模式到JSON数据集中。在这种情况下，Spark SQL将把这个模式和JSON数据集进行绑定，并且将不再会去推测它的模式。用户不需要了解JSON数据集中所有的字段。指定的模式可以是固定数据集的一个子集，也可以包含JSON数据集中不存在的字段。

当用户创建好代表JSON数据集的表时，用户可以很简单地利用SQL来对这个JSON数据集进行查询，就像你查询普通的表一样。在Spark
SQL中所有的查询，查询的返回值是SchemaRDD对象。例如：

```
val nameAndAddress = sqlContext.sql("SELECT name, address.city, address.state FROM people
")
nameAndAddress.collect.foreach(println)
```

查询的结果可以直接使用，或者是被其他的分析任务使用，比如机器学习。当然，JSON数据集可以通过Spark
SQL内置的内存列式存储格式进行存储，也可以存储成其他格式，比如Parquet或者 Avro。

## 将SchemaRDD对象保存成JSON文件

在Spark SQL中，SchemaRDDs可以通过toJSON
方法保存成JSON格式的文件。因为SchemaRDD中已经包含了相应的模式，所以Spark SQL可以自动地将该数据集转换成JSON，而不需要用户显示地指定。当然，SchemaRDDs可以通过很多其他格式的数据源进行创建，比如Hive tables、 Parquet文件、 JDBC、Avro文件以及其他SchemaRDD的结果。这就意味着用户可以很方便地将数据写成JSON格式，而不需要考虑到源数据集的来源。

本文翻译自：http://databricks.com/blog/2015/02/02/an-introduction-to-json-support-in-spark-sql.html