

Spark SQL, DataFrames and Datasets Guide

zhen-peng.yang@hpe.om | 2016-10-27

|  |  |  |  |
| --- | --- | --- | --- |
| Document Change History | | | |
| Version | **Date** | **Author** | **Description of Changes** |
| v1 | 2016-11-3 | Arvin | Intitial |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[**Overview** 4](#_Toc469303750)

[**SQL** 4](#_Toc469303751)

[**Datasets and DataFrames** 4](#_Toc469303752)

[**Getting Started** 5](#_Toc469303753)

[**Starting Point: SparkSession** 5](#_Toc469303754)

[**Starting Point: SQLContext（旧版，自己添加）** 7](#_Toc469303755)

[**Creating DataFrames** 8](#_Toc469303756)

[**Untyped Dataset Operations (aka DataFrame Operations)** 9](#_Toc469303757)

[**Running SQL Queries Programmatically** 16](#_Toc469303758)

[**Creating Datasets** 17](#_Toc469303759)

[**Interoperating with RDDs （DataFrames与RDDs的相互转换）** 22](#_Toc469303760)

[**Inferring the Schema Using Reflection使用反射获取Schema** 22](#_Toc469303761)

[**Programmatically Specifying the Schema通过编程接口指定Schema** 27](#_Toc469303762)

[**Data Sources** 33](#_Toc469303763)

[**Generic Load/Save Functions一般Load/Save方法** 33](#_Toc469303764)

[**Manually Specifying Options手动指定选项** 34](#_Toc469303765)

[**Run SQL on files directly** 34](#_Toc469303766)

[**Save Modes存储模式** 35](#_Toc469303767)

[**Saving to Persistent Tables持久化到表** 36](#_Toc469303768)

[**Parquet Files** 37](#_Toc469303769)

[**Loading Data Programmatically读取Parquet文件** 37](#_Toc469303770)

[**Partition Discovery解析分区信息** 40](#_Toc469303771)

[**Schema Merging（Schema合并）** 41](#_Toc469303772)

[**Hive metastore Parquet table conversion（Hive metastore Parquet表转换）** 44](#_Toc469303773)

[**Configuration（配置）** 46](#_Toc469303774)

[**JSON Datasets （JSON数据集）** 47](#_Toc469303775)

[**Hive Tables（Hive表）** 50](#_Toc469303776)

[**Interacting with Different Versions of Hive Metastore（访问不同版本的Hive Metastore）** 55](#_Toc469303777)

[**JDBC To Other Databases** 56](#_Toc469303778)

[**Troubleshooting（故障排除）** 59](#_Toc469303779)

[**Performance Tuning（性能调优）** 59](#_Toc469303780)

[**Caching Data In Memory（缓存数据至内存）** 59](#_Toc469303781)

[**Other Configuration Options（调优参数）** 60](#_Toc469303782)

[**Distributed SQL Engine（分布式SQL引擎）** 61](#_Toc469303783)

[**Running the Thrift JDBC/ODBC server(运行Thrift JDBC/ODBC服务)** 62](#_Toc469303784)

[**Running the Spark SQL CLI(运行Spark SQL CLI)** 63](#_Toc469303785)

[**Migration Guide** 64](#_Toc469303786)

[**Upgrading From Spark SQL 1.6 to 2.0** 64](#_Toc469303787)

[**Upgrading From Spark SQL 1.5 to 1.6** 64](#_Toc469303788)

[**Upgrading From Spark SQL 1.4 to 1.5** 65](#_Toc469303789)

[**Upgrading from Spark SQL 1.3 to 1.4** 66](#_Toc469303790)

[**Upgrading from Spark SQL 1.0-1.2 to 1.3** 67](#_Toc469303791)

[**Compatibility with Apache Hive** 68](#_Toc469303792)

[**Supported Hive Features(Spark SQL支持的Hive特性)** 68](#_Toc469303793)

[**Unsupported Hive Functionality** 70](#_Toc469303794)

[**Reference** 71](#_Toc469303795)

[**Data Types** 71](#_Toc469303796)

[**NaN Semantics** 74](#_Toc469303797)

**Overview**

Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally, Spark SQL uses this extra information to perform extra optimizations. There are several ways to interact with Spark SQL including SQL and the Dataset API. When computing a result the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation.

Spark SQL是Spark的一个组件，用于结构化数据的计算。与基础的Spark RDD应用程序接口不同，Spark SQL提供的接口在数据结构和计算结构的执行中为Spark提供了更多的信息。在内部，Spark SQL使用这些额外的信息进行额外的优化设计。与Spark SQL（包括SQL和数据集API）交互有几种方式，当计算同一个结果时，同样的执行引擎将被调用，和你用于表达计算的API/语言相独立。这种统一意味着开发人员可以很方便地在对给定变换选择最自然方式的不同的API之间来回切换。

All of the examples on this page use sample data included in the Spark distribution and can be run in the spark-shell, pyspark shell, or sparkRshell.

本页使用的所有实例数据都包含在Spark分类中，而且可以在spark-shell、pyshark命令行或者sparkRshell中运行。

**SQL**

One use of Spark SQL is to execute SQL queries. Spark SQL can also be used to read data from an existing Hive installation. For more on how to configure this feature, please refer to the [Hive Tables](http://spark.apache.org/docs/latest/sql-programming-guide.html#hive-tables) section. When running SQL from within another programming language the results will be returned as a [Dataset/DataFrame](http://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes). You can also interact with the SQL interface using the [command-line](http://spark.apache.org/docs/latest/sql-programming-guide.html#running-the-spark-sql-cli) or over [JDBC/ODBC](http://spark.apache.org/docs/latest/sql-programming-guide.html#running-the-thrift-jdbcodbc-server).

Spark SQL的一个用途是执行SQL查询。Spark SQL也可以用于从既有安装的Hive中读取数据。有关如何配置这一特性的更多信息，请参考[Hive Tables](http://spark.apache.org/docs/latest/sql-programming-guide.html#hive-tables)章节。在另一种编程语言中运行SQL将作为一个Dataset或者DataFrame返回。你也可以使用[命令行](http://spark.apache.org/docs/latest/sql-programming-guide.html#running-the-spark-sql-cli)或者通过[JDBC/ODBC](http://spark.apache.org/docs/latest/sql-programming-guide.html#running-the-thrift-jdbcodbc-server)与SQL接口交互。

**Datasets and DataFrames**

A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL’s optimized execution engine. A Dataset can be [constructed](http://spark.apache.org/docs/latest/sql-programming-guide.html#creating-datasets) from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). The Dataset API is available in [Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset) and [Java](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html). Python does not have the support for the Dataset API. But due to Python’s dynamic nature, many of the benefits of the Dataset API are already available (i.e. you can access the field of a row by name naturally row.columnName). The case for R is similar.

Dataset是一个分布式的数据集合。Dataset是在Spark1.0中新添加的一个接口，它在RDD（强类型化，强大的lambda函数应用能力）的优势基础上提供了Spark SQL最优化执行引擎的优势。一个Dataset可以从JVM对象开始构造，然后使用函数转换（map，flatMap，filter等等）进行操作。Dataset API在Scala和Java中均可用。Python不支持Dataset API。但是由于Python的动态性，很多Dataset API的优势都已经可用（比如你可以很轻而易举的通过name访问row的一个字段）。R语言的情况类似。

A DataFrame is a *Dataset* 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](http://spark.apache.org/docs/latest/sql-programming-guide.html#data-sources) such as: structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, [Python](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame), and [R](http://spark.apache.org/docs/latest/api/R/index.html). In Scala and Java, a DataFrame is represented by a Dataset of Rows. In [the Scala API](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset), DataFrame is simply a type alias of Dataset[Row]. While, in [Java API](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html), users need to use Dataset<Row> to represent a DataFrame.

DataFrame是一个以命名列方式整合的数据集合。从概念上讲它与关系型数据库中的一个表或者R/Python中的一个数据框架类似，但是后台拥有更丰富的优化。DataFrames可以通过多种数据构造，例如：结构化的数据文件、hive中的表、外部数据库、Spark计算过程中生成的RDD等。DataFrame的API支持4种语言：Scala、Java、Python、R。

Throughout this document, we will often refer to Scala/Java Datasets of Rows as DataFrames.

本文档中我们将引用Rpw的Scala/Java数据集作为DataFrame。

**Getting Started**

**Starting Point: SparkSession**

The entry point into all functionality in Spark is the [SparkSession](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.SparkSession) class. To create a basic SparkSession, just use SparkSession.builder():

所有Spark功能的入口点是SparkSession类。要创建一个基本的SparkSession对象，使用SparkSession.builder()方法。

Scala:

**import** **org.apache.spark.sql.SparkSession**

**val** spark **=** **SparkSession**

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

.getOrCreate()

*// For implicit conversions like converting RDDs to DataFrames*

**import** **spark.implicits.\_**

Python:

**from** **pyspark.sql** **import** SparkSession

spark = SparkSession \

.builder \

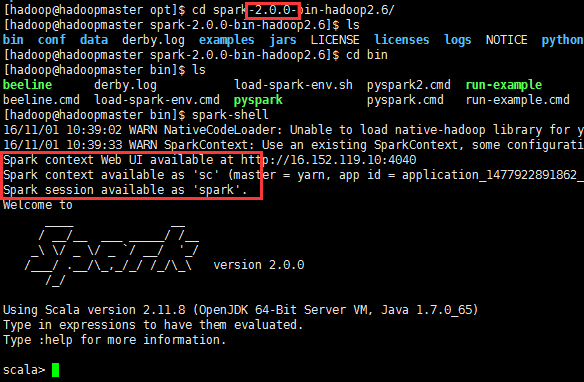
.appName("Python Spark SQL basic example") \

.config("spark.some.config.option", "some-value") \

.getOrCreate()

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

SparkSession in Spark 2.0 provides builtin support for Hive features including the ability to write queries using HiveQL, access to Hive UDFs, and the ability to read data from Hive tables. To use these features, you do not need to have an existing Hive setup.



Spark 2.0中的SparkSession类内置了对Hive特性（包括使用HiveQL编写查询的能力）提供的支持。

**Starting Point: SQLContext（old version）**

Spark SQL程序的主入口是SQLContext类或它的子类。创建一个基本的SQLContext，你只需要SparkContext，创建代码示例如下：

* **Scala**

val sc: SparkContext // An existing SparkContext.

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

* **Java**

JavaSparkContext sc = ...; // An existing JavaSparkContext.

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

除了基本的SQLContext，也可以创建HiveContext。SQLContext和HiveContext区别与联系为：

* SQLContext现在只支持SQL语法解析器（SQL-92语法）
* HiveContext现在支持SQL语法解析器和HiveSQL语法解析器，默认为HiveSQL语法解析器，用户可以通过配置切换成SQL语法解析器，来运行HiveSQL不支持的语法。
* 使用HiveContext可以使用Hive的UDF，读写Hive表数据等Hive操作。SQLContext不可以对Hive进行操作。
* Spark SQL未来的版本会不断丰富SQLContext的功能，做到SQLContext和HiveContext的功能容和，最终可能两者会统一成一个Context

HiveContext包装了Hive的依赖包，把HiveContext单独拿出来，可以在部署基本的Spark的时候就不需要Hive的依赖包，需要使用HiveContext时再把Hive的各种依赖包加进来。

SQL的解析器可以通过配置spark.sql.dialect参数进行配置。在SQLContext中只能使用Spark SQL提供的”sql“解析器。在HiveContext中默认解析器为”hiveql“，也支持”sql“解析器。

**Creating DataFrames**

With a SparkSession, applications can create DataFrames from an [existing RDD](http://spark.apache.org/docs/latest/sql-programming-guide.html#interoperating-with-rdds), from a Hive table, or from [Spark data sources](http://spark.apache.org/docs/latest/sql-programming-guide.html#data-sources).

使用SparkSession，应用程序可以从已有的RDD、Hive表或者Spark数据源创建DataFrame。

As an example, the following creates a DataFrame based on the content of a JSON file:

下面是基于JSON文件创建DataFrame的示例：

Scala:

**val** df **=** spark.read.json("examples/src/main/resources/people.json")

*// Displays the content of the DataFrame to stdout*

df.show()

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

*// | 19| Justin|*

*// +----+-------+*

Python:

*# spark is an existing SparkSession*

df = spark.read.json("examples/src/main/resources/people.json")

*# Displays the content of the DataFrame to stdout*

df.show()

*# +----+-------+*

*# | age| name|*

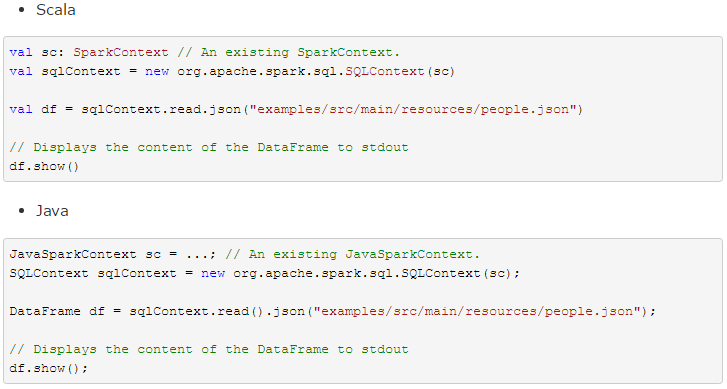
*# +----+-------+*

*# |null|Michael|*

*# | 30| Andy|*

*# | 19| Justin|*

*# +----+-------+*



Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

**Untyped Dataset Operations (aka DataFrame Operations)**

DataFrames provide a domain-specific language for structured data manipulation in [Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset), [Java](http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/Dataset.html), [Python](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame) and [R](http://spark.apache.org/docs/latest/api/R/DataFrame.html).

DataFrame在Scala、Java、Python和R语言中为结构化数据操作提供了一个领域特定语言。

As mentioned above, in Spark 2.0, DataFrames are just Dataset of Rows in Scala and Java API. These operations are also referred as “untyped transformations” in contrast to “typed transformations” come with strongly typed Scala/Java Datasets.

如上所述，在Spark 2.0中，DataFrame在Scala和Java API中仅仅是RowS的数据集合。这些操作也可以被称为”无类型转换“与Scala/Java数据集中的”类型转换“形成对照。

Here we include some basic examples of structured data processing using Datasets:

下面是几个操作示例：

Scala:

*// This import is needed to use the $-notation*

**import** **spark.implicits.\_**

*// Print the schema in a tree format*

df.printSchema()

*// root*

*// |-- age: long (nullable = true)*

*// |-- name: string (nullable = true)*

*// Select only the "name" column*

df.select("name").show()

*// +-------+*

*// | name|*

*// +-------+*

*// |Michael|*

*// | Andy|*

*// | Justin|*

*// +-------+*

*// Select everybody, but increment the age by 1*

df.select($"name", $"age" + 1).show()

*// +-------+---------+*

*// | name|(age + 1)|*

*// +-------+---------+*

*// |Michael| null|*

*// | Andy| 31|*

*// | Justin| 20|*

*// +-------+---------+*

*// Select people older than 21*

df.filter($"age" > 21).show()

*// +---+----+*

*// |age|name|*

*// +---+----+*

*// | 30|Andy|*

*// +---+----+*

*// Count people by age*

df.groupBy("age").count().show()

*// +----+-----+*

*// | age|count|*

*// +----+-----+*

*// | 19| 1|*

*// |null| 1|*

*// | 30| 1|*

*// +----+-----+*

Python:

*# spark, df are from the previous example*

*# Print the schema in a tree format*

df.printSchema()

*# root*

*# |-- age: long (nullable = true)*

*# |-- name: string (nullable = true)*

*# Select only the "name" column*

df.select("name").show()

*# +-------+*

*# | name|*

*# +-------+*

*# |Michael|*

*# | Andy|*

*# | Justin|*

*# +-------+*

*# Select everybody, but increment the age by 1*

df.select(df['name'], df['age'] + 1).show()

*# +-------+---------+*

*# | name|(age + 1)|*

*# +-------+---------+*

*# |Michael| null|*

*# | Andy| 31|*

*# | Justin| 20|*

*# +-------+---------+*

*# Select people older than 21*

df.filter(df['age'] > 21).show()

*# +---+----+*

*# |age|name|*

*# +---+----+*

*# | 30|Andy|*

*# +---+----+*

*# Count people by age*

df.groupBy("age").count().show()

*# +----+-----+*

*# | age|count|*

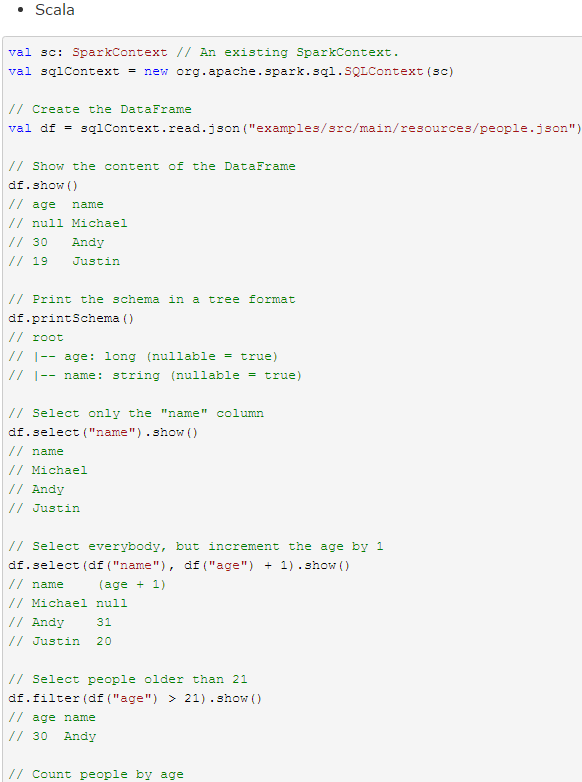
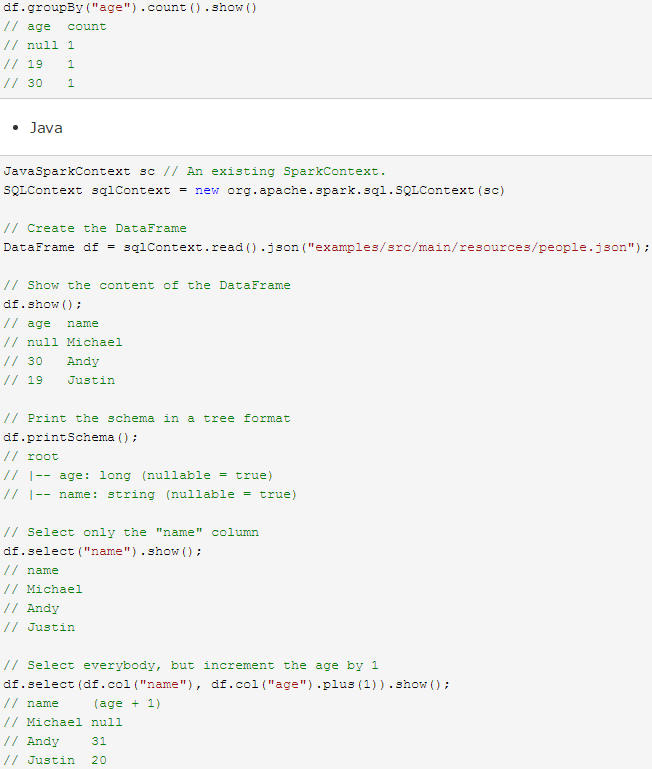
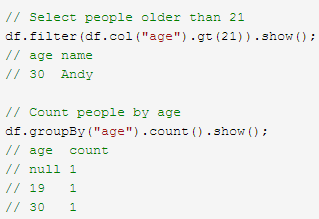
*# +----+-----+*

*# | 19| 1|*

*# |null| 1|*

*# | 30| 1|*

*# +----+-----+*

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

For a complete list of the types of operations that can be performed on a Dataset refer to the [API Documentation](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset).

In addition to simple column references and expressions, Datasets also have a rich library of functions including string manipulation, date arithmetic, common math operations and more. The complete list is available in the [DataFrame Function Reference](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$).

除了简单列引用和表达式，DataFrames还有丰富的library，功能包括string操作、date操作、常见数学操作等。详细内容请参考 [DataFrame Function Reference](http://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/functions.html)。

**Running SQL Queries Programmatically**

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

Spark Application可以使用SparkSession的sql()方法执行SQL查询操作，sql()方法返回的查询结果为DataFrame格式。代码如下：

Scala:

*// Register the DataFrame as a SQL temporary view*

df.createOrReplaceTempView("people")

**val** sqlDF **=** spark.sql("SELECT \* FROM people")

sqlDF.show()

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

*// | 19| Justin|*

*// +----+-------+*

Python:

*# Register the DataFrame as a SQL temporary view*

df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT \* FROM people")

sqlDF.show()

*# +----+-------+*

*# | age| name|*

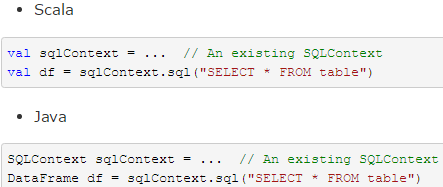
*# +----+-------+*

*# |null|Michael|*

*# | 30| Andy|*

*# | 19| Justin|*

*# +----+-------+*



Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

**Creating Datasets**

Datasets are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized [Encoder](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Encoder) to serialize the objects for processing or transmitting over the network. While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.

数据集与RDD类似，然而，他们使用一个专门的编码器（Encoder）将对象序列化从而在网络上处理或者传送，而不用Java的序列化或者Kryo。然而两种编码器和标准系列化都可以将一个对象转化为字节，编码器是动态生成的代码，同时使用的是可以让Spark在不反序列化字节为对象的前提下执行像过滤，排序和哈希操作的格式。

Scala:

*// 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**:** Long)

*// Encoders are created for case classes*

**val** caseClassDS **=** **Seq**(**Person**("Andy", 32)).toDS()

caseClassDS.show()

*// +----+---+*

*// |name|age|*

*// +----+---+*

*// |Andy| 32|*

*// +----+---+*

*// Encoders for most common types are automatically provided by importing spark.implicits.\_*

**val** primitiveDS **=** **Seq**(1, 2, 3).toDS()

primitiveDS.map(**\_** + 1).collect() *// Returns: Array(2, 3, 4)*

*// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name*

**val** path **=** "examples/src/main/resources/people.json"

**val** peopleDS **=** spark.read.json(path).as[Person]

peopleDS.show()

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

*// | 19| Justin|*

*// +----+-------+*

Java:

**import** **java.util.Arrays**;

**import** **java.util.Collections**;

**import** **java.io.Serializable**;

**import** **org.apache.spark.api.java.function.MapFunction**;

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

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

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

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

**public** **static** **class** **Person** **implements** Serializable {

**private** String name;

**private** int age;

**public** String getName() {

**return** name;

}

**public** void setName(String name) {

**this**.name = name;

}

**public** int getAge() {

**return** age;

}

**public** void setAge(int age) {

**this**.age = age;

}

}

*// Create an instance of a Bean class*

Person person = **new** Person();

person.setName("Andy");

person.setAge(32);

*// Encoders are created for Java beans*

Encoder<Person> personEncoder = Encoders.bean(Person.class);

Dataset<Person> javaBeanDS = spark.createDataset(

Collections.singletonList(person),

personEncoder

);

javaBeanDS.show();

*// +---+----+*

*// |age|name|*

*// +---+----+*

*// | 32|Andy|*

*// +---+----+*

*// Encoders for most common types are provided in class Encoders*

Encoder<Integer> integerEncoder = Encoders.INT();

Dataset<Integer> primitiveDS = spark.createDataset(Arrays.asList(1, 2, 3), integerEncoder);

Dataset<Integer> transformedDS = primitiveDS.map(**new** MapFunction<Integer, Integer>() {

**@Override**

**public** Integer call(Integer value) **throws** Exception {

**return** value + 1;

}

}, integerEncoder);

transformedDS.collect(); *// Returns [2, 3, 4]*

*// DataFrames can be converted to a Dataset by providing a class. Mapping based on name*

String path = "examples/src/main/resources/people.json";

Dataset<Person> peopleDS = spark.read().json(path).as(personEncoder);

peopleDS.show();

*// +----+-------+*

*// | age| name|*

*// +----+-------+*

*// |null|Michael|*

*// | 30| Andy|*

*// | 19| Justin|*

*// +----+-------+*

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

**Interoperating with RDDs （DataFrames与RDDs的相互转换）**

Spark SQL supports two different methods for converting existing RDDs into Datasets. The first method uses reflection to infer the schema of an RDD that contains specific types of objects. This reflection based approach leads to more concise code and works well when you already know the schema while writing your Spark application.

The second method for creating Datasets is through a programmatic interface that allows you to construct a schema and then apply it to an existing RDD. While this method is more verbose, it allows you to construct Datasets when the columns and their types are not known until runtime.

Spark SQL支持两种RDDs转换为Datasets的方式：

* 使用反射获取RDD内的Schema
  + 当已知类的Schema的时候，使用这种基于反射的方法会让代码更加简洁而且效果也很好。
  + 通过编程接口指定Schema
* 通过Spark SQL的接口创建RDD的Schema，这种方式会让代码比较冗长。
  + 这种方法的好处是，在运行时才知道数据的列以及列的类型的情况下，可以动态生成Schema

**Inferring the Schema Using Reflection**

The Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame. The case class defines the schema of the table. The names of the arguments to the case class are read using reflection and become the names of the columns. Case classes can also be nested or contain complex types such as Seqs or Arrays. This RDD can be implicitly converted to a DataFrame and then be registered as a table. Tables can be used in subsequent SQL statements.

Scala的Spark SQL接口支持将包含case class的RDD自动转换成DataFrame。case class定义表的schema，通过反射读取case class的参数名成为列名。case class也可以是嵌套的或包含如Seq或者Array等复杂类型的。这个RDD可以隐性转换为一个DataFrame，从而注册为一个表。表可在随后的SQL语句中使用。

Scala：

**import** **org.apache.spark.sql.catalyst.encoders.ExpressionEncoder**

**import** **org.apache.spark.sql.Encoder**

*// For implicit conversions from RDDs to DataFrames*

**import** **spark.implicits.\_**

*// Create an RDD of Person objects from a text file, convert it to a Dataframe*

**val** peopleDF **=** spark.sparkContext

.textFile("examples/src/main/resources/people.txt")

.map(**\_**.split(","))

.map(attributes **=>** **Person**(attributes(0), attributes(1).trim.toInt))

.toDF()

*// Register the DataFrame as a temporary view*

peopleDF.createOrReplaceTempView("people")

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

**val** teenagersDF **=** spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")

*// The columns of a row in the result can be accessed by field index*

teenagersDF.map(teenager **=>** "Name: " + teenager(0)).show()

*// +------------+*

*// | value|*

*// +------------+*

*// |Name: Justin|*

*// +------------+*

*// or by field name*

teenagersDF.map(teenager **=>** "Name: " + teenager.getAs[String]("name")).show()

*// +------------+*

*// | value|*

*// +------------+*

*// |Name: Justin|*

*// +------------+*

*// No pre-defined encoders for Dataset[Map[K,V]], define explicitly*

**implicit** **val** mapEncoder **=** org.apache.spark.sql.**Encoders**.kryo[Map[String, Any]]

*// Primitive types and case classes can be also defined as*

*// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()*

*// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]*

teenagersDF.map(teenager **=>** teenager.getValuesMap[Any](**List**("name", "age"))).collect()

*// Array(Map("name" -> "Justin", "age" -> 19))*

Python：

**from** **pyspark.sql** **import** Row

sc = spark.sparkContext

*# Load a text file and convert each line to a Row.*

lines = sc.textFile("examples/src/main/resources/people.txt")

parts = lines.map(**lambda** l: l.split(","))

people = parts.map(**lambda** p: Row(name=p[0], age=int(p[1])))

*# Infer the schema, and register the DataFrame as a table.*

schemaPeople = spark.createDataFrame(people)

schemaPeople.createOrReplaceTempView("people")

*# SQL can be run over DataFrames that have been registered as a table.*

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

*# The results of SQL queries are Dataframe objects.*

*# rdd returns the content as an :class:`pyspark.RDD` of :class:`Row`.*

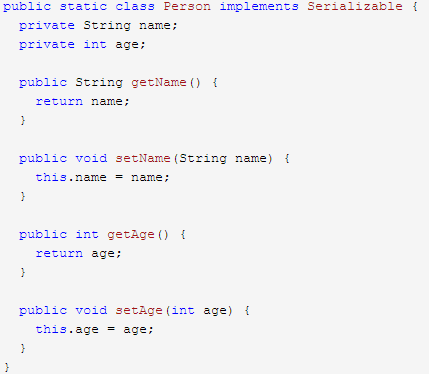
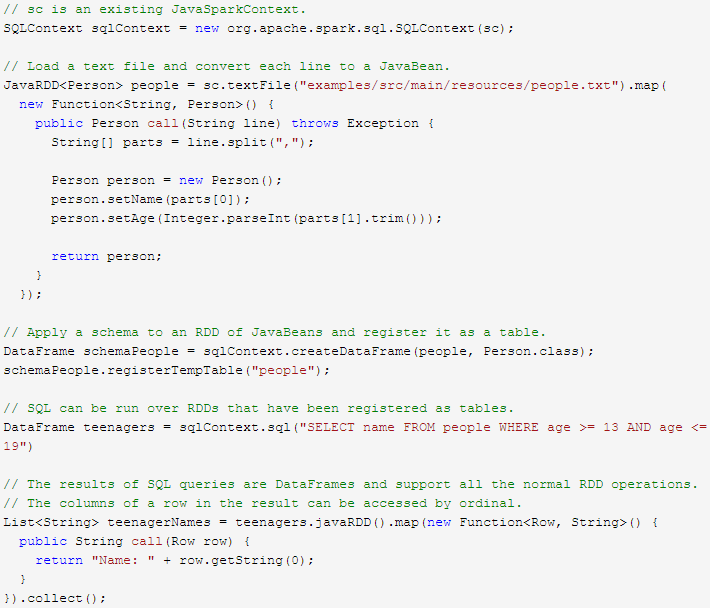
teenNames = teenagers.rdd.map(**lambda** p: "Name: " + p.name).collect()

**for** name **in** teenNames:

**print**(name)

*# Name: Justin*

Spark SQL支持将JavaBean的RDD自动转换成DataFrame。通过反射获取Bean的基本信息，依据Bean的信息定义Schema。当前Spark SQL版本（Spark 1.5.2）不支持嵌套的JavaBeans和复杂数据类型（如：List、Array）。创建一个实现Serializable接口包含所有属性getters和setters的类来创建一个JavaBean。通过调用createDataFrame并提供JavaBean的Class object，指定一个Schema给一个RDD。示例如下：

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

**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 SparkSession.

For example:

当case class不能被预先定义的时候（比如，记录的结构在string中编写，或者一个文本数据集将被解析且字段对不同的用户进行不同的解析），编程创建DataFrame分为三步：

* 从原来的RDD创建一个Row格式的RDD
* 创建与RDD中Rows结构匹配的StructType，通过该StructType创建表示RDD的Schema
* 通过SparkSession提供的createDataFrame方法创建DataFrame，

示例如下：

Scala：

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

*// Create an RDD*

**val** peopleRDD **=** spark.sparkContext.textFile("examples/src/main/resources/people.txt")

*// The schema is encoded in a string*

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

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

**val** fields **=** schemaString.split(" ")

.map(fieldName **=>** **StructField**(fieldName, **StringType**, nullable **=** **true**))

**val** schema **=** **StructType**(fields)

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

**val** rowRDD **=** peopleRDD

.map(**\_**.split(","))

.map(attributes **=>** **Row**(attributes(0), attributes(1).trim))

*// Apply the schema to the RDD*

**val** peopleDF **=** spark.createDataFrame(rowRDD, schema)

*// Creates a temporary view using the DataFrame*

peopleDF.createOrReplaceTempView("people")

*// SQL can be run over a temporary view created using DataFrames*

**val** results **=** spark.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 field index or by field name*

results.map(attributes **=>** "Name: " + attributes(0)).show()

*// +-------------+*

*// | value|*

*// +-------------+*

*// |Name: Michael|*

*// | Name: Andy|*

*// | Name: Justin|*

*// +-------------+*

Python：

*# Import data types*

**from** **pyspark.sql.types** **import** \*

sc = spark.sparkContext

*# Load a text file and convert each line to a Row.*

lines = sc.textFile("examples/src/main/resources/people.txt")

parts = lines.map(**lambda** l: l.split(","))

*# Each line is converted to a tuple.*

people = parts.map(**lambda** p: (p[0], p[1].strip()))

*# The schema is encoded in a string.*

schemaString = "name age"

fields = [StructField(field\_name, StringType(), True) **for** field\_name **in** schemaString.split()]

schema = StructType(fields)

*# Apply the schema to the RDD.*

schemaPeople = spark.createDataFrame(people, schema)

*# Creates a temporary view using the DataFrame*

schemaPeople.createOrReplaceTempView("people")

*# Creates a temporary view using the DataFrame*

schemaPeople.createOrReplaceTempView("people")

*# SQL can be run over DataFrames that have been registered as a table.*

results = spark.sql("SELECT name FROM people")

results.show()

*# +-------+*

*# | name|*

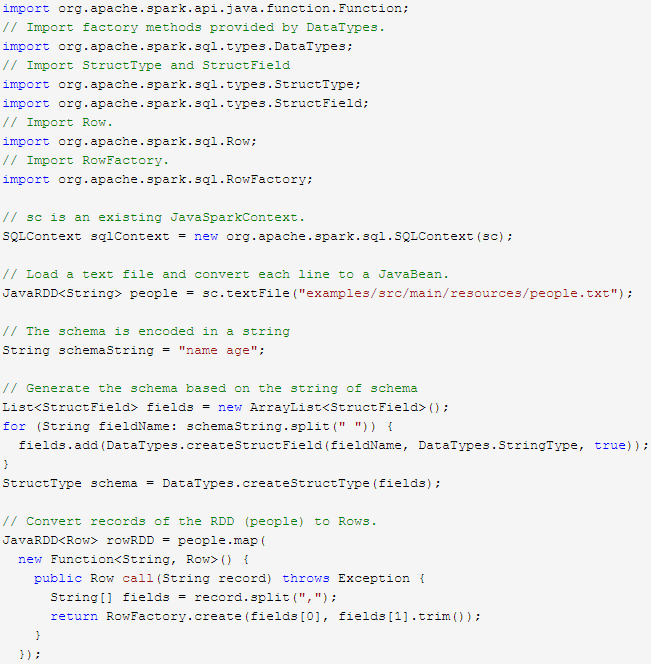
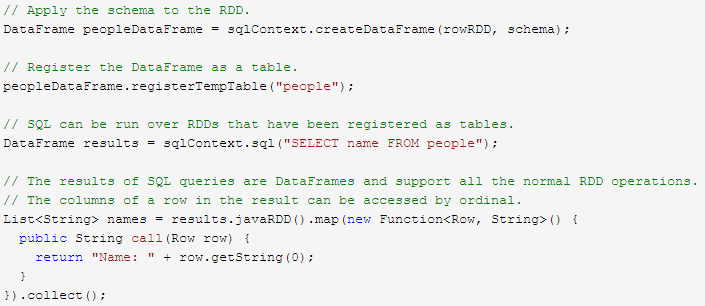
*# +-------+*

*# |Michael|*

*# | Andy|*

*# | Justin|*

*# +-------+*

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

**Data Sources**

Spark SQL supports operating on a variety of data sources through the DataFrame interface. A DataFrame can be operated on using relational transformations and can also be used to create a temporary view. Registering a DataFrame as a temporary view allows you to run SQL queries over its data. This section describes the general methods for loading and saving data using the Spark Data Sources and then goes into specific options that are available for the built-in data sources.

Spark SQL的DataFrame接口支持多种数据源的操作。一个DataFrame可以进行关系型转换方式的操作，也可以被注册为临时表。把DataFrame注册为临时表之后，就可以对该DataFrame的数据执行SQL查询。Data Sources这部分首先描述了对Spark的数据源执行加载和保存的常用方法，然后对内置数据源进行深入介绍。

**Generic Load/Save Functions**

In the simplest form, the default data source (parquet unless otherwise configured by spark.sql.sources.default) will be used for all operations.

Spark SQL的默认数据源为Parquet格式。数据源为Parquet文件时，Spark SQL可以方便的执行所有的操作。修改配置项spark.sql.sources.default，可修改默认数据源格式。读取Parquet文件示例如下：

Scala:

**val** usersDF **=** spark.read.load("examples/src/main/resources/users.parquet")

usersDF.select("name", "favorite\_color").write.save("namesAndFavColors.parquet")

Python:

df = spark.read.load("examples/src/main/resources/users.parquet")

df.select("name", "favorite\_color").write.save("namesAndFavColors.parquet")

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

**Manually Specifying Options**

You can also manually specify the data source that will be used along with any extra options that you would like to pass to the data source. Data sources are specified by their fully qualified name (i.e., org.apache.spark.sql.parquet), but for built-in sources you can also use their short names (json, parquet, jdbc). DataFrames loaded from any data source type can be converted into other types using this syntax.

当数据源格式不是parquet格式文件时，需要手动指定数据源的格式。数据源格式需要指定全名（例如：org.apache.spark.sql.parquet），如果数据源格式为内置格式，则只需要指定简称（json,parquet,jdbc）。通过指定的数据源格式名，可以对DataFrames进行类型转换操作。示例如下：

Scala:

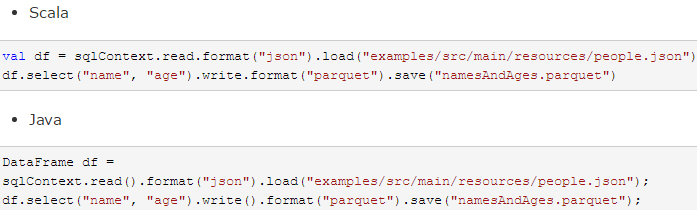
**val** peopleDF **=** spark.read.format("json").load("examples/src/main/resources/people.json")

peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet")

Python:

df = spark.read.load("examples/src/main/resources/people.json", format="json")

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



Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

**Run SQL on files directly**

Instead of using read API to load a file into DataFrame and query it, you can also query that file directly with SQL.

除了使用read API将文件加载到DataFrame中去执行，也可以直接使用SQL运行该文件。

Scala:

**val** sqlDF **=** spark.sql("SELECT \* FROM parquet.`examples/src/main/resources/users.parquet`")

Python:

df = spark.sql("SELECT \* FROM parquet.`examples/src/main/resources/users.parquet`")

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

**Save Modes**

Save operations can optionally take a SaveMode, that specifies how to handle existing data if present. It is important to realize that these save modes do not utilize any locking and are not atomic. Additionally, when performing an Overwrite, the data will be deleted before writing out the new data.

可以采用SaveMode执行存储操作，SaveMode定义了对数据的处理模式。需要注意的是，这些保存模式不使用任何锁定，不是原子操作。此外，当使用Overwrite方式执行时，在输出新数据之前原数据就已经被删除。SaveMode详细介绍如下表：

|  |  |  |
| --- | --- | --- |
| **Scala/Java** | **Any Language** | **Meaning** |
| SaveMode.ErrorIfExists(default) | "error"(default) | When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.  当保存DataFrame到数据源中时，如果数据已经存在，则抛出异常。 |
| SaveMode.Append | "append" | When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data. |
| SaveMode.Overwrite | "overwrite" | Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame. |
| SaveMode.Ignore | "ignore" | Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data. This is similar to a CREATE TABLE IF NOT EXISTS in SQL. |

**Saving to Persistent Tables**

DataFrames can also be saved as persistent tables into Hive metastore using the saveAsTable command. Notice existing Hive deployment is not necessary to use this feature. Spark will create a default local Hive metastore (using Derby) for you. Unlike the createOrReplaceTempViewcommand, saveAsTable will materialize the contents of the DataFrame and create a pointer to the data in the Hive metastore. Persistent tables will still exist even after your Spark program has restarted, as long as you maintain your connection to the same metastore. A DataFrame for a persistent table can be created by calling the table method on a SparkSession with the name of the table.

By default saveAsTable will create a “managed table”, meaning that the location of the data will be controlled by the metastore. Managed tables will also have their data deleted automatically when a table is dropped.

可以通过saveAsTable方法将DataFrames作为持久表保存到Hive metastore中。注意如果存在Hive部署环境，则不用使用该特性。与createOrReplaceTempView方法不同的是，saveAsTable将DataFrame中的内容持久化到表中，并在HiveMetastore中创建一个数据的指针。只要保持对同一metastore的连接，持久表甚至可以在你Spark程序重启后仍存在。持久表的DataFrame可以用通过表名调用SparkSession的table方法来创建。

默认的saveAsTable方法将创建一个“managed table”，表示数据的位置可以通过metastore获得。当存储数据的表被删除时，managed table也将自动删除。

**Parquet Files**

[Parquet](http://parquet.io/) is a columnar format that is supported by many other data processing systems. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data. When writing Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

Parquet是一种支持多种数据处理系统的柱状的数据格式，Parquet文件中保留了原始数据的模式。Spark SQL提供了Parquet文件的读写功能。写入Parquet文件时，所有列将自动转换为可空类型以适应兼容性。

**Loading Data Programmatically**

Using the data from the above example:

Scala:

*// Encoders for most common types are automatically provided by importing spark.implicits.\_*

**import** **spark.implicits.\_**

**val** peopleDF **=** spark.read.json("examples/src/main/resources/people.json")

*// DataFrames can be saved as Parquet files, maintaining the schema information*

peopleDF.write.parquet("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** parquetFileDF **=** spark.read.parquet("people.parquet")

*// Parquet files can also be used to create a temporary view and then used in SQL statements*

parquetFileDF.createOrReplaceTempView("parquetFile")

**val** namesDF **=** spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19")

namesDF.map(attributes **=>** "Name: " + attributes(0)).show()

*// +------------+*

*// | value|*

*// +------------+*

*// |Name: Justin|*

*// +------------+*

Python:

peopleDF = spark.read.json("examples/src/main/resources/people.json")

*# DataFrames can be saved as Parquet files, maintaining the schema information.*

peopleDF.write.parquet("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.*

parquetFile = spark.read.parquet("people.parquet")

*# Parquet files can also be used to create a temporary view and then used in SQL statements.*

parquetFile.createOrReplaceTempView("parquetFile")

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

teenagers.show()

*# +------+*

*# | name|*

*# +------+*

*# |Justin|*

*# +------+*

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

**Partition Discovery**

Table partitioning is a common optimization approach used in systems like Hive. In a partitioned table, data are usually stored in different directories, with partitioning column values encoded in the path of each partition directory. The Parquet data source is now able to discover and infer partitioning information automatically. For example, we can store all our previously used population data into a partitioned table using the following directory structure, with two extra columns, gender and country as partitioning columns:

对表进行分区是对数据进行优化的方式之一。在分区的表内，数据通过分区列将数据存储在不同的目录下。Parquet数据源现在能够自动发现并解析分区信息。例如，对人口数据进行分区存储，分区列为gender和country，使用下面的目录结构：

path

└── to

└── table

├── gender=male

│   ├── ...

│   │

│   ├── country=US

│   │   └── data.parquet

│   ├── country=CN

│   │   └── data.parquet

│   └── ...

└── gender=female

   ├── ...

   │

   ├── country=US

   │   └── data.parquet

   ├── country=CN

   │   └── data.parquet

   └── ...

By passing path/to/table to either SparkSession.read.parquet or SparkSession.read.load, Spark SQL will automatically extract the partitioning information from the paths. Now the schema of the returned DataFrame becomes:

通过传递path/to/table给 SQLContext.read.parquet或SQLContext.read.load，Spark SQL将自动解析分区信息。返回的DataFrame的Schema如下：

root

|-- name: string (nullable = true)

|-- age: long (nullable = true)

|-- gender: string (nullable = true)

|-- country: string (nullable = true)

Notice that the data types of the partitioning columns are automatically inferred. Currently, numeric data types and string type are supported. Sometimes users may not want to automatically infer the data types of the partitioning columns. For these use cases, the automatic type inference can be configured by spark.sql.sources.partitionColumnTypeInference.enabled, which is default to true. When type inference is disabled, string type will be used for the partitioning columns.

需要注意的是，数据的分区列的数据类型是自动解析的。当前，支持数值类型和字符串类型。有时用户并不希望分区列的数据类型被自动解析。基于这些情况，自动解析分区类型可以通过spark.sql.sources.partitionColumnTypeInference.enabled进行配置，它的默认值为true。如果想关闭该功能，直接将该参数设置为disabled。此时，分区列数据格式将被默认设置为string类型，不再进行类型解析。

Starting from Spark 1.6.0, partition discovery only finds partitions under the given paths by default. For the above example, if users pass path/to/table/gender=male to either SparkSession.read.parquet or SparkSession.read.load, gender will not be considered as a partitioning column. If users need to specify the base path that partition discovery should start with, they can set basePath in the data source options. For example, when path/to/table/gender=male is the path of the data and users set basePath to path/to/table/, gender will be a partitioning column.

从Spark1.6.0版本开始，分区解析默认只查找指定路径下的分区信息。上述示例中，如果用户把path/to/table/gender=male值传递给SparkSession.read.parquet或者SparkSession.read.load，gender字段将不会被视为一个分区列。通过在数据源选项中设置basePath，用户可以指定分区解析的起始路径。比如，当数据路径为path/to/table/gender=male且用户设置basePath为path/to/table/时，gender将被视为一个分区列。

**Schema Merging**

Like ProtocolBuffer, Avro, and Thrift, Parquet also supports schema evolution. Users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. The Parquet data source is now able to automatically detect this case and merge schemas of all these files.

像ProtocolBuffer、Avro和Thrift那样，Parquet也支持Schema evolution（Schema演变）。用户可以先定义一个简单的Schema，然后逐渐的向Schema中增加列描述。通过这种方式，用户可以获取多个有不同Schema但相互兼容的Parquet文件。现在Parquet数据源能自动检测这种情况，并合并这些文件的schema。

Since schema merging is a relatively expensive operation, and is not a necessity in most cases, we turned it off by default starting from 1.5.0. You may enable it by

1. setting data source option mergeSchema to true when reading Parquet files (as shown in the examples below), or
2. setting the global SQL option spark.sql.parquet.mergeSchema to true.

因为Schema合并是一个高消耗的操作，在大多数情况下并不需要，所以Spark SQL从1.5.0开始默认关闭了该功能。可以通过下面两种方式开启该功能：

1. 当数据源为Parquet文件时，将数据源选项mergeSchema设置为true
2. 设置全局SQL选项spark.sql.parquet.mergeSchema为true

示例如下：

Scala：

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

**import** **spark.implicits.\_**

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

**val** squaresDF **=** spark.sparkContext.makeRDD(1 to 5).map(i **=>** (i, i \* i)).toDF("value", "square")

squaresDF.write.parquet("data/test\_table/key=1")

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

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

**val** cubesDF **=** spark.sparkContext.makeRDD(6 to 10).map(i **=>** (i, i \* i \* i)).toDF("value", "cube")

cubesDF.write.parquet("data/test\_table/key=2")

*// Read the partitioned table*

**val** mergedDF **=** spark.read.option("mergeSchema", "true").parquet("data/test\_table")

mergedDF.printSchema()

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

*// with the partitioning column appeared in the partition directory paths*

*// root*

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

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

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

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

Python：

**from** **pyspark.sql** **import** Row

*# spark is from the previous example.*

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

sc = spark.sparkContext

squaresDF = spark.createDataFrame(sc.parallelize(range(1, 6))

.map(**lambda** i: Row(single=i, double=i \*\* 2)))

squaresDF.write.parquet("data/test\_table/key=1")

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

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

cubesDF = spark.createDataFrame(sc.parallelize(range(6, 11))

.map(**lambda** i: Row(single=i, triple=i \*\* 3)))

cubesDF.write.parquet("data/test\_table/key=2")

*# Read the partitioned table*

mergedDF = spark.read.option("mergeSchema", "true").parquet("data/test\_table")

mergedDF.printSchema()

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

*# with the partitioning column appeared in the partition directory paths.*

*# root*

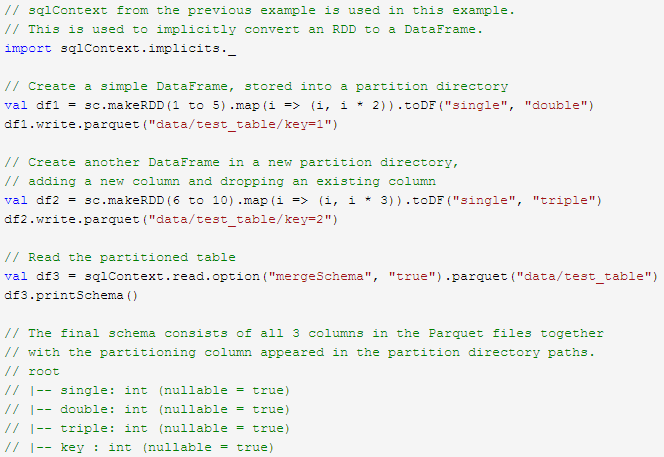
*# |-- double: long (nullable = true)*

*# |-- single: long (nullable = true)*

*# |-- triple: long (nullable = true)*

*# |-- key: integer (nullable = true)*

*Old Version：*



Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

**Hive metastore Parquet table conversion**

When reading from and writing to Hive metastore Parquet tables, Spark SQL will try to use its own Parquet support instead of Hive SerDe for better performance. This behavior is controlled by the spark.sql.hive.convertMetastoreParquet configuration, and is turned on by default.

当向Hive metastore中读写Parquet表时，Spark SQL将使用Spark SQL自带的Parquet SerDe（SerDe：Serialize/Deserilize的简称,目的是用于序列化和反序列化），而不是用Hive的SerDe，Spark SQL自带的SerDe拥有更好的性能。这个优化的配置参数为spark.sql.hive.convertMetastoreParquet，默认值为开启。

**Hive/Parquet Schema Reconciliation**

There are two key differences between Hive and Parquet from the perspective of table schema processing.

1. Hive is case insensitive, while Parquet is not
2. Hive considers all columns nullable, while nullability in Parquet is significant

从表Schema处理的角度对比Hive和Parquet，有两个区别：

* Hive区分大小写，Parquet不区分大小写
* hive允许所有的列为空，而Parquet不允许所有的列全为空

Due to this reason, we must reconcile Hive metastore schema with Parquet schema when converting a Hive metastore Parquet table to a Spark SQL Parquet table. The reconciliation rules are:

1. Fields that have the same name in both schema must have the same data type regardless of nullability. The reconciled field should have the data type of the Parquet side, so that nullability is respected.
2. The reconciled schema contains exactly those fields defined in Hive metastore schema.
   * Any fields that only appear in the Parquet schema are dropped in the reconciled schema.
   * Any fields that only appear in the Hive metastore schema are added as nullable field in the reconciled schema.

由于这两个区别，当将Hive metastore Parquet表转换为Spark SQL Parquet表时，需要将Hive metastore schema和Parquet schema进行一致化。一致化规则如下：

* 这两个schema中的同名字段必须具有相同的数据类型。一致化后的字段必须为Parquet的字段类型。这个规则同时也解决了空值的问题。
* 一致化后的schema只包含Hive metastore中出现的字段。
  + 忽略只出现在Parquet schema中的字段
  + 只在Hive metastore schema中出现的字段设为nullable字段，并加到一致化后的schema中

**Metadata Refreshing**

Spark SQL caches Parquet metadata for better performance. When Hive metastore Parquet table conversion is enabled, metadata of those converted tables are also cached. If these tables are updated by Hive or other external tools, you need to refresh them manually to ensure consistent metadata.

Spark SQL缓存了Parquet元数据以达到良好的性能。当Hive metastore Parquet表转换为enabled时，表修改后缓存的元数据并不能刷新。所以，当表被Hive或其它工具修改时，则必须手动刷新元数据，以保证元数据的一致性。示例如下：

Scala：

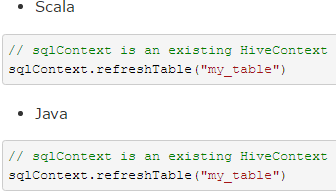
*// spark is an existing SparkSession*

spark.catalog.refreshTable("my\_table")

Python：

*# spark is an existing SparkSession*

spark.catalog.refreshTable("my\_table")



**Configuration**

Configuration of Parquet can be done using the setConf method on SparkSession or by running SET key=value commands using SQL.

配置Parquet可以使用SparkSession的setConf方法或使用SQL执行SET key=value命令。详细参数说明如下：

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.sql.parquet.binaryAsString | false | Some other Parquet-producing systems, in particular Impala, Hive, and older versions of Spark SQL, do not differentiate between binary data and strings when writing out the Parquet schema. This flag tells Spark SQL to interpret binary data as a string to provide compatibility with these systems. |
| spark.sql.parquet.int96AsTimestamp | true | Some Parquet-producing systems, in particular Impala and Hive, store Timestamp into INT96. This flag tells Spark SQL to interpret INT96 data as a timestamp to provide compatibility with these systems. |
| spark.sql.parquet.cacheMetadata | true | Turns on caching of Parquet schema metadata. Can speed up querying of static data. |
| spark.sql.parquet.compression.codec | snappy | Sets the compression codec use when writing Parquet files. Acceptable values include: uncompressed, snappy, gzip, lzo. |
| spark.sql.parquet.filterPushdown | true | Enables Parquet filter push-down optimization when set to true. |
| spark.sql.hive.convertMetastoreParquet | true | When set to false, Spark SQL will use the Hive SerDe for parquet tables instead of the built in support. |
| spark.sql.parquet.mergeSchema | false | When true, the Parquet data source merges schemas collected from all data files, otherwise the schema is picked from the summary file or a random data file if no summary file is available. |

**JSON Datasets**

Spark SQL can automatically infer the schema of a JSON dataset and load it as a Dataset[Row]. This conversion can be done using SparkSession.read.json() on either an RDD of String, or a JSON file.

Spark SQL能自动解析JSON数据集的Schema并读取JSON数据集为DataFrame格式。读取JSON数据集方法为SparkSession.read().json()。该方法将String格式的RDD或JSON文件转换为DataFrame。

Note that the file that is offered as *a json file* 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.

需要注意的是，这里的JSON文件不是常规的JSON格式。JSON文件每一行必须包含一个独立的、自满足有效的JSON对象。如果用多行描述一个JSON对象，会导致读取出错。读取JSON数据集示例如下：

Scala：

*// 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 **=** "examples/src/main/resources/people.json"

**val** peopleDF **=** spark.read.json(path)

*// The inferred schema can be visualized using the printSchema() method*

peopleDF.printSchema()

*// root*

*// |-- age: long (nullable = true)*

*// |-- name: string (nullable = true)*

*// Creates a temporary view using the DataFrame*

peopleDF.createOrReplaceTempView("people")

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

**val** teenagerNamesDF **=** spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")

teenagerNamesDF.show()

*// +------+*

*// | name|*

*// +------+*

*// |Justin|*

*// +------+*

*// Alternatively, a DataFrame can be created for a JSON dataset represented by*

*// an RDD[String] storing one JSON object per string*

**val** otherPeopleRDD **=** spark.sparkContext.makeRDD(

"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: **Nil**)

**val** otherPeople **=** spark.read.json(otherPeopleRDD)

otherPeople.show()

*// +---------------+----+*

*// | address|name|*

*// +---------------+----+*

*// |[Columbus,Ohio]| Yin|*

*// +---------------+----+*

Python：

*# spark is from the previous example.*

sc = spark.sparkContext

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

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

path = "examples/src/main/resources/people.json"

peopleDF = spark.read.json(path)

*# The inferred schema can be visualized using the printSchema() method*

peopleDF.printSchema()

*# root*

*# |-- age: long (nullable = true)*

*# |-- name: string (nullable = true)*

*# Creates a temporary view using the DataFrame*

peopleDF.createOrReplaceTempView("people")

*# SQL statements can be run by using the sql methods provided by spark*

teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")

teenagerNamesDF.show()

*# +------+*

*# | name|*

*# +------+*

*# |Justin|*

*# +------+*

*# Alternatively, a DataFrame can be created for a JSON dataset represented by*

*# an RDD[String] storing one JSON object per string*

jsonStrings = ['{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}']

otherPeopleRDD = sc.parallelize(jsonStrings)

otherPeople = spark.read.json(otherPeopleRDD)

otherPeople.show()

*# +---------------+----+*

*# | address|name|*

*# +---------------+----+*

*# |[Columbus,Ohio]| Yin|*

*# +---------------+----+*



Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

**Hive Tables**

Spark SQL also supports reading and writing data stored in [Apache Hive](http://hive.apache.org/). However, since Hive has a large number of dependencies, these dependencies are not included in the default Spark distribution. If Hive dependencies can be found on the classpath, Spark will load them automatically. Note that these Hive dependencies must also be present on all of the worker nodes, as they will need access to the Hive serialization and deserialization libraries (SerDes) in order to access data stored in Hive.

Spark SQL支持对Hive的读写操作。需要注意的是，Hive所依赖的包，没有包含在Spark assembly包中。如果classpath中添加了Hive依赖包，Spark将会自动加载它们。需要注意的是，这些Hive的assembly包必须添加到所有的worker节点上，因为worker节点在访问Hive中数据时，会调用Hive的 serialization and deserialization libraries（SerDes），此时将用到Hive的依赖包。

Configuration of Hive is done by placing your hive-site.xml, core-site.xml (for security configuration), and hdfs-site.xml (for HDFS configuration) file in conf/.

Hive的配置文件为conf/目录下的hive-site.xml, core-site.xml和hdfs-site.xml文件。

When working with Hive, one must instantiate SparkSession with Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions. Users who do not have an existing Hive deployment can still enable Hive support. When not configured by the hive-site.xml, the context automatically creates metastore\_db in the current directory and creates a directory configured by spark.sql.warehouse.dir, which defaults to the directory spark-warehouse in the current directory that the spark application is started. Note that the hive.metastore.warehouse.dir property in hive-site.xml is deprecated since Spark 2.0.0. Instead, use spark.sql.warehouse.dir to specify the default location of database in warehouse. You may need to grant write privilege to the user who starts the spark application.

操作Hive时，必须创建一个SparkSession对象，该对象包含了对MetaStore、Hive Serdes和Hive用户自定义函数的支持，使得用户在没有部署Hive的情况下仍然可以得到Hive的支持。如果没有配置hive-site.xml，上下文就会在当前目录自动创建metastore\_db，并通过spark.sql.warehouse.dir创建一个默认名为spark-warehouse目录，作为spark应用程序的启动目录。注意hive-site.xml中的hivemetastore.warehouse.dir属性自从Spark 2.0.0开始被弃用，取而代之的是spark.sql.warehouse.dir指定数据仓库中数据库的默认路径。对使用Spark程序的用户，可能需要赋予他们对该目录的写权限。

**Scala：**

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

**import** **org.apache.spark.sql.SparkSession**

**case** **class** **Record**(key**:** Int, value**:** String)

*// warehouseLocation points to the default location for managed databases and tables*

**val** warehouseLocation **=** "file:${system:user.dir}/spark-warehouse"

**val** spark **=** **SparkSession**

.builder()

.appName("Spark Hive Example")

.config("spark.sql.warehouse.dir", warehouseLocation)

.enableHiveSupport()

.getOrCreate()

**import** **spark.implicits.\_**

**import** **spark.sql**

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")

sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

*// Queries are expressed in HiveQL*

sql("SELECT \* FROM src").show()

*// +---+-------+*

*// |key| value|*

*// +---+-------+*

*// |238|val\_238|*

*// | 86| val\_86|*

*// |311|val\_311|*

*// ...*

*// Aggregation queries are also supported.*

sql("SELECT COUNT(\*) FROM src").show()

*// +--------+*

*// |count(1)|*

*// +--------+*

*// | 500 |*

*// +--------+*

*// The results of SQL queries are themselves DataFrames and support all normal functions.*

**val** sqlDF **=** sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

*// The items in DaraFrames are of type Row, which allows you to access each column by ordinal.*

**val** stringsDS **=** sqlDF.map {

**case** **Row**(key**:** Int, value**:** String) **=>** s"Key: $key, Value: $value"

}

stringsDS.show()

*// +--------------------+*

*// | value|*

*// +--------------------+*

*// |Key: 0, Value: val\_0|*

*// |Key: 0, Value: val\_0|*

*// |Key: 0, Value: val\_0|*

*// ...*

*// You can also use DataFrames to create temporary views within a SparkSession.*

**val** recordsDF **=** spark.createDataFrame((1 to 100).map(i **=>** **Record**(i, s"val\_$i")))

recordsDF.createOrReplaceTempView("records")

*// Queries can then join DataFrame data with data stored in Hive.*

sql("SELECT \* FROM records r JOIN src s ON r.key = s.key").show()

*// +---+------+---+------+*

*// |key| value|key| value|*

*// +---+------+---+------+*

*// | 2| val\_2| 2| val\_2|*

*// | 4| val\_4| 4| val\_4|*

*// | 5| val\_5| 5| val\_5|*

*// ...*

**Python：**

**from** **os.path** **import** expanduser, join

**from** **pyspark.sql** **import** SparkSession

**from** **pyspark.sql** **import** Row

*# warehouse\_location points to the default location for managed databases and tables*

warehouse\_location = 'spark-warehouse'

spark = SparkSession \

.builder \

.appName("Python Spark SQL Hive integration example") \

.config("spark.sql.warehouse.dir", warehouse\_location) \

.enableHiveSupport() \

.getOrCreate()

*# spark is an existing SparkSession*

spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")

spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

*# Queries are expressed in HiveQL*

spark.sql("SELECT \* FROM src").show()

*# +---+-------+*

*# |key| value|*

*# +---+-------+*

*# |238|val\_238|*

*# | 86| val\_86|*

*# |311|val\_311|*

*# ...*

*# Aggregation queries are also supported.*

spark.sql("SELECT COUNT(\*) FROM src").show()

*# +--------+*

*# |count(1)|*

*# +--------+*

*# | 500 |*

*# +--------+*

*# The results of SQL queries are themselves DataFrames and support all normal functions.*

sqlDF = spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

*# The items in DaraFrames are of type Row, which allows you to access each column by ordinal.*

stringsDS = sqlDF.rdd.map(**lambda** row: "Key: *%d*, Value: *%s*" % (row.key, row.value))

**for** record **in** stringsDS.collect():

**print**(record)

*# Key: 0, Value: val\_0*

*# Key: 0, Value: val\_0*

*# Key: 0, Value: val\_0*

*# ...*

*# You can also use DataFrames to create temporary views within a SparkSession.*

Record = Row("key", "value")

recordsDF = spark.createDataFrame(map(**lambda** i: Record(i, "val\_" + str(i)), range(1, 101)))

recordsDF.createOrReplaceTempView("records")

*# Queries can then join DataFrame data with data stored in Hive.*

spark.sql("SELECT \* FROM records r JOIN src s ON r.key = s.key").show()

*# +---+------+---+------+*

*# |key| value|key| value|*

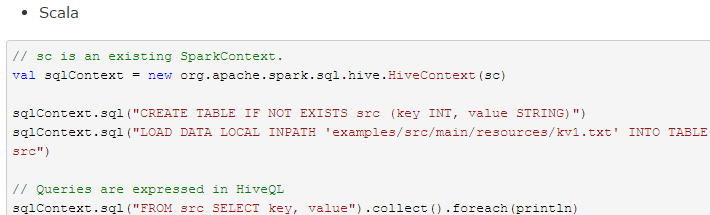
*# +---+------+---+------+*

*# | 2| val\_2| 2| val\_2|*

*# | 4| val\_4| 4| val\_4|*

*# | 5| val\_5| 5| val\_5|*

*# ...*



Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/hive/SparkHiveExample.scala" in the Spark repo.

**Interacting with Different Versions of Hive Metastore**

One of the most important pieces of Spark SQL’s Hive support is interaction with Hive metastore, which enables Spark SQL to access metadata of Hive tables. Starting from Spark 1.4.0, a single binary build of Spark SQL can be used to query different versions of Hive metastores, using the configuration described below. Note that independent of the version of Hive that is being used to talk to the metastore, internally Spark SQL will compile against Hive 1.2.1 and use those classes for internal execution (serdes, UDFs, UDAFs, etc).

The following options can be used to configure the version of Hive that is used to retrieve metadata:

Spark SQL对Hive的支持的很重要的一条就是访问Hive metastore，这个特性使得Spark SQL可以通过Hive metastore获取Hive表的元数据。从Spark 1.4.0开始，Spark SQL只需简单的配置，就支持各版本Hive metastore的访问。注意，涉及到metastore时Spark SQL忽略了Hive的版本。Spark SQL内部将Hive反编译至Hive 1.2.1版本，Spark SQL的内部操作(serdes, UDFs, UDAFs, etc)都调用Hive 1.2.1版本的class。

版本配置项见下面表格：

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.sql.hive.metastore.version | 1.2.1 | Version of the Hive metastore. Available options are 0.12.0through 1.2.1. |
|  |  |  |
| spark.sql.hive.metastore.jars | builtin | Location of the jars that should be used to instantiate the HiveMetastoreClient. This property can be one of three options:  1.builtin  Use Hive 1.2.1, which is bundled with the Spark assembly when -Phive is enabled. When this option is chosen, spark.sql.hive.metastore.version must be either 1.2.1 or not defined.  2.maven  Use Hive jars of specified version downloaded from Maven repositories. This configuration is not generally recommended for production deployments.  3.A classpath in the standard format for the JVM. This classpath must include all of Hive and its dependencies, including the correct version of Hadoop. These jars only need to be present on the driver, but if you are running in yarn cluster mode then you must ensure they are packaged with you application. |
| spark.sql.hive.metastore.sharedPrefixes | com.mysql.jdbc, org.postgresql, com.microsoft.sqlserver, oracle.jdbc | A comma separated list of class prefixes that should be loaded using the classloader that is shared between Spark SQL and a specific version of Hive. An example of classes that should be shared is JDBC drivers that are needed to talk to the metastore. Other classes that need to be shared are those that interact with classes that are already shared. For example, custom appenders that are used by log4j. |
| spark.sql.hive.metastore.barrierPrefixes | (empty) | A comma separated list of class prefixes that should explicitly be reloaded for each version of Hive that Spark SQL is communicating with. For example, Hive UDFs that are declared in a prefix that typically would be shared (i.e. org.apache.spark.\*). |

**JDBC To Other Databases**

Spark SQL also includes a data source that can read data from other databases using JDBC. This functionality should be preferred over using [JdbcRDD](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.JdbcRDD). This is because the results are returned as a DataFrame and they can easily be processed in Spark SQL or joined with other data sources. The JDBC data source is also easier to use from Java or Python as it does not require the user to provide a ClassTag. (Note that this is different than the Spark SQL JDBC server, which allows other applications to run queries using Spark SQL).

Spark SQL支持使用JDBC访问其他数据库。当时用JDBC访问其它数据库时，最好使用JdbcRDD。使用JdbcRDD时，Spark SQL操作返回的DataFrame会很方便，也会很方便的添加其他数据源数据。JDBC数据源因为不需要用户提供ClassTag，所以很适合使用Java或Python进行操作。

To get started you will need to include the JDBC driver for you particular database on the spark classpath. For example, to connect to postgres from the Spark Shell you would run the following command:

使用JDBC访问数据源，需要在spark classpath添加JDBC driver配置。例如，从Spark Shell连接postgres的配置为：

bin/spark-shell --driver-class-path postgresql-9.4.1207.jar --jars postgresql-9.4.1207.jar

Tables from the remote database can be loaded as a DataFrame or Spark SQL Temporary table using the Data Sources API. The following options are supported:

远程数据库的表，可用DataFrame或Spark SQL临时表的方式调用数据源API。支持的参数有：

|  |  |
| --- | --- |
| **Property Name** | **Meaning** |
| url | The JDBC URL to connect to. |
| dbtable | The JDBC table that should be read. Note that anything that is valid in a FROM clause of a SQL query can be used. For example, instead of a full table you could also use a subquery in parentheses. |
| driver | The class name of the JDBC driver to use to connect to this URL. |
| partitionColumn, lowerBound, upperBound, numPartitions | These options must all be specified if any of them is specified. They describe how to partition the table when reading in parallel from multiple workers. partitionColumn must be a numeric column from the table in question. Notice that lowerBound and upperBound are just used to decide the partition stride, not for filtering the rows in table. So all rows in the table will be partitioned and returned. |
| fetchsize | The JDBC fetch size, which determines how many rows to fetch per round trip. This can help performance on JDBC drivers which default to low fetch size (eg. Oracle with 10 rows). |

**Scala：**

**val** jdbcDF **=** spark.read

.format("jdbc")

.option("url", "jdbc:postgresql:dbserver")

.option("dbtable", "schema.tablename")

.option("user", "username")

.option("password", "password")

.load()

Python：

jdbcDF = spark.read \

.format("jdbc") \

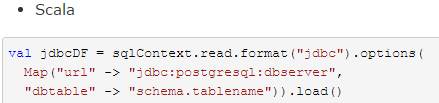
.option("url", "jdbc:postgresql:dbserver") \

.option("dbtable", "schema.tablename") \

.option("user", "username") \

.option("password", "password") \

.load()



Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala" in the Spark repo.

**Troubleshooting**

* The JDBC driver class must be visible to the primordial class loader on the client session and on all executors. This is because Java’s DriverManager class does a security check that results in it ignoring all drivers not visible to the primordial class loader when one goes to open a connection. One convenient way to do this is to modify compute\_classpath.sh on all worker nodes to include your driver JARs.
* Some databases, such as H2, convert all names to upper case. You’ll need to use upper case to refer to those names in Spark SQL.
* 在客户端session和所有的executors上，JDBC driver必须对启动类加载器（primordial class loader）设置为visible。因为当创建一个connection时，Java的DriverManager类会执行安全验证，安全验证将忽略所有对启动类加载器为非visible的driver。一个很方便的解决方法是，修改所有worker节点上的compute\_classpath.sh脚本，将driver JARs添加至脚本。
* 有些数据库（例：H2）将所有的名字转换为大写，所以在这些数据库中，Spark SQL也需要将名字全部大写。

**Performance Tuning**

For some workloads it is possible to improve performance by either caching data in memory, or by turning on some experimental options.

**Caching Data In Memory**

Spark SQL can cache tables using an in-memory columnar format by calling spark.cacheTable("tableName") or dataFrame.cache(). Then Spark SQL will scan only required columns and will automatically tune compression to minimize memory usage and GC pressure. You can call spark.uncacheTable("tableName") to remove the table from memory.

Spark SQL可以通过调用sqlContext.cacheTable("tableName") 或者dataFrame.cache()，将表用一种柱状格式（ an in­memory columnar format）缓存至内存中。然后Spark SQL在执行查询任务时，只需扫描必需的列，从而以减少扫描数据量、提高性能。通过缓存数据，Spark SQL还可以自动调节压缩，从而达到最小化内存使用率和降低GC压力的目的。调用sqlContext.uncacheTable("tableName")可将缓存的数据移出内存。

Configuration of in-memory caching can be done using the setConf method on SparkSession or by running SET key=value commands using SQL.

可通过两种配置方式开启缓存数据功能：

* 使用SQLContext的setConf方法
* 执行SQL命令 SET key=value

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.sql.inMemoryColumnarStorage.compressed | true | When set to true Spark SQL will automatically select a compression codec for each column based on statistics of the data. |
| spark.sql.inMemoryColumnarStorage.batchSize | 10000 | Controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk OOMs when caching data. |

**Other Configuration Options**

The following options can also be used to tune the performance of query execution. It is possible that these options will be deprecated in future release as more optimizations are performed automatically.

可以通过配置下表中的参数调节Spark SQL的性能。在后续的Spark版本中将逐渐增强自动调优功能，下表中的参数在后续的版本中或许将不再需要配置。

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.sql.files.maxPartitionBytes | 134217728 (128 MB) | The maximum number of bytes to pack into a single partition when reading files. |
| spark.sql.files.openCostInBytes | 4194304 (4 MB) | The estimated cost to open a file, measured by the number of bytes could be scanned in the same time. This is used when putting multiple files into a partition. It is better to over estimated, then the partitions with small files will be faster than partitions with bigger files (which is scheduled first). |
| spark.sql.broadcastTimeout | 300 | Timeout in seconds for the broadcast wait time in broadcast joins |
| spark.sql.autoBroadcastJoinThreshold | 10485760 (10 MB) | Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. By setting this value to -1 broadcasting can be disabled. Note that currently statistics are only supported for Hive Metastore tables where the command ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan has been run. |
| spark.sql.shuffle.partitions | 200 | Configures the number of partitions to use when shuffling data for joins or aggregations. |

**Distributed SQL Engine**

Spark SQL can also act as a distributed query engine using its JDBC/ODBC or command-line interface. In this mode, end-users or applications can interact with Spark SQL directly to run SQL queries, without the need to write any code.

使用Spark SQL的JDBC/ODBC或者CLI，可以将Spark SQL作为一个分布式查询引擎。终端用户或应用不需要编写额外的代码，可以直接使用Spark SQL执行SQL查询。

**Running the Thrift JDBC/ODBC server**

The Thrift JDBC/ODBC server implemented here corresponds to the [HiveServer2](https://cwiki.apache.org/confluence/display/Hive/Setting+Up+HiveServer2) in Hive 1.2.1 You can test the JDBC server with the beeline script that comes with either Spark or Hive 1.2.1.

To start the JDBC/ODBC server, run the following in the Spark directory:

./sbin/start-thriftserver.sh

This script accepts all bin/spark-submit command line options, plus a --hiveconf option to specify Hive properties. You may run ./sbin/start-thriftserver.sh --help for a complete list of all available options. By default, the server listens on localhost:10000. You may override this behaviour via either environment variables, i.e.:

这里运行的Thrift JDBC/ODBC服务与Hive 1.2.1中的HiveServer2一致。可以在Spark目录下执行如下命令来启动JDBC/ODBC服务：

./sbin/start-thriftserver.sh

这个命令接收所有 bin/spark-submit 命令行参数，添加一个 --hiveconf 参数来指定Hive的属性。详细的参数说明请执行命令 ./sbin/start-thriftserver.sh --help 。  
服务默认监听端口为localhost:10000。有两种方式修改默认监听端口：

export HIVE\_SERVER2\_THRIFT\_PORT=<listening-port>

export HIVE\_SERVER2\_THRIFT\_BIND\_HOST=<listening-host>

./sbin/start-thriftserver.sh **\**

--master <master-uri> **\**

...

or system properties:

./sbin/start-thriftserver.sh **\**

--hiveconf hive.server2.thrift.port=<listening-port> **\**

--hiveconf hive.server2.thrift.bind.host=<listening-host> **\**

--master <master-uri>

...

Now you can use beeline to test the Thrift JDBC/ODBC server:

./bin/beeline

Connect to the JDBC/ODBC server in beeline with:

beeline> !connect jdbc:hive2://localhost:10000

Beeline will ask you for a username and password. In non-secure mode, simply enter the username on your machine and a blank password. For secure mode, please follow the instructions given in the [beeline documentation](https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients).

在非安全模式下，只需要输入机器上的一个用户名即可，无需密码。在安全模式下，beeline会要求输入用户名和密码。安全模式下的详细要求，请阅读[beeline documentation](https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients)的说明。

Configuration of Hive is done by placing your hive-site.xml, core-site.xml and hdfs-site.xml files in conf/.

You may also use the beeline script that comes with Hive.

Thrift JDBC server also supports sending thrift RPC messages over HTTP transport. Use the following setting to enable HTTP mode as system property or in hive-site.xml file in conf/:

Thrift JDBC服务也支持通过HTTP传输发送thrift RPC messages。开启HTTP模式需要将下面的配参数配置到系统属性或 conf/: 下的 hive-site.xml中

hive.server2.transport.mode - Set this to value: http

hive.server2.thrift.http.port - HTTP port number fo listen on; default is 10001

hive.server2.http.endpoint - HTTP endpoint; default is cliservice

To test, use beeline to connect to the JDBC/ODBC server in http mode with:

beeline> !connect jdbc:hive2://<host>:<port>/<database>?hive.server2.transport.mode=http;hive.server2.thrift.http.path=<http\_endpoint>

**Running the Spark SQL CLI**

The Spark SQL CLI is a convenient tool to run the Hive metastore service in local mode and execute queries input from the command line. Note that the Spark SQL CLI cannot talk to the Thrift JDBC server.

To start the Spark SQL CLI, run the following in the Spark directory:

Spark SQL CLI可以很方便的在本地运行Hive元数据服务以及从命令行执行查询任务。需要注意的是，Spark SQL CLI不能与Thrift JDBC服务交互。  
在Spark目录下执行如下命令启动Spark SQL CLI：

./bin/spark-sql

Configuration of Hive is done by placing your hive-site.xml, core-site.xml and hdfs-site.xml files in conf/. You may run ./bin/spark-sql --help for a complete list of all available options.

**Migration Guide**

**Upgrading From Spark SQL 1.6 to 2.0**

* SparkSession is now the new entry point of Spark that replaces the old SQLContext and HiveContext. Note that the old SQLContext and HiveContext are kept for backward compatibility. A new catalog interface is accessible from SparkSession - existing API on databases and tables access such as listTables, createExternalTable, dropTempView, cacheTable are moved here.
* Dataset API and DataFrame API are unified. In Scala, DataFrame becomes a type alias for Dataset[Row], while Java API users must replace DataFrame with Dataset<Row>. Both the typed transformations (e.g. map, filter, and groupByKey) and untyped transformations (e.g. selectand groupBy) are available on the Dataset class. Since compile-time type-safety in Python and R is not a language feature, the concept of Dataset does not apply to these languages’ APIs. Instead, DataFrame remains the primary programing abstraction, which is analogous to the single-node data frame notion in these languages.
* Dataset and DataFrame API unionAll has been deprecated and replaced by union
* Dataset and DataFrame API explode has been deprecated, alternatively, use functions.explode() with select or flatMap
* Dataset and DataFrame API registerTempTable has been deprecated and replaced by createOrReplaceTempView

**Upgrading From Spark SQL 1.5 to 1.6**

* From Spark 1.6, by default the Thrift server runs in multi-session mode. Which means each JDBC/ODBC connection owns a copy of their own SQL configuration and temporary function registry. Cached tables are still shared though. If you prefer to run the Thrift server in the old single-session mode, please set option spark.sql.hive.thriftServer.singleSession to true. You may either add this option to spark-defaults.conf, or pass it to start-thriftserver.sh via --conf:

./sbin/start-thriftserver.sh **\**

--conf spark.sql.hive.thriftServer.singleSession=true **\**

...

* Since 1.6.1, withColumn method in sparkR supports adding a new column to or replacing existing columns of the same name of a DataFrame.
* From Spark 1.6, LongType casts to TimestampType expect seconds instead of microseconds. This change was made to match the behavior of Hive 1.2 for more consistent type casting to TimestampType from numeric types. See [SPARK-11724](https://issues.apache.org/jira/browse/SPARK-11724) for details.

**Upgrading From Spark SQL 1.4 to 1.5**

* Optimized execution using manually managed memory (Tungsten) is now enabled by default, along with code generation for expression evaluation. These features can both be disabled by setting spark.sql.tungsten.enabled to false.
* Parquet schema merging is no longer enabled by default. It can be re-enabled by setting spark.sql.parquet.mergeSchema to true.
* Resolution of strings to columns in python now supports using dots (.) to qualify the column or access nested values. For example df['table.column.nestedField']. However, this means that if your column name contains any dots you must now escape them using backticks (e.g., table.`column.with.dots`.nested).
* In-memory columnar storage partition pruning is on by default. It can be disabled by settingspark.sql.inMemoryColumnarStorage.partitionPruning to false.
* Unlimited precision decimal columns are no longer supported, instead Spark SQL enforces a maximum precision of 38. When inferring schema from BigDecimal objects, a precision of (38, 18) is now used. When no precision is specified in DDL then the default remains Decimal(10, 0).
* Timestamps are now stored at a precision of 1us, rather than 1ns
* In the sql dialect, floating point numbers are now parsed as decimal. HiveQL parsing remains unchanged.
* The canonical name of SQL/DataFrame functions are now lower case (e.g. sum vs SUM).
* JSON data source will not automatically load new files that are created by other applications (i.e. files that are not inserted to the dataset through Spark SQL). For a JSON persistent table (i.e. the metadata of the table is stored in Hive Metastore), users can use REFRESH TABLESQL command or HiveContext’s refreshTable method to include those new files to the table. For a DataFrame representing a JSON dataset, users need to recreate the DataFrame and the new DataFrame will include new files.
* DataFrame.withColumn method in pySpark supports adding a new column or replacing existing columns of the same name.

**Upgrading from Spark SQL 1.3 to 1.4**

**DataFrame data reader/writer interface**

Based on user feedback, we created a new, more fluid API for reading data in (SQLContext.read) and writing data out (DataFrame.write), and deprecated the old APIs (e.g. SQLContext.parquetFile, SQLContext.jsonFile).

See the API docs for SQLContext.read ( [Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.SQLContext@read:DataFrameReader), [Java](http://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/SQLContext.html#read()), [Python](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.SQLContext.read) ) and DataFrame.write ( [Scala](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame@write:DataFrameWriter), [Java](http://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/DataFrame.html#write()), [Python](http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.write) ) more information.

**DataFrame.groupBy retains grouping columns**

Based on user feedback, we changed the default behavior of DataFrame.groupBy().agg() to retain the grouping columns in the resulting DataFrame. To keep the behavior in 1.3, set spark.sql.retainGroupColumns to false.

Scala：

*// In 1.3.x, in order for the grouping column "department" to show up,*

*// it must be included explicitly as part of the agg function call.*

df.groupBy("department").agg($"department", max("age"), sum("expense"))

*// In 1.4+, grouping column "department" is included automatically.*

df.groupBy("department").agg(max("age"), sum("expense"))

*// Revert to 1.3 behavior (not retaining grouping column) by:*

sqlContext.setConf("spark.sql.retainGroupColumns", "false")

Python：

**import** **pyspark.sql.functions** **as** **func**

*# In 1.3.x, in order for the grouping column "department" to show up,*

*# it must be included explicitly as part of the agg function call.*

df.groupBy("department").agg(df["department"], func.max("age"), func.sum("expense"))

*# In 1.4+, grouping column "department" is included automatically.*

df.groupBy("department").agg(func.max("age"), func.sum("expense"))

*# Revert to 1.3.x behavior (not retaining grouping column) by:*

sqlContext.setConf("spark.sql.retainGroupColumns", "false")

**Behavior change on DataFrame.withColumn**

Prior to 1.4, DataFrame.withColumn() supports adding a column only. The column will always be added as a new column with its specified name in the result DataFrame even if there may be any existing columns of the same name. Since 1.4, DataFrame.withColumn() supports adding a column of a different name from names of all existing columns or replacing existing columns of the same name.

Note that this change is only for Scala API, not for PySpark and SparkR.

**Upgrading from Spark SQL 1.0-1.2 to 1.3**

In Spark 1.3 we removed the “Alpha” label from Spark SQL and as part of this did a cleanup of the available APIs. From Spark 1.3 onwards, Spark SQL will provide binary compatibility with other releases in the 1.X series. This compatibility guarantee excludes APIs that are explicitly marked as unstable (i.e., DeveloperAPI or Experimental).

**Rename of SchemaRDD to DataFrame**

The largest change that users will notice when upgrading to Spark SQL 1.3 is that SchemaRDD has been renamed to DataFrame. This is primarily because DataFrames no longer inherit from RDD directly, but instead provide most of the functionality that RDDs provide though their own implementation. DataFrames can still be converted to RDDs by calling the .rdd method.

In Scala there is a type alias from SchemaRDD to DataFrame to provide source compatibility for some use cases. It is still recommended that users update their code to use DataFrame instead. Java and Python users will need to update their code.

**Unification of the Java and Scala APIs**

Prior to Spark 1.3 there were separate Java compatible classes (JavaSQLContext and JavaSchemaRDD) that mirrored the Scala API. In Spark 1.3 the Java API and Scala API have been unified. Users of either language should use SQLContext and DataFrame. In general theses classes try to use types that are usable from both languages (i.e. Array instead of language specific collections). In some cases where no common type exists (e.g., for passing in closures or Maps) function overloading is used instead.

Additionally the Java specific types API has been removed. Users of both Scala and Java should use the classes present in org.apache.spark.sql.types to describe schema programmatically.

**Isolation of Implicit Conversions and Removal of dsl Package (Scala-only)**

Many of the code examples prior to Spark 1.3 started with import sqlContext.\_, which brought all of the functions from sqlContext into scope. In Spark 1.3 we have isolated the implicit conversions for converting RDDs into DataFrames into an object inside of the SQLContext. Users should now write import sqlContext.implicits.\_.

Additionally, the implicit conversions now only augment RDDs that are composed of Products (i.e., case classes or tuples) with a method toDF, instead of applying automatically.

When using function inside of the DSL (now replaced with the DataFrame API) users used to import org.apache.spark.sql.catalyst.dsl. Instead the public dataframe functions API should be used: import org.apache.spark.sql.functions.\_.

**Removal of the type aliases in org.apache.spark.sql for DataType (Scala-only)**

Spark 1.3 removes the type aliases that were present in the base sql package for DataType. Users should instead import the classes in org.apache.spark.sql.types

**UDF Registration Moved to sqlContext.udf (Java & Scala)**

Functions that are used to register UDFs, either for use in the DataFrame DSL or SQL, have been moved into the udf object in SQLContext.

Scala：

sqlContext.udf.register("strLen", (s**:** String) **=>** s.length())

Python UDF registration is unchanged.

**Python DataTypes No Longer Singletons**

When using DataTypes in Python you will need to construct them (i.e. StringType()) instead of referencing a singleton.

**Compatibility with Apache Hive**

Spark SQL is designed to be compatible with the Hive Metastore, SerDes and UDFs. Currently Hive SerDes and UDFs are based on Hive 1.2.1, and Spark SQL can be connected to different versions of Hive Metastore (from 0.12.0 to 1.2.1. Also see [Interacting with Different Versions of Hive Metastore] (#interacting-with-different-versions-of-hive-metastore)).

**Deploying in Existing Hive Warehouses**

The Spark SQL Thrift JDBC server is designed to be “out of the box” compatible with existing Hive installations. You do not need to modify your existing Hive Metastore or change the data placement or partitioning of your tables.

**Supported Hive Features**

Spark SQL supports the vast majority of Hive features, such as:

* Hive query statements, including:
  + SELECT
  + GROUP BY
  + ORDER BY
  + CLUSTER BY
  + SORT BY
* All Hive operators, including:
  + Relational operators (=, ⇔, ==, <>, <, >, >=, <=, etc)
  + Arithmetic operators (+, -, \*, /, %, etc)
  + Logical operators (AND, &&, OR, ||, etc)
  + Complex type constructors
  + Mathematical functions (sign, ln, cos, etc)
  + String functions (instr, length, printf, etc)
* User defined functions (UDF)
* User defined aggregation functions (UDAF)
* User defined serialization formats (SerDes)
* Window functions
* Joins
  + JOIN
  + {LEFT|RIGHT|FULL} OUTER JOIN
  + LEFT SEMI JOIN
  + CROSS JOIN
* Unions
* Sub-queries
  + SELECT col FROM ( SELECT a + b AS col from t1) t2
* Sampling
* Explain
* Partitioned tables including dynamic partition insertion
* View
* All Hive DDL Functions, including:
  + CREATE TABLE
  + CREATE TABLE AS SELECT
  + ALTER TABLE
* Most Hive Data types, including:
  + TINYINT
  + SMALLINT
  + INT
  + BIGINT
  + BOOLEAN
  + FLOAT
  + DOUBLE
  + STRING
  + BINARY
  + TIMESTAMP
  + DATE
  + ARRAY<>
  + MAP<>
  + STRUCT<>

**Unsupported Hive Functionality**

Below is a list of Hive features that we don’t support yet. Most of these features are rarely used in Hive deployments.

**Major Hive Features**

* Tables with buckets: bucket is the hash partitioning within a Hive table partition. Spark SQL doesn’t support buckets yet.

**Esoteric Hive Features**

* UNION type
* Unique join
* Column statistics collecting: Spark SQL does not piggyback scans to collect column statistics at the moment and only supports populating the sizeInBytes field of the hive metastore.

**Hive Input/Output Formats**

* File format for CLI: For results showing back to the CLI, Spark SQL only supports TextOutputFormat.
* Hadoop archive

**Hive Optimizations**

A handful of Hive optimizations are not yet included in Spark. Some of these (such as indexes) are less important due to Spark SQL’s in-memory computational model. Others are slotted for future releases of Spark SQL.

* Block level bitmap indexes and virtual columns (used to build indexes)
* Automatically determine the number of reducers for joins and groupbys: Currently in Spark SQL, you need to control the degree of parallelism post-shuffle using “SET spark.sql.shuffle.partitions=[num\_tasks];”.
* Meta-data only query: For queries that can be answered by using only meta data, Spark SQL still launches tasks to compute the result.
* Skew data flag: Spark SQL does not follow the skew data flags in Hive.
* STREAMTABLE hint in join: Spark SQL does not follow the STREAMTABLE hint.
* Merge multiple small files for query results: if the result output contains multiple small files, Hive can optionally merge the small files into fewer large files to avoid overflowing the HDFS metadata. Spark SQL does not support that.

**Reference**

**Data Types**

Spark SQL and DataFrames support the following data types:

* Numeric types
  + ByteType: Represents 1-byte signed integer numbers. The range of numbers is from -128 to 127.
  + ShortType: Represents 2-byte signed integer numbers. The range of numbers is from -32768 to 32767.
  + IntegerType: Represents 4-byte signed integer numbers. The range of numbers is from -2147483648 to 2147483647.
  + LongType: Represents 8-byte signed integer numbers. The range of numbers is from -9223372036854775808 to 9223372036854775807.
  + FloatType: Represents 4-byte single-precision floating point numbers.
  + DoubleType: Represents 8-byte double-precision floating point numbers.
  + DecimalType: Represents arbitrary-precision signed decimal numbers. Backed internally by java.math.BigDecimal. A BigDecimal consists of an arbitrary precision integer unscaled value and a 32-bit integer scale.
* String type
  + StringType: Represents character string values.
* Binary type
  + BinaryType: Represents byte sequence values.
* Boolean type
  + BooleanType: Represents boolean values.
* Datetime type
  + TimestampType: Represents values comprising values of fields year, month, day, hour, minute, and second.
  + DateType: Represents values comprising values of fields year, month, day.
* Complex types
  + ArrayType(elementType, containsNull): Represents values comprising a sequence of elements with the type of elementType. containsNull is used to indicate if elements in a ArrayType value can have null values.
  + MapType(keyType, valueType, valueContainsNull): Represents values comprising a set of key-value pairs. The data type of keys are described by keyType and the data type of values are described by valueType. For a MapType value, keys are not allowed to have nullvalues. valueContainsNull is used to indicate if values of a MapType value can have null values.
  + StructType(fields): Represents values with the structure described by a sequence of StructFields (fields).
    - StructField(name, dataType, nullable): Represents a field in a StructType. The name of a field is indicated by name. The data type of a field is indicated by dataType. nullable is used to indicate if values of this fields can have null values.

All data types of Spark SQL are located in the package org.apache.spark.sql.types. You can access them by doing

**Scala：import** **org.apache.spark.sql.types.\_**

**Python：from** **pyspark.sql.types** **import** \*

Find full example code at "examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala" in the Spark repo.

|  |  |  |
| --- | --- | --- |
| **Data type** | **Value type in Scala** | **API to access or create a data type** |
| **ByteType** | Byte | ByteType |
| **ShortType** | Short | ShortType |
| **IntegerType** | Int | IntegerType |
| **LongType** | Long | LongType |
| **FloatType** | Float | FloatType |
| **DoubleType** | Double | DoubleType |
| **DecimalType** | java.math.BigDecimal | DecimalType |
| **StringType** | String | StringType |
| **BinaryType** | Array[Byte] | BinaryType |
| **BooleanType** | Boolean | BooleanType |
| **TimestampType** | java.sql.Timestamp | TimestampType |
| **DateType** | java.sql.Date | DateType |
| **ArrayType** | scala.collection.Seq | ArrayType(*elementType*, [*containsNull*]) **Note:** The default value of *containsNull* is *true*. |
| **MapType** | scala.collection.Map | MapType(*keyType*, *valueType*, [*valueContainsNull*]) **Note:** The default value of *valueContainsNull* is *true*. |
| **StructType** | org.apache.spark.sql.Row | StructType(*fields*) **Note:** *fields* is a Seq of StructFields. Also, two fields with the same name are not allowed. |
| **StructField** | The value type in Scala of the data type of this field (For example, Int for a StructField with the data type IntegerType) | StructField(*name*, *dataType*, *nullable*) |

**NaN Semantics**

There is specially handling for not-a-number (NaN) when dealing with float or double types that does not exactly match standard floating point semantics. Specifically:

当处理float或double类型时，如果类型不符合标准的浮点语义，则使用专门的处理方式NaN。需要注意的是：

* NaN = NaN 返回 true
* 可以对NaN值进行聚合操作
* 在join操作中，key为NaN时，NaN值与普通的数值处理逻辑相同
* NaN值大于所有的数值型数据，在升序排序中排在最后
* NaN = NaN returns true.
* In aggregations all NaN values are grouped together.
* NaN is treated as a normal value in join keys.
* NaN values go last when in ascending order, larger than any other numeric value.