# Spark SQL, DataFrames and Datasets Guide

* [Overview](http://spark.apache.org/docs/latest/sql-programming-guide.html#overview)
  + [SQL](http://spark.apache.org/docs/latest/sql-programming-guide.html#sql)
  + [Datasets and DataFrames](http://spark.apache.org/docs/latest/sql-programming-guide.html#datasets-and-dataframes)
* [Getting Started](http://spark.apache.org/docs/latest/sql-programming-guide.html#getting-started)
  + [Starting Point: SparkSession](http://spark.apache.org/docs/latest/sql-programming-guide.html#starting-point-sparksession)
  + [Creating DataFrames](http://spark.apache.org/docs/latest/sql-programming-guide.html#creating-dataframes)
  + [Untyped Dataset Operations (aka DataFrame Operations)](http://spark.apache.org/docs/latest/sql-programming-guide.html#untyped-dataset-operations-aka-dataframe-operations)
  + [Running SQL Queries Programmatically](http://spark.apache.org/docs/latest/sql-programming-guide.html#running-sql-queries-programmatically)
  + [Global Temporary View](http://spark.apache.org/docs/latest/sql-programming-guide.html#global-temporary-view)
  + [Creating Datasets](http://spark.apache.org/docs/latest/sql-programming-guide.html#creating-datasets)
  + [Interoperating with RDDs](http://spark.apache.org/docs/latest/sql-programming-guide.html#interoperating-with-rdds)
    - [Inferring the Schema Using Reflection](http://spark.apache.org/docs/latest/sql-programming-guide.html#inferring-the-schema-using-reflection)
    - [Programmatically Specifying the Schema](http://spark.apache.org/docs/latest/sql-programming-guide.html#programmatically-specifying-the-schema)
  + [Aggregations](http://spark.apache.org/docs/latest/sql-programming-guide.html#aggregations)
    - [Untyped User-Defined Aggregate Functions](http://spark.apache.org/docs/latest/sql-programming-guide.html#untyped-user-defined-aggregate-functions)
    - [Type-Safe User-Defined Aggregate Functions](http://spark.apache.org/docs/latest/sql-programming-guide.html#type-safe-user-defined-aggregate-functions)
* [Data Sources](http://spark.apache.org/docs/latest/sql-programming-guide.html#data-sources)
  + [Generic Load/Save Functions](http://spark.apache.org/docs/latest/sql-programming-guide.html#generic-loadsave-functions)
    - [Manually Specifying Options](http://spark.apache.org/docs/latest/sql-programming-guide.html#manually-specifying-options)
    - [Run SQL on files directly](http://spark.apache.org/docs/latest/sql-programming-guide.html#run-sql-on-files-directly)
    - [Save Modes](http://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes)
    - [Saving to Persistent Tables](http://spark.apache.org/docs/latest/sql-programming-guide.html#saving-to-persistent-tables)
  + [Parquet Files](http://spark.apache.org/docs/latest/sql-programming-guide.html#parquet-files)
    - [Loading Data Programmatically](http://spark.apache.org/docs/latest/sql-programming-guide.html#loading-data-programmatically)
    - [Partition Discovery](http://spark.apache.org/docs/latest/sql-programming-guide.html#partition-discovery)
    - [Schema Merging](http://spark.apache.org/docs/latest/sql-programming-guide.html#schema-merging)
    - [Hive metastore Parquet table conversion](http://spark.apache.org/docs/latest/sql-programming-guide.html#hive-metastore-parquet-table-conversion)
      * [Hive/Parquet Schema Reconciliation](http://spark.apache.org/docs/latest/sql-programming-guide.html#hiveparquet-schema-reconciliation)
      * [Metadata Refreshing](http://spark.apache.org/docs/latest/sql-programming-guide.html#metadata-refreshing)
    - [Configuration](http://spark.apache.org/docs/latest/sql-programming-guide.html#configuration)
  + [JSON Datasets](http://spark.apache.org/docs/latest/sql-programming-guide.html#json-datasets)
  + [Hive Tables](http://spark.apache.org/docs/latest/sql-programming-guide.html#hive-tables)
    - [Interacting with Different Versions of Hive Metastore](http://spark.apache.org/docs/latest/sql-programming-guide.html#interacting-with-different-versions-of-hive-metastore)
  + [JDBC To Other Databases](http://spark.apache.org/docs/latest/sql-programming-guide.html#jdbc-to-other-databases)
  + [Troubleshooting](http://spark.apache.org/docs/latest/sql-programming-guide.html#troubleshooting)
* [Performance Tuning](http://spark.apache.org/docs/latest/sql-programming-guide.html#performance-tuning)
  + [Caching Data In Memory](http://spark.apache.org/docs/latest/sql-programming-guide.html#caching-data-in-memory)
  + [Other Configuration Options](http://spark.apache.org/docs/latest/sql-programming-guide.html#other-configuration-options)
* [Distributed SQL Engine](http://spark.apache.org/docs/latest/sql-programming-guide.html#distributed-sql-engine)
  + [Running the Thrift JDBC/ODBC server](http://spark.apache.org/docs/latest/sql-programming-guide.html#running-the-thrift-jdbcodbc-server)
  + [Running the Spark SQL CLI](http://spark.apache.org/docs/latest/sql-programming-guide.html#running-the-spark-sql-cli)
* [Migration Guide](http://spark.apache.org/docs/latest/sql-programming-guide.html#migration-guide)
  + [Upgrading From Spark SQL 2.0 to 2.1](http://spark.apache.org/docs/latest/sql-programming-guide.html#upgrading-from-spark-sql-20-to-21)
  + [Upgrading From Spark SQL 1.6 to 2.0](http://spark.apache.org/docs/latest/sql-programming-guide.html#upgrading-from-spark-sql-16-to-20)
  + [Upgrading From Spark SQL 1.5 to 1.6](http://spark.apache.org/docs/latest/sql-programming-guide.html#upgrading-from-spark-sql-15-to-16)
  + [Upgrading From Spark SQL 1.4 to 1.5](http://spark.apache.org/docs/latest/sql-programming-guide.html#upgrading-from-spark-sql-14-to-15)
  + [Upgrading from Spark SQL 1.3 to 1.4](http://spark.apache.org/docs/latest/sql-programming-guide.html#upgrading-from-spark-sql-13-to-14)
    - [DataFrame data reader/writer interface](http://spark.apache.org/docs/latest/sql-programming-guide.html#dataframe-data-readerwriter-interface)
    - [DataFrame.groupBy retains grouping columns](http://spark.apache.org/docs/latest/sql-programming-guide.html#dataframegroupby-retains-grouping-columns)
    - [Behavior change on DataFrame.withColumn](http://spark.apache.org/docs/latest/sql-programming-guide.html#behavior-change-on-dataframewithcolumn)
  + [Upgrading from Spark SQL 1.0-1.2 to 1.3](http://spark.apache.org/docs/latest/sql-programming-guide.html#upgrading-from-spark-sql-10-12-to-13)
    - [Rename of SchemaRDD to DataFrame](http://spark.apache.org/docs/latest/sql-programming-guide.html#rename-of-schemardd-to-dataframe)
    - [Unification of the Java and Scala APIs](http://spark.apache.org/docs/latest/sql-programming-guide.html#unification-of-the-java-and-scala-apis)
    - [Isolation of Implicit Conversions and Removal of dsl Package (Scala-only)](http://spark.apache.org/docs/latest/sql-programming-guide.html#isolation-of-implicit-conversions-and-removal-of-dsl-package-scala-only)
    - [Removal of the type aliases in org.apache.spark.sql for DataType (Scala-only)](http://spark.apache.org/docs/latest/sql-programming-guide.html#removal-of-the-type-aliases-in-orgapachesparksql-for-datatype-scala-only)
    - [UDF Registration Moved to sqlContext.udf (Java & Scala)](http://spark.apache.org/docs/latest/sql-programming-guide.html#udf-registration-moved-to-sqlcontextudf-java--scala)
    - [Python DataTypes No Longer Singletons](http://spark.apache.org/docs/latest/sql-programming-guide.html#python-datatypes-no-longer-singletons)
  + [Compatibility with Apache Hive](http://spark.apache.org/docs/latest/sql-programming-guide.html#compatibility-with-apache-hive)
    - [Deploying in Existing Hive Warehouses](http://spark.apache.org/docs/latest/sql-programming-guide.html#deploying-in-existing-hive-warehouses)
    - [Supported Hive Features](http://spark.apache.org/docs/latest/sql-programming-guide.html#supported-hive-features)
    - [Unsupported Hive Functionality](http://spark.apache.org/docs/latest/sql-programming-guide.html#unsupported-hive-functionality)
* [Reference](http://spark.apache.org/docs/latest/sql-programming-guide.html#reference)
  + [Data Types](http://spark.apache.org/docs/latest/sql-programming-guide.html#data-types)
  + [NaN Semantics](http://spark.apache.org/docs/latest/sql-programming-guide.html#nan-semantics)

# Overview

Spark SQL是用于结构化数据处理的Spark模块。 与基本的Spark RDD API不同，Spark SQL提供的接口为Spark提供了有关正在执行的数据和计算结构的更多信息。 在内部，Spark SQL使用这些额外的信息执行额外的优化。 有几种与Spark SQL进行交互的方法，包括SQL和Dataset API。 当计算结果时，使用相同的执行引擎，独立于您用来表达计算的API /语言。 这种统一意味着开发人员可以轻松地在不同API之间来回切换，从而提供了最自然的方式来表达给定的转换。

此页面上的所有示例都使用Spark分发中包含的示例数据，并且可以在spark-shell，pyspark shell或sparkR shell中运行。

## SQL

Spark SQL的一个用途是执行SQL查询。 Spark SQL也可用于从现有的Hive安装中读取数据。 有关如何配置此功能的更多信息，请参阅Hive Tables部分。 当从另一种编程语言运行SQL时，结果将作为Dataset / DataFrame返回。 您还可以使用命令行或JDBC / ODBC与SQL界面进行交互。

**数据集和数据**

数据集是分布式数据集合。数据集是Spark 1.6中添加的一个新界面，它提供了RDD（强类型，使用强大的lambda函数的功能）的好处，并具有Spark SQL优化的执行引擎的优点。可以从JVM对象构建数据集，然后使用功能转换（map，flatMap，filter等）进行操作。数据集API在Scala和Java中可用。 Python没有对Dataset API的支持。但是由于Python的动态性质，Dataset API的许多优点已经可用（即您可以按名称自然地访问行的字段row.columnName）。 R的情况类似。

DataFrame是一个组织成命名列的数据集。它在概念上等同于关系数据库中的表或R / Python中的数据框架，但是在更加优化的范围内。 DataFrames可以从各种来源构建，例如：结构化数据文件，Hive中的表，外部数据库或现有RDD。 DataFrame API在Scala，Java，Python和R中可用。在Scala和Java中，DataFrame由“数据集”行表示。在Scala API中，DataFrame只是Dataset [Row]的类型别名。而在Java API中，用户需要使用Dataset <Row>来表示DataFrame。

在本文档中，我们经常将Scala / Java数据集的Rows称为DataFrames。

# Getting Started

## Starting Point: SparkSession

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_0)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_0)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_0)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_0)

Spark中所有功能的入口点是SparkSession类。 要创建基本的SparkSession，只需使用SparkSession.builder：

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/python/sql/basic.py" in the Spark repo.**

Spark 2.0中的SparkSession为Hive功能提供内置支持，包括使用HiveQL编写查询，访问Hive UDF以及从Hive表读取数据的功能。 要使用这些功能，您不需要具有现有的Hive设置。

## Creating DataFrames

* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_1)

使用SparkSession，应用程序可以从现有的RDD，Hive表或Spark数据源创建DataFrames。

例如，以下内容基于JSON文件的内容创建一个DataFrame：

# 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/python/sql/basic.py" in the Spark repo.

非类型数据集操作（又名DataFrame操作）

DataFrames为Scala，Java，Python和R中的结构化数据操作提供了一个域专用语言。

如上所述，在Spark 2.0中，DataFrames只是Scala和Java API中的Rows数据集。 与“类型转换”相反，这些操作也称为“无类型转换”，并带有强类型的Scala / Java数据集。

这里我们包括使用Datasets的结构化数据处理的一些基本示例：

* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_2)

在Python中，可以通过属性（df.age）或索引（df ['age']）访问DataFrame的列。 虽然前者方便了交互式数据挖掘，但是高度鼓励用户使用后一种形式，这是将来的证明，也不会因属于DataFrame类的列名而中断。

# 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/python/sql/basic.py" in the Spark repo.

有关可在DataFrame上执行的操作类型的完整列表，请参阅API文档。

除了简单的列引用和表达式，DataFrames还具有丰富的函数库，包括字符串操作，日期算术，常用的数学运算等。 完整的列表在DataFrame函数参考中可用。

## Running SQL Queries Programmatically

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_3)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_3)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_3)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_3)

SparkSession上的sql函数使应用程序以编程方式运行SQL查询，并将结果作为DataFrame返回。

# 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/python/sql/basic.py" in the Spark repo.

全局临时视图

Spark SQL中的临时视图是会话范围的，如果创建它的会话终止，它将消失。 如果要在所有会话之间共享临时视图，并保持活动状态，直到Spark应用程序终止，您可以创建一个全局临时视图。 全局临时视图与系统保留的数据库global\_temp相关联，我们必须使用限定名称来引用它。 SELECT \* FROM global\_temp.view1。

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_4)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_4)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_4)
* [**Sql**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_sql_4)

# Register the DataFrame as a global temporary view

df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database `global\_temp`

spark.sql("SELECT \* FROM global\_temp.people").show()

# +----+-------+

# | age| name|

# +----+-------+

# |null|Michael|

# | 30| Andy|

# | 19| Justin|

# +----+-------+

# Global temporary view is cross-session

spark.newSession().sql("SELECT \* FROM global\_temp.people").show()

# +----+-------+

# | age| name|

# +----+-------+

# |null|Michael|

# | 30| Andy|

# | 19| Justin|

# +----+-------+

Find full example code at "examples/src/main/python/sql/basic.py" in the Spark repo.

创建数据集

数据集与RDD类似，但是，不是使用Java序列化或Kryo，它们使用专门的编码器来串行化对象以便通过网络进行处理或传输。 虽然编码器和标准序列化都是负责将对象转换成字节的，但编码器是动态生成的代码，并使用允许Spark执行许多操作（如过滤，排序和散列）的格式，而无需将字节反序列化到对象中。

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_5)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_5)

// 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|

// +----+-------+

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

**与RDD进行互操作**

Spark SQL支持将现有RDD转换为Datasets的两种不同方法。 第一种方法使用反射来推断包含特定类型对象的RDD模式。 当您在编写Spark应用程序时已经知道模式，这种基于反射的方法会导致更简洁的代码，并且运行良好。

创建数据集的第二种方法是通过编程接口，允许您构建一个模式，然后将其应用到现有的RDD。 虽然此方法更详细，但它允许您在运行时不知道列及其类型时构造Datasets。

### Inferring the Schema Using Reflection

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_6)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_6)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_6)

Spark SQL可以将Row对象的RDD转换为DataFrame，推断出数据类型。 通过将key / value对的列表作为kwargs传递给Row类来构造行。 此列表的键定义表的列名称，并且通过对整个数据集进行抽样来推断类型，类似于对JSON文件执行的推断。

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

Find full example code at "examples/src/main/python/sql/basic.py" in the Spark repo.

以编程方式指定模式

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_7)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_7)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_7)

**当kwargs的字典(列名)**不能被提前定义时（例如，记录的结构被编码成一个字符串，或者文本数据集将被解析，字段将被不同的用户投射不同），DataFrame可以以编程方式 三步。

1. 从原始RDD创建元组或列表的RDD;
2. 创建由符合在步骤1中创建的RDD中的元组或列表的结构的StructType表示的模式。
3. 通过SparkSession提供的createDataFrame方法将模式应用于RDD。

For example:

# 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. #重新定义列名

**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")

# 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/python/sql/basic.py" in the Spark repo.

**聚合**

内置的DataFrames函数提供常用的聚合，如count（），countDistinct（），avg（），max（），min（）等）。虽然这些函数是为DataFrames设计的，但Spark SQL也具有类型安全的版本 其中一些在Scala和Java中使用强类型数据集。 此外，用户不限于预定义的聚合功能，并且可以自己创建。

### Untyped User-Defined Aggregate Functions

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_8)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_8)

Users have to extend the [UserDefinedAggregateFunction](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.expressions.UserDefinedAggregateFunction) abstract class to implement a custom untyped aggregate function. For example, a user-defined average can look like:

import org.apache.spark.sql.expressions.MutableAggregationBuffer

import org.apache.spark.sql.expressions.UserDefinedAggregateFunction

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

import org.apache.spark.sql.Row

import org.apache.spark.sql.SparkSession

object MyAverage extends UserDefinedAggregateFunction {

// Data types of input arguments of this aggregate function

def inputSchema: StructType = StructType(StructField("inputColumn", LongType) :: Nil)

// Data types of values in the aggregation buffer

def bufferSchema: StructType = {

StructType(StructField("sum", LongType) :: StructField("count", LongType) :: Nil)

}

// The data type of the returned value

def dataType: DataType = DoubleType

// Whether this function always returns the same output on the identical input

def deterministic: Boolean = true

// Initializes the given aggregation buffer. The buffer itself is a `Row` that in addition to

// standard methods like retrieving a value at an index (e.g., get(), getBoolean()), provides

// the opportunity to update its values. Note that arrays and maps inside the buffer are still

// immutable.

def initialize(buffer: MutableAggregationBuffer): Unit = {

buffer(0) = 0L

buffer(1) = 0L

}

// Updates the given aggregation buffer `buffer` with new input data from `input`

def update(buffer: MutableAggregationBuffer, input: Row): Unit = {

if (!input.isNullAt(0)) {

buffer(0) = buffer.getLong(0) + input.getLong(0)

buffer(1) = buffer.getLong(1) + 1

}

}

// Merges two aggregation buffers and stores the updated buffer values back to `buffer1`

def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {

buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)

buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)

}

// Calculates the final result

def evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1)

}

// Register the function to access it

spark.udf.register("myAverage", MyAverage)

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

df.createOrReplaceTempView("employees")

df.show()

// +-------+------+

// | name|salary|

// +-------+------+

// |Michael| 3000|

// | Andy| 4500|

// | Justin| 3500|

// | Berta| 4000|

// +-------+------+

val result = spark.sql("SELECT myAverage(salary) as average\_salary FROM employees")

result.show()

// +--------------+

// |average\_salary|

// +--------------+

// | 3750.0|

// +--------------+

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

### Type-Safe User-Defined Aggregate Functions

User-defined aggregations for strongly typed Datasets revolve around the [Aggregator](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.expressions.Aggregator) abstract class. For example, a type-safe user-defined average can look like:

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_9)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_9)

import org.apache.spark.sql.expressions.Aggregator

import org.apache.spark.sql.Encoder

import org.apache.spark.sql.Encoders

import org.apache.spark.sql.SparkSession

case class Employee(name: String, salary: Long)

case class Average(var sum: Long, var count: Long)

object MyAverage extends Aggregator[Employee, Average, Double] {

// A zero value for this aggregation. Should satisfy the property that any b + zero = b

def zero: Average = Average(0L, 0L)

// Combine two values to produce a new value. For performance, the function may modify `buffer`

// and return it instead of constructing a new object

def reduce(buffer: Average, employee: Employee): Average = {

buffer.sum += employee.salary

buffer.count += 1

buffer

}

// Merge two intermediate values

def merge(b1: Average, b2: Average): Average = {

b1.sum += b2.sum

b1.count += b2.count

b1

}

// Transform the output of the reduction

def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count

// Specifies the Encoder for the intermediate value type

def bufferEncoder: Encoder[Average] = Encoders.product

// Specifies the Encoder for the final output value type

def outputEncoder: Encoder[Double] = Encoders.scalaDouble

}

val ds = spark.read.json("examples/src/main/resources/employees.json").as[Employee]

ds.show()

// +-------+------+

// | name|salary|

// +-------+------+

// |Michael| 3000|

// | Andy| 4500|

// | Justin| 3500|

// | Berta| 4000|

// +-------+------+

// Convert the function to a `TypedColumn` and give it a name

val averageSalary = MyAverage.toColumn.name("average\_salary")

val result = ds.select(averageSalary)

result.show()

// +--------------+

// |average\_salary|

// +--------------+

// | 3750.0|

// +--------------+

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

数据源

Spark SQL支持通过DataFrame界面对各种数据源进行操作。 DataFrame可以使用关系转换操作，也可用于创建临时视图。 将DataFrame注册为临时视图允许您对其数据运行SQL查询。 本节介绍使用Spark数据源加载和保存数据的一般方法，然后介绍可用于内置数据源的特定选项。

通用加载/保存功能

在最简单的形式中，默认数据源（镶木地板，除非由spark.sql.sources.default另有配置）将用于所有操作。

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_10)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_10)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_10)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_10)

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/python/sql/datasource.py" in the Spark repo.

手动指定选项

您还可以手动指定将要使用的数据源以及要传递给数据源的任何其他选项。 数据源由其完全限定名称（即org.apache.spark.sql.parquet）指定，但对于内置源，您还可以使用其短名称（json，parquet，jdbc，orc，libsvm，csv，text）。 可以使用此语法将从任何数据源类型加载的DataFrames转换为其他类型。

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_11)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_11)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_11)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_11)

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/python/sql/datasource.py" in the Spark repo.

直接在文件上运行SQL

而不是使用读取API将文件加载到DataFrame并对其进行查询，还可以使用SQL直接查询该文件。

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_12)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_12)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_12)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_12)

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

Find full example code at "examples/src/main/python/sql/datasource.py" in the Spark repo.

**保存模式**

保存操作可以选择使用SaveMode，它指定如何处理现有数据（如果存在）。 重要的是要意识到这些保存模式不会使用任何锁定，也不是原子的。 另外，当执行覆盖时，数据将在写出新数据之前被删除。

|  |  |  |
| --- | --- | --- |
| Scala/Java | Any Language | Meaning |
| SaveMode.ErrorIfExists (default) | "error" (default) | 如果数据已经存在，则报错 |
| SaveMode.Append | "append" | 将DataFrame保存到数据源时，如果数据/表已存在，则DataFrame的内容将被附加到现有数据中。 |
| SaveMode.Overwrite | "overwrite" | 覆盖已有数据 |
| SaveMode.Ignore | "ignore" | 忽略模式意味着当将DataFrame保存到数据源时，如果数据已经存在，则保存操作预期不会保存DataFrame的内容，并且不更改现有数据。 这与SQL中的CREATE TABLE IF NOT EXISTS类似。 |

**保存到持久表**

DataFrames也可以使用**saveAsTable命令作为持久表保存到Hive metastore中**。请注意，使用**此功能不需要现有的Hive部署。 Spark将为您创建默认的本地Hive转移（使用Derby）**。与createOrReplaceTempView命令不同，**saveAsTable将实现DataFrame的内容，并创建一个指向Hive metastore中的数据的指针。**只要您保持与同一个转移点的连接，即使在您的Spark程序重新启动后，持久性表仍然存在。可以通过调用具有表名称的SparkSession上的表方法来创建持久化表的DataFrame。

**默认情况下，saveAsTable将创建一个“托管表”，**这意味着数据的位置将由转移控制。托管表还会在删除表时自动删除其数据。

目前，saveAsTable不会从DataFrame公开一个支持创建“外部表”的API。但是，通过为DataFrameWriter提供一个路径选项，通过在使用saveAsTable保存表时，以路径作为外部表的键和位置作为值（字符串），可以实现此功能。当外部表被删除时，仅删除其元数据。

从Spark 2.1开始，持久性数据源表将Hive转移中存储每个分区元数据。这带来了几个好处：

* 由于转移只能返回查询的必要分区，因此不再需要将第一个查询上的所有分区发现到表中。
* Hive DDL如ALTER TABLE PARTITION ... SET LOCATION现在可用于使用Datasource API创建的表。

请注意，创建外部数据源表（带有路径选项的表）时，默认情况下不会收集分区信息。 要同步转移中的分区信息，可以调用MSCK维修表。

## Parquet Files

Hive是由许多其他数据处理系统支持的柱状格式。 Spark SQL支持读取和写入Parquet文件，可自动保留原始数据的模式。 当编写Parquet文件时，由于兼容性原因，所有列都将自动转换为可空。

### Loading Data Programmatically

Using the data from the above example:

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_13)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_13)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_13)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_13)
* [**Sql**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_sql_13)

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/python/sql/datasource.py" in the Spark repo.

**分区发现**

**表分区是一种常见的优化方法，用于像Hive这样的系统**。 在分区表中，数据通常存储在不同的目录中，分区列值编码在每个分区目录的路径中。 Parquet数据源现在能够自动发现和推断分区信息。 例如，我们可以使用以下目录结构将所有以前使用的人口数据存储到分区表中，其中包含两个额外的列：性别和国家/地区作为分区列：

path

└── to

└── table

├── gender=male

│   ├── ...

│   │

│   ├── country=US

│   │   └── data.parquet

│   ├── country=CN

│   │   └── data.parquet

│   └── ...

└── gender=female

   ├── ...

   │

   ├── country=US

   │   └── data.parquet

   ├── country=CN

   │   └── data.parquet

   └── ...

通过将path / to / table传递给SparkSession.read.parquet或SparkSession.read.load，Spark SQL将自动从路径中提取分区信息。 现在返回的DataFrame的模式变成：

root

|-- name: string (nullable = true)

|-- age: long (nullable = true)

|-- gender: string (nullable = true)

|-- country: string (nullable = true)

请注意，会自动推断分区列的数据类型。 目前，支持数字数据类型和字符串类型。 有些用户可能不想自动推断分区列的数据类型。 对于这些用例，自动类型推断可以由spark.sql.sources.partitionColumnTypeInference.enabled配置，默认值为true。 当禁用类型推断时，字符串类型将用于分区列。

从Spark 1.6.0开始，默认情况下，分区发现只能找到给定路径下的分区。 对于上述示例，如果用户将path / to / table / gender = male传递给SparkSession.read.parquet或SparkSession.read.load，则性别将不被视为分区列。 如果用户需要指定分区发现应该开始的基本路径，则可以在数据源选项中设置basePath。 例如，当path / to / table / gender = male是数据的路径，用户将basePath设置为path / to / table /时，性别将是一个分区列。

**模式合并**

像ProtocolBuffer，Avro和Thrift一样，**Parquet也支持模式演进**。 用户可以从简单的架构开始，并根据需要逐渐向架构添加更多的列。 这样，用户可能会使用多个Parquet文件，而不同但相互兼容的模式。 Parquet数据源现在能够自动检测这种情况并合并所有这些文件的模式。

由于模式合并是一个相对昂贵的操作，并且在大多数情况下不是必需的，所以默认情况下从1.5.0开始。 你可以启用它

1. 读取Parquet文件时，将数据源选项mergeSchema设置为true（如下面的示例所示）或
2. 将全局SQL选项spark.sql.parquet.mergeSchema设置为true。

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_14)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_14)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_14)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_14)

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)

Find full example code at "examples/src/main/python/sql/datasource.py" in the Spark repo.

当阅读和写入Hive转移Parquet表时，Spark SQL将尝试使用自己的Parquet支持，而不是Hive SerDe来获得更好的性能。 此行为由spark.sql.hive.convertMetastoreParquet配置控制，默认情况下打开。

#### Hive/Parquet Schema Reconciliation

从表格模式处理的角度来说，Hive和Parquet之间有两个关键的区别。

1. Hive不区分大小写，而 Parquet 不是
2. Hive认为所有列可以为空，而Parquet中的可空性是重要的

由于这个原因，当将Hive转移Parquet表转换为Spark SQL Parquet表时，我们必须调整Hive转移模式与Parquet模式。 和解规则是：

1. 在两个模式中具有相同名称的字段必须具有相同的数据类型，而不管可空性。 对帐字段应具有实木复合地板的数据类型，以便可靠性得到尊重。
2. 调和模式正好包含Hive转移模式中定义的那些字段。
   * 仅在Parquet模式中出现的任何字段将被删除在对帐模式中。
   * 只有在Hive转移模式中出现的任何字段才会在协调模式中添加为可空字段。

**元数据刷新**

Spark SQL缓存Parquet元数据以获得更好的性能。 当启用Hive转移Parquet表转换时，这些转换表的元数据也被缓存。 如果这些表由Hive或其他外部工具更新，则需要手动刷新它们以确保一致的元数据。

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_15)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_15)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_15)
* [**Sql**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_sql_15)

# spark is an existing SparkSession

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

### Configuration

可以使用SparkSession上的setConf方法或使用SQL运行SET key = value命令来完成Parquet的配置。

|  |  |  |
| --- | --- | --- |
| Property Name | Default | Meaning |
| spark.sql.parquet.binaryAsString | false | 一些其他Parquet生产系统，特别是Impala，Hive和旧版本的Spark SQL，在写出Parquet模式时，不区分二进制数据和字符串。 该标志告诉Spark SQL将二进制数据解释为字符串以提供与这些系统的兼容性。 |
| spark.sql.parquet.int96AsTimestamp | true | 一些parquet 系统，特别是Impala和Hive，将Timestamp存储到INT96中。 该标志告诉Spark SQL将INT96数据解析为时间戳以提供与这些系统的兼容性。 |
| spark.sql.parquet.cacheMetadata | true | 打开Parquet模式元数据的缓存。 可以加快查询静态数据。 |
| spark.sql.parquet.compression.codec | snappy | 在编写Parquet文件时设置压缩编解码器的使用。 可接受的值包括：未压缩，snappy，gzip，lzo。 |
| spark.sql.parquet.filterPushdown | true | 设置为true时启用实木复合材料滤镜下推优化。 |
| spark.sql.hive.convertMetastoreParquet | true | 当设置为false时，Spark SQL将使用Hive SerDe作为镶木表，而不是内置的支持。 |
| spark.sql.parquet.mergeSchema | false | 当真实时，Parquet数据源合并从所有数据文件收集的模式，否则如果没有可用的摘要文件，则从摘要文件或随机数据文件中挑选模式。 |
| spark.sql.optimizer.metadataOnly | true | 如果为真，则启用使用表的元数据的仅元数据查询优化来生成分区列而不是表扫描。 当扫描的所有列都是分区列并且查询具有满足不同语义的聚合运算符时，它将适用。 |

## JSON Datasets

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_16)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_16)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_16)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_16)
* [**Sql**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_sql_16)

Spark SQL可以自动推断JSON数据集的模式，并将其作为DataFrame加载。 可以使用SparkSession.read.json对JSON文件进行此转换。

请注意，作为json文件提供的文件不是典型的JSON文件。 每行必须包含一个单独的，独立的有效的JSON对象。 有关更多信息，请参阅JSON Lines文本格式，也称为换行符分隔的JSON。 因此，常规的多行JSON文件通常会失败。

# 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/python/sql/datasource.py" in the Spark repo.

## Hive Tables

Spark SQL还支持读取和写入存储在Apache Hive中的数据。但是，由于Hive具有大量依赖关系，因此这些依赖关系不包含在默认Spark分发中。如果在类路径中找到Hive依赖项，Spark将自动加载它们。请注意，这些Hive依赖项也必须存在于所有工作节点上，因为它们将需要访问Hive序列化和反序列化库（SerDes），以便访问Hive中存储的数据。

Hive的配置是通过在conf /中放置您的hive-site.xml，core-site.xml（用于安全配置）和hdfs-site.xml（用于HDFS配置）文件来完成的。

使用Hive时，必须使用Hive支持实例化SparkSession，包括连接到持续的Hive转移，支持Hive serdes和Hive用户定义的功能。没有现有Hive部署的用户仍然可以启用Hive支持。当hive-site.xml未配置时，上下文会自动在当前目录中创建metastore\_db，并创建由spark.sql.warehouse.dir配置的目录，该目录默认为Spark应用程序当前目录中的spark-warehouse目录开始了请注意，自从2.0.0以来，hive-site.xml中的hive.metastore.warehouse.dir属性已被弃用。而是使用spark.sql.warehouse.dir来指定仓库中数据库的默认位置。您可能需要向启动Spark应用程序的用户授予写权限。

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_17)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_17)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_17)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_17)

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([Record(i, "val\_" + str(i)) for i in 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/python/sql/hive.py" in the Spark repo.

与不同版本的Hive Metastore进行交互

Spark SQL的Hive支持最重要的部分之一是与Hive metastore进行交互，这使得Spark SQL能够访问Hive表的元数据。 从Spark 1.4.0开始，使用Spark SQL的单一二进制构建可以使用下面描述的配置来查询不同版本的Hive转移。 请注意，独立于用于与转移对话的Hive版本，内部Spark SQL将针对Hive 1.2.1进行编译，并使用这些类进行内部执行（serdes，UDF，UDAF等）。

以下选项可用于配置用于检索元数据的Hive版本：

需要补的内容http://spark.apache.org/docs/latest/sql-programming-guide.html#hive-tables

**JDBC到其他数据库**

Spark SQL还包括可以使用JDBC从其他数据库读取数据的数据源。此功能应优于使用JdbcRDD。这是因为结果作为DataFrame返回，并且可以轻松地在Spark SQL中处理或与其他数据源连接。 JDBC数据源也比Java或Python更容易使用，因为它不需要用户提供ClassTag。 （请注意，这不同于Spark SQL JDBC服务器，允许其他应用程序使用Spark SQL运行查询）。

要开始使用，您需要在Spark类路径中包含特定数据库的JDBC驱动程序。例如，要从Spark Shell连接到postgres，您将运行以下命令：

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

来自远程数据库的表可以使用Data Sources API作为DataFrame或Spark SQL临时视图进行加载。用户可以在数据源选项中指定JDBC连接属性。用户和密码通常作为登录数据源的连接属性提供。除了连接属性之外，Spark还支持以下不区分大小写的选项：

|  |  |
| --- | --- |
| Property Name | Meaning |
| url | 要连接的JDBC URL。 源特定的连接属性可以在URL中指定。 例如jdbc：postgresql：// localhost / test？user = fred＆password = secret |
| dbtable | 应该读的JDBC表。 请注意，可以使用在SQL查询的FROM子句中有效的任何内容。 例如，您可以使用括号中的子查询代替完整表。 |
| driver | 用于连接到此URL的JDBC驱动程序的类名。 |
| partitionColumn, lowerBound, upperBound, numPartitions | 如果指定了这些选项，则必须指定这些选项。 它们描述了如何在从多个工作人员并行阅读时分割表。 partitionColumn必须是有问题的表中的数字列。 请注意，lowerBound和upperBound仅用于决定分区的大小，而不是用于过滤表中的行。 因此，表中的所有行将被分区并返回。 此选项仅适用于阅读。 |
| fetchsize | JDBC提取大小，用于确定每次往返行程的行数。 这可以帮助JDBC驱动程序的性能降低，这些驱动程序默认具有较低的提取大小（例如，具有10行的Oracle）。 此选项仅适用于阅读。 |
| batchsize | JDBC批量大小，用于确定每往返行数要插入的行数。 这可以帮助JDBC驱动程序的性能。 此选项仅适用于写作。 默认为1000。 |
| isolationLevel | 事务隔离级别，适用于当前连接。 它可以是NONE，READ\_COMMITTED，READ\_UNCOMMITTED，REPEATABLE\_READ或SERIALIZABLE之一，对应于JDBC连接对象定义的标准事务隔离级别，默认为READ\_UNCOMMITTED。 此选项仅适用于写作。 请参考java.sql.Connection中的文档。 |
| truncate | 这是一个与JDBC相关的选项。 启用SaveMode.Overwrite时，此选项会导致Spark截断现有表，而不是删除并重新创建。 这可以更有效，并且防止表元数据（例如，索引）被移除。 然而，在某些情况下，例如当新数据具有不同的模式时，它将不起作用。 它默认为false。 此选项仅适用于写。 |
| createTableOptions | 这是一个与JDBC相关的选项。 如果指定，此选项允许在创建表时设置特定于数据库的表和分区选项（例如，CREATE TABLE t（名称字符串）ENGINE = InnoDB。）。 此选项仅适用于写作。 |

* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_18)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_18)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_18)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_18)
* [**Sql**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_sql_18)

# Note: JDBC loading and saving can be achieved via either the load/save or jdbc methods

# Loading data from a JDBC source

jdbcDF = spark.read \

.format("jdbc") \

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

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

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

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

.load()

jdbcDF2 = spark.read \

.jdbc("jdbc:postgresql:dbserver", "schema.tablename",

properties={"user": "username", "password": "password"})

# Saving data to a JDBC source

jdbcDF.write \

.format("jdbc") \

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

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

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

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

.save()

jdbcDF2.write \

.jdbc("jdbc:postgresql:dbserver", "schema.tablename",

properties={"user": "username", "password": "password"})

Find full example code at "examples/src/main/python/sql/datasource.py" in the Spark repo.

**故障排除**

JDBC驱动程序类必须对客户端会话和所有执行程序上的原始类加载器可见。 这是因为Java的DriverManager类执行安全检查，导致它忽略原始类加载器不可见的所有驱动程序，当打开连接时。 一个方便的方法是修改所有工作节点上的compute\_classpath.sh以包含您的驱动程序JAR。

一些数据库，例如H2，将所有名称转换为大写。 您需要使用大写字母来引用Spark SQL中的这些名称。

性能调优

对于某些工作负载，可以通过缓存内存中的数据或打开一些实验选项来提高性能。

缓存内存中的数据

Spark SQL可以通过调用spark.catalog.cacheTable（“tableName”）或dataFrame.cache（）来使用内存中的列格式来缓存表。 然后，Spark SQL将仅扫描所需的列，并将自动调整压缩以最小化内存使用量和GC压力。 您可以调用spark.catalog.uncacheTable（“tableName”）从内存中删除该表。

内存缓存的配置可以使用SparkSession上的setConf方法或使用SQL运行SET key = value命令来完成。

|  |  |  |
| --- | --- | --- |
| Property Name | Default | Meaning |
| spark.sql.inMemoryColumnarStorage.compressed | true | 当设置为true时，Spark SQL将根据数据的统计信息为每个列自动选择一个压缩编解码器。 |
| spark.sql.inMemoryColumnarStorage.batchSize | 10000 | 控制批量的柱状缓存。 较大的批量大小可以提高内存利用率和压缩率，但是在缓存数据时会冒出OOM风险。 |

其他配置选项

以下选项也可用于调整查询执行的性能。 这些选项有可能在将来的版本中被废弃，因为更多的优化是自动执行的。

|  |  |  |
| --- | --- | --- |
| Property Name | Default | Meaning |
| spark.sql.files.maxPartitionBytes | 134217728 (128 MB) | 读取文件时，最多可打包到单个分区中的字节数。 |
| spark.sql.files.openCostInBytes | 4194304 (4 MB) | 按照字节数来衡量的打开文件的估计费用可以在同一时间进行扫描。 将多个文件放入分区时使用。 最好过度估计，那么具有小文件的分区将比具有较大文件的分区（这是首先计划的）更快。 |
| spark.sql.broadcastTimeout | 300 | 广播连接广播等待时间超时（秒） |
| spark.sql.autoBroadcastJoinThreshold | 10485760 (10 MB) | 配置执行连接时将广播给所有工作节点的表的最大大小（以字节为单位）。 通过将此值设置为-1可以禁用广播。 请注意，目前的统计信息仅支持Hive Metastore表，其中运行了命令ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan。 |
| spark.sql.shuffle.partitions | 200 | 配置混合连接或聚合数据时使用的分区数。 |

**分布式SQL引擎**

Spark SQL也可以充当使用其JDBC / ODBC或命令行界面的分布式查询引擎。 在这种模式下，终端用户或应用程序可以直接与Spark SQL交互运行SQL查询，而无需编写任何代码。

**运行Thrift JDBC / ODBC服务器**

此处实现的Thrift JDBC / ODBC服务器对应于Hive 1.2中的HiveServer2。您可以使用Spark或Hive 1.2.1附带的直线脚本测试JDBC服务器。

要启动JDBC / ODBC服务器，请在Spark目录中运行以下命令：

./sbin/start-thriftserver.sh

此脚本接受所有bin / spark-submit命令行选项，以及--hiveconf选项来指定Hive属性。 您可以运行./sbin/start-thriftserver.sh --help查看所有可用选项的完整列表。 默认情况下，服务器侦听localhost：10000。 您可以通过环境变量覆盖此行为，即：

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

**连接JDBC服务器**

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

Beeline将要求您提供用户名和密码。 在非安全模式下，只需输入机器上的用户名和空白密码即可。 对于安全模式，请按照直线文档中的说明进行操作。

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/:

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

hive.server2.thrift.http.port - HTTP port number to 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>

**运行Spark SQL CLI**

Spark SQL CLI是在本地模式下运行Hive转移服务并执行从命令行输入的查询的便捷工具。请注意，Spark SQL CLI不能与Thrift JDBC服务器通信。

要启动Spark SQL CLI，请在Spark目录中运行以下命令：

./bin/spark-sql

Hive的配置是通过将hive-site.xml，core-site.xml和hdfs-site.xml文件放在conf /中完成的。您可以运行./bin/spark-sql --help获取所有可用选项的完整列表。

移民指南

**从Spark SQL 2.0升级到2.1**

数据源表现在在Hive转移中存储分区元数据。这意味着Hive DDL如ALTER TABLE PARTITION ... SET LOCATION现在可用于使用Datasource API创建的表。

传统的数据源表可以通过MSCK REPAIR TABLE命令迁移到这种格式。建议迁移遗留表以利用Hive DDL支持和改进的规划性能。

要确定表是否已迁移，请在表上发出DESCRIBE FORMATTED时查找PartitionProvider：Catalog属性。

更改INSERT OVERWRITE TABLE ... PARTITION ...数据源表的行为。

在先前的Spark版本中，即使给定了分区规范，INSERT OVERWRITE也覆盖了整个Datasource表。现在只有匹配规范的分区被覆盖。

请注意，这仍然与Hive表的行为不同，Hive表仅覆盖与新插入数据重叠的分区。

从Spark SQL 1.6升级到2.0

SparkSession现在是Spark的新入口点，可以替换旧的SQLContext和HiveContext。请注意，旧的SQLContext和HiveContext保持向后兼容。可以从SparkSession访问新的目录界面 - 数据库上的现有API和诸如listTable，createExternalTable，dropTempView，cacheTable之类的表访问。

数据集API和DataFrame API是统一的。在Scala中，DataFrame成为Dataset [Row]的类型别名，而Java API用户必须使用Dataset <Row>替换DataFrame。类型化变换（例如，map，filter和groupByKey）和无类型转换（例如select和groupBy）在Dataset类上都可用。由于Python和R中的编译时类型安全性不是语言特性，因此Dataset的概念不适用于这些语言的API。相反，DataFrame仍然是主要的编程抽象，它类似于这些语言中的单节点数据帧概念。

数据集和DataFrame API unionAll已被弃用并替换为联合

数据集和DataFrame API爆炸已被弃用，或者使用functions.explode（）与select或flatMap

数据集和DataFrame API registerTempTable已被弃用，并被createOrReplaceTempView替换

更改CREATE TABLE ... Hive表的LOCATION行为。

从Spark 2.0，CREATE TABLE ... LOCATION等同于CREATE EXTERNAL TABLE ... LOCATION，以防止意外丢弃用户提供的位置中的现有数据。这意味着，在用户指定位置的Spark SQL中创建的Hive表始终是Hive外部表。删除外部表将不会删除数据。用户不能指定Hive管理表的位置。请注意，这与Hive行为不同。

因此，这些表上的DROP TABLE语句将不会删除数据。

从Spark SQL 1.5升级到1.6

从Spark 1.6，默认情况下，Thrift服务器以多会话模式运行。这意味着每个JDBC / ODBC连接都拥有自己的SQL配置和临时功能注册表的副本。缓存表仍然是共享的。如果您希望以旧的单会话模式运行Thrift服务器，请将选项spark.sql.hive.thriftServer.singleSession设置为true。您可以将此选项添加到spark-defaults.conf，或者通过--conf将其传递给start-thriftserver.sh：

./sbin/start-thriftserver.sh \

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

     ...

自1.6.1以来，在sparkR中的COLOL方法支持向新的列添加新的列或替换DataFrame同名的现有列。

从Spark 1.6，LongType转换为TimestampType，期望秒而不是微秒。这种更改是为了匹配Hive 1.2的行为，以便从数字类型进行更一致的类型转换到TimestampType。详见SPARK-11724。

**从Spark SQL 1.4升级到1.5**

默认情况下，使用手动管理内存（Tungsten）进行优化的执行，以及表达式求值的代码生成。通过将spark.sql.tungsten.enabled设置为false，可以禁用这些功能。

默认情况下，Parquet模式合并不再启用。可以通过将spark.sql.parquet.mergeSchema设置为true来重新启用。

字符串到python列中的分辨率现在支持使用点（。）来限定列或访问嵌套值。例如df ['table.column.nestedField']。但是，这意味着如果您的列名称包含任何点，您现在必须使用反引号（例如，table.`column.with.dots`.nested）来转义它们。

内存中柱状存储分区修剪默认情况下处于打开状态。将spark.sql.inMemoryColumnarStorage.partitionPruning设置为false可以禁用它。

不再支持无限精度小数列，而不是Spark SQL强制执行最大精度38.当从BigDecimal对象推断模式时，现在使用（38,18）的精度。当DDL中没有指定精度时，默认值保持为十进制（10，0）。

时间戳现在以1us的精度存储，而不是1ns

在sql方言中，浮点数现在被解析为十进制。 HiveQL解析保持不变。

SQL / DataFrame函数的规范名称现在是小写（例如sum和SUM）。

JSON数据源不会自动加载由其他应用程序创建的新文件（即未通过Spark SQL插入数据集的文件）。对于JSON持久表（即表的元数据存储在Hive Metastore中），用户可以使用REFRESH TABLE SQL命令或HiveContext的refreshTable方法将这些新文件包含到表中。对于表示JSON数据集的DataFrame，用户需要重新创建DataFrame，新的DataFrame将包含新文件。

pySpark中的DataFrame.withColumn方法支持添加新列或替换同名的现有列。

从Spark SQL 1.3升级到1.4

**DataFrame数据读写器接口**

基于用户反馈，我们创建了一个新的更流畅的API，用于读取（SQLContext.read）中的数据并写入数据（DataFrame.write），并且废弃了旧的API（例如，SQLContext.parquetFile，SQLContext.jsonFile）。

请参阅SQLContext.read（Scala，Java，Python）和DataFrame.write（Scala，Java，Python）的API文档更多信息。

**DataFrame.groupBy保留分组列**

基于用户反馈，我们更改了DataFrame.groupBy（）。agg（）的默认行为，以便在生成的DataFrame中保留分组列。为了保持1.3中的行为，将spark.sql.retainGroupColumns设置为false。

斯卡拉

Java的

将pyspark.sql.functions导入为func

＃在1.3.x中，为了使分组列“部门”出现，

＃它必须明确地包含在agg函数调用的一部分。

df.groupBy（“department”）。agg（df [“department”]，func.max（“age”），func.sum（“expenses”））

＃在1.4+以上，分组列“部门”被自动包含。

df.groupBy（“department”）。agg（func.max（“age”），func.sum（“expenses”））

＃恢复为1.3.x行为（不保留分组列）：

sqlContext.setConf（“spark.sql.retainGroupColumns”，“false”）

DataFrame.withColumn上的行为更改

在1.4之前，DataFrame.withColumn（）只支持添加列。即使可能存在任何现有的相同名称的列，该列也将始终作为新的列添加到结果DataFrame中的指定名称。自1.4以来，DataFrame.withColumn（）支持从所有现有列的名称添加不同名称的列或替换同名的现有列。

请注意，此更改仅适用于Scala API，而不适用于PySpark和SparkR。

从Spark SQL 1.0-1.2升级到1.3

在Spark 1.3中，我们从Spark SQL中删除了“Alpha”标签，并且其中一部分清除了可用的API。从Spark 1.3起，Spark SQL将提供与1.X系列中其他版本的二进制兼容性。此兼容性保证不包括明确标记为不稳定的API（即DeveloperAPI或实验）。

将SchemaRDD重命名为DataFrame

升级到Spark SQL 1.3时，用户会注意到的最大变化是将SchemaRDD重命名为DataFrame。这主要是因为DataFrames不再直接从RDD继承，而是通过自己的实现来提供RDD提供的大部分功能。 DataFrames仍然可以通过调用.rdd方法转换为RDD。

在Scala中，从SchemaRDD到DataFrame有一个类型别名，可以为某些用例提供源兼容性。仍然建议用户更新他们的代码以使用DataFrame。 Java和Python用户将需要更新其代码。

统一Java和Scala API

在Spark 1.3之前，有独立的Java兼容类（JavaSQLContext和JavaSchemaRDD）镜像了Scala API。在Spark 1.3中，Java API和Scala API已被统一。任何一种语言的用户应使用SQLContext和DataFrame。通常，这些类尝试使用可从两种语言使用的类型（即Array而不是语言特定的集合）。在某些情况下，不存在通用类型（例如，传递闭包或地图）会改用函数重载。另外，Java特定类型的API已被删除。 Scala和Java的用户应该使用org.apache.spark.sql.types中存在的类以编程方式描述模式。隐式转换的解除和dsl包的删除（仅限Scala）Spark 1.3之前的许多代码示例开始使用导入sqlContext.\_，它将所有从sqlContext的函数带入范围。在Spark 1.3中，我们隔离了将RDD转换为DataFrames的隐式转换到SQLContext内的对象。用户现在应该编写导入sqlContext.implicits .\_。此外，隐式转换现在只能使用方法toDF来增加由产品（即案例类或元组）组成的RDD，而不是自动应用。当使用DSL内部的函数（现在替换为DataFrame API）用户用于导入org.apache.spark.sql.catalyst.dsl。应该使用公共数据框函数API：import org.apache.spark.sql.functions .\_。删除DataType（仅限Scala）的org.apache.spark.sql中的类型别名Spark 1.3将删除类型别名存在于DataType的基本sql包中。用户应该将org.apache.spark.sql.typesUDF注册中的类导入到sqlContext.udf（Java＆Scala）中注册的用于在DataFrame DSL或SQL中使用的UDF的函数已被移入udf对象在SQLContext.ScalaJavasqlContext.udf.register（“strLen”，（s：String）=> s.length（））Python UDF注册是不变的.Python DataTypes不再单调在Python中使用DataTypes时，您将需要构造它们即StringType（）），而不是引用单例。

**与Apache Hive的兼容性**

Spark SQL旨在与Hive Metastore，SerDes和UDF兼容。 目前，Hive SerDes和UDF基于Hive 1.2.1，Spark SQL可以连接到Hive Metastore的不同版本（从0.12.0到1.2.1），另请参阅[与Hive Metastore的不同版本进行交互]（＃interactioning-与-不同版本-的蜂箱 - metastore））。

部署在现有的Hive仓库

Spark SQL Thrift JDBC服务器设计为与现有Hive安装兼容的“开箱即用”。 您不需要修改现有的Hive Metastore或更改表的数据放置或分区。

### 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<>

不支持的Hive功能

以下是我们还不支持的Hive功能列表。这些功能大部分在Hive部署中很少使用。

主要蜂巢特色

具有桶的表：bucket是Hive表分区中的哈希分区。 Spark SQL尚不支持Bucket。

神秘的蜂巢特色

UNION类型

独特的加入

列统计信息收集：Spark SQL不会随身携带扫描来收集列统计信息，只支持填充蜂巢转移的sizeInBytes字段。

蜂巢输入/输出格式

CLI的文件格式：对于返回CLI的结果，Spark SQL仅支持TextOutputFormat。

Hadoop档案

蜂巢优化

Spark中尚未包含少量Hive优化。由于Spark SQL的内存计算模型，其中一些（如索引）不太重要。其他的是为未来版本的Spark SQL开发的。

块级位图索引和虚拟列（用于构建索引）

自动确定连接和groupbys的reducer数：目前在Spark SQL中，您需要使用“SET spark.sql.shuffle.partitions = [num\_tasks];”来控制随机播放的并行度。

仅Meta数据查询：对于只能使用元数据回答的查询，Spark SQL仍然启动计算结果的任务。

倾斜数据标志：Spark SQL不遵循Hive中的偏斜数据标志。

STREAMTABLE提示连接：Spark SQL不遵循STREAMTABLE提示。

合并查询结果的多个小文件：如果结果输出包含多个小文件，则Hive可以选择将小文件合并为较少的大文件，以避免溢出HDFS元数据。 Spark SQL不支持。

# 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 null values. 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.
* [**Scala**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_scala_21)
* [**Java**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_java_21)
* [**Python**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_python_21)
* [**R**](http://spark.apache.org/docs/latest/sql-programming-guide.html#tab_r_21)

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

from pyspark.sql.types import \*

|  |  |  |
| --- | --- | --- |
| Data type | Value type in Python | API to access or create a data type |
| **ByteType** | int or long  **Note:** Numbers will be converted to 1-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -128 to 127. | ByteType() |
| **ShortType** | int or long  **Note:** Numbers will be converted to 2-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -32768 to 32767. | ShortType() |
| **IntegerType** | int or long | IntegerType() |
| **LongType** | long  **Note:** Numbers will be converted to 8-byte signed integer numbers at runtime. Please make sure that numbers are within the range of -9223372036854775808 to 9223372036854775807. Otherwise, please convert data to decimal.Decimal and use DecimalType. | LongType() |
| **FloatType** | float  **Note:** Numbers will be converted to 4-byte single-precision floating point numbers at runtime. | FloatType() |
| **DoubleType** | float | DoubleType() |
| **DecimalType** | decimal.Decimal | DecimalType() |
| **StringType** | string | StringType() |
| **BinaryType** | bytearray | BinaryType() |
| **BooleanType** | bool | BooleanType() |
| **TimestampType** | datetime.datetime | TimestampType() |
| **DateType** | datetime.date | DateType() |
| **ArrayType** | list, tuple, or array | ArrayType(*elementType*, [*containsNull*])  **Note:** The default value of *containsNull* is *True*. |
| **MapType** | dict | MapType(*keyType*, *valueType*, [*valueContainsNull*])  **Note:** The default value of *valueContainsNull* is *True*. |
| **StructType** | list or tuple | StructType(*fields*)  **Note:** *fields* is a Seq of StructFields. Also, two fields with the same name are not allowed. |
| **StructField** | The value type in Python of the data type of this field (For example, Int for a StructField with the data type IntegerType) | StructField(*name*, *dataType*, [*nullable*])  **Note:** The default value of *nullable* is *True*. |

## 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:

* 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.