»

Getting Started

Starting Point: SparkSession

Scala Java Python R

The entry point into all functionality in Spark is the `SparkSession` class. To create a basic `SparkSession`, just use `SparkSession.builder()`:

```
import org.apache.spark.sql.SparkSession;

SparkSession spark = SparkSession
  .builder()
  .appName("Java Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate();
```

Find full example code at "examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java" 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.

Creating DataFrames

Scala Java Python R

With a `SparkSession`, applications can create DataFrames from an existing `RDD`, from a Hive table, or from Spark data sources.

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

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

Dataset<Row> 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/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java" in the Spark repo.

Untyped Dataset Operations (aka DataFrame Operations)

DataFrames provide a domain-specific language for structured data manipulation in Scala, Java, Python and R.

As mentioned above, in Spark 2.0, DataFrames are just Dataset of `Row`s 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.

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

Scala Java Python R

»

```
// col("...") is preferable to df.col("...")
import static org.apache.spark.sql.functions.col;

// 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(col("name"), col("age").plus(1)).show();
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter(col("age").gt(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/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java" 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.

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.

Running SQL Queries Programmatically

Scala Java Python R

The `sql` function on a `SparkSession` enables applications to run SQL queries programmatically and returns the result as a `Dataset<Row>`.

»

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

// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people");

Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

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

Global Temporary View

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view. Global temporary view is tied to a system preserved database `global_temp`, and we must use the qualified name to refer it, e.g. `SELECT * FROM global_temp.view1`.

Scala   Java   Python   SQL

```
// 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/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java" in the Spark repo.

Creating Datasets

Datasets are similar to RDDs, however, instead of using Java serialization or Kryo they use a specialized 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.

Scala   Java

»

```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 long age;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public long getAge() {
    return age;
  }

  public void setAge(long 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<Long> longEncoder = Encoders.LONG();
Dataset<Long> primitiveDS = spark.createDataset(Arrays.asList(1L, 2L, 3L), longEncoder);
Dataset<Long> transformedDS = primitiveDS.map(
    (MapFunction<Long, Long>) value -> value + 1L,
    longEncoder);
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/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java" in the Spark repo.

Interoperating with RDDs

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.

Inferring the Schema Using Reflection

Scala Java Python

»

Spark SQL supports automatically converting an RDD of JavaBeans into a DataFrame. The `BeanInfo`, obtained using reflection, defines the schema of the table. Currently, Spark SQL does not support JavaBeans that contain `Map` field(s). Nested JavaBeans and `List` or `Array` fields are supported though. You can create a JavaBean by creating a class that implements Serializable and has getters and setters for all of its fields.

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;
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;

// Create an RDD of Person objects from a text file
JavaRDD<Person> peopleRDD = spark.read()
  .textFile("examples/src/main/resources/people.txt")
  .javaRDD()
  .map(line -> {
    String[] parts = line.split(",");
    Person person = new Person();
    person.setName(parts[0]);
    person.setAge(Integer.parseInt(parts[1].trim()));
    return person;
  });

// Apply a schema to an RDD of JavaBeans to get a DataFrame
Dataset<Row> peopleDF = spark.createDataFrame(peopleRDD, Person.class);
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people");

// SQL statements can be run by using the sql methods provided by spark
Dataset<Row> teenagersDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19");

// The columns of a row in the result can be accessed by field index
Encoder<String> stringEncoder = Encoders.STRING();
Dataset<String> teenagerNamesByIndexDF = teenagersDF.map(
    (MapFunction<Row, String>) row -> "Name: " + row.getString(0),
    stringEncoder);
teenagerNamesByIndexDF.show();
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// or by field name
Dataset<String> teenagerNamesByFieldDF = teenagersDF.map(
    (MapFunction<Row, String>) row -> "Name: " + row.<String>getAs("name"),
    stringEncoder);
teenagerNamesByFieldDF.show();
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

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

Programmatically Specifying the Schema

Scala Java Python

When JavaBean 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 `Dataset<Row>` can be created programmatically with three steps.

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

For example:

»

```java
import java.util.ArrayList;
import java.util.List;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;

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

import org.apache.spark.sql.types.DataTypes;
import org.apache.spark.sql.types.StructField;
import org.apache.spark.sql.types.StructType;

// Create an RDD
JavaRDD<String> peopleRDD = spark.sparkContext()
  .textFile("examples/src/main/resources/people.txt", 1)
  .toJavaRDD();

// The schema is encoded in a string
String schemaString = "name age";

// Generate the schema based on the string of schema
List<StructField> fields = new ArrayList<>();
for (String fieldName : schemaString.split(" ")) {
  StructField field = DataTypes.createStructField(fieldName, DataTypes.StringType, true);
  fields.add(field);
}
StructType schema = DataTypes.createStructType(fields);

// Convert records of the RDD (people) to Rows
JavaRDD<Row> rowRDD = peopleRDD.map((Function<String, Row>) record -> {
  String[] attributes = record.split(",");
  return RowFactory.create(attributes[0], attributes[1].trim());
});

// Apply the schema to the RDD
Dataset<Row> peopleDataFrame = spark.createDataFrame(rowRDD, schema);

// Creates a temporary view using the DataFrame
peopleDataFrame.createOrReplaceTempView("people");

// SQL can be run over a temporary view created using DataFrames
Dataset<Row> 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
Dataset<String> namesDS = results.map(
    (MapFunction<Row, String>) row -> "Name: " + row.getString(0),
    Encoders.STRING());
namesDS.show();
// +-------------+
// |        value|
// +-------------+
// |Name: Michael|
// |   Name: Andy|
// | Name: Justin|
// +-------------+
```

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

Scalar Functions

Scalar functions are functions that return a single value per row, as opposed to aggregation functions, which return a value for a group of rows. Spark SQL supports a variety of Built-in Scalar Functions. It also supports User Defined Scalar Functions.

Aggregate Functions

Aggregate functions are functions that return a single value on a group of rows. The Built-in Aggregation Functions provide common aggregations such as `count()`, `count_distinct()`, `avg()`, `max()`, `min()`, etc. Users are not limited to the predefined aggregate functions and can create their own. For more details about user defined aggregate functions, please refer to the documentation of User Defined Aggregate Functions.

Spark SQL Guide