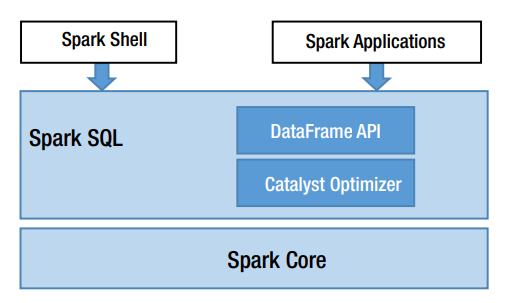
# Spark DataFrame and SQL

The RDD was the initial core programming abstraction when Spark was introduced to the world in 2012

In Spark 1.6, a new programming abstraction, called Structured APIs, was introduced

**Spark SQL components:**



The Spark SQL module consists of two parts:

1. Representation of the Structure APIs, called DataFrames and Datasets
2. The second part of the Spark SQL module is the Catalyst optimizer, which does all the hard work behind the scenes to make your life easier and to speed up your data processing logic

**Schema:** Defines the structure of the data in the form of column names and associated data types

Structured data is often captured in:

* Text based: csv, XML, JSON
* Binary based: ORC, parquet, and Avro

# DataFrame vs Dataset:

# DataFrames:

A DataFrame is an immutable, distributed collection of data that is organized into rows, where each one consists a set of columns and each column has a name and an associated type (distributed collection of data has a structure defined by a schema)

## Creating DataFrames

There are many ways to create a DataFrame; one common thing among them is the need to provide a schema, either implicitly or explicitly

### Creating DataFrames from RDDs

Create an RDD with two columns of the integer type

Call the toDF implicit function to convert an RDD to a DataFrame using the specified column names

The column types are inferred from the data in the RDD

import scala.util.Random

val rdd = spark.sparkContext.parallelize(1 to 10).map(x => (x,Random.nextInt(100)\* x))

val kvDF = rdd.toDF("key","value")

**kvDF: org.apache.spark.sql.DataFrame = [key: int, value: int]**

kvDF.printSchema

**root**

**|-- key: integer (nullable = false)**

**|-- value: integer (nullable = false)**

kvDF.show(5)

**+---+-----+**

**|key|value|**

**+---+-----+**

**| 1| 65|**

**| 2| 88|**

**| 3| 30|**

**| 4| 272|**

**| 5| 425|**

**+---+-----+**

### Creating DF by specifying an RDD with a schema:

* Import spark sql types(DF data types)
* Import spark sql Row
* Create schema (column name and type)
* Using CreateDataFrame(RDD, schema-name)

import org.apache.spark.sql.Row

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

val peopleRDD = spark.sparkContext.parallelize(Array(

Row(1L, "JohnDoe", 30L),

Row(2L, "Mary Jane", 25L)))

val schema = StructType(Array(StructField("id", LongType, true),

StructField("name", StringType, true),

StructField("salary", LongType, true)))

val peopleDF = spark.createDataFrame(peopleRDD, schema)

peopleDF.printSchema

root

|-- id: long (nullable = true)

|-- name: string (nullable = true)

|-- salary: long (nullable = true)

peopleDF.show()

+---+---------+------+

| id| name|salary|

+---+---------+------+

| 1| JohnDoe| 30|

| 2|Mary Jane| 25|

+---+---------+------+

**Schema template: StructType(Array(StructField(),StructField(),StructField())**

Each **StructField** object has three pieces of information: name, type, and whether the value is nullable

Below is the table Spark type mapping with Scala Type

|  |  |
| --- | --- |
| **Data Type** | **Scala Type** |
| BooleanType | Boolean |
| ByteType | Byte |
| ShortType | Short |
| IntegerType | Int |
| LongType | Long |
| FloatType | Float |
| DoubleType | Double |
| DecimalType | java.math.BigDecial |
| StringType | String |
| BinaryType | Array[Byte] |
| TimestampType | java.sql.Timestamp |
| DateType | java.sql.Date |
| ArrayType | scala.collection.Seq |
| MapType | scala.collection.Map |
| StructType | org.apache.spark.sql.Row |

### Converting a Collection Tuple to a DataFrame:

val movies = Seq(("Damon, Matt", "The Bourne Ultimatum", 2007),

("Damon, Matt", "Good Will Hunting", 1997))

movies: Seq[(String, String, Int)] = List((Damon, Matt,The Bourne Ultimatum,2007), (Damon, Matt,Good Will Hunting,1997))

//Converting RDD to DF and adding column names

val moviesDF = movies.toDF("actor","title","year")

moviesDF: org.apache.spark.sql.DataFrame = [actor: string, title: string ... 1 more field]

moviesDF.show()

+-----------+--------------------+----+

| actor| title|year|

+-----------+--------------------+----+

|Damon, Matt|The Bourne Ultimatum|2007|

|Damon, Matt| Good Will Hunting|1997|

+-----------+--------------------+----+

### Creating Dataframes using case class:

This method is same as creating DF with tuple type, but instead of giving column names, CASE CLASS gives the column names and data types automatically to DF

val infile = sc.textFile("file:/databricks/driver/auctiondata")

val infileRDD = infile.map(x=>x.split("~"))

// create an RDD of Auction objects

val ebay = infileRDD.map(p=>Auction(p(0), p(1).toFloat, p(2).toFloat,p(3), p(4).toInt, p(5).toFloat, p(6).toFloat, p(7), p(8).toInt))

// change ebay RDD of Auction objects to a DataFrame

val auction = spark.createDataFrame(ebay)🡪 **No need to pass schema**

auction.printSchema

|-- auctionid: string (nullable = true)

|-- bid: float (nullable = false)

|-- bidtime: float (nullable = false)

|-- bidder: string (nullable = true)

|-- bidderrate: integer (nullable = true)

|-- openbid: float (nullable = false)

|-- price: float (nullable = false)

|-- item: string (nullable = true)

|-- daystolive: integer (nullable = true)

## Creating DataFrames from Data Sources:

Spark SQL supports six built-in data sources using DataFrameReader and DataFrameWriter

1. Cvs - spark.read.csv()
2. Json - spark.read.json()
3. ORC - spark.read.orc()
4. Text - spark.read.text()
5. Parquet - spark.read.parquet()
6. Jdbc - spark.read.jdbc()

Common Pattern for Interacting with DataFrameReader

spark.read.format(...).option("key", value").schema(...).load()

Where format is mandatory

### Text File:

val textFile = spark.read.text("README.md")

### CSV File:

Csv option:

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** | **Value** | **Default** | **Explanation** |
| sep | Single character | comma (,) | This is a single-character value used as a delimiter for each colum |
| header | true,false | FALSE | If the value is true, it means the first line in the file represents the column name |
| escape | Any character | \ | This is the character to use to escape the character in the column value that is the same as sep |
| inferSchema | true, false | true, false | This specifies whether Spark should try to infer the column type based on column value. |

Specifying the header and inferSchema options as true won’t require you to specify a schema. Otherwise, you need to define a schema by hand or programmatically create it and pass it into the schema function. If the inferSchema option is false and no schema is provided, Spark will assume the data type of all the columns to be the string type

val moviesCSV = spark.read.format("csv")

.option("sep",",")

.option("header","true")

.option("inferSchema","true")

.load("file:/databricks/driver/movies.csv")

//https://raw.githubusercontent.com/Apress/beginning-apache-spark-2/master/chapter4/data/movies/movies.csv

moviesCSV.printSchema

|-- actor: string (nullable = true)

|-- title: string (nullable = true)

|-- year: integer (nullable = true)

moviesCSV.show(5)

+-----------------+-------------+----+

| actor| title|year|

+-----------------+-------------+----+

|McClure, Marc (I)|Freaky Friday|2003|

|McClure, Marc (I)| Coach Carter|2005|

|McClure, Marc (I)| Superman II|1980|

|McClure, Marc (I)| Apollo 13|1995|

|McClure, Marc (I)| Superman|1978|

+-----------------+-------------+----+

**manually provide a schema**

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

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

val moviesSchema = StructType(Array(StructField("actor",StringType, true),

StructField("title",StringType, true),

StructField("year",IntegerType,true)))

val moviesCSVwithShema = spark.read.format("csv")

.option("sep",",")

.option("header","true")

.schema(moviesSchema)

.load("file:/databricks/driver/movies.csv")

moviesCSVwithShema.printSchema

|-- actor: string (nullable = true)

|-- title: string (nullable = true)

|-- year: integer (nullable = true)

### Reading JSON Files

A JSON object can be expressed on a single line or across multiple lines, and this is something you need to let Spark know

Tip: Given that a JSON data file contains only column names and no data type, how is Spark able to come up with a schema? Spark tries its best to infer the schema by parsing a set of sample records. The number of records to sample is determined by the samplingRatio option, which has a default value of 1.0. Therefore, it is quite expensive to read a large JSON file. In this case, you can lower the samplingRatio value to speed up the data loading process

Options:

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** | **Value** | **Default** | **Explanation** |
| allowComments | true,false | FALSE | Ignores comments in the JSON file |
| multiLine | true,false | FALSE | Treats the entire file as a large JSON object that spans many lines |
| samplingRatio | 0.3 | 1.0 | Secifies the sampling size to read to infer the schema |

**option("mode","failFast")** 🡪 tells sparks, if any bad encounter while inferring schema fail. Do not place with NULL values

**with infer schema:**

//Load json file

val moviesJson = spark.read.format("json").option("inferSchema","true").load("file:/databricks/driver/movies.json")

moviesJson.printSchema

|-- actor\_name: string (nullable = true)

|-- movie\_title: string (nullable = true)

|-- produced\_year: long (nullable = true)

+-----------------+------------+-------------+

| actor\_name| movie\_title|produced\_year|

+-----------------+------------+-------------+

|McClure, Marc (I)|Coach Carter| 2005|

|McClure, Marc (I)| Superman II| 1980|

|McClure, Marc (I)| Apollo 13| 1995|

+-----------------+------------+-------------+

**With bad schema – which will load null values**

val badMovieSchema = StructType(Array(StructField**("actor\_name",BooleanType**, true),

StructField("movie\_title",StringType, true),

StructField("produced\_year",IntegerType, true)))

val moviesJsonwithBadSchema = spark.read.format("json").schema(badMovieSchema).load("file:/databricks/driver/movies.json")

moviesJsonwithBadSchema.printSchema

**|-- actor\_name: boolean (nullable = true)**

|-- movie\_title: string (nullable = true)

|-- produced\_year: integer (nullable = true)

moviesJsonwithBadSchema.show(3)

+----------+------------+-------------+

|actor\_name| movie\_title|produced\_year|

+----------+------------+-------------+

| **null**|Coach Carter| 2005|

| **null**| Superman II| 1980|

| **null**| Apollo 13| 1995|

+----------+------------+-------------+

Avoid loading null – just throw error when infer is wrong:

val moviesJsonwithBadSchemaError = spark.read.format("json")

.schema(badMovieSchema)

.option("mode","failFast")

.load("file:/databricks/driver/movies.json")

moviesJsonwithBadSchemaError.show(3)

Job aborted due to stage failure.

Caused by: org.apache.spark.sql.catalyst.util.BadRecordException: java.lang.RuntimeException: Failed to parse a value for data type boolean (current token: VALUE\_STRING). Caused by: java.lang.RuntimeException: Failed to parse a value for data type boolean (current token: VALUE\_STRING)

### Parquet files:

* Parquet is one of the most popular open source columnar storage formats in the Hadoop
* It is a self-describing data format and it stores data in a highly compact structure by leveraging compressions
* Parquet stores the data of **each column in a separate file**; therefore, columns that are not needed in a data analysis would not have to be unnecessarily read in
* Spark works extremely well with the Parquet file format, and in fact Parquet is the default file format for reading and writing data in Spark
* Parquet files are self contained -schema is stored inside the Parquet data file
* you don’t need to provide a schema or ask Spark to infer the schema
* One of the cool optimizations that Spark does when reading data from Parquet is that it does decompression and decoding in column batches

// Parquet is the default format, so we don't need to specify the format when reading

*// No need to specify Format - its default for spark*

*// No need to give Schema or option("inferSchema","true") --> already contained in parquet file*

val moviesParquet = spark.read.load("file:/databricks/driver/movies.parquet")

*//https://raw.githubusercontent.com/Apress/beginning-apache-spark-2/master/chapter4/data/movies/movies.parquet*

moviesParquet.printSchema

|-- actor\_name: string (nullable = true)

|-- movie\_title: string (nullable = true)

|-- produced\_year: long (nullable = true)

### ORC Files:

Optimized Row Columnar (ORC) is another popular open source self-describing columnar storage format in the Hadoop ecosystem

It was created by a company called Cloudera as part of the initiative to massively speed up Hive

Only speficy format, no need to give schema() or option(“inferSchema”,”true”)

val moviesORC = spark.read.format("orc").load("file:/databricks/driver/movies.orc")