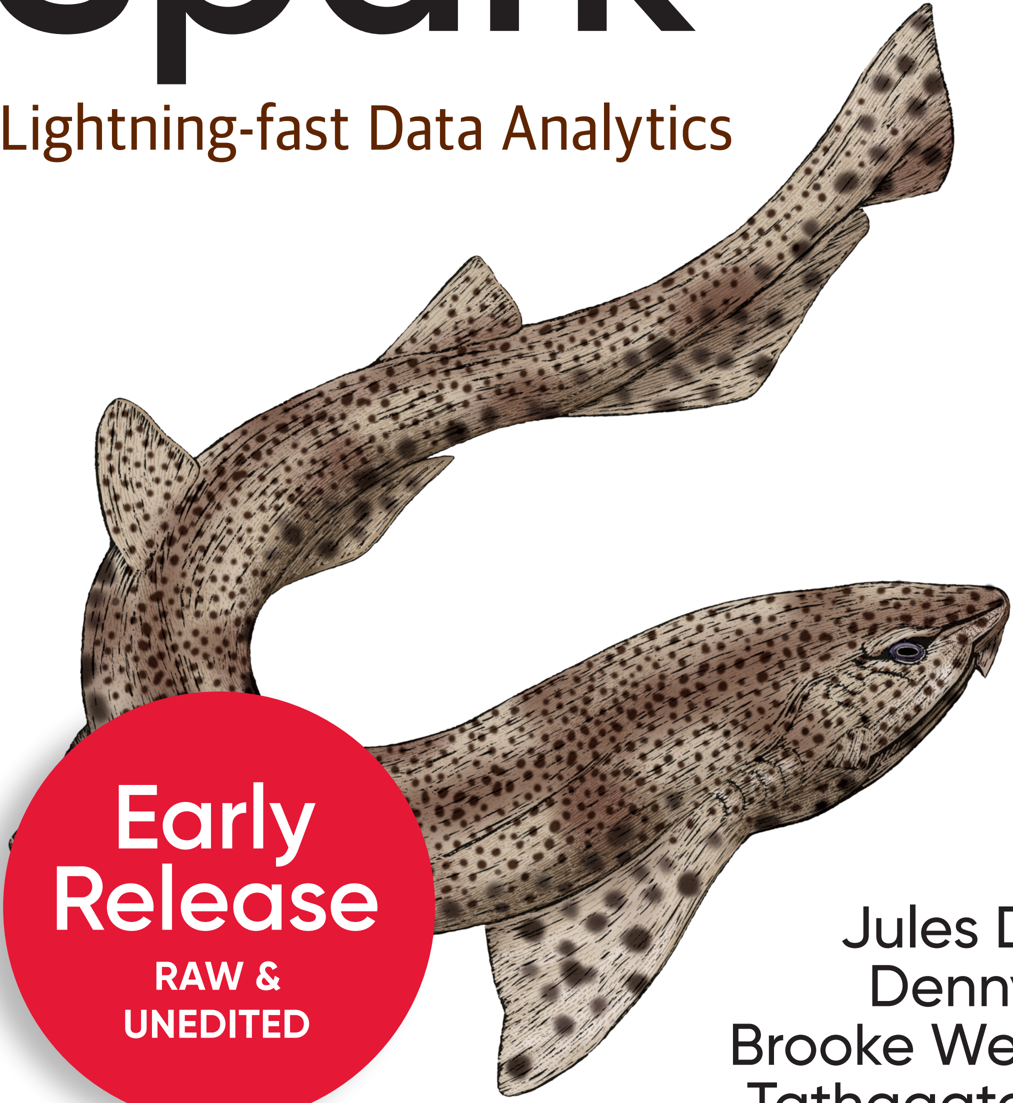
O'REILLY®

Second Edition

Sponsored by databricks

Learning Spark

Lightning-fast Data Analytics



Early Release

RAW & UNEDITED

Jules Damji,
Denny Lee,
Brooke Wenig &
Tathagata Das

WEBINAR

Making Apache Spark™ Better With Delta Lake

Michael Armbrust,
Principal Engineer
databricks™

SECOND EDITION

Learning Spark

This Preview Edition of *Learning Spark*, Second Edition, Chapter 3, is a work in progress. The final book is currently scheduled for release in July 2020 and will be available through O'Reilly online learning and other retailers once it is published.

Jules Damji, Denny Lee, Brooke Wenig, and Tathagata Das

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Learning Spark
by Jules Damji , Denny Lee , Brooke Wenig , and Tathagata Das

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Jonathan Hassell and Nicole Tache
Production Editor: Kristen Brown
Copyeditor:
Proofreader:
Indexer:
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January 2015:	First Edition
January 2020:	Second Edition

Revision History for the Early Release

2019-06-28:	First Release
2019-09-25:	Second Release
2020-01-17:	Third Release
2020-02-25:	Fourth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492050049* for release details.

978-1-492-05004-9

[LSI]

Table of Contents

CHAPTER 1

Apache Spark's Structured APIs

In this chapter, we will explore the principal motivations behind adding structure to Apache Spark, how structure led to the creation of high-level APIs - DataFrames and Datasets - and their unification in Spark 2.x across its components, and the Spark SQL engine that underpins these structured high-level APIs.

When Spark SQL was first introduced in the early Spark 1.x releases[1], followed by DataFrames as a successor to SchemaRDDs[2] [3] in Spark 1.3, we got our first glimpse of structure in Spark. At this time, Spark SQL introduced high-level expressive operational functions, mimicking SQL-like syntax, and DataFrames by providing spreadsheet-like or tabular named columns with data types dictated by a schema. DataFrames laid the foundation for more structure in subsequent releases and paved the path to performant operations in Spark's computational queries.

But before we talk about structure in Spark, let's get a brief glimpse of what it means to not have structure in Spark by peeking into the simple RDD programming API model.

Spark: What's Underneath an RDD?

These are the three vital characteristics associated with an RDD:[4] [5]

1 https://spark.apache.org/releases/spark-release-1-1-0.html

2 https://spark.apache.org/docs/1.1.0/api/java/org/apache/spark/sql/SchemaRDD.html

3 https://databricks.com/blog/2015/02/02/an-introduction-to-json-support-in-spark-sql.html

4 https://databricks.com/session/structuring-spark-dataframes-datasets-and-streaming

5 https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Dependencies

Partitions (with some locality information)

Compute Function: Partition => Iterator [T]

All three are integral to the simple RDD programming API model upon which all higher-level functionality is constructed. First, a list of dependencies that instruct Spark how an RDD is constructed with its inputs. When needed to reproduce results, Spark can by reproduce an RDD from these dependencies and replicate operations on it. This characteristic gives RDD resiliency.

Second, partitions provide Spark the ability to split the work to parallelize computation on partitions across Executors. In some cases, for example, reading from HDFS, Spark will use locality information, to send work to Executors close to the data. That way less data is transmitted over the network.

And finally, RDD has a compute function that produces an Iterator[T] for the data that will be stored in the RDD.

Simple and elegant! Yet there are a couple of problems with this original model. For one, the compute function (or computation) is opaque to Spark. That is, Spark does not know what you are doing in the compute function: whether you are performing a join, filter, select, or an aggregation; Spark only sees it as a lambda expression. Another problem is that the Iterator[T] data type is also opaque for Python RDDs; Spark only knows that it's a generic object in Python.

Second, unable to inspect the computation or expression in the function, Spark has no way to optimize the expression since it has no comprehension of its intention. And finally Spark has no knowledge of the specific data type in T. To Spark it's an opaque object; it has no idea if you are accessing a column or a column of a certain type within an object. Therefore, all Spark can do is serialize the opaque object as a series of bytes, at the expense of using any form of data compression techniques.

This opacity hampers Spark's ability to rearrange your computation into an efficient query plan. So what's the solution?

Structuring Spark

Spark 2.x introduces a few key schemes to structuring Spark. One is to express a computation by using common patterns found in data analysis. These patterns are expressed as high-level operations such as filtering, selecting, counting, aggregating, averaging, grouping, etc. By limiting these operations to a few common patterns in data analysis, Spark is now equipped to do more with less—because of clarity and specificity in your computation.

This specificity is further narrowed by a second scheme through a set of common operators in a domain-specific language (DSL). Through a set of operations in DSL, available as APIs in Spark's supported languages (Java, Python, Spark, R and SQL), Spark knows *what* you wish to compute with your data, and as a result, it can construct an efficient query plan for execution.

And the final scheme of order and structure is to allow you to arrange your data in a tabular format, such as a SQL table or spreadsheet, with supported Structured Data Types (which we will cover shortly).

Collectively, order, arrangement, clarity, and tabular data render structure to Spark. But what's it good for—all this structure?

Key Merits and Benefits

Structure yields a number of benefits, including better performance and space efficiency across Spark components. We will explore these benefits further when we talk about the use of DataFrames and Datasets APIs shortly, but for now a structure in Spark, in essence, offers developers expressivity, simplicity, composability, and uniformity.

Let's demonstrate expressivity and composability first with a simple code snippet. In the code below, we want to aggregate all ages, group them by name, and then average the age—a common recurring data analysis or discovery pattern. If we were to use low-level RDD API (as with the above opacity problem) the code would look as follows:

```
# In Python
        # Create an RDDs of tuples (name, age)
        dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
("TD", 35), ("Brooke", 25)])
        # Use map and reduceByKey transformations with their
        # lambda expressions to aggregate and then compute average
        agesRDD = dataRDD.map(lambda x, y: (x, (y, 1))) \
         .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
         .map(lambda x, y, z: (x, y / z))
```

Now, no one would dispute the above code is cryptic and hard to read. It instructs Spark *how-to* aggregate keys and compute averages with a string of lambda functions. In other words, the computation is instructing Spark *how-to* compute my query. It's completely opaque to Spark, for it has no idea what your intention is.

By contrast, what if we express the same query with high-level DSL operators and the DataFrames API, thereby instructing Spark *what-to-do*. Have a look:

```
# In Python
        from pyspark.sql import SparkSession
        from pyspark.sql.functions import avg
        # Create a DataFrame using SparkSession
        data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31),
("Jules", 30), ("TD", 35), ("Brooke", 25)], ["name", "age"])
        # Group the same names together, aggregate their age, and compute an
average
        avg_df = data_df.groupBy("name").agg(avg("age"))
        # Show the results of the final execution
        avg_df.show()
        +------+--------+
        | name|avg(age)|
        +------+--------+
        |Brooke| 22.5|
        | Jules| 30.0|
        | TD| 35.0|
        | Denny| 31.0|
        +------+--------+
```

Again, no one would dispute that this version of the code is far more expressive as well as simpler than the earlier version because we are using high-level DSL operators and APIs to instruct Spark *what-to-do*. In effect, we have employed these operators to compose our query. And because Spark can inspect or parse this query and understand your intention, it can optimize or arrange the operations for efficient execution. Spark knows exactly *what* you wish to do: group people by their names, aggregate their ages and then compute an average of all ages with the same name. You have composed an entire computation using high-level operators as a single simple query.

How expressive is that!

While many would contend that by using only high-level DSL expressive operators to map to common or recurring data analysis patterns to introduce order and structure, we are limiting the scope of the developers' ability to instruct or control how his or her query should be computed. Rest assured that you are not confined to these structured patterns; you can switch back at any time to unstructured low-level RDDs API, albeit we hardly find a need to do so.

As well as being simpler to read, the structure of Spark's high-level APIs also introduces uniformity across its components and languages. For example, the Scala code below does the same thing as above—and the API looks identical. In short, not only do we have parity among high-level APIs across languages, but they bring uniformity across languages and Spark components. (You will discover this uniformity in subsequent chapters on structured streaming and machine learning.)

```
// In Scala
        import org.apache.spark.sql.functions.avg
        // Create a DataFrame using SparkSession
        val dataDF = spark.createDataFrame(Seq(("Brooke", 20), ("Denny", 31),
("Jules", 30), ("TD", 35))).toDF("name", "age")
        // Group the same names together, aggregate their age, and compute an
average
```

```
val avgDF = dataDF.groupBy("name").agg(avg("age"))
// Show the results of the final execution
avgDF.show()
+------+--------+
| name|avg(age)|
+------+--------+
|Brooke| 22.5|
| Jules| 30.0|
| TD| 35.0|
| Denny| 31.0|
+------+--------+
```

Hint: Some of these DSL operators perform relational-like operations, such as select, filter, groupBy or aggregate, something you may be familiar with if you know SQL.

Consequently, all this simplicity and expressivity that we developers cherish is possible because of the underpinning Spark SQL engine upon which the high-level structured APIs are built. We will talk about Spark SQL engine shortly[6], but first let's explore the high-level structured APIs.

Structured APIs: DataFrames and Datasets APIs

The Spark SQL engine (which we will discuss below) supports Structured APIs in Spark 2.x and Spark 3.0. It is because of this underpinning engine across all Spark components that we get uniform APIs. Whether you express a query against a DataFrame in Structured Streaming or MLlib, you are always transforming and operating on DataFrames as structured data. (We will explore more of these APIs in Structured Streaming and Machine Learning in later chapters.)

For now, let's explore those APIs and DSLs for common operations and how to use them for data analytics.

DataFrame API

Inspired by Pandas[7] in structure, format, and a few specific operations, Spark DataFrames are like a distributed table with named columns with a schema, where each column has a specific data type: integer, string, array, map, real, date, timestamp, etc. To a human's eye, a Spark DataFrame is like a table:

Id (Int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int)	Campaigns (List[Strings])
1	Jules	Damji	https://tinyurl.1	1/4/2016	4535	[twitter, LinkedIn]

6 https://databricks.com/blog/2016/07/26/introducing-apache-spark-2-0.html

7 https://pandas.pydata.org/

2	Brooke	Wenig	https://tinyurl.2	5/5/2018	8908	[twitter, LinkedIn]
3	Denny	Lee	https://tinyurl.3	6/7/2019	7659	[web, twitter, FB, LinkedIn]
4	Tathagata	Das	https://tinyurl.4	5/12/2018	10568	[twitter, FB]
5	Matei	Zaharia	https://tinyurl.5	5/14/2014	40578	[web, twitter, FB, LinkedIn]
6	Reynold	Xin	https://tinyurl.6	3/2/2015	25568	[twitter, LinkedIn]

Table 3-1 Table like format of a DataFrame

When data is visualized as a structured table, it's not only easy to digest but it's easy to work with on common operations you might want to execute on rows and columns. Like RDDs, DataFrames are immutable. And as with RDDs, Spark keeps a lineage of all transformations, even with the columns. A named column in a DataFrame and its associated Spark data type can be declared in the schema.

Let's examine both the generic and structured data types in Spark before we use them to define a schema. Then, we will illustrate how to create a DataFrame with a schema, capturing the data in Table 3-1.

Spark's Basic Data Types

Matching its supported programming languages, Spark supports basic internal data types. These data types can be declared in your Spark application or defined in your schema. For example, in Scala, you can define or declare a particular column name to be of type String or Byte or Long.

```
$SPARK_HOME/bin/spark-shell…
          scala> import org.apache.spark.sql.types._
          import org.apache.spark.sql.types._
          scala> val nameTypes = StringType
          nameTypes: org.apache.spark.sql.types.StringType.type = StringType
          scala> val firstName = nameTypes
          firstName: org.apache.spark.sql.types.StringType.type = StringType
          scala> val lastName = nameTypes
          lastName: org.apache.spark.sql.types.StringType.type = StringType
```

Here are the other basic data types supported in Spark.[8] [9] They all are subtypes of class DataTypes, except for DecimalType.

Data Type	Value Assigned in Scala	APIs to instantiate
ByteType	Byte	DataTypes.ByteType

8 Spark: The Definitive Guide, pg 54

9 https://spark.apache.org/docs/latest/api/java/org/apache/spark/sql/types/DataTypes.html

ShortType	Short	DataTypes.ShortType
IntegerType	Int	DataTypes.IntegerType
LongType	Long	DataTypes.LongType
FloatType	Float	DataTypes.FloatType
DoubleType	Double	DataTypes.DoubleType
StringType	String	DataTypes.StringType
DecimalType	java.math.BigDecimal	DecimalType

Table 3-2 Scala Basic Data Types in Spark

Likewise, Python too supports similar basic data types as enumerated in the table below: [10] [11]

Data Type	Value Assigned in Python	APIs to instantiate
ByteType	Int or long	DataTypes.ByteType
ShortType	Short	DataTypes.ShortType
IntegerType	Int or long	DataTypes.IntegerType
LongType	long	DataTypes.LongType
FloatType	Float	DataTypes.FloatType
DoubleType	Float	DataTypes.DoubleType
StringType	String	DataTypes.StringType
DecimalType	decimal.Decimal	DecimalType

Table 3-3 Python Basic Data Types in Spark

Spark's Structured and Complex Data Types

For complex data analytics, you will hardly deal only with simple or basic data types. Rather, your data will be complex, often structured or nested. For that you will need Spark to handle these complex data types. They come in many forms: maps, arrays, structs, date, timestamp, fields etc,.

Data Type	Value Assigned in Scala	APIs to instantiate
BinaryType	Array[Byte]	DataTypes.BinaryType
TimestampType	java.sql.Timestamp	DataTypes.TimestampType
DateType	java.sql.Date	DataTypes.DateType

10 Spark: The Definitive Guide, pg 54

11 https://spark.apache.org/docs/2.4.0/api/python/pyspark.sql.html#module-pyspark.sql.types

ArrayType	scala.collection.Seq	DataTypes.createArrayType(ElementType)
		For example, DataTypes.createArray(BooleanType)
		DataTypes.createArrayType(IntegerType)

MapType	scala.collection.Map	DataTypes.createMapType(keyType, valueType)
		For example,
		DataTypes.createMapType(IntegerType, StringType)

StructType	org.apache.spark.sql.Row	StructType(ArrayType[fieldTypes]) This is an array of FieldTypes

StructField	A value type in Scala of this type of the field	StructField(name, dataType, [nullable])
		For example,
		StructField("jules", StringType)

Table 3-4 Scala Structured Data Types in Spark

The equivalent structured data types in Python are enumerated below:[12] [13]

Data Type	Value Assigned in Python	APIs to instantiate
BinaryType	Array[Byte]	BinaryType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	List, tuple, or array	ArrayType(dataType, [nullable])
MapType	dict	MapType(keyType, valueType, [nullable])
StructType	List or tuple	StructType(fields). Fields is a list of StructFields

StructField	The value of type in Python of this field	StructField(name, dataType, [nullable])
		StructField("jules", StringType)

Table 3-5 Python Structured Data Types in Spark

While these tables showcase the myriad types supported, it's far more important to see how these types come together when you define a schema for your data.

Schemas and Creating DataFrames

A *schema* in Spark defines the column names and its associated data types for a DataFrame. More often schemas come into play when you are reading structured data from DataSources (more on this in the next chapter). Defining schema as opposed to schema-on-read offers three benefits: 1) you relieve Spark from the onus of inferring data types 2) you prevent Spark from creating a separate job just to read a large portion of your file to ascertain the schema, which for a very large data file, can be

12 Spark: Definitive Guide, pg 53

13 https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.types

expensive and time consuming, and 3) detects errors early if data doesn't match the schema.

So we encourage you to always define your schema upfront whenever you want to read a very large file from a data source. For a short illustration, and using **Table 3-1** for our data, let's define a schema and use that schema to create a DataFrame.

Two Ways t o Define Schema . Spark allows you to define schema in two ways. One way is to define it programmatically. Another way is to employ a Data Definition Language (DDL) string, which is much simpler and easier to read.

To define a schema programmatically for a DataFrame with three named columns author, title, pages, you can use the Spark DataFrame API. For example,

```
// In Scala
            import org.apache.spark.sql.types._
            val schema = StructType(Array(StructField("author", StringType,
false),
             StructField("title", StringType, false),
             StructField("pages", IntegerType, false)))
            # In Python
            from pyspark.sql.types import *
            schema = StructType([StructField("author", StringType(), False),
             StructField("title", StringType(), False),
             StructField("pages", IntegerType(), False)])
            To define the same schema using DDL is much simpler.
            // In Scala
            val schema = "author STRING, title STRING, pages INT"
            # In Python
            schema = "author STRING, title STRING, pages INT"
```

Of the two ways to define schema, the one you should choose is up to you. For many examples, we will use both.

```
# In Python
            from pyspark.sql.types import *
            from pyspark.sql import SparkSession
            # define schema for our data using DDL
            schema = "`Id` INT,`First` STRING,`Last` STRING,`Url` STRING,`Pub-
lished` STRING,`Hits` INT,`Campaigns` ARRAY<STRING>"
            # create our static data
            data = [[1, "Jules", "Damji", "https://tinyurl.1", "1/4/2016",
4535, ["twitter", "LinkedIn"]],
             [2, "Brooke","Wenig","https://tinyurl.2", "5/5/2018", 8908, ["twit-
ter", "LinkedIn"]],
             [3, "Denny", "Lee", "https://tinyurl.3","6/7/2019",7659, ["web",
"twitter", "FB", "LinkedIn"]],
             [4, "Tathagata", "Das","https://tinyurl.4", "5/12/2018", 10568,
["twitter", "FB"]],
             [5, "Matei","Zaharia", "https://tinyurl.5", "5/14/2014", 40578,
["web", "twitter", "FB", "LinkedIn"]],
             [6, "Reynold", "Xin", "https://tinyurl.6", "3/2/2015", 25568,
["twitter", "LinkedIn"]]
             ]
            # main program
            if __name__ == "__main__":
             # create a SparkSession
             spark = (SparkSession
             .builder
             .appName("Example-3_6")
             .getOrCreate())
             # create a DataFrame using the schema defined above
             blogs_df = spark.createDataFrame(data, schema)
             # show the DataFrame; it should reflect our table above
             blogs_df.show()
             print()
             # print the schema used by Spark to process the DataFrame
             print(blogs_df.printSchema())
```

Running this program from the console will produce the following output

```
$ python Example-3_6.py
           …
           …
           +-------+---------+-------+-----------------+---------+-----
+---------------------------+
           |Id|First |Last |Url |Published|Hits |Campaigns |
           +-------+---------+-------+-----------------+---------+-----
+---------------------------+
           |1 |Jules |Damji |https://tinyurl.1|1/4/2016 |4535 |[twitter,
LinkedIn] |
           |2 |Brooke |Wenig |https://tinyurl.2|5/5/2018 |8908 |[twitter,
LinkedIn] |
           |3 |Denny |Lee |https://tinyurl.3|6/7/2019 |7659 |[web, twitter,
FB, LinkedIn]|
           |4 |Tathagata|Das |https://tinyurl.4|5/12/2018|10568|[twitter, FB] |
           |5 |Matei |Zaharia|https://tinyurl.5|5/14/2014|40578|[web, twitter,
FB, LinkedIn]|
           |6 |Reynold |Xin |https://tinyurl.6|3/2/2015 |25568|[twitter,
LinkedIn] |
           +-------+---------+-------+-----------------+---------+-----
+---------------------------+
           root
            |-- Id: integer (nullable = false)
            |-- First: string (nullable = false)
            |-- Last: string (nullable = false)
            |-- Url: string (nullable = false)
            |-- Published: string (nullable = false)
            |-- Hits: integer (nullable = false)
            |-- Campaigns: array (nullable = false)
            | |-- element: string (containsNull = false)
```

If you want to use this schema elsewhere in the code, simply execute blogs_df.schema and it will return this schema definition:

```
StructType(List(StructField(“Id”,IntegerType,false),
            StructField(“First”,StringType,false),
            StructField(“Last”,StringType,false),
            StructField(“Url”,StringType,false),
            StructField(“Published”,StringType,false),
            StructField(“Hits”,IntegerType,false),
            StructField(“Campaigns”,ArrayType(StringType,true),false)))
```

Note: The last boolean field argument indicates whether this field can have null values. If your data can be null for a particular field then specifying true means this field is “nullable.”

As you can observe, the DataFrame layout matches that of our Table 3-1 along with the respective data types and schema output.

Similarly if you were to read the data from a JSON file, instead of creating static data, the schema definition would be identical. Let’s illustrate the same code with a Scala example, except reading from a JSON file.

```
// In Scala
            package main.scala.chapter3
            import org.apache.spark.sql.SparkSession
            import org.apache.spark.sql.types._
            object Example3_7 {
             def main(args: Array[String]) {
             val spark = SparkSession
             .builder
             .appName("Example-3_7")
             .getOrCreate()
             if (args.length <= 0) {
             println("usage Example3_7 <file path to blogs.json>")
             System.exit(1)
             }
             // get the path to the JSON file
             val jsonFile = args(0)
             // define our schema programmatically
             val schema = StructType(Array(StructField("Id", IntegerType,false),
             StructField("First", StringType, false),
             StructField("Last", StringType, false),
             StructField("Url", StringType, false),
             StructField("Published", StringType, false),
             StructField("Hits", IntegerType, false),
             StructField("Campaigns", ArrayType(StringType), false)))
             // Create a DataFrame by reading from the JSON file
             // with a predefined Schema
             val blogsDF = spark.read.schema(schema).json(jsonFile)
             print()
             // show the DataFrame schema as output
             blogsDF.show(false)
             // print the schemas
             print(blogsDF.printSchema)
             print(blogsDF.schema)
             }
            }
```

Not surprisingly, the output from the Scala program is no different than the one from Python.

```
()+---+---------+-------+----------------+--------+-----
+---------------------------+
            |Id |First |Last |Url |Published|Hits |Campaigns |
            +---+---------+-------+----------------+--------+-----
+---------------------------+
            |1 |Jules |Damji |https://tinyurl.1|1/4/2016 |4535 |[twitter,
LinkedIn] |
            |2 |Brooke |Wenig |https://tinyurl.2|5/5/2018 |8908 |[twitter,
LinkedIn] |
            |3 |Denny |Lee |https://tinyurl.3|6/7/2019 |7659 |[web, twitter,
FB, LinkedIn]|
            |4 |Tathagata|Das |https://tinyurl.4|5/12/2018|10568|[twitter, FB] |
            |5 |Matei |Zaharia|https://tinyurl.5|5/14/2014|40578|[web, twitter,
FB, LinkedIn]|
            |6 |Reynold |Xin |https://tinyurl.6|3/2/2015 |25568|[twitter,
LinkedIn] |
            +---+---------+-------+----------------+--------+-----
+---------------------------+
            root
             |-- Id: integer (nullable = true)
             |-- First: string (nullable = true)
             |-- Last: string (nullable = true)
             |-- Url: string (nullable = true)
             |-- Published: string (nullable = true)
             |-- Hits: integer (nullable = true)
             |-- Campaigns: array (nullable = true)
             | |-- element: string (containsNull = true)
            StructType(StructField("Id",IntegerType,true),
             StructField("First",StringType,true),
             StructField("Last",StringType,true),
             StructField("Url",StringType,true),
             StructField("Published",StringType,true),
             StructField("Hits",IntegerType,true),
             StructField("Campaigns",ArrayType(StringType,true),true))
```

We hope that you now have an intuition for using structured data and schema in DataFrames. Now, let's focus on DataFrames columns and rows, and what it means to operate on them with the DataFrame API.

Columns and Expressions

As mentioned before, named columns in DataFrames are conceptually similar to named columns in pandas or R DataFrames or an RDBMS table: They describe the type of field. On their values, you can perform some operations using relational or computational expressions; you can list all columns by their names. But more importantly, in Spark's supported languages, columns are like objects with public methods. As such you can manipulate column values with expressions, or a series of mathematical computations.

You can also use expressions on columns of other Spark data types as well. For example, you create a simple expression using the expr("columnName * 5") or (expr("columnName - 5") > col(anothercolumnName)). Expr is part of the pyspark.sql.functions and org.apache.spark.sql.functions package. Like any other function in that package, expr is like a function taking arguments that Spark will parse as an expression and compute its result. We show some expr examples below.

All three languages Scala [14], Java[15], and Python[16] have public methods associated with columns.

Note: The Spark documentation refers to both Col and Column. Column is the name of the object, but you can also abbreviate it as col. More importantly, Spark needs to understand whether you are working with a DataFrame Column type, not a string. For brevity, we shall use col in the book.

```
// In Scala
          import org.apache.spark.sql.functions.{col, expr}
          blogsDF.columns
          Array[String] = Array(Campaigns, First, Hits, Id, Last, Published,
Url)
          // access a particular column
          blogsDF.col("Id")
          // use an expression to compute a value
          blogsDF.select(expr("Hits * 2")).show(2)
          // or use col to compute value
          blogsDF.select(col("Hits") * 2).show(2)
          +----------+
          |(Hits * 2)|
          +----------+
          | 9070|
          | 17816|
          +----------+
          // use expression to compute big hitters for blogs
          // this adds a new column Big Hitters based on the conditional expres-
sion
          blogsDF.withColumn("Big Hitters", (expr("Hits > 10000"))).show()
          +---+---------+-------+-----------------+---------+-----
+--------------------+------------+
          | Id| First| Last| Url|Published| Hits| Campaigns|Big Hitters|
          +---+---------+-------+-----------------+---------+-----
```

14 http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column

15 https://spark.apache.org/docs/2.1.0/api/java/org/apache/spark/sql/Column.html

16 https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Column

```
+-------------------+-----------+
          | 1| Jules| Damji|https://tinyurl.1| 1/4/2016| 4535| [twitter,
LinkedIn]| false|
          | 2| Brooke| Wenig|https://tinyurl.2| 5/5/2018| 8908| [twitter,
LinkedIn]| false|
          | 3| Denny| Lee|https://tinyurl.3| 6/7/2019| 7659|[web, twitter,
FB...| false|
          | 4|Tathagata| Das|https://tinyurl.4|5/12/2018|10568| [twitter, FB]|
true|
          | 5| Matei|Zaharia|https://tinyurl.5|5/14/2014|40578|[web, twitter,
FB...| true|
          | 6| Reynold| Xin|https://tinyurl.6| 3/2/2015|25568| [twitter,
LinkedIn]| true|
          +---+---------+-------+----------------+--------+-----
+-------------------+-----------+
          // use expression to concatenate three columns, create a new column
          // and show the newly created concatenated column
          blogsDF.withColumn("AuthorsId", (concat(expr("First"), expr("Last"),
expr("Id")))).select(expr("AuthorsId")).show(n=4)
          +-------------+
          | AuthorsId|
          +-------------+
          | JulesDamji1|
          | BrookeWenig2|
          | DennyLee3|
          |TathagataDas4|
          +-------------+
```

// And these three statements return the same value, showing that

```
// expr are same as column method call
          blogsDF.select(expr("Hits")).show(2)
          blogsDF.select(col("Hits")).show(2)
          blogsDF.select("Hits").show(2)
          +-----+
          | Hits|
          +-----+
          | 4535|
          | 8908|
          +-----+
          // Sort by column "Id" in descending order
          blogsDF.sort(col("Id").desc).show()
          blogsDF.sort($"Id".desc).show()
          +-------------------+--------+-----+---+-------+---------
+-----------------+
          | Campaigns| First| Hits| Id| Last|Published| Url|
          +-------------------+--------+-----+---+-------+---------
+-----------------+
          | [twitter, LinkedIn]| Reynold|25568| 6| Xin| 3/2/2015|https://
tinyurl.6|
          |[web, twitter, FB...| Matei|40578| 5|Zaharia|5/14/2014|https://
tinyurl.5|
          | [twitter, FB]|Tathagata|10568| 4| Das|5/12/2018|https://tinyurl.4|
          |[web, twitter, FB...| Denny| 7659| 3| Lee| 6/7/2019|https://tinyurl.
3|
          | [twitter, LinkedIn]| Brooke| 8908| 2| Wenig| 5/5/2018|https://
tinyurl.2|
          | [twitter, LinkedIn]| Jules| 4535| 1| Damji| 1/4/2016|https://
tinyurl.1|
          +-------------------+---------+-----+---+-------+---------
+-----------------+
```

Both these expressions—blogs_df.sort(col("Id").desc) and blogsDF.sort($"Id".desc)—above are identical. That is, they sort the DataFrame column name "Id" in a descending order: one uses an explicit function col("Id"), while the other uses "$" before the name of the column, which is a function in Spark that converts it to column.

Note: We have only scratched the surface and employed a couple of methods on Column object. For a complete list of all public methods for Column object, we refer you to the Spark documentation.[17]

Column objects in a DataFrame can't exist in isolation; each column is part of a row in a record; and all rows constitute an entire DataFrame, which as we will see later in the chapter is really a Dataset[Row] in Scala.

Rows

A row in Spark is a generic Row object, containing one or more columns. And each column may be of the same data type (Integer or String) or distinct with different types (Integer, String, Map, Array, etc.). Because Row is an object in Spark and an ordered collection of fields, you can instantiate a Row in each of Spark's supported languages and access its fields by an index starting at 0. [18] [19]

```
// In Scala
          import org.apache.spark.sql.Row
          // create a Row
          val blogRow = Row(6, "Reynold", "Xin", "https://tinyurl.6", 255568,
"3/2/2015", Array("twitter", "LinkedIn"))
          // access using index for individual items
          blogRow(1)
          res62: Any = Reynold
          # In Python
          from pyspark.sql import Row
          blog_row = Row(6, "Reynold", "Xin", "https://tinyurl.6", 255568,
"3/2/2015", ["twitter", "LinkedIn"])
          # access using index for individual items
          blog_rowr[1]
```

17 http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Column

18 http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Row

19 https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Row

```
            'Reynold'
            Row objects can be used to create DataFrames if you need them for
quick interactivity and exploration.
            # In Python
            from pyspark.sql import Row
            from pyspark.sql.type import *
            # using DDL String to define a schema
            schema = "`Author` STRING, `State` STRING"
            rows = [Row("Matei Zaharia", "CA"), Row("Reynold Xin", "CA")]
            authors_df = spark.createDataFrame(rows, schema)
            authors_df.show()
            // In Scala
            import org.apache.spark.sql.Row
            import org.apache.spark.sql.types._
            val rows = Seq(("Matei Zaharia", "CA"), ("Reynold Xin", "CA"))
            val authorsDF = rows.toDF("Author", "State")
            authorsDF.show()
            +-------------+-----+
            | Author|State|
            +-------------+-----+
            |Matei Zaharia| CA|
            | Reynold Xin| CA|
            +-------------+-----+
```

Even though we demonstrated a quick way to create DataFrames from Row objects, in practice, you will want to read them from a file as illustrated earlier. In most cases, because your files are going to be huge, defining a schema and using it is a quicker and more efficient way to create DataFrames.

After you have created a large distributed DataFrame, you are going to want to perform some common data operations on them. Let's examine some of these Spark operations with high-level relational operators in the Structured APIs.

Common DataFrame Operations

To perform common data operations on a DataFrame, you'll first need to load a DataFrame from a data source that holds your structured data. Spark provides an interface, DataFrameReaders [20] to read data into a DataFrame from myriad data sources and formats such as JSON, CSV, Parquet, Text, Avro, ORC etc. Likewise, to write back the DataFrame to a data source in a particular format, Spark uses DataFrameWriter [21].

20 https://spark.apache.org/docs/latest/api/python/pyspark.sql.html?highlight=dataframereader#pyspark.sql.DataFrameReader

21 http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameWriter

DataFrameReader and DataFrameWriter . Reading and writing is much simpler in Spark 2.x because of these high-level abstraction and contributions from the community to connect to myriad data sources, including common NoSQL stores, RDMS, streaming engines (Apache Kafka and Kinesis), etc.

To begin with a data source, let's read a large CSV file containing San Francisco fire calls.[22] As noted above, we will define a schema for this file and use DataFrameReader class and its methods to instruct Spark what to do. Because this file contains 28 columns and over 4,380,660 records, it's more efficient to define a schema than have Spark infer it.

Hint: If you don't want to specify the schema, Spark can infer schema from a sample at a lesser cost. For example you can use the sampleRatio option.

```
// In Scala
          val sampleDF = spark.read
           .option("samplingRatio",0.001)
           .option("header", "true")
           .csv("path_to_csv_file")
```

Note: The original data set has over 60 columns. We dropped a few unnecessary columns for this example. For example, some dates had null and or invalid values.

```
# In Python define a schema
          from pyspark.sql.types import *
          # Programmatic way to define a schema
          fire_schema = StructType([StructField('CallNumber', IntegerType(),
True),
           StructField('UnitID', StringType(), True),
           StructField('IncidentNumber', IntegerType(), True),
           StructField('CallType', StringType(), True),
           StructField('CallDate', StringType(), True),
           StructField('WatchDate', StringType(), True),
           StructField('CallFinalDisposition', StringType(), True),
           StructField('AvailableDtTm', StringType(), True),
           StructField('Address', StringType(), True),
           StructField('City', StringType(), True),
           StructField('Zipcode', IntegerType(), True),
           StructField('Battalion', StringType(), True),
           StructField('StationArea', StringType(), True),
           StructField('Box', StringType(), True),
           StructField('OriginalPriority', StringType(), True),
           StructField('Priority', StringType(), True),
           StructField('FinalPriority', IntegerType(), True),
           StructField('ALSUnit', BooleanType(), True),
           StructField('CallTypeGroup', StringType(), True),
           StructField('NumAlarms', IntegerType(), True),
```

22 This public data is available at https://data.sfgov.org/Public-Safety/Fire-Incidents/wr8u-xric/data

```
              StructField('UnitType', StringType(), True),
              StructField('UnitSequenceInCallDispatch', IntegerType(), True),
              StructField('FirePreventionDistrict', StringType(), True),
              StructField('SupervisorDistrict', StringType(), True),
              StructField('Neighborhood', StringType(), True),
              StructField('Location', StringType(), True),
              StructField('RowID', StringType(), True),
              StructField("Delay", FloatType(), True)])
             # read the file using DataFrameReader using format CSV
             sf_fire_file = "databricks-datasets/learning-spark-v2/sf-fire/sf-fire-
calls.csv"
             fire_df = spark.read.csv(sf_fire_file, header=True,
schema=fire_schema)
             // In Scala it would be similar
             val fileSchema = StructType(Array(StructField("CallNumber", Integer-
Type, true),
              StructField("UnitID", StringType, true),
              StructField("IncidentNumber", IntegerType, true),
              StructField("CallType", StringType, true),
              StructField("Location", StringType, true)),
              …
              …
              StructField("Location", StringType, true)),
              StructField("Delay", FloatType, true)))
             // read the file using the CSV DataFrameReader
             val sfFireFile = "databricks-datasets/learning-spark-v2/sf-fire/sf-
fire-calls.csv"
             val fireDF = spark.read.schema(fireSchema)
              .option("header", "true")
              .csv(sfFireFile)
```

The code spark.read.csv creates a DataFrameReader object to read a CSV file type DataSource and reads the file according to the schema supplied. With success, it returns a DataFrame of rows and named columns with their respective types, as dictated in the schema.

Using a DataFrameWriter, you can do one of two things. First, write the DataFrame as another format into an external data source. As with DataFrameReader, DataFrameWriter supports multiple data sources. [23] [24] Parquet, a popular columnar format, is the default format and it uses compression to store the data. And second, if written as Parquet, it preserves the schema as part of the Parquet metadata. Subsequent reads back into DataFrame do not require to manually supply a schema.

Saving DataFrame as Parquet File Format and SQL Table . Another common data operation is to explore and transform data, and then persist the file as Parquet or save it as

23 https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameWriter

24 http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameWriter

a SQL table. Persisting a transformed DataFrame is as easy as reading it. For example, to persist the above DataFrame both as a file and table immediately after reading is simple:

```
// In Scala to save as a parquet file or Table
          val parquetPath = …
          fireDF.write
           .format("parquet")
           .save(parquetPath)
```

Alternatively, you can save it as a table, which registers metadata with the Hive metastore (we will cover SQL managed and unmanaged tables, metastores, and DataFrames in the next chapter).

```
// In Scala to save as a parquet file or Table
          val parquetTable = … //name of the table
          fireDF.write
           .format("parquet")
           .saveAsTable(parquetTable)
          # In Python
          parquet_table = … # name of the table
          (fire_df.write
           .format("parquet")
           .saveAsTable(parquetTable))
```

Let's walk through common data operations on DataFrames after you have read the data.

Transformations & Actions . Now that we have an entire distributed DataFrame composed of SF Fire department calls in memory, the first thing you as a data engineer or data scientist will want to do is examine your data: see what the columns look like. That is, are they of the correct data type? Do they need conversion to the right data type? Do they have null values etc.

In Chapter 2 (see section on Transformation, actions, and Lazy evaluations), you got a glimpse on some transformations and actions as high-level API and Domain Specific Language (DSL) operators on DataFrames. What can we find out from our San Francisco Fire calls?

Projections and Filters . A projection in the relational parlance is the ability to select all or individual rows matching a certain relational condition, by using filters. In Spark, projections equate to the select() method, while filter can be expressed as filter() or where() methods. Let's use both these methods to examine parts of our SF Fire incidents.

```
# In Python
          few_fire_df = (fire_df.select("IncidentNumber", "AvailableDtTm",
"CallType")
           .where(col("CallType") != "Medical Incident"))
          few_fire_df.show(5, truncate=False)
          // In Scala
          val fewFireDF = fireDF.select("IncidentNumber", "AvailableDtTm",
"CallType")
           .where($"CallType" =!= "Medical Incident")
          fewFireDF.show(5, false)
          This code gives us the following output:
          +--------------+--------------------+---------------+
          |IncidentNumber|AvailableDtTm |CallType |
          +--------------+--------------------+---------------+
          |2003234 |01/11/2002 01:58:43 AM|Medical Incident|
          |2003233 |01/11/2002 02:10:17 AM|Medical Incident|
          |2003235 |01/11/2002 01:47:00 AM|Structure Fire |
          |2003235 |01/11/2002 01:51:54 AM|Structure Fire |
          |2003235 |01/11/2002 01:47:00 AM|Structure Fire |
          +--------------+--------------------+---------------+
          (only showing top 5 rows)
```

However, what if you are only interested in distinct CallTypes as the causes of the fire calls, and you want to count the total. These simple and expressive queries do the job:

```
# In Python, return the count using the action distinctCount()
          fire_df.select("CallType").where(col("CallType") != "null").distinct-
Count()
          # filter for only distinct non-null CallTypes from all the rows
          fire_df.select("CallType").where(col("CallType") != "null").dis-
tinct().show(10, False)
          // In Scala, return the count using the action count()
          fireDF.select("CallType").where(col("CallType") =!= "null").dis-
tinct().count()
          // In Scala, filter for only distinct non-null CallTypes from all the
rows
          fireDF.select("CallType").where($"CallType" =!= lit("null")).dis-
tinct().show(10, false)
```

This code returns the output of 32 distinct call types for the fire calls.

```
Out[20]: 32
          +-----------------------------------+
          |CallType |
          +-----------------------------------+
          |Elevator / Escalator Rescue |
          |Marine Fire |
          |Aircraft Emergency |
          |Confined Space / Structure Collapse|
          |Administrative |
          |Alarms |
          |Odor (Strange / Unknown) |
          |Lightning Strike (Investigation) |
          |Citizen Assist / Service Call |
          |HazMat |
          +-----------------------------------+
          (only showing top 10 rows)
```

Renaming, Adding or Dropping Columns, and Aggregating . Sometimes you wish to rename particular columns for reasons of style or conventions; other times for readability or brevity. Our original column names in the SF Fire calls file had spaces in them. For example, the column "IncidentNumber" was "Incident Number." In fact, all columns had spaces in them, which can be problematic, especially when you wish to write or save a DataFrame as a Parquet file (which prohibits spaces in column names).

By specifying the desired column name in the schema StructField, as we did, we effectively changed the ultimate name in the returning DataFrame for all our column names.

Alternatively, you could selectively rename columns with DataFrame's public withColumnRenamed() method. For instance, let's change our Delay column to ResponseDelayedinMin and issue the count query again.

```
# In Python
          new_fire_df = fire_df.withColumnRenamed("Delay", "ResponseDelayedin-
Mins")
          new_fire_df.select("ResponseDelayedinMins").where(col("ResponseDelaye-
dinMins") > 5).show(5, False)
          // In Scala
          val newFireDF = fireDF.withColumnRenamed("Delay", "ResponseDelayedin-
Mins")
          newFireDF.select("ResponseDelayedinMins").where($"ResponseDelayedin-
Mins" > 5).show(5, false)
```

This gives us a new renamed column:

```
+--------------------+
          |ResponseDelayedinMins|
          +--------------------+
          |5.233333 |
          |6.9333334 |
          |6.116667 |
          |7.85 |
          |77.333336 |
          +--------------------+
          (only showing top 5 rows)
```

Note: Because DataFrame transformations are immutable, in both the above queries, when we rename a column, we get a new DataFrame while retaining the original with the old column name.

Modifying the contents of a column or its type is often a common data operation during exploration. In some cases, the data is raw or dirty, or its types are not amenable to supply as arguments to relational operators. For example, in our SF Fire calls data set, the columns "CallDate", "WatchDate", and "AlarmDtTm" are strings rather than either Unix timestamp or SQL Date, which Spark supports and Spark can easily manipulate during transformations or actions, especially during a date or time analysis of your data.

So how do we convert them into a more usable format? It's quite simple with a set of high-level API methods.

spark.sql.function has a set of to/from date/timestamp functions such as to_timestamp() or to_date().

```
# In Python
          fire_ts_df = (fire_df.withColumn("IncidentDate", to_time-
stamp(col("CallDate"), "MM/dd/yyyy")).drop("CallDate")
           .withColumn("OnWatchDate", to_timestamp(col("WatchDate"), "MM/dd/
yyyy")).drop("WatchDate")
           .withColumn("AvailableDtTS", to_timestamp(col("AvailableDtTm"),
"MM/dd/yyyy hh:mm:ss aa")).drop("AvailableDtTm"))
          # select the converted columns
          fire_ts_df.select("IncidentDate", "OnWatchDate", "Availa-
bleDtTS").show(5, False)
          // In Scala
          val fireTsDF = fireDF.withColumn("IncidentDate", to_time-
stamp(col("CallDate"), "MM/dd/yyyy")).drop("CallDate")
           .withColumn("OnWatchDate", to_timestamp(col("WatchDate"), "MM/dd/
yyyy")).drop("WatchDate")
           .withColumn("AvailableDtTS", to_timestamp(col("AvailableDtTm"),
"MM/dd/yyyy hh:mm:ss aa")).drop("AvailableDtTm")
          // select the converted columns
          fireTsDF.select("IncidentDate", "OnWatchDate", "Availa-
bleDtTS").show(5, false)
```

The query above packs quite a punch. A number of things are happening. Let's unpack them:

Convert the existing column's data type from string to a Spark supported timestamp.

Use the new format as specified in the format string "MM/dd/yyyy" or "MM/dd/yyyy hh:mm:ss aa" where appropriate.

After converting to new data type, drop() the old column, and append the new one specified in the first argument to withColumn() method.

Assign the new modified DataFrame to fire_ts_df

This query results in new columns:

```
+-------------------+-------------------+-------------------+
          |IncidentDate |OnWatchDate |AvailableDtTS |
          +-------------------+-------------------+-------------------+
          |2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:58:43|
          |2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 02:10:17|
          |2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:47:00|
          |2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:51:54|
          |2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:47:00|
          +------------------+------------------+------------------+
          (only showing top 5 rows)
```

Now that we have modified the dates, we can query using spark.sql.functions like mon(), year(), day() etc., for instance, to explore our data further. Let's calculate all the years data of fire calls in our data set. We can also ask how many calls were logged in the last seven days.

```
# In Python
          fire_ts_df.select(year('IncidentDate')).distinct().orderBy(year('Inci-
dentDate')).show()
          // In Scala
          fireTSDF.select(year($"IncidentDate")).distinct().orderBy(year($"Inci-
dentDate")).show()
```

This query gives us all the years up to 2018 in this data set:

```
+------------------+
          |year(IncidentDate)|
          +------------------+
          | 2000|
          | 2001|
          | 2002|
          | 2003|
          | 2004|
          | 2005|
          | 2006|
          | 2007|
          | 2008|
          | 2009|
          | 2010|
          | 2011|
          | 2012|
          | 2013|
          | 2014|
          | 2015|
          | 2016|
          | 2017|
          | 2018|
          +------------------+
```

So far in this section, we have explored a number of concepts: DataFrameReaders and DataFrameWriters; defining a schema and using it in the DataFrame readers; saving a DataFrame as a Parquet file or table; projecting and filtering selected columns from an existing DataFrame; and modifying, renaming, and dropping some columns—all common data operations.

One final common operation is grouping data by values in a column, and aggregating the data in some way, like simply counting it. This pattern of grouping and counting is as common as projecting and filtering. Let's have a go at it.

Aggregations. What if we want to ask the question what were the common types of fire calls? And another question: what ZIP codes accounted for most of the fire calls? All these are common patterns of data analysis and exploration.

A handful of transformations and actions on DataFrame such as groupBy(), orderBy() and count() offer the ability to aggregate by column names and then aggregate count across them. [25]

For larger DataFrames on which you plan to conduct frequent or repeated queries, you could benefit from caching a DataFrame. We will cover DataFrames caching strategies and their benefits in later chapters.

Caution: Also, for extremely large DataFrames, collect() is dangerous and resource-heavy (expensive), as it can cause Out of Memory (OOM) exceptions. Unlike count(), which returns a single number to the driver, collect() returns a collection of all the Row objects back to the driver. Instead, to take a peek at some Row records, you're better off with take(n), where n will return only the first n Row objects of the DataFrame.

Let's take our first question: what were the common types of fire calls?

```
# In Python
          fire_ts_df.select("CallType").where(col("CallType").isNot-
Null()).groupBy("CallType").count().orderBy("count", ascend-
ing=False).show(n=10, truncate=False)
          // In Scala
          fireTSDF.select("CallType").where(col("CallType").isNot-
Null).groupBy("CallType").count().orderBy(desc("count")).show(10, false)
```

Both produce the following output:

```
+-----------------------------+-------+
          |CallType |count |
          +-----------------------------+-------+
          |Medical Incident |2843475|
          |Structure Fire |578998 |
          |Alarms |483518 |
          |Traffic Collision |175507 |
          |Citizen Assist / Service Call |65360 |
          |Other |56961 |
          |Outside Fire |51603 |
          |Vehicle Fire |20939 |
          |Water Rescue |20037 |
          |Gas Leak (Natural and LP Gases)|17284 |
          +-----------------------------+-------+
```

25 https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset

From the above, we can conclude that the most common call type is medical incident.

Other Common DataFrame Operations . Along with the above, DataFrames provide descriptive statistical methods like min(), max(), sum(), avg(), etc. Here are some examples showing how to compute them with our SF Fire calls data set.

Let's compute the sum of alarms, the average response time, and the min and max response time to all fire calls in our dataset.

```
# In Python
          fire_ts_df.select(sum("NumAlarms"), avg("ResponseDelayedinMins"),
min("ResponseDelayedinMins"), max("ResponseDelayedinMins")).show()
          // In Scala
          fireTSDF.select(sum("NumAlarms"), avg("ResponseDelayedinMins"),
min("ResponseDelayedinMins"), max("ResponseDelayedinMins")).show()
```

Both queries generate the following output:

```
+--------------+-------------------------+-------------------------
+-------------------------+
          |sum(NumAlarms)|avg(ResponseDelayedinMins)|min(ResponseDelayedinMins)|
max(ResponseDelayedinMins)|
          +--------------+-------------------------+-------------------------
+-------------------------+
          | 4403441| 3.902170335891614| 0.016666668| 1879.6167|
          +--------------+-------------------------+-------------------------
+-------------------------+
```

For more advanced statistics, common with data science workloads, read the DataFrame documentation for methods like stat(), describe(), correlation(), covariance(), sampleBy(), approxQuantile(), frequentItems() etc.

As you can see, it's easy to compose and chain expressive queries with DataFrames' high-level API and DSL operators. We can't imagine the opacity and readability of the code if we were to do the same with RDDs!

End-to-End DataFrame Example

In this end-to-end example we conduct exploratory data analysis, ETL (extract, transform, and load), and common data operations, above and beyond what we shared above on the San Francisco 2018 Fire calls public data set.

For brevity, we won't include the entire example code here. However, we have furnished links to both Python and Scala Databricks Community Edition Notebooks for you to try. In short, the notebooks explore and answer the following common questions you might ask of this dataset. Using DataFrames APIs and DSL relational operators, we can answer:

What were all the different types of fire calls in 2018?

What months within the year 2018 saw for the highest number of fire calls?

Which neighborhood in SF generated the most fire calls in 2018?

Which neighborhoods in SF had the worst response time to fire calls in 2018?

Which week in the year in 2018 had the most fire calls?

Is there a correlation between neighborhood, zip code, and fire calls?

How can we use Parquet files or SQL tables to store this data and read it back?

Import the complete Python or Scala notebook from the Learning Spark 2nd GitHub.

So far we have extensively discussed DataFrames as one of the Structured APIs that span Spark's components (MLlib and Structured Streaming, which we cover later in the book).

Let's shift our focus to the Dataset API. Let's explore how Dataset and DataFrame APIs provide a unified, structured interface to developers to program Spark. Let's examine the relationship between RDDs, DataFrames, and Datasets APIs. And, let's determine when to use which API and why.

Datasets API

Apache Spark 2.0 introduced a structured API. Its primary goal was to simplify DataFrame and Dataset APIs so that you as a developer only have to grapple with one set of APIs. [26] [27] LIke RDD, Datasets are language-native type classes and objects in Scala and Java, whereas DataFrames lose that characteristic. Foundational in their API types, Dataset takes on two characteristics: *strongly-typed* and *untyped* APIs, as shown in Figure 3-1.

Conceptually, consider DataFrame as an alias for a collection of generic objects Dataset[Row], where a Row is a generic typed JVM object, which may hold different types of fields. Dataset, by contrast, is a collection of strongly-typed JVM objects in Scala or a class in Java. Put another way, Datasets are a "strongly-typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each Dataset [in Scala] also has an untyped view called a DataFrame, which is a Dataset of Row." [28]

26 https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html

27 https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

28 http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset

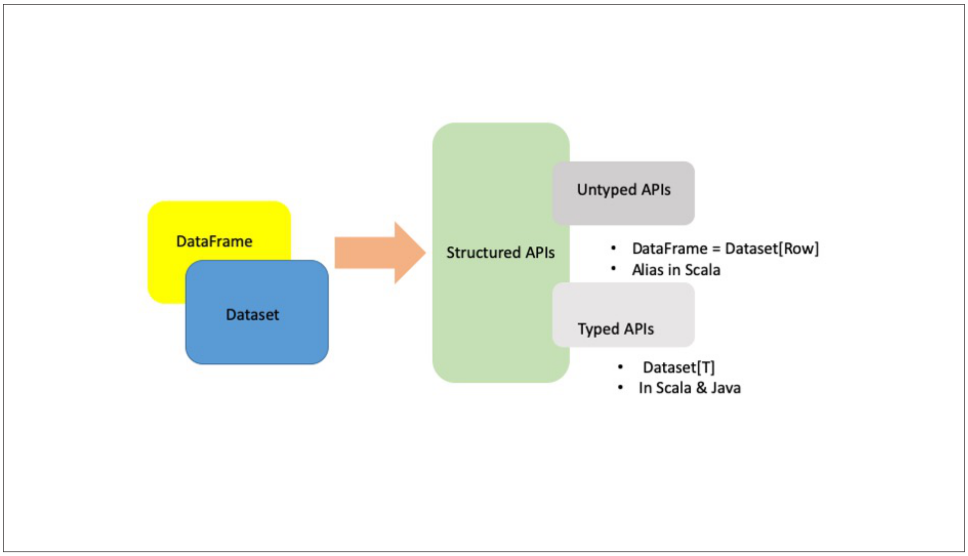


Figure 1-1. Structured APIs in Apache Spark

Typed Objects, Untyped Objects, and Generic Rows . In Spark's supported languages, only Dataset exists in Java and Scala, whereas in Python and R, only DataFrames APIs make sense. There are a few reasons for this. First, these languages are not compile-time type-safe; types are dynamically inferred or assigned during execution, not during compile time. And second, typed-object in Scala or Java is a characteristic of a Java Virtual Machine (JVM); types to objects or variables are bound at compile time, not during execution time. Our **Table 3-6** distills it in a nutshell.

Language	Typed and Untyped Main Abstraction	Typed or Untyped
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])	Both typed and untyped
Java	Dataset<T>	Typed
Python	DataFrame	Generic Row Untyped
R	DataFrame	Generic Row Untyped

Table 3-6 Typed and Untyped Objects in Spark

Row is a generic object type in Spark, holding a collection of mixed types that can be accessed using an index. Internally, Spark manipulates Row objects, converting them to equivalent Spark types covered in **Tables 3.4** and **3.5**. For example, an Int as one of your fields in a Row will be mapped or converted to IntegerType or IntegerType() respectively for Scala or Java and Python.

```
// In Scala
        import org.apache.spark.sql.Row
```

```
          val row = Row(350, true, "Learning Spark 2E", null)
          # In Python
          From pyspark.sql import Row
          row = Row(350, True, "Learning Spark 2E", None)
          Returns into val
          row: org.apache.spark.sql.Row = [350,true,Learning Spark 2E,null]
```

Using index into the Row object, you can access individual fields using Row's public *getter* methods.

```
// In Scala
          row.getInt(0)
          row.getBoolean(1)
          row.getString(2)
```

Returning these values:

```
res23: Int = 350
          res24: Boolean = true
          res25: String = Learning Spark 2E
          # In Python
          row[0]
          row[1]
          row[2]
```

Returning values:

```
Out[13]: 350
          Out[14]: True
          Out[15]: 'Learning Spark 2E'
```

By contrast, typed objects are actual Java or Scala class objects in the JVM. Each element in a Dataset maps to a JVM object.

Creating Datasets

As with creating DataFrames from data sources, you have to know a schema. In other words, you need to know the data types so that you can define your schema. Although for JSON and CSV, you can infer the schema, for large datasets inferring schema is resource-intensive (expensive). The easiest way to specify the schema of the resulting Dataset in Scala is to use case classes, which can be introspected to determine its schema. In Java, JavaBean classes are used (we discuss JavaBeans and case classes in chapter 6).

Scala: Case Classes . When you wish to instantiate your own domain-specific object as a Dataset, you can do so by defining a case class in Scala. As an example, let's look at a collection of internet of devices in a JSON file (We use this file in the end-to-end example below).[29]

Our file has rows of JSON strings that look as follows:

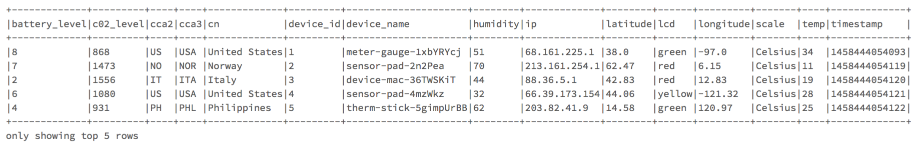
```
{"device_id": 198164, "device_name": "sensor-pad-198164owomcJZ", "ip":
"80.55.20.25", "cca2": "PL", "cca3": "POL", "cn": "Poland", "latitude":
53.080000, "longitude": 18.620000, "scale": "Celsius", "temp": 21, "humidity":
65, "battery_level": 8, "c02_level": 1408, "lcd": "red", "timestamp" :
1458081226051}
```

To express each JSON entry as DeviceIoTData, a domain-specific object, define a Scala case class:

```
case class DeviceIoTData (battery_level: Long, c02_level: Long,
              cca2: String, cca3: String, cn: String, device_id: Long,
              device_name: String, humidity: Long, ip: String, latitude: Double,
               lcd: String, longitude: Double, scale:String, temp: Long,
              timestamp: Long)
```

Once defined, we can use it to read our file and convert the returned Dataset[Row] into Dataset[DeviceIoTData]

```
// In Scala
              val ds = spark.read.json("/databricks-datasets/learning-spark-v2/
iot-devices/iot_devices.json").as[DeviceIoTData]
              ds: org.apache.spark.sql.Dataset[DeviceIoTData] = [battery_level:
bigint, c02_level: bigint ... 13 more fields]
              ds.show(5, false)
```



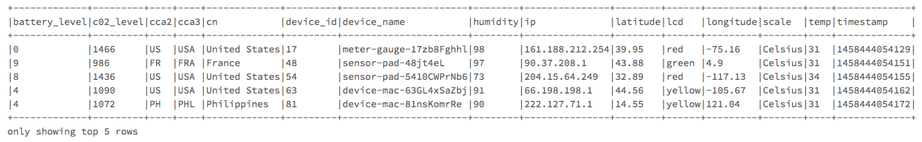
```
+-------------+---------+----+----+-------------+---------+---------------------+--------+-------------+--------+------+---------+-------+----+-------------+
|battery_level|c02_level|cca2|cca3|cn           |device_id|device_name          |humidity|ip           |latitude|lcd   |longitude|scale  |temp|timestamp    |
+-------------+---------+----+----+-------------+---------+---------------------+--------+-------------+--------+------+---------+-------+----+-------------+
|8            |868      |US  |USA |United States|1        |meter-gauge-1xbYRYcj |51      |68.161.225.1 |38.0    |green |-97.0    |Celsius|34  |1458444054093|
|7            |1473     |NO  |NOR |Norway       |2        |sensor-pad-2n2Pea    |70      |213.161.254.1|62.47   |red   |6.15     |Celsius|11  |1458444054119|
|2            |1556     |IT  |ITA |Italy        |3        |device-mac-36TWSKiT  |44      |88.36.5.1    |42.83   |red   |12.83    |Celsius|19  |1458444054120|
|6            |1080     |US  |USA |United States|4        |sensor-pad-4mzWkz    |32      |66.39.173.154|44.06   |yellow|-121.32  |Celsius|28  |1458444054121|
|4            |931      |PH  |PHL |Philippines  |5        |therm-stick-5gimpUrBB|62      |203.82.41.9  |14.58   |green |120.97   |Celsius|25  |1458444054122|
+-------------+---------+----+----+-------------+---------+---------------------+--------+-------------+--------+------+---------+-------+----+-------------+
only showing top 5 rows
```

Datasets Operations

Just as you can perform transformations and actions on DataFrames, so can you on Datasets. Depending on the kind of operations, the results type will vary.

```
// In Scala
            Val filterTempDS = ds.filter(d => {d.temp > 30 && d.humidity > 70})
            filterTempDS: org.apache.spark.sql.Dataset[DeviceIoTData] = [bat-
tery_level: bigint, c02_level: bigint ... 13 more fields]
            filterTempDs.show(5, false)
```

29 https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html



```
+-------------+---------+----+----+-------------+---------+--------------------+--------+---------------+--------+------+---------+-------+----+-------------+
|battery_level|c02_level|cca2|cca3|cn           |device_id|device_name         |humidity|ip             |latitude|lcd   |longitude|scale  |temp|timestamp    |
+-------------+---------+----+----+-------------+---------+--------------------+--------+---------------+--------+------+---------+-------+----+-------------+
|0            |1466     |US  |USA |United States|17       |meter-gauge-17zb8Fghhl|98      |161.188.212.254|39.95   |red   |-75.16   |Celsius|31  |1458444054129|
|9            |986      |FR  |FRA |France       |48       |sensor-pad-48jt4eL  |97      |90.37.208.1    |43.88   |green |4.9      |Celsius|31  |1458444054151|
|8            |1436     |US  |USA |United States|54       |sensor-pad-5410CWPrNb6|73      |204.15.64.249  |32.89   |red   |-117.13  |Celsius|34  |1458444054155|
|4            |1090     |US  |USA |United States|63       |device-mac-63GL4xSaZbj|91      |66.198.198.1   |44.56   |yellow|-105.67  |Celsius|31  |1458444054162|
|4            |1072     |PH  |PHL |Philippines  |81       |device-mac-81nsKomrRe |90      |222.127.71.1   |14.55   |yellow|121.04   |Celsius|31  |1458444054172|
+-------------+---------+----+----+-------------+---------+--------------------+--------+---------------+--------+------+---------+-------+----+-------------+
only showing top 5 rows
```

Note: In the query above, we used a function as a argument to the Dataset method filter() because the method is overloaded with many signatures. This method usage filter(func: (T) ⇒ Boolean): Dataset[T] takes a lambda function: func: (T) ⇒ Boolean as its argument.

One thing is quite simple and clear from the above query: filter() method can take a lambda function in Scala. The argument to the lambda function is a JVM object of type DeviceIoTData. As such, we can access its individual data fields using the "." notation, like you would in a Scala class or JavaBean.

Another thing to note is that with DataFrames, you express your filter() conditions as SQL-like DSL operations, which are language agnostic, as we saw above in our Fire calls example. In Dataset, these operations use language-native expressions as Scala or Java code.

Here's another example that results into another smaller Dataset.

```
// In Scala
            case class DeviceTempByCountry(temp: Long, device_name: String,
device_id: Long, cca3: String)
            val dsTemp = ds.filter(d => {d.temp > 25})
             .map(d => (d.temp, d.device_name, d.device_id, d.cca3))
             .toDF("temp","device_name","device_id","cca3")
             .as[DeviceTempByCountry]
            dsTemp.show(5, false)
            +----+--------------------+---------+----+
            |temp|device_name |device_id|cca3|
            +----+--------------------+---------+----+
            |34 |meter-gauge-1xbYRYcj |1 |USA |
            |28 |sensor-pad-4mzWkz |4 |USA |
            |27 |sensor-pad-6al7RTAobR|6 |USA |
            |27 |sensor-pad-8xUD6pzsQI|8 |JPN |
            |26 |sensor-pad-10BsywSYUF|10 |USA |
            +----+--------------------+---------+----+
            only showing top 5 rows
```

Or you can inspect only the first row of your Dataset.

```
val device = dsTemp.first()
            print(device)
            DeviceTempByCountry(34,meter-gauge-1xbYRYcj,1,USA)device: Device-
TempByCountry = DeviceTempByCountry(34,meter-gauge-1xbYRYcj,1,USA)
```

Alternatively, you could express the same query using column names and then cast to a Dataset[DeviceTempByCountry].

```
// In Scala
            val dsTemp2 = ds.select($"temp", $"device_name", $"device_id",
$"device_id", $"cca3").where("temp > 25").as[DeviceTempByCountry]
```

Tip: Semantically, select() is like map() in the above queries. In that, both these queries select fields and generate equivalent results.

To recap, the operations—filter(), map(), groupBy(), select(), take(), etc—on Datasets are similar to the ones on DataFrames. In a way, Datasets are similar to RDDs in that they provide a similar interface to its aforementioned methods and compile-time safety but with a much easier to read and an object-oriented programming interface.

When using Datasets, the underlying Spark SQL engine does all the creation, conversion, serialization and deserialization of JVM objects, and off-Java heap memory management with the help of Dataset Encoders.[30] [31] (We will cover more about Datasets and memory management in chapter 6.)

End-to-End Dataset Example

In this end-to-end Dataset example we conduct similar exploratory data analysis, ETL (extract, transform, and load), and perform common data operations on IoT dataset. Although the dataset is small and fake , we want to illustrate the readability and clarity with which you can express a query with Datasets, just as we did with DataFrames.

For brevity, we won't include the entire example code here. However, we have furnished links to a Databricks Community Edition Notebook for you to try. In short, the notebook explores following common operations you might conduct for this dataset. Using Dataset's structured API, we attempt the following:

detect failing devices with low battery below a threshold;

identify offending countries with high-levels of C02 emissions;

compute the min and max values for temperature, battery_level, C02, and humidity; and

sort and group by average temperature, C02, humidity, and country

For a complete Scala notebook, import the notebook from the Learning Spark GitHub link.

30 https://databricks.com/session/deep-dive-apache-spark-memory-management

31 https://www.youtube.com/watch?v=-Aq1LMpzaKw

DataFrames vs Datasets

By now you may be asking why and when should I use DataFrames or Datasets. The reasons are handful, and here are a few notable ones [32] [33]:

If you want rich semantics, high-level abstractions, and domain specific language operators, use DataFrame or Dataset.

If you want strict compile type safety and don't mind creating multiple case classes for specific Dataset[T], use Dataset.

If your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of relational operators on semi-structured data, use DataFrame or Dataset.

If your processing dictates relational transformation similar to SQL like queries, use DataFrames.

If you want a higher degree of type-safety at compile time, want typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation and serialization with Encoders use Dataset.[34]

If you want unification, code optimization, simplification of APIs across Spark Libraries, use DataFrame.[35]

If you are an R user, use DataFrames.

If you are a Python user, use DataFrames and at will drop down to RDDs if you need more control.

If you want space and speed efficiency use DataFrames or Datasets.

If you want errors caught during compile time vs analysis time, choose the appropriate API as depicted in Figure 3-2.

Finally, and importantly, if you want to instruct Spark *what-to-do*, not *how-to-do*, use DataFrame or Dataset.

32 https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

33 https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html

34 https://databricks.com/session/demystifying-dataframe-and-dataset

35

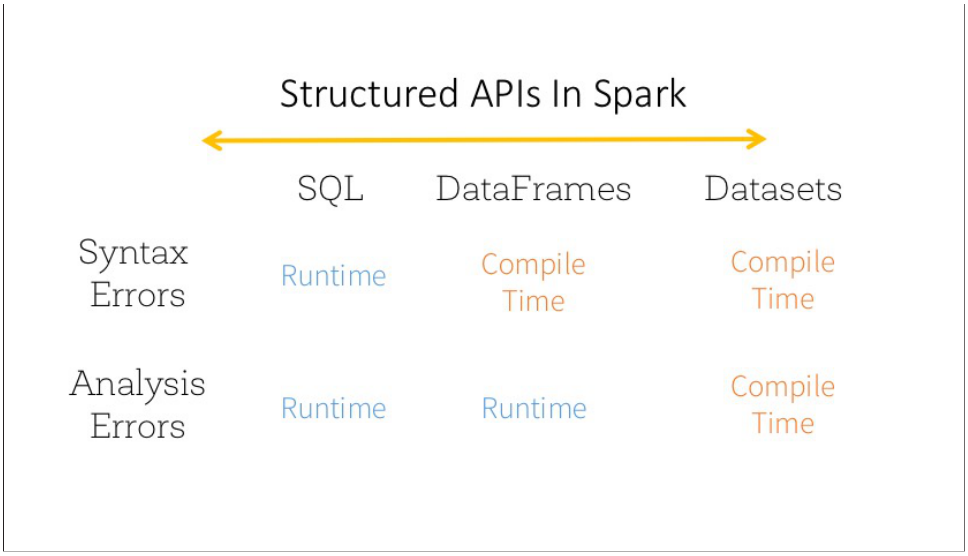


Figure 1-2. Spectrum of errors detected using Structured APIs

What about RDDs?

You may ask: Are RDDs being relegated as second class citizens? Are they being deprecated? And the answer is a resounding NO!

All future development work, such as DataSource API v2 in Spark 2.x and Spark 3.0, will continue to have a DataFrame interface and semantics rather than RDDs.

So Then When to use RDDs?

Consider these scenarios for using RDDs:

you are using a third-party package that's written in RDD;

you can forgo code optimization, efficient space utilization, and performance benefits available with DataFrames and Datasets; and

you want to precisely instruct Spark *how-to-do* a query

What's more, you can seamlessly move between DataFrame or Dataset and RDDs at will—by a simple API method call df.rdd. (Note that moving back-and-forth between RDD and DataFrame has its cost and should be avoided unless necessary.) After all, DataFrames and Datasets are built on top of RDDs, and they get decomposed to compact RDDs code during whole-stage-code-generation (we discuss this in the next section).

Finally, in the above sections with DataFrames, Datasets, and SQL, we got an intuition in how structure enables developers to use easy and friendly APIs to compose expressive queries on structured data. In other words, you instruct Spark *what-to-do*, not *how-to-do* using high-level operations and Spark ascertains what's the most efficient way to build a query and generate compact code for you.

This process of building the most efficient query and generating compact code is the job of Spark SQL engine. It's the substrate upon which the above structured APIs are built. Let's peek under the hood to understand Spark SQL engine.

Spark SQL and the Underlying Engine

At the programmatic level, Spark SQL allows developers to issue ANSI SQL 2003-compatible queries to structured data with a schema. Since its introduction in Spark 1.3, Spark SQL has evolved over the Spark 2.x releases and is now a substantial engine upon which many high-level structured functionalities have been built. Apart from allowing you to issue SQL-like queries to your data, Spark SQL provides a few main capabilities as shown in Figure 3-3:

Unifies Spark components and offers abstraction to DataFrames/Datasets in Java, Scala, Python, and R, which simplifies working with structured datasets

Connects to Apache Hive metastore and tables

Reads and writes structured data with a specific schema from structured file formats (JSON, CSV, Text, Avro, Parquet, ORC etc) and converts them into temporary tables

Offers an interactive Spark SQL shell for quick data exploration

Offers a bridge to (and from) external tools via standard database JDBC/ODBC connectors

Generates optimized query plan and compact-code for the JVM for final execution

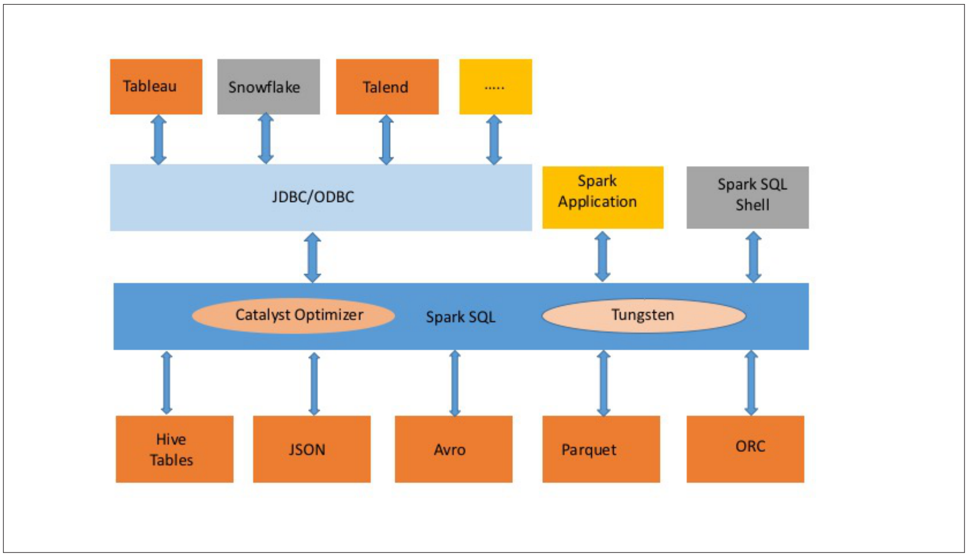


Figure 1-3. Spark SQL and its stack

Under the hood, the Spark SQL engine has the Catalyst Optimizer and Project Tungsten[36] [37]Together, support high-level structured DataFrame and Datasets APIs and SQL queries. Let's take a closer look at the Catalyst Optimizer.

Catalyst Optimizer

The Catalyst Optimizer takes a computational query, whether in SQL, DataFrame or Dataset, and converts into an execution plan, as it undergoes four transformational phases, as shown in Figure 3-4 [38]:

Analysis

Logical Plan and Optimization

Physical Planning

Code Generation

36 https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

37 https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

38 https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

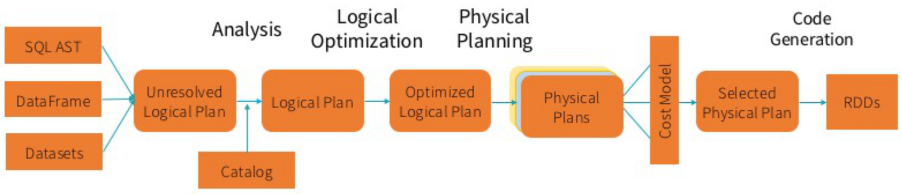


Figure 1-4. Spark Computation's Four-Phase Journey

For example, consider the DataFrame (we discuss DataFrame below) computation or SQL query from our M&M example from Chapter 2; its queries traverse and transform across the four phases. Both sample code blocks below will eventually end up with a similar query plan and identical byte code for execution. That is, if you write to the Spark 2.x Structured API, regardless of the supported language, your computation undergoes the same journey and the resulting bytecode is the same.

```
# In Python
        count_mnm_df = (mnm_df.select("State", "Color", "Count")
         .groupBy("State", "Color")
         .agg(count("Count")
         .alias("Total"))
         .orderBy("Total", ascending=False))
        -- In SQL
        SELECT State, Color, Count sum(Count) AS Total
        FROM MNM_TABLE_NAME
        GROUP BY State, Color, Count
        ORDER BY Total DESC
```

To look at the different stages of the Python code, you can use count_mnm_df.explain(True) method on the DataFrame. (In Chapter 8, we will discuss more about tuning and debugging Spark and how to read query plans.) This gives us the following output:

```
== Parsed Logical Plan ==
        'Sort ['Total DESC NULLS LAST], true
        +- Aggregate [State#10, Color#11], [State#10, Color#11, count(Count#12)
AS Total#24L]
```

```
        +- Project [State#10, Color#11, Count#12]
        +- Relation[State#10,Color#11,Count#12] csv
       == Analyzed Logical Plan ==
       State: string, Color: string, Total: bigint
       Sort [Total#24L DESC NULLS LAST], true
       +- Aggregate [State#10, Color#11], [State#10, Color#11, count(Count#12)
AS Total#24L]
        +- Project [State#10, Color#11, Count#12]
        +- Relation[State#10,Color#11,Count#12] csv
       == Optimized Logical Plan ==
       Sort [Total#24L DESC NULLS LAST], true
       +- Aggregate [State#10, Color#11], [State#10, Color#11, count(Count#12)
AS Total#24L]
        +- Relation[State#10,Color#11,Count#12] csv
       == Physical Plan ==
       *(3) Sort [Total#24L DESC NULLS LAST], true, 0
       +- Exchange rangepartitioning(Total#24L DESC NULLS LAST, 200)
        +- *(2) HashAggregate(keys=[State#10, Color#11], func-
tions=[count(Count#12)], output=[State#10, Color#11, Total#24L])
        +- Exchange hashpartitioning(State#10, Color#11, 200)
        +- *(1) HashAggregate(keys=[State#10, Color#11], functions=[par-
tial_count(Count#12)], output=[State#10, Color#11, count#29L])
        +- *(1) FileScan csv [State#10,Color#11,Count#12] Batched: false, For-
mat: CSV, Location: InMemoryFileIndex[file:/Users/jules/gits/LearningSpark2.0/
chapter2/py/src/data/mnm_dataset.csv], PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<State:string,Color:string,Count:int>
```

Consider another DataFrame computation example, the below Scala code, that undergoes a similar journey and how the underlying engine optimizes its logical and physical plan.

```
// In Scala
       // Users DataFrame read from a Parquet Table
       val usersDF = …
       // Events DataFrame read from Parquet Table
       val eventsDF = ...
       // Join two DataFrames
       val joinedDF = users.join(events, users("id") === events("uid"))
        .filter(events("date") > "2015-01-01")
```

After going through an initial analysis phase, the query plan is transformed and rearranged by the Catalyst Optimizer as shown in the following diagram.

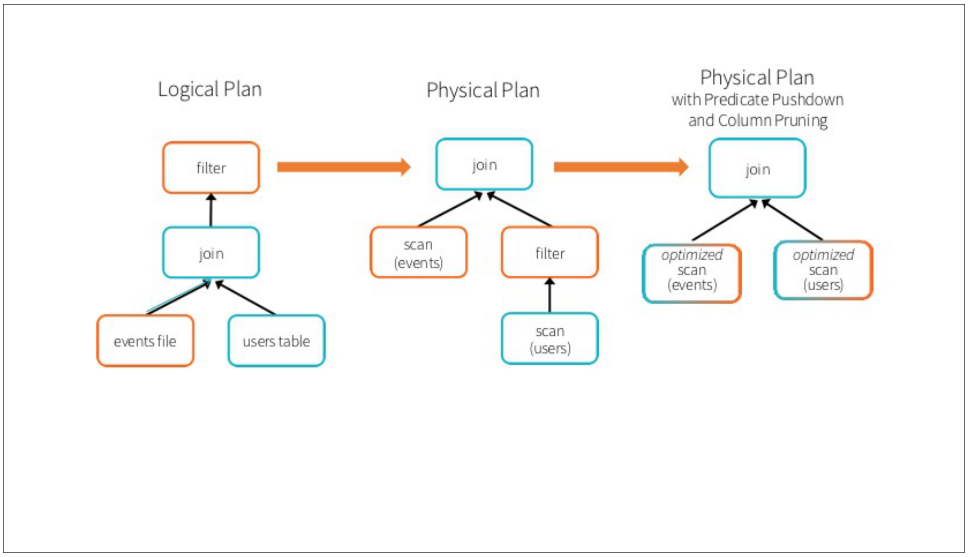


Figure 1-5. An Example of a specific query transformation

Let's go through each of the four phases of a Spark computation.

Phase 1: Analysis

In all the above code examples, Spark SQL engine will generate an Abstract Syntax Tree (AST)[39] for the SQL or DataFrame query. In this phase, any columns or table names will be resolved by consulting an internal Catalog, a programmatic interface to Spark SQL that holds a list of names of columns, data types, functions, tables, databases, etc. After all successful resolutions, the query proceeds to the next phase.

Phase 2: Logical Optimizations

As Figure 3-4 shows, this phase comprises two internal stages. Applying a standard-rule based optimization, the Catalyst Optimizer will first construct a set of multiple plans and then, using cost-based optimizer (CBO), assign costs to each plan. [40] These plans are laid out as operator trees (like Figure 3-5); they may include, for example, the process of constant folding, predicate pushdown, projection pruning, boolean expression simplification, etc. [41] (More on these concepts in chapter 8). For each of the trees, a computation cost is associated. This is the input into the physical plan.

39 https://en.wikipedia.org/wiki/Abstract_syntax_tree

40 https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html

41 https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html

Phase 3: Physical Planning

In the physical planning phase, Spark SQL picks a logical plan and generates one physical plan, using physical operators that match the Spark execution engine.

Phase 4: Code Generation

The final phase of query optimization involves generating efficient Java bytecode to run on each machine. Because Spark SQL can operate datasets loaded in memory datasets, Spark can use a state-of-the-art compiler technology for code generation to speed up execution. In other words, it acts as a compiler. Project Tungsten, which facilitates whole-stage code-generation, plays a role here.[42]

Whole-Stage Code-Generation . At a high-level, this aspect of physical planning in the query's journey is really the second generation of Project Tungsten[43]. Introduced in Spark 2.0, it employs the latest compiler generation techniques to generate the compact-code RDD code for final execution. "Built upon ideas from modern compilers and MPP databases and applied to data processing queries, it emits optimized bytecode in the last phase by collapsing the entire query into a single function, eliminating virtual function calls and leveraging CPU registers for intermediate data."

As a result of this streamlined strategy called "whole-stage code generation," we significantly improve CPU efficiency and improve performance.[44]

Note: We have talked at a conceptual level the workings of the Spark SQL engine, with its two principal components: Catalyst Optimizer and Project Tungsten. The internal technical workings are beyond the scope of this book; however, for the curious, we encourage you to check out the references for in-depth technical discussions.

Summary

In this chapter, we covered a short history of structure in Spark—its merits, justification, and simplification in Spark 2.x.

Through illustrative common data operations and code examples, we demonstrated that high-level DataFrames and Datasets APIs are far more expressive and intuitive than low-level RDDs APIs. Designed to make large data sets processing easier, high-level structured APIs provide domain specific operators for common data operations.

42 https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

43 https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

44 https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

We shared when to use RDDs, DataFrames, and Datasets, depending on your use case scenarios.

And finally, we explored under the hood how Spark SQL engine's components—Catalyst Optimizer and Project Tungsten—support structured high-level APIs and DSL operators: no matter what Spark's supported language you use, a Spark query undergoes the same journey, from logical and physical plan construction to final compact code generation.

This chapter's concepts and code examples have laid the context for the next two chapters, in which we will further show the seamless interoperability between DataFrames, Datasets, and Spark SQL.

About the Authors

Jules S. Damji is an Apache Spark Community and Developer Advocate at Databricks. He is a hands-on developer with over 20 years of experience and has worked at leading companies, such as Sun Microsystems, Netscape, @Home, LoudCloud/Opsware, VeriSign, ProQuest, and Hortonworks, building large-scale distributed systems. He holds a B.Sc and M.Sc in Computer Science and MA in Political Advocacy and Communication from Oregon State University, Cal State, and Johns Hopkins University respectively.

Denny Lee is a Technical Product Manager at Databricks. He is a hands-on distributed systems and data sciences engineer with extensive experience developing internet-scale infrastructure, data platforms, and predictive analytics systems for both on-premise and cloud environments. He also has a Masters of Biomedical Informatics from Oregon Health and Sciences University and has architected and implemented powerful data solutions for enterprise Healthcare customers. His current technical focuses include Distributed Systems, Apache Spark, Deep Learning, Machine Learning, and Genomics.

Brooke Wenig is the Machine Learning Practice Lead at Databricks. She advises and implements machine learning pipelines for customers, as well as educates them on how to use Spark for Machine Learning and Deep Learning. She received an MS in Computer Science from UCLA with a focus on distributed machine learning. She speaks Mandarin Chinese fluently and enjoys cycling.

Tathagata Das is an Apache Spark committer and a member of the PMC. He's the lead developer behind Spark Streaming and currently develops Structured Streaming. Previously, he was a grad student in the UC Berkeley at AMPLab, where he conducted research about data-center frameworks and networks with Scott Shenker and Ion Stoica.