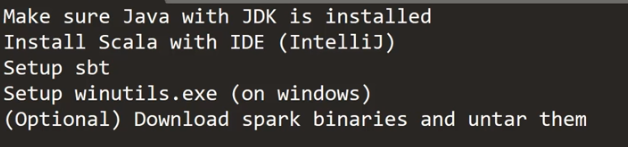
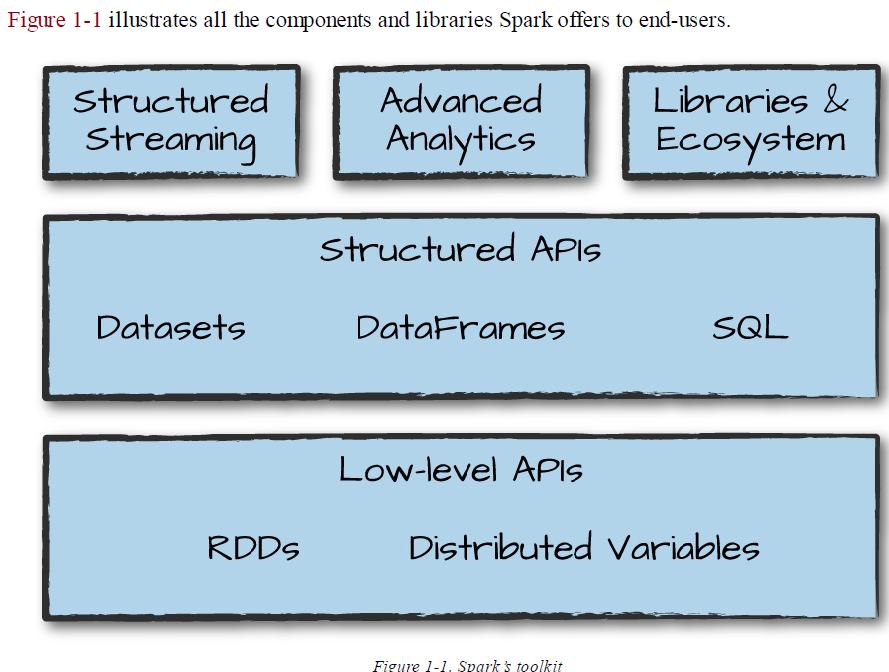
**Spark 2**

**Steps to set up spark locally:**

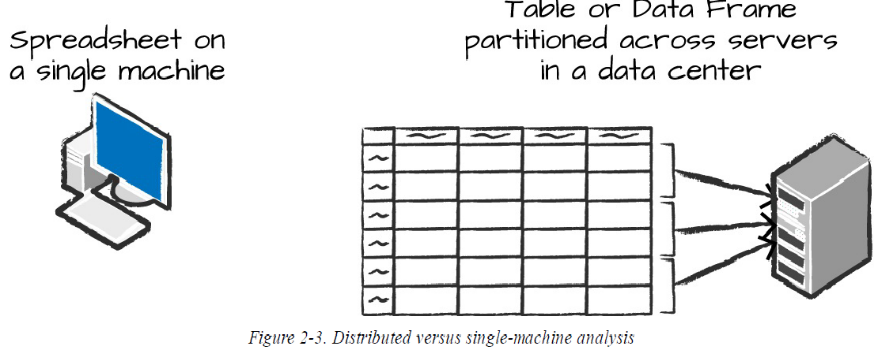


**What components are present in spark-tool kit?**



**What is a dataframe?**

A DataFrame is the most common Structured API and simply represents a table of data with rows and columns. The list that defines the columns and the types within those columns is called the schema. You can think of a DataFrame as a spreadsheet with named columns. Figure 2-3 illustrates the fundamental difference: a spreadsheet sits on one computer in one specific location, whereas a Spark DataFrame can span thousands of computers. The reason for putting the data on more than one computer should be intuitive: either the data is too large to fit on one machine or it would simply take too long to perform that computation on one machine. Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). These different abstractions all represent distributed collections of data. The easiest and most efficient are DataFrames, which are available in all languages.



**Why data is partitioned?**

To allow every executor to perform work in parallel, Spark breaks up the data into chunks called partitions. A partition is a collection of rows that sit on one physical machine in your cluster. A DataFrame’s partitions represent how the data is physically distributed across the cluster of machines during execution. If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors. If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource. An important thing to note is that with DataFrames you do not (for the most part) manipulate partitions manually or individually. You simply specify high-level transformations of data in the physical partitions, and Spark determines how this work will actually execute on the cluster.

**What are transformations?**

In Spark, the core data structures are immutable, meaning they cannot be changed after they’re created. This might seem like a strange concept at first: if you cannot change it, how are you supposed to use it? To “change” a DataFrame, you need to instruct Spark how you would like to modify it to do what you want. These instructions are called transformations. Let’s perform a simple transformation to find all even numbers in our current DataFrame:

// in Scala

val myRange = spark.range(1000).toDF(“number”)

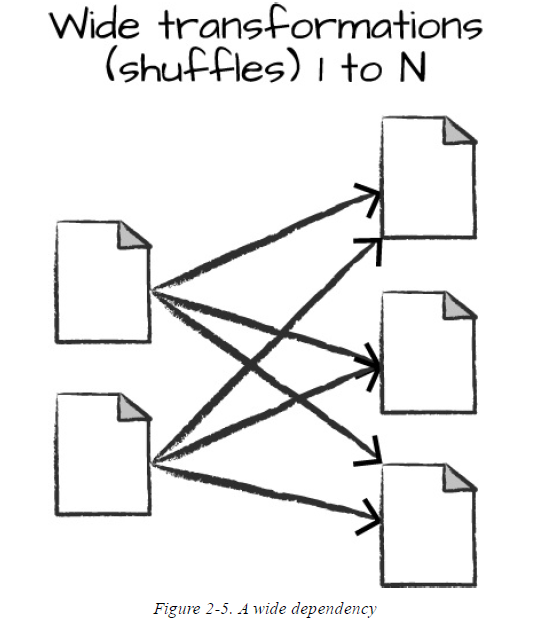
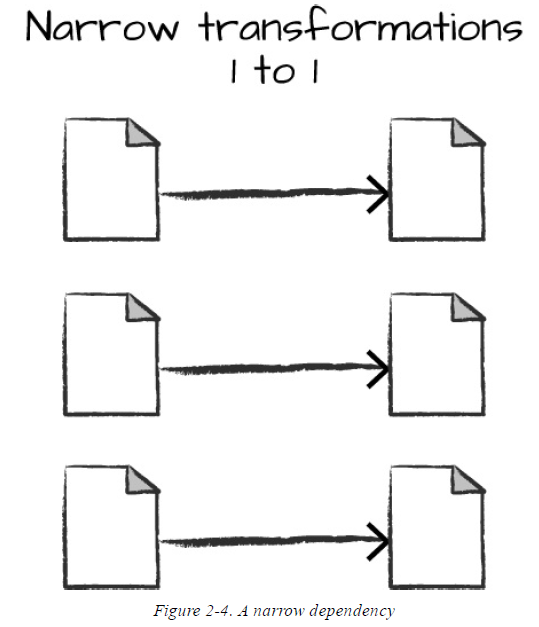
val divisBy2 = myRange.where("number % 2 = 0")

# in Python

divisBy2 = myRange.where("number % 2 = 0")

Notice that these return no output. This is because we specified only an abstract transformation, and Spark will not act on transformations until we call an action (we discuss this shortly). Transformations are the core of how you express your business logic using Spark. There are two types of transformations: those that specify narrow dependencies, and those that specify wide dependencies.

Transformations consisting of narrow dependencies (we’ll call them narrow transformations) are those for which each input partition will contribute to only one output partition. In the preceding code snippet, the where statement specifies a narrow dependency, where only one partition contributes to at most one output partition, as you can see in Figure 2-4.



A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions. You will often hear this referred to as a shuffle whereby Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called pipelining, meaning that if we specify multiple filters on DataFrames,they’ll all be performed in-memory. The same cannot be said for shuffles. When we perform a shuffle, Spark writes the results to disk. Wide transformations are illustrated in Figure 2-5.

**What is Lazy Evaluation?**

Lazy evaulation means that Spark will wait until the very last moment to execute the graph of computation instructions. In Spark, instead of modifying the data immediately when you express some operation, you build up a plan of transformations that you would like to apply to your source data. By waiting until the last minute to execute the code, Spark compiles this plan from your raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster. This provides immense benefits because Spark can optimize the entire data flow from end to end. An example of this is something called predicate pushdown on DataFrames. If we build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need. Spark will actually optimize this for us by pushing the filter down automatically.

**What are actions in Spark?**

Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an action. An action instructs Spark to compute a result from a series of transformations. The simplest action is count, which gives us the total number of records in the DataFrame:

divisBy2.count()

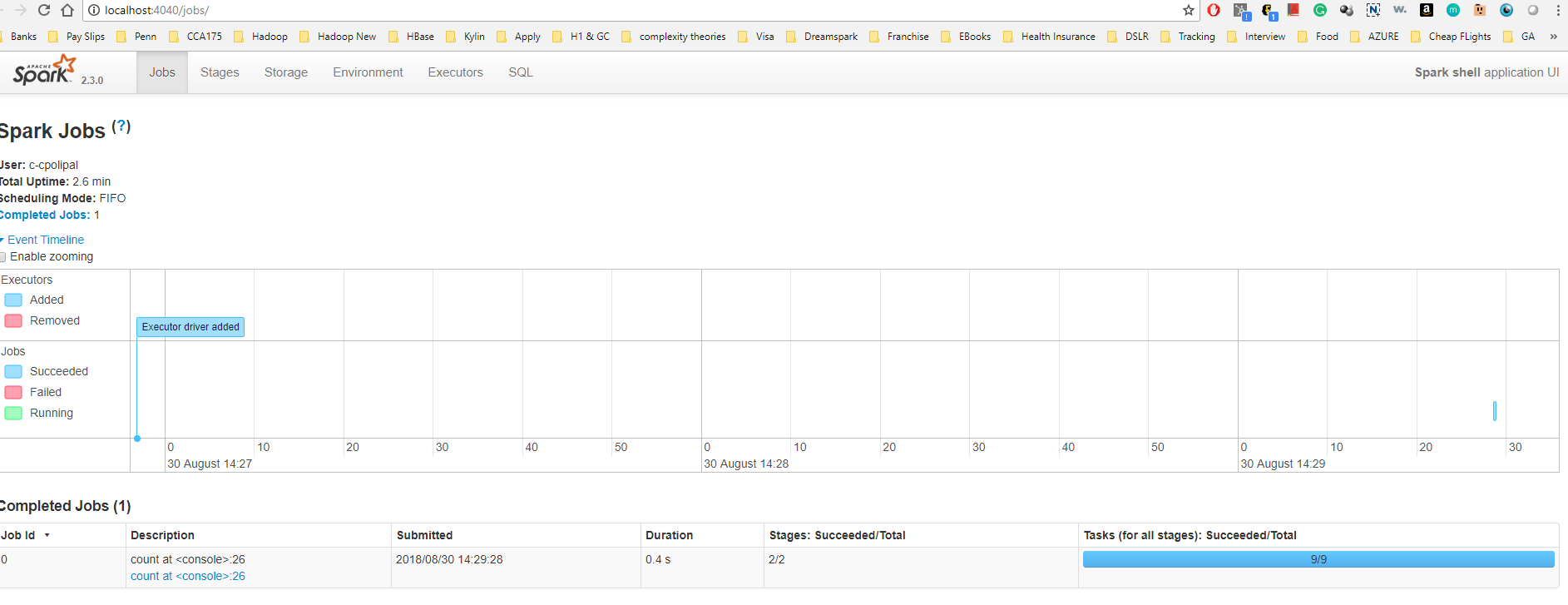
The output of the preceding code should be 500. Of course, count is not the only action. There are three kinds of actions:

* Actions to view data in the console
* Actions to collect data to native objects in the respective language
* Actions to write to output data sources

In specifying this action, we started a Spark job that runs our filter transformation (a narrow transformation), then an aggregation (a wide transformation) that performs the counts on a per partition basis, and then a collect, which brings our result to a native object in the respective language. You can see all of this by inspecting the Spark UI, a tool included in Spark with which you can monitor the Spark jobs running on a cluster.

**What is Spark UI?**

You can monitor the progress of a job through the Spark web UI. The Spark UI is available on port 4040 of the driver node. If you are running in local mode, this will be http://localhost:4040. The Spark UI displays information on the state of your Spark jobs, its environment, and cluster state. It’s very useful, especially for tuning and debugging. Figure 2-6 shows an example UI for a Spark job where two stages containing nine tasks were executed.



At this point, all you need to understand is that a Spark job represents a set of transformations triggered by an individual action, and you can monitor that job from the Spark UI.

**What is schema inference?**

Spark includes the ability to read and write from a large number of data sources. To read this data, we will use a DataFrameReader that is associated with our SparkSession. In doing so, we will specify the file format as well as any options we want to specify. In our case, we want to do something called schema inference, which means that we want Spark to take a best guess at what the schema of our DataFrame should be.

#Examples from Definitive guide

val flightData = spark.read.

option("inferSchema","true").option("header","true").csv("/user/chaitanyapolipalli/flight-data/csv/2010-summary.csv")

#Sorting

val flightDataSort = flightData.sort("count").show //ascending order

val

flightDataSortDesc = flightData.sort($"count".desc).show //descending order

spark.conf.set("spark.sql.shuffle.partitions","5")

#Registering data frame as table or view

flightData.createOrReplaceTempView("flight\_data")

val sqlWay = spark.sql("select DEST\_COUNTRY\_NAME, count(1) from flight\_data group by DEST\_COUNTRY\_NAME")

val dfWay = flightData.groupBy("DEST\_COUNTRY\_NAME").count()

sqlWay.explain & dfWay.explain should give you similar physical plan

#Using max function

spark.sql("select max(count) from flight\_data").take(1)

flightData.select(max("count")).take(1)

#Find the top five destination countries in the data

spark.sql("select DEST\_COUNTRY\_NAME, sum(count) as destination\_total from flight\_data group by DEST\_COUNTRY\_NAME order by destination\_total desc limit 5").show

flightData.groupBy("DEST\_COUNTRY\_NAME").sum("count").withColumnRenamed("sum(count)","destination\_total").sort($"destination\_total".desc).limit(5).show

or

flightData.groupBy("DEST\_COUNTRY\_NAME").sum("count").withColumnRenamed("sum(count)","destination\_total").sort(desc("destination\_total")).limit(5).show

**What are type safe structured api’s?**

Type-safe version of Spark’s structured API called Datasets, for writing statically typed code in Java and Scala. The Dataset API is not available in Python and R, because those languages are dynamically typed.

Recall that DataFrames, which we saw in the previous chapter, are a distributed collection of objects of type Row that can hold various types of tabular data. The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java ArrayList or Scala Seq. The APIs available on Datasets are type-safe, meaning that you cannot accidentally view the objects in a Dataset as being of another class than the class you put in initially. This makes Datasets especially attractive for writing large applications, with which multiple software engineers must interact through well-defined interfaces.

The Dataset class is parameterized with the type of object contained inside: Dataset<T> in Java and Dataset[T] in Scala. For example, a Dataset[Person] will be guaranteed to contain objects of class Person. As of Spark 2.0, the supported types are classes following the JavaBean pattern in Java and case classes in Scala. These types are restricted because Spark needs to be able to automatically analyze the type T and create an appropriate schema for the tabular data within your Dataset.

Eg:

#Case Class Example

case class Flight(DEST\_COUNTRY\_NAME: String,ORIGIN\_COUNTRY\_NAME: String,count: BigInt)

val flightsDF =

spark.read.parquet("/user/chaitanyapolipalli/flight-data/parquet/2010-summary.parquet/")

val flights = flightsDF.as[Flight]

flights.filter

(x =>

x.ORIGIN\_COUNTRY\_NAME != "Canada").map(x => x).take(5).foreach(println)

**What are Lower-Level API’s in Spark?**

Spark includes a number of lower-level primitives to allow for arbitrary Java and Python object manipulation via Resilient Distributed Datasets (RDDs). Virtually everything in Spark is built on top of RDDs. DataFrame operations are built on top of RDDs and compile down to these lower-level tools for convenient and extremely efficient distributed execution. There are some things that you might use RDDs for, especially when you’re reading or manipulating raw data, but for the most part you should stick to the Structured APIs. RDDs are lower level than DataFrames because they reveal physical execution characteristics (like partitions) to end users. One thing that you might use RDDs for is to parallelize raw data that you have stored in memory on the driver machine.

**What are Structured API?**

The Structured APIs are a tool for manipulating all sorts of data, from unstructured log files to semi-structured CSV files and highly structured Parquet files. These APIs refer to three core types of distributed collection APIs:

1. Datasets
2. DataFrames
3. SQL tables and views

**What is a Schema?**

A schema defines the column names and types of a DataFrame. You can define schemas manually or read a schema from a data source (often called schema on read). Schemas consist of types, meaning that you need a way of specifying what lies where.

**Difference between DataFrames and DataSets?**

In essence, within the Structured APIs, there are two more APIs, the “untyped” DataFrames and the “typed” Datasets. To say that DataFrames are untyped is aslightly inaccurate; they have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at runtime. Datasets, on the other hand, check whether types conform to the specification at compile time. Datasets are only available to Java Virtual Machine (JVM)–based languages (Scala and Java) and we specify types with case classes or Java beans.

For the most part, you’re likely to work with DataFrames. To Spark (in Scala), DataFrames are simply Datasets of Type Row. The “Row” type is Spark’s internal representation of its optimized inmemory format for computation. This format makes for highly specialized and efficient computation because rather than using JVM types, which can cause high garbage-collection and object instantiation costs, Spark can operate on its own internal format without incurring any of those costs. To Spark (in Python or R), there is no such thing as a Dataset: everything is a DataFrame and therefore we always operate on that optimized format.

**Overview of structured API execution:**

This section will demonstrate how this code is actually executed across a cluster. This will help you understand (and potentially debug) the process of writing and executing code on clusters, so let’s walk through the execution of a single structured API query from user code to executed code. Here’s an overview of the steps:

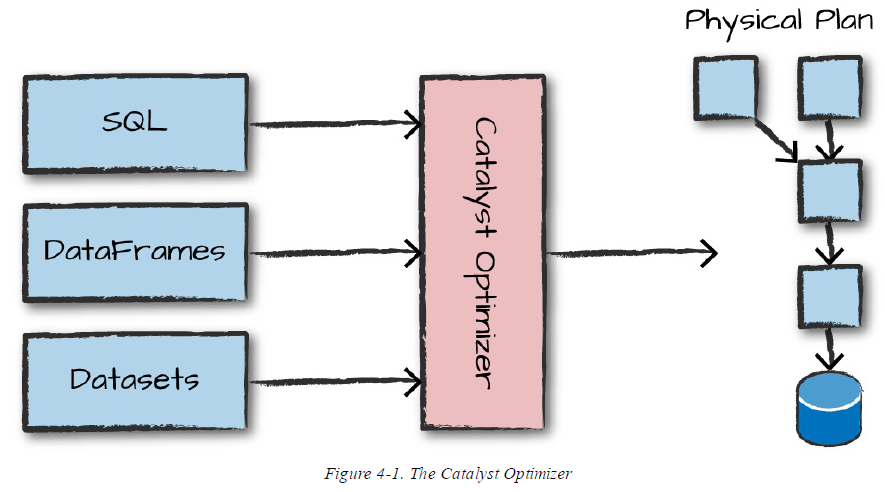
1. Write DataFrame/Dataset/SQL Code.

2. If valid code, Spark converts this to a Logical Plan.

3. Spark transforms this Logical Plan to a Physical Plan, checking for optimizations along the way.

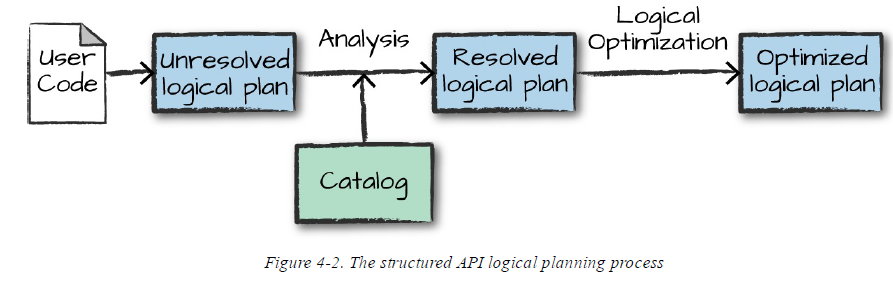
4. Spark then executes this Physical Plan (RDD manipulations) on the cluster.

To execute code, we must write code. This code is then submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user. Figure 4-1 shows the process.



**Logical Planning**

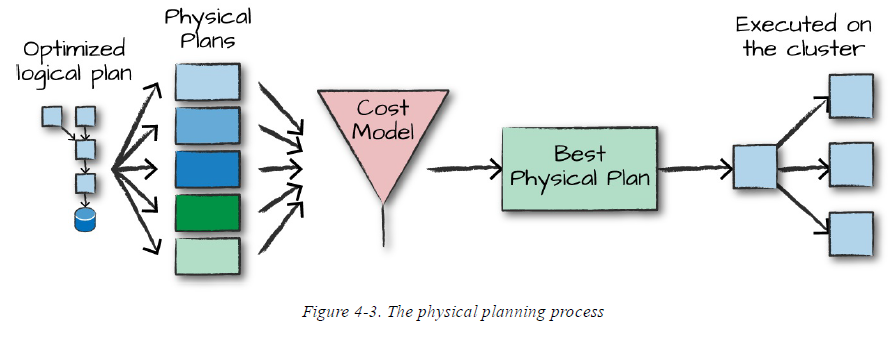
The first phase of execution is meant to take user code and convert it into a logical plan. Figure 4-2 illustrates this process.



This logical plan only represents a set of abstract transformations that do not refer to executors or drivers, it’s purely to convert the user’s set of expressions into the most optimized version. It does this by converting user code into an unresolved logical plan. This plan is unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist. Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer. The analyzer might reject the unresolved logical plan if the required table or column name does not exist in the catalog. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections. Packages can extend the Catalyst to include their own rules for domainspecific optimizations.

**Physical Planning:**

After successfully creating an optimized logical plan, Spark then begins the physical planning process. The physical plan, often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model, as depicted in Figure 4-3. An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are).



Physical planning results in a series of RDDs and transformations. This result is why you might have heard Spark referred to as a compiler—it takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations for you.

**Execution:**

Upon selecting a physical plan, Spark runs all of this code over RDDs, the lower-level programming interface of Spark (which we cover in Part III). Spark performs further optimizations at runtime, generating native Java bytecode that can remove entire tasks or stages during execution. Finally the result is returned to the user.

**What is catalog in spark?**

Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer. The analyzer might reject the unresolved logical plan if the required table or column name does not exist in the catalog. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections.

**What is Schema in spark?**

Any DataFrame consists of a series of records (like rows in a table), that are of type Row, and a number of columns (like columns in a spreadsheet) that represent a computation expression that can be performed on each individual record in the Dataset. Schemas define the name as well as the type of data in each column. A schema is a StructType made up of a number of fields, StructFields, that have a name, type, a Boolean flag which specifies whether that column can contain missing or null values, and, finally, users can optionally specify associated metadata with that column. The metadata is a way of storing information about this column (Spark uses this in its machine learning library). Schemas can contain other StructTypes (Spark’s complex types). We will see this in Chapter 6 when we discuss working with complex types. If the types in the data (at runtime) do not match the schema, Spark will throw an error. The example that follows shows how to create and enforce a specific schema on a DataFrame.

Eg:

df.printSchema()

or

spark.read.format("json").load("/data/flight-data/json/2015-summary.json").schema

or

val myManualSchema = StructType(Array

(StructField

("DEST\_COUNTRY\_NAME", StringType, true),StructField("ORIGIN\_COUNTRY\_NAME", StringType, true),StructField("count", LongType,false,Metadata.fromJson("{\"hello\":\"world\"}"))

))

val df =

spark.read.format("json").schema(myManualSchema).load("/data/flight-data/json/2015-summary.json")

**What is partitioning?**

Partitioning of the DataFrame defines the layout of the DataFrame or Dataset’s physical distribution across the cluster. The partitioning scheme defines how that is allocated. You can set this to be based on values in a certain column or nondeterministically.

**What are columns and expressions in Spark?**

Columns in Spark are similar to columns in a spreadsheet, R dataframe, or pandas DataFrame. You can select, manipulate, and remove columns from DataFrames and these operations are represented as expressions.

To Spark, columns are logical constructions that simply represent a value computed on a per-record basis by means of an expression. This means that to have a real value for a column, we need to have a row; and to have a row, we need to have a DataFrame. You cannot manipulate an individual column outside the context of a DataFrame; you must use Spark transformations within a DataFrame to modify the contents of a column.

**How to access columns:**

spark.read.json("/user/chaitanyapolipalli/flight-data/json/2010-summary.json").columns

res2: Array[String] = Array(DEST\_COUNTRY\_NAME, ORIGIN\_COUNTRY\_NAME, count)

**How to create dataframes on the fly?**

import org.apache.spark.sql.Row

import org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}

val myManualSchema = new StructType(Array(new StructField("some", StringType, true),new StructField("col", StringType, true),new StructField("names", LongType, false)))

val myRows = Seq(Row("Hello", null, 1L))

val myRDD = spark.sparkContext.parallelize(myRows)

val myDf = spark.createDataFrame(myRDD, myManualSchema)

myDf.show()

**Some useful dadtaframe methods:**

#Examples of 'select' and 'selectExpr' methods

df.select("DEST\_COUNTRY\_NAME").show(2)

You can select multiple columns by using the same style of query, just add more column name strings to your select method call:

df.select("DEST\_COUNTRY\_NAME", "ORIGIN\_COUNTRY\_NAME").show(2)

#Multiple ways of printing column

import org.apache.spark.sql.functions.{expr, col, column}

df.select(

df.col("DEST\_COUNTRY\_NAME"),

col("DEST\_COUNTRY\_NAME"),

column("DEST\_COUNTRY\_NAME"),

'DEST\_COUNTRY\_NAME,

$"DEST\_COUNTRY\_NAME",

expr("DEST\_COUNTRY\_NAME"))

.show(2)

#One common error is attempting to mix Column objects and strings. For example, the following code will result in a compiler error:

df.select(col("DEST\_COUNTRY\_NAME"), "DEST\_COUNTRY\_NAME")

#As we’ve seen thus far, expr is the most flexible reference that we can use. It can refer to a plain column or a string manipulation of a column. To illustrate, let’s change the column name, and then change it back by using the AS keyword and then the alias method on the column:

df.select(expr("DEST\_COUNTRY\_NAME AS destination")).show(2)

#This changes the column name to “destination.” You can further manipulate the result of your expression as another expression:

df.select(expr("DEST\_COUNTRY\_NAME as destination").

alias("DEST\_COUNTRY\_NAME")).show(2)

#The preceding operation changes the column name back to its original name.Because select followed by a series of expr is such a common pattern, Spark has a shorthand for doing this efficiently: selectExpr. This is probably the most convenient interface for everyday use:

df.selectExpr

("DEST\_COUNTRY\_NAME as newColumnName", "DEST\_COUNTRY\_NAME").show(2)

#This opens up the true power of Spark. We can treat selectExpr as a simple way to build up complex expressions that create new DataFrames. In fact, we can add any valid non-aggregating SQL statement, and as long as the columns resolve, it will be valid! Here’s a simple example that adds a new column withinCountry to our DataFrame that specifies whether the destination and origin are the same:

df.selectExpr

("\*",

"(DEST\_COUNTRY\_NAME = ORIGIN\_COUNTRY\_NAME) as withinCountry").show(2)

#With select expression, we can also specify aggregations over the entire DataFrame by taking advantage of the functions that we have. These look just like what we have been showing so far:

df.selectExpr("avg(count)", "count(distinct(DEST\_COUNTRY\_NAME))").show

#Sometimes, we need to pass explicit values into Spark that are just a value (rather than a new column). This might be a constant value or something we’ll need to compare to later on.The way we do this is through literals. This is basically a translation from a given programming language’s literal value to one that Spark understands. Literals are expressions and you can use them in the same way:

import org.apache.spark.sql.functions.lit

df.select(expr("\*"), lit(1).as("One")).show(2)

#There’s also a more formal way of adding a new column to a DataFrame, and that’s by using the withColumn method on our DataFrame. For example, let’s add a column that just adds the number one as a column:

df.withColumn("numberOne", lit(1)).show(2)

df.withColumn

("withInCountry",expr("ORIGIN\_COUNTRY\_NAME == DEST\_COUNTRY\_NAME"))

#Renaming columns

df.withColumnRenamed("DEST\_COUNTRY\_NAME", "dest").columns

#Escaping column names appropriately. In Spark, we do this by using backtick(`) characters.In this example, however, we need to use backticks because we’re referencing a column in an expression:

val dfWithLongColName =

df.withColumn("This Long Column-Name",expr("ORIGIN\_COUNTRY\_NAME"))

dfWithLongColName.selectExpr("`This Long Column-Name`","`This Long Column-Name` as `new col`").show(2)

#By default Spark is case insensitive; however, you can make Spark case sensitive by setting the configuration:

set spark.sql.caseSensitive true

#Removing columns

df.drop("ORIGIN\_COUNTRY\_NAME")

dfWithLongColName.drop("ORIGIN\_COUNTRY\_NAME", "DEST\_COUNTRY\_NAME")

#Changing column type

df.withColumn("count2", col("count").cast("long"))

#Filtering rows - below commands will yield same results

df.filter(col("count") < 2).show(2)

df.where("count < 2").show(2)

#Instinctually, you might want to put multiple filters into the same expression. Although this is possible,it is not always useful, because Spark automatically performs all filtering operations at the same time regardless of the filter ordering. This means that if you want to specify multiple AND filters, just chain them sequentially and let Spark handle the rest:

#In Scala, you must use the =!= operator so that you don’t just compare the unevaluated column expression to a string but instead to the evaluated one:

df.where

(col("count") < 2).where(col("ORIGIN\_COUNTRY\_NAME") =!= "Croatia").show(2)

#Getting distinct/unique rows

df.select("ORIGIN\_COUNTRY\_NAME", "DEST\_COUNTRY\_NAME").distinct().count()

df.select("ORIGIN\_COUNTRY\_NAME").distinct().count()

#To more explicitly specify sort direction, you need to use the asc and desc functions if operating on a column. These allow you to specify the order in which a given column should be sorted:

df.orderBy(expr("count desc")).show(2)

df.orderBy(desc("count"), asc("DEST\_COUNTRY\_NAME")).show(2)

**An advanced tip is to use asc\_nulls\_first, desc\_nulls\_first, asc\_nulls\_last, or desc\_nulls\_last to specify where you would like your null values to appear in an ordered DataFrame.**

#For optimization purposes, it’s sometimes advisable to sort within each partition before another set of transformations. You can use the sortWithinPartitions method to do this:

spark.read.json("/data/flight-data/json/\*-summary.json").sortWithinPartitions("count")

#Another important optimization opportunity is to partition the data according to some frequently filtered columns, which control the physical layout of data across the cluster including the partitioning scheme and the number of partitions. Repartition will incur a full shuffle of the data, regardless of whether one is necessary. **This means that you should typically only repartition when the future number of partitions is greater than your current number of partitions or when you are looking to partition by a set of columns**:

df.rdd.getNumPartitions #get number of repartitions

df.repartition(5)

#If you know that you’re going to be filtering by a certain column often, it can be worth repartitioning based on that column:

df.repartition(col("DEST\_COUNTRY\_NAME"))

#You can optionally specify the number of partitions you would like, too:

df.repartition(5, col("DEST\_COUNTRY\_NAME"))

#Coalesce, on the other hand, will not incur a full shuffle and will try to combine partitions. This operation will shuffle your data into five partitions based on the destination country name, and then coalesce them (without a full shuffle):

df.repartition(5, col("DEST\_COUNTRY\_NAME")).coalesce(2)

**/\*Functions on Booleans\*/**

#Filtering Example

Scala has some particular semantics regarding the use of == and ===. In Spark, if you want to filter by equality you should use === (equal) or =!= (not equal). You can also use the not function and the equalTo method.

Eg:

val df =

spark.read.format("csv").option("inferSchema","true").option("header","true").load("/user/chaitanyapolipalli/retail-data/by-day/2010-12-01.csv")

df.where(col("InvoiceNo").equalTo(536365)).select("InvoiceNo","Description").show(false)

or

df.where("InvoiceNo = 536365").show(false)

or

df.where

(col("InvoiceNo") === 536365).select("InvoiceNo", "Description").show(false)

#Complex filter example

val priceFilter = col("UnitPrice") > 600

val descripFilter = col("Description").contains("POSTAGE")

df.where(col("StockCode").isin("DOT")).where(priceFilter.or(descripFilter)).show()

-- in SQL

SELECT \* FROM dfTable WHERE StockCode in ("DOT") AND (UnitPrice > 600 OR instr(Description, "POSTAGE") >= 1)

#One “gotcha” that can come up is if you’re working with null data when creating Boolean expressions. If there is a null in your data, you’ll need to treat things a bit differently. Here’s how you can ensure that you perform a null-safe equivalence test:

df.where(col("Description").eqNullSafe("hello")).show()

**/\*Functions on Numbers\*/**

#numerical function 'pow' function example

val fabricatedQuantity = pow(col("Quantity") \* col("UnitPrice"),2) + 5

df.select(expr("CustomerId"), fabricatedQuantity.alias("realQuantity")).show(2)

#The round function rounds up if you’re exactly in between two numbers. You can round down by using the bround:

df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)

#Another common task is to compute summary statistics for a column or set of columns. We can use the describe method to achieve exactly this. This will take all numeric columns and calculate the count, mean, standard deviation, min, and max.

df.describe().show()

#As a last note, we can also add a unique ID to each row by using the function monotonically\_increasing\_id. This function generates a unique value for each row, starting with 0:

df.select(monotonically\_increasing\_id()).show(2)

**/\*Functions on Strings\*/**

#The initcap function will capitalize every word in a given string when that word is separated from another by a space

df.select(initcap(col("Description"))).show(2, false)

df.select(col("Description"),lower(col("Description")),upper(lower(col("Description")))).show(2)

#Another trivial task is adding or removing spaces around a string. You can do this by using lpad, ltrim, rpad and rtrim, trim:

#Note that if lpad or rpad takes a number less than the length of the string, it will always remove values from the right side of the string.

df.select(

ltrim(lit(" HELLO ")).as("ltrim"),

rtrim(lit(" HELLO ")).as("rtrim"),

trim(lit(" HELLO ")).as("trim"),

lpad(lit("HELLO"), 3, " ").as("lp"),

rpad(lit("HELLO"), 10, " ").as("rp")).show(2)

**/\*Regular Expressions\*/**

val simpleColors = Seq("black", "white", "red", "green", "blue")

val regexString = simpleColors.map(\_.toUpperCase).mkString("|")

// the | signifies `OR` in regular expression syntax

df.select(regexp\_replace(col("Description"),regexString, "COLOR").alias("color\_clean"),col("Description")).show(false)

#Another task might be to replace given characters with other characters. Building this as a regular expression could be tedious, so Spark also provides the translate function to replace these values. This is done at the character level and will replace all instances of a character with the indexed character in the replacement string:

df.select

(translate(col("Description"), "LEET", "1337"), col("Description")).show(2)

# “regexp\_extract” example

val regexString = simpleColors.map(\_.toUpperCase).mkString("(", "|", ")")

// the | signifies OR in regular expression syntax

df.select(regexp\_extract(col("Description"),regexString, 1).alias("color\_clean"),col("Description")).show(2)

# “contains” example

val containsBlack = col("Description").contains("BLACK")

val containsWhite = col("DESCRIPTION").contains("WHITE")

df.

withColumn("hasSimpleColor", containsBlack.or(containsWhite)).

where("hasSimpleColor").select("Description").show(3, false)

**/\*Dates Examples\*/**

val

dateDF = spark.range(10).withColumn("today", current\_date()).withColumn("now", current\_timestamp())dateDF.createOrReplaceTempView("dateTable")

#add 5 days and subtract five days

dateDF.select(date\_sub(col("today"), 5), date\_add(col("today"), 5)).show()

#difference between months and days

dateDF.

withColumn("week\_ago",date\_sub(col("today"), 7)).select(datediff(col("today"), col("week\_ago"))).show()

dateDF.select(to\_date(lit("2016-01-01")).alias("start"),to\_date(lit("2017-05-22")).alias("end")).select(months\_between(col("end"), col("start"))).show()

#The to\_date function allows you to convert a string to a date, optionally with a specified format. to\_date (optionally requires a format) and to\_timestamp (always requires a format) examples

# “to\_date” example

val dateFormat = "yyyy-dd-MM"

val

cleanDateDF = spark.range(1).select(to\_date(lit("2017-12-11"), dateFormat).

alias("date"),to\_date(lit("2017-20-12"), dateFormat).alias("date2"))

cleanDateDF.createOrReplaceTempView("dateTable2")

# “to\_timestamp” example

cleanDateDF.select(to\_timestamp(col("date"), dateFormat)).show()

# "drop" function. The simplest function is drop, which removes rows that contain nulls. The default is to drop any row in which any value is null:

#Specifying "any" as an argument drops a row if any of the values are null. Using “all” drops the row only if all values are null or NaN for that row:

df.na.drop("any")

df.na.drop("all")

#We can also apply this to certain sets of columns by passing in an array of columns:

df.na.drop("all", Seq("StockCode", "InvoiceNo"))

# "fill" example. Using the fill function, you can fill one or more columns with a set of values. This can be done by specifying a map—that is a particular value and a set of columns. For example, to fill all null values in columns of type String, you might specify the following:

df.na.fill("All Null values become this string")

#We could do the same for columns of type Integer by using df.na.fill(5:Integer), or for Doubles

df.na.fill(5:Double)

#To specify columns, we just pass in an array of column names like we did in the previous example:

df.na.fill(5, Seq("StockCode", "InvoiceNo"))

#We can also do this with with a Scala Map, where the key is the column name and the value is the value we would like to use to fill null values:

val fillColValues = Map("StockCode" -> 5, "Description" -> "No Value")

df.na.fill(fillColValues)

# "replace" example. In addition to replacing null values like we did with drop and fill, there are more flexible options that you can use with more than just null values. Probably the most common use case is to replace all values in a certain column according to their current value. The only requirement is that this value be the same type as the original value:

df.na.replace("Description", Map("" -> "UNKNOWN"))

**/\*Working with Complex Types\*/**

# "Structs" Example

#You can think of structs as DataFrames within DataFrames. A worked example will illustrate this more clearly. We can create a struct by wrapping a set of columns in parenthesis in a query:

df.selectExpr("(Description, InvoiceNo) as complex", "\*")

df.selectExpr("struct(Description, InvoiceNo) as complex", "\*")

val complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))

complexDF.createOrReplaceTempView("complexDF")

# We now have a DataFrame with a column complex. We can query it just as we might another DataFrame, the only difference is that we use a dot syntax to do so, or the column method getField:

complexDF.select("complex.Description")

complexDF.select(col("complex").getField("Description"))

#We can also query all values in the struct by using \*. This brings up all the columns to the top-level DataFrame:

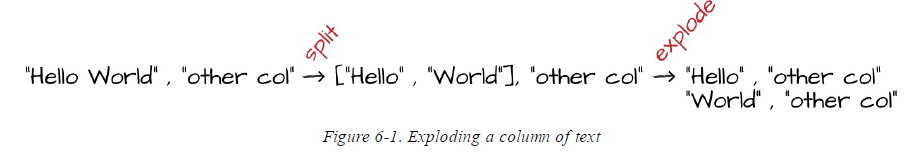
complexDF.select("complex.\*")

# "array" example. To define arrays, let’s work through a use case. With our current data, our objective is to take every single word in our Description column and convert that into a row in our DataFrame. The first task is to turn our Description column into a complex type, an array.

#We do this by using the split function and specify the delimiter:

df.select(split(col("Description"), " ")).show(2)

# "explode" example. The explode function takes a column that consists of arrays and creates one row (with the rest of the values duplicated) per value in the array.



df.withColumn("splitted", split(col("Description"), " ")).

withColumn("exploded", explode(col("splitted"))).

select("Description", "InvoiceNo", "exploded").show(2)

# "Map" example

df.select(map(col("Description"), col("InvoiceNo")).alias("complex\_map")).show(false)

#You can also explode map types, which will turn them into columns:

df.select(map(col("Description"), col("InvoiceNo")).alias("complex\_map")).selectExpr("explode(complex\_map)").show(false)

**/\*Working with JSON types\*/**

#You can use the get\_json\_object to inline query a JSON object, be it a dictionary or array. You can use json\_tuple if this object has only one level of nesting:

val jsonDF = spark.range(1).selectExpr("""'{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as jsonString""")

jsonDF.select(get\_json\_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]") as "column",json\_tuple(col("jsonString"), "myJSONKey")).show(2)

#You can also turn a StructType into a JSON string by using the to\_json function:

df.selectExpr

("(InvoiceNo, Description) as myStruct").select(to\_json(col("myStruct")))

#This function also accepts a dictionary (map) of parameters that are the same as the JSON data source. You can use the from\_json function to parse this (or other JSON data) back in. This naturally requires you to specify a schema, and optionally you can specify a map of options, as well:

val parseSchema = new

StructType(Array(new StructField("InvoiceNo",StringType,true),

new StructField("Description",StringType,true)))

df.selectExpr("(InvoiceNo, Description) as myStruct").

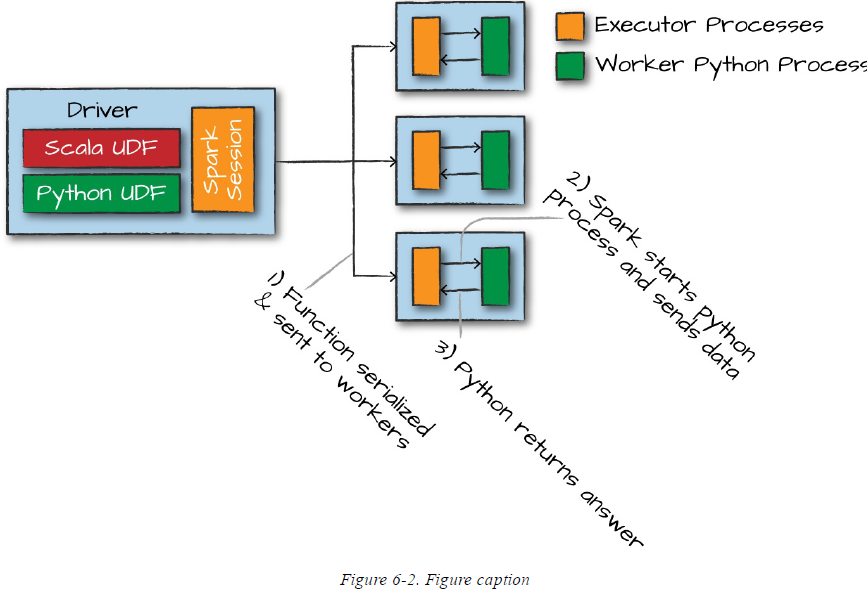
select(to\_json(col("myStruct")).alias("newJSON")).select(from\_json(col("newJSON"), parseSchema), col("newJSON")).show(2)

**/\*User Defined Functions\*/**

#UDFs can take and return one or more columns as input. Spark UDFs are incredibly powerful because you can write them in several different programming languages; you do not need to create them in an esoteric format or domain-specific language. They’re just functions that operate on the data, record by record. By default, these functions are registered as temporary functions to be used in that specific SparkSession or Context. Although you can write UDFs in Scala, Python, or Java, there are performance considerations that you should be aware of. To illustrate this, we’re going to walk through exactly what happens when you create UDF, pass that into Spark, and then execute code using that UDF.

#When you use the function, there are essentially two different things that occur. If the function is written in Scala or Java, you can use it within the Java Virtual Machine (JVM). This means that there will be little performance penalty aside from the fact that you can’t take advantage of code generation capabilities that Spark has for built-in functions. There can be performance issues if you create or use a lot of objects; we cover that in the section on optimization in Chapter 19. If the function is written in Python, something quite different happens. Spark starts a Python process on the worker, serializes all of the data to a format that Python can understand (remember, it was in the JVM earlier), executes the function row by row on that data in the Python process, and then finally returns the results of the row operations to the JVM and Spark.

#Starting this Python process is expensive, but the real cost is in serializing the data to Python. This is costly for two reasons: it is an expensive computation, but also, after the data enters Python, Spark cannot manage the memory of the worker. This means that you could potentially cause a worker to fail if it becomes resource constrained (because both the JVM and Python are competing for memory on the same machine). We recommend that you write your UDFs in Scala or Java—the small amount of time it should take you to write the function in Scala will always yield significant speed ups, and on top of that, you can still use the function from Python!



val udfExampleDF = spark.range(5).toDF("num")

def power3(number:Double):Double = number \* number \* number

val power3udf = udf(power3(\_:Double):Double)

udfExampleDF.select(power3udf(col("num"))).show()

#At this juncture, we can use this only as a DataFrame function. That is to say, we can’t use it within a string expression, only on an expression. However, we can also register this UDF as a Spark SQL function. This is valuable because it makes it simple to use this function within SQL as well as across languages. Let’s register the function in Scala:

spark.udf.register("power3", power3(\_:Double):Double)

udfExampleDF.selectExpr("power3(num)").show(2)

#Because this function is registered with Spark SQL—and we’ve learned that any Spark SQL function or expression is valid to use as an expression when working with DataFrames—we can turn around and use the UDF that we wrote in Scala, in Python. However, rather than using it as a DataFrame function, we use it as a SQL expression

#As a last note, you can also use UDF/UDAF creation via a Hive syntax. To allow for this, first you must enable Hive support when they create their SparkSession (via SparkSession.builder().enableHiveSupport()). Then you can register UDFs in SQL. This is only supported with precompiled Scala and Java packages, so you’ll need to specify them as a dependency:

-- in SQL

CREATE TEMPORARY FUNCTION myFunc AS 'com.organization.hive.udf.FunctionName'

#Additionally, you can register this as a permanent function in the Hive Metastore by removing TEMPORARY.

**/\*Aggregations\*/**

# "count" example

#There are a number of gotchas when it comes to null values and counting. For instance, when performing a count(\*), Spark will count null values (including rows containing all nulls). However, when counting an individual column, Spark will not count the null values.

val df = spark.read.format("csv").option("header", "true").

option("inferSchema", "true").load("/data/retail-data/all/\*.csv").coalesce(5)

df.cache()

df.createOrReplaceTempView("dfTable")

df.select(count("StockCode")).show()

# "countDistinct" example

#Sometimes, the total number is not relevant; rather, it’s the number of unique groups that you want. To get this number, you can use the countDistinct function. This is a bit more relevant for individual columns:

df.select(countDistinct("StockCode")).show()

# "approx\_count\_distinct" example

#Often, we find ourselves working with large datasets and the exact distinct count is irrelevant. There are times when an approximation to a certain degree of accuracy will work just fine, and for that, you can use the approx\_count\_distinct function:

#You will notice that approx\_count\_distinct took another parameter with which you can specify the maximum estimation error allowed. In this case, we specified a rather large error and thus receive an answer that is quite far off but does complete more quickly than countDistinct. You will see much greater performance gains with larger datasets.

df.select(approx\_count\_distinct("StockCode", 0.1)).show()

# "first" and "last" example

df.select(first("StockCode"), last("StockCode")).show()

# "min" and "max" example

df.select(min("Quantity"), max("Quantity")).show()

# "sum" example

df.select(sum("Quantity")).show()

# "sumDistinct" example. In addition to summing a total, you also can sum a distinct set of values by using the sumDistinct function:

df.select(sumDistinct("Quantity")).show()

**/\*Aggregating to Complex Types\*/**

#In Spark, you can perform aggregations not just of numerical values using formulas, you can also perform them on complex types. For example, we can collect a list of values present in a given column or only the unique values by collecting to a set. You can use this to carry out some more programmatic access later on in the pipeline or pass the entire collection in a user-defined function (UDF):

df.agg(collect\_set("Country"), collect\_list("Country")).show()

# "groupBy" example

df.groupBy("InvoiceNo", "CustomerId").count().show()

# Grouping with Expressions. As we saw earlier, counting is a bit of a special case because it exists as a method. For this, usually we prefer to use the count function. Rather than passing that function as an expression into a select statement, we specify it as within agg. This makes it possible for you to pass-in arbitrary expressions that just need to have some aggregation specified. You can even do things like alias a column after transforming it for later use in your data flow:

df.groupBy("InvoiceNo").agg(count("Quantity").alias("quan"),expr("count(Quantity)")).show()

# "cube", "rollup" and "groupBy" examples

https://stackoverflow.com/questions/37975227/what-is-the-difference-between-cube-rollup-and-groupby-operators

**/\* Types of Joins\*/**

#Inner joins (keep rows with keys that exist in the left and right datasets)

#Outer joins (keep rows with keys in either the left or right datasets)

#Left outer joins (keep rows with keys in the left dataset)

#Right outer joins (keep rows with keys in the right dataset)

#Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)

#Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)

#Natural joins (perform a join by implicitly matching the columns between the two datasets with the same names)

#Cross (or Cartesian) joins (match every row in the left dataset with every row in the right dataset)

#Examples:

val person = Seq((0, "Bill Chambers", 0, Seq(100)),(1, "Matei Zaharia", 1, Seq(500, 250, 100)),(2, "Michael Armbrust", 1, Seq(250, 100))).toDF("id", "name", "graduate\_program","spark\_status")

val graduateProgram = Seq((0, "Masters", "School of Information", "UC Berkeley"),(2, "Masters", "EECS", "UC Berkeley"),(1, "Ph.D.", "EECS", "UC Berkeley")).toDF("id", "degree","department", "school")

val sparkStatus = Seq((500, "Vice President"),(250, "PMC Member"),(100, "Contributor")).toDF("id", "status")

person.createOrReplaceTempView("person")

graduateProgram.createOrReplaceTempView("graduateProgram")

sparkStatus.createOrReplaceTempView("sparkStatus")

**/\*Inner Join\*/**

val joinExpression = person.col("graduate\_program") === graduateProgram.

col("id")

person.join(graduateProgram, joinExpression).show()

#We can also specify this explicitly by passing in a third parameter, the joinType:

var joinType = "inner"

person.join(graduateProgram, joinExpression, joinType).show()

**/\*Outer Join\*/**

var joinType = "outer"

person.join(graduateProgram,joinExpression,joinType).show(false)

**/\*Left Outer Joins\*/**

var joinType = "left\_outer"

graduateProgram.join(person,joinExpression,joinType).show(false)

**/\*Right Outer Joins\*/**

var joinType = "right\_outer"

person.join(graduateProgram,joinExpression,joinType).show(false)

**/\*Left Semi Joins\*/**

#Semi joins are a bit of a departure from the other joins. They do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. If the value does exist, those rows will be kept in the result, even if there are duplicate keys in the left DataFrame. Think of left semi joins as filters on a DataFrame, as opposed to the function of a conventional join:

joinType = "left\_semi"

person.join(graduateProgram,joinExpression,joinType).show(false)

#To prove it returns duplicates as well

val gradProgram2 = graduateProgram.union(Seq((0, "Masters", "Duplicated Row", "Duplicated School")).toDF())

gradProgram2.createOrReplaceTempView("gradProgram2")

gradProgram2.join(person, joinExpression, joinType).show()

**/\*Left Anti Joins\*/**

#Left anti joins are the opposite of left semi joins. Like left semi joins, they do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame. However, rather than keeping the values that exist in the second DataFrame, they keep only the values that do not have a corresponding key in the second DataFrame. Think of anti joins as a NOT IN SQL-style filter:

joinType = "left\_anti"

graduateProgram.join(person, joinExpression, joinType).show()

**/\*Cross Joins\*/**

joinType = "cross"

graduateProgram.join(person, joinExpression, joinType).show()

or

person.crossJoin(graduateProgram).show()

**/\*Joins on Complex Types\*/**

person.withColumnRenamed("id", "personId").

join(sparkStatus, expr("array\_contains(spark\_status, id)")).show()

**/\*Handling Duplicate Column Names\*/**

#One of the tricky things that come up in joins is dealing with duplicate column names in your results DataFrame. In a DataFrame, each column has a unique ID within Spark’s SQL Engine, Catalyst. This unique ID is purely internal and not something that you can directly reference. This makes it quite difficult to refer to a specific column when you have a DataFrame with duplicate column names. This can occur in two distinct situations:

#The join expression that you specify does not remove one key from one of the input DataFrames and the keys have the same column name

#Two columns on which you are not performing the join have the same name

#Let’s create a problem dataset that we can use to illustrate these problems:

val

gradProgramDupe = graduateProgram.withColumnRenamed("id", "graduate\_program")

val joinExpr =

gradProgramDupe.col("graduate\_program") === person.col("graduate\_program")

#Note that there are now two graduate\_program columns, even though we joined on that key:

person.join(gradProgramDupe, joinExpr).show()

#The challenge arises when we refer to one of these columns:

person.join(gradProgramDupe, joinExpr).select("graduate\_program").show()

#Given the previous code snippet, we will receive an error. In this particular example, Spark generates this message:

org.apache.spark.sql.AnalysisException: Reference 'graduate\_program' is ambiguous, could be: graduate\_program#40, graduate\_program#1079.;

#Approach 1: Different join expression

#When you have two keys that have the same name, probably the easiest fix is to change the join expression from a Boolean expression to a string or sequence. This automatically removes one of the columns for you during the join:

person.join(gradProgramDupe,"graduate\_program").select("graduate\_program").show()

#Approach 2: Dropping the column after the join

#Another approach is to drop the offending column after the join. When doing this, we need to refer to the column via the original source DataFrame. We can do this if the join uses the same key names or if the source DataFrames have columns that simply have the same name:

person.join(gradProgramDupe, joinExpr).drop(person.col("graduate\_program")).

select("graduate\_program").show()

val joinExpr = person.col("graduate\_program") === graduateProgram.col("id")

person.join(graduateProgram, joinExpr).drop(graduateProgram.col("id")).show()

#This is an artifact of Spark’s SQL analysis process in which an explicitly referenced column will pass analysis because Spark has no need to resolve the column. Notice how the column uses the .col method instead of a column function. That allows us to implicitly specify that column by its specific ID.

#Approach 3: Renaming a column before the join We can avoid this issue altogether if we rename one of our columns before the join:

val gradProgram3 = graduateProgram.withColumnRenamed("id", "grad\_id")

val joinExpr = person.col("graduate\_program") === gradProgram3.col("grad\_id")

person.join(gradProgram3, joinExpr).show()

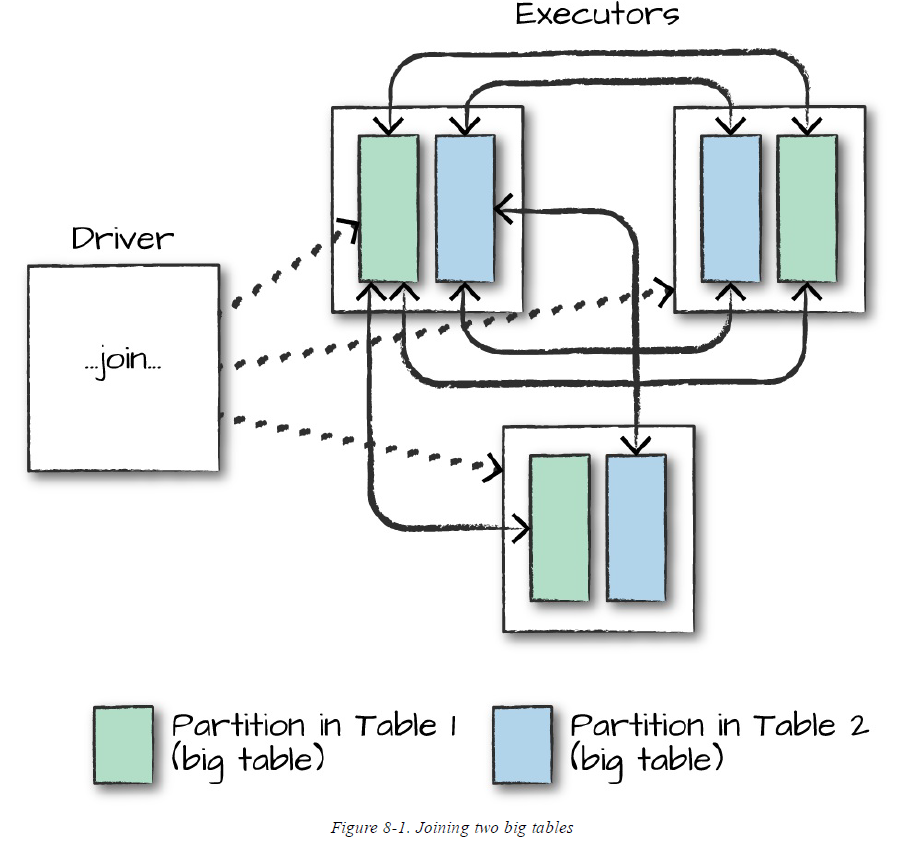
#Communication Strategies

#Spark approaches cluster communication in two different ways during joins. It either incurs a shuffle join, which results in an all-to-all communication or a broadcast join. Keep in mind that there is a lot more detail than we’re letting on at this point, and that’s intentional. Some of these internal optimizations are likely to change over time with new improvements to the cost-based optimizer and improved communication strategies. For this reason, we’re going to focus on the high-level examples to help you understand exactly what’s going on in some of the more common scenarios, and let you take advantage of some of the low-hanging fruit that you can use right away to try to speed up some of your workloads.

#The core foundation of our simplified view of joins is that in Spark you will have either a big table or a small table. Although this is obviously a spectrum (and things do happen differently if you have a “medium-sized table”), it can help to be binary about the distinction for the sake of this explanation.

**/\*Big table–to–big table \*/**

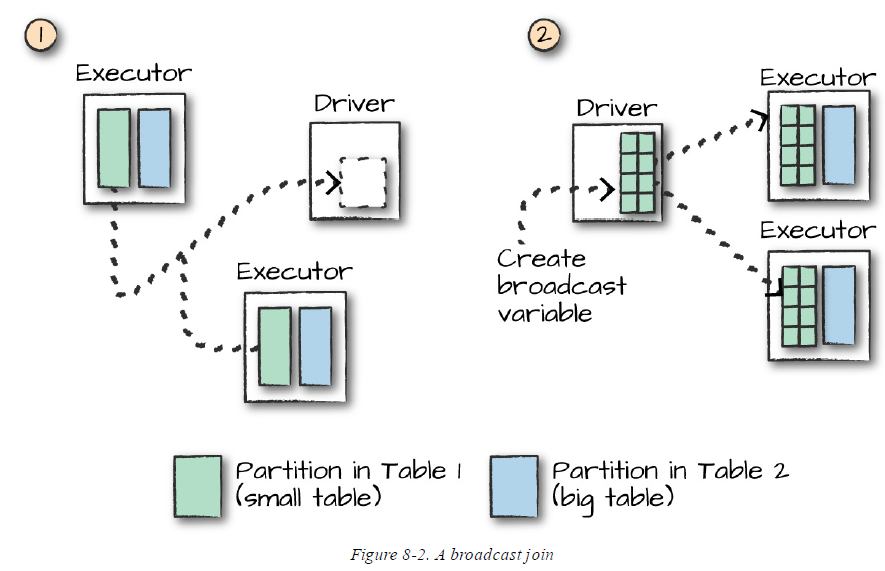
#When you join a big table to another big table, you end up with a shuffle join, such as that illustrates in Figure 8-1.



#In a shuffle join, every node talks to every other node and they share data according to which node has a certain key or set of keys (on which you are joining). These joins are expensive because the network can become congested with traffic, especially if your data is not partitioned well. This join describes taking a big table of data and joining it to another big table of data. An example of this might be a company that receives billions of messages every day from the Internet of Things, and needs to identify the day-over-day changes that have occurred. The way to do this is by joining on deviceId, messageType, and date in one column, and date - 1 day in the other column. In Figure 8-1, DataFrame 1 and DataFrame 2 are both large DataFrames. This means that all worker nodes (and potentially every partition) will need to communicate with one another during the entire join process (with no intelligent partitioning of data).

**/\*Big table–to–small table\*/**

#When the table is small enough to fit into the memory of a single worker node, with some breathing room of course, we can optimize our join. Although we can use a big table–to–big table communication strategy, it can often be more efficient to use a broadcast join. What this means is that we will replicate our small DataFrame onto every worker node in the cluster (be it located on one machine or many). Now this sounds expensive. However, what this does is prevent us from performing the all-to-all communication during the entire join process. Instead, we perform it only once at the beginning and then let each individual worker node perform the work without having to wait or communicate with any other worker node, as is depicted in Figure 8-2.



#At the beginning of this join will be a large communication, just like in the previous type of join. However, immediately after that first, there will be no further communication between nodes. This means that joins will be performed on every single node individually, making CPU the biggest bottleneck. For our current set of data, we can see that Spark has automatically set this up as a broadcast join by looking at the explain plan:

val joinExpr = person.col("graduate\_program") === graduateProgram.col("id")

person.join(graduateProgram, joinExpr).explain()

== Physical Plan ==

\*BroadcastHashJoin [graduate\_program#40], [id#5....

:- LocalTableScan [id#38, name#39, graduate\_progr...

+- BroadcastExchange HashedRelationBroadcastMode(....

+- LocalTableScan [id#56, degree#57, departmen....

#With the DataFrame API, we can also explicitly give the optimizer a hint that we would like to use a broadcast join by using the correct function around the small DataFrame in question. In this example, these result in the same plan we just saw; however, this is not always the case:

import org.apache.spark.sql.functions.broadcast

val joinExpr = person.col("graduate\_program") === graduateProgram.col("id")

person.join(broadcast(graduateProgram), joinExpr).explain()

#The SQL interface also includes the ability to provide hints to perform joins. These are not enforced, however, so the optimizer might choose to ignore them. You can set one of these hints by using a special comment syntax. MAPJOIN, BROADCAST, and BROADCASTJOIN all do the same thing and are all supported:

-- in SQL

SELECT /\*+ MAPJOIN(graduateProgram) \*/ \* FROM person JOIN graduateProgram

ON person.graduate\_program = graduateProgram.id

#This doesn’t come for free either: if you try to broadcast something too large, you can crash your driver node (because that collect is expensive). This is likely an area for optimization in the future.

**/\*Little table–to–little table\*/**

#When performing joins with small tables, it’s usually best to let Spark decide how to join them. You can always force a broadcast join if you’re noticing strange behavior.

#One thing we did not mention but is important to consider is if you partition your data correctly prior to a join, you can end up with much more efficient execution because even if a shuffle is planned, if data from two different DataFrames is already located on the same machine, Spark can avoid the shuffle. Experiment with some of your data and try partitioning beforehand to see if you can notice the increase in speed when performing those joins.

**/\*Data Sources\*/**

#Core structure for reading data is as follows:

DataFrameReader.format(...).option("key", "value").schema(...).load()

#We will use this format to read from all of our data sources. format is optional because by default Spark will use the Parquet format. option allows you to set key-value configurations to parameterize how you will read data. Lastly, schema is optional if the data source provides a schema or if you intend to use schema inference. Naturally, there are some required options for each format, which we will discuss when we look at each format

#The foundation for reading data in Spark is the DataFrameReader. We access this through the SparkSession via the read attribute:

spark.read

#After we have a DataFrame reader, we specify several values:

* The format
* The schema
* The read mode
* A series of options

#The format, options, and schema each return a DataFrameReader that can undergo further transformations and are all optional, except for one option. Each data source has a specific set of options that determine how the data is read into Spark (we cover these options shortly). At a minimum, you must supply the DataFrameReader a path to from which to read.

#Here’s an example of the overall layout:

spark.read.format("csv")

.option("mode", "FAILFAST")

.option("inferSchema", "true")

.option("path", "path/to/file(s)")

.schema(someSchema)

.load()

#There are a variety of ways in which you can set options; for example, you can build a map and pass in your configurations. The default is permissive.

Read mode Description

-------------- ------------

permissive Sets all fields to null when it encounters a corrupted record and places all corrupted records in a string column called \_corrupt\_record

dropMalformed Drops the row that contains malformed records

failFast Fails immediately upon encountering malformed records

#The core structure for writing data is as follows:

DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save()

#We will use this format to write to all of our data sources. format is optional because by default, Spark will use the arquet format. option, again, allows us to configure how to write out our given data. PartitionBy, bucketBy, and sortBy work only for file-based data sources; you can use them to control the specific layout of files at the destination. The default is errorIfExists

Save mode Description

-------------- ---------------

append Appends the output files to the list of files that already exist at that location

overwrite Will completely overwrite any data that already exists there

errorIfExists Throws an error and fails the write if data or files already exist at the specified location

ignore If data or files exist at the location, do nothing with the current DataFrame

**/\*Reading CSV File\*/**

import

org.apache.spark.sql.types.{StructField, StructType, StringType, LongType}

val myManualSchema = new StructType(Array(new StructField("DEST\_COUNTRY\_NAME", StringType, true),new StructField("ORIGIN\_COUNTRY\_NAME", StringType, true),new StructField("count", LongType, false)))

val csvFile = spark.read.format("csv").option("header", "true").option("mode", "FAILFAST").schema(myManualSchema).load("/data/flight-data/csv/2010-summary.csv")

**/\*Writing CSV Files\*/**

csvFile.write.format("csv").mode("overwrite").option("sep", "\t").save("/tmp/my-tsv-file.tsv")

#When you list the destination directory, you can see that my-tsv-file is actually a folder with numerous files within it: This actually reflects the number of partitions in our DataFrame at the time we write it out. If we were to repartition our data before then, we would end up with a different number of files.

**/\*JSON Files\*/**

#Those coming from the world of JavaScript are likely familiar with JavaScript Object Notation, or JSON, as it’s commonly called. There are some catches when working with this kind of data that are worth considering before we jump in. In Spark, when we refer to JSON files, we refer to linedelimited JSON files. This contrasts with files that have a large JSON object or array per file.

#The line-delimited versus multiline trade-off is controlled by a single option: multiLine. When you set this option to true, you can read an entire file as one json object and Spark will go through the work of parsing that into a DataFrame. Line-delimited JSON is actually a much more stable format because it allows you to append to a file with a new record (rather than having to read in an entire file and then write it out), which is what we recommend that you use. Another key reason for the popularity of line-delimited JSON is because JSON objects have structure, and JavaScript (on which JSON is based) has at least basic types. This makes it easier to work with because Spark can make more assumptions on our behalf about the data. You’ll notice that there are significantly less options than we saw for CSV because of the objects.

#Reading JSON file

spark.read.format("json").option("mode", "FAILFAST").schema(myManualSchema).load("/data/flight-data/json/2010-summary.json").show(5)

#Writing JSON File

csvFile.write.format("json").mode("overwrite").save("/tmp/my-json-file.json")

#Writing JSON files is just as simple as reading them, and, as you might expect, the data source does not matter. Therefore, we can reuse the CSV DataFrame that we created earlier to be the source for our JSON file. This, too, follows the rules that we specified before: one file per partition will be written out, and the entire DataFrame will be written out as a folder. It will also have one JSON object per line

**/\*Parquet Files\*/**

#Parquet is an open source column-oriented data store that provides a variety of storage optimizations, especially for analytics workloads. It provides columnar compression, which saves storage space and allows for reading individual columns instead of entire files. It is a file format that works exceptionally well with Apache Spark and is in fact the default file format. We recommend writing data out to Parquet for long-term storage because reading from a Parquet file will always be more efficient than JSON or CSV. Another advantage of Parquet is that it supports complex types. This means that if your column is an array (which would fail with a CSV file, for example), map, or struct, you’ll still be able to read and write that file without issue.

#Reading Parquet Files

#Parquet has very few options because it enforces its own schema when storing data. Thus, all you need to set is the format and you are good to go. We can set the schema if we have strict requirements for what our DataFrame should look like. Oftentimes this is not necessary because we can use schema on read, which is similar to the inferSchema with CSV files. However, with Parquet files, this method is more powerful because the schema is built into the file itself (so no inference needed).

#Parquet Read Example

spark.read.format("parquet").load("/data/flight-data/parquet/2010-summary.parquet").show(5)

#Parquet Write Example

csvFile.write.format("parquet").mode("overwrite").save("/tmp/my-parquet-file.parquet")

**/\*ORC Files\*/**

#ORC is a self-describing, type-aware columnar file format designed for Hadoop workloads. It is optimized for large streaming reads, but with integrated support for finding required rows quickly. ORC actually has no options for reading in data because Spark understands the file format quite well. An often-asked question is: What is the difference between ORC and Parquet? For the most part, they’re quite similar; the fundamental difference is that Parquet is further optimized for use with Spark, whereas ORC is further optimized for Hive.

#ORC Read Example

spark.read.format("orc").load("/data/flight-data/orc/2010-summary.orc").show(5)

#ORC Write Example

csvFile.write.format("orc").mode("overwrite").save("/tmp/my-json-file.orc")

**/\*Reading from SQL Databases\*/**

#There are a number of options that are available when reading from and writing to SQL databases.



#Connecting to MySql Database in itversity

spark-shell --jars /usr/share/java/mysql-connector-java.jar

val dbDataFrame =

spark.read.format("jdbc").option("driver","com.mysql.jdbc.Driver")

.option("url","jdbc:mysql://ms.itversity.com:3306/retail\_export")

.option("dbtable", "chaitanyapolipalli\_flight\_data")

.option("user","retail\_user").option("password","itversity").load()

val driver = "com.mysql.jdbc.Driver"

val url = "jdbc:mysql://ms.itversity.com:3306/retail\_export"

val tablename = "chaitanyapolipalli\_flight\_data"

val user = "retail\_user"

val password = "itversity"

#As we create this DataFrame, it is no different from any other: you can query it, transform it, and join it without issue. You’ll also notice that there is already a schema, as well. That’s because Spark gathers this information from the table itself and maps the types to Spark data types

dbDataFrame.select("DEST\_COUNTRY\_NAME").distinct.show

#Query Pushdown:

#Spark makes a best-effort attempt to filter data in the database itself before creating the DataFrame. For example, in the previous sample query, we can see from the query plan that it selects only the relevant column name from the table:

dbDataFrame.select("DEST\_COUNTRY\_NAME").distinct.explain

== Physical Plan ==

\*(2) HashAggregate(keys=[DEST\_COUNTRY\_NAME#46], functions=[])

+- Exchange hashpartitioning(DEST\_COUNTRY\_NAME#46, 200)

+- \*(1) HashAggregate(keys=[DEST\_COUNTRY\_NAME#46], functions=[])

+- \*(1) Scan JDBCRelation(chaitanyapolipalli\_flight\_data) [numPartitions=1] [DEST\_COUNTRY\_NAME#46] PushedFilters: [], ReadSchema: struct<DEST\_COUNTRY\_NAME:string>

#Spark can actually do better than this on certain queries. For example, if we specify a filter on our DataFrame, Spark will push that filter down into the database. We can see this in the explain plan under PushedFilters.

dbDataFrame.filter("DEST\_COUNTRY\_NAME in ('Anguilla', 'Sweden')").explain

#Spark can’t translate all of its own functions into the functions available in the SQL database in which you’re working. Therefore, sometimes you’re going to want to pass an entire query into your SQL that will return the results as a DataFrame. Now, this might seem like it’s a bit complicated, but it’s actually quite straightforward. Rather than specifying a table name, you just specify a SQL query. Of course, you do need to specify this in a special way; you must wrap the query in parenthesis and rename it to something—in this case, I just gave it the same table name

val pushdownQuery =

"""(SELECT DISTINCT(DEST\_COUNTRY\_NAME) FROM chaitanyapolipalli\_flight\_data)AS flight\_info"""

val dbDataFrame = spark.read.format("jdbc")

.option("url",url).option("dbtable",pushdownQuery)

.option("driver",driver).option("user",user).option("password",password).load()

#Reading from databases in parallel:

#All throughout this book, we have talked about partitioning and its importance in data processing. Spark has an underlying algorithm that can read multiple files into one partition, or conversely, read multiple partitions out of one file, depending on the file size and the “splitability” of the file type and compression. The same flexibility that exists with files, also exists with SQL databases except that you must configure it a bit more manually. What you can configure, as seen in the previous options, is the ability to specify a maximum number of partitions to allow you to limit how much you are reading and writing in parallel:

#Partitioning a file while reading from DB

val dbDataFrame = spark.read.format("jdbc").option("url", url)

.option("dbtable", tablename)

.option("driver",driver).option("numPartitions",10).option("user",user)

.option("password",password).load()

#There are several other optimizations that unfortunately only seem to be under another API set. You can explicitly push predicates down into SQL databases through the connection itself. This optimization allows you to control the physical location of certain data in certain partitions by specifying predicates. That’s a mouthful, so let’s look at a simple example. We only need data from two countries in our data: Anguilla and Sweden. We could filter these down and have them pushed into the database, but we can also go further by having them arrive in their own partitions in Spark. We do that by specifying a list of predicates when we create the data source:

#Specifying predicates while reading from DB

val props = new java.util.Properties

props.setProperty("driver", driver)

props.setProperty("user", user)

props.setProperty("password", password)

val predicates = Array("DEST\_COUNTRY\_NAME = 'Sweden' OR ORIGIN\_COUNTRY\_NAME = 'Sweden'","DEST\_COUNTRY\_NAME = 'Anguilla' OR ORIGIN\_COUNTRY\_NAME = 'Anguilla'")

spark.read.jdbc(url, tablename, predicates, props).show()

spark.read.jdbc(url, tablename, predicates, props).rdd.getNumPartitions

#Partitioning based on a sliding window:

Let’s take a look to see how we can partition based on predicates. In this example, we’ll partition based on our numerical count column. Here, we specify a minimum and a maximum for both the first partition and last partition. Anything outside of these bounds will be in the first partition or final partition. Then, we set the number of partitions we would like total (this is the level of parallelism). Spark then queries our database in parallel and returns numPartitions partitions. We simply modify the upper and lower bounds in order to place certain values in certain partitions. No filtering is taking place like we saw in the previous example:

val colName = "count"

val lowerBound = 0L

val upperBound = 348113L // this is the max count in our database

val numPartitions = 10

spark.read.jdbc(url,tablename,colName,lowerBound,upperBound,numPartitions,props).count()

**/\*Writing to SQL Databases\*/**

flightData.write.format("jdbc")

.option("url","jdbc:mysql://ms.itversity.com:3306/retail\_export")

.option("driver","com.mysql.jdbc.Driver")

.option("dbtable","chaitanyapolipalli\_flight\_data")

.option("user","retail\_user")

.option("password","itversity").mode("append").save()

**/\*Text Files\*/**

#Reading text files is straightforward: you simply specify the type to be textFile. With textFile, partitioned directory names are ignored. To read and write text files according to partitions, you should use text, which respects partitioning on reading and writing

#Reading

spark.read.textFile("/user/chaitanyapolipalli/flight-data/csv/2010-summary.csv").selectExpr("split(value, ',') as rows").show(false)

#Writing

val csvFile = spark.read.format("csv").option("header", "true").option("mode", "FAILFAST").schema(myManualSchema).load("/data/flight-data/csv/2010-summary.csv")

csvFile.select("DEST\_COUNTRY\_NAME").write.text("/user/chaitanyapolipalli/tmp/simple-text-file.txt")

#If you perform some partitioning when performing your write (we’ll discuss partitioning in the next couple of pages), you can write more columns. However, those columns will manifest as directories in the folder to which you’re writing out to, instead of columns on every single file:

csvFile.limit(10).select("DEST\_COUNTRY\_NAME","count")

.write.partitionBy("count")

.text("/user/chaitanyapolipalli/tmp/five-csv-files2.csv")

**/\*Advanced I/O Concepts\*/**

#we can control the parallelism of files that we write by controlling the partitions prior to writing. We can also control specific data layout by controlling two things:

#Bucketing and Partitioning

#Splittable File Types and Compression

#Certain file formats are fundamentally “splittable.” This can improve speed because it makes it possible for Spark to avoid reading an entire file, and access only the parts of the file necessary to satisfy your query. Additionally if you’re using something like Hadoop Distributed File System (HDFS), splitting a file can provide further optimization if that file spans multiple blocks. In conjunction with this is a need to manage compression. Not all compression schemes are splittable. How you store your data is of immense consequence when it comes to making your Spark jobs run smoothly.

#We recommend Parquet with gzip compression.

#Reading Data in Parallel

#Multiple executors cannot read from the same file at the same time necessarily, but they can read different files at the same time. In general, this means that when you read from a folder with multiple files in it, each one of those files will become a partition in your DataFrame and be read in by available executors in parallel (with the remaining queueing up behind the others).

#Writing Data in Parallel

#The number of files or data written is dependent on the number of partitions the DataFrame has at the time you write out the data. By default, one file is written per partition of the data. This means that although we specify a “file,” it’s actually a number of files within a folder, with the name of the specified file, with one file per each partition that is written.

#For example, the following code

csvFile.repartition(5).write.format("csv").save("/tmp/multiple.csv")

#will end up with five files inside of that folder.

#Partitioning

#Partitioning is a tool that allows you to control what data is stored (and where) as you write it. When you write a file to a partitioned directory (or table), you basically encode a column as a folder. What this allows you to do is skip lots of data when you go to read it in later, allowing you to read in only the data relevant to your problem instead of having to scan the complete dataset. These are supported for all file-based data sources:

csvFile.limit(10).write.mode("overwrite").partitionBy("DEST\_COUNTRY\_NAME").save("/tmp/partitioned-files.parquet")

#This is probably the lowest-hanging optimization that you can use when you have a table that readers frequently filter by before manipulating. For instance, date is particularly common for a partition because, downstream, often we want to look at only the previous week’s data (instead of scanning the entire list of records). This can provide massive speedups for readers.

#Bucketing

#Bucketing is another file organization approach with which you can control the data that is specifically written to each file. This can help avoid shuffles later when you go to read the data because data with the same bucket ID will all be grouped together into one physical partition. This means that the data is prepartitioned according to how you expect to use that data later on, meaning you can avoid expensive shuffles when joining or aggregating. Rather than partitioning on a specific column (which might write out a ton of directories), it’s probably worthwhile to explore bucketing the data instead. This will create a certain number of files and organize our data into those “buckets”:

val numberBuckets = 10

val columnToBucketBy = "count"

csvFile.write.format("parquet").mode("overwrite").bucketBy(numberBuckets, columnToBucketBy).saveAsTable("bucketedFiles")

#Bucketing is supported only for Spark-managed tables

**/\*Managing File Size\*/**

#Managing file sizes is an important factor not so much for writing data but reading it later on. When you’re writing lots of small files, there’s a significant metadata overhead that you incur managing all of those files. Spark especially does not do well with small files, although many file systems (like HDFS) don’t handle lots of small files well, either. You might hear this referred to as the “small file problem.” The opposite is also true: you don’t want files that are too large either, because it becomes inefficient to have to read entire blocks of data when you need only a few rows. Spark 2.2 introduced a new method for controlling file sizes in a more automatic way. We saw previously that the number of output files is a derivative of the number of partitions we had at write time (and the partitioning columns we selected). Now, you can take advantage of another tool in order to limit output file sizes so that you can target an optimum file size. You can use the maxRecordsPerFile option and specify a number of your choosing. This allows you to better control file sizes by controlling the number of records that are written to each file. For example, if you set an option for a writer as

df.write.option("maxRecordsPerFile", 5000)

#Spark will ensure that files will contain at most 5,000 records.

**/\*Spark SQL\*/**

#Spark SQL is intended to operate as an online analytic processing (OLAP) database, not an online transaction processing (OLTP) database. This means that it is not intended to perform extremely low-latency queries. Even though support for inplace modifications is sure to be something that comes up in the future, it’s not something that is currently available.

**/\*Spark’s Relationship to Hive\*/**

#Spark SQL has a great relationship with Hive because it can connect to Hive metastores. The Hive metastore is the way in which Hive maintains table information for use across sessions. With Spark SQL, you can connect to your Hive metastore (if you already have one) and access table metadata to reduce file listing when accessing information. This is popular for users who are migrating from a legacy Hadoop environment and beginning to run all their workloads using Spark.

**/\*The Hive metastore\*/**

#To connect to the Hive metastore, there are several properties that you’ll need. First, you need to set the Metastore version (spark.sql.hive.metastore.version) to correspond to the proper Hive metastore that you’re accessing. By default, this value is 1.2.1. You also need to set spark.sql.hive.metastore.jars if you’re going to change the way that the HiveMetastoreClient is initialized. Spark uses the default versions, but you can also specify Maven repositories or a classpath in the standard format for the Java Virtual Machine JVM). In addition, you might need to supply proper class prefixes in order to communicate with different databases that store the Hive metastore. You’ll set these as shared prefixes that both Spark and Hive will share (spark.sql.hive.metastore.sharedPrefixes).

#Tables

#To do anything useful with Spark SQL, you first need to define tables. Tables are logically equivalent to a DataFrame in that they are a structure of data against which you run commands. We can join tables, filter them, aggregate them, and perform different manipulations that we saw in previous chapters. The core difference between tables and DataFrames is this: you define DataFrames in the scope of a programming language, whereas you define tables within a database. This means that when you create a table (assuming you never changed the database), it will belong to the default database. We discuss databases more fully later on in the chapter. An important thing to note is that in Spark 2.X, tables always contain data. There is no notion of a temporary table, only a view, which does not contain data. This is important because if you go to drop a table, you can risk losing the data when doing so

**/\*Spark-Managed Tables\*/**

#One important note is the concept of managed versus unmanaged tables. Tables store two important pieces of information. The data within the tables as well as the data about the tables; that is, the metadata. You can have Spark manage the metadata for a set of files as well as for the data. When you define a table from files on disk, you are defining an unmanaged table. When you use saveAsTable on a DataFrame, you are creating a managed table for which Spark will track of all of the relevant information. This will read your table and write it out to a new location in Spark format. You can see this reflected in the new explain plan. In the explain plan, you will also notice that this writes to the default Hive warehouse location. You can set this by setting the spark.sql.warehouse.dir configuration to the directory of your choosing when you create your SparkSession. By default Spark sets this to /user/hive/warehouse: Note in the results that a database is listed. Spark also has databases which we will discuss later in this chapter, but for now you should keep in mind that you can also see tables in a specific database by using the query show tables IN databaseName, where databaseName represents the name of the database that you want to query. If you are running on a new cluster or local mode, this should return zero results.

**/\*Creating Tables\*/**

#You can create tables from a variety of sources. Something fairly unique to Spark is the capability of reusing the entire Data Source API within SQL. This means that you do not need to define a table and then load data into it; Spark lets you create one on the fly. You can even specify all sorts of sophisticated options when you read in a file. For example, here’s a simple way to read in the flight data we worked with in previous chapters:

CREATE TABLE flights (DEST\_COUNTRY\_NAME STRING, ORIGIN\_COUNTRY\_NAME STRING, count LONG) USING JSON OPTIONS (path '/data/flight-data/json/2015-summary.json')

#'USING' AND 'STORED AS'

#The specification of the USING syntax in the previous example is of significant importance. If you do not specify the format, Spark will default to a Hive SerDe configuration. This has performance implications for future readers and writers because Hive SerDes are much slower than Spark’s native serialization. Hive users can also use the STORED AS syntax to specify that this should be a Hive table.

#You can also add comments to certain columns in a table, which can help other developers understand the data in the tables:

CREATE TABLE IF NOT EXISTS flights\_csv

(DEST\_COUNTRY\_NAME STRING,ORIGIN\_COUNTRY\_NAME STRING COMMENT "remember, the US will be most prevalent",count LONG)

USING csv OPTIONS (header true, path'/data/flight-data/csv/2015-summary.csv')

#Refreshing Table Metadata

#Maintaining table metadata is an important task to ensure that you’re reading from the most recent set of data. There are two commands to refresh table metadata. REFRESH TABLE refreshes all cached entries (essentially, files) associated with the table. If the table were previously cached, it would be cached lazily the next time it is scanned:

REFRESH table partitioned\_flights

#Another related command is REPAIR TABLE, which refreshes the partitions maintained in the catalog for that given table. This command’s focus is on collecting new partition information—an example might be writing out a new partition manually and the need to repair the table accordingly:

MSCK REPAIR TABLE partitioned\_flights

#Dropping a table deletes the data in the table, so you need to be very careful when doing this.

DROP TABLE IF EXISTS flights\_csv;

#Dropping unmanaged tables

#If you are dropping an unmanaged table (e.g., hive\_flights), no data will be removed but you will no longer be able to refer to this data by the table name.

**/\*Caching Tables\*/**

#Just like DataFrames, you can cache and uncache tables. You simply specify which table you would like using the following syntax:

CACHE TABLE flights

#Here’s how you uncache them:

UNCACHE TABLE FLIGHTS

**/\*Views\*/**

#Creating Views

#To an end user, views are displayed as tables, except rather than rewriting all of the data to a new location, they simply perform a transformation on the source data at query time. This might be a filter, select, or potentially an even larger GROUP BY or ROLLUP. For instance, in the following example, we create a view in which the destination is United States in order to see only those flights:

CREATE VIEW just\_usa\_view AS SELECT \* FROM flights WHERE dest\_country\_name = 'United States'

#Like tables, you can create temporary views that are available only during the current session and are not registered to a database:

CREATE TEMP VIEW just\_usa\_view\_temp AS SELECT \* FROM flights WHERE dest\_country\_name = 'United States'

#Or, it can be a global temp view. Global temp views are resolved regardless of database and are viewable across the entire Spark application, but they are removed at the end of the session:

CREATE GLOBAL TEMP VIEW just\_usa\_global\_view\_temp AS SELECT \* FROM flights WHERE dest\_country\_name = 'United States

#You can also specify that you would like to overwite a view if one already exists by using the keywords shown in the sample that follows. We can overwrite both temp views and regular views:

CREATE OR REPLACE TEMP VIEW just\_usa\_view\_temp AS SELECT \* FROM flights WHERE dest\_country\_name = 'United States'

#Dropping Views

DROP VIEW IF EXISTS just\_usa\_view;

**/\*Databases\*/**

#Databases are a tool for organizing tables. As mentioned earlier, if you do not define one, Spark will use the default database. Any SQL statements that you run from within Spark (including DataFrame commands) execute within the context of a database. This means that if you change the database, any user-defined tables will remain in the previous database and will need to be queried differently.

#Creating Databases

CREATE DATABASE some\_db

#Setting the Database

USE some\_db

#However, you can query different databases by using the correct prefix:

SELECT \* FROM default.flights

#You can see what database you’re currently using by running the following command:

SELECT current\_database()

#Dropping Databases

DROP DATABASE IF EXISTS some\_db;

**/\*case…when…then Statements\*/**

#Oftentimes, you might need to conditionally replace values in your SQL queries. You can do this by using a case...when...then...end style statement. This is essentially the equivalent of programmatic if statements:

SELECT

CASE WHEN DEST\_COUNTRY\_NAME = 'UNITED STATES' THEN 1

WHEN DEST\_COUNTRY\_NAME = 'Egypt' THEN 0

ELSE -1

END

FROM partitioned\_flights

**/\*Complex Types\*/**

#Complex types are a departure from standard SQL and are an incredibly powerful feature that does not exist in standard SQL. Understanding how to manipulate them appropriately in SQL is essential. There are three core complex types in Spark SQL: structs, lists, and maps.

#Structs

#Structs are more akin to maps. They provide a way of creating or querying nested data in Spark. To create one, you simply need to wrap a set of columns (or expressions) in parentheses:

CREATE VIEW IF NOT EXISTS nested\_data AS SELECT (DEST\_COUNTRY\_NAME, ORIGIN\_COUNTRY\_NAME) as country, count FROM flights

#Now, you can query this data to see what it looks like:

SELECT \* FROM nested\_data

#You can even query individual columns within a struct—all you need to do is use dot syntax:

SELECT country.DEST\_COUNTRY\_NAME, count FROM nested\_data

#If you like, you can also select all the subvalues from a struct by using the struct’s name and select all of the subcolumns. Although these aren’t truly subcolumns, it does provide a simpler way to think about them because we can do everything that we like with them as if they were a column:

SELECT country.\*, count FROM nested\_data

**/\*Functions\*/**

#In addition to complex types, Spark SQL provides a variety of sophisticated functions. You can find most of these functions in the DataFrames function reference; however, it is worth understanding how to find these functions in SQL, as well. To see a list of functions in Spark SQL, you use the SHOW FUNCTIONS statement:

SHOW FUNCTIONS

#You can also more specifically indicate whether you would like to see the system functions (i.e., those built into Spark) as well as user functions:

SHOW SYSTEM FUNCTIONS

#User functions are those defined by you or someone else sharing your Spark environment. These are the same user-defined functions that we talked about in earlier chapters (we will discuss how to create them later on in this chapter):

SHOW USER FUNCTIONS

#You can filter all SHOW commands by passing a string with wildcard (\*) characters. Here, we can see all functions that begin with “s”:

SHOW FUNCTIONS "s\*";

#Optionally, you can include the LIKE keyword, although this is not necessary:

SHOW FUNCTIONS LIKE "collect\*";

#Even though listing functions is certainly useful, often you might want to know more about specific functions themselves. To do this, use the DESCRIBE keyword, which returns the documentation for a specific function.

**/\*Subqueries\*/**

#With subqueries, you can specify queries within other queries. This makes it possible for you to specify some sophisticated logic within your SQL. In Spark, there are two fundamental subqueries. Correlated subqueries use some information from the outer scope of the query in order to supplement information in the subquery. Uncorrelated subqueries include no information from the outer scope. Each of these queries can return one (scalar subquery) or more values. Spark also includes support for predicate subqueries, which allow for filtering based on values.

#Uncorrelated predicate subqueries

#For example, let’s take a look at a predicate subquery. In this example, this is composed of two uncorrelated queries. The first query is just to get the top five country destinations based on the data we have:

SELECT dest\_country\_name FROM flights GROUP BY dest\_country\_name ORDER BY sum(count) DESC LIMIT 5

#This gives us the following result:

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

|dest\_country\_name|

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

| United States|

| Canada|

| Mexico|

| United Kingdom|

| Japan|

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

#Now we place this subquery inside of the filter and check to see if our origin country exists in that list:

SELECT \* FROM flights WHERE origin\_country\_name IN (SELECT dest\_country\_name FROM flights GROUP BY dest\_country\_name ORDER BY sum(count) DESC LIMIT 5)

#This query is uncorrelated because it does not include any information from the outer scope of the query. It’s a query that you can run on its own.

#Correlated predicate subqueries

#Correlated predicate subqueries allow you to use information from the outer scope in your inner query. For example, if you want to see whether you have a flight that will take you back from your destination country, you could do so by checking whether there is a flight that has the destination country as an origin and a flight that had the origin country as a destination:

SELECT \* FROM flights f1 WHERE EXISTS (SELECT 1 FROM flights f2 WHERE f1.dest\_country\_name = f2.origin\_country\_name) AND EXISTS (SELECT 1 FROM flights f2 WHERE f2.dest\_country\_name = f1.origin\_country\_name)

#EXISTS just checks for some existence in the subquery and returns true if there is a value. You can flip this by placing the NOT operator in front of it. This would be equivalent to finding a flight to a destination from which you won’t be able to return!

#Uncorrelated scalar queries

#Using uncorrelated scalar queries, you can bring in some supplemental information that you might not have previously. For example, if you wanted to include the maximum value as its own column from the entire counts dataset, you could do this:

SELECT \*, (SELECT max(count) FROM flights) AS maximum FROM flights

#Setting Configuration Values in SQL

#We talk about configurations in Chapter 15, but as a preview, it’s worth mentioning how to set configurations from SQL. Naturally, you can only set Spark SQL configurations that way, but here’s how you can set shuffle partitions:

SET spark.sql.shuffle.partitions=20

**/\*DataSets\*/**

#When to Use Datasets

#You might ponder, if I am going to pay a performance penalty when I use Datasets, why should I use them at all? If we had to condense this down into a canonical list, here are a couple of reasons:

* When the operation(s) you would like to perform cannot be expressed using DataFrame manipulations
* When you want or need type-safety, and you’re willing to accept the cost of performance to achieve it

#Let’s explore these in more detail. There are some operations that cannot be expressed using the Structured APIs we have seen in the previous chapters. Although these are not particularly common, you might have a large set of business logic that you’d like to encode in one specific function instead of in SQL or DataFrames. This is an appropriate use for Datasets. Additionally, the Dataset API is type-safe. Operations that are not valid for their types, say subtracting two string types, will fail at compilation time not at runtime. If correctness and bulletproof code is your highest priority, at the cost of some performance, this can be a great choice for you. This does not protect you from malformed data but can allow you to more elegantly handle and organize it.

**/\*Creating Datasets\*/**

#Creating Datasets is somewhat of a manual operation, requiring you to know and define the schemas ahead of time.

#In Scala: Case Classes

#To create Datasets in Scala, you define a Scala case class. A case class is a regular class that has the following characteristics:

* Immutable
* Decomposable through pattern matching
* Allows for comparison based on structure instead of reference
* Easy to use and manipulate

#These traits make it rather valuable for data analysis because it is quite easy to reason about a case class. Probably the most important feature is that case classes are immutable and allow for comparison by structure instead of value.

#To begin creating a Dataset, let’s define a case class for one of our datasets:

case class Flight(DEST\_COUNTRY\_NAME: String, ORIGIN\_COUNTRY\_NAME: String, count: BigInt)

#Now that we defined a case class, this will represent a single record in our dataset. More succintly, we now have a Dataset of Flights. This doesn’t define any methods for us, simply the schema. When we read in our data, we’ll get a DataFrame. However, we simply use the as method to cast it to our specified row type:

val flightsDF = spark.read.

parquet("/user/chaitanyapolipalli/flight-data/parquet/2010-summary.parquet/")

val flights = flightsDF.as[Flight]

**/\*Actions\*/**

#Even though we can see the power of Datasets, what’s important to understand is that actions like collect, take, and count apply to whether we are using Datasets or DataFrames:

flights.show(2)

#You’ll also notice that when we actually go to access one of the case classes, we don’t need to do any type coercion, we simply specify the named attribute of the case class and get back, not just the expected value but the expected type, as well:

flights.first.DEST\_COUNTRY\_NAME

**/\*Transformations\*/**

#Filtering

#Let’s look at a simple example by creating a simple function that accepts a Flight and returns a Boolean value that describes whether the origin and destination are the same. This is not a UDF (at least, in the way that Spark SQL defines UDF) but a generic function.

#You’ll notice in the following example that we’re going to create a function to define this filter. This is an important difference from what we have done thus far in the book. By specifying a function, we are forcing Spark to evaluate this function on every row in our Dataset. This can be very resource intensive. For simple filters it is always preferred to write SQL expressions. This will greatly reduce the cost of filtering out the data while still allowing you to manipulate it as a Dataset later on:

def originIsDestination(flight\_row: Flight): Boolean = {

return flight\_row.ORIGIN\_COUNTRY\_NAME == flight\_row.DEST\_COUNTRY\_NAME

}

#We can now pass this function into the filter method specifying that for each row it should verify that this function returns true and in the process will filter our Dataset down accordingly:

flights.filter(flight\_row => originIsDestination(flight\_row)).first()

#The result is:

Flight = Flight(United States,United States,348113)

#As we saw earlier, this function does not need to execute in Spark code at all. Similar to our UDFs, we can use it and test it on data on our local machines before using it within Spark. For example, this dataset is small enough for us to collect to the driver (as an Array of Flights) on which we can operate and perform the exact same filtering operation:

flights.collect().filter(flight\_row => originIsDestination(flight\_row))

#The result is:

Array[Flight] = Array(Flight(United States,United States,348113))

#We can see that we get the exact same answer as before.

**/\*Mapping\*/**

#Filtering is a simple transformation, but sometimes you need to map one value to another value. We did this with our function in the previous example: it accepts a flight and returns a Boolean, but other times we might actually need to perform something more sophisticated like extract a value, compare a set of values, or something similar. The simplest example is manipulating our Dataset such that we extract one value from each row. This is effectively performing a DataFrame like select on our Dataset. Let’s extract the destination:

val destinations = flights.map(f => f.DEST\_COUNTRY\_NAME)

#Notice that we end up with a Dataset of type String. That is because Spark already knows the JVM type that this result should return and allows us to benefit from compile-time checking if, for some reason, it is invalid. We can collect this and get back an array of strings on the driver:

val localDestinations = destinations.take(5)

#This might feel trivial and unnecessary; we can do the majority of this right on DataFrames. We in fact recommend that you do this because you gain so many benefits from doing so. You will gain advantages like code generation that are simply not possible with arbitrary user-defined functions. However, this can come in handy with much more sophisticated row-by-row manipulation.

**/\*Joins\*/**

#Joins, as we covered earlier, apply just the same as they did for DataFrames. However Datasets also provide a more sophisticated method, the joinWith method. joinWith is roughly equal to a co-group (in RDD terminology) and you basically end up with two nested Datasets inside of one. Each column represents one Dataset and these can be manipulated accordingly. This can be useful when you need to maintain more information in the join or perform some more sophisticated manipulation on the entire result, like an advanced map or filter. Let’s create a fake flight metadata dataset to demonstrate joinWith:

case class FlightMetadata(count: BigInt, randomData: BigInt)

val flightsMeta = spark.range(500).map(x => (x, scala.util.Random.nextLong))

.withColumnRenamed("\_1", "count").withColumnRenamed("\_2", "randomData")

.as[FlightMetadata]

val flights2 = flights

.joinWith(flightsMeta, flights.col("count") === flightsMeta.col("count"))

#Notice that we end up with a Dataset of a sort of key-value pair, in which each row represents a Flight and the Flight Metadata. We can, of course, query these as a Dataset or a DataFrame with complex types:

flights2.selectExpr("\_1.DEST\_COUNTRY\_NAME")

#We can collect them just as we did before:

flights2.take(2)

**/\*Grouping and Aggregations\*/**

#Grouping and aggregations follow the same fundamental standards that we saw in the previous aggregation chapter, so groupBy rollup and cube still apply, but these return DataFrames instead of Datasets (you lose type information):

flights.groupBy("DEST\_COUNTRY\_NAME").count()

#This often is not too big of a deal, but if you want to keep type information around there are other groupings and aggregations that you can perform. An excellent example is the groupByKey method. This allows you to group by a specific key in the Dataset and get a typed Dataset in return. This function, however, doesn’t accept a specific column name but rather a function. This makes it possible for you to specify more sophisticated grouping functions that are much more akin to something like this:

flights.groupByKey(x => x.DEST\_COUNTRY\_NAME).count()

#It should be straightfoward enough to understand that this is a more expensive process than aggregating immediately after scanning, especially because it ends up in the same end result:

#This should motivate using Datasets only with user-defined encoding surgically and only where it makes sense. This might be at the beginning of a big data pipeline or at the end of one.

**/\*Low-Level APIs\*/**

#What Are the Low-Level APIs?

#There are two sets of low-level APIs: there is one for manipulating distributed data (RDDs), and another for distributing and manipulating distributed shared variables (broadcast variables and accumulators).

#When to Use the Low-Level APIs?

#You should generally use the lower-level APIs in three situations:

#You need some functionality that you cannot find in the higher-level APIs; for example, if you need very tight control over physical data placement across the cluster.

#You need to maintain some legacy codebase written using RDDs.

#You need to do some custom shared variable manipulation.

#Even if you are an advanced developer hoping to get the most out of Spark, we still recommend focusing on the Structured APIs. However, there are times when you might want to “drop down” to some of the lower-level tools to complete your task. You might need to drop down to these APIs to use some legacy code, implement some custom partitioner, or update and track the value of a variable over the course of a data pipeline’s execution. These tools give you more fine-grained control at the expense of safeguarding you from shooting yourself in the foot.

#How to Use the Low-Level APIs?

#A SparkContext is the entry point for low-level API functionality. You access it through the SparkSession, which is the tool you use to perform computation across a Spark cluster. But for now, you simply need to know that you can access a SparkContext via the following call:

spark.sparkContext

**/\*Resilient Distributed Datasets (RDDs)\*/**

#RDDs were the primary API in the Spark 1.X series and are still available in 2.X, but they are not as commonly used. However, as we’ve pointed out earlier in this book, virtually all Spark code you run, whether DataFrames or Datasets, compiles down to an RDD. The Spark UI, covered in the next part of the book, also describes job execution in terms of RDDs. Therefore, it will behoove you to have at least a basic understanding of what an RDD is and how to use it.

#In short, an RDD represents an immutable, partitioned collection of records that can be operated on in parallel. Unlike DataFrames though, where each record is a structured row containing fields with a known schema, in RDDs the records are just Java, Scala, or Python objects of the programmer’s choosing. RDDs give you complete control because every record in an RDD is a just a Java or Python object. You can store anything you want in these objects, in any format you want. This gives you great power, but not without potential issues. Every manipulation and interaction between values must be defined by hand, meaning that you must “reinvent the wheel” for whatever task you are trying to carry out. Also, optimizations are going to require much more manual work, because Spark does not understand the inner structure of your records as it does with the Structured APIs. For instance, Spark’s Structured APIs automatically store data in an optimzied, compressed binary format, so to achieve the same space-efficiency and performance, you’d also need to implement this type of format inside your objects and all the low-level operations to compute over it. Likewise, optimizations like reordering filters and aggregations that occur automatically in Spark SQL need to be implemented by hand. For this reason and others, we highly recommend using the Spark Structured APIs when possible.

#The RDD API is similar to the Dataset, which we saw in the previous part of the book, except that RDDs are not stored in, or manipulated with, the structured data engine. However, it is trivial to convert back and forth between RDDs and Datasets, so you can use both APIs to take advantage of each API’s strengths and weaknesses. We’ll show how to do this throughout this part of the book.

#Types of RDDs

#If you look through Spark’s API documentation, you will notice that there are lots of subclasses of RDD. For the most part, these are internal representations that the DataFrame API uses to create optimized physical execution plans. As a user, however, you will likely only be creating two types of RDDs: the “generic” RDD type or a key-value RDD that provides additional functions, such as aggregating by key. For your purposes, these will be the only two types of RDDs that matter. Both just represent a collection of objects, but key-value RDDs have special operations as well as a concept of custom partitioning by key. Let’s formally define RDDs. Internally, each RDD is characterized by five main properties:

* A list of partitions
* A function for computing each split
* A list of dependencies on other RDDs
* Optionally, a Partitioner for key-value RDDs (e.g., to say that the RDD is hashpartitioned)
* Optionally, a list of preferred locations on which to compute each split (e.g., block locations for a Hadoop Distributed File System [HDFS] file)

#The Partitioner is probably one of the core reasons why you might want to use RDDs in your code. Specifying your own custom Partitioner can give you significant performance and stability improvements if you use it correctly. This is discussed in more depth in Chapter 13 when we introduce Key–Value Pair RDDs. These properties determine all of Spark’s ability to schedule and execute the user program. Different kinds of RDDs implement their own versions of each of the aforementioned properties, allowing you to define new data sources. RDDs follow the exact same Spark programming paradigms that we saw in earlier chapters. They provide transformations, which evaluate lazily, and actions, which evaluate eagerly, to manipulate data in a distributed fashion. These work the same way as transformations and actions on DataFrames and Datasets. However, there is no concept of “rows” in RDDs; individual records are just raw Java/Scala/Python objects, and you manipulate those manually instead of tapping into the repository of functions that you have in the structured APIs. The RDD APIs are available in Python as well as Scala and Java. For Scala and Java, the performance is for the most part the same, the large costs incurred in manipulating the raw objects. Python, however, can lose a substantial amount of performance when using RDDs. Running Python RDDs equates to running Python user-defined functions (UDFs) row by row. Just as we saw in Chapter 6. We serialize the data to the Python process, operate on it in Python, and then serialize it back to the Java Virtual Machine (JVM). This causes a high overhead for Python RDD manipulations. Even though many people ran production code with them in the past, we recommend building on the Structured APIs in Python and only dropping down to RDDs if absolutely necessary.

#When to Use RDDs?

#In general, you should not manually create RDDs unless you have a very, very specific reason for doing so. They are a much lower-level API that provides a lot of power but also lacks a lot of the optimizations that are available in the Structured APIs. For the vast majority of use cases, DataFrames will be more efficient, more stable, and more expressive than RDDs. The most likely reason for why you’ll want to use RDDs is because you need fine-grained control over the physical distribution of data (custom partitioning of data).

**How to set name for a RDD?**

myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ")

words = spark.sparkContext.parallelize(myCollection, 2)

#Set name for RDD

words.setName("myWords")

#Get name of RDD

words.name

**Reading data from Data Sources?**

This creates an RDD for which each record in the RDD represents a line in that text file or files.

spark.sparkContext.textFile("/some/path/withTextFiles")

Alternatively, you can read in data for which each text file should become a single record. The use case here would be where each file is a file that consists of a large JSON object or some document that you will operate on as an individual:

spark.sparkContext.wholeTextFiles("/some/path/withTextFiles")

In this RDD, the name of the file is the first object and the value of the text file is the second string object.

**/\*Saving Files\*/**

RDD’s can be saved as textfile or Sequence Files.

Saving files means writing to plain-text files. With RDDs, you cannot actually “save” to a data source in the conventional sense. You must iterate over the partitions in order to save the contents of each partition to some external database. This is a low-level approach that reveals the underlying operation that is being performed in the higher-level APIs. Spark will take each partition, and write that out to the destination.

#saveAsTextFile

To save to a text file, you just specify a path and optionally a compression codec:

words.saveAsTextFile("file:/tmp/bookTitle")

To set a compression codec, we must import the proper codec from Hadoop. You can find these in the

org.apache.hadoop.io.compress library:

import org.apache.hadoop.io.compress.BZip2Codec

words.saveAsTextFile("file:/tmp/bookTitleCompressed", classOf[BZip2Codec])

#SequenceFiles

Spark originally grew out of the Hadoop ecosystem, so it has a fairly tight integration with a variety of Hadoop tools. A sequenceFile is a flat file consisting of binary key–value pairs. It is extensively used in MapReduce as input/output formats. Spark can write to sequenceFiles using the saveAsObjectFile method or by explicitly writing key–value pairs

words.saveAsObjectFile("/tmp/my/sequenceFilePath")

**What is Checkpointing?**

One feature not available in the DataFrame API is the concept of checkpointing. Checkpointing is the act of saving an RDD to disk so that future references to this RDD point to those intermediate partitions on disk rather than recomputing the RDD from its original source. This is similar to caching except that it’s not stored in memory, only disk. This can be helpful when performing iterative computation, similar to the use cases for caching:

spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")

words.checkpoint()

Now, when we reference this RDD, it will derive from the checkpoint instead of the source data. This can be a helpful optimization.

**What is foreachPartition?**

foreachPartition simply iterates over all the partitions of the data. The difference is that the function has no return value. This makes it great for doing something with each partition like writing it out to a database. In fact, this is how many data source connectors are written. You can create our own text file source if you want by specifying outputs to the temp directory with a random ID:

words.foreachPartition { iter =>

import java.io.\_

import scala.util.Random

val randomFileName = new Random().nextInt()

val pw = new PrintWriter(new File(s"/tmp/random-file-${randomFileName}.txt"))

while (iter.hasNext) {

pw.write(iter.next())

}

pw.close()

}

You’ll find these two files if you scan your /tmp directory.

**What is glom?**

glom is an interesting function that takes every partition in your dataset and converts them to arrays. This can be useful if you’re going to collect the data to the driver and want to have an array for each partition. However, this can cause serious stability issues because if you have large partitions or a large number of partitions, it’s simple to crash the driver. In the following example, you can see that we get two partitions and each word falls into one partition each:

words.glom().collect()

**/\*Key-Value Basics (Key-Value RDDs)\*/**

#Whenever you see ByKey in a method name, it means that you can perform this only on a PairRDD type.

**How to change normal rdd to key-value rdd?**

words.map(word => (word.toLowerCase, 1))

or

val keyword = words.keyBy(word => word.toLowerCase.toSeq(0).toString)

**How to perform transformations only on values in key-value paired rdd?**

#Using mapValues method

If we have a tuple, Spark will assume that the first element is the key, and the second is the value. When in this format, you can explicitly choose to map-over the values (and ignore the individual keys). Of course, you could do this manually, but this can help prevent errors when you know that you are just going to modify the values:

keyword.mapValues(word => word.toUpperCase).collect()

**How to extract Keys and Values?**

When we are in the key–value pair format, we can also extract the specific keys or values by using the following methods:

keyword.keys.collect()

keyword.values.collect()

**What is lookup method?**

One interesting task you might want to do with an RDD is look up the result for a particular key. Note that there is no enforcement mechanism with respect to there being only one key for each input, so if we lookup “s”, we are going to get both values associated with that key:

keyword.lookup("s")

**What is zip?**

Zip combines two RDDs, so it’s worth labeling it as a join. zip allows you to “zip” together two RDDs, assuming that they have the same length. This creates a PairRDD. The two RDDs must have the same number of partitions as well as the same number of elements:

val numRange = sc.parallelize(0 to 9, 2)

words.zip(numRange).collect()

**/\*Controlling Partitions\*/**

#coalesce

Coalesce effectively collapses partitions on the same worker in order to avoid a shuffle of the data when repartitioning. For instance, our words RDD is currently two partitions, we can collapse that to one partition by using coalesce without bringing about a shuffle of the data:

words.coalesce(1).getNumPartitions

#repartition

The repartition operation allows you to repartition your data up or down but performs a shuffle across nodes in the process. Increasing the number of partitions can increase the level of parallelism when operating in map- and filter-type operations:

words.repartition(10)

**Why do we need custom partitioner?**

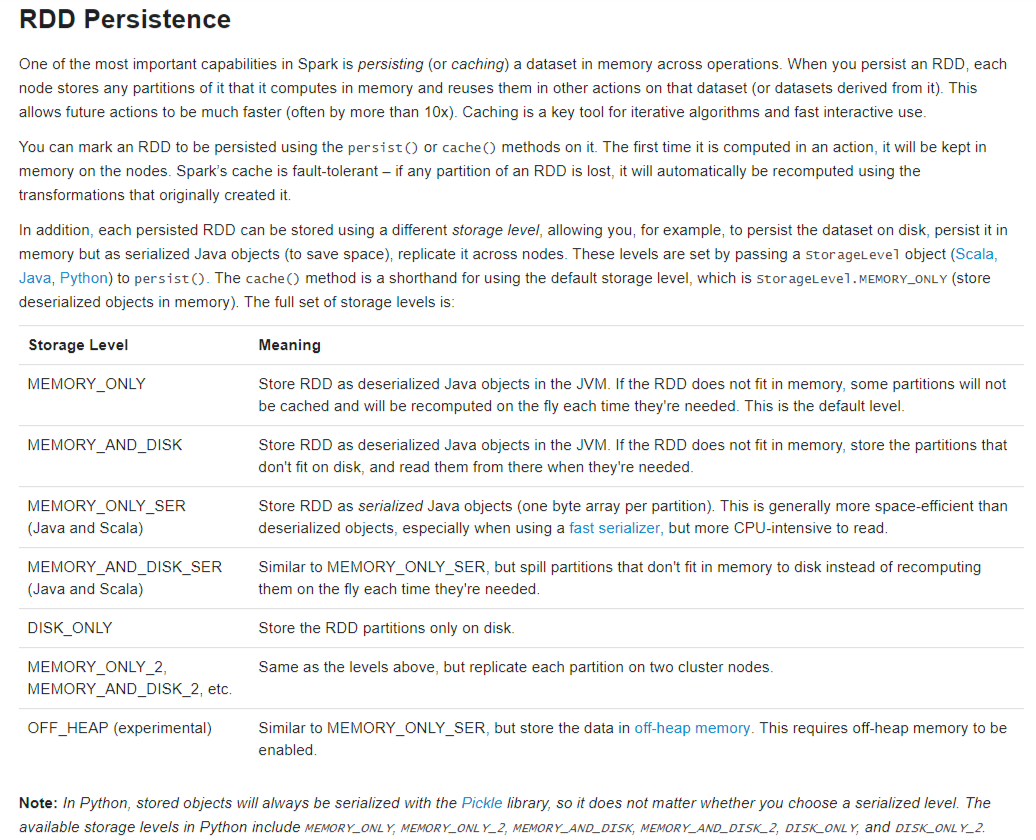
#Custom Partitioning

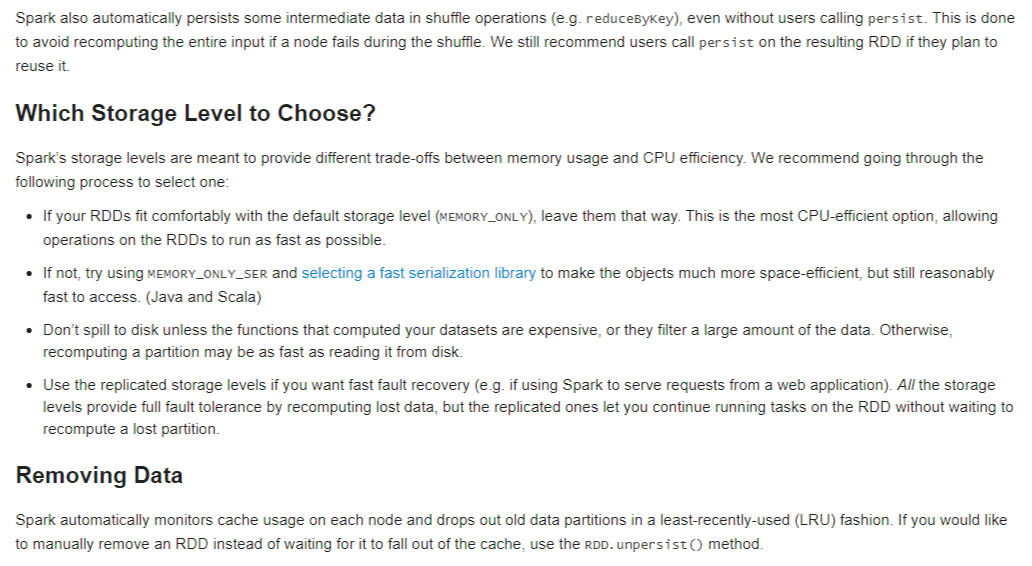
This ability is one of the primary reasons you’d want to use RDDs. Custom partitioners are not available in the Structured APIs because they don’t really have a logical counterpart. They’re a lowlevel, implementation detail that can have a significant effect on whether your jobs run successfully. The canonical example to motivate custom partition for this operation is PageRank whereby we seek to control the layout of the data on the cluster and avoid shuffles. In our shopping dataset, this might mean partitioning by each customer ID (we’ll get to this example in a moment). In short, the sole goal of custom partitioning is to even out the distribution of your data across the cluster so that you can work around problems like data skew. If you’re going to use custom partitioners, you should drop down to RDDs from the Structured APIs, apply your custom partitioner, and then convert it back to a DataFrame or Dataset. This way, you get the best of both worlds, only dropping down to custom partitioning when you need to. To perform custom partitioning you need to implement your own class that extends Partitioner. You need to do this only when you have lots of domain knowledge about your problem space—if you’re just looking to partition on a value or even a set of values (columns), it’s worth just doing it in the DataFrame API.

Spark has two built-in Partitioners that you can leverage off in the RDD API, a HashPartitioner for discrete values and a RangePartitioner. These two work for discrete values and continuous values, respectively. Spark’s Structured APIs will already use these, although we can use the same thing in RDDs:

Although the hash and range partitioners are useful, they’re fairly rudimentary. At times, you will need to perform some very low-level partitioning because you’re working with very large data and large key skew. Key skew simply means that some keys have many, many more values than other keys. You want to break these keys as much as possible to improve parallelism and prevent OutOfMemoryErrors during the course of execution. One instance might be that you need to partition more keys if and only if the key matches a certain

format. For instance, we might know that there are two customers in your dataset that always crash your analysis and we need to break them up further than other customer IDs. In fact, these two are so skewed that they need to be operated on alone, whereas all of the others can be lumped into large groups.



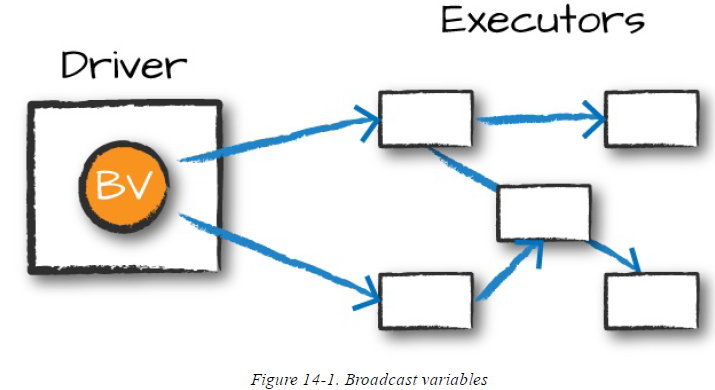


**/\*Distributed Shared Variables\*/**

The second kind of low-level API in Spark is two types of “distributed shared variables”: broadcast variables and accumulators

**What are Broadcast Variables?**

Broadcast variables are a way you can share an immutable value efficiently around the cluster without encapsulating that variable in a function closure. The normal way to use a variable in your driver node inside your tasks is to simply reference it in your function closures (e.g., in a map operation), but this can be inefficient, especially for large variables such as a lookup table or a machine learning model. The reason for this is that when you use a variable in a closure, it must be deserialized on the worker nodes many times (one per task). Moreover, if you use the same variable in multiple Spark actions and jobs, it will be re-sent to the workers with every job instead of once. This is where broadcast variables come in. Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task. The canonical use case is to pass around a large lookup table that fits in memory on the executors and use that in a function



val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ")

val words = spark.sparkContext.parallelize(myCollection, 2)

val supplementalData = Map("Spark" -> 1000, "Definitive" -> 200,"Big" -> -300, "Simple" -> 100)

We can broadcast above structure across Spark and reference it by using suppBroadcast. This value is immutable and is lazily replicated across all nodes in the cluster when we trigger an action:

val suppBroadcast = spark.sparkContext.broadcast(supplementalData)

We reference this variable via the value method, which returns the exact value that we had earlier. This method is accessible within serialized functions without having to serialize the data. This can save you a great deal of serialization and deserialization costs because Spark transfers data more efficiently around the cluster using broadcasts:

suppBroadcast.value

Now we could transform our RDD using this value. In this instance, we will create a key–value pair according to the value we might have in the map. If we lack the value, we will simply replace it with 0:

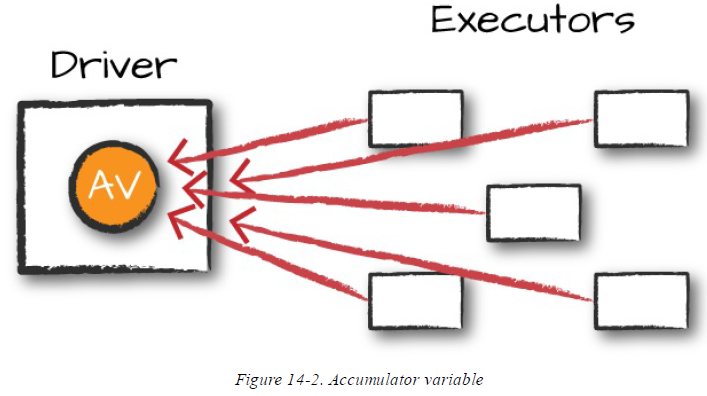
words.

map(word => (word, suppBroadcast.value.getOrElse(word, 0))).sortBy(wordPair => wordPair.\_2).collect()

The only difference between this and passing it into the closure is that we have done this in a much more efficient manner (Naturally, this depends on the amount of data and the number of executors. For very small data (low KBs) on small clusters, it might not be). Although this small dictionary probably is not too large of a cost, if you have a much larger value, the cost of serializing the data for every task can be quite significant. One thing to note is that we used this in the context of an RDD; we can also use this in a UDF or in a Dataset and achieve the same result.

**What are Accumulators?**

Spark’s second type of shared variable, are a way of updating a value inside of a variety of transformations and propagating that value to the driver node in an efficient and fault-tolerant way.



Accumulators provide a mutable variable that a Spark cluster can safely update on a per-row basis. You can use these for debugging purposes (say to track the values of a certain variable per partition in order to intelligently use it over time) or to create low-level aggregation. Accumulators are variables that are “added” to only through an associative and commutative operation and can therefore be efficiently supported in parallel. You can use them to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

For accumulator updates performed inside actions only, Spark guarantees that each task’s update to the accumulator will be applied only once, meaning that restarted tasks will not update the value. In transformations, you should be aware that each task’s update can be applied more than once if tasks or job stages are reexecuted.

Accumulators do not change the lazy evaluation model of Spark. If an accumulator is being updated within an operation on an RDD, its value is updated only once that RDD is actually computed (e.g., when you call an action on that RDD or an RDD that depends on it). Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like map(). Accumulators can be both named and unnamed. Named accumulators will display their running results in the Spark UI, whereas unnamed ones will not.

case class Flight(DEST\_COUNTRY\_NAME: String,ORIGIN\_COUNTRY\_NAME: String,count: BigInt)

val flightsDF = spark.read.

parquet("/user/chaitanyapolipalli/flight-data/parquet/2010-summary.parquet/")

val flights = flightsDF.as[Flight]

#Unnamed Accumulator

import org.apache.spark.util.LongAccumulator

val accUnnamed = new LongAccumulator

val acc = spark.sparkContext.register(accUnnamed)

#Named Accumulator

val accChina = new LongAccumulator

val accChina2 = spark.sparkContext.longAccumulator("China")

spark.sparkContext.register(accChina, "China")

flights.foreach(flight\_row => accChinaFunc(flight\_row))

accChina.value

#Custom Accumulators

Although Spark does provide some default accumulator types, sometimes you might want to build your own custom accumulator. In order to do this you need to subclass the AccumulatorV2 class. There are several abstract methods that you need to implement, as you can see in the example that follows.

import scala.collection.mutable.ArrayBuffer

import org.apache.spark.util.AccumulatorV2

val arr = ArrayBuffer[BigInt]()

class EvenAccumulator extends AccumulatorV2[BigInt, BigInt] {

private var num:BigInt = 0

def reset(): Unit = {

this.num = 0

}

def add(intValue: BigInt): Unit = {

if (intValue % 2 == 0) {

this.num += intValue

}

}

def merge(other: AccumulatorV2[BigInt,BigInt]): Unit = {

this.num += other.value

}

def value():BigInt = {

this.num

}

def copy(): AccumulatorV2[BigInt,BigInt] = {

new EvenAccumulator

}

def isZero():Boolean = {

this.num == 0

}

}

val acc = new EvenAccumulator

val newAcc = sc.register(acc, "evenAcc")

// in Scala

acc.value // 0

flights.foreach(flight\_row => acc.add(flight\_row.count))

acc.value // 31390

**Difference between coalesce and repartition?**

<https://stackoverflow.com/questions/31610971/spark-repartition-vs-coalesce>

<https://hackernoon.com/managing-spark-partitions-with-coalesce-and-repartition-4050c57ad5c4>