



PySpark_DataFrame_Operations:(Vijay Bhaskar Reddy)

- 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
- 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.

Creating DataFrame:

```
df = spark.read.format("json").load("/data/flight-data/json/2015-summary.json")
df.printSchema()
```

schema importance: it is often a good idea to define your schemas manually, especially when working with untyped data sources like CSV and JSON because schema inference can vary depending on the type of data that you read in.

Columns:

we can use :

- 1.col("name_of_col")
- 2.column("name_of_col")

Expressions:

- An expression is a set of transformations on one or more values in a record in a DataFrame.
- Think of it like a function that takes as input one or more column names, resolves them, and then potentially applies more expressions to create a single value for each record in the dataset.
- `expr("someCol") <=> col("someCol")`
- `expr("someCol - 5") <=> col("someCol") - 5,expr("someCol") - 5.`
- Columns are just expressions.
- Columns and transformations of those columns compile to the same logical plan as parsed expressions.

Records and Rows:

In Spark, each row in a DataFrame is a single record

Row objects internally represent arrays of bytes.

The byte array interface is never shown

to users because we only use column expressions to manipulate them.

```
from pyspark.sql import Row
myRow = Row("Hello", None, 1, False)
```

💡 It's important to note that only DataFrames have schemas. Rows themselves do not have schemas.

- Accessing data in rows is equally as easy: you just specify the position that you would like.

```
myRow[0]
myRow[2]
```

DataFrame Transformations:

we will move onto manipulating DataFrames.

- We can add rows or columns
- We can remove rows or columns
- We can transform a row into a column (or vice versa)
- We can change the order of rows based on the values in columns

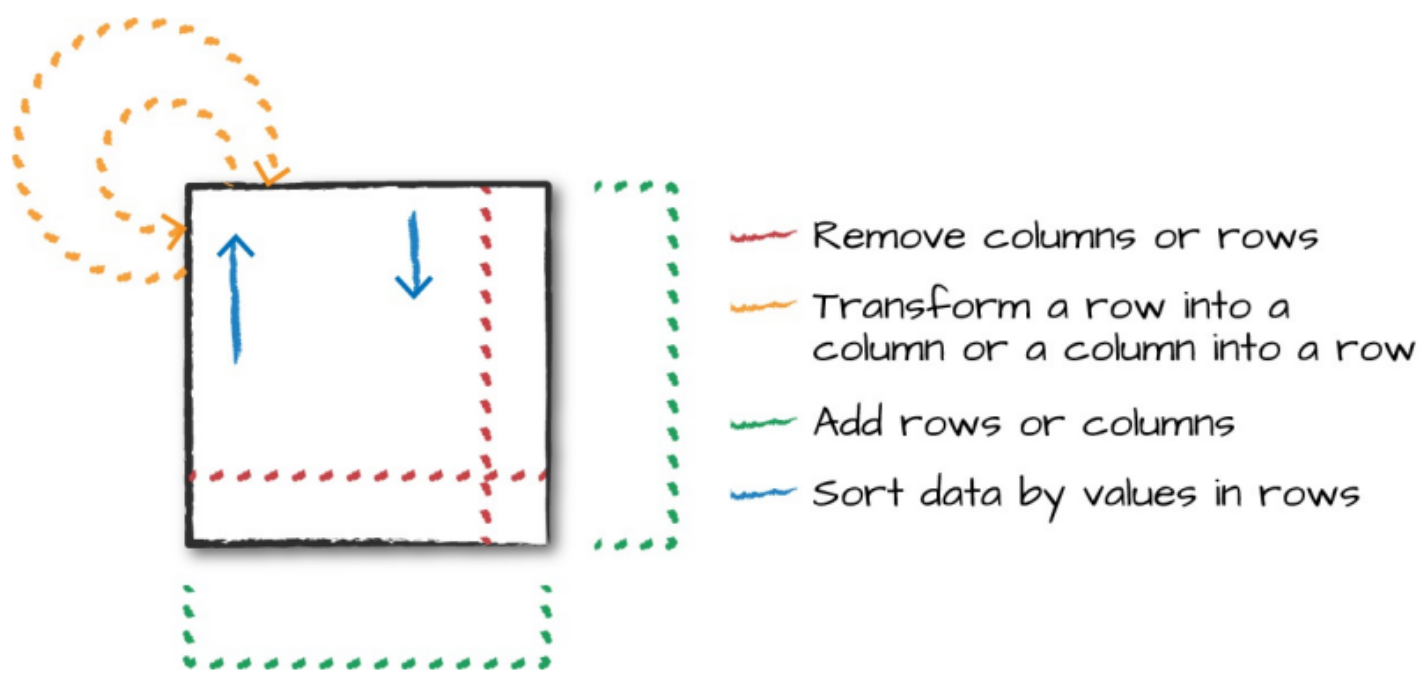


Figure 5-2. Different kinds of transformations

DataFrame creation using StructType:

```
from pyspark.sql import Row
from pyspark.sql.types import StructField, StructType, StringType, LongType
myManualSchema = StructType([
    StructField("some", StringType(), True),
    StructField("col", StringType(), True),
    StructField("names", LongType(), False)
])
myRow = Row("Hello", None, 1)
myDf = spark.createDataFrame([myRow], myManualSchema)
```

```
myDf.show()
```

Giving an output of:

```
+-----+-----+-----+
| some | col | names |
+-----+-----+-----+
| Hello | null | 1 |
+-----+-----+-----+
```

The **select** method when you’re working with *columns or expressions*, and the **selectExpr** method when you’re working with *expressions in strings*.

select and selectExpr:

- **select** and **selectExpr** allow you to do the DataFrame equivalent of SQL queries on a table of data
- The easiest way is just to use the select method and pass in the column names as strings with which you would like to work:

```
#In python
df.select("DEST_COUNTRY_NAME").show(2)
-- in SQL
SELECT DEST_COUNTRY_NAME FROM dfTable LIMIT 2
```

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

```
#in python
df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)
-- in SQL
SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME FROM dfTable LIMIT 2
Giving an output of:
+-----+-----+-----+
| DEST_COUNTRY_NAME | ORIGIN_COUNTRY_NAME |
+-----+-----+-----+
| United States | Romania |
| United States | Croatia |
+-----+-----+-----+
```

- As discussed in “Columns and Expressions”, you can refer to columns in a number of different ways, all you need to keep in mind is that you can use them interchangeably:

#in Python

```
from pyspark.sql.functions import expr, col, column
df.select(
    expr("DEST_COUNTRY_NAME"),
    col("DEST_COUNTRY_NAME"),
    column("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:

```
#in Python
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)
-- in SQL
SELECT DEST_COUNTRY_NAME as destination FROM dfTable LIMIT 2
```

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

```
#in Python

df.select(expr("DEST_COUNTRY_NAME as destination").alias("DEST_COUNTRY_NAME"))\
.show(2)
```

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)
```

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.

```
df.selectExpr(
    "*", # all original columns
    "(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")\
.show(2)
-- in SQL
SELECT *, (DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry
FROM dfTable
LIMIT 2
```

Giving an output of:

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count	withinCountry
United States	Romania	15	false
United States	Croatia	1	false

advantage of the functions that we have. These look just like what we have been showing so far:

select() method is useful when you simply need to select a subset of columns from a particular Spark DataFrame. On the other hand, **selectExpr()** comes in handy when you need to select particular columns while at the same time you also need to apply some sort of transformation over particular column(s)

```
#in python
df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(2)
-- in SQL
```

```
SELECT avg(count), count(distinct(DEST_COUNTRY_NAME)) FROM dfTable LIMIT 2
Giving an output of:
+-----+-----+
| avg(count)|count(DISTINCT DEST_COUNTRY_NAME)|
+-----+-----+
|1770.765625| 132|
+-----+-----+
```

Literals:

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:

```
#in Python
from pyspark.sql.functions import lit
df.select(expr("*"), lit(1).alias("One")).show(2)
```

Adding Columns

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)
-- in SQL
SELECT *, 1 as numberOne FROM dfTable LIMIT 2
Giving an output of:
+-----+-----+-----+-----+
| DEST_COUNTRY_NAME | ORIGIN_COUNTRY_NAME | count | numberOne |
+-----+-----+-----+-----+
| United States    | Romania             | 15    | 1         |
| United States    | Croatia             | 1     | 1         |
+-----+-----+-----+-----+
```

Another way

```
df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME"))\
.show(2)
```

Notice that the **withColumn function takes two arguments**: the column name and the expression that will create the value for that given row in the DataFrame. Interestingly, we can also rename a column this way. The SQL syntax is the same as we had previously, so we can omit it in this example:

```
df.withColumn("Destination", expr("DEST_COUNTRY_NAME")).columns
Resulting in:
... DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count, Destination
```

Renaming Columns

Although we can rename a column in the manner that we just described, another alternative is to use the withColumnRenamed method. This will rename the column with the name of the string in the first argument to the string in the second argument:

```
in Python
```



```
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns
... dest, ORIGIN_COUNTRY_NAME, count
```

Case Sensitivity

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

```
-- in SQL
set spark.sql.caseSensitive true
```

Removing Columns:

- We can remove the column or delete the column using key word Drop

```
df.drop("ORIGIN_COUNTRY_NAME").columns
#We can drop multiple columns by passing in multiple columns as arguments:
dfWithLongColName.drop("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME")
```

Changing a Column’s Type (cast):

Sometimes, we might need to convert from one type to another; for example, if we have a set of StringType that should be integers. We can convert columns from one type to another by casting the column from one type to another. For instance, let’s convert our count column from an integer to a type Long:

```
df.withColumn("count2", col("count").cast("long"))
-- in SQL
SELECT *, cast(count as long) AS count2 FROM dfTable
```

Filtering Rows:

▼ There are two methods to perform this operation: you can use where or filter and they both will perform the same operation and accept the same argument types when used with DataFrames. We will stick to where because of its familiarity to SQL; however, filter is valid as well.

```
df.filter(col("count") < 2).show(2)
df.where("count < 2").show(2)
-- in SQL
SELECT * FROM dfTable WHERE count < 2 LIMIT 2
Giving an output of:
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| United States| Croatia| 1|
| United States| Singapore| 1|
+-----+-----+-----+
```

▼ 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:

```
df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") != "Croatia")\
.show(2)
-- in SQL
SELECT * FROM dfTable WHERE count < 2 AND ORIGIN_COUNTRY_NAME != "Croatia"
LIMIT 2
Giving an output of:
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----+-----+-----+
| United States| Singapore| 1|
| Moldova| United States| 1|
+-----+-----+-----+
```

Getting Unique Rows:

The way we do this is by using the distinct method on a DataFrame

```
# in Python
df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct().count()
-- in SQL
SELECT COUNT(DISTINCT(ORIGIN_COUNTRY_NAME, DEST_COUNTRY_NAME)) FROM dfTable
Results in 256.

# in Python
df.select("ORIGIN_COUNTRY_NAME").distinct().count()
-- in SQL
SELECT COUNT(DISTINCT ORIGIN_COUNTRY_NAME) FROM dfTable
Results in 125.
```

- Random Sampling
- Random splits

Concatenating and Appending Rows (Union):

▼ As you learned in the previous section, DataFrames are immutable. This means users cannot append to DataFrames because that would be changing it. To append to a DataFrame, you must union the original DataFrame along with the new DataFrame. This just concatenates the two DataFrames. To union two DataFrames, you must be sure that they have the **same schema and number of columns** otherwise, the union will fail.

```
# in Python
from pyspark.sql import Row
schema = df.schema
newRows = [
    Row("New Country", "Other Country", 5L),
    Row("New Country 2", "Other Country 3", 1L)
]
parallelizedRows = spark.sparkContext.parallelize(newRows)
newDF = spark.createDataFrame(parallelizedRows, schema)
# in Python
df.union(newDF)\
.where("count = 1")\
.where(col("ORIGIN_COUNTRY_NAME") != "United States")\
.show()
Giving the output of:
+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
```

```
+-----+-----+-----+
| United States| Croatia| 1|
...
| United States| Namibia| 1|
| New Country 2| Other Country 3| 1|
+-----+-----+-----+
```

Sorting Rows:

When we sort the values in a DataFrame, we always want to sort with either the largest or smallest values at the top of a DataFrame. There are two equivalent operations to do this sort and `orderBy` that work the exact same way. They accept both column expressions and strings as well as multiple columns. The default is to sort in ascending order

in Python

```
df.sort("count").show(5)
df.orderBy("count", "DEST_COUNTRY_NAME").show(5)
df.orderBy(col("count"), col("DEST_COUNTRY_NAME")).show(5)
```

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:

in Python

```
from pyspark.sql.functions import desc, asc
df.orderBy(expr("count desc")).show(2)
df.orderBy(col("count").desc(), col("DEST_COUNTRY_NAME").asc()).show(2)
-- in SQL
SELECT * FROM dfTable ORDER BY count DESC, DEST_COUNTRY_NAME ASC LIMIT 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:

in Python

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

Limit

Oftentimes, you might want to restrict what you extract from a DataFrame; for example, you might want just the top ten of some DataFrame. You can do this by using the `limit` method:

```
// in Scala
df.limit(5).show()
```

in Python

```
df.limit(5).show()
```



```
-- in SQL
SELECT * FROM dfTable LIMIT 6
// in Scala
df.orderBy(expr("count desc")).limit(6).show()

# in Python

df.orderBy(expr("count desc")).limit(6).show()
-- in SQL
SELECT * FROM dfTable ORDER BY count desc LIMIT 6
```

Coalesce:

Coalesce is a function in PySpark that is **used to work with the partition data in a PySpark Data Frame**. The Coalesce method is used to decrease the number of partitions in a Data Frame. The **coalesce function avoids the full shuffling of data**.

Repartition and Coalesce:

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(happens) 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:

```
in Python

df.rdd.getNumPartitions() # 1

in Python

df.repartition(5)
If you know that you're going to be filtering by a certain column often, it can be
repartitioning based on that column:
#in Python

df.repartition(col("DEST_COUNTRY_NAME"))
You can optionally specify the number of partitions you would like, too:

#in Python

df.repartition(5, col("DEST_COUNTRY_NAME"))
Coalesce, on the other hand, will not incur a full shuffle and will try to combine
operation will shuffle your data into five partitions based on the destination coun
then coalesce them (without a full shuffle):

#in Python

df.repartition(5, col("DEST_COUNTRY_NAME")).coalesce(2)
```

Collecting Rows to the Driver:

As discussed in previous chapters, Spark maintains the state of the cluster in the driver. There are times when you'll want to collect some of your data to the driver in order to manipulate it on

your local machine.

Thus far, we did not explicitly define this operation. However, we used several different methods for doing so that are effectively all the same. `collect` gets all data from the entire `DataFrame`, `take` selects the first N rows, and `show` prints out a number of rows nicely.

in Python

```
collectDF = df.limit(10)
collectDF.take(5) # take works with an Integer count
collectDF.show() # this prints it out nicely
collectDF.show(5, False)
collectDF.collect()
```

There's an additional way of collecting rows to the driver in order to iterate over the entire dataset. The method `toLocalIterator` collects partitions to the driver as an iterator. This method allows you to iterate over the entire dataset partition-by-partition in a serial manner:

```
collectDF.toLocalIterator()
```

Random Samples:

Sometimes, you might just want to sample some random records from your `DataFrame`. You can do this by using the `sample` method on a `DataFrame`, which makes it possible for you to specify a fraction of rows to extract from a `DataFrame` and whether you'd like to sample with or without replacement:

in Python

```
seed = 5
withReplacement = False
fraction = 0.5
df.sample(withReplacement, fraction, seed).count()
Giving an output of 126.
```

Random Splits:

Random splits can be helpful when you need to break up your `DataFrame` into a random “splits” of the original `DataFrame`. This is often used with machine learning algorithms to create training, validation, and test sets. In this next example, we'll split our `DataFrame` into two different `DataFrames` by setting the weights by which we will split the `DataFrame` (these are the arguments to the function). Because this method is designed to be randomized, we will also specify a seed (just replace seed with a number of your choosing in the code block). It's important to note that if you don't specify a proportion for each `DataFrame` that adds up to one, they will be normalized so that they do

in Python

```
dataFrames = df.randomSplit([0.25, 0.75], seed)
dataFrames[0].count() > dataFrames[1].count()
```

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:

Working with Nulls in Data:

There are two ways to handle the null values:

- 1.Drop the nulls.
- 2.fill the null values with appropriate values.

ifnull, nullif, nvl, and nvl2

There are several other SQL functions that you can use to achieve similar things. ifnull allows you to select the second value if the first is null, and defaults to the first. Alternatively, you could use nullif, which returns null if the two values are equal or else returns the second if they are not. nvl returns the second value if the first is null, but defaults to the first. Finally, nvl2 returns the second value if the first is not null; otherwise, it will return the last specified value (else_value in the following example):

Drop:

The default is to drop any row in which any value is null.

```
df.na.drop("any")
df.na.drop("all")
df.na.drop("all", subset=['col_1', 'col_2'])
```

Fill:

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.

```
df.na.fill('All Null values')
values={'a':10, 'b':22}
df.na.fill(values)
```

Replace:

Probably the most common use case is to replace all values in a certain column according to their current value.

```
df.na.replace([""], ["UnKnown"], 'col')
```

Complex Types:

- structs
- arrays
- maps

structs :

You can think of structs as DataFrames within DataFrames.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", "")
```

```
# in Python
from pyspark.sql.functions import struct
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"))
(or)
complexDF.select("complex.*")
```

Arrays:

To define arrays, let's work through a use case.

“our objective is to take every single word in our Description column and convert that into a row in our DataFrame.”

split:

We do this by using the split function and specify the delimiter and it will split based on split values or delimiter.

Array Length:(size)

We can determine the array's length by querying for its size.

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

Explode:

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.

split
explode

"Hello World" , "other col" → ["Hello" , "World"], "other col" → "Hello" , "other col"
 "World" , "other col"

Figure 6-1. Exploding a column of text

```
# in Python
from pyspark.sql.functions import split, explode
df.withColumn("splitted", split(col("Description"), " "))\
  .withColumn("exploded", explode(col("splitted")))\
  .select("Description", "InvoiceNo", "exploded").show(2)
```

Maps:

Maps are created by using the map function and key-value pairs of columns

```
# in Python
from pyspark.sql.functions import create_map
```

```
df.select(create_map(col("Description"), col("InvoiceNo")).alias("complex_map"))\
.show(2)
```

Working with JSON

Spark has some unique support for working with JSON data. You can operate directly on strings of JSON in Spark and parse from JSON or extract JSON objects.

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

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:

User-Defined Functions:

One of the most powerful things that you can do in Spark is define your own functions. These user-defined functions (UDFs) make it possible for you to write your own custom transformations using Python or Scala and even use external libraries. 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.

```
# in Python
udfExampleDF = spark.range(5).toDF("num")
def power3(double_value):
    return double_value ** 3
power3(2.0)
```

Now that we’ve created these functions and tested them, we need to register them with Spark so that we can use them on all of our worker machines. Spark will serialize the function on the driver and transfer it over the network to all executor processes. This happens regardless of language.

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.

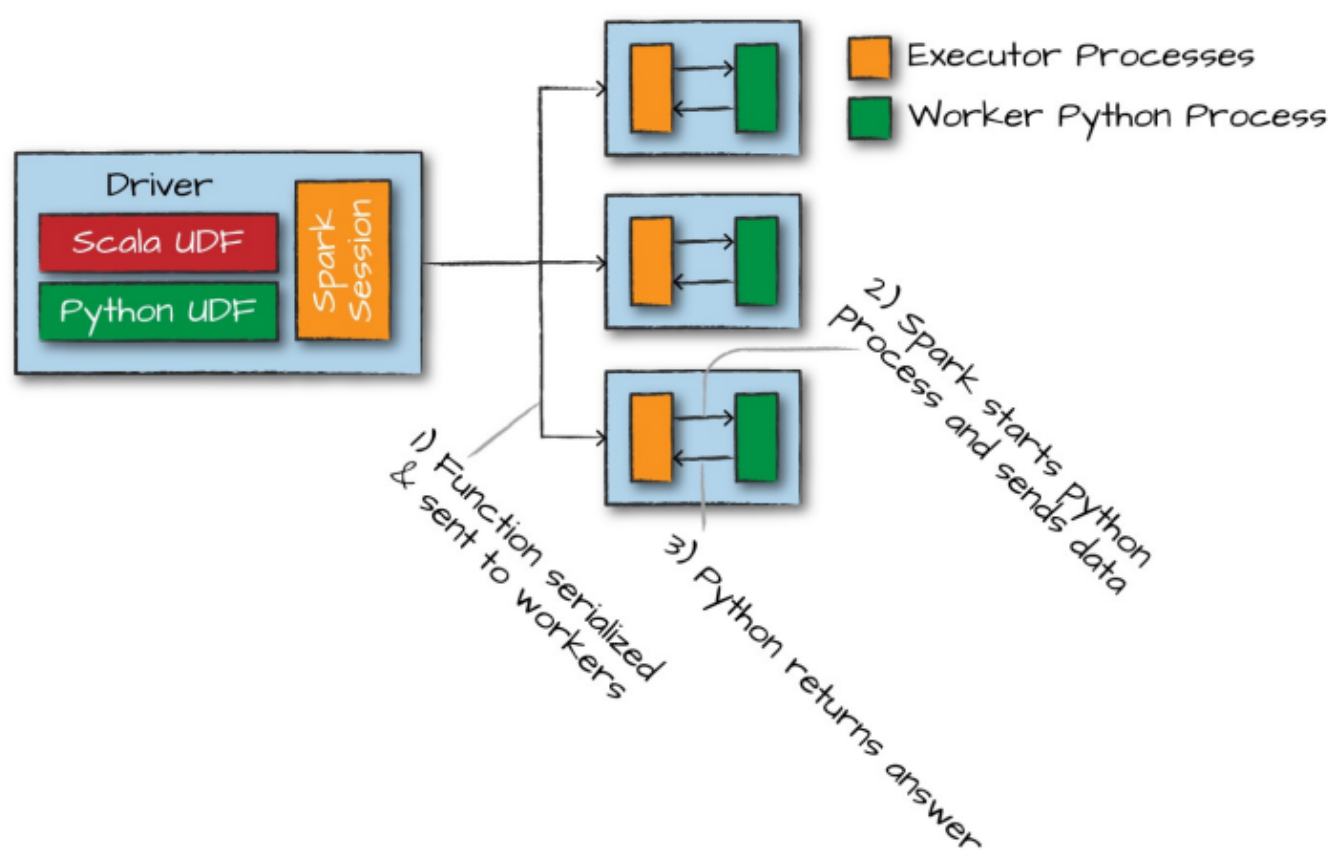


Figure 6-2. Figure caption



Note:

UDF's are the most expensive operations hence use them only you have no choice and when essential. When you creating UDF's you need to design them very carefully otherwise you will come across optimization & performance issues.

- In PySpark, you create a function in a Python syntax and wrap it with PySpark SQL `udf()` or register it as udf and use it on DataFrame and SQL respectively.
- Before you create any UDF, do your research to check if the similar function you wanted is already available in [Spark SQL Functions](#).

```
spark.udf.register("squaredWithPython", squared_typed, LongType())
```

```
# in Python
from pyspark.sql.types import IntegerType, DoubleType
spark.udf.register("power3py", power3, DoubleType())
# in Python
udfExampleDF.selectExpr("power3py(num)").show(2)
# registered via Python
```

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:

=====

Aggrigations:

- Aggregating is the act of collecting something together and is a cornerstone of big data analytics.

The simplest grouping is to just summarize a complete DataFrame by performing an aggregation in a select statement.

1.A “group by” allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns.

2.A “window” gives you the ability to specify one or more keys as well as one or more aggregation functions to transform the value columns. However, the rows input to the function are somehow related to the current row.

3.A “grouping set,” which you can use to aggregate at multiple different levels. Grouping sets are available as a primitive in SQL and via rollups and cubes in DataFrames.

4.A “rollup” makes it possible for you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized hierarchically.

5.A “cube” allows you to specify one or more keys as well as one or more aggregation functions to transform the value columns, which will be summarized across all combinations of columns.

Each grouping returns a ***RelationalGroupedDataset*** on which we specify our aggregations.

Aggregation Functions:

- count:

It is used to to count the number of values occured in the cloumn.

- countDistinct:

It is used to to count the number of values occured in the cloumn.

- approx_count_distinct:

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

```
from pyspark.sql.functions import approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show()
```

0.1 → Maximum estimation error .

first and last:

You can get the first and last values from a DataFrame by using these two obviously named functions. This will be based on the rows in the DataFrame, not on the values in the DataFrame:

```
#in Python
from pyspark.sql.functions import first, last
df.select(first("StockCode"), last("StockCode")).show()
```

min and max:

To extract the minimum and maximum values from a DataFrame, use the min and max functions

```
# in Python
from pyspark.sql.functions import min, max
df.select(min("Quantity"), max("Quantity")).show()
-- in SQL
SELECT min(Quantity), max(Quantity) FROM dfTable
```

sum:

Another simple task is to add all the values in a row using the sum function:

```
# in Python
from pyspark.sql.functions import sum
df.select(sum("Quantity")).show() # 5176450
-- in SQL
SELECT sum(Quantity) FROM dfTable
```

sumDistinct:

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

```
# in Python
from pyspark.sql.functions import sumDistinct
df.select(sumDistinct("Quantity")).show() # 29310
-- in SQL
SELECT SUM(Quantity) FROM dfTable -- 29310
```

avg:

Although you can calculate average by dividing sum by count, Spark provides an easier way to get that value via the avg or mean functions.

Variance and Standard Deviation

Calculating the mean naturally brings up questions about the variance and standard deviation. These are both measures of the spread of the data around the mean. The variance is the average of the squared differences from the mean, and the standard deviation is the square root of the variance. You can calculate these in Spark by using their respective functions. However, something to note is that Spark has both the formula for the sample standard deviation as well as the formula for the population standard deviation. These are fundamentally different statistical formulae, and we need to differentiate between them. By default, Spark performs the formula for the sample standard deviation or variance if you use the variance or stddev functions. You can also specify these explicitly or refer to the population standard deviation or variance:

```
# in Python
from pyspark.sql.functions import var_pop, stddev_pop
from pyspark.sql.functions import var_samp, stddev_samp
df.select(var_pop("Quantity"), var_samp("Quantity"),
stddev_pop("Quantity"), stddev_samp("Quantity")).show()
-- in SQL
SELECT var_pop(Quantity), var_samp(Quantity),
stddev_pop(Quantity), stddev_samp(Quantity)
FROM dfTable
```

skewness and kurtosis:(measurements of extreme points)

Skewness—>measures the asymmetry of the values in your data around the mean.

kurtosis—>measure of the tail of data.

```
# in Python
from pyspark.sql.functions import skewness, kurtosis
df.select(skewness("Quantity"), kurtosis("Quantity")).show()
-- in SQL
SELECT skewness(Quantity), kurtosis(Quantity) FROM dfTable
```

Covariance and Correlation:

Correlation measures the Pearson correlation coefficient, which is scaled between -1 and $+1$.

covariance is scaled according to the inputs in the data.

```
# in Python
from pyspark.sql.functions import corr, covar_pop, covar_samp
df.select(corr("InvoiceNo", "Quantity"), covar_samp("InvoiceNo", "Quantity"),
covar_pop("InvoiceNo", "Quantity")).show()
-- in SQL
SELECT corr(InvoiceNo, Quantity), covar_samp(InvoiceNo, Quantity),
covar_pop(InvoiceNo, Quantity)
FROM dfTable
```

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.

```
# in Python
from pyspark.sql.functions import collect_set, collect_list
df.agg(collect_set("Country"), collect_list("Country")).show()
-- in SQL
SELECT collect_set(Country), collect_set(Country) FROM dfTable
```

Grouping:

This is typically done on categorical data for which we group our data on one column and perform some calculations on the other columns that end up in that group.

```
df.groupBy("InvoiceNo", "CustomerId").count().show()
-- in SQL
SELECT count(*) FROM dfTable GROUP BY InvoiceNo, CustomerId
```

Grouping with Expressions:

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:

```
# in Python
from pyspark.sql.functions import count
df.groupBy("InvoiceNo").agg(
count("Quantity").alias("quan"),
expr("count(Quantity)")).show()
```

Grouping with Maps:

Sometimes, it can be easier to specify your transformations as a series of Maps for which the key is the column, and the value is the aggregation function (as a string) that you would like to perform. You can reuse multiple column names if you specify them inline, as well:

```
# in Python
df.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),expr("stddev_pop(Quantity)"))\
.show()
-- in SQL
SELECT avg(Quantity), stddev_pop(Quantity), InvoiceNo FROM dfTable
GROUP BY InvoiceNo
```

Window Functions:

- You can also use window functions to carry out some unique aggregations by either computing some aggregation on a specific “window” of data, which you define by using a reference to the current data. This window specification determines which rows will be passed in to this function. Now this is a bit abstract and probably similar to a standard group-by, so let’s differentiate them a bit more.
- A group-by takes data, and every row can go only into one grouping. **A window function calculates a return value for every input row of a table based on a group of rows**, called a frame. Each row can fall into one or more frames. A common use case is to take a look at a rolling average of some value for which each row represents one day. If you were to do this, each row would end up in seven different frames. We cover defining frames a little later, but for your reference, Spark supports three kinds of window functions:
ranking functions, analytic functions, and aggregate functions

Grouping Sets:

Thus far in this chapter, we’ve seen simple group-by expressions that we can use to aggregate on a set of columns with the values in those columns. However, sometimes we want something a bit more complete—

an aggregation across multiple groups. We achieve this by using grouping sets.

Grouping sets are a low-level tool for combining sets of aggregations together. They give you the ability to create arbitrary aggregation in their group-by statements.

Rollups:

. When we set our grouping keys of multiple columns, Spark looks at those as well as the actual combinations that are visible in the dataset.

Let’s create a rollup that looks across time (with our new Date column) and space (with the Country column)

```
# in Python
rolledUpDF = dfNotNull.rollup("Date", "Country").agg(sum("Quantity"))\
.selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")\
.orderBy("Date")
rolledUpDF.show()
```

Cube:

A cube takes the rollup to a level deeper. Rather than treating elements hierarchically, a cube does the same thing across all dimensions. This means that it won't just go by date over the entire time period, but also the country. To pose this as a question again, can you make a table that includes the following?

The total across all dates and countries

The total for each date across all countries

The total for each country on each date

The total for each country across all dates

The method call is quite similar, but instead of calling rollup, we call cube:

Pivot:

Pivots make it possible for you to convert a
row into a column

.

```
# in Python
pivoted = dfWithDate.groupBy("date").pivot("Country").sum()
```

Conclusion:

This chapter walked through the different types and kinds of aggregations that you can perform in Spark

JOINS:

we have many joins in the SQL and python here spark support the some of Joins that are:

1.INNER JOIN

2.OUTER JOIN

3.LEFT_OUTER JOIN :If there is no equivalent row in the RIGHT DataFrame, Spark will insert null

4.RIGHT_OUTER JOIN :If there is no equivalent row in the LEFT DataFrame, Spark will insert null

5.LEFT_SEMI JOIN: Returns only left table from **matched rows**

6.LEFT_ANTI JOIN: Returns only left table from **not matched rows**

7.NATURAL JOIN

8.CROSS JOIN

Some of complex join condition use cases:

1.Handling Duplicate Column Names:

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

solutions:

- 1.Dropping the column after the join.
- 2.Renaming a column before the join

Data Sources

Spark has six “core” data sources and hundreds of external data sources written by the community

Read API Structure:

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

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.

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

```
#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()
```

Table 9-1. Spark’s read modes

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 <code>_corrupt_record</code>
dropMalformed	Drops the row that contains malformed records
failFast	Fails immediately upon encountering malformed records

The default is permissive.

Write API Structure:


```
Syntax:
DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save()
```

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 foundation for writing data is quite similar to that of reading data. Instead of the `DataFrameReader`, we have the `DataFrameWriter`. Because we always need to write out some given data source.

```
dataframe.write.format("csv")
.option("mode", "OVERWRITE")
.option("dateFormat", "yyyy-MM-dd")
.option("path", "path/to/file(s)")
.save()
```

Save modes

Save modes specify what will happen if Spark finds data at the specified location (assuming all else equal)

Table 9-2. Spark’s save modes

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

The default is **errorIfExists**. This means that if Spark finds data at the location to which you’re writing, it will fail the write immediately.

parameters for read/write :

Table 9-3. CSV data source options

Read/write Key		Potential values	Default	Description
Both	sep	Any single string character	,	The single character that is used as separator for each field and value.
Both	header	true, false	false	A Boolean flag that declares whether the first line in the file(s) are the names of the columns.
Read	escape	Any string character	\	The character Spark should use to escape other characters in the file.
Read	inferSchema	true, false	false	Specifies whether Spark should infer column types when reading the file.
Read	ignoreLeadingWhiteSpace	true, false	false	Declares whether leading spaces from values being read should be skipped.
Read	ignoreTrailingWhiteSpace	true, false	false	Declares whether trailing spaces from values being read should be skipped.
Both	nullValue	Any string character	""	Declares what character represents a null value in the file.
Both	nanValue	Any string character	NaN	Declares what character represents a NaN or missing character in the CSV file.
Both	positiveInf	Any string or character	Inf	Declares what character(s) represent a positive infinite value.
Both	negativeInf	Any string or character	-Inf	Declares what character(s) represent a negative infinite value.
Both	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or	none	Declares what compression codec Spark should use to read or write the file.

Both	dateFormat	Any string or character that conforms to java’s SimpleDateFormat.	yyyy-MM-dd	Declares the date format for any columns that are date type.
Both	timestampFormat	Any string or character that conforms to java’s SimpleDateFormat.	yyyy-MM-dd’T’HH:mm:ss.SSSZZ	Declares the timestamp format for any columns that are timestamp type.
Read	maxColumns	Any integer	20480	Declares the maximum number of columns in the file.
Read	maxCharsPerColumn	Any integer	1000000	Declares the maximum number of characters in a column.
Read	escapeQuotes	true, false	true	Declares whether Spark should escape quotes that are found in lines.
Read	maxMalformedLogPerPartition	Any integer	10	Sets the maximum number of malformed rows Spark will log for each partition. Malformed records beyond this number will be ignored.
Write	quoteAll	true, false	false	Specifies whether all values should be enclosed in quotes, as opposed to just escaping values that have a quote character.
Read	multiline	true, false	false	This option allows you to read multiline CSV files where each logical row in the CSV file might span

- *In general, Spark will fail only at job execution time rather than DataFrame definition time—even if, for example, we point to a file that does not exist. This is due to lazy evaluation*

Writing csv Files:

```
#Read The file
csvFile = spark.read.format("csv")\
.option("header", "true")\
.option("mode", "FAILFAST")\
.option("inferSchema", "true")\
.load("/data/flight-data/csv/2010-summary.csv")
#For instance, we can take our CSV file and write it out as a TSV file quite easily

# in Python
#write into the file
csvFile.write.format("csv").mode("overwrite").option("sep", "\t")\
.save("/tmp/my-tsv-file.tsv")
```

Reading JSON Files

Let’s look at an example of reading a JSON file and compare the options that we’re seeing:

```
spark.read.format("json").option("mode", "FAILFAST")\
.option("inferSchema", "true")\
.load("/data/flight-data/json/2010-summary.json").show(5)
```

Writing JSON Files:

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:

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

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

```
# in Python
spark.read.format("parquet")\
.load("/data/flight-data/parquet/2010-summary.parquet").show(5)
```

Read/Write	Key	Potential Values	Default	Description
Write	compression or codec	None, uncompressed, bzip2, deflate, gzip, lz4, or snappy	None	Declares what compression codec Spark should use to read or write the file.
Read	mergeSchema	true, false	Value of the configuration spark.sql.parquet.mergeSchema	You can incrementally add columns to newly written Parquet files in the same table/folder. Use this option to enable or disable this feature.

Writing Parquet Files:

Writing Parquet is as easy as reading it. We simply specify the location for the file. The same partitioning rules apply:

```
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**.

Reading ORC File:

here the method to read orc file:

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

Writing ORC File:

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

Query PushDown:

Spark predicate push down to database allows for better optimized Spark queries. A predicate is a condition on a query that returns true or false, typically located in the `WHERE` clause. A predicate push down filters the data in the database query, reducing the number of entries retrieved from the database and improving query performance. By default the Spark Dataset API will automatically push down valid `WHERE` clauses to the database.

Reading from SQL Databases:

When it comes to reading a file, SQL databases are no different from the other data sources that we looked at earlier. As with those sources, we specify the format and options, and then load in the data:

```
# in Python
driver = "org.sqlite.JDBC"
path = "/data/flight-data/jdbc/my-sqlite.db"
url = "jdbc:sqlite:" + path
tablename = "flight_info"

# in Python
dbDataFrame = spark.read.format("jdbc").option("url", url)\
.option("dbtable", tablename).option("driver", driver).load()

# in Python
pgDF = spark.read.format("jdbc")\
.option("driver", "org.postgresql.Driver")\
.option("url", "jdbc:postgresql://database_server")\
.option("dbtable", "schema.tablename")\
.option("user", "username").option("password", "my-secret-password").load()
```

```
dbDataFrame.select("DEST_COUNTRY_NAME").distinct().show(5)
```

Writing to SQL Databases:

Writing out to SQL databases is just as easy as before. You simply specify the URI and write out the data according to the specified write mode that you want. In the following example, we specify overwrite, which overwrites the entire table. We'll use the CSV DataFrame that we defined earlier in order to do this:

```
newPath = "jdbc:sqlite://tmp/my-sqlite.db"
csvFile.write.jdbc(newPath, tablename, mode="overwrite", properties=props)
# in Python
spark.read.jdbc(newPath, tablename, properties=props).count()

csvFile.write.jdbc(newPath, tablename, mode="append", properties=props)
```

Text Files

Spark also allows you to read in plain-text files. Each line in the file becomes a record in the DataFrame. It is then up to you to transform it accordingly. As an example of how you would do this, suppose that you need to parse some Apache log files to some more structured format, or perhaps you want to parse some plain text for natural-language processing.

Reading 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:

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

Writing Text Files

When you write a text file, you need to be sure to have only one string column; otherwise, the write will fail:

```
csvFile.select("DEST_COUNTRY_NAME").write.text("/tmp/simple-text-file.txt")
# in Python
csvFile.limit(10).select("DEST_COUNTRY_NAME", "count")\
.write.partitionBy("count").text("/tmp/five-csv-files2py.csv")
```

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. As you can see from the list call:

```
ls /tmp/multiple.csv
```

```
/tmp/multiple.csv/part-00000-767df509-ec97-4740-8e15-4e173d365a8b.csv
```

```
/tmp/multiple.csv/part-00001-767df509-ec97-4740-8e15-4e173d365a8b.csv
```

```
/tmp/multiple.csv/part-00002-767df509-ec97-4740-8e15-4e173d365a8b.csv
```

```
/tmp/multiple.csv/part-00003-767df509-ec97-4740-8e15-4e173d365a8b.csv
```

```
/tmp/multiple.csv/part-00004-767df509-ec97-4740-8e15-4e173d365a8b.csv
```

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:

// in Scala

```
csvFile.limit(10).write.mode("overwrite").partitionBy("DEST_COUNTRY_NAME")
```

```
.save("/tmp/partitioned-files.parquet")
```

in Python

```
csvFile.limit(10).write.mode("overwrite").partitionBy("DEST_COUNTRY_NAME")\
    .save("/tmp/partitioned-files.parquet")
```

Upon writing, you get a list of folders in your Parquet “file”:

```
$ ls /tmp/partitioned-files.parquet
```

```
...
```

```
DEST_COUNTRY_NAME=Costa Rica/
```

```
DEST_COUNTRY_NAME=Egypt/
```

```
DEST_COUNTRY_NAME=Equatorial Guinea/
```

```
DEST_COUNTRY_NAME=Senegal/
```

```
DEST_COUNTRY_NAME=United States/
```

Each of these will contain Parquet files that contain that data where the previous predicate was true:

```
$ ls /tmp/partitioned-files.parquet/DEST_COUNTRY_NAME=Senegal/
```

```
part-00000-tid.....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")
```

```
$ ls /user/hive/warehouse/bucketedfiles/
```

```
part-00000-tid-1020575097626332666-8....parquet
```

```
part-00000-tid-1020575097626332666-8....parquet
```

```
part-00000-tid-1020575097626332666-8....parquet
```

```
...
```

Bucketing is supported only for Spark-managed tables. For more information on bucketing and partitioning, watch this talk from Spark Summit 2017.

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.

Chapter 10. Spark SQL

- Spark SQL is intended to operate as an online analytic processing (OLAP) database, not an online transaction processing (OLTP) database.
- **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.
- **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.
- You can express multiline queries quite simply by passing a multiline string into the function.

```
spark.sql("""SELECT user_id, department, first_name FROM professors
WHERE department IN
(SELECT name FROM department WHERE created_date >= '2016-01-01')""")
```

Catalog:

The highest level abstraction in Spark SQL is the Catalog. The Catalog is an abstraction for the storage of metadata about the data stored in your tables as well as other helpful things like databases, tables, functions, and views. The catalog is available in the `org.apache.spark.sql.catalog.Catalog` package and contains a number of helpful functions

Tables:

Tables are logically equivalent to a `DataFrame` in that they are a structure of data against which you run commands.

we can perform multiple operations:

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.

Global Managed Table:

- A managed table is a Spark SQL table for which Spark manages both the data and metadata. It will be available across the clusters.
- When you drop the table, it will lose both data and metadata.

```
dataframe.write.saveAsTable("my_table")
```

Global Unmanaged Table:

- Spark manages the metadata, while you control the data location. As soon as you add the `path` option in the `DataFrame` writer, it will be treated as a global unmanaged table, and the table will be available across the clusters.
- When you drop the table, it loses the metadata, whereas the table will not be lost.

```
dataframe.write.option('path', "<your-storage-path>").saveAsTable("my_table")
```

Local Table/ Temporary Table / Temporary View:

A Spark session is scoped. A local table is not accessible from other clusters and it is not registered in the metastore.

```
dataframe.createOrReplaceTempView()
```

Global Temporary View:

Spark application is scoped , global temporary views are tied to a system preserved temporary database `global_temp`. This **view** can shared across the different spark sessions.

```
dataframe.createOrReplaceGlobalTempView("my_global_view")
#Accessed by-->
spark.read.table("global_temp.my_global_view"
```


Global Perment View:

Persist a data frame as permanent view. The view defination is reordered in the underlying metastore. you can only create permanent view on **global managed table** or **global unmanaged table** . Not Allowed to create perment view on temporary view or dataframe.

- Note Permanent view available in SQL API - Not available in Dataframe API.
- There *isn't* a function like **`dataframe.createOrReplacePermanentView()`**

```
spark.sql("CREATE VIEW permanent_view AS SELECT * FROM table")
```

[Spark DG 15-19](#)

 [Spark memoy architecture](#)

 [Apache Spark Associate Developer](#)

[LakeHouse Certification](#)