1.
When using Spark from Python or R, you don’t write explicit JVM instructions; instead, you
write Python and R code that Spark translates into code that it then can run on the executor
JVMs

2.
The SparkSession instance is the way Spark executes
user-defined manipulations across the cluster

```
val spark = SparkSession
      .builder()
      .appName("Spark DF example")
      .config("spark.master", "local")
      .getOrCreate()
```

3.
A DataFrame is the most common Structured API and simply represents a table of data with
rows and columns
it’s quite easy to convert Pandas (Python) DataFrames to Spark DataFrames

4.
Spark will not act on transformations until we call an action

Transformations consisting of narrow dependencies ->only one partition
contributes to at most one output partition

A wide dependency (or wide transformation) style transformation will have input partitions
contributing to many output partitions

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

5.Lazy evaulation means that Spark will wait until the very last moment to execute
Spark compiles this plan from
your raw DataFrame transformations to a streamlined physical plan that will run as efficiently as
possible across the cluster

6.
```
val df = spark.read.format("csv").option("header","true").option("inferSchema","true")
          .option("path","home/csv").option("mode","failfast").load()
```

By default, when we perform a shuffle, Spark
outputs 200 shuffle partitions. Let’s set this value to 5 to reduce the number of the output
partitions from the shuffle
```
spark.conf.set("spark.sql.shuffle.partitions", "5")
flightData2015.sort("count").take(2)
```

7. use sql and df are same

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
flightData2015.select(max("count")).take(1)

val maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
```

```
""")

flightData2015
.groupBy("DEST_COUNTRY_NAME")
.sum("count")
.withColumnRenamed("sum(count)", "destination_total")
.sort(desc("destination_total"))
.limit(5)
.show()


8 create df
val someData = Seq(
  Row(8, "bat"),
  Row(64, "mouse"),
  Row(-27, "horse")
)

val someSchema = List(
  StructField("number", IntegerType, true),
  StructField("word", StringType, true)
)

val someDF = spark.createDataFrame(
  spark.sparkContext.parallelize(someData),
  StructType(someSchema)
)


2)toDF()
import spark.implicits._
val someDF = Seq(
  (8, "bat"),
  (64, "mouse"),
  (-27, "horse")
).toDF("number", "word")

9. Columns represent a simple type like an integer or string, a complex type like an array or map

10. process of execution
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.

11.
We can either let a data source
define the schema (called schema-on-read) or we can define it explicitly ourselves

12. StructType made up of a number of fields, StructFields
StructType(StructField(DEST_COUNTRY_NAME,StringType,true),
StructField(ORIGIN_COUNTRY_NAME,StringType,true),
StructField(count,LongType,true))

If the types in the data (at runtime) do not match
the schema, Spark will throw an error.

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")
```

13,
There are a lot of different ways to construct and refer to columns but the two simplest ways are
by using the col or column functions.

```
col("someColumnName")
column("someColumnName")
scala: $"myColumn"
```

```
expr("someCol") is equivalent to col("someCol").
expr("someCol - 5") => col("someCol") - 5 => expr("someCol") - 5
expr("(((someCol + 5) * 200) - 6) < otherCol")
```

14 Creating Rows

```
val myRow = Row("Hello", null, 1, false)
//access rows
myRow(0) // type Any
myRow(0).asInstanceOf[String] // String
myRow.getString(0) // String
myRow.getInt(2) // Int
```

15 DataFrame Transformations
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

16 Creating DataFrames

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

We can also create DataFrames on the fly by taking a set of rows and converting them to a
DataFrame

```
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()

val myDF = Seq(("Hello", 2, 1L)).toDF("col1", "col2", "col3")
```

17
selectExpr method when you’re working with expressions in strings

You can select multiple columns by using the same style of query

```
df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)
```

18 you can refer to columns in a number of different ways

```
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)

//but don't mix, this iwll cause error
df.select(col("DEST_COUNTRY_NAME"), "DEST_COUNTRY_NAME")
```

19 rename column names
```
df.select(expr("DEST_COUNTRY_NAME AS destination")).show(2)

//The preceding operation changes the column name back to its original name.
df.select(expr("DEST_COUNTRY_NAME as destination").alias("DEST_COUNTRY_NAME"))
.show(2)
```

20.Because select followed by a series of expr is such a common pattern, Spark has a shorthand
for doing this efficiently: selectExpr

```
df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME").show(2)
```

adds a new column withinCountry to our DataFrame that specifies whether the destination and
origin are the same
```
df.selectExpr(
"*", // include all original columns
"(DEST_COUNTRY_NAME = ORIGIN_COUNTRY_NAME) as withinCountry")
.show(2)
```

21
we can also specify aggregations over the entire DataFrame by taking
advantage
```
df.selectExpr("avg(count)", "count(distinct(DEST_COUNTRY_NAME))").show(2)
```

22.
pass explicit values into Spark that are just a value (rather than a new
column)
```
df.select(expr("*"), lit(1).as("One")).show(2)
```

23. Adding Columns
```
//add numberOne
df.withColumn("numberOne", lit(1)).show(2)

//or by expr
df.withColumn("withinCountry", expr("ORIGIN_COUNTRY_NAME == DEST_COUNTRY_NAME"))
.show(2)
df.withColumn("Destination", expr("DEST_COUNTRY_NAME")).columns
```

24. Renaming Columns
This will rename the column with the name of the string in
the first argument to the string in the second argument:
```
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns
```

25. characters like spaces or dashes in column names
```
// column name is 'this long column-name'
val dfWithLongColName = df.withColumn(
"This Long Column-Name",
expr("ORIGIN_COUNTRY_NAME"))

//In this example, however, we need to use backticks because we’re
referencing a column in an expression:
dfWithLongColName.selectExpr(
"`This Long Column-Name`",
"`This Long Column-Name` as `new col`")
.show(2)
```

27 Case Sensitivity
By default Spark is case insensitive
you can make Spark case sensitive by setting: set spark.sql.caseSensitive true

28
Removing Columns
```
df.drop("ORIGIN_COUNTRY_NAME").columns
```

29 Changing a Column’s Type (cast)
convert our count column from
an integer to a type Long: df.withColumn("count2", col("count").cast("long"))

30.filters

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

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)
```

31Getting Unique Rows

```
df.select("ORIGIN_COUNTRY_NAME", "DEST_COUNTRY_NAME").distinct().count()
```

32 sample
总数 256，

```
val seed = 5
val withReplacement = false
val fraction = 0.5
df.sample(withReplacement, fraction, seed).count()
```

output of 126.

33 Random Splits

```
//we’ll split our DataFrame into two different
DataFrames by setting the weights by which we will split the DataFrame
// in Scala
val dataFrames = df.randomSplit(Array(0.25, 0.75), seed)
dataFrames(0).count() > dataFrames(1).count() // False
```

34 union
To append to a DataFrame, you must
union the original DataFrame along with the new DataFrame.
To union two DataFrames, you must be sure that they have the same schema and
number of columns; otherwise, the union will fail.

```
df.union(newDF)
.where("count = 1")
.where($"ORIGIN_COUNTRY_NAME" =!= "United States")
.show() // get all of them and we'll see our new rows at the end
```

you must use the =!= operator so that you don’t just compare the unevaluated column

35
Sorting Rows

There are two equivalent operations to do this sort
and orderBy that work the exact same way

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

```
#use the asc and desc functions
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

36 repartition
it can be worth

repartitioning based on that column

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

Coalesce, on the other hand, will not incur a full shuffle and will try to combine partitions

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

37

collect some of your data to the driver in order to manipulate

```
collectDF.collect()
```

The method toLocalIterator collects partitions to the driver as an iterator

```
collectDF.toLocalIterator()
```

38

the lit function. This function converts a
type in another language to its correspnding Spark representation

```
df.select(lit(5), lit("five"), lit(5.0))
```

#

39 Working with Booleans

```
df.where(col("InvoiceNo").equalTo(536365))
.select("InvoiceNo", "Description")
.show(5, false)

df.where(col("InvoiceNo") === 536365)
.select("InvoiceNo", "Description")
.show(5, false)

df.where("InvoiceNo = 536365")
.show(5, false)

val priceFilter = col("UnitPrice") > 600
val descripFilter = col("Description").contains("POSTAGE")
df.where(col("StockCode").isin("DOT")).where(priceFilter.or(descripFilter))
.show()

val DOTCodeFilter = col("StockCode") === "DOT"
val priceFilter = col("UnitPrice") > 600
val descripFilter = col("Description").contains("POSTAGE")
df.withColumn("isExpensive", DOTCodeFilter.and(priceFilter.or(descripFilter)))
.where("isExpensive")
.select("unitPrice", "isExpensive").show(5)

df.withColumn("isExpensive", expr("NOT UnitPrice <= 250"))
.filter("isExpensive")
.select("Description", "UnitPrice").show(5)

df.withColumn("isExpensive", not(col("UnitPrice").leq(250)))
.filter("isExpensive")
.select("Description", "UnitPrice").show(5)
```

39 Working with Numbers

```
val fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(expr("CustomerId"), fabricatedQuantity.alias("realQuantity")).show(2)

=> df.selectExpr(
"CustomerId",
"(POWER((Quantity * UnitPrice), 2.0) + 5) as realQuantity").show(2)
```

we round to one decimal place:

```
df.select(round(col("UnitPrice"), 1).alias("rounded"), col("UnitPrice")).show(5)
```

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)

#compute the correlation of two columns
df.stat.corr("Quantity", "UnitPrice")
#compute summary statistics for a column or set of columns
df.describe().show()

add a unique ID to each row by using the function
monotonically_increasing_id.
df.select(monotonically_increasing_id()).show(2)


40. Working with Strings
#The initcap function will
capitalize every word in a given string
df.select(initcap(col("Description"))).show(2, false)

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

#d
so that length of this column should be 4 characters.
If length is less than 4 characters, then add 0's
import org.apache.spark.sql.functions.{lit, ltrim, rtrim, rpad, lpad, trim}
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)


41

DESC is "WHITE HANGING HEA...|"
1)
regular replace()
regex_string = "BLACK|WHITE|RED|GREEN|BLUE"
df.select(
regexp_replace(col("Description"), regex_string, "COLOR").alias("color_clean"),
col("Description")).show(2)


+--------------------+--------------------+
| color_clean| Description|
+--------------------+--------------------+
|COLOR HANGING HEA...|WHITE HANGING HEA...|
| COLOR METAL LANTERN| WHITE METAL LANTERN|
+--------------------+--------------------
2)
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

val simpleColors = Seq("black", "white", "red", "green", "blue")
val regexString = simpleColors.map(_.toUpperCase).mkString("|")
// the
// 'L' replaced to 1, 'E' replaced to 3.
df.select(translate(col("Description"), "LEET", "1337"), col("Description"))
.show(2)


3)
pulling out the first mentioned color
```

```
# get the 'WHITE"
extract_str = "(BLACK|WHITE|RED|GREEN|BLUE)"
df.select(
regexp_extract(col("Description"), regexString, 1).alias("color_clean"),
col("Description")).show(2)
+-------------+-------------------+
| color_clean| Description|
+-------------+-------------------+
| WHITE|WHITE HANGING HEA...|
| WHITE| WHITE METAL LANTERN|
+-------------+-------------------+

4) check for their existence
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)


42 Date and timestamp

dateDF = spark.range(10)\
.withColumn("today", current_date())\
.withColumn("now", current_timestamp())
dateDF.createOrReplaceTempView("dateTable")


#add , substract dateDF
dateDF.select(date_sub(col("today"), 5), date_add(col("today"), 5)).show(1)

#datediff function that will return the number of days in between two dates
dateDF.withColumn("week_ago", date_sub(col("today"), 7))
.select(datediff(col("week_ago"), col("today"))).show(1)

dateDF.select(
to_date(lit("2016-01-01")).alias("start"),
to_date(lit("2017-05-22")).alias("end"))
.select(months_between(col("start"), col("end"))).show(1)

#The to_date function allows
you to convert a string to a date
spark.range(5).withColumn("date", lit("2017-01-01"))
.select(to_date(col("date"))).show(1)

#use format
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")

#timestamp
dateFormat = "yyyy-dd-MM"
to_timestamp, which always requires a format to be specified
//use columne date defined in previous eample,
cleanDateDF.select(to_timestamp(col("date"), dateFormat)).show()

cleanDateDF.filter(col("date2") > lit("2017-12-12")).show()

Handle null data.
#Coalesce
first non-null value from a set of columns by
using the coalesce function
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

43.
#
ifnull allows you to select the second value if the first is null

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.

nvl2 returns
the second value if the first is not null; otherwise, it will return the last specified value

```
SELECT
ifnull(null, 'return_value'),
nullif('value', 'value'),
nvl(null, 'return_value'),
nvl2('not_null', 'return_value', "else_value")
FROM dfTable LIMIT 1

+------------+----+-----------+-----------+
| a| b| c| d|
+------------+----+-----------+-----------+
|return_value|null|return_value|return_value|
+------------+----+-----------+-----------+
```

Using “all” drops the
row only if all values are null or NaN for that row
```
df.na.drop("all")
```

The default is to drop anyrow in which any value is null:
```
df.na.drop()
df.na.drop("any")
```

We can also apply this to certain sets of columns by passing in an array of columns:
```
//drop, 'stockCode' 和invoiceNo 都是null的列
df.na.drop("all", Seq("StockCode", "InvoiceNo"))
```

fill all null values in columns of type String
```
df.na.fill("All Null values become this string")
```

```
//把列类型Int的为空的， fill 5
or df.na.fill(5:Integer)

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:
```
// in Scala
val fillColValues = Map("StockCode" -> 5, "Description" -> "No Value")
df.na.fill(fillColValues)
```

replace all values in a certain column according to their current value
```
df.na.replace("Description", Map("" -> "UNKNOWN"))
```

44. Ordering
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

45
Working with Complex Types

```
1) You can think of 'structs as DataFrames within DataFrames
//把两个列合成一个struct,
val complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")
complexDF.select("complex.Description")
We can also query all values in the struct by using *.
complexDF.select("complex.*")

2) array
df.select(split(col("Description"), " ")).show(2)
+---------------------+
|split(Description, )|
+---------------------+
| [WHITE, HANGING, ...|
| [WHITE, METAL, LA...|
+---------------------+


Spark allows us to manipulate this complex type as another
column

df.select(split(col("Description"), " ").alias("array_col"))
.selectExpr("array_col[0]").show(2)

This gives us the following result:
+------------+
|array_col[0]|
+------------+
| WHITE|
| WHITE|
+------------+
Array Length
//use size() get length
df.select(size(split(col("Description"), " "))).show(2) // shows 5 and 3

# array contains
df.select(array_contains(split(col("Description"), " "), "WHITE")).show(2) => true

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

#use array in desc, explode two multiple rows.
df.withColumn("splitted", split(col("Description"), " "))
.withColumn("exploded", explode(col("splitted")))
.select("Description", "InvoiceNo", "exploded").show(2)

[WHITE, HANGING] explode后变成两列
+--------------------+---------+--------+
| Description|InvoiceNo|exploded|
+--------------------+---------+--------+
|WHITE HANGING HEA...| 536365| WHITE|
|WHITE HANGING HEA...| 536365| HANGING|
+--------------------+---------+--------+

Maps
// in Scala
import org.apache.spark.sql.functions.map
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map")).show(2)
+--------------------+
| complex_map|
+--------------------+
|Map(WHITE HANGING...|
|Map(WHITE METAL L...|
+--------------------+
//query by key 得到invoiceNo
```

```
df.select(map(col("Description"), col("InvoiceNo")).alias("complex_map"))
.selectExpr("complex_map['WHITE METAL LANTERN']").show(2)
+--------------------------------+
|complex_map[WHITE METAL LANTERN]|
+--------------------------------+
| null|
| 536365|
+--------------------------------+
```

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(2)
+--------------------+------+
| key| value|
+--------------------+------+
|WHITE HANGING HEA...|536365|
| WHITE METAL LANTERN|536365|
+--------------------+------+
```

46 JSON process

get_json_object to inline query a JSON object
json_tuple if this object has only one level of nesting:

```
val jsonDF = spark.range(1).selectExpr("""
'{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as jsonString""")
//get the array inside json
jsonDF.select(
get_json_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]") as "column",
json_tuple(col("jsonString"), "myJSONKey")).show(2)
```

This results in the following table:

```
+------+--------------------+
|column| c0|
+------+--------------------+
| 2|{"myJSONValue":[1...|
+------+--------------------+
```

turn a StructType into a JSON string by using the to_json

```
df.selectExpr("(InvoiceNo, Description) as myStruct")
.select(to_json(col("myStruct")))
#
```

from_json function to parse this (or other JSON data) back in

```
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)
```

47 UFD
Spark will serialize the function on the
driver and transfer it over the network to all executor processes

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.

```
#register
val power3udf = udf(power3(_:Double):Double)
udfExampleDF.select(power3udf(col("num"))).show()
```

```
#another wasy

spark.udf.register("power3", power3(_:Double):Double)
udfExampleDF.selectExpr("power3(num)").show(2)

return type in the following function to be a DoubleType:
spark.udf.register("power3py", power3, DoubleType())

48 Join

1) count
df.select(count("StockCode")).show() // 541909

df.select(countDistinct("StockCode")).show() // 4070
df.select(approx_count_distinct("StockCode", 0.1)).show() // 3364

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

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

df.select(sum("Quantity")).show() // 5176450

df.select(sumDistinct("Quantity")).show() // 29310

df.select(
count("Quantity").alias("total_transactions"),
sum("Quantity").alias("total_purchases"),
avg("Quantity").alias("avg_purchases"),
expr("mean(Quantity)").alias("mean_purchases"))

The variance is the average
of the squared differences from the mean, and the standard deviation is the square root of the
variance
df.select(var_pop("Quantity"), var_samp("Quantity"),
stddev_pop("Quantity"), stddev_samp("Quantity")).show()

Skewness
measures the asymmetry of the values in your data around the mean, whereas kurtosis is a
measure of the tail of data
df.select(skewness("Quantity"), kurtosis("Quantity")).show()


df.select(corr("InvoiceNo", "Quantity"), covar_samp("InvoiceNo", "Quantity"),
covar_pop("InvoiceNo", "Quantity")).show()
#
Aggregating to Complex Types
import org.apache.spark.sql.functions.{collect_set, collect_list}
df.agg(collect_set("Country"), collect_list("Country")).show()


df.groupBy("InvoiceNo", "CustomerId").count().show()
+---------+---------+-----+
|InvoiceNo|CustomerId|count|
+---------+---------+-----+
| 536846| 14573| 76|
...
| C544318| 12989| 1|
+---------+---------+-----+
# Grouping with Expressions
df.groupBy("InvoiceNo").agg(
count("Quantity").alias("quan"),
expr("count(Quantity)")).show()
|InvoiceNo|quan|count(Quantity)|
+---------+----+---------------+
```

```
| 536596| 6| 6|
...
| C542604| 8| 8|
+---------+----+---------------+
```

Grouping with Maps

```
df.groupBy("InvoiceNo").agg("Quantity"->"avg", "Quantity"->"stddev_pop").show()
+---------+------------------+--------------------+
|InvoiceNo| avg(Quantity)|stddev_pop(Quantity)|
+---------+------------------+--------------------+
| 536596| 1.5| 1.1180339887498947|
...
| C542604| -8.0| 15.173990905493518|
+---------+------------------+--------------------+
```

#partitionby is unrelated to the partitioning scheme concept that we have covered thus far.
It’s just a
similar concept that describes how we will be breaking up our group

```
val windowSpec = Window
.partitionBy("CustomerId", "date")
.orderBy(col("Quantity").desc).rowsBetween(Window.unboundedPreceding, Window.currentRow)

#establishing the maximum purchase quantity over all time.
maxPurchaseQuantity = max(col("Quantity")).over(windowSpec)

val purchaseDenseRank = dense_rank().over(windowSpec)
```

Now we can perform a select to
view the calculated window values

```
dfWithDate.where("CustomerId IS NOT NULL").orderBy("CustomerId")
.select(
col("CustomerId"),
col("date"),
col("Quantity"),
purchaseRank.alias("quantityRank"),
purchaseDenseRank.alias("quantityDenseRank"),
maxPurchaseQuantity.alias("maxPurchaseQuantity")).show()
```

you also want to include the total number of items, regardless of
customer or stock code? With a conventional group-by statement, this would be impossible. But,
it’s simple with grouping sets: we simply specify that we would like to aggregate at that level

```
SELECT CustomerId, stockCode, sum(Quantity) FROM dfNoNull
GROUP BY customerId, stockCode GROUPING SETS((customerId, stockCode),())
ORDER BY CustomerId DESC, stockCode DESC
```

49 Rollup

creates a new DataFrame that includes the grand total over all dates, the
grand total for each 'date' in the DataFrame, and the subtotal for each 'country' on each date in the
DataFrame:

```
val rolledUpDF = dfNoNull.rollup("Date", "Country").agg(sum("Quantity"))
.selectExpr("Date", "Country", "`sum(Quantity)` as total_quantity")
.orderBy("Date")
rolledUpDF.show()

+----------+-------------+-------------+
| Date| Country|total_quantity|
+----------+-------------+-------------+
```

```
| null| null| 5176450|
|2010-12-01|United Kingdom| 23949|
|2010-12-01| Germany| 117|
|2010-12-01| France| 449|
...
|2010-12-03| France| 239|
|2010-12-03| Italy| 164|
|2010-12-03| Belgium| 528|
+----------+-------------+-------------+
```

Cube
across all dimensions
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

```
dfNoNull.cube("Date", "Country").agg(sum(col("Quantity")))
.select("Date", "Country", "sum(Quantity)").orderBy("Date").show()
```

grouping_id,
which gives us a column specifying the level of aggregation that we have in our result set

```
dfNoNull.cube("customerId", "stockCode").agg(grouping_id(), sum("Quantity"))
.orderBy(expr("grouping_id()").desc)
.show()
```

51.
User-Defined Aggregation Functions
52. join

```
val joinExpression = person.col("graduate_program") === graduateProgram.col("id")
person.join(graduateProgram, joinExpression).show()
```

or

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

53 Outer Joins
If there is no equivalent row in either the left or
right DataFrame, Spark will insert null:

```
joinType = "outer"
person.join(graduateProgram, joinExpression, joinType).show()
```

54
Left Outer Joins
includes all rows from
the left DataFrame If there is no equivalent row in the right DataFrame, Spark will insert null

```
joinType = "left_outer"
graduateProgram.join(person, joinExpression, joinType).show()
```

```
SELECT * FROM graduateProgram LEFT OUTER JOIN person
ON person.graduate_program = graduateProgram.id
```

55.

```
joinType = "right_outer"
person.join(graduateProgram, joinExpression, joinType).show()
```

56 left-semi
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

```
joinType = "left_semi"
graduateProgram.join(person, joinExpression, joinType).show()
```

57 Left Anti Joins

```
joinType = "left_anti"
```

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

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

58

```
person.crossJoin(graduateProgram).show()
joinType = "cross"
graduateProgram.join(person, joinExpression, joinType).show()
```

59

```
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.;
```

Joins on Complex Types

```
person.withColumnRenamed("id", "personId")
.join(sparkStatus, expr("array_contains(spark_status, id)")).show()
```

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

Approach 2: Dropping the column after the join

```
person.join(gradProgramDupe, joinExpr).drop(person.col("graduate_program"))
.select("graduate_program").show()
```

Approach 3: Renaming a column 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()
```

Big table–to–small table

we can use a big table–to–big table communication strategy, it can often be more efficient to use a broadcast join

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.

```
import org.apache.spark.sql.functions.broadcast
val joinExpr = person.col("graduate_program") === graduateProgram.col("id")
person.join(broadcast(graduateProgram), joinExpr).explain()
```

```
val dbDataFrame = spark.read.format("jdbc")
.option("url", url).option("dbtable", tablename).option("driver", driver)
.option("numPartitions", 10).load()
```

Partitioning based on a sliding window

```
// in Scala
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() // 255


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

csvFile.limit(10).select("DEST_COUNTRY_NAME", "count")
.write.partitionBy("count").text("/tmp/five-csv-files2.csv")

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

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

CSV files do not support complex types, whereas Parquet and ORC do.

df.write.option("maxRecordsPerFile", 5000), Spark
will ensure that files will contain at most 5,000 records.

SparkSQL Thrift JDBC/ODBC Server
Spark provides a Java Database Connectivity (JDBC) interface by which either you or a remote
program connects to the Spark driver in order to execute Spark SQL queries
./sbin/start-thriftserver.sh

You can then test this connection by running the following commands:
beeline> !connect jdbc:hive2://localhost:10000

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

you can also see tables in a specific
database by using the query show tables IN databaseName

Creating Tables

```
CREATE TABLE flights (
DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count LONG)
USING JSON OPTIONS (path '/data/flight-data/json/2015-summary.json')
```

It is possible to create a table from a query as well:

```
CREATE TABLE flights_from_select USING parquet AS SELECT * FROM flights
```


we create an unmanaged table. Spark will
manage the table’s metadata; however, the files are not managed by Spark at all. You create this
table by using the CREATE EXTERNAL TABLE statement

```
CREATE EXTERNAL TABLE hive_flights (
DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count LONG)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION '/data/flight-data-hive/'


INSERT INTO partitioned_flights
PARTITION (DEST_COUNTRY_NAME="UNITED STATES")
```

```
SELECT count, ORIGIN_COUNTRY_NAME FROM flights
WHERE DEST_COUNTRY_NAME='UNITED STATES' LIMIT 12
```

You can also create an external table from a select clause:

```
CREATE EXTERNAL TABLE hive_flights_2
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/data/flight-data-hive/' AS SELECT * FROM flights
```

65. Describing Table Metadata

```
DESCRIBE TABLE flights_csv
SHOW PARTITIONS partitioned_flights
```

66.Refreshing Table Metadata

```
REFRESH table partitioned_flights
MSCK REPAIR TABLE partitioned_flights
```

67

```
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

68. View

A view specifies a set of transformations on top of an existing table—basically just saved query plans

```
CREATE VIEW just_usa_view AS
SELECT * FROM flights WHERE dest_country_name = 'United States'
```

you can create temporary views that are available only during the current session

```
CREATE TEMP VIEW just_usa_view_temp AS
SELECT * FROM flights WHERE dest_country_name = 'United States'
```

or CREATE OR REPLACE TEMP VIEW

views are equivalent to creating a new DataFrame from an existing DataFrame

```
val flights = spark.read.format("json")
.load("/data/flight-data/json/2015-summary.json")
val just_usa_df = flights.where("dest_country_name = 'United States'")
just_usa_df.selectExpr("*").explain


DROP VIEW IF EXISTS just_usa_view;
```

69 database

```
CREATE DATABASE some_db
USE some_db
SHOW tables
SELECT current_database()
DROP DATABASE IF EXISTS some_db;
```

70 select

```
SELECT
CASE WHEN DEST_COUNTRY_NAME = 'UNITED STATES' THEN 1
WHEN DEST_COUNTRY_NAME = 'Egypt' THEN 0
ELSE -1 END
FROM partitioned_flights
```

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'
```

71 complex type
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
SELECT country.DEST_COUNTRY_NAME, count FROM nested_data
```

List
```
SELECT DEST_COUNTRY_NAME as new_name, collect_list(count) as flight_counts,
collect_set(ORIGIN_COUNTRY_NAME) as origin_set
FROM flights GROUP BY DEST_COUNTRY_NAME
```

sub query
```
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)
```

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)
```

spark.sql.shuffle.partitions 200 (default)

71 cluster mode
The cluster
manager then launches the driver process on a worker node inside the cluster, in addition to the
executor processes

cliet mode
Spark driver remains on the client
machine that submitted the application. This means that the client machine is responsible for
maintaining the Spark driver process, and the cluster manager maintains the executor processses

Local mode
it runs the entire Spark
Application on a single machine

72 The Life Cycle of a Spark Application
1) Client Request
```
./bin/spark-submit \
--class <main-class> \
--master <master-url> \
--deploy-mode cluster \
--conf <key>=<value> \
... # other options
<application-jar> \
[application-arguments]
```

2) Launch
The SparkSession will subsequently communicate with the cluster manager

(the darker line), asking it to launch Spark executor processes across the cluster
3)Execution
The driver and the workers communicate among themselves, executing code and
moving data around. The driver schedules tasks onto each worker, and each worker responds
with the status of those tasks and success or failure
4) After a Spark Application completes, the driver processs exits with either success or failure

73
The Life Cycle of a Spark Application (Inside Spark)

1) session
```
val spark = SparkSession.builder().appName("Databricks Spark Example")
.config("spark.sql.warehouse.dir", "/user/hive/warehouse")
.getOrCreate()
```

SparkContext, you can create RDDs, accumulators, and broadcast variables
```
import org.apache.spark.SparkContext
val sc = SparkContext.getOrCreate()
```

It is important to note that you should never need to use the SQLContext and
rarely need to use the SparkContext

2) A Spark Job
In general, there should be one Spark job for one action. Actions always return results. Each job
breaks down into a series of stages, the number of which depends on how many shuffle
operations need to take place

stages
In general, Spark will try to pack as much work as possible (i.e.,
as many transformations as possible inside your job) into the same stage, but the engine starts
new stages after operations called shuffles
c
A shuffle represents a physical repartitioning of the
data—for example, sorting a DataFrame, or grouping data that was loaded from a file

Spark starts a new stage after each
shuffle

The
spark.sql.shuffle.partitions default value is 200, which means that when there is a shuffle
performed during execution, it outputs 200 shuffle partitions by default

entire stage is computed in parallel. The final result aggregates those
partitions individually, brings them all to a single partition before finally sending the final result
to the driver

Task:
A task is just a unit of computation applied to a unit of data (the
partition)

74 shuffle
Spark always executes
shuffles by first having the “source” tasks (those sending data) write shuffle files to their local
disks during their execution stage. Then, the stage that does the grouping and reduction launches
and runs tasks that fetch their corresponding records from each shuffle file and performs that
computation

75 spark-submit
--packages Comma-separated list of Maven coordinates of JARs to include on the driver and executor
classpaths
--drivermemory
--executormemory
MEM
--driverjava-
options

```
./bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master spark://207.184.161.138:7077 \
--executor-memory 20G \
--total-executor-cores 100 \
replace/with/path/to/examples.jar \
1000
```

76
The SparkConf

```
val conf = new SparkConf().setMaster("local[2]").setAppName("DefinitiveGuide")
.set("some.conf", "to.some.value")
```

```
./bin/spark-submit --name "DefinitiveGuide" --master local[4] ...
```

77.
Application Properties
spark.driver.cores 1 Number of cores to use for the driver process, only in cluster mode.
spark.driver.memory 1g
spark.executor.memory 1g Amount of memory to use per executor process (e.g., 2g, 8g).
spark.submit.deployMode (none) client/cluster

spark.files.maxPartitionBytes (maximum
partition size when reading files)

78Configuring Memory Management

79 scheduler
By default, Spark’s scheduler runs jobs in FIFO fashion
Under fair sharing, Spark assigns tasks
between jobs in a round-robin fashion so that all jobs get a roughly equal share of cluster
resources spark.scheduler.mode property to FAIR

80 cluster
onpremises
deployment, your cluster is fixed in size
on-premises, the best way to combat the resource utilization problem
is to use a cluster manager that allows you to run many Spark applications and dynamically
reassign resources between them

a better idea to use global storage systems that are decoupled
from a specific cluster, such as Amazon S3, Azure Blob Storage, or Google Cloud Storage and
spin up machines dynamically for each Spark workload

Standalone Mode
disadvantage of the standalone mode is that it’s more limited than the other cluster managers—in
particular, your cluster can only run Spark

The first step is to start the master process on the machine

```
$SPARK_HOME/sbin/start-master.sh
```

Once started, the master prints out a spark://HOST:PORT URI

With that URI, start the worker nodes by
logging in to each machine and running the following script using the URI you just received
from the master node

```
$SPARK_HOME/sbin/start-slave.sh <master-spark-URI>
```

81
Hadoop configurations
If you plan to read and write from HDFS using Spark, you need to include two Hadoop
configuration files on Spark’s classpath: hdfs-site.xml, which provides default behaviors for the
HDFS client; and core-site.xml

To make these files visible to Spark, set HADOOP_CONF_DIR in $SPARK_HOME/spark-env.sh to a
location containing the configuration files or as an environment variable

82
Mesos is the heaviest-weight cluster manager, simply because you might
choose this cluster manager only if your organization already has a large-scale deployment of
Mesos

Coarse-grained mode means that each Spark executor runs as a single
Mesos task

```
val spark = SparkSession.builder
.master("mesos://HOST:5050")
.appName("my app")
.config("spark.executor.uri", "<path to spark-2.2.0.tar.gz uploaded above>")
.getOrCreate()
```

Dynamic allocation
dynamic allocation (described next) can be turned on to let applications
scale up and down dynamically based on their current number of pending tasks
set spark.dynamicAllocation.enabled to true,
default is disableed.

second:
spark.shuffle.service.enabled to true in your application. The purpose of
the external shuffle service is to allow executors to be removed without deleting shuffle files
written by them

83,
YARN is great for HDFS-based applications but is not commonly used for
much else. Additionally, it’s not well designed to support the cloud

Spark standalone mode is the lightest-weight cluster
manager and is relatively simple to understand and take advantage of, but then you’re going to
be building more application management infrastructure

it doesn’t really make sense to have a Mesos cluster
for only running Spark Applications

84
The metrics system is
configured via a configuration file that Spark expects to be present at
$SPARK_HOME/conf/metrics.properties
These metrics can be output to a variety of
different sinks, including cluster monitoring solutions like Ganglia

spark.sparkContext.setLogLevel("INFO")
The logs themselves will be printed to standard error when running a local mode application, or saved to files by your cluster manager when running Spark on a cluster.
Refer to each cluster manager’s documentation about how to find them—typically, they are available through the cluster manager’s web UI

86 spark GUI

If you’re running multiple applications, they will launch web UIs on increasing port numbers (4041, 4042, …)

87 Error toubleshoting

Ensure that machines can communicate with one another
Ensure that your Spark resource configurations are correct and that your cluster manager is properly set up for Spark
Double-check to verify that the cluster has the network connectivity
There might be issues with libraries or classpaths that are causing the wrong version of a library to be loaded for accessing storage

88
Potential treatments
Check to see if your data exists or is in the format that you expect

If an error quickly pops up when you run a query (i.e., before tasks are launched), it is most likely an analysis error while planning the query

It’s also possible that your own code for processing the data is crashing, in which case Spark will show you the exception thrown by your code. In this case, you will see a task marked as “failed” on the Spark UI,

89 Slow Tasks or Stragglers
symptoms:
Spark stages seem to execute until there are only a handful of tasks left. Those tasks then take a long time.
These slow tasks show up in the Spark UI and occur consistently on the same dataset(s).
These occur in stages, one after the other.
Scaling up the number of machines given to the Spark Application doesn’t really help—some tasks still take much longer than others.
In the Spark metrics, certain executors are reading and writing much more data than others

solution:
Try increasing the number of partitions to have less data per partition
Try repartitioning by another combination of columns
Check whether your user-defined functions (UDFs) are wasteful in their object allocation or business logic. Try to convert them to DataFrame code if possible
Try increasing the memory allocated to your executors if possible
Monitor the executor that is having trouble
Another common issue can arise when you’re working with Datasets. Because Datasets perform a lot of object instantiation to convert records to Java objects for UDFs, they can cause a lot of garbage collection

90 Slow Aggregations
Slow tasks during a groupBy call. Jobs after the aggregation are slow, as well

treatments:
Increasing the number of partitions, prior to an aggregation
Increasing executor memory can help alleviate this issue, as well. If a single key has lots

of data, this will allow its executor to spill to disk less often and finish faster,

If you find that tasks after the aggregation are also slow, this means that your dataset might have remained unbalanced after the aggregation. Try inserting a repartition call to partition it randomly

Ensuring that all filters and SELECT statements that can be are above the aggregation can help to ensure that you’re working only on the data that you need to be working
Ensure null values are represented correctly (using Spark’s concept of null) and not as some default value like " " or "EMPTY". Spark often optimizes for skipping nulls

For instance,
collect_list and collect_set are very slow aggregation functions because they must return all the matching objects to the driver

91
Slow Joins
A join stage seems to be taking a long time , Stages before and after the join seem to be operating normally

treatments:
Many joins can be optimized

Experimenting with different join orderings can really help speed up jobs, especially if some of those joins filter out a large amount of data; do those first.

Partitioning a dataset prior to joining can be very helpful for reducing data movement across the cluster, especially if the same dataset will be used in multiple join operations.
It’s worth experimenting with different prejoin partitioning. Keep in mind, again, that this isn’t “free” and does come at the cost of a shuffle.

Ensuring that all filters and select statements that can be are above the join

Ensure that null values are handled correctly

Sometimes Spark can’t properly plan for a broadcast join if it doesn’t know any statistics about the input DataFrame or table. If you know that one of the tables that you are joining is small, you can try to force a broadcast

92 Slow Reads and Writes
Slow reading of data from a distributed file system

treatments:
Turning on speculation (set spark.speculation to true) can help with slow reads and writes. This will launch additional tasks with the same operation in an attempt to see whether it’s just some transient issue in the first task

Ensuring sufficient network connectivity can be important
For distributed file systems such as HDFS running on the same nodes as Spark, make sure Spark sees the same hostnames for nodes as the file system

93 Driver OutOfMemoryError or Driver Unresponsive
Signs and symptoms
Spark Application is unresponsive or crashed.
OutOfMemoryErrors or garbage collection messages in the driver logs.
Commands take a very long time to run or don’t run at all.
Interactivity is very low or non-existent.
Memory usage is high for the driver JVM.

Potential treatments
Your code might have tried to collect an overly large dataset to the driver node using operations such as collect
You might be using a broadcast join where the data to be broadcast is too big

A long-running application generated a large number of objects on the driver and is unable to release them. Java’s jmap tool can be useful to see what objects are filling most of the memory

Issues with JVMs running out of memory can happen if you are using another language binding, such as Python, due to data conversion between the two requiring too much memory in the JVM

94 Executor OutOfMemoryError or Executor Unresponsive
OutOfMemoryErrors or garbage collection messages in the executor logs. You can find these in the Spark UI.
Executors that crash or become unresponsive.
Slow tasks on certain nodes that never seem to recover

Try increasing the memory available to executors and the number of executors.
Try increasing PySpark worker size via the relevant Python configurations.
Look for garbage collection error messages in the executor logs. Some of the tasks that are running, especially if you’re using UDFs, can be creating lots of objects that need to be garbage collected.

95
No Space Left on Disk Errors
If you have a cluster with limited storage space, some nodes may run out first due to skew. Repartitioning the data as described earlier may help here.

Some of these determine how long logs should be kept on the machine

Serialization Errors
Try not to refer to any fields of the enclosing object in your UDFs when creating UDFs inside a Java or Scala class. This can cause Spark to try to serialize the whole enclosing object, which may not be possible