Développement d'applications Big Data Spark

EDAH Kodjo
Consultant Systèmes d'Information, Big-Data

Découvrir spark : vue globale

- Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters
- Python, Java, Scala, and R
- Run everywhere (one computer to thousand of machine on a cluster)
- makes it an easy system to start with and scale-up to big data processing or incredibly large scale.

Découvrir spark : unified

- Spark is designed to support a wide range of data analytics tasks, ranging from simple data loading and SQL queries to machine learning and streaming computation, over the same computing engine and with a consistent set of APIs

Découvrir spark : Computing engine

- At the same time that Spark strives for unification, it carefully limits its scope to a computing engine.
- Spark handles loading data from storage systems and performing computation on it, not permanent storage as the end itself
- Azure Storage and Amazon S3, distributed file systems such as Apache Hadoop, key-value stores such as Apache Cassandra, and message buses such as Apache Kafka

Découvrir spark : Librairies

- Spark includes libraries for SQL and structured data (Spark SQL), machine learning (MLlib),
- stream processing (Spark Streaming and the newer Structured Streaming), and graph analytics (GraphX).
- Beyond these libraries, there are hundreds of open source external libraries ranging from connectors for various storage systems to machine learning algorithms.
- One index of external libraries is available at spark-packages.org

Découvrir spark : Pourquoi spark

- The Big Data Problem
- Parallel processing
- MapReduce drawbacks
- Functional programming

Découvrir spark : Pourquoi spark

- Python, Java, Scala, R, or SQL.

- Runs on the Java Virtual Machine (JVM) : all you need is an installation of Java
- downloading and installing Apache Spark on your laptop, or running a web-based version in Databricks Community Edition, a free cloud environment for learning Spark that includes the code in this book

Downloading spark

- https://spark.apache.org/downloads.html

- pip install pyspark

```
cd ~/Downloads
tar -xf spark-2.2.0-bin-hadoop2.7.tgz
cd spark-2.2.0-bin-hadoop2.7.tgz
```

Using conda

- https://spark.apache.org/docs/latest/api/python/getting_started/install.html#using-conda

Lancer spark

- Console python ./bin/pyspark
- Scala console ./bin/spark-shell
- SQL console : /bin/spark-sql

Basic Architecture

- Execute over one machine => performance issue
- Execute over cluster => Need organization
- A cluster, or group, of computers, pools the resources of many machines together, giving us the ability to use all the cumulative resources as if they were a single computer. Now, a group of machines alone is not powerful, you need a framework to coordinate work across them. Spark does just that, managing and coordinating the execution of tasks on data across a cluster of computers.
- Tasks is managed by a cluster manager like
- Spark’s standalone cluster manager, YARN, or Mesos

Spark Applications

- The executors are responsible for actually carrying out the work that the driver assigns them.
- This means that each executor is responsible for only two things: executing code assigned to it by the driver, and reporting the state of the computation on that executor back to the driver node

Spark Applications

Spark Applications

- Spark employs a cluster manager that keeps track of the resources available.
- The driver process is responsible for executing the driver program's commands across the executors to complete a given task.

Spark’s Language APIs

- Scala : Spark is primarily written in Scala, making it Spark’s “default” language.
- Java Even though Spark is written in Scala, Spark’s authors have been careful to ensure that you can write Spark code in Java.
- Python Python supports nearly all constructs that Scala supports.
- SQL Spark supports a subset of the ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to take advantage of the big data powers of Spark.
- R Spark has two commonly used R libraries: Spark core (SparkR) and R community-driven package (sparklyr).

Starting Spark

- ❑ Install Python
- ❑ ./bin/pyspark

- ❑ Interactive Spark Application is started

- ❑ Implicitly create a SparkSession that manages the Spark Application

- ❑ Install Python
- ❑ ./bin/spark-submit

- ❑ Submit a precompiled application to Spark
- ❑ Must create the SparkSession object yourself in your application code

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.3.1
      /_/

Using Python version 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020 22:37:02)
Spark context Web UI available at http://DESKTOP-D3KPQ5O:4040
Spark context available as 'sc' (master = local[*], app id = local-1673871967027)
.
SparkSession available as 'spark'.
>>> 23/01/16 13:26:20 WARN ProcfsMetricsGetter: Exception when trying to compute
pagesize, as a result reporting of ProcessTree metrics is stopped

>>>
```

SparkSession

- Control your Spark Application through a driver process called the SparkSession
- SparkSession instance is the way Spark executes user-defined manipulations across the cluster
- one-to-one correspondence between a SparkSession and a Spark Application.
- In Scala and Python, the variable is available as spark when you start the console.

```
>>> spark
<pyspark.sql.session.SparkSession object at 0x000001E3495A6A30>
>>>
```

SparkSession

- Let's now perform the simple task of creating a range of numbers. This range of numbers is just like a named column in a spreadsheet:

- We created a DataFrame with one column containing 1,000
- rows with values from 0 to 999

```
# First spark code
myRange = spark.range(1000).toDF("number")
myRange.show()
```

```
+------+
|number|
+------+
|     0|
|     1|
|     2|
|     3|
|     4|
|     5|
```

DataFrames

- A DataFrame is the most common Structured API and simply represents a table of data with rows and columns.
- DataFrame as a spreadsheet with named columns.
- The fundamental difference is that while a spreadsheet sits on one computer in one specific location, a Spark DataFrame can span thousands of computers
- Why ? data is too large to fit on one machine or it would simply take too long to perform that computation on one machine
- Spark has several core abstractions: Datasets, DataFrames, SQL Tables, and Resilient Distributed Datasets (RDDs). The easiest and most efficient are DataFrames, which are available in all languages

Partitions

- To allow every executor to perform work in parallel, Spark breaks up the data into chunks called partitions.
- A partition is a collection of rows that sit on one physical machine in your cluster
- If you have one partition, Spark will have a parallelism of only one, even if you have thousands of executors
- If you have many partitions but only one executor, Spark will still have a parallelism of only one because there is only one computation resource
- DataFrames you do not (for the most part) manipulate partitions manually or individually.

Transformations

- Core data structures are immutable, meaning they cannot be changed after they are created.
- To change a DataFrame, you need to instruct Spark how you would like to modify it to do what you want.
- These instructions are called transformations
- Notice that these return no output. This is because we specified only an abstract transformation, and Spark will not act on transformations until we call an action
- Transformations are the core of how you express your business logic using Spark

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

- There are two types of transformations: those that specify narrow dependencies, and those that specify wide dependencies.

Narrow transformations

- Transformations consisting of narrow dependencies (we'll call them narrow transformations) are those for which each input partition will contribute to only one output partition.
- Spark will automatically perform an operation called pipelining, meaning that if we specify multiple filters on DataFrames, they'll all be performed in-memory.

Narrow transformations

- A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions
- You will often hear this referred to as a shuffle whereby Spark will exchange partitions across the cluster.
- When we perform a shuffle, Spark writes the results to disk.

Lazy Evaluation

- Lazy evaulation means that Spark will wait until the very last moment to execute the graph of computation instructions.
- Instead of modifying the data immediately when you express some operation, you build up a plan of transformations that you would like to apply to your source data
- Spark compiles this plan from your raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster

- This provides immense benefits because Spark can optimize the entire data flow from end to end.

- Predicate pushdown on DataFrames (filter)

Actions

- Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an action
- An action instructs Spark to compute a result from a series of transformations

```
#Action
divisBy2.count()
```

```
500
```

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

- Action, we started a Spark job

Spark UI

- You can monitor the progress of a job through the Spark web UI. The Spark UI is available on port 4040 of the driver node. If you are running in local mode, this will be http://localhost:4040.

- It's very useful, especially for tuning and debugging

An End-to-End Example

- Download the repo : https://github.com/databricks/Spark-The-Definitive-Guide

An End-to-End Example

- Load csv

Ce PC › OS (C:) › spark › Spark-The-Definitive-Guide-master › data › flight-data › csv

Nom	Modifié le
2010-summary	12/09/2019 17:38
2011-summary	12/09/2019 17:38
2012-summary	12/09/2019 17:38
2013-summary	12/09/2019 17:38
2014-summary	12/09/2019 17:38
2015-summary	12/09/2019 17:38

2015-summary.csv

```
DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
United States,Romania,15
United States,Croatia,1
United States,Ireland,344
Egypt,United States,15
United States,India,62
United States,Singapore,1
United States,Grenada,62
Costa Rica,United States,588
Senegal,United States,40
Moldova,United States,1
United States,Sint Maarten,325
```

An End-to-End Example

- Read csv data

```python
# Read csv data
flightData2015 = spark\
  .read\
  .option("inferSchema", "true")\
  .option("header", "true")\
  .csv("data/2015-summary.csv")
```

```python
# Get first n lines
flightData2015.take(5)
```

```
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count=1),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count=344),
 Row(DEST_COUNTRY_NAME='Egypt', ORIGIN_COUNTRY_NAME='United States', count=15),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='India', count=62)]
```

An End-to-End Example

- Physical plan
- The top being the end result
- Bottom being the source(s) of data
- Sort, Exchange (wide), FileScan
- Remember : wide transformation means Shuffle

```
#Explain physical plan
flightData2015.sort("count").explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [count#42 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(count#42 ASC NULLS FIRST, 200), ENSURE_REQUIREMENTS, [plan_id=89]
      +- FileScan csv [DEST_COUNTRY_NAME#40,ORIGIN_COUNTRY_NAME#41,count#42] Batched: false, DataFil
```

An End-to-End Example

- Set shuffle partition
- Default is 200

```
[13] # set shuffle
     spark.conf.set("spark.sql.shuffle.partitions", "5")
```

```
[14] flightData2015.sort("count").take(2)
```

```
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Singapore', count=1),
 Row(DEST_COUNTRY_NAME='Moldova', ORIGIN_COUNTRY_NAME='United States', count=1)]
```

An End-to-End Example

- DataFrames and SQL
- Register any DataFrame as a table or view (a temporary table) and query it using pure SQL

```
#Create sql table
flightData2015.createOrReplaceTempView("flight_data_2015")
```

An End-to-End Example

- DataFrames and SQL
- There is no performance difference between writing SQL queries or writing DataFrame code, they both “compile” to the same underlying plan that we specify in DataFrame code.

```
sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")
dataFrameWay = flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.count()
sqlWay.explain()
dataFrameWay.explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[DEST_COUNTRY_NAME#40], functions=[count(1)])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#40, 5), ENSURE_REQUIREMENTS, [plan_id=117]
      +- HashAggregate(keys=[DEST_COUNTRY_NAME#40], functions=[partial_count(1)])
         +- FileScan csv [DEST_COUNTRY_NAME#40] Batched: false, DataFilters: [], Format: CSV, Location: InMe


== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[DEST_COUNTRY_NAME#40], functions=[count(1)])
   +- Exchange hashpartitioning(DEST_COUNTRY_NAME#40, 5), ENSURE_REQUIREMENTS, [plan_id=130]
      +- HashAggregate(keys=[DEST_COUNTRY_NAME#40], functions=[partial_count(1)])
         +- FileScan csv [DEST_COUNTRY_NAME#40] Batched: false, DataFilters: [], Format: CSV, Location: InMe
```

An End-to-End Example

- Maximum number of flights to and from any given location

```
# Maximum number of flights
from pyspark.sql.functions import max
flightData2015.select(max("count")).take(1)
```

```
[Row(max(count)=370002)]
```

- Top five destination countries SQL

```
#Top 5 destination
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
""")
maxSql.show()
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

An End-to-End Example

- Top five destination countries

```python
#Top 5
from pyspark.sql.functions import desc

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

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

An End-to-End Example

- Execution plan is a directed acyclic graph (DAG) of transformations, each resulting in a new immutable DataFrame, on which we call an action to generate a result
- Conceptual plan

An End-to-End Example

- Limit statement as well as the orderBy (first line)
- Aggregation happens in two phases, in the partial_sum calls (commutative)

```
#Physical plan
flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- TakeOrderedAndProject(limit=5, orderBy=[destination_total#153L DESC NULLS LAST],
   +- HashAggregate(keys=[DEST_COUNTRY_NAME#40], functions=[sum(count#42)])
      +- Exchange hashpartitioning(DEST_COUNTRY_NAME#40, 5), ENSURE_REQUIREMENTS, [
         +- HashAggregate(keys=[DEST_COUNTRY_NAME#40], functions=[partial_sum(count
            +- FileScan csv [DEST_COUNTRY_NAME#40,count#42] Batched: false, DataFil
```

Recap

- Transformations
- Action
- Lazy evaluation
- Partition
- Shuffle
- Dataframes
- Optimization of execution plan

Chapter 3. : A Tour of Spark’s Toolset

Global view

Structured Streaming
Advanced Analytics
Libraries & Ecosystem
Structured APIs
Datasets
DataFrames
SQL
Low-level APIs
RDDs
Distributed Variables

Running Production Applications

- spark-submit
- send your application code to a cluster and launch it to execute there
- You can do this with all of Spark's support cluster managers including Standalone, Mesos, and YARN.

- Java/Scala

```
./bin/spark-submit \
--class org.apache.spark.examples.SparkPi \
--master local \
./examples/jars/spark-examples_2.11-2.2.0.jar 10
```

- Python

```
./bin/spark-submit \
--master local \
./examples/src/main/python/pi.py 10
```

Datasets: Type-Safe Structured APIs

- Java and Scala
- The Dataset API is not available in Python and R, because those languages are dynamically typed.
- The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java ArrayList or Scala Seq
- Datasets are type-safe
- This makes Datasets especially attractive for writing large applications, with which multiple software engineers must interact through well-defined interfaces.

Datasets: Type-Safe Structured APIs

- The Dataset class is parameterized with the type of object contained inside: Dataset<T> in Java and Dataset[T] in Scala. For example, a Dataset[Person] will be guaranteed to contain objects of class Person.
- In fact, if you use Scala or Java, all “DataFrames” are actually Datasets of type Row
- To efficiently support domain-specific objects, a special concept called an “Encoder” is required. The encoder maps the domain-specific type T to Spark’s internal type system
- When you use the Dataset API, for every row it touches, this domain specifies type, Spark converts the Spark Row format to the object you specified (a case class or Java class). This conversion slows down your operations but can provide more flexibility

Datasets: Type-Safe Structured APIs

- Start spark shell

```
case class Flight(DEST_COUNTRY_NAME: String,
                  ORIGIN_COUNTRY_NAME: String,
                  count: BigInt)

val flightsDF = spark.read
                     .parquet("/data/parquet/2010-summary.parquet/")

val flights = flightsDF.as[Flight]
```

Datasets: Type-Safe Structured APIs

- Start spark shell

```
// in Scala
flights
  .filter(flight_row => flight_row.ORIGIN_COUNTRY_NAME != "Canada")
  .map(flight_row => flight_row)
  .take(5)

flights
  .take(5)
  .filter(flight_row => flight_row.ORIGIN_COUNTRY_NAME != "Canada")
  .map(fr => Flight(fr.DEST_COUNTRY_NAME, fr.ORIGIN_COUNTRY_NAME, fr.count + 5))
```

Structured Streaming

- High-level API for stream processing that became production-ready in Spark 2.2
- Can reduce latency and allow for incremental processing.

- Download https://github.com/databricks/Spark-The-Definitive-Guide/tree/master/data/retail-data

1	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
2	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.0	United Kingdom
3	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
4	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.0	United Kingdom
5	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
6	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850.0	United Kingdom
7	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	2010-12-01 08:26:00	7.65	17850.0	United Kingdom
8	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	2010-12-01 08:26:00	4.25	17850.0	United Kingdom

Structured Streaming

- Load static data

```
# Streaming
staticDataFrame = spark.read.format("csv")\
.option("header", "true")\
.option("inferSchema", "true")\
.load("data/retail-data/by-day/*.csv")

staticDataFrame.createOrReplaceTempView("retail_data")
staticSchema = staticDataFrame.schema
```

- Example : we'll take a look at the sale hours during which a given customer (identified by CustomerId) makes a large purchase. For example, let's add a total cost column and see on what days a customer spent the most. The window function will include all data from each day in the aggregation. It's simply a window over the time–series column in our data.

```
from pyspark.sql.functions import window, column, desc, col
staticDataFrame.selectExpr("CustomerId",
                           "(UnitPrice * Quantity) as total_cost",
                           "InvoiceDate")\
.groupBy(col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
.sum("total_cost")\
.show(5)
```

```
+----------+------------------+-----------------+
|CustomerId|            window|  sum(total_cost)|
+----------+------------------+-----------------+
```

Structured Streaming

- Streaming version

```
#Stream version

streamingDataFrame = spark.readStream\
.schema(staticSchema)\
.option("maxFilesPerTrigger", 1)\
.format("csv")\
.option("header", "true")\
.load("/data/retail-data/by-day/*.csv")

streamingDataFrame.isStreaming
```

```
True
```

```
#Transformations
purchaseByCustomerPerHour = streamingDataFrame\
.selectExpr(
"CustomerId",
"(UnitPrice * Quantity) as total_cost",
"InvoiceDate")\
.groupBy(
col("CustomerId"), window(col("InvoiceDate"), "1 day"))\
.sum("total_cost")
```

- This is still a lazy operation, so we will need to call a streaming action to start the execution of this data flow.

```
#Start
purchaseByCustomerPerHour.writeStream\
.format("memory")\
.queryName("customer_purchases")\
.outputMode("complete")\
.start()
```

Structured Streaming

- Run queries against it to debug what our result will look like

```
spark.sql("""
SELECT *
FROM customer_purchases_3
ORDER BY `sum(total_cost)` DESC
""")\
.show(5)
```

```
+----------+--------------------+------------------+
|CustomerId|              window|   sum(total_cost)|
+----------+--------------------+------------------+
```

Machine Learning and Advanced Analytics

- Another popular aspect of Spark is its ability to perform large-scale machine learning with a built-in library of machine learning algorithms called MLlib.
- Spark provides a sophisticated machine learning API for performing a variety of machine learning tasks, from classification to regression, and clustering to deep learning

Machine Learning and Advanced Analytics

- To demonstrate this functionality, we will perform some basic clustering on our data using a standard algorithm called k-means

Demonstration of the standard algorithm

1. *k* initial "means" (in this case *k*=3) are randomly generated within the data domain (shown in color).

2. *k* clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means.

3. The centroid of each of the *k* clusters becomes the new mean.

4. Steps 2 and 3 are repeated until convergence has been reached.

https://en.wikipedia.org/wiki/K-means_clustering

Machine Learning and Advanced Analytics

- Machine learning algorithms in MLlib require that data is represented as numerical values. Our current data is represented by a variety of different types, including timestamps, integers, and strings. Therefore we need to transform this data into some numerical representation

```
staticDataFrame.printSchema()
```

```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

Machine Learning and Advanced Analytics

- Fill na (reokace missing data)
- Coalesce : reduce the number of partition

```
from pyspark.sql.functions import date_format, col
preppedDataFrame = staticDataFrame\
.na.fill(0)\
.withColumn("day_of_week", date_format(col("InvoiceDate"), "EEEE"))\
.coalesce(5)
```

Machine Learning and Advanced Analytics

- Create manually test and training set

```
trainDataFrame = preppedDataFrame\
.where("InvoiceDate < '2011-07-01'")
testDataFrame = preppedDataFrame\
.where("InvoiceDate >= '2011-07-01'")
```

- DataFrame transformations

```
#DataFrame transformations
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer()\
.setInputCol("day_of_week")\
.setOutputCol("day_of_week_index")
```

- This will turn our days of weeks into corresponding numerical values. For example, Spark might represent Saturday as 6, and Monday as 1
- We are implicitly stating that Saturday is greater than Monday (by pure numerical values). This is obviously incorrect. To fix this, we therefore need to use a OneHotEncoder to encode each of these values as their own column.

Machine Learning and Advanced Analytics

- OneHotEncoder

```
#DataFrame transformations OneHotEncoder
from pyspark.ml.feature import OneHotEncoder
encoder = OneHotEncoder()\
.setInputCol("day_of_week_index")\
.setOutputCol("day_of_week_encoded")
```

- Each of these will result in a set of columns that we will “assemble” into a vector. All machine learning algorithms in Spark take as input a Vector type, which must be a set of numerical values:

```
from pyspark.ml.feature import VectorAssembler
vectorAssembler = VectorAssembler()\
.setInputCols(["UnitPrice", "Quantity", "day_of_week_encoded"])\
.setOutputCol("features")
```

- We have three key features: the price, the quantity, and the day of week

Machine Learning and Advanced Analytics

- Pipeline

```
from pyspark.ml import Pipeline
transformationPipeline = Pipeline()\
.setStages([indexer, encoder, vectorAssembler])
```

- Fit

- Transform

```
fittedPipeline = transformationPipeline.fit(trainDataFrame)
transformedTraining = fittedPipeline.transform(trainDataFrame)
transformedTraining.cache()
```

- Cache

Machine Learning and Advanced Analytics

- Create model

```
from pyspark.ml.clustering import KMeans
kmeans = KMeans()\
.setK(20)\
.setSeed(1)
```

- Train model

```
kmModel = kmeans.fit(transformedTraining)
```

Lower-Level APIs

- Virtually everything in Spark is built on top of RDDs.
- Resilient Distributed Datasets
- RDDs are lower level than DataFrames because they reveal physical execution characteristics (like partitions) to end users
- As end users, you shouldn’t need to use RDDs much in order to perform many tasks unless you’re maintaining older Spark code

Structured APIs—DataFrames, SQL, and Datasets

Chapter 4. Structured API Overview

Structured API Overview

- The Structured APIs are a tool for manipulating all sorts of data, from unstructured log files to semi-structured CSV files and highly structured Parquet files

  - Datasets
  - DataFrames
  - SQL tables and views

- Tips : The logical structures that we manipulate with transformations and actions are DataFrames and Datasets. To create a new DataFrame or Dataset, you call a transformation. To start computation or convert to native language types, you call an action

DataFrames and Datasets

- DataFrames and Datasets are (distributed) table-like collections with well-defined rows and columns.
- Each column must have the same number of rows as all the other columns (although you can use null to specify the absence of a value) and each column has type information that must be consistent for every row in the collection.
- Tables and views are basically the same thing as DataFrames. We just execute SQL against them instead of DataFrame code.

Schemas

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

Structured Spark Types

- Spark is effectively a programming language of its own
- Spark uses an engine called Catalyst that maintains its own type information through the planning and processing of work
- In doing so, this opens up a wide variety of execution optimizations that make significant differences.
- Even if we use Spark's Structured APIs from Python or R, the majority of our manipulations will operate strictly on Spark types, not Python types

DataFrames Versus Datasets

- The “untyped” DataFrames and the “typed” Datasets
- DataFrames are untyped is aslightly inaccurate; they have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at runtime
- Datasets, on the other hand, check whether types conform to the specification at compile time. Datasets are only available to Java Virtual Machine (JVM)– based languages (Scala and Java)
- To Spark (in Scala), DataFrames are simply Datasets of Type Row. The “Row” type is Spark’s internal representation of its optimized in-memory format for computation

Columns and Rows

- Columns represent a simple type like an integer or string, a complex type like an array or map, or a null value.

- Spark Column types as columns in a table.

- A row is nothing more than a record of data
- Each record in a DataFrame must be of type Row, as we can see when we collect the following DataFrames.
- We can create these rows manually from SQL, from Resilient Distributed Datasets (RDDs), from data sources, or manually from scratch

```
# Create rows
spark.range(2).collect()
```

```
[Row(id=0), Row(id=1)]
```

Spark Types

- Spark has a large number of internal type representations

```
# Type
from pyspark.sql.types import *
s = StringType()
s
```

```
StringType()
```

- https://spark.apache.org/docs/latest/sql-ref-datatypes.html

Structured API Execution

- How this code is actually executed across a cluster.
  - 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.

The Catalyst Optimizer

Logical Planning

- The first phase of execution is meant to take user code and convert it into a logical plan.
- This logical plan only represents a set of abstract transformations that do not refer to executors or drivers, it's purely to convert the user's set of expressions into the most optimized version.

The structured API logical planning process

Physical Planning

- The physical plan, often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model
- This result is why you might have heard Spark referred to as a compiler—it takes queries in DataFrames, Datasets, and SQL and compiles them into RDD transformations for you.

The structured API logical planning process

Chapter 5. Basic Structured Operations

Dataframes schemas

- Tips : what is Dataframes/Dataset et schemas

```
# Dataframe operations

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

```
root
 |-- DEST_COUNTRY_NAME: string (nullable = true)
 |-- ORIGIN_COUNTRY_NAME: string (nullable = true)
 |-- count: long (nullable = true)
```

Dataframes schemas

- Datasource define the schema

```
spark.read.format("json") \
.load("2015-summary.json")
df.schema
```

```
StructType([StructField('DEST_COUNTRY_NAME', StringType(), True),
StructField('ORIGIN_COUNTRY_NAME', StringType(), True), StructField('count',
LongType(), True)])
```

Dataframes schemas

- Define explicitly the schema

```
# explicit schema
from pyspark.sql.types import StructField, StructType, StringType, LongType
myManualSchema = StructType([
StructField("DEST_COUNTRY_NAME", StringType(), True),
StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
StructField("count", LongType(), False, metadata={"hello":"world"})
])
df = spark.read.format("json").schema(myManualSchema)\
.load("2015-summary.json")
```

Columns and Expressions

- Columns in Spark are similar to columns in a spreadsheet, R dataframe, or pandas DataFrame.
- You can select, manipulate, and remove columns from DataFrames and these operations are represented as expressions
- To Spark, columns are logical constructions that simply represent a value computed on a perrecord basis by means of an expression. This means that to have a real value for a column, we need to have a row; and to have a row, we need to have a DataFrame

- Construct by two simplest ways are by using the col or column functions

```
# construct column
from pyspark.sql.functions import col, column
col("someColumnName")
column("someColumnName")
```

```
Column<'someColumnName'>
```

- Get columns on dataframe

```
spark.read.format("json") \
.load("2015-summary.json").columns
```

```
['DEST_COUNTRY_NAME', 'ORIGIN_COUNTRY_NAME', 'count']
```

Columns and Expressions

- An expression is a set of transformations on one or more values in a record in a DataFrame.
- In the simplest case, an expression, created via the expr function, is just a DataFrame column reference. In the simplest case, expr("someCol") is equivalent to col("someCol").

```
from pyspark.sql.functions import expr

expr("columnName")
```

```
Column<'columnName'>
```

Columns and Expressions

- Columns as expressions
- When using an expression, the expr function can actually pars transformations and column references from a string and can subsequently be passed into further transformations.
- expr("someCol - 5") is the same transformation as performing col("someCol") - 5, or even expr("someCol") - 5.

```
(((col("someCol") + 5) * 200) - 6) < col("otherCol")
```

```
Column<'((((someCol + 5) * 200) - 6) < otherCol)'>
```

A logical tree

Records and Rows

- You can create rows by manually instantiating a Row object
- Rows themselves do not have schemas.

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

```
<Row('Hello', None, 1, False)>
```

- Get row value by index

```
myRow[2]
```

```
1
```

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

DataFrame Transformations

- Create dataFrame from Datasource
- Register into temporary view for SQL

```
# Create dataframes

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

```
DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: bigint]
```

DataFrame Transformations

- Create dataFrame manually

```python
# Create dataFrame manually

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()
```

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

select and selectExpr

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

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
+-----------------+
only showing top 2 rows
```

```
#in SQL
#SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME FROM dfTable LIMIT 2
df.select("DEST_COUNTRY_NAME", "ORIGIN_COUNTRY_NAME").show(2)
```

```
+-----------------+-------------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|
+-----------------+-------------------+
|    United States|            Romania|
|    United States|            Croatia|
+-----------------+-------------------+
only showing top 2 rows
```

select and selectExpr

- 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

```
from pyspark.sql.functions import expr, col, column
df.select(
expr("DEST_COUNTRY_NAME"),
col("DEST_COUNTRY_NAME"),
column("DEST_COUNTRY_NAME"))\
.show(2)
```

```
+-----------------+-----------------+-----------------+
|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|DEST_COUNTRY_NAME|
+-----------------+-----------------+-----------------+
|    United States|    United States|    United States|
|    United States|    United States|    United States|
+-----------------+-----------------+-----------------+
only showing top 2 rows
```

- Create alias

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

```
+-------------+
|  destination|
+-------------+
|United States|
|United States|
+-------------+
only showing top 2 rows
```

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

```
+-----------------+
|DEST_COUNTRY_NAME|
+-----------------+
|    United States|
|    United States|
+-----------------+
only showing top 2 rows
```

select and selectExpr

- 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

```
#Best way
df.selectExpr("DEST_COUNTRY_NAME as newColumnName", "DEST_COUNTRY_NAME") \
.show(2)
```

```
+-------------+-----------------+
|newColumnName|DEST_COUNTRY_NAME|
+-------------+-----------------+
|United States|    United States|
|United States|    United States|
+-------------+-----------------+
only showing top 2 rows
```

select and selectExpr

- In fact, we can add any valid non-aggregating SQL statement, and as long as the columns resolve, it will be valid

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

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
only showing top 2 rows
```

```
#in SQL
#SELECT avg(count), count(distinct(DEST_COUNTRY_NAME)) FROM dfTable LIMIT 2
```

Converting to Spark Types (Literals)

- Sometimes, we need to pass explicit values into Spark that are just a value (rather than a new column). This might be a constant value or something we'll need to compare to later on. The way we do this is through literals

```
# add literal
# in SQL
#SELECT *, 1 as One FROM dfTable LIMIT 2
from pyspark.sql.functions import lit
df.select(expr("*"), lit(1).alias("One")).show(2)
```

```
+-----------------+-------------------+-----+---+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|One|
+-----------------+-------------------+-----+---+
|    United States|            Romania|   15|  1|
|    United States|            Croatia|    1|  1|
+-----------------+-------------------+-----+---+
only showing top 2 rows
```

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

```
#in SQL
#SELECT *, 1 as numberOne FROM dfTable LIMIT 2
df.withColumn("numberOne", lit(1)).show(2)
```

```
+-----------------+-------------------+-----+---------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|numberOne|
+-----------------+-------------------+-----+---------+
|    United States|            Romania|   15|        1|
|    United States|            Croatia|    1|        1|
+-----------------+-------------------+-----+---------+
only showing top 2 rows
```

Adding Columns

- Set a Boolean flag for when the origin country is the same as the destination country

- The withColumn function takes two arguments: the column name and the expression that will create the value for that given row in the DataFrame

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

```
+-----------------+-------------------+-----+-------------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|withinCountry|
+-----------------+-------------------+-----+-------------+
|    United States|            Romania|   15|        false|
|    United States|            Croatia|    1|        false|
+-----------------+-------------------+-----+-------------+
only showing top 2 rows
```

- Rename column name

```
#Rename column first way
df.withColumn("Destination", expr("DEST_COUNTRY_NAME")).columns
```

```
['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.

```
#Rename column second way
df.withColumnRenamed("DEST_COUNTRY_NAME", "dest").columns
```

```
['dest', 'ORIGIN_COUNTRY_NAME', 'count']
```

Reserved Characters and Keywords

- In Spark, we do this by using backtick (`) characters

```
dfWithLongColName = df.withColumn(
"This Long Column-Name",
expr("ORIGIN_COUNTRY_NAME"))

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

```
+---------------------+-------+
|This Long Column-Name|new col|
+---------------------+-------+
|              Romania|Romania|
|              Croatia|Croatia|
+---------------------+-------+
only showing top 2 rows
```

Removing columns

```
[51] df.columns
```

```
['DEST_COUNTRY_NAME', 'ORIGIN_COUNTRY_NAME', 'count']
```

```
# Drop column

df.drop("ORIGIN_COUNTRY_NAME").columns
```

```
['DEST_COUNTRY_NAME', 'count']
```

Changing a Column's Type (cast)

- We can convert columns from one type to another by casting the column from one type to another

```
#in SQL
#SELECT *, cast(count as long) AS count2 FROM dfTable

df.withColumn("count2", col("count").cast("integer"))
```

```
DataFrame[DEST_COUNTRY_NAME: string, ORIGIN_COUNTRY_NAME: string, count: bigint, count2: int]
```

Filtering Rows

- To filter rows, we create an expression that evaluates to true or false. You then filter out the rows with an expression that is equal to false.

```
#Filter

#in SQL
#SELECT * FROM dfTable WHERE count < 2 LIMIT 2
df.filter(col("count") < 2).show(2)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Croatia|    1|
|    United States|          Singapore|    1|
+-----------------+-------------------+-----+
only showing top 2 rows
```

- You can use where or filter

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

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Croatia|    1|
|    United States|          Singapore|    1|
+-----------------+-------------------+-----+
only showing top 2 rows
```

Filtering Rows

- To filter rows, we create an expression that evaluates to true or false. You then filter out the rows with an expression that is equal to false.

- You can use where or filter

```
# in SQL
#SELECT * FROM dfTable WHERE count < 2 AND ORIGIN_COUNTRY_NAME != "Croatia"
#LIMIT 2
df.where(col("count") < 2).where(col("ORIGIN_COUNTRY_NAME") != "Croatia")\
.show(2)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|          Singapore|    1|
|          Moldova|      United States|    1|
+-----------------+-------------------+-----+
only showing top 2 rows
```

Getting Unique Rows

- A very common use case is to extract the unique or distinct values in a DataFrame.

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

```
256
```

Concatenating and Appending Rows (Union)

- 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
- To union two DataFrames, you must be sure that they have the same schema and number of columns; otherwise, the union will fail

```python
# union two dataframe

from pyspark.sql import Row
schema = df.schema
newRows = [Row("New Country", "Other Country", 5),
           Row("New Country 2", "Other Country 3", 1)]
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()
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Croatia|    1|
|    United States|          Singapore|    1|
|    United States|          Gibraltar|    1|
|    United States|             Cyprus|    1|
|    United States|            Estonia|    1|
|    United States|          Lithuania|    1|
|    United States|           Bulgaria|    1|
|    United States|            Georgia|    1|
|    United States|            Bahrain|    1|
|    United States|   Papua New Guinea|    1|
|    United States|         Montenegro|    1|
|    United States|            Namibia|    1|
|    New Country 2|    Other Country 3|    1|
+-----------------+-------------------+-----+
```

Sorting Rows

- 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.

```python
df.sort("count").show(2)
df.orderBy("count", "DEST_COUNTRY_NAME").show(2)
df.orderBy(col("count"), col("DEST_COUNTRY_NAME")).show(2)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|          Moldova|      United States|    1|
|    United States|            Croatia|    1|
+-----------------+-------------------+-----+
only showing top 2 rows

+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|     Burkina Faso|      United States|    1|
|    Cote d'Ivoire|      United States|    1|
+-----------------+-------------------+-----+
only showing top 2 rows

+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|     Burkina Faso|      United States|    1|
|    Cote d'Ivoire|      United States|    1|
+-----------------+-------------------+-----+
only showing top 2 rows
```

Sorting Rows

- To more explicitly specify sort direction, you need to use the asc and desc functions if operating on a column
- 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.

- sortWithinPartitions for optimization

```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)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|          Moldova|      United States|    1|
|    United States|            Croatia|    1|
+-----------------+-------------------+-----+
only showing top 2 rows

+-----------------+-------------------+------+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME| count|
+-----------------+-------------------+------+
|    United States|      United States|370002|
|    United States|             Canada|  8483|
+-----------------+-------------------+------+
only showing top 2 rows
```

Limit

```
df.orderBy(expr("count desc")).limit(6).show()
```

```
+--------------------+-------------------+-----+
|   DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+--------------------+-------------------+-----+
|               Malta|      United States|    1|
|Saint Vincent and...|      United States|    1|
|       United States|            Croatia|    1|
|       United States|          Gibraltar|    1|
|       United States|          Singapore|    1|
|             Moldova|      United States|    1|
+--------------------+-------------------+-----+
```

Repartition and Coalesce

- Repartition will incur a full shuffle of the data
- This means that you should typically only repartition when the future number of partitions is greater than your current number of partitions or when you are looking to partition by a set of columns

```
df.rdd.getNumPartitions() # 1
```

```
1
```

```
dfRepart = df.repartition(5)
dfRepart.rdd.getNumPartitions()
```

```
5
```

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

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

Repartition and Coalesce

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

```
dfCo = df.repartition(5, col("DEST_COUNTRY_NAME")).coalesce(2)
dfCo.rdd.getNumPartitions()
```

```
2
```

Collecting Rows to the Driver

- 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.

- Collecting rows to the driver in order to iterate over the entire dataset.

```
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()
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|   15|
|    United States|            Croatia|    1|
|    United States|            Ireland|  344|
|            Egypt|      United States|   15|
```

```
collectDF.toLocalIterator()
```

Chapter 6. Working with Different Types of Data

Where to Look for APIs

- https://spark.apache.org/docs/latest/api/python/reference/index.html

Different Types of Data

- Booleans
- Numbers
- Strings
- Dates and timestamps
- Handling null
- Complex types
- User-defined functions

```
df = spark.read.format("csv")\
.option("header", "true")\
.option("inferSchema", "true")\
.load("2010-12-01.csv")
df.printSchema()
df.createOrReplaceTempView("dfTable")
```

```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: timestamp (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: double (nullable = true)
 |-- Country: string (nullable = true)
```

Converting to Spark Types

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

```
from pyspark.sql.functions import lit
df.select(lit(5), lit("five"), lit(5.0))
```

```
DataFrame[5: int, five: string, 5.0: double]
```

Working with Booleans

- Booleans are essential when it comes to data analysis because they are the foundation for all filtering.
- Boolean statements consist of four elements: and, or, true, and false

```
from pyspark.sql.functions import col
df.where(col("InvoiceNo") != 536365)\
.select("InvoiceNo", "Description")\
.show(5, False)
```

```
+---------+----------------------------+
|InvoiceNo|Description                 |
+---------+----------------------------+
|536366   |HAND WARMER UNION JACK      |
|536366   |HAND WARMER RED POLKA DOT   |
|536367   |ASSORTED COLOUR BIRD ORNAMENT|
|536367   |POPPY'S PLAYHOUSE BEDROOM   |
|536367   |POPPY'S PLAYHOUSE KITCHEN   |
+---------+----------------------------+
only showing top 5 rows
```

Working with Booleans

- Another option and probably the cleanest is to specify the predicate as an expression in a string

```
df.where("InvoiceNo = 536365").show(5, False)
df.where("InvoiceNo <> 536365").show(5, False)
```

```
+---------+---------+-----------------------------------+--
|InvoiceNo|StockCode|Description                        |Qu
+---------+---------+-----------------------------------+--
|536365   |85123A   |WHITE HANGING HEART T-LIGHT HOLDER |6
|536365   |71053    |WHITE METAL LANTERN                |6
|536365   |84406B   |CREAM CUPID HEARTS COAT HANGER     |8
|536365   |84029G   |KNITTED UNION FLAG HOT WATER BOTTLE|6
|536365   |84029E   |RED WOOLLY HOTTIE WHITE HEART.     |6
+---------+---------+-----------------------------------+--
only showing top 5 rows

+---------+---------+----------------------------+--------
|InvoiceNo|StockCode|Description                 |Quantity
+---------+---------+----------------------------+--------
|536366   |22633    |HAND WARMER UNION JACK      |6
|536366   |22632    |HAND WARMER RED POLKA DOT   |6
|536367   |84879    |ASSORTED COLOUR BIRD ORNAMENT|32
|536367   |22745    |POPPY'S PLAYHOUSE BEDROOM   |6
|536367   |22748    |POPPY'S PLAYHOUSE KITCHEN   |6
+---------+---------+----------------------------+--------
only showing top 5 rows
```

Working with Booleans

- Multiple parts when you use and or or

```
from pyspark.sql.functions import instr
priceFilter = col("UnitPrice") > 600
descripFilter = instr(df.Description, "POSTAGE") >= 1
df.where(df.StockCode.isin("DOT")).where(priceFilter | descripFilter).show()
```

```
+---------+---------+--------------+--------+-------------------+---------+--
|InvoiceNo|StockCode|   Description|Quantity|        InvoiceDate|UnitPrice|Cu
+---------+---------+--------------+--------+-------------------+---------+--
|   536544|      DOT|DOTCOM POSTAGE|       1|2010-12-01 14:32:00|   569.77|
|   536592|      DOT|DOTCOM POSTAGE|       1|2010-12-01 17:06:00|   607.49|
+---------+---------+--------------+--------+-------------------+---------+--
```

Working with Booleans

- Notice how we did not need to specify our filter as an expression and how we could use a column name without any extra work.

```
from pyspark.sql.functions import instr
DOTCodeFilter = col("StockCode") == "DOT"
priceFilter = col("UnitPrice") > 600
descripFilter = instr(col("Description"), "POSTAGE") >= 1
df.withColumn("isExpensive", DOTCodeFilter & (priceFilter | descripFilter))\
.where("isExpensive")\
.select("unitPrice", "isExpensive").show(5)
```

```
+---------+-----------+
|unitPrice|isExpensive|
+---------+-----------+
|   569.77|       true|
|   607.49|       true|
+---------+-----------+
```

Working with Booleans

- Notice how we did not need to specify our filter as an expression and how we could use a column name without any extra work.

```
from pyspark.sql.functions import instr
DOTCodeFilter = col("StockCode") == "DOT"
priceFilter = col("UnitPrice") > 600
descripFilter = instr(col("Description"), "POSTAGE") >= 1
df.withColumn("isExpensive", DOTCodeFilter & (priceFilter | descripFilter))\
.where("isExpensive")\
.select("unitPrice", "isExpensive").show(5)
```

```
+---------+-----------+
|unitPrice|isExpensive|
+---------+-----------+
|   569.77|       true|
|   607.49|       true|
+---------+-----------+
```

Working with Booleans

- if you're working with null data when creating Boolean expressions. If there is a null in your data, you'll need to treat things a bit differently.

```
# null safe
df.where(col("Description").eqNullSafe("POSTAGE")).show()
```

```
+---------+---------+-----------+--------+-------------------+---------+
|InvoiceNo|StockCode|Description|Quantity|        InvoiceDate|UnitPrice|
+---------+---------+-----------+--------+-------------------+---------+
|   536370|     POST|    POSTAGE|       3|2010-12-01 08:45:00|     18.0|
|   536403|     POST|    POSTAGE|       1|2010-12-01 11:27:00|     15.0|
|   536527|     POST|    POSTAGE|       1|2010-12-01 13:04:00|     18.0|
+---------+---------+-----------+--------+-------------------+---------+
```

Working with Numbers

- When working with big data, the second most common task you will do after filtering things is counting things.

- Imagine that we found out that we mis-recorded the quantity in our retail dataset and the true quantity is equal to (the current quantity * the unit price)^2 + 5

```python
# Manipulate numbers
from pyspark.sql.functions import expr, pow
fabricatedQuantity = pow(col("Quantity") * col("UnitPrice"), 2) + 5
df.select(expr("CustomerId"), fabricatedQuantity.alias("realQuantity")).show(2)
```

```
+----------+------------------+
|CustomerId|      realQuantity|
+----------+------------------+
|   17850.0|239.08999999999997|
|   17850.0|          418.7156|
+----------+------------------+
```

Working with Numbers

- Common numerical task is rounding

```
from pyspark.sql.functions import lit, round, bround
df.select(round(lit("2.5")), bround(lit("2.5"))).show(2)
```

```
+-------------+-------------+
|round(2.5, 0)|bround(2.5, 0)|
+-------------+-------------+
|          3.0|          2.0|
|          3.0|          2.0|
+-------------+-------------+
only showing top 2 rows
```

- Common task is to compute summary statistics for a column or set of columns.

```
# describe

df.describe().show()
```

```
+-------+-----------------+-----------------+
|summary|        InvoiceNo|        StockCode|
+-------+-----------------+-----------------+
|  count|             3108|             3108|
|   mean| 536516.684944841|27834.304044117645|
| stddev|72.89447869788873|17407.897548583845|
|    min|           536365|            10002|
|    max|          C536548|             POST|
+-------+-----------------+-----------------+
```

Working with Strings

- String manipulation shows up in nearly every data flow, and it's worth explaining what you can do with strings.

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

```
from pyspark.sql.functions import initcap
df.select(initcap(col("Description"))).show()
```

```
+--------------------+
|initcap(Description)|
+--------------------+
|White Hanging Hea...|
| White Metal Lantern|
|Cream Cupid Heart...|
```

Working with Strings

- You can cast strings in uppercase and lowercase

```
from pyspark.sql.functions import lower, upper
df.select(col("Description"),
lower(col("Description")),
upper(lower(col("Description")))).show(2)
```

```
+--------------------+--------------------+-------------------------+
|         Description|  lower(Description)|upper(lower(Description))|
+--------------------+--------------------+-------------------------+
|WHITE HANGING HEA...|white hanging hea...|     WHITE HANGING HEA...|
| WHITE METAL LANTERN| white metal lantern|      WHITE METAL LANTERN|
+--------------------+--------------------+-------------------------+
only showing top 2 rows
```

- adding or removing spaces around a string

```
from pyspark.sql.functions import lit, ltrim, rtrim, \
 rpad, lpad, trim
df.select(
ltrim(lit(" HELLO ")).alias("ltrim"),
rtrim(lit(" HELLO ")).alias("rtrim"),
trim(lit(" HELLO ")).alias("trim"),
lpad(lit("HELLO"), 3, " ").alias("lp"),
rpad(lit("HELLO"), 10, " ").alias("rp")).show(2)
```

```
+------+------+-----+---+----------+
| ltrim| rtrim| trim| lp|        rp|
+------+------+-----+---+----------+
|HELLO | HELLO|HELLO|HEL|HELLO     |
|HELLO | HELLO|HELLO|HEL|HELLO     |
+------+------+-----+---+----------+
only showing top 2 rows
```

Regular Expressions

- Probably one of the most frequently performed tasks is searching for the existence of one string in another or replacing all mentions of a string with another value.

```
from pyspark.sql.functions import regexp_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|
+--------------------+--------------------+
only showing top 2 rows
```

Regular Expressions

- Replace given characters with other characters

```
from pyspark.sql.functions import translate
df.select(translate(col("Description"), "LEET", "1337"),col("Description"))\
.show(2)
```

```
+----------------------------------+--------------------+
|translate(Description, LEET, 1337)|         Description|
+----------------------------------+--------------------+
|              WHI73 HANGING H3A...|WHITE HANGING HEA...|
|               WHI73 M37A1 1AN73RN| WHITE METAL LANTERN|
+----------------------------------+--------------------+
only showing top 2 rows
```

Regular Expressions

- Replace given characters with other characters

```
from pyspark.sql.functions import translate
df.select(translate(col("Description"), "LEET", "1337"),col("Description"))\
.show(2)
```

```
+----------------------------------+--------------------+
|translate(Description, LEET, 1337)|         Description|
+----------------------------------+--------------------+
|              WHI73 HANGING H3A...|WHITE HANGING HEA...|
|               WHI73 M37A1 1AN73RN| WHITE METAL LANTERN|
+----------------------------------+--------------------+
only showing top 2 rows
```

Regular Expressions

- Check for existence

```
from pyspark.sql.functions import instr
containsBlack = instr(col("Description"), "BLACK") >= 1
containsWhite = instr(col("Description"), "WHITE") >= 1
df.withColumn("hasSimpleColor", containsBlack | containsWhite)\
.where("hasSimpleColor")\
.select("Description").show(3, False)
```

```
+-----------------------------------+
|Description                        |
+-----------------------------------+
|WHITE HANGING HEART T-LIGHT HOLDER|
|WHITE METAL LANTERN                |
|RED WOOLLY HOTTIE WHITE HEART.     |
+-----------------------------------+
only showing top 3 rows
```

Working with Dates and Timestamps

- Dates and times are a constant challenge in programming languages and databases.

- get the current date and the current timestamps

```
from pyspark.sql.functions import current_date, current_timestamp
dateDF = spark.range(10)\
.withColumn("today", current_date())\
.withColumn("now", current_timestamp())
dateDF.createOrReplaceTempView("dateTable")
dateDF.printSchema()
```

```
root
 |-- id: long (nullable = false)
 |-- today: date (nullable = false)
 |-- now: timestamp (nullable = false)
```

Working with Dates and Timestamps

- Add and subtract five days from today

```
from pyspark.sql.functions import date_add, date_sub
dateDF.select(date_sub(col("today"), 5), date_add(col("today"), 5)).show(1)
```

```
+------------------+------------------+
|date_sub(today, 5)|date_add(today, 5)|
+------------------+------------------+
|        2023-01-12|        2023-01-22|
+------------------+------------------+
only showing top 1 row
```

Working with Dates and Timestamps

- Compute date between

```
from pyspark.sql.functions import datediff, months_between, to_date
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)
```

```
+-------------------------+
|datediff(week_ago, today)|
+-------------------------+
|                       -7|
+-------------------------+
only showing top 1 row

+--------------------------------+
|months_between(start, end, true)|
+--------------------------------+
|                     -16.67741935|
+--------------------------------+
only showing top 1 row
```

Working with Dates and Timestamps

- Precise date format to avoid null value if date is wrong

```
from pyspark.sql.functions import to_date
dateFormat = "yyyy-dd-MM"
cleanDateDF = spark.range(1).select(
to_date(lit("2017-12-11"), dateFormat).alias("date"),
to_date(lit("2017-20-12"), dateFormat).alias("date2"))

cleanDateDF.show()
```

```
+----------+----------+
|      date|     date2|
+----------+----------+
|2017-11-12|2017-12-20|
+----------+----------+
```

```
from pyspark.sql.functions import to_timestamp
cleanDateDF.select(to_timestamp(col("date"), dateFormat))\
.show()
```

```
+------------------------------+
|to_timestamp(date, yyyy-dd-MM)|
+------------------------------+
|           2017-11-12 00:00:00|
+------------------------------+
```

Working with Nulls in Data

- As a best practice, you should always use nulls to represent missing or empty data in your DataFrames
- The primary way of interacting with null values, at DataFrame scale, is to use the .na subpackage on a DataFrame
- Spark includes a function to allow you to select the first non-null value from a set of columns by using the coalesce function. In this case, there are no null values, so it simply returns the first column

```
from pyspark.sql.functions import coalesce
df.select(coalesce(col("Description"), col("CustomerId"))).show()
```

```
+---------------------------------+
|coalesce(Description, CustomerId)|
+---------------------------------+
|              WHITE HANGING HEA...|
|               WHITE METAL LANTERN|
|              CREAM CUPID HEART...|
```

Working with Nulls in Data

- Drop and fill
- The simplest function is drop, which removes rows that contain nulls. The default is to drop any row in which any value is null
- Specifying "any" as an argument drops a row if any of the values are null. Using “all” drops the row only if all values are null or NaN for that row:
- We can also apply this to certain sets of columns by passing in an array of columns

```
df.na.drop()
df.na.drop("any")
```

```
df.na.drop("all")
```

```
df.na.drop("all", subset=["StockCode", "InvoiceNo"])
```

Working with Nulls in Data

- Drop and fill
- Using the fill function, you can fill one or more columns with a set of values
- For example, to fill all null values in columns of type String, you might specify the following

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

```
fill_cols_vals = {"StockCode": 5, "Description" : "No Value"}
df.na.fill(fill_cols_vals)
```

Working with Complex Types

- Complex types can help you organize and structure your data in ways that make more sense for the problem that you are hoping to solve.
- There are three kinds of complex types:
  - Structs
  - arrays
  - maps

Working with Complex Types

- We can create a struct by wrapping a set of columns in parenthesis in a query

```
from pyspark.sql.functions import struct
complexDF = df.select(struct("Description", "InvoiceNo").alias("complex"))
complexDF.createOrReplaceTempView("complexDF")
```

```
complexDF.select("complex.Description").show(1)
complexDF.select(col("complex").getField("Description")).show(1)
```

```
+--------------------+
|         Description|
+--------------------+
|WHITE HANGING HEA...|
+--------------------+
only showing top 1 row

+--------------------+
| complex.Description|
+--------------------+
|WHITE HANGING HEA...|
+--------------------+
only showing top 1 row
```

```
complexDF.select("complex").show(1,False)
```

```
+-------------------------------------------+
|complex                                    |
+-------------------------------------------+
|{WHITE HANGING HEART T-LIGHT HOLDER, 536365}|
+-------------------------------------------+
only showing top 1 row
```

Working with Complex Types

- Arrays
- To define arrays, let's work through a use case. With our current data, our objective is to take every single word in our Description column and convert that into a row in our DataFrame. The first task is to turn our Description column into a complex type, an array
- We can also query the values of the array using Python-like syntax:

```
from pyspark.sql.functions import split
df.select(split(col("Description"), " ")).show(2,False)
```

```
+-----------------------------------------+
|split(Description,  , -1)                |
+-----------------------------------------+
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|
|[WHITE, METAL, LANTERN]                  |
+-----------------------------------------+
only showing top 2 rows
```

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

```
+------------+
|array_col[0]|
+------------+
|       WHITE|
|       WHITE|
+------------+
only showing top 2 rows
```

Working with Complex Types

- Arrays
- To define arrays, let's work through a use case. With our current data, our objective is to take every single word in our Description column and convert that into a row in our DataFrame. The first task is to turn our Description column into a complex type, an array
- We can also query the values of the array using Python-like syntax:

```
from pyspark.sql.functions import split
df.select(split(col("Description"), " ")).show(2,False)
```

```
+-----------------------------------------+
|split(Description,  , -1)                |
+-----------------------------------------+
|[WHITE, HANGING, HEART, T-LIGHT, HOLDER]|
|[WHITE, METAL, LANTERN]                  |
+-----------------------------------------+
only showing top 2 rows
```

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

```
+------------+
|array_col[0]|
+------------+
|       WHITE|
|       WHITE|
+------------+
only showing top 2 rows
```

Working with Complex Types

- Arrays
- 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.

```
from pyspark.sql.functions import split, explode
df.withColumn("splitted", split(col("Description"), " "))\
.withColumn("exploded", explode(col("splitted")))\
.select("Description", "InvoiceNo", "exploded").show(4)
```

```
+--------------------+---------+--------+
|         Description|InvoiceNo|exploded|
+--------------------+---------+--------+
|WHITE HANGING HEA...|   536365|   WHITE|
|WHITE HANGING HEA...|   536365| HANGING|
|WHITE HANGING HEA...|   536365|   HEART|
|WHITE HANGING HEA...|   536365| T-LIGHT|
+--------------------+---------+--------+
only showing top 4 rows
```

Working with Complex Types

- Maps

- Maps are created by using the map function and key-value pairs of columns. You then can select them just like you might select from an array:

```
from pyspark.sql.functions import create_map
df.select(create_map(col("Description"), col("InvoiceNo"))
.alias("complex_map"))\
.show(2, False)
```

```
+---------------------------------------------+
|complex_map                                  |
+---------------------------------------------+
|{WHITE HANGING HEART T-LIGHT HOLDER -> 536365}|
|{WHITE METAL LANTERN -> 536365}              |
+---------------------------------------------+
only showing top 2 rows
```

Working with Complex Types

- Maps
- Maps are created by using the map function and key-value pairs of columns. You then can select them just like you might select from an array:

```
from pyspark.sql.functions import create_map

df.select(create_map(col("Description"), col("InvoiceNo")).alias("complex_map"))\
.selectExpr("complex_map['WHITE METAL LANTERN']").show(2)
```

```
+--------------------------------+
|complex_map[WHITE METAL LANTERN]|
+--------------------------------+
|                            null|
|                          536365|
+--------------------------------+
only showing top 2 rows
```

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
- 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

```
from pyspark.sql.functions import get_json_object, json_tuple

jsonDF = spark.range(1).selectExpr("""
'{"myJSONKey" : {"myJSONValue" : [1, 2, 3]}}' as jsonString""")
jsonDF.select(get_json_object(col("jsonString"), "$.myJSONKey.myJSONValue[1]").alias("column"),
              json_tuple(col("jsonString"), "myJSONKey")).show(2,False)
```

```
+------+-----------------------+
|column|c0                     |
+------+-----------------------+
|2     |{"myJSONValue":[1,2,3]}|
+------+-----------------------+
```

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
- Although you can write UDFs in Scala, Python, or Java, there are performance considerations that you should be aware of.

User-Defined Functions

- Let's write a power^3 function that takes a number and raises it to a power of three:

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

```
8.0
```

- 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

User-Defined Functions

- Starting this Python process is expensive, but the real cost is in serializing the data to Python. This is costly for two reasons: it is an expensive computation, but also, after the data enters Python, Spark cannot manage the memory of the worker.
- We recommend that you write your UDFs in Scala or Java

User-Defined Functions

- First, we need to register the function to make it available as a DataFrame function

```
from pyspark.sql.functions import udf
power3udf = udf(power3)
```

- Then, we can use it in our DataFrame code:

```
from pyspark.sql.functions import col
udfExampleDF.select(power3udf(col("num"))).show(4)
```

```
+-----------+
|power3(num)|
+-----------+
|          0|
|          1|
|          8|
|         27|
+-----------+
```

- Register the return type

```
from pyspark.sql.types import IntegerType, DoubleType
spark.udf.register("power3py", power3, IntegerType())
udfExampleDF.selectExpr("power3py(num)").show()
```

Chapter 7. Aggregations

What is aggregations

- Aggregating is the act of collecting something together and is a cornerstone of big data analytics
- In an aggregation, you will specify a key or grouping and an aggregation function that specifies how you should transform one or more columns.
- This function must produce one result for each group, given multiple input values

```python
# Dataset for aggreagation chapter
df = spark.read.format("csv")\
.option("header", "true")\
.option("inferSchema", "true")\
.load("all/*.csv")\
.coalesce(5)
df.cache()
df.createOrReplaceTempView("dfTable")
df.show(2)
```

```
+---------+---------+--------------------+--------+
|InvoiceNo|StockCode|         Description|Quantity|
+---------+---------+--------------------+--------+
|   536365|   85123A|WHITE HANGING HEA...|       6|
|   536365|    71053| WHITE METAL LANTERN|       6|
+---------+---------+--------------------+--------+
only showing top 2 rows
```

count

- The simplest example is the count method

```
# count method
df.count()
```

- What is the difference between df.count() and df.select(count('column'))

```
from pyspark.sql.functions import count
df.select(count("StockCode")).show()
```

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

countDistinct , approx_count_distinct

- Number of unique groups

```
from pyspark.sql.functions import countDistinct
df.select(countDistinct("StockCode")).show() # 4070
```

```
+-------------------------+
|count(DISTINCT StockCode)|
+-------------------------+
|                     4070|
+-------------------------+
```

```
from pyspark.sql.functions import approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show() # 3364
```

```
+--------------------------------+
|approx_count_distinct(StockCode)|
+--------------------------------+
|                            3364|
+--------------------------------+
```

countDistinct , approx_count_distinct

- Number of unique groups

```
from pyspark.sql.functions import countDistinct
df.select(countDistinct("StockCode")).show() # 4070
```

```
+-------------------------+
|count(DISTINCT StockCode)|
+-------------------------+
|                     4070|
+-------------------------+
```

```
from pyspark.sql.functions import approx_count_distinct
df.select(approx_count_distinct("StockCode", 0.1)).show() # 3364
```

```
+--------------------------------+
|approx_count_distinct(StockCode)|
+--------------------------------+
|                            3364|
+--------------------------------+
```

First and last , mix and max

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

```
+----------------+---------------+
|first(StockCode)|last(StockCode)|
+----------------+---------------+
|           23084|          22168|
+----------------+---------------+
```

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

```
+-------------+-------------+
|min(Quantity)|max(Quantity)|
+-------------+-------------+
|       -80995|        80995|
+-------------+-------------+
```

Sum and sum_disctinct

- Number of unique groups

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

```
+-------------+
|sum(Quantity)|
+-------------+
|      5176450|
+-------------+
```

```
from pyspark.sql.functions import sum_distinct
df.select(sum_distinct ("Quantity")).show() # 29310
```

```
+----------------------+
|sum(DISTINCT Quantity)|
+----------------------+
|                 29310|
+----------------------+
```

```
d = spark.createDataFrame([
    { "q" :10 },
    { "q" :10 },
    { "q" :10 }
])

d.select(sum("q")).show()
d.select(sum_distinct("q")).show()
```

```
+------+
|sum(q)|
+------+
|    30|
+------+

+---------------+
|sum(DISTINCT q)|
+---------------+
|             10|
+---------------+
```

avg

```python
from pyspark.sql.functions import sum, count, avg, expr
df.select(
count("Quantity").alias("total_transactions"),
sum("Quantity").alias("total_purchases"),
avg("Quantity").alias("avg_purchases"),
expr("mean(Quantity)").alias("mean_purchases"))\
.selectExpr(
"total_purchases/total_transactions",
"avg_purchases",
"mean_purchases").show()
```

```
+-------------------------------------+----------------+----------------+
|(total_purchases / total_transactions)|   avg_purchases|  mean_purchases|
+-------------------------------------+----------------+----------------+
|                      9.55224954743324|9.55224954743324|9.55224954743324|
+-------------------------------------+----------------+----------------+
```

Grouping

- We will group by each unique invoice number and get the count of items on that invoice

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

```
+---------+----------+-----+
|InvoiceNo|CustomerId|count|
+---------+----------+-----+
|   580657|   14696.0|   20|
|   581474|   12748.0|   24|
|   576641|   17549.0|    4|
```

Grouping with Expressions

- We specify it as within agg.

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

```
+---------+----+---------------+
|InvoiceNo|quan|count(Quantity)|
+---------+----+---------------+
|   574966|   8|              8|
|   575091|  38|             38|
|   578057|  28|             28|
|   537252|   1|              1|
```

Grouping with Maps

- We specify it as within agg.

```
df.groupBy("InvoiceNo").agg(expr("avg(Quantity)"),sum("Quantity"))\
.show()
```

```
+---------+------------------+-------------+
|InvoiceNo|     avg(Quantity)|sum(Quantity)|
+---------+------------------+-------------+
|   574966|               6.0|           48|
|   575091|11.552631578947368|          439|
|   578057| 4.607142857142857|          129|
|   537252|              31.0|           31|
|   578459|              28.0|          224|
```

Window Functions

- A group-by takes data, and every row can go only into one grouping
- 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
- Ranking functions, analytic functions, and aggregate functions

Visualizing window functions

Window Functions

- To demonstrate, we will add a date column that will convert our invoice date into a column that contains only date information

```
from pyspark.sql.functions import col, to_date
dfWithDate = df.withColumn("date", to_date(col("InvoiceDate"), "MM/d/yyyy H:mm"))
dfWithDate.createOrReplaceTempView("dfWithDate")
dfWithDate.schema
```

```
StructType([StructField('InvoiceNo', StringType(), True), StructField('StockCode', StringType(), True),
StructField('Description', StringType(), True), StructField('Quantity', IntegerType(), True),
StructField('InvoiceDate', TimestampType(), True), StructField('UnitPrice', DoubleType(), True),
StructField('CustomerID', DoubleType(), True), StructField('Country', StringType(), True), StructField('date',
DateType(), True)])
```

Window Functions

- The first step to a window function is to create a window specification

```
from pyspark.sql.window import Window
from pyspark.sql.functions import desc
windowSpec = Window\
.partitionBy("CustomerId", "date")\
.orderBy(desc("Quantity"))\
.rowsBetween(Window.unboundedPreceding, Window.currentRow)
```

- Now we want to use an aggregation function to learn more about each specific customer.

```
from pyspark.sql.functions import max
maxPurchaseQuantity = max(col("Quantity"))\
.over(windowSpec)
```

Window Functions

- Before doing so, though, we will create the purchase quantity rank. To do that we use the dense_rank function to determine which date had the maximum purchase quantity for every customer
- We use dense_rank as opposed to rank to avoid gaps in the ranking sequence when there are tied values (or in our case, duplicate rows):

```
from pyspark.sql.functions import dense_rank, rank
purchaseDenseRank = dense_rank().over(windowSpec)
purchaseRank = rank().over(windowSpec)
```

Window Functions

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

```
from pyspark.sql.functions import col
dfWithDate.where("CustomerId IS NOT NULL")\
.orderBy("CustomerId")\
.select(
col("CustomerId").alias("id"),
col("date"),
col("Quantity"),
purchaseRank.alias("qRank"),
purchaseDenseRank.alias("qDenseRank"),
maxPurchaseQuantity.alias("maxPurchaseQ"))\
.show()
```

id	date	Quantity	qRank	qDenseRank	maxPurchaseQ
12346.0	2011-01-18	74215	1	1	74215
12346.0	2011-01-18	-74215	2	2	74215
12347.0	2010-12-07	36	1	1	36
12347.0	2010-12-07	30	2	2	36
12347.0	2010-12-07	24	3	3	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	12	4	4	36
12347.0	2010-12-07	6	17	5	36
12347.0	2010-12-07	6	17	5	36

Pivot

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

```
pivoted = dfWithDate.groupBy("date").pivot("Country").sum()
pivoted.columns
```

```
'Israel_sum(CustomerID)',
'Italy_sum(Quantity)',
'Italy_sum(UnitPrice)',
'Italy_sum(CustomerID)',
'Japan_sum(Quantity)',
'Japan_sum(UnitPrice)',
'Japan_sum(CustomerID)',
'Lebanon_sum(Quantity)',
'Lebanon_sum(UnitPrice)',
'Lebanon_sum(CustomerID)',
'Lithuania_sum(Quantity)',
'Lithuania_sum(UnitPrice)',
'Lithuania_sum(CustomerID)',
```

```
pivoted.where("date > '2011-12-05'")\
.select("date" ,"`USA_sum(Quantity)`")\
.show()
```

```
+----------+-----------------+
|      date|USA_sum(Quantity)|
+----------+-----------------+
|2011-12-06|             null|
|2011-12-09|             null|
|2011-12-08|             -196|
|2011-12-07|             null|
+----------+-----------------+
```

Pivot

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

```
pivoted = dfWithDate.groupBy("date").pivot("Country").sum()
pivoted.columns
```

```
'Israel_sum(CustomerID)',
'Italy_sum(Quantity)',
'Italy_sum(UnitPrice)',
'Italy_sum(CustomerID)',
'Japan_sum(Quantity)',
'Japan_sum(UnitPrice)',
'Japan_sum(CustomerID)',
'Lebanon_sum(Quantity)',
'Lebanon_sum(UnitPrice)',
'Lebanon_sum(CustomerID)',
'Lithuania_sum(Quantity)',
'Lithuania_sum(UnitPrice)',
'Lithuania_sum(CustomerID)',
```

```
pivoted.where("date > '2011-12-05'")\
.select("date" ,"`USA_sum(Quantity)`")\
.show()
```

```
+----------+-----------------+
|      date|USA_sum(Quantity)|
+----------+-----------------+
|2011-12-06|             null|
|2011-12-09|             null|
|2011-12-08|             -196|
|2011-12-07|             null|
+----------+-----------------+
```

Chapter 8. Joins

Join Expressions

- A join brings together two sets of data, the left and the right, by comparing the value of one or more keys of the left and right

- Inner joins (keep rows with keys that exist in the left and right datasets)
- Outer joins (keep rows with keys in either the left or right datasets)
- Left outer joins (keep rows with keys in the left dataset)
- Right outer joins (keep rows with keys in the right dataset)
- Left semi joins (keep the rows in the left, and only the left, dataset where the key appears in the right dataset)
- Left anti joins (keep the rows in the left, and only the left, dataset where they do not appear in the right dataset)
- Natural joins (perform a join by implicitly matching the columns between the two datasets with the same names)
- Cross (or Cartesian) joins (match every row in the left dataset with every row in the right dataset)

Join Expressions

- Quick examples

```
person = spark.createDataFrame([
(0, "Bill Chambers", 0, [100]),
(1, "Matei Zaharia", 1, [500, 250, 100]),
(2, "Michael Armbrust", 1, [250, 100])])\
.toDF("id", "name", "graduate_program", "spark_status")
graduateProgram = spark.createDataFrame([
(0, "Masters", "School of Information", "UC Berkeley"),
(2, "Masters", "EECS", "UC Berkeley"),
(1, "Ph.D.", "EECS", "UC Berkeley")])\
.toDF("id", "degree", "department", "school")
sparkStatus = spark.createDataFrame([
(500, "Vice President"),
(250, "PMC Member"),
(100, "Contributor")])\
.toDF("id", "status")
```

```
person.createOrReplaceTempView("person")
graduateProgram.createOrReplaceTempView("graduateProgram")
sparkStatus.createOrReplaceTempView("sparkStatus")
```

Inner Joins

- Inner joins evaluate the keys in both of the DataFrames or tables and include (and join together) only the rows that evaluate to true
- Inner joins are the default join, so we just need to specify our left DataFrame and join the right in the JOIN expression

```
joinExpression = person["graduate_program"] == graduateProgram['id']
person.join(graduateProgram, joinExpression).show()
```

```
+---+----------------+----------------+---------------+---+-------+--------------------+-----------+
| id|            name|graduate_program|   spark_status| id| degree|          department|     school|
+---+----------------+----------------+---------------+---+-------+--------------------+-----------+
|  0|   Bill Chambers|               0|          [100]|  0|Masters|School of Informa...|UC Berkeley|
|  1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|                EECS|UC Berkeley|
|  2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|                EECS|UC Berkeley|
+---+----------------+----------------+---------------+---+-------+--------------------+-----------+
```

Inner Joins

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

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

```
+---+----------------+----------------+---------------+---+-------+--------------------+-----------+
| id|            name|graduate_program|   spark_status| id| degree|          department|     school|
+---+----------------+----------------+---------------+---+-------+--------------------+-----------+
|  0|   Bill Chambers|               0|          [100]|  0|Masters|School of Informa...|UC Berkeley|
|  1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|                EECS|UC Berkeley|
|  2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|                EECS|UC Berkeley|
+---+----------------+----------------+---------------+---+-------+--------------------+-----------+
```

(Full) Outer Joins

- Outer joins evaluate the keys in both of the DataFrames or tables and includes (and joins together) the rows that evaluate to true or false. 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()
```

```
+----+----------------+----------------+---------------+---+-------+--------------------+-----------+
|  id|            name|graduate_program|   spark_status| id| degree|          department|     school|
+----+----------------+----------------+---------------+---+-------+--------------------+-----------+
|   0|   Bill Chambers|               0|          [100]|  0|Masters|School of Informa...|UC Berkeley|
|   1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|                EECS|UC Berkeley|
|   2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|                EECS|UC Berkeley|
|null|            null|            null|           null|  2|Masters|                EECS|UC Berkeley|
+----+----------------+----------------+---------------+---+-------+--------------------+-----------+
```

Right Outer Joins

- Right outer joins evaluate the keys in both of the DataFrames or tables and includes all rows from the right DataFrame as well as any rows in the left DataFrame that have a match in the right DataFrame. If there is no equivalent row in the left DataFrame, Spark will insert null

```
#-- in SQL
#SELECT * FROM person RIGHT OUTER JOIN graduateProgram
#ON person.graduate_program = graduateProgram.id
joinType = "right_outer"
person.join(graduateProgram, joinExpression, joinType).show()
```

```
+----+----------------+----------------+---------------+---+-------+--------------------+-----------+
|  id|            name|graduate_program|   spark_status| id| degree|          department|     school|
+----+----------------+----------------+---------------+---+-------+--------------------+-----------+
|   0|   Bill Chambers|               0|          [100]|  0|Masters|School of Informa...|UC Berkeley|
|   2|Michael Armbrust|               1|     [250, 100]|  1|  Ph.D.|                EECS|UC Berkeley|
|   1|   Matei Zaharia|               1|[500, 250, 100]|  1|  Ph.D.|                EECS|UC Berkeley|
|null|            null|            null|           null|  2|Masters|                EECS|UC Berkeley|
+----+----------------+----------------+---------------+---+-------+--------------------+-----------+
```

Left Outer Joins

- Left outer joins evaluate the keys in both of the DataFrames or tables and includes all rows from the left DataFrame as well as any rows in the right DataFrame that have a match in 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()
```

```
+---+-------+--------------------+-----------+----+----------------+----------------+---------------+
| id| degree|          department|     school|  id|            name|graduate_program|   spark_status|
+---+-------+--------------------+-----------+----+----------------+----------------+---------------+
|  0|Masters|School of Informa...|UC Berkeley|   0|   Bill Chambers|               0|          [100]|
|  1|  Ph.D.|                EECS|UC Berkeley|   2|Michael Armbrust|               1|     [250, 100]|
|  1|  Ph.D.|                EECS|UC Berkeley|   1|   Matei Zaharia|               1|[500, 250, 100]|
|  2|Masters|                EECS|UC Berkeley|null|            null|            null|           null|
+---+-------+--------------------+-----------+----+----------------+----------------+---------------+
```

Left Semi Joins

- Semi joins are a bit of a departure from the other joins. They do not actually include any values from the right DataFrame. They only compare values to see if the value exists in the second DataFrame.
- Think of left semi joins as filters on a DataFrame, as opposed to the function of a conventional join

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

```
+---+-------+--------------------+-----------+
| id| degree|          department|     school|
+---+-------+--------------------+-----------+
|  0|Masters|School of Informa...|UC Berkeley|
|  1|  Ph.D.|                EECS|UC Berkeley|
+---+-------+--------------------+-----------+
```

Left Anti Joins

- Left anti joins are the opposite of left semi joins
- They only compare values to see if the value exists in the second DataFrame. However, rather than keeping the values that exist in the second DataFrame, they keep only the values that do not have a corresponding key in the second DataFrame.

```
joinType = "left_anti"
graduateProgram.join(person, joinExpression, joinType).show()
#-- in SQL
#SELECT * FROM graduateProgram LEFT ANTI JOIN person
#ON graduateProgram.id = person.graduate_program
```

```
+---+-------+----------+-----------+
| id| degree|department|     school|
+---+-------+----------+-----------+
|  2|Masters|      EECS|UC Berkeley|
+---+-------+----------+-----------+
```

Natural Joins

- Natural joins make implicit guesses at the columns on which you would like to join. It finds matching columns and returns the results. Left, right, and outer natural joins are all supported
- Implicit is always dangerous! The following query will give us incorrect results because the two DataFrames/tables share a column name (id), but it means different things in the datasets. You should always use this join with caution.

```
spark.sql(
  """SELECT * FROM graduateProgram NATURAL JOIN person""").show()
```

```
+---+-------+--------------------+----------+---------------+----------------+---------------+
| id| degree|          department|    school|           name|graduate_program|   spark_status|
+---+-------+--------------------+----------+---------------+----------------+---------------+
|  0|Masters|School of Informa...|UC Berkeley|   Bill Chambers|               0|          [100]|
|  1|  Ph.D.|                EECS|UC Berkeley|   Matei Zaharia|               1|[500, 250, 100]|
|  2|Masters|                EECS|UC Berkeley|Michael Armbrust|               1|     [250, 100]|
+---+-------+--------------------+----------+---------------+----------------+---------------+
```

Cross (Cartesian) Joins

- Cross-joins in simplest terms are inner joins that do not specify a predicate
- This will cause an absolute explosion in the number of rows contained in the resulting DataFrame
- If you have 1,000 rows in each DataFrame, the crossjoin of these will result in 1,000,000 (1,000 x 1,000) rows.

```
graduateProgram.crossJoin(person).show()
#-- in SQL
#SELECT * FROM graduateProgram CROSS JOIN person
#ON graduateProgram.id = person.graduate_program
```

```
+---+-------+--------------------+----------+---+---------------+---------------+--------------+
| id| degree|          department|    school| id|           name|graduate_program|  spark_status|
+---+-------+--------------------+----------+---+---------------+---------------+--------------+
|  0|Masters|School of Informa...|UC Berkeley|  0|   Bill Chambers|              0|         [100]|
|  0|Masters|School of Informa...|UC Berkeley|  1|   Matei Zaharia|              1|[500, 250, 100]|
|  0|Masters|School of Informa...|UC Berkeley|  2|Michael Armbrust|              1|     [250, 100]|
|  2|Masters|                EECS|UC Berkeley|  0|   Bill Chambers|              0|         [100]|
|  1|  Ph.D.|                EECS|UC Berkeley|  0|   Bill Chambers|              0|         [100]|
|  2|Masters|                EECS|UC Berkeley|  1|   Matei Zaharia|              1|[500, 250, 100]|
|  2|Masters|                EECS|UC Berkeley|  2|Michael Armbrust|              1|     [250, 100]|
|  1|  Ph.D.|                EECS|UC Berkeley|  1|   Matei Zaharia|              1|[500, 250, 100]|
|  1|  Ph.D.|                EECS|UC Berkeley|  2|Michael Armbrust|              1|     [250, 100]|
+---+-------+--------------------+----------+---+---------------+---------------+--------------+
```

Joins on Complex Types

- Even though this might seem like a challenge, it's actually not. Any expression is a valid join expression, assuming that it returns a Boolean

```
from pyspark.sql.functions import expr
person.withColumnRenamed("id", "personId")\
.join(sparkStatus, expr("array_contains(spark_status, id)")).show()
```

```
+--------+----------------+----------------+---------------+---+--------------+
|personId|            name|graduate_program|   spark_status| id|        status|
+--------+----------------+----------------+---------------+---+--------------+
|       0|   Bill Chambers|               0|          [100]|100|   Contributor|
|       1|   Matei Zaharia|               1|[500, 250, 100]|500|Vice President|
|       1|   Matei Zaharia|               1|[500, 250, 100]|250|    PMC Member|
|       1|   Matei Zaharia|               1|[500, 250, 100]|100|   Contributor|
|       2|Michael Armbrust|               1|     [250, 100]|250|    PMC Member|
|       2|Michael Armbrust|               1|     [250, 100]|100|   Contributor|
+--------+----------------+----------------+---------------+---+--------------+
```

Handling Duplicate Column Names

- The join expression that you specify does not remove one key from one of the input DataFrames and the keys have the same column name
- Two columns on which you are not performing the join have the same name

```
gradProgramDupe = graduateProgram.withColumnRenamed("id", "graduate_program")
joinExpr = gradProgramDupe["graduate_program"] == person["graduate_program"]
person.join(gradProgramDupe, joinExpr).show()
```

```
+---+----------------+----------------+---------------+----------------+-------+
| id|            name|graduate_program|   spark_status|graduate_program| degree|
+---+----------------+----------------+---------------+----------------+-------+
|  0|   Bill Chambers|               0|          [100]|               0|Masters|
|  1|   Matei Zaharia|               1|[500, 250, 100]|               1|  Ph.D.|
|  2|Michael Armbrust|               1|     [250, 100]|               1|  Ph.D.|
+---+----------------+----------------+---------------+----------------+-------+
```

Handling Duplicate Column Names

- The join expression that you specify does not remove one key from one of the input DataFrames and the keys have the same column name
- Two columns on which you are not performing the join have the same name

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

```
---------------------------------------------------------------------------
AnalysisException                         Traceback (most recent call last)
<ipython-input-38-27dae62d059b> in <module>
----> 1 person.join(gradProgramDupe, joinExpr).select("graduate_program").show()

                              2 frames
/usr/local/lib/python3.8/dist-packages/pyspark/sql/utils.py in deco(*a, **kw)
    194                 # Hide where the exception came from that shows a non-Pythonic
    195                 # JVM exception message.
--> 196                 raise converted from None
    197             else:
    198                 raise

AnalysisException: Reference 'graduate_program' is ambiguous, could be: graduate_program, graduate_program.
```

Handling Duplicate Column Names

- Approach 1: Different join expression
- When you have two keys that have the same name, probably the easiest fix is to change the join expression from a Boolean expression to a string or sequence. This automatically removes one of the columns for you during the join:

```
person.join(gradProgramDupe,"graduate_program") \
.select("graduate_program").show()
```

```
+----------------+
|graduate_program|
+----------------+
|               0|
|               1|
|               1|
+----------------+
```

Handling Duplicate Column Names

- Approach 2: Dropping the column after the join
- Another approach is to drop the offending column after the join

```
person.join(gradProgramDupe, joinExpr)\
.drop(person["graduate_program"])\
.select("graduate_program").show()
```

```
+----------------+
|graduate_program|
+----------------+
|               0|
|               1|
|               1|
+----------------+
```

Handling Duplicate Column Names

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

```
gradProgram3 = graduateProgram.withColumnRenamed("id", "grad_id")
joinExpr = person["graduate_program"] == gradProgram3["grad_id"]
person.join(gradProgram3, joinExpr).show()
```

```
+---+----------------+----------------+---------------+-------+-------+--------------------+-----------+
| id|            name|graduate_program|   spark_status|grad_id| degree|          department|     school|
+---+----------------+----------------+---------------+-------+-------+--------------------+-----------+
|  0|   Bill Chambers|               0|          [100]|      0|Masters|School of Informa...|UC Berkeley|
|  1|   Matei Zaharia|               1|[500, 250, 100]|      1|  Ph.D.|                EECS|UC Berkeley|
|  2|Michael Armbrust|               1|     [250, 100]|      1|  Ph.D.|                EECS|UC Berkeley|
+---+----------------+----------------+---------------+-------+-------+--------------------+-----------+
```

How Spark Performs Joins

- Spark approaches cluster communication in two different ways during joins. It either incurs a shuffle join, which results in an all-to-all communication or a broadcast join

- Big table–to–big table

  - In a shuffle join, every node talks to every other node and they share data according to which node has a certain key or set of keys (on which you are joining). These joins are expensive because the network can become congested with traffic, especially if your data is not partitioned well.

Partition in Table 1 (big table)

Partition in Table 2 (big table)

How Spark Performs Joins

- Big table–to–small table

- When the table is small enough to fit into the memory of a single worker node, with some breathing room of course, we can optimize our join.
- What this means is that we will replicate our small DataFrame onto every worker node in the cluster (be it located on one machine or many).
- However, what this does is prevent us from performing the all-to-all communication during the entire join process

How Spark Performs Joins

- Big table–to–Big table

```
joinExpr = person["graduate_program"] == graduateProgram["id"]
person.join(graduateProgram, joinExpr).explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- SortMergeJoin [graduate_program#590L], [id#604L], Inner
   :- Sort [graduate_program#590L ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(graduate_program#590L, 200), ENSURE_REQUIREMENTS, [plan_id=2777]
   :     +- Project [_1#580L AS id#588L, _2#581 AS name#589, _3#582L AS graduate_program#590L, _4#583 AS spar
   :        +- Filter isnotnull(_3#582L)
   :           +- Scan ExistingRDD[_1#580L,_2#581,_3#582L,_4#583]
   +- Sort [id#604L ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(id#604L, 200), ENSURE_REQUIREMENTS, [plan_id=2778]
         +- Project [_1#596L AS id#604L, _2#597 AS degree#605, _3#598 AS department#606, _4#599 AS school#607
            +- Filter isnotnull(_1#596L)
               +- Scan ExistingRDD[_1#596L,_2#597,_3#598,_4#599]
```

How Spark Performs Joins

- Big table–to–small table

```
from pyspark.sql.functions import broadcast
joinExpr = person["graduate_program"] == graduateProgram["id"]
person.join(broadcast(graduateProgram), joinExpr).explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- BroadcastHashJoin [graduate_program#590L], [id#604L], Inner, BuildRight, false
   :- Project [_1#580L AS id#588L, _2#581 AS name#589, _3#582L AS graduate_program#59
   :  +- Filter isnotnull(_3#582L)
   :     +- Scan ExistingRDD[_1#580L,_2#581,_3#582L,_4#583]
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, true]),fals
      +- Project [_1#596L AS id#604L, _2#597 AS degree#605, _3#598 AS department#606,
         +- Filter isnotnull(_1#596L)
            +- Scan ExistingRDD[_1#596L,_2#597,_3#598,_4#599]
```

How Spark Performs Joins

- Little table–to–little table
- When performing joins with small tables, it’s usually best to let Spark decide how to join them. You can always force a broadcast join if you’re noticing strange behavior
- One thing we did not mention but is important to consider is if you partition your data correctly prior to a join

Chapter 9. Data Sources

How Spark Performs Joins

- Following are Spark's core data sources
  - CSV
  - JSON
  - Parquet
  - ORC
  - JDBC/ODBC connections
  - Plain-text files

- Spark has numerous community-created data sources
  - Cassandra
  - HBase
  - MongoDB
  - AWS Redshift
  - XML
  - And many, many others
  - ……

Read API Structure

- The core structure for reading data is as follows:

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

- format is optional because by default Spark will use the Parquet format.
- schema is optional if the data source provides a schema or if you intend to use schema inference.

Basics of Reading Data

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

```
spark.read
```

```
<pyspark.sql.readwriter.DataFrameReader at 0x7f2ec1495c10>
```

- After we have a DataFrame reader, we specify several values:
  - The format
  - The schema
  - The read mode
  - A series of options

```
spark.read.format("csv")
  .option("mode", "FAILFAST")
  .option("inferSchema", "true")
  .option("path", "path/to/file(s)")
  .schema(someSchema)
  .load()
```

Read modes

- Read modes specify what will happen when Spark does come across malformed records

Read mode	Description
`permissive`	Sets all fields to `null` when it encounters a corrupted record and places all corrupted records in a string column called `_corrupt_record`
`dropMalformed`	Drops the row that contains malformed records
`failFast`	Fails immediately upon encountering malformed records

The default is `permissive`.

Write API Structure

- The core structure for writing data is as follows:

```
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

- Because we always need to write out some given data source, we access the DataFrameWriter on a per-DataFrame basis via the write attribute:

```
df = spark.createDataFrame([("0" , "2")])

df.write
```

```
<pyspark.sql.readwriter.DataFrameWriter at 0x7f2eb7c5a9d0>
```

Write API Structure

- Save modes
- The default is errorIfExists.

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

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

CSV Files

- CSV stands for commma-separated values. This is a common text file format in which each line represents a single record, and commas separate each field within a record

```python
# Read csv

df = spark.read.format("csv")\
.option("header", "true")\
.option("mode", "FAILFAST")\
.option("inferSchema", "true")\
.load("2010-summary.csv")

df.show(1)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|    1|
+-----------------+-------------------+-----+
only showing top 1 row
```

CSV Files

- Define manual schema

```
# Precise manual schema
from pyspark.sql.types import StructType, StructField,StringType, LongType
myManualSchema = StructType([
 StructField("DEST_COUNTRY_NAME", StringType(), True),
 StructField("ORIGIN_COUNTRY_NAME", StringType(), True),
 StructField("count", LongType(), True)
])

# Read csv with manual schema

df = spark.read.format("csv")\
.option("header", "true")\
.option("mode", "FAILFAST")\
.schema(myManualSchema)\
.load("2010-summary.csv")

df.show(1)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|    1|
+-----------------+-------------------+-----+
only showing top 1 row
```

CSV Files

- Writing CSV Files

- For instance, we can take our CSV file and write it out as a TSV file quite easily:

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|    1|
+-----------------+-------------------+-----+
only showing top 1 row
```

```
df.write\
.format("csv")\
.mode("overwrite")\
.option("sep", "\t")\
.save("my-tsv-file.tsv")
```

JSON Files

- Those coming from the world of JavaScript are likely familiar with JavaScript Object Notation, or JSON, as it's commonly called
- The line-delimited versus multiline trade-off is controlled by a single option: multiline
- When you set this option to true, you can read an entire file as one json object and Spark will go through the work of parsing that into a DataFrame. Line-delimited JSON is actually a much more stable format because it allows you to append to a file with a new record

Reading JSON Files

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

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|    1|
|    United States|            Ireland|  264|
|    United States|              India|   69|
|            Egypt|      United States|   24|
|Equatorial Guinea|      United States|    1|
+-----------------+-------------------+-----+
only showing top 5 rows
```

Writing JSON Files

- 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

```
df.write.format("json")\
.mode("overwrite")\
.save("my-json-file.json")
```

Parquet Files

- Parquet is an open source column-oriented data store that provides a variety of storage optimizations, especially for analytics workloads.

- It provides columnar compression, which saves storage space and allows for reading individual columns instead of entire files. It is a file format that works exceptionally well with Apache Spark and is in fact the default file format

- We recommend writing data out to Parquet for long-term storage because reading from a Parquet file will always be more efficient than JSON or CSV

- Another advantage of Parquet is that it supports complex types. This means that if your column is an array (which would fail with a CSV file, for example), map, or struct, you'll still be able to read and write that file without issue.

Reading Parquet Files and writing parquet files

```
spark.read.format("parquet")\
.load("2010-summary.parquet").show(5)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|    1|
|    United States|            Ireland|  264|
|    United States|              India|   69|
|            Egypt|      United States|   24|
|Equatorial Guinea|      United States|    1|
+-----------------+-------------------+-----+
only showing top 5 rows
```

```
df.write.format("parquet")\
.mode("overwrite")\
.save("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 Files,Writing Orc Files

```
spark.read.format("orc")\
.load("2010-summary.orc")\
.show(5)
```

```
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
|    United States|            Romania|    1|
|    United States|            Ireland|  264|
|    United States|              India|   69|
|            Egypt|      United States|   24|
|Equatorial Guinea|      United States|    1|
+-----------------+-------------------+-----+
only showing top 5 rows
```

```
df.write.format("orc")\
.mode("overwrite")\
.save("my-json-file.orc")
```

SQL Databases

- SQL datasources are one of the more powerful connectors because there are a variety of systems to which you can connect (as long as that system speaks SQL).

- For instance you can connect to a MySQL database, a PostgreSQL database, or an Oracle database. You also can connect to SQLite, which is what we'll do in this example

SQL Databases

- SQLite

```
from pyspark.sql import SparkSession


# Create a Spark Session
spark = SparkSession.builder.master("local[*]")\
.config("spark.jars", "sqlite-jdbc-3.20.1.jar")\
.getOrCreate()
# Check Spark Session Information
spark
```

```
driver = "org.sqlite.JDBC"
path = "my-sqlite.db"
url = "jdbc:sqlite:" + path
tablename = "flight_info"
```

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

dbDataFrame.show(2)
```

SQL Databases

- User/Password

```
# 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()
```

- Rather than specifying a table name, you just specify a SQL query

```
# in Python
pushdownQuery = """(SELECT DISTINCT(DEST_COUNTRY_NAME) FROM flight_info)
  AS flight_info"""
dbDataFrame = spark.read.format("jdbc")\
  .option("url", url).option("dbtable", pushdownQuery).option("driver",  driver)\
  .load()
```

SQL Databases

- Reading from databases in parallel

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

- Writing to SQL Databases

```
# in Python
newPath = "jdbc:sqlite://tmp/my-sqlite.db"
csvFile.write.jdbc(newPath, tablename, mode="overwrite", 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.

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

+--------------------+
|                rows|
+--------------------+
|[DEST_COUNTRY_NAM...|
```

- Writing Text Files

```
# in Python

csvFile.limit(10).select("DEST_COUNTRY_NAME", "count")\
  .write.partitionBy("count").text("/tmp/five-csv-files2py.csv")
```

Text Files

- Spark also allows you to read in plain-text files. Each line in the file becomes a record in the DataFrame.

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

+--------------------+
|                rows|
+--------------------+
|[DEST_COUNTRY_NAM...|
```

- Writing Text Files

```
# in Python

csvFile.limit(10).select("DEST_COUNTRY_NAME", "count")\
  .write.partitionBy("count").text("/tmp/five-csv-files2py.csv")
```

Reading Data in Parallel and Writing 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

Reading Data in Parallel and 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

```
df.repartition(5).write.format("csv").save("multiple.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.

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

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

```
numberBuckets = 10
columnToBucketBy = "count"
df.write.format("parquet").mode("overwrite")\
.bucketBy(numberBuckets, columnToBucketBy)\
.saveAsTable("bucketedFiles")
```

Chapter 10. Spark SQL

Spark SQL

- Spark SQL is arguably one of the most important and powerful features in Spark
- Run SQL queries against views or tables organized into databases.
- This integrates directly into the DataFrame and Dataset API

- SQL or Structured Query Language is a domain-specific language for expressing relational operations over data
- It is used in all relational databases, and many “NoSQL” databases create their SQL dialect in order to make working with their databases easier

Big Data and SQL: Apache Hive

- Before Spark's rise, Hive was the de facto big data SQL access layer.
- Originally developed at Facebook, Hive became an incredibly popular tool across industry for performing SQL operations on big data
- In many ways it helped propel Hadoop into different industries because analysts could run SQL queries

Big Data and SQL: Spark SQL

- With the release of Spark 2.0, its authors created a superset of Hive's support, writing a native SQL parser that supports both ANSI-SQL as well as HiveQL queries

- The power of Spark SQL derives from several key facts: SQL analysts can now take advantage of Spark's computation abilities by plugging into the Thrift Server or Spark's SQL interface

- 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
- With Spark SQL, you can connect to your Hive metastore (if you already have one) and access table metadata to reduce file listing when accessing information

Spark SQL CLI

- The Spark SQL CLI is a convenient tool with which you can make basic Spark SQL queries in local mode from the command line.
- To start the Spark SQL CLI, run the following in the Spark directory
- Note that the Spark SQL CLI cannot communicate with the Thrift JDBC server

```
./bin/spark-sql
```

You configure Hive by placing your hive-site.xml, core-site.xml, and hdfs-site.xml files in conf/. For a complete list of all available options, you can run ./bin/spark-sql --help.

Spark's Programmatic SQL Interface

- You can do this via the method sql on the SparkSession object. This returns a DataFrame

```
spark.sql("SELECT 1 + 1 as column").show()
```

```
+------+
|column|
+------+
|     2|
+------+
```

- 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')""")
```

- Dataframe <=> SQL

Spark's Programmatic SQL Interface

- Dataframe <=> SQL

```
spark.read.json("2015-summary.json")\
.createOrReplaceTempView("some_sql_view") # DF => SQL

spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count)
FROM some_sql_view GROUP BY DEST_COUNTRY_NAME
""")\
.where("DEST_COUNTRY_NAME like 'S%'").where("`sum(count)` > 10")\
.count() # SQL => DF
```

Creating Tables

- You can create tables from a variety of sources. Something fairly unique to Spark is the capability of reusing the entire Data Source API within SQL.

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

- Creating External Tables : Spark will manage the table's metadata; however, the files are not managed by Spark at a

```
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/'
```

Dropping Tables

- You cannot delete tables: you can only “drop” them. You can drop a table by using the DROP keyword. If you drop a managed table (e.g., flights_csv), both the data and the table definition will be removed:

```
DROP TABLE flights_csv;
```

```
DROP TABLE IF EXISTS flights_csv;
```

- 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

Databases

- Databases are a tool for organizing tables. As mentioned earlier, if you do not define one, Spark will use the default database
- Any SQL statements that you run from within Spark (including DataFrame commands) execute within the context of a database

```
CREATE DATABASE some_db
```

```
USE some_db
```

```
SELECT * FROM default.flights
```

Part III. Low-Level APIs

Chapter 12. Resilient Distributed Datasets (RDDs)

What Are the Low-Level APIs

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

- When to Use the Low-Level APIs?
- if you need very tight control over physical data placement across the cluster.
- You need to maintain some legacy codebase written using RDDs.
- You need to do some custom shared variable manipulation

RDDs

- An RDD represents an immutable, partitioned collection of records that can be operated on in parallel.
- The RDD API is similar to the Dataset, which we saw in the previous part of the book, except that RDDs are not stored in, or manipulated with, the structured data engine.
- Every manipulation and interaction between values must be defined by hand, meaning that you must “reinvent the wheel” for whatever task you are trying to carry out
- Also, optimizations are going to require much more manual work, because Spark does not understand the inner structure of your records as it does with the Structured APIs

RDDs

- Let’s formally define RDDs. Internally, each RDD is characterized by five main properties:

- The Partitioner is probably one of the core reasons why you might want to use RDDs in your code. Specifying your own custom Partitioner can give you significant performance and stability improvements if you use it correctly.

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

Interoperating Between DataFrames, Datasets, and RDDs

```
#converts a Dataframe[Long] to RDD[Long]
spark.range(10).rdd
```

```
MapPartitionsRDD[52] at javaToPython at NativeMethodAccessorImpl.java:0
```

```
spark.range(10).toDF("id").rdd.map(lambda row: row[0])
```

```
PythonRDD[72] at RDD at PythonRDD.scala:53
```

```
#converts a  RDD[Long] to Dataframe[Long]
spark.range(10).rdd.toDF()
```

```
DataFrame[id: bigint]
```

From a Local Collection

- To create an RDD from a collection, you will need to use the parallelize method on a SparkContext (within a SparkSession).
- This turns a single node collection into a parallel collection.
- When creating this parallel collection, you can also explicitly state the number of partitions into which you would like to distribute this array

```
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
.split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
words.getNumPartitions()
```

```
2
```

From a Local Collection

- To create an RDD from a collection, you will need to use the parallelize method on a SparkContext (within a SparkSession).
- This turns a single node collection into a parallel collection.
- When creating this parallel collection, you can also explicitly state the number of partitions into which you would like to distribute this array

```
myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
.split(" ")
words = spark.sparkContext.parallelize(myCollection, 2)
words
```

```
ParallelCollectionRDD[85] at readRDDFromFile at PythonRDD.scala:274
```

Chapter 14. Distributed Shared Variables

Broadcast Variables

- Broadcast variables are a way you can share an immutable value efficiently around the cluster without encapsulating that variable in a function closure

- The normal way to use a variable inyour driver node inside your tasks is to simply reference it in your function closures (e.g., in a map operation), but this can be inefficient, especially for large variables such as a lookup table ora machine learning model.

Broadcast Variables

- This is where broadcast variables come in. Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task

Broadcast Variables

- Example : You would like to supplement your list of words with other information that you have, which is many kilobytes, megabytes, or potentially even gigabytes in size. This is technically a right join

```
my_collection = "Spark The Definitive Guide : Big Data Processing Made Simple"\
.split(" ")
words = spark.sparkContext.parallelize(my_collection, 2)

supplementalData = {"Spark":1000, "Definitive":200,
"Big":-300, "Simple":100}
```

```
suppBroadcast = spark.sparkContext.broadcast(supplementalData)
```

Broadcast Variables

- Example

```
suppBroadcast.value
```

```
{'Spark': 1000, 'Definitive': 200, 'Big': -300, 'Simple': 100}
```

```
words.map(lambda word: (word, suppBroadcast.value.get(word, 0)))\
.sortBy(lambda wordPair: wordPair[1])\
.collect()
```

```
[('Big', -300),
 ('The', 0),
 ('Guide', 0),
 (':', 0),
```

Accumulators

- Accumulators, Spark's second type of shared variable, are a way of updating a value inside of a variety of transformations and propagating that value to the driver node in an efficient and fault-tolerant way
- Accumulators provide a mutable variable that a Spark cluster can safely update on a per-row basis
- Accumulator updates performed inside actions only

Accumulators

- Example : let’s create an accumulator that will count the number of flights to or from China.(Even though we could do this in a fairly straightfoward manner in SQL)

```
# Accumulators
flights = spark.read.format("csv")\
.option("inferSchema", "true")\
.option("header", "true")\
.load("2010-summary.csv")
```

```
accChina = spark.sparkContext.accumulator(0)
```

Accumulators

- Example

```
def accChinaFunc(flight_row):
  destination = flight_row["DEST_COUNTRY_NAME"]
  origin = flight_row["ORIGIN_COUNTRY_NAME"]
  if destination == "China":
    accChina.add(flight_row["count"])
  if origin == "China":
    accChina.add(flight_row["count"])
```

```
flights.foreach(lambda flight_row: accChinaFunc(flight_row))

accChina.value
```

```
953
```

Accumulators

- Named accumulators will display in the Spark UI, whereas unnamed ones will not.

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.5 s	0.5 s	0.5 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks
driver	10.172.238.229:44026	0.5 s	1	0	1

Accumulators

Accumulable	Value
China	953

Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	210	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/01/17 21:33:27	0.5 s		China: 953	

Part IV. Production Applications

Chapter 15. How Spark Runs on a Cluster

The Architecture of a Spark Application

- The Spark driver :

- It is the controller of the execution of a Spark Application and maintains all of the state of the Spark cluster (the state and tasks of the executors).
- Process on a physical machine that is responsible for maintaining the state of the application running on the cluster

The Architecture of a Spark Application

- The Spark executors :

- Spark executors are the processes that perform the tasks assigned by the Spark driver

- Executors have one core responsibility: take the tasks assigned by the driver, run them, and report back their state (success or failure) and results.

The Architecture of a Spark Application

- The cluster manager:

- The cluster manager is responsible for maintaining a cluster of machines that will run your Spark Application(s)
- A cluster manager will have its own “driver” (sometimes called master) and “worker” abstractions
- The core difference is that these are tied to physical machines rather than processes (as they are in Spark).

Execution Modes

- Cluster mode
- Client mode
- Local mode

Execution Modes : Cluster mode

- User submits a pre-compiled JAR, Python script, or R script to a cluster manager
- The cluster manager then launches the driver process on a worker node inside the cluster, in addition to the executor processes.
- This means that the cluster manager is responsible for maintaining all Spark Application–related processes

Execution Modes : Client mode

- Client mode is nearly the same as cluster mode except that the 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 processes
- We are running the Spark Application from a machine that is not colocated on the cluster
- These machines are commonly referred to as gateway machines or edge nodes

Execution Modes : Local mode

- Runs the entire Spark Application on a single machine.
- It achieves parallelism through threads on that single machine.
- This is a common way to learn Spark, to test your applications, or experiment iteratively with local development

The Life Cycle of a Spark Application (Outside Spark)

- **Client Request**
- Darker arrows represent communication by Spark or Spark-related processes, whereas dashed lines represent more general communication
- At this point, you are executing code on your local machine and you're going to make a request to the cluster manager driver node
- We are explicitly asking for resources for the Spark driver process only

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode cluster \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

The Life Cycle of a Spark Application (Outside Spark)

- **Launch**
- Running user code
- This code must include a SparkSession that initializes a Spark cluster (e.g., driver + executors).
- The SparkSession will subsequently communicate with the cluster manager (the darker line), asking it to launch Spark executor processes across the cluster (the lighter lines)
- The cluster manager responds by launching the executor processes (assuming all goes well) and sends the relevant information about their locations to the driver process

The Life Cycle of a Spark Application (Outside Spark)

- **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

The Life Cycle of a Spark Application (Outside Spark)

- **Completion**
- After a Spark Application completes, the driver processs exits with either success or failure
- The cluster manager then shuts down the executors in that Spark cluster for the driver

The Life Cycle of a Spark Application (Inside Spark)

- Each application is made up of one or more Spark jobs.
- Spark jobs within an application are executed serially (unless you use threading to launch multiple actions in parallel)
- The first step of any Spark Application is creating a SparkSession
- A SparkContext object within the SparkSession represents the connection to the Spark cluster (communicate with some of Spark's lower-level APIs, such as RDDs)
- Logical instructions to physical execution

```python
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local")\
.appName("Word Count")\
.getOrCreate()
```

```python
df1 = spark.range(2, 10000000, 2)
df2 = spark.range(2, 10000000, 4)
step1 = df1.repartition(5)
step12 = df2.repartition(6)
step2 = step1.selectExpr("id * 5 as id")
step3 = step2.join(step12, ["id"])
step4 = step3.selectExpr("sum(id)")
step4.collect() # 2500000000000
```

The Life Cycle of a Spark Application (Inside Spark)

- Logical plan

```
+- == Initial Plan ==
   HashAggregate(keys=[], functions=[sum(id#8L)])
   +- Exchange SinglePartition, ENSURE_REQUIREMENTS, [plan_id=44]
      +- HashAggregate(keys=[], functions=[partial_sum(id#8L)])
         +- Project [id#8L]
            +- SortMergeJoin [id#8L], [id#2L], Inner
               :- Sort [id#8L ASC NULLS FIRST], false, 0
               :  +- Exchange hashpartitioning(id#8L, 200), ENSURE_REQUIREMENTS, [plan_id=36]
               :     +- Project [(id#0L * 5) AS id#8L]
               :        +- Exchange RoundRobinPartitioning(5), REPARTITION_BY_NUM, [plan_id=26]
               :           +- Range (2, 10000000, step=2, splits=1)
               +- Sort [id#2L ASC NULLS FIRST], false, 0
                  +- Exchange hashpartitioning(id#2L, 200), ENSURE_REQUIREMENTS, [plan_id=37]
                     +- Exchange RoundRobinPartitioning(6), REPARTITION_BY_NUM, [plan_id=29]
                        +- Range (2, 10000000, step=4, splits=1)
```

The Life Cycle of a Spark Application (Inside Spark)

- Physical plan : AdaptiveSparkPlan (spark3)

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=true
+- == Final Plan ==
   *(7) HashAggregate(keys=[], functions=[sum(id#8L)])
   +- ShuffleQueryStage 4
      +- Exchange SinglePartition, ENSURE_REQUIREMENTS, [plan_id=264]
         +- *(6) HashAggregate(keys=[], functions=[partial_sum(id#8L)])
            +- *(6) Project [id#8L]
               +- *(6) SortMergeJoin [id#8L], [id#2L], Inner
                  :- *(4) Sort [id#8L ASC NULLS FIRST], false, 0
                  :  +- AQEShuffleRead coalesced
                  :     +- ShuffleQueryStage 2
                  :        +- Exchange hashpartitioning(id#8L, 200), ENSURE_REQUIREMENTS, [plan_id=135]
                  :           +- *(3) Project [(id#0L * 5) AS id#8L]
                  :              +- ShuffleQueryStage 0
                  :                 +- Exchange RoundRobinPartitioning(5), REPARTITION_BY_NUM, [plan_id=68]
                  :                    +- *(1) Range (2, 10000000, step=2, splits=1)
                  +- *(5) Sort [id#2L ASC NULLS FIRST], false, 0
                     +- AQEShuffleRead coalesced
                        +- ShuffleQueryStage 3
                           +- Exchange hashpartitioning(id#2L, 200), ENSURE_REQUIREMENTS, [plan_id=80]
                              +- ShuffleQueryStage 1
                                 +- Exchange RoundRobinPartitioning(6), REPARTITION_BY_NUM, [plan_id=76]
                                    +- *(2) Range (2, 10000000, step=4, splits=1)
```

The Life Cycle of a Spark Application (Inside Spark)

- **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

- **Stages**

- Stages in Spark represent groups of tasks that can be executed together to compute the same operation on multiple machines

- **Tasks**

- Stages in Spark consist of tasks. Each task corresponds to a combination of blocks of data and a set of transformations that will run on a single executor

The Life Cycle of a Spark Application (Inside Spark)

- **Execution Details**

- Spark automatically pipelines stages and tasks that can be done together, such as a map operation followed by another map operation.

- All shuffle operations, Spark writes the data to stable storage (e.g., disk), and can reuse it across multiple jobs.

Monitoring and Debugging

- **Components of a Spark application that you can monitor**

Monitoring and Debugging

- **Spark Ui => http://localhost:4040/**

- The Jobs tab refers to Spark jobs.
- The Stages tab pertains to individual stages (and their relevant tasks).
- The Storage tab includes information and the data that is currently cached in our Spark Application.
- The Environment tab contains relevant information about the configurations and current settings of the Spark application.
- The SQL tab refers to our Structured API queries (including SQL and DataFrames).
- The Executors tab provides detailed information about each executor running our
- application.

Monitoring and Debugging

❑ Example

```
spark.read\
.option("header", "true")\
.csv("online-retail-dataset.csv")\
.repartition(2)\
.selectExpr("instr(Description, 'GLASS') >= 1 as is_glass")\
.groupBy("is_glass")\
.count()\
.collect()
```

```
[Row(is_glass=None, count=1454),
 Row(is_glass=True, count=12861),
 Row(is_glass=False, count=527594)]
```

Monitoring and Debugging

- **Example**

- **Directed Acyclic Graph (DAG)**

Monitoring and Debugging

- The box on top, labeled WholeStateCodegen, represents a full scan of the CSV file.
- The box below that represents a shuffle that we forced when we called repartition

Monitoring and Debugging

- The next step is our projection (selecting/adding/filtering columns) and the aggregation
- Output rows is six : the number of output rows (3) multiplied by the number of partitions (2) at aggregation time.

Monitoring and Debugging

- aggregation of the subaggregations that we saw happen on a per-partition basis in the previous stage

Monitoring and Debugging

- Job's execution

Spark 2.1.0 | Jobs | Stages | Storage | Environment | Executors | SQL | Spark shell application UI

Details for Job 2

Status: SUCCEEDED
Completed Stages: 3

- Event Timeline
- DAG Visualization

Completed Stages (3)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
4	collect at <console>:31	+details	2017/04/08 16:24:43	0.4 s	200/200			380.0 B	
3	collect at <console>:31	+details	2017/04/08 16:24:42	0.3 s	2/2			10.7 MB	380.0 B
2	collect at <console>:31	+details	2017/04/08 16:24:42	0.7 s	8/8	43.4 MB			10.7 MB

Part V. Streaming

What Is Stream Processing?

- Stream processing is the act of continuously incorporating new data to compute a result
- In stream processing, the input data is unbounded and has no predetermined beginning or end.

- Batch processing, in which the computation runs on a fixed-input dataset

- Credit card transactions,
- Clicks on a website,
- Sensor readings from Internet of Things [IoT] devices

- Notifications and alerting
- Real-time reporting
- Incremental ETL
- Update data to serve in real time

Advantages of Stream Processing

- Lower latency: when your application needs to respond quickly (on a timescale of minutes, seconds, or milliseconds)

- Efficient in updating a result than repeated batch jobs, because it automatically incrementalizes the computation.

Challenges of Stream Processing

- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of state
- Supporting high-data throughput
- Processing each event exactly once despite machine failures
- Handling load imbalance and stragglers
- Responding to events at low latency
- Joining with external data in other storage systems
- Determining how to update output sinks as new events arrive
- Writing data transactionally to output systems
- Updating your application's business logic at runtime

Record-at-a-Time Versus Declarative APIs

- **Record-at-a-Time**
- The simplest way to design a streaming API would be to just pass each event to the application and let it react using custom code (Apache Storm)
- Maintaining state, are solely governed by the application

- **Declarative APIs**
- Your application specifies what to compute but not how to compute it in response to each new event and how to recover from failure (Spark Dstreams, Google Dataflow, Apache Kafka)

Event Time Versus Processing Time

- **Event Time**
- Event time is the idea of processing data based on timestamps inserted into each record at the source

- **Declarative APIs**
- Processing time is the time when the record is received at the streaming application

Continuous Versus Micro-Batch Execution

- **Continuous**
- Read record one by one
- Low latency
- Maximum throughput

- **Micro-Batch**
- Accumulate small batches of input data (say, 500 ms'worth), then process each batch in parallel using a distributed collection of tasks

Structured Streaming Basics

- Rather than introducing a separate API, Structured Streaming uses the existing structured APIs in Spark (DataFrames, Datasets, and SQL), meaning that all the operations you are familiar with there are supported.

Input Sources

- Apache Kafka
- Files on a distributed file system like HDFS or S3 (Spark will continuously read new files in a directory)
- A socket source for testing

Sinks and Output Modes

- Sinks specify the destination for the result set of that stream.

- Apache Kafka
- Almost any file format
- A foreach sink for running arbitary computation on the output records
- A console sink for testing
- A memory sink for debugging

- Output modes define how we want Spark to write data to that sink.

- Append (only add new records to the output sink)
- Update (update changed records in place)
- Complete (rewrite the full output)

Structured Streaming in Action

- Smartphone and smartwatch sensor readings from a variety of devices

```
static = spark.read.json("data/activity-data/")
dataSchema = static.schema
```

```
static.printSchema()
```

```
root
 |-- Arrival_Time: long (nullable = true)
 |-- Creation_Time: long (nullable = true)
 |-- Device: string (nullable = true)
 |-- Index: long (nullable = true)
 |-- Model: string (nullable = true)
 |-- User: string (nullable = true)
 |-- _corrupt_record: string (nullable = true)
 |-- gt: string (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)
```

Structured Streaming in Action

- Create streaming

```
streaming = spark.readStream\
.schema(dataSchema)\
.option("maxFilesPerTrigger", 1)\
.json("data/activity-data")
```

- Structured Streaming does not let you perform schema inference without explicitly enabling it.

- Enable schema inference for this by setting the configuration *spark.sql.streaming.schemaInference* to *true*.

```
# Reduce partition due to local env
spark.conf.set("spark.sql.shuffle.partitions", 5)
# Some transformations
activityCounts = streaming.groupBy("gt").count()
```

- To prevent the dirver to exit *activityQuery.awaitTermination()*

```
activityQuery = activityCounts.writeStream\
.queryName("activity_counts")\
.format("memory").outputMode("complete")\
.start()
```

Structured Streaming in Action

- Query table by adding new files

```python
# Loop to show table state
# Sleep 30s
# Betwen evry sleep add new files from activity-data to simulate
# new input files
from time import sleep
for x in range(5):
    spark.sql("SELECT * FROM activity_counts").show()
    sleep(30)
```

```
+----------+-----+
|        gt|count|
+----------+-----+
|       sit| 6992|
|     stand| 6359|
|stairsdown| 4243|
|      null|    2|
|      walk| 7161|
|  stairsup| 4614|
|      null| 5622|
|      bike| 3614|
+----------+-----+
```

```python
# Loop to show table state
# Sleep 30s
# Betwen evry sleep add new files from activity-data to simulate
# new input files
from time import sleep
for x in range(5):
    spark.sql("SELECT * FROM activity_counts").show()
    sleep(30)
```

```
+----------+-----+
|        gt|count|
+----------+-----+
|       sit| 6992|
|     stand| 6359|
|stairsdown| 4243|
|      null|    2|
|      walk| 7161|
|  stairsup| 4614|
|      null| 5622|
|      bike| 3614|
+----------+-----+

+----------+-----+
|        gt|count|
+----------+-----+
|       sit|19300|
|     stand|17744|
|stairsdown|13608|
|      null|    2|
|      walk|20417|
|  stairsup|15066|
|      null|16070|
|      bike|14412|
+----------+-----+
```

Structured Streaming in Action

- Selections and Filtering

```
from pyspark.sql.functions import expr
simpleTransform = streaming.withColumn("stairs", expr("gt like '%stairs%'"))\
.where("stairs")\
.where("gt is not null")\
.select("gt", "model", "arrival_time", "creation_time")\
.writeStream\
.queryName("simple_transform")\
.format("memory")\
.outputMode("append")\
.start()
```

Structured Streaming in Action

- Selections and Filtering

```
from pyspark.sql.functions import expr
simpleTransform = streaming.withColumn("stairs", expr("gt like '%stairs%'"))\
.where("stairs")\
.where("gt is not null")\
.select("gt", "model", "arrival_time", "creation_time")\
.writeStream\
.queryName("simple_transform")\
.format("memory")\
.outputMode("append")\
.start()
```

Questions ?

Merci