Chapter-3

June 25, 2020

1 A tour of Sparks's Toolset

→ Spark is composed of primitives - the lower-level APIs and the Structured APIs - and then a series of standard libraries for additional functionality

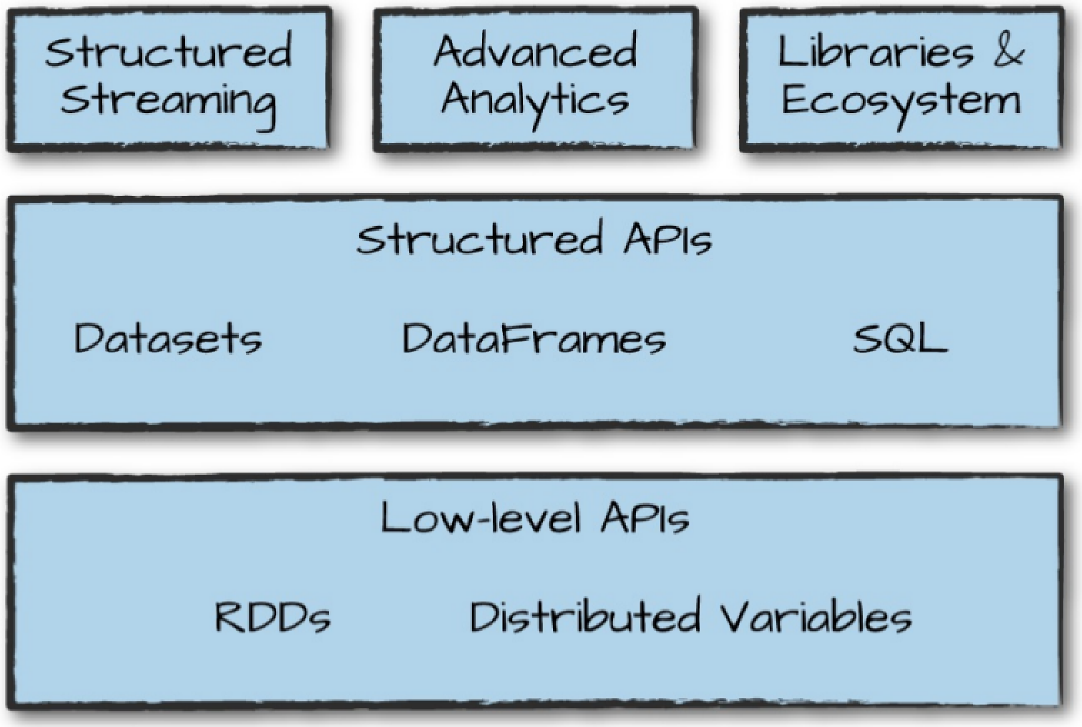

Figure 3-1. Spark's toolset

→

Spark's libraries support a variety of different tasks, from graph analysisi and machine learning to streaming and integrations with a host of computing and storage systems

1.1 Overview

1. Running production applications with **spark-submit**
2. Datasets: type-safe APIs for structured data
3. Structured Straming
4. Machine learning and advanced analytics
5. Resilient Distributed Datasets(RDD): Spark's low level APIs

6. Spark R
7. The third-party package ecosystem

1.2 Running Production Applications

- Spark makes it easy to develop and create big data programs
- It also makes it easy to turn interactive exploration into production applications with **spark-submit**, a built-in command-line tool.
- **Spark-submit** lets us send application code to a cluster and launch it to execute there.
- Upon submission, the application will run until it exists with completion or encounters an error.
- This can be used with all of Spark's support cluster managers:
  - Standalone
  - Mesos
  - YARN
- **Spark-submit** offers several controls with which we can specify the resources our application needs as well as how it should be run and its command-line arguments
- **Spark-submit** can be used with any applications written in any of Spark's supported language.

[1]:
```
! spark-submit --master local ../../spark-2.1.1-bin-hadoop2.7/examples/src/main/
↪python/pi.py 10
```

```
20/06/25 00:33:21 WARN Utils: Your hostname, raj-Predator-G3-572 resolves to a
loopback address: 127.0.1.1; using 192.168.1.118 instead (on interface enp3s0f1)
20/06/25 00:33:21 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
20/06/25 00:33:22 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform… using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
20/06/25 00:33:22 INFO SparkContext: Running Spark version 2.4.5
20/06/25 00:33:22 INFO SparkContext: Submitted application: PythonPi
20/06/25 00:33:22 INFO SecurityManager: Changing view acls to: raj
20/06/25 00:33:22 INFO SecurityManager: Changing modify acls to: raj
20/06/25 00:33:22 INFO SecurityManager: Changing view acls groups to:
20/06/25 00:33:22 INFO SecurityManager: Changing modify acls groups to:
20/06/25 00:33:22 INFO SecurityManager: SecurityManager: authentication
disabled; ui acls disabled; users  with view permissions: Set(raj); groups with
view permissions: Set(); users  with modify permissions: Set(raj); groups with
modify permissions: Set()
20/06/25 00:33:23 INFO Utils: Successfully started service 'sparkDriver' on port
43243.
20/06/25 00:33:23 INFO SparkEnv: Registering MapOutputTracker
20/06/25 00:33:23 INFO SparkEnv: Registering BlockManagerMaster
20/06/25 00:33:23 INFO BlockManagerMasterEndpoint: Using
org.apache.spark.storage.DefaultTopologyMapper for getting topology information
20/06/25 00:33:23 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up
```

```
20/06/25 00:33:23 INFO DiskBlockManager: Created local directory at
/tmp/blockmgr-67bf85da-cbae-45f0-8af5-c76261eb1734
20/06/25 00:33:23 INFO MemoryStore: MemoryStore started with capacity 366.3 MB
20/06/25 00:33:23 INFO SparkEnv: Registering OutputCommitCoordinator
20/06/25 00:33:23 INFO Utils: Successfully started service 'SparkUI' on port
4040.
20/06/25 00:33:23 INFO SparkUI: Bound SparkUI to 0.0.0.0, and started at
http://192.168.1.118:4040
20/06/25 00:33:23 INFO Executor: Starting executor ID driver on host localhost
20/06/25 00:33:23 INFO Utils: Successfully started service
'org.apache.spark.network.netty.NettyBlockTransferService' on port 45169.
20/06/25 00:33:23 INFO NettyBlockTransferService: Server created on
192.168.1.118:45169
20/06/25 00:33:23 INFO BlockManager: Using
org.apache.spark.storage.RandomBlockReplicationPolicy for block replication
policy
20/06/25 00:33:23 INFO BlockManagerMaster: Registering BlockManager
BlockManagerId(driver, 192.168.1.118, 45169, None)
20/06/25 00:33:23 INFO BlockManagerMasterEndpoint: Registering block manager
192.168.1.118:45169 with 366.3 MB RAM, BlockManagerId(driver, 192.168.1.118,
45169, None)
20/06/25 00:33:23 INFO BlockManagerMaster: Registered BlockManager
BlockManagerId(driver, 192.168.1.118, 45169, None)
20/06/25 00:33:23 INFO BlockManager: Initialized BlockManager:
BlockManagerId(driver, 192.168.1.118, 45169, None)
20/06/25 00:33:24 INFO SharedState: Setting hive.metastore.warehouse.dir
('null') to the value of spark.sql.warehouse.dir ('file:/home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/Notebooks/spark-warehouse').
20/06/25 00:33:24 INFO SharedState: Warehouse path is 'file:/home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/Notebooks/spark-warehouse'.
20/06/25 00:33:24 INFO StateStoreCoordinatorRef: Registered
StateStoreCoordinator endpoint
20/06/25 00:33:24 INFO SparkContext: Starting job: reduce at /home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/Notebooks/../../spark-2.1.1-bin-
hadoop2.7/examples/src/main/python/pi.py:43
20/06/25 00:33:24 INFO DAGScheduler: Got job 0 (reduce at /home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/Notebooks/../../spark-2.1.1-bin-
hadoop2.7/examples/src/main/python/pi.py:43) with 10 output partitions
20/06/25 00:33:24 INFO DAGScheduler: Final stage: ResultStage 0 (reduce at
/home/raj/online-courses/pyspark/Spark-The-Definitive-
Guide/Notebooks/../../spark-2.1.1-bin-
hadoop2.7/examples/src/main/python/pi.py:43)
20/06/25 00:33:24 INFO DAGScheduler: Parents of final stage: List()
20/06/25 00:33:24 INFO DAGScheduler: Missing parents: List()
20/06/25 00:33:24 INFO DAGScheduler: Submitting ResultStage 0 (PythonRDD[1] at
reduce at /home/raj/online-courses/pyspark/Spark-The-Definitive-
Guide/Notebooks/../../spark-2.1.1-bin-
hadoop2.7/examples/src/main/python/pi.py:43), which has no missing parents
```

```
20/06/25 00:33:24 INFO MemoryStore: Block broadcast_0 stored as values in memory
(estimated size 6.4 KB, free 366.3 MB)
20/06/25 00:33:24 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in
memory (estimated size 4.4 KB, free 366.3 MB)
20/06/25 00:33:24 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on
192.168.1.118:45169 (size: 4.4 KB, free: 366.3 MB)
20/06/25 00:33:24 INFO SparkContext: Created broadcast 0 from broadcast at
DAGScheduler.scala:1163
20/06/25 00:33:24 INFO DAGScheduler: Submitting 10 missing tasks from
ResultStage 0 (PythonRDD[1] at reduce at /home/raj/online-courses/pyspark/Spark-
The-Definitive-Guide/Notebooks/../../spark-2.1.1-bin-
hadoop2.7/examples/src/main/python/pi.py:43) (first 15 tasks are for partitions
Vector(0, 1, 2, 3, 4, 5, 6, 7, 8, 9))
20/06/25 00:33:24 INFO TaskSchedulerImpl: Adding task set 0.0 with 10 tasks
20/06/25 00:33:24 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0,
localhost, executor driver, partition 0, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:24 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
20/06/25 00:33:25 INFO PythonRunner: Times: total = 730, boot = 583, init = 52,
finish = 95
20/06/25 00:33:25 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1421
bytes result sent to driver
20/06/25 00:33:25 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1,
localhost, executor driver, partition 1, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:25 INFO Executor: Running task 1.0 in stage 0.0 (TID 1)
20/06/25 00:33:25 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in
815 ms on localhost (executor driver) (1/10)
20/06/25 00:33:25 INFO PythonAccumulatorV2: Connected to AccumulatorServer at
host: 127.0.0.1 port: 39075
20/06/25 00:33:25 INFO PythonRunner: Times: total = 145, boot = 3, init = 46,
finish = 96
20/06/25 00:33:25 INFO Executor: Finished task 1.0 in stage 0.0 (TID 1). 1421
bytes result sent to driver
20/06/25 00:33:25 INFO TaskSetManager: Starting task 2.0 in stage 0.0 (TID 2,
localhost, executor driver, partition 2, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:25 INFO Executor: Running task 2.0 in stage 0.0 (TID 2)
20/06/25 00:33:25 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in
155 ms on localhost (executor driver) (2/10)
20/06/25 00:33:26 INFO PythonRunner: Times: total = 143, boot = 3, init = 42,
finish = 98
20/06/25 00:33:26 INFO Executor: Finished task 2.0 in stage 0.0 (TID 2). 1464
bytes result sent to driver
20/06/25 00:33:26 INFO TaskSetManager: Starting task 3.0 in stage 0.0 (TID 3,
localhost, executor driver, partition 3, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:26 INFO TaskSetManager: Finished task 2.0 in stage 0.0 (TID 2) in
152 ms on localhost (executor driver) (3/10)
20/06/25 00:33:26 INFO Executor: Running task 3.0 in stage 0.0 (TID 3)
20/06/25 00:33:26 INFO PythonRunner: Times: total = 172, boot = 8, init = 57,
finish = 107
```

```
20/06/25 00:33:26 INFO Executor: Finished task 3.0 in stage 0.0 (TID 3). 1421
bytes result sent to driver
20/06/25 00:33:26 INFO TaskSetManager: Starting task 4.0 in stage 0.0 (TID 4,
localhost, executor driver, partition 4, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:26 INFO Executor: Running task 4.0 in stage 0.0 (TID 4)
20/06/25 00:33:26 INFO TaskSetManager: Finished task 3.0 in stage 0.0 (TID 3) in
181 ms on localhost (executor driver) (4/10)
20/06/25 00:33:26 INFO PythonRunner: Times: total = 145, boot = 3, init = 44,
finish = 98
20/06/25 00:33:26 INFO Executor: Finished task 4.0 in stage 0.0 (TID 4). 1421
bytes result sent to driver
20/06/25 00:33:26 INFO TaskSetManager: Starting task 5.0 in stage 0.0 (TID 5,
localhost, executor driver, partition 5, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:26 INFO Executor: Running task 5.0 in stage 0.0 (TID 5)
20/06/25 00:33:26 INFO TaskSetManager: Finished task 4.0 in stage 0.0 (TID 4) in
152 ms on localhost (executor driver) (5/10)
20/06/25 00:33:26 INFO PythonRunner: Times: total = 150, boot = 3, init = 45,
finish = 102
20/06/25 00:33:26 INFO Executor: Finished task 5.0 in stage 0.0 (TID 5). 1464
bytes result sent to driver
20/06/25 00:33:26 INFO TaskSetManager: Starting task 6.0 in stage 0.0 (TID 6,
localhost, executor driver, partition 6, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:26 INFO TaskSetManager: Finished task 5.0 in stage 0.0 (TID 5) in
157 ms on localhost (executor driver) (6/10)
20/06/25 00:33:26 INFO Executor: Running task 6.0 in stage 0.0 (TID 6)
20/06/25 00:33:26 INFO PythonRunner: Times: total = 162, boot = 3, init = 50,
finish = 109
20/06/25 00:33:26 INFO Executor: Finished task 6.0 in stage 0.0 (TID 6). 1421
bytes result sent to driver
20/06/25 00:33:26 INFO TaskSetManager: Starting task 7.0 in stage 0.0 (TID 7,
localhost, executor driver, partition 7, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:26 INFO Executor: Running task 7.0 in stage 0.0 (TID 7)
20/06/25 00:33:26 INFO TaskSetManager: Finished task 6.0 in stage 0.0 (TID 6) in
168 ms on localhost (executor driver) (7/10)
20/06/25 00:33:26 INFO PythonRunner: Times: total = 143, boot = 3, init = 45,
finish = 95
20/06/25 00:33:26 INFO Executor: Finished task 7.0 in stage 0.0 (TID 7). 1421
bytes result sent to driver
20/06/25 00:33:26 INFO TaskSetManager: Starting task 8.0 in stage 0.0 (TID 8,
localhost, executor driver, partition 8, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:26 INFO TaskSetManager: Finished task 7.0 in stage 0.0 (TID 7) in
150 ms on localhost (executor driver) (8/10)
20/06/25 00:33:26 INFO Executor: Running task 8.0 in stage 0.0 (TID 8)
20/06/25 00:33:27 INFO PythonRunner: Times: total = 158, boot = 6, init = 46,
finish = 106
20/06/25 00:33:27 INFO Executor: Finished task 8.0 in stage 0.0 (TID 8). 1421
bytes result sent to driver
20/06/25 00:33:27 INFO TaskSetManager: Starting task 9.0 in stage 0.0 (TID 9,
localhost, executor driver, partition 9, PROCESS_LOCAL, 7852 bytes)
20/06/25 00:33:27 INFO Executor: Running task 9.0 in stage 0.0 (TID 9)
20/06/25 00:33:27 INFO TaskSetManager: Finished task 8.0 in stage 0.0 (TID 8) in
165 ms on localhost (executor driver) (9/10)
20/06/25 00:33:27 INFO PythonRunner: Times: total = 151, boot = 4, init = 45,
finish = 102
20/06/25 00:33:27 INFO Executor: Finished task 9.0 in stage 0.0 (TID 9). 1378
bytes result sent to driver
20/06/25 00:33:27 INFO TaskSetManager: Finished task 9.0 in stage 0.0 (TID 9) in
161 ms on localhost (executor driver) (10/10)
20/06/25 00:33:27 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have
all completed, from pool
20/06/25 00:33:27 INFO DAGScheduler: ResultStage 0 (reduce at /home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/Notebooks/../../spark-2.1.1-bin-
hadoop2.7/examples/src/main/python/pi.py:43) finished in 2.373 s
20/06/25 00:33:27 INFO DAGScheduler: Job 0 finished: reduce at /home/raj/online-
courses/pyspark/Spark-The-Definitive-Guide/Notebooks/../../spark-2.1.1-bin-
hadoop2.7/examples/src/main/python/pi.py:43, took 2.441946 s
Pi is roughly 3.142600
20/06/25 00:33:27 INFO SparkUI: Stopped Spark web UI at
http://192.168.1.118:4040
20/06/25 00:33:27 INFO MapOutputTrackerMasterEndpoint:
MapOutputTrackerMasterEndpoint stopped!
20/06/25 00:33:27 INFO MemoryStore: MemoryStore cleared
20/06/25 00:33:27 INFO BlockManager: BlockManager stopped
20/06/25 00:33:27 INFO BlockManagerMaster: BlockManagerMaster stopped
20/06/25 00:33:27 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint:
OutputCommitCoordinator stopped!
20/06/25 00:33:27 INFO SparkContext: Successfully stopped SparkContext
20/06/25 00:33:28 INFO ShutdownHookManager: Shutdown hook called
20/06/25 00:33:28 INFO ShutdownHookManager: Deleting directory /tmp/spark-cbed8e
dd-b907-4696-86d2-3926f24e82e4/pyspark-7677be89-73f3-4f2d-b7e5-09a5bb1bd2f3
20/06/25 00:33:28 INFO ShutdownHookManager: Deleting directory /tmp/spark-
cbed8edd-b907-4696-86d2-3926f24e82e4
20/06/25 00:33:28 INFO ShutdownHookManager: Deleting directory
/tmp/spark-d2d17421-7871-405f-989d-57744de51391
```

1.3 Datasets: Type-Safe Structured APIs

- The Dataset API is used for writing statically typed code (that does type checking at compile time as opposed to runtime) in Java and Scala.
- **The Dataset API is not available in Python and R, because those languages are dynamically typed.**
- DataFrames are a distributed collection of objects of type Row that can hold various types of tabular data. The Dataset API gives users the ability to assign a Java/Scala class to the records within a DataFrame and manipulate it as a collection of typed objects, similar to a Java *ArrayList* or Scala *Seq*.

- The APIs available on Datasets are **type-safe**, meaning that we cannot accidentally view the objects in a Dataset as being of another class than the class you put in initially.
- With it, we can define our own data type and manipulate it via arbitrary functions. After we've performed our manipulations, Spark can automatically turn it back into a DataFrame, and we can manipulate it further by using the hundreds of functions that Spark includes. This makes it easy to drop down to lower level, perform type-safe coding when necessary, and move higher up to SQL for more rapid analysis.

1.4 Structured Streaming

- Structured Steaming is a high-level API for steam processing.
- With Structured Steaming, we can take the same operations that we perform in batch mode using Spark's structured APIs and run them in streaming fachion.
- This can reduce latency and allow for incremental processing.
- It allows us to rapidly and quickly extract value out of straming systems with virtually no code changes.
- It also makes it easy to conceptualize because we can write our batch job as a way to prototype it and then convert it to a streaming job.
- The way all of this works is by incrementally processing that data.

[2]:
```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("chapter3").getOrCreate()
spark.conf.set("spark.sql.shuffle.partitions","5")
```

[3]:
```
retail_dataset_path = "../data/retail-data/by-day/*.csv"
```

[4]:
```
staticDataFrame = spark.read.format("csv").option("header", "true").
↪option("inferSchema","true").load(retail_dataset_path)
```

[5]:
```
# staticDataFrame = spark.read.option("header","true").
↪option("inferSchema","true").csv(retail_dataset_path)
```

[6]:
```
staticDataFrame.show()
```

```
+---------+---------+--------------------+--------+-------------------+---------
+----------+--------------+
|InvoiceNo|StockCode|         Description|Quantity|
InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------------+--------+-------------------+---------
+----------+--------------+
|   580538|    23084|  RABBIT NIGHT LIGHT|      48|2011-12-05 08:38:00|
1.79|   14075.0|United Kingdom|
|   580538|    23077| DOUGHNUT LIP GLOSS |      20|2011-12-05 08:38:00|
1.25|   14075.0|United Kingdom|
|   580538|    22906|12 MESSAGE CARDS …|      24|2011-12-05 08:38:00|
1.65|   14075.0|United Kingdom|
|   580538|    21914|BLUE HARMONICA IN…|      24|2011-12-05 08:38:00|
1.25|   14075.0|United Kingdom|
|   580538|    22467|   GUMBALL COAT RACK|       6|2011-12-05 08:38:00|
2.55|   14075.0|United Kingdom|
|   580538|    21544|SKULLS  WATER TRA…|      48|2011-12-05 08:38:00|
0.85|   14075.0|United Kingdom|
|   580538|    23126|FELTCRAFT GIRL AM…|       8|2011-12-05 08:38:00|
4.95|   14075.0|United Kingdom|
|   580538|    21833|CAMOUFLAGE LED TORCH|      24|2011-12-05 08:38:00|
1.69|   14075.0|United Kingdom|
|   580539|    21479|WHITE SKULL HOT W…|       4|2011-12-05 08:39:00|
4.25|   18180.0|United Kingdom|
|   580539|   84030E|ENGLISH ROSE HOT …|       4|2011-12-05 08:39:00|
4.25|   18180.0|United Kingdom|
|   580539|    23355|HOT WATER BOTTLE …|       4|2011-12-05 08:39:00|
4.95|   18180.0|United Kingdom|
|   580539|    22111|SCOTTIE DOG HOT W…|       3|2011-12-05 08:39:00|
4.95|   18180.0|United Kingdom|
|   580539|    21115|ROSE CARAVAN DOOR…|       8|2011-12-05 08:39:00|
1.95|   18180.0|United Kingdom|
|   580539|    21411|GINGHAM HEART  DO…|       8|2011-12-05 08:39:00|
1.95|   18180.0|United Kingdom|
|   580539|    23235|STORAGE TIN VINTA…|      12|2011-12-05 08:39:00|
1.25|   18180.0|United Kingdom|
|   580539|    23239|SET OF 4 KNICK KN…|       6|2011-12-05 08:39:00|
1.65|   18180.0|United Kingdom|
|   580539|    22197|      POPCORN HOLDER|      36|2011-12-05 08:39:00|
0.85|   18180.0|United Kingdom|
|   580539|    22693|GROW A FLYTRAP OR…|      24|2011-12-05 08:39:00|
1.25|   18180.0|United Kingdom|
|   580539|    22372|AIRLINE BAG VINTA…|       4|2011-12-05 08:39:00|
4.25|   18180.0|United Kingdom|
|   580539|    22375|AIRLINE BAG VINTA…|       4|2011-12-05 08:39:00|
4.25|   18180.0|United Kingdom|
+---------+---------+--------------------+--------+-------------------+---------
+----------+--------------+
only showing top 20 rows
```

[7]:
```
staticDataFrame.createOrReplaceTempView("retail_data")
staticSchema = staticDataFrame.schema
```

[8]:
```
from pyspark.sql.functions import window, column, desc, col, dayofmonth, month
staticDataFrame\
.selectExpr(
    "CustomerId",
    "(UnitPrice*Quantity) as total_cost",
    "InvoiceDate")\
.groupBy(
    col("CustomerId"),
    window(col("InvoiceDate"), "1 day"))\
.sum("total_cost")\
.show()
```

```
+----------+--------------------+------------------+
|CustomerId|              window|   sum(total_cost)|
+----------+--------------------+------------------+
|   14075.0|[2011-12-05 05:45...|316.78000000000003|
|   18180.0|[2011-12-05 05:45...|            310.73|
|   15358.0|[2011-12-05 05:45...| 830.0600000000003|
|   15392.0|[2011-12-05 05:45...|304.40999999999997|
|   15290.0|[2011-12-05 05:45...|263.02000000000004|
|   16811.0|[2011-12-05 05:45...|             232.3|
|   12748.0|[2011-12-05 05:45...| 363.7899999999999|
|   16500.0|[2011-12-05 05:45...| 52.74000000000001|
|   16873.0|[2011-12-05 05:45...|1854.8300000000002|
|   14060.0|[2011-12-05 05:45...|297.47999999999996|
|   14649.0|[2011-12-05 05:45...| 513.9899999999998|
|   16904.0|[2011-12-05 05:45...| 349.0200000000001|
|   17857.0|[2011-12-05 05:45...|            2979.6|
|   14083.0|[2011-12-05 05:45...| 446.5700000000001|
|   14777.0|[2011-12-05 05:45...|             -2.95|
|   16684.0|[2011-12-05 05:45...| 5401.979999999999|
|   13685.0|[2011-12-05 05:45...|              5.48|
|   15159.0|[2011-12-05 05:45...|           1730.84|
|   18015.0|[2011-12-05 05:45...|            120.03|
|   13305.0|[2011-12-05 05:45...|213.15999999999997|
+----------+--------------------+------------------+
only showing top 20 rows
```

[9]:
```
# spark.sql("with tmp as (select CustomerId, (UnitPrice*Quantity) as␣
↪total_cost, InvoiceDate from retail_data) select CustomerId, sum(total_cost)␣
↪as total_cost from tmp group by CustomerId").show()
```

1.4.1 Streaming code

- Changes
  - Use **readStream** instead of read
  - **maxFilesPerTrigger** option which simply specifies the number of files we should read in at once.(This is to make demonstration more ”streaming” and in a production scenarios this woul probably be omitted)

[10]:
```
streamingDataFrame = spark.readStream\
.schema(staticSchema)\
.option("maxFilesPerTrigger", 1)\
.format("csv")\
.option("header", "true")\
.load(retail_dataset_path)
```

We can see whether our DataFrame is streaming using **.isStreaming** param

[11]:
```
streamingDataFrame.isStreaming
```

[11]:
```
True
```

[12]:
```
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.
- Sreaming actions are a bit different form our conventional static action because we're going to be populating data somewhere instead of just calling something like count(which doesn't make any sense on a stream anyways.
- **The action we will use will output to an in-memory table that we will update after each trigger. In this case, each trigger is based on an individual file (the read option that we set).**
- **Spark will mutate the data in the in-memory table such that we will always have the highest value as specified in our previous aggregation.**

[13]:
```
purchaseByCustomerPerHour.writeStream\
.format("memory")\
.queryName("customer_purchases")\
.outputMode("complete")\
.start()
```

[13]:
```
<pyspark.sql.streaming.StreamingQuery at 0x7f0894d2be10>
```

When we start the steam, we can run queries against it to debug what our result will look lik if we were to write this out to a production sink:

[14]:
```
spark.sql("""
SELECT *
FROM customer_purchases
ORDER BY 'sum(total_cost)' DESC
""")\
.show(5)
```

```
+----------+--------------------+------------------+
|CustomerId|              window|   sum(total_cost)|
+----------+--------------------+------------------+
|   15671.0|[2011-04-11 05:45...|375.96000000000004|
|   17576.0|[2010-12-13 05:45...|177.35000000000002|
|   13240.0|[2011-03-27 05:45...|218.33999999999997|
|   14911.0|[2011-03-11 05:45...|               0.0|
|   13668.0|[2011-04-11 05:45...|            132.44|
+----------+--------------------+------------------+
only showing top 5 rows
```

We'll notice that the composition of our table changes as we read in more data With each file, the results might or might not be changing based on the data.
Another option is to write the results out to the console

[15]:
```
purchaseByCustomerPerHour.writeStream\
.format("console")\
.queryName("customer_purchases_2")\
.outputMode("complete")\
.start()
```

[15]:
```
<pyspark.sql.streaming.StreamingQuery at 0x7f0894d3f0b8>
```

Notice how this window is built on event time, as well, not th time at which Spark process the data. This was one of the shortcomings of Spark Sreaming that Structured Sreaming has resolved

1.5 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 algorithms called **MLlib**. + MLlib allows for preprocessing , munging, training of models, and make predictions at scale on data. + Models trained in MLlib can be used to make predictions in Structured Streaming + Spark provides a sophisticated machine learning API for performaing a variety of machine learning tasks, from classification to regression, and clustering to deep learning.

1.5.1 K-Means

→ Machine learning algorithms in MLlib require that data is represented as numberical values. We need to transform data of other types into some numerical representation for which we will use several DataFrame transformations.

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

```
[17]: # Splitting Data set based on InvoiceDate
      trainDataFrame = preppedDataFrame\
      .where("InvoiceDate < '2011-07-01'")
      testDataFrame = preppedDataFrame\
      .where("InvoiceDate >= '2011-07-01'")
```

```
[18]: trainDataFrame.count()
```

```
[18]: 245903
```

```
[19]: testDataFrame.count()
```

```
[19]: 296006
```

StringIndexer → Spark's MLlib provides a number of transformations, with which we can automate some of our general transformations. One of such transformer is a **StringIndexer**.

```
[20]: from pyspark.ml.feature import StringIndexer
      indexer = StringIndexer()\
      .setInputCol("day_of_week")\
      .setOutputCol("day_of_week_index")
```

This will trun our days of weeks into corresponding numerical values.

OneHotEncoder

```
[21]: 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:

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

Here, we have three key features: the price, the quantity, and the day of week. Next we will set this up into a **pipeline so that any future data we need to transform can go through the same process**

[23]:
```
from pyspark.ml import Pipeline
transformationPipeline = Pipeline()\
.setStages([indexer, encoder, vectorAssembler])
```

Preparing for training is a two-step process:
+ First, we need to fit our transformers to this dataset; they need to know the uniques values present to be indexed. + After that, Spark can encode the values

[24]:
```
fittedPipeline = transformationPipeline.fit(trainDataFrame)
```

After we fit the training data, we are ready to take that fitted pipeline and use it to transform all of our data in a consistent and repeatable way

[25]:
```
transformedTraining = fittedPipeline.transform(trainDataFrame)
```

We could have included our model training in our pipeline.
But for the hyperparameter tunning on the model we don't do because we don't want to repeat the exact same transformations over and over again.
We'll use **caching**, an optimization which will put a copy of the intermediately transformed dataset into memory, allowing us to repeatedly access it at much lower cost than running the entire pipeline again.

Without caching

[26]:
```
from pyspark.ml.clustering import KMeans
kmeans = KMeans()\
.setK(20)\
.setSeed(1)
```

MLLib's DataFrame API follow the naming pattern of + **Algorithm** for the unntrained version like **KMeans** + **AlgorithmModel** for the trained version like **KMeansModel**

[27]:
```
%timeit -n 1 -r 1 kmeans.fit(transformedTraining)
```

```
14.9 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[28]:
```
kmModel = kmeans.fit(transformedTraining)
```

With caching

[29]:
```
transformedTraining.cache()
```

[29]:
```
DataFrame[InvoiceNo: string, StockCode: string, Description: string, Quantity:
int, InvoiceDate: string, UnitPrice: double, CustomerID: double, Country:
string, day_of_week: string, day_of_week_index: double, day_of_week_encoded:
 vector, features: vector]
```

[30]:
```python
from pyspark.ml.clustering import KMeans
cachedkmeans = KMeans()\
.setK(20)\
.setSeed(1)
```

[31]:
```python
%timeit -n 1 -r 1 cachedkmeans.fit(transformedTraining)
```

```
9.87 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

[32]:
```python
cachedkmModel = cachedkmeans.fit(transformedTraining)
```

Evaluation

[33]:
```python
from pyspark.ml.evaluation import ClusteringEvaluator
```

[34]:
```python
evaluator = ClusteringEvaluator()
predictions = kmModel.transform(transformedTraining)
evaluator.evaluate(predictions)
```

[34]:
```
0.6842576726028763
```

[35]:
```python
transformedTest = fittedPipeline.transform(testDataFrame)
predictions = kmModel.transform(transformedTest)
evaluator.evaluate(predictions)
```

[35]:
```
0.5427938390491535
```

1.6 Lower-Level APIs

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

[36]:
```python
from pyspark.sql import Row
```

[37]:
```python
spark.sparkContext.parallelize([Row(1), Row(2), Row(3)]).toDF().show()
```

```
+---+
| _1|
+---+
|  1|
|  2|
|  3|
+---+
```