Jiahao Meng

Aug 20 · 8 min read · Listen

PySpark cheat sheet with code samples

how to initialise Spark, read data, transform it, and build data pipelines In Python.

Markdown Note

Source: Pyspark

How can we easily parallelize our calculations if our data is too large to work with on a single machine?

1. Introduction

1.1 Spark DataFrames VS RDDs

RDD

> Spark's core data structure

✅: A low level object that lets Spark work its magic by splitting data across multiple nodes in the cluster.

❌: However, RDDs are hard to work with directly, so we'll be using the Spark DataFrame abstraction built on top of RDDs.

Spark DataFrames

> Designed to behave a lot like a SQL table

✅:

- easier to understand,
- Operations using DataFrames are automatically optimized
- When using RDDs, it's up to the data scientist to figure out the right way to optimize the query, but the DataFrame implementation has much of this optimization built in!

Create a SparkSession

- `SparkContext` as our connection to the cluster
- `SparkSession` as our interface with that connection.

```
# To start working with Spark DataFrames

from pyspark.sql import SparkSession
my_spark = SparkSession.builder.getOrCreate()
```

```
print(spark.catalog.listTables())
```

2. Spark Schemas

- Define the format of a DataFrame
- May contain various data types:
- Strings, dates, integers, arrays
- Can filter garbage data during import
- Improves read performance

```
# Import the pyspark.sql.types library
from pyspark.sql.types import *

# Define a new schema using the StructType method
people_schema = StructType([
  # Define a StructField for each field
  StructField('name', StringType(), False),
  StructField('age', IntegerType(), False),
  StructField('city', StringType(), False)])

udfpeople = F.udf(peopleParse, ArrayType(people_schema))
```

2.1 Transformations

- Lazy
- only executed when we run a Spark action: `.count()` , `.write()` , etc
- can often be modified before being assigned.
- occasionally cause unexpected behaviors:
- IDs not being added until after other transformations have completed
- built-in function: `monotonically_increasing_id()`

```
# new column "Col+1" = Col + 1
df = df.withColumn("Col+1", df.Col + 1)
```

Filtering

```
long = flights.filter("distance > 1000")
long = flights.filter(flights.distance > 1000)

users_df = users_df.filter(~ col('Name').isNull())
users_df = users_df.filter(users_df.Name.isNotNull())

voter_df = voter_df.withColumn('random_val',
                               when(voter_df.TITLE == 'Councilmember',
F.rand())
                               .when(voter_df.TITLE == 'Mayor', 2)
                               .otherwise(0))
```

Selecting

- `.select()`
- returns only the columns we specify

```
flights.select("air_time", "origin") flights.select(flights.origin)
flights.select(flights.air_time/60).alias("in_hours"))  #SQL expressions
flights.selectExpr("air_time/60 as in_hours")

#SQL expressions
flights.selectExpr("air_time/60 as in_hours")
```

- `.withColumn()`
- returns all the columns of the DataFrame + we defined

Aggregating and Grouping

1. `.min()`
2. `.max()`

5. .avg()

```
# creates a GroupedData to use aboved functions
flights.filter(flights.origin == 'PDX').groupBy().min('distance').show()

+----+-----+---+--------+---------+--------+---------+-------+-------+------+-
-----+----+--------+--------+----+------+
|year|month|day|dep_time|dep_delay|arr_time|arr_delay|carrier|tailnum|flight|o
rigin|dest|air_time|distance|hour|minute|
+----+-----+---+--------+---------+--------+---------+-------+-------+------+-
-----+----+--------+--------+----+------+
|2014|   12|  8|     658|       -7|     935|       -5|     VX| N846VA|  1780|
SEA| LAX|     132|     954|   6|    58|
|2014|    1| 22|    1040|        5|    1505|        5|     AS| N559AS|   851|
SEA| HNL|     360|    2677|  10|    40|
+----+-----+---+--------+---------+--------+---------+-------+-------+------+-
-----+----+--------+--------+----+------+

# Average duration of Delta flights
flights.filter(flights.carrier == "DL").groupBy().avg("air_time").show()

                +------------------+
    |     avg(air_time)|
    +------------------+
    |188.20689655172413|
    +------------------+

# Total air_time in hours: create air_time/60 column, then sum
flights.withColumn("duration_hrs",
flights.air_time/60).groupBy().sum("duration_hrs").show()

    +------------------+
    | sum(duration_hrs)|
    +------------------+
    |25289.600000000126|
    +------------------+

by_month_dest = flights.groupBy('month', 'dest')
by_month_dest.avg('dep_delay').show()

+-----+----+--------------------+
|month|dest|      avg(dep_delay)|
+-----+----+--------------------+
|   11| TUS| -2.3333333333333335|
|   11| ANC|   7.529411764705882|
|    1| BUR|               -1.45|
+-----+----+--------------------+

by_month_dest.agg(F.stddev('dep_delay')).show()
+-----+----+----------------------+
```

Joining

```
def getFirstAndMiddle(names):
  # Return a space separated string of names
  return ' '.join(names[:-1])

# Define the method as a UDF
udfFirstAndMiddle = F.udf(getFirstAndMiddle, StringType())

# Create a new column using your UDF
voter_df = voter_df.withColumn('first_and_middle_name',

udfFirstAndMiddle(voter_df.splits))

+----------+-------------+-------------------+--------------------+---------
-+---------+--------------------+
|      DATE|        TITLE|         VOTER_NAME|
splits|first_name|last_name|first_and_middle_name|
+----------+-------------+-------------------+--------------------+---------
-+---------+--------------------+
|02/08/2017|Councilmember|  Jennifer S. Gates|[Jennifer, S., Gates  |
Jennifer|    Gates|           Jennifer S.|
|02/08/2017|Councilmember| Philip T. Kingston|[Philip, T., Kingston |
Philip| Kingston|            Philip T.|
|02/08/2017|        Mayor|Michael S. Rawlings|[Michael, S., Rawlings|
Michael| Rawlings|           Michael S.|
+----------+-------------+-------------------+--------------------+---------
-+---------+--------------------+
```

2.2 Performance

Caching

✅

- Stores DataFrames in memory or on disk
- Improves speed on later transformations / actions
- Reduces resource usage

❌

- Local disk based caching may not be a performance improvement
- Cached objects may not be available

Tips

- Cache only if you need it
- Try caching DataFrames at various points and determine if our performance improves
- Cache in memory and fast SSD / NVMe storage
- Cache to slow local disk if needed
- Use intermediate files!
- Stop caching objects when finished

```
start_time = time.time()

# Add caching
departures_df = departures_df.distinct().cache()

# noting how long the operation takes
print("Counting %d rows took %f seconds" % (departures_df.count(),

time.time() - start_time))
# Counting 139358 rows took 2.679007 seconds

------------------------------------------------------------------------
# noting the variance in time of a cached DataFrame
start_time = time.time()
print("Counting %d rows again took %f seconds" % (departures_df.count(),

time.time() - start_time))
# Counting 139358 rows again took 1.184970 seconds

------------------------------------------------------------------------
# Determine if is in the cache
departures_df.is_cached)

# Remove from the cache
departures_df.unpersist()
```

- Single node
- Standalone
- Managed
- YARN
- Mesos
- Kubernetes

1. Driver Process

- Task assignment
- Result consolidation
- Shared data access
- Tips:
- Driver node should have double the memory of the worker
- Fast local storage helpful

2. Worker Process

- Runs actual tasks
- Ideally has all code, data, and resources for a given task
- Tips:
- **More** worker nodes is often better than **larger** workers
- Test to find the balance
- Fast local storage extremely useful

- More objects better than larger ones
- Can import via wildcard

```
airport_df=spark.read.csv('airports-*.txt.qz')
```

- General size of objects
- Spark performs better if objects are of similar size

```
# use OS utilities / scripts (split, cut, awk)
split -l 10000 -d largefile chunk-

# to parquet
df_csv = spark.read.csv('singlelargefile.csv')
df_csv.write.parquet('data.parquet')
df = spark.read.parquet('data.parquet')

# read conf setting
spark.conf.get('configuration name')

        # write conf setting
spark.conf.set(<configuration name>)
```

Broadcasting

- Provides a copy of an object to each worker
- Prevents undue / excess communication between nodes
- Can drastically speed up .join() operations

```
from pyspark.sql.functions import broadcast
normal_count = df_1.join(df_2).count()
broadcast_count = df_1.join(broadcast(df_2)).count()

# Normal count:            119910        duration: 3.502130
# Broadcast count:       119910   duration: 1.712519
```

- Hides complexity from the user
- Can be slow to complete
- Lowers overall throughput
- Is often necessary, but try to minimize
- LIMIT:
- Limit use of `.repartition (num_partitions)`
- Use `.coalesce (num_partitions)` instead
- Use care when calling `.join()`
- Use `.broadcast()`
- May not need to limit it

2.3 Pipelines

1. Input(s)

> CSV, JSON, web services, databases

```
# Import the data to a DataFrame
departures_df = spark.read.csv('2015-departures.csv.gz', header=True)

# see the column names / order
departures_df.printSchema()

# root
#  |-- Date (MM/DD/YYYY): string (nullable = true)
#  |-- Flight Number: string (nullable = true)
#  |-- Destination Airport: string (nullable = true)
#  |-- Actual elapsed time (Minutes): string (nullable = true)

# Remove any duration of 0
departures_df = departures_df.filter(departures_df[3] > 0)

# Add an ID column
departures_df = departures_df.withColumn('id',
F.monotonically_increasing_id())
```

2. Transformations

> 2.1 Transformations : `.withcolumn()` , `.filter()` , `.drop()`

- Parse:

1. Incorrect data

- Empty rows
- Commented lines
- Headers

2. Nested structures

- Multiple delimiters:
- `200 300 affenpinscher;0`

3. Non-regular data

- Differing numbers of columns per row:
- `600 450 Collie;307 Collie;101 600 449 Japanese_spaniel;23`

4. Focused on CSV data

```
# Can remove comments using an optional argument
df1 = spark.read.csv('datafile.csv.gz', comment='#')

# Handles header fields:
# Defined via argument; Ignored if a schema is defined
df1 = spark.read.csv('datafile.csv.gz', header='True')

# Automatically create columns in a DataFrame based on sep argument df1 =
spark.read.csv('datafile.csv.gz', sep=',')

# Can still successfully parse if sep is not in string
df1 =
spark.read.csv('datafile.csv.gz', sep='*')
```

```
# Split _c0 on the tab character and store the list in a variable
split_cols = F.split(df1['_c0'], '\\t')

# folder, filename, width, height  split_df =
split_cols.withColumn('filename', split_cols.getItem(1))
```

3. Validation

```
# Rename the column
df1 = df1.withColumnRenamed('_c0', 'folder')

# Count the number of rows in split_df
split_count = split_df.count()

# Join the DataFrames
joined_df = split_df.join(F.broadcast(df1), folder', 'left_anti')
```

4. Output(s)

> CSV, Parquet, database

```
# Write the file out to JSON format
departures_df.write.json('output.json', mode='overwrite')
```

5. Analysis

- Analysis Calculations (UDF)

```
def getAvgSale(saleslist):
  totalsales = 0
  count = 0
  for sale in saleslist:
              totalsales += sale[2] + sale[3]
    count += 2
  return totalsales / count

udfGetAvgSale = udf(getAvgSale, DoubleType())
df = df.withColumn('avg_sale', udfGetAvgSale(df.sales_list))
```

```
df = df.withColumn('sq_ft', df.width * df.length)
df = df.withColumn('total_avg_size', udfComputeTotal(df.entries) /
df.numEntries)
```

3. Pandas

3.1 Spark2Pandas

- Spark DataFrames make that easy with the `.toPandas()` method. Calling this method on a Spark DataFrame returns the corresponding `pandas` DataFrame.

3.2 Pandas2Spark

- The `.createDataFrame()` method takes a `pandas` DataFrame and returns a Spark DataFrame.
- In the last exercise, you saw how to move data from Spark to `pandas`. However, maybe you want to go the other direction, and put a `pandas` DataFrame into a Spark cluster! The `SparkSession` class has a method for this as well.
- The `.createDataFrame()` method takes a `pandas` DataFrame and returns a Spark DataFrame.
- The output of this method is stored locally, not in the `SparkSession` catalog. This means that you can use all the Spark DataFrame methods on it, but you can't access the data in other contexts.
- For example, a SQL query (using the `.sql()` method) that references your DataFrame will throw an error. To access the data in this way, you have to save it as a *temporary table*.
- You can do this using the `.createTempView()` Spark DataFrame method, which takes as its only argument the name of the temporary table you'd like to register. This method registers the DataFrame as a table in the catalog, but as this table is temporary, it can only be accessed from the specific `SparkSession` used to create the Spark DataFrame.
- There is also the method `.createOrReplaceTempView()`. This safely creates a new temporary table if nothing was there before, or updates an existing table if one was already defined. You'll use this method to avoid running into problems with duplicate tables.
- Check out the diagram to see all the different ways your Spark data structures interact with

```
# Create spark_temp from pd_temp
spark_temp = spark.createDataFrame(pd_temp)

# Examine the tables in the catalog
print(spark.catalog.listTables())

# Add spark_temp to the catalog
spark_temp.createOrReplaceTempView("temp")

# Examine the tables in the catalog again
print(spark.catalog.listTables())
```