spark-intro

November 10, 2019

1 Spark introduction

[1]:
```
import re
import os
from collections import Counter
import numpy as np
import pandas as pd
from termcolor import cprint
import matplotlib.pyplot as plt

plt.style.use('seaborn')
%matplotlib inline

SMALL_SIZE = 10
MEDIUM_SIZE = 11
LARGE_SIZE = 12

plt.rc('font', size=SMALL_SIZE)          # controls default text sizes
plt.rc('axes', titlesize=SMALL_SIZE)     # fontsize of the axes title
plt.rc('axes', labelsize=MEDIUM_SIZE)    # fontsize of the x and y labels
plt.rc('xtick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('ytick', labelsize=SMALL_SIZE)    # fontsize of the tick labels
plt.rc('legend', fontsize=SMALL_SIZE)    # legend fontsize
plt.rc('figure', titlesize=LARGE_SIZE)   # fontsize of the figure title

def slide_print(text, color='white'):
    cprint(text, color, 'on_grey')
```

Sometimes a single machine simple cannot perform a given task fast enough. Sometimes there are too many tasks for a single machine to properly handle and yet othertimes there are so much data that the data must be distributed across resources. These scenarios describe several of the situations where the use of Apache Spark has the potential to directly impact a business opportunity. Whether you are working with spark locally, on Watson Studio, from within Docker or as part of a computer cluster the basics will cover in this video will still apply.

1.1 High performance computing

- Symmetric multiprocessing -Two or more identical processors connected to a single unit of memory.
- Distributed computing - Processing elements are connected by a network.
- Cluster computing - Group of coupled computers that work together in a way that they can be viewed as a single system.
- Massive parallel processing - Many networked processors usually > 100 used to perform computations in parallel.
- Grid computing - Distributed computing making use of a middle layer to create a `virtual super computer`.

  Spark is a cluster-computing framework. When you compare it to hadoop it essentially competes with which the MapReduce component of the Hadoop ecosystem.

There are many types of high performance computing environments. READ STATEMENT. Spark does not have its own distributed filesystem, but can use the Hadoop Distributed File System or HDFS. Spark uses memory and can use disk for processing, whereas MapReduce has strictly disk-based processing.

1.1.1 Spark Applications (overview)

Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program).

Here we have a diagram of Spark Application. When we start a spark environment a Spark Session is first created and this manages the driver process. The driver program can be controlled using API's in Scala, Python SQL, Java and R. The worker nodes on the right are usually distinct machines. Executors are the worker nodes' processes, each in charge of running and individual task and they are shown as the orange squares on the worker nodes. One cluster configuration is to assign several cores to each executor leaving one for additional overhead. The cluster manager, shown on the bottom, which is often YARN, Mesos or Kubernetes helps coordinate between the driver program and worker nodes.

1.1.2 Spark Applications (general process)

Spark applications are run as an independent sets of processes on a cluster coordinated by the SparkContext object in your main program. Each application gets its own executor processes, which remains allocated for the duration of application. The driver program, that encapsulates both the SparkContext and the SparkSession is used to submit Spark Applications in Spark .The driver program, once it is given instructions in the form of user code will then ask the cluster manager to launch executors.

1.2 SparkSession

- SparkContext
- SQLContext

- HiveContext
- StreamingContext

[2]:
```
import pyspark as ps
import random

spark = (ps.sql.SparkSession.builder
         .appName("sandbox")
         .getOrCreate()
         )

slide_print(spark.version)

sc = spark.sparkContext

for attribute in sc._conf.getAll():
    slide_print(attribute)
```

```
2.4.4
('spark.app.name', 'sandbox')
('spark.driver.memory', '1G')
('spark.master', 'local[4]')
('spark.driver.port', '43991')
('spark.sql.catalogImplementation', 'hive')
('spark.rdd.compress', 'True')
('spark.serializer.objectStreamReset', '100')
('spark.executor.id', 'driver')
('spark.submit.deployMode', 'client')
('spark.driver.host', 'localhost')
('spark.app.id', 'local-1573426719980')
('spark.ui.showConsoleProgress', 'true')
```

Prior to Spark 2.0 there were multiple points of entry for a Spark application, including: SparkContext, SQLContext, HiveContext, and the StreamingContext. More recent versions of Spark combine all of these objects into a single point of entry that can be used for all Spark applications. The SparkContext is a child process of the SparkSession. In this cell first we create a SparkSession using the SparkSession builder then we show how to access the Spark Context. Using the SparkContext we show here how to print some of the configuration properties.

1.3 RDD (Resilient Distributed Dataset)

1.3.1 An example workflow

1. Create the enviromnent to run spark from python
2. Extract RDDs or DataFrames from files
3. Carry out transformations
4. Execute actions to obtain values (local objects in python)

1.3.2 The RDD API has two types of operations:

- Transformations - Define a new dataset based on a previous one
- Actions - launch a job for execution on a cluster

[3]:
```
text_file = sc.textFile("../data/sherlock-holmes.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
slide_print(type(text_file))  # show that the data are read into an RDD
results = counts.collect()    # output the content in python (CAUTION)
slide_print(results[:7])      # print output
```

```
<class 'pyspark.rdd.RDD'>
[('', 3136), ('Project', 81), ('The', 272), ('of', 2723), ('Arthur',14), ('Conan', 4), ('is',
```

Theses four enumerated steps represent an example workflow. The code version of these steps are shown just below, with the exception of setting up the environment which we just did. Spark revolves around the concept of resilient distributed datasets or RDDs. An RDD is a fault-tolerant collection of elements that can be operated on in parallel. The RDD API, uses two types types of operations: transformations and and actions. On top of Spark's RDD API, high level APIs are provided, including the DataFrame API and Machine Learning API, both of which we will cover in this course. The text_file is an RDD that once subjected to a chain of transformations is used to create counts, another RDD. The action we use here is 'collect', which brings the data back into Python and it is something you should exercise caution with, especially when working with very large datasets.

1.3.3 To keep in mind

- Transformations are the main way of expressing business logic in Spark.
- Spark doesn't apply the transformation right away, it just builds on the DAG
- Transformations can be chained together
- RDDs are an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated on in parallel

[4]:
```
# an RDD from a an array
rand_nums = np.random.randint(0,500,500000)
rdd = sc.parallelize(rand_nums)
divis_by_3 = rdd.filter(lambda x: x % 3 == 0)

total1 = rdd.count()
total2 = divis_by_3.count()

print("total divisible by 3: {}/{}".format(total2,total1))
```

```
total divisible by 3: 166667/500000
```

There are several important things to remember about RDDs. The first is that they use what is known as lazy evaluation. This means that Spark will wait until the very last moment to

execute your transformations. It does this by constructing a Direct Acyclic Graph or DAG of the transformations. When an action like count or collect is called the task is sent for execution. Here we show another example on an RDD, but this time we create it using a numpy array with the parallelize function. The transformation is a simple filter and the actions are obtain counts rather than all of the data since we generally want to avoid pulling all data into local memory unless we have too.

1.4 Spark-submit

[5]:
```
%%writefile calculate-pi.py

import pyspark as ps
import random

spark = (ps.sql.SparkSession.builder
         .appName("get-pi")
         .getOrCreate()
        )
,
sc = spark.sparkContext
random.seed(1)

def sample(p):
    x, y = random.random(), random.random()
    return 1 if x*x + y*y < 1 else 0

count = (sc.parallelize(range(0, 10000000))
           .map(sample)
           .reduce(lambda a, b: a + b)
        )

result = {"pi": (4.0 * count / 10000000)}
print(result, file=open('calculate-pi-out.txt', 'w'))
```

```
Overwriting calculate-pi.py
```

```
~$ spark-submit calculate-pi.py
```

[6]:
```
import ast
with open("calculate-pi-out.txt") as file:
    results = file.read()
    results = ast.literal_eval(results)

slide_print(results)
```

```
{'pi': 3.1420044}
```

Applications can be submitted to a cluster of any type using the command spark-submit and an

accompanying script. The writefile magic function shown here saved the code in the cell as a python script to be called by spark-submit. The file calculate-pi, unsurprisingly calculates pi, but more importantly it shows an example of how to map a custom function over an RDD. Spark-submit can be run from the command line as shown. Generally, it is called with a number of options using a bash file, but it can be called using all the defaults as shown here. Once the script is run it will produce the outfile that is being printed here. This procedure could be used to make a prediction using a machine learning model that has been tuned, trained and saved.