Understanding SparkContext - Codealong

Introduction

SparkContext is the entry point for using the Unstructured API of Spark. In this lesson we'll go over how SparkContext works in PySpark, create a SparkContext called `sc`, and explore `sc`'s properties.

Objectives

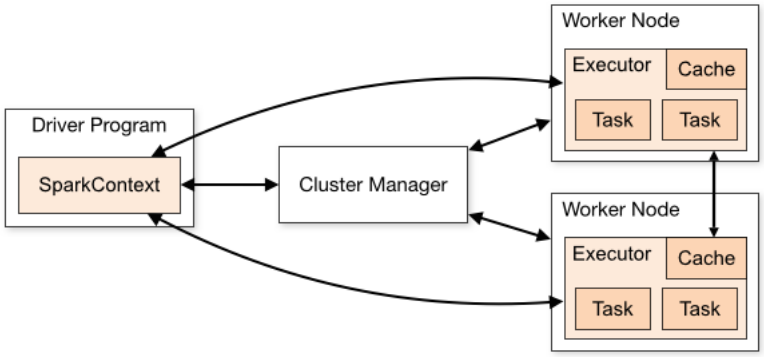
You will be able to:

- Define a SparkContext and why it is important to a Spark application
- Create a SparkContext with PySpark
- List the major properties and methods of SparkContext

The Purpose of the SparkContext

Spark Application Architecture

Recall this figure from the Cluster Mode Overview (https://spark.apache.org/docs/latest/cluster-overview.html):



When you are writing Spark code, your code is the "Driver Program" pictured here. Your code needs to instantiate a SparkContext if we want to be able to use the Spark Unstructured API.

PySpark Stack

Since we are not writing Spark code in Scala, but instead are writing PySpark code in Python, there is some additional architecture to be aware of.

Specifically, all Spark code needs to be able to run on the JVM (Java Virtual Machine), because **PySpark is built on top of Spark's Java API**. PySpark uses the Py4J (https://www.py4j.org/) library under the hood to accomplish this.

This is relevant to your development process because:

- Sometimes you will see error messages or warnings related to Java code.
- Many of the function and variable names follow Java naming conventions rather than Python. In particular, you will see many examples of `camelCase` names in places where you would expect `snake_case` Python names.

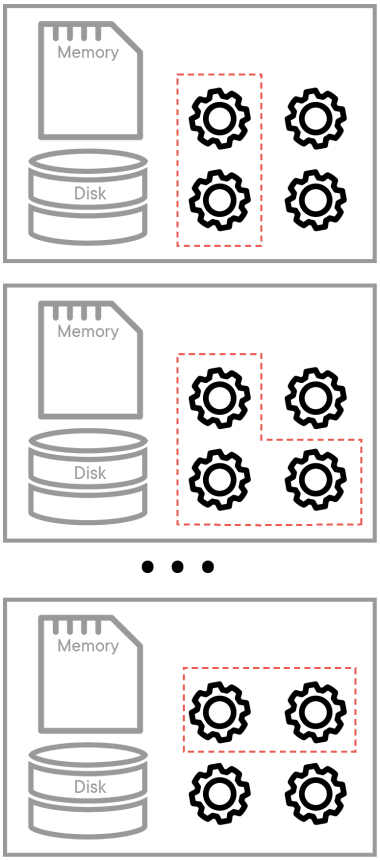
The architecture including Py4J is something like this (from the PySpark Internals wiki (https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals)):

The driver program launches parallel operations on executor Java Virtual Machines (JVMs). This can occur either locally on a single machine using multiple cores to create parallel processing or across a cluster of computers that are controlled by a master computer. When running locally, "PySparkShell" is the application name. The driver program contains the key instructions for the program and it determines how to best distribute datasets across the cluster and apply operations to those datasets.

The key takeaways for SparkContext are listed below:

- SparkContext is a client of Spark's execution environment and it acts as the master of the Spark application
- SparkContext sets up internal services and establishes a connection to a Spark execution environment
- The driver is the program that creates the SparkContext, connecting to a given Spark Master

After creation, SparkContext asks the master for some cores to use to do work. The master sets these cores aside and they are used to complete whatever operation they are assigned to do. You can visualize the setup in the figure below:



This image depicts the worker nodes at work. Every worker has 4 cores to work with, and the master allocates tasks to run on certain cores within each worker node.

Creating a Local SparkContext

While the SparkContext conceptual framework is fairly complex, creating a SparkContext with PySpark is fairly simple. All we need to do is import the relevant class and instantiate it.

Importing the SparkContext Class

As we can see from the documentation (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.SparkContext.html), there is an example import statement:

```
# Import the SparkContext class from the pyspark.context submodule
from pyspark.context import SparkContext
```

Type this code in the cell below and execute the cell.

In [1]:
```
# Import the SparkContext class from the pyspark.context submodule
from pyspark.context import SparkContext
```

Instantiating `sc`

Naming Convention

The conventional name for the SparkContext object is `sc`. In fact, in some (Py)Spark environments, there will already be an object in memory called `sc` as soon as the environment is loaded. Therefore unless you have a very specific reason for changing the name, you are strongly encouraged to use the name `sc` to represent the SparkContext.

Parameters

In theory you could simply call `SparkContext()` to create your SparkContext, but in practice you should specify values for two parameters: `master` and `appName`.

The `master` parameter is the cluster URL to connect to. If you were using a full-fledged Cluster Manager this URL might be something like `"mesos://host:5050"` but we are just running a local cluster. Therefore we'll specify a `master` value of `"local[*]"`. The `*` means that we are telling Spark to run on all available cores of our machine.

The `appName` parameter is just a label for the application. It's similar to a Python variable name -- just there to help you understand what the code is doing. You can put any string value you like.

Codealong

In the cell below, instantiate a variable `sc` using the `SparkContext` class, a `master` of `"local[*]"`, and an `appName` of `"sc practice"`.

```
# Instantiate sc
sc = SparkContext("local[*]", "sc practice")
```

In [2]:
```
# Instantiate sc
sc = SparkContext("local[*]", "sc practice")
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

22/11/21 17:32:44 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes w
here applicable
```

You may see some Java warnings appear below this line of code (or other lines of code). In general you can safely ignore these warnings, although they may provide relevant information for debugging.

One SparkContext at a Time

Note that you can only have one SparkContext at a time. If you try to make another one without stopping the first one, you will get an error:

In [3]:
```
# Bad idea - creating a second SparkContext
try:
    another_sc = SparkContext("local[*]", "double trouble")
except Exception as e:
    print(type(e))
    print(e)
```

```
<class 'ValueError'>
Cannot run multiple SparkContexts at once; existing SparkContext(app=sc practice, master=local[*]) created by __init__ at /tmp/
ipykernel_6297/3439793671.py:2
```

Properties and Methods of SparkContext

Now we have a SparkContext object! Let's investigate it like any other Python object.

Type

What is the type of our SparkContext?

Click to Reveal Code

In [4]:
```
# Type of sc
type(sc)
```

Out[4]:
```
pyspark.context.SparkContext
```

Click to Reveal Expected Output

All Attributes

Use Python's `dir` built-in function (documentation here (https://docs.python.org/3/library/functions.html#dir)) to get a list of all attributes (including methods) accessible through the `sc` object.

Click to Reveal Code

In [5]:
```
# Get a list of all attributes
dir(sc)
```

Out[5]:
```
['PACKAGE_EXTENSIONS',
 '__annotations__',
 '__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__enter__',
 '__eq__',
 '__exit__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
```

Click to Reveal Expected Output

Python Help

We have a list of attributes, but no explanation of how to use them. Use Python's `help` function (documentation here (https://docs.python.org/3/library/functions.html#help)) to get an easier-to-read list of all the attributes, including examples, that the `sc` object has.

Click to Reveal Code

In [6]:
```
# Use Python's help function to get information on attributes and methods for sc object
help(sc)
```

```
Help on SparkContext in module pyspark.context object:

class SparkContext(builtins.object)
 |  SparkContext(master: Optional[str] = None, appName: Optional[str] = None, sparkHome: Optional[str] = None, pyFiles: Optio
nal[List[str]] = None, environment: Optional[Dict[str, Any]] = None, batchSize: int = 0, serializer: 'Serializer' = CloudPick
leSerializer(), conf: Optional[pyspark.conf.SparkConf] = None, gateway: Optional[py4j.java_gateway.JavaGateway] = None, jsc:
Optional[py4j.java_gateway.JavaObject] = None, profiler_cls: Type[pyspark.profiler.BasicProfiler] = <class 'pyspark.profiler.
BasicProfiler'>, udf_profiler_cls: Type[pyspark.profiler.UDFBasicProfiler] = <class 'pyspark.profiler.UDFBasicProfiler'>)
 |
 |  Main entry point for Spark functionality. A SparkContext represents the
 |  connection to a Spark cluster, and can be used to create :class:`RDD` and
 |  broadcast variables on that cluster.
 |
 |  When you create a new SparkContext, at least the master and app name should
 |  be set, either through the named parameters here or through `conf`.
 |
 |  Parameters
 |  ----------
 |  master : str, optional
```

Click to Reveal Expected Output

Investigating Specific Attributes

Refer to the PySpark documentation (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.SparkContext.html) to find the appropriate attributes to answer these questions.

Spark Version

What version of Spark is the application running?

Click to Reveal Code

In [7]:
```
# Spark version
sc.version
```

Out[7]:
```
'3.3.1'
```

Start Time

What time was the Spark Context created?

Click to Reveal Code

In [8]:
```
# Start time
sc.startTime
```

Out[8]:
```
1669051964640
```

Note that this is the epoch time so it will appear as a large integer.

All Configuration Settings

We can access the complete configuration settings (including all defaults) for the current SparkContext by chaining together the `getConf()` method and the `getAll()` method.

Click to Reveal Code

In [9]:
```
# All configuration settings
sc.getConf().getAll()
```

Out[9]:
```
[('spark.driver.extraJavaOptions',
  '-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke=ALL-UNNA
MED --add-opens=java.base/java.lang.reflect=ALL-UNNAMED --add-opens=java.base/java.io=ALL-UNNAMED --add-opens=java.base/java.ne
t=ALL-UNNAMED --add-opens=java.base/java.nio=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED --add-opens=java.base/jav
a.util.concurrent=ALL-UNNAMED --add-opens=java.base/java.util.concurrent.atomic=ALL-UNNAMED --add-opens=java.base/sun.nio.ch=AL
L-UNNAMED --add-opens=java.base/sun.nio.cs=ALL-UNNAMED --add-opens=java.base/sun.security.action=ALL-UNNAMED --add-opens=java.b
ase/sun.util.calendar=ALL-UNNAMED --add-opens=java.security.jgss/sun.security.krb5=ALL-UNNAMED'),
 ('spark.app.name', 'sc practice'),
 ('spark.app.startTime', '1669051964640'),
 ('spark.executor.id', 'driver'),
 ('spark.app.id', 'local-1669051965711'),
 ('spark.app.submitTime', '1669051964425'),
 ('spark.driver.host',
  'w-bonfa-ds-course-phase4-b2cd3769c3d84600a27d1497e1fdb428-kpc2b'),
 ('spark.rdd.compress', 'True'),
 ('spark.executor.extraJavaOptions',
  '-XX:+IgnoreUnrecognizedVMOptions --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.lang.invoke=ALL-UNNA
MED --add-opens=java.base/java.lang.reflect=ALL-UNNAMED --add-opens=java.base/java.io=ALL-UNNAMED --add-opens=java.base/java.ne
t=ALL-UNNAMED --add-opens=java.base/java.nio=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED --add-opens=java.base/jav
a.util.concurrent=ALL-UNNAMED --add-opens=java.base/java.util.concurrent.atomic=ALL-UNNAMED --add-opens=java.base/sun.nio.ch=AL
L-UNNAMED --add-opens=java.base/sun.nio.cs=ALL-UNNAMED --add-opens=java.base/sun.security.action=ALL-UNNAMED --add-opens=java.b
ase/sun.util.calendar=ALL-UNNAMED --add-opens=java.security.jgss/sun.security.krb5=ALL-UNNAMED'),
 ('spark.serializer.objectStreamReset', '100'),
 ('spark.master', 'local[*]'),
 ('spark.submit.pyFiles', ''),
 ('spark.submit.deployMode', 'client'),
 ('spark.driver.port', '35045'),
 ('spark.ui.showConsoleProgress', 'true')]
```

Shutting Down the SparkContext

When you are finished using a SparkContext, be sure to call the `stop` method. This will allow you to create another SparkContext in the future.

Click to Reveal Code

In [10]:
```
# Shut down SparkContext
sc.stop()
```

Once shut down, you can no longer access Spark functionality before starting a new SparkContext.

In [11]:
```
try:
    sc.version
except Exception as e:
    print(type(e))
    print(e)
```

```
<class 'AttributeError'>
'NoneType' object has no attribute 'version'
```

Additional Resources

- Apache Spark Context (https://data-flair.training/blogs/learn-apache-spark-sparkcontext/)

Summary

In this codealong, we saw how SparkContext is used as an entry point to Spark applications. We learned how to start a SparkContext, how to list and use some of the attributes and methods in SparkContext and how to shut it down. Students are encouraged to explore other attributes and methods offered by the `sc` object. Some of these, namely creating and transforming datasets as RDDs, will be explored in later labs.