Apache Spark

1 What is Spark?

- "A fast and general engine for large-scale data processing"
- Like any other Hadoop based technology it is scalable

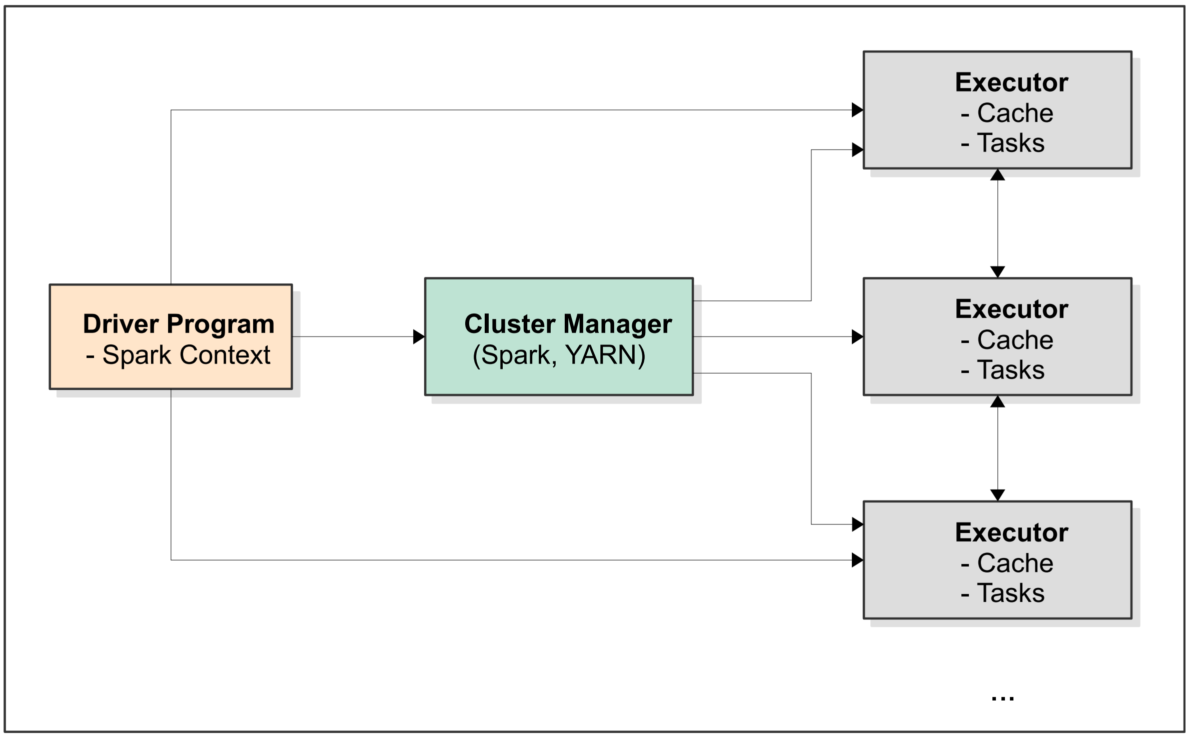

Illustration 1: Architecture of a Spark Application

In a Spark application there will be a ***Driver Program***, a small script that controls what is going to happen in the Spark job. Then it connects to a ***Cluster Manager*** — Spark can run on Hadoop in which case it can use YARN, but it need not to always run on Hadoop; and in that case Spark can use *own built-in cluster manager* or another cluster manager like Apache Mesos. The role of the cluster manager is to distribute the job across the entire cluster of commodity computers to process the data in parallel in a set of one or more ***Executor*** processes. Each executor process has some *cache* and is responsible for some given task. This cache is the key item for Spark performance, which, unlike a disk based system, tries keep as much of its computation as it can in RAM.

- It is really fast, about 100 times faster than Hadoop MapReduce in memory. For speed, apart from the cache, Spark also uses a built-in *DAG engine* that optimizes workflows.
- Coding with Spark is also not that difficult — we can use programming languages like Python, Java or Scala to code some really complex analysis on our cluster. Spark itself is written in Scala and hence in production environment it is recommended to use Scala with Spark. Scala compiles to Java byte code and it's functional programming model is a good fit for distributed processing and therefore gives better performance.
- Built around one main concept: *Resilient Distributed Dataset* (*RDD*). This is simply an object representing a data set and we can various functions on that RDD object to transform or reduce it or even analyse it in whatever way we want to produce new RDDs. In 2016 with Spark 2.0 *data sets* have been introduced that are SQL like entities built on top of RDDs.

2 Components of Spark

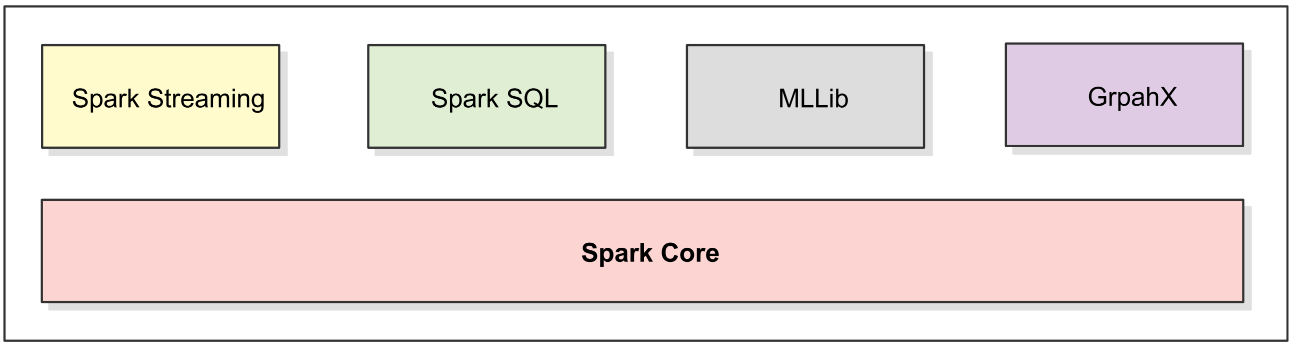

Illustration 2: Components of Spark

There are many important libraries that are built on top of Spark Core and comes packaged with Spark:

- ***Spark Streaming***: Allows processing of input data that, instead of batch, comes in real-time as streams, for example, ingestion of continuously produced logs from a set of web servers, and analyse the same across some window of time outputting the results of analysis to a database or a NoSQL data store.
- ***Spark SQL***: A SQL interface to Spark — uses data sets.
- ***MLLib***: A suit of machine learning and data mining libraries that run on a Spark data set.
- ***GraphX***: An extensible way to build and analyse a graph, of say, on social media platform showing the connections between the people in the platform.

3 RDD

RDDs are just sort of way of storing key-value information or any information that can automatically do the right thing on a cluster. From programming stand point these are just data sets but under the hood those resilient and distributed. The A `SparkContext`, a sort of environment that the driver program runs within Spark, creates RDDs. It is also responsible for making the RDDs resilient and distributed. The Spark shell by default creates an `sc` object. It can also be created from inside a stand-alone script.

3.1 Creating RDD

There are several ways to create RDDs:

- A mock creation: `nums = parallelize([1, 2, 3, 4])`
- Loading data from a local file

  `sc.textFile("file:///c:/users/kaushikd/some-input-text.txt")`

  - or from Amazon S3 (`s3n://`) or HDFS (hdfs`://`)
- From a Hive context like

  `hiveCtx = HiveContext(sc)`

  `rows = hiveCtx.sql("SELECT name, age FROM users")`
- or from:
  - JDBC – any database connected to JDBC
  - Cassandra
  - Hbase
  - ElasticSearch
  - JSON, CSV, sequence files, object files, various compressed formats

3.2 Transforming RDD

- map – apply some input function to *every* input row of RDD and create a new transformed RDD; 1-to-1 relation ship of 1 row of input RDD to 1 row of output RDD.

```
rdd  = sc.paralleize([1, 2, 3, 4])
squaredRdd = rdd.map(lambda x: x*x) # This yields [1, 4, 9, 16]
```

Illustration 3: map example

- flatmap – an *m*-to-*n* relation ship of *m* rows of input RDD to *n* rows of output RDD, like split each input row to multiple rows or discarding few input rows.

- filter
- distinct
- sample – get random sample(s) from input RDD
- union, intersection, subtract, cartesian

3.3 RDD actions

These are actually the reduce functions. Possible actions are:

- collect – takes whatever is in the RDD and return that to a Python object
- count
- countByValue
- take
- top
- reduce
- ... and more

In Spark nothing actually happens in the driver program until an action is called (***Lazy evaluation***) and only when an action is invoked Spark works backward to figure out the fastest way to achieve the desired result. Till then it keeps building the graph with chain of dependencies within the driver script.

4 Using RDDs with Spark

Example: Find worst rated movies in the movie lens database

```
from pyspark import SparkConf, SparkContext

'''
This function just creates a Python dictionary that we use later to
convert movie ID's to movie names while printing the final result
'''
def loadMovieNames(): 1
    movieNames = {}
    with open('ml-100k/u.item') as f:
        for line in f:
            fields = line.split('|')
            movieNames[int(fields[0])] = fields[1]
    return movieNames

'''
This function takes each line of u.data and converts that to
(movieID, (rating, 1.0)). This way we can then add up all the ratings
for each movie and the total number of ratings for each (which lets
up compute the average)
'''
```

```
def parseInput(line):                                          2
    fields = line.split()
    return (int(fields[1]), (fields[2], 1.0)) # 1.0 is the frequency of this rating

# The main function
def main():
    '''
    Using SparkConf we can configure how the job will distributed or how much RAM the executor
     should have, etc. But for now let us just set the application name. We can see the
     application in the below given name in Ambari.
    '''
    conf = SparkConf.setAppName('WorstMovies')
    sc = SparkContext(conf=conf) # Create our Spark context                   3

    # Load uo movieID -> movie name lookup table                         4
    movieNames = loadMovieNames()

    # Load up the raw u.data file
    lines = sc.textFile('hdfs:///user/maria_dev/ml-100k/u.data')

    # Convert to (movieID, (rating, 1.0))                    5
    movieRatings = lines.map(parseInput)

    # Reduce to (movieID, (ratingTotal, ratingCount))            6
    ratingTotalAndFreq = movieRatings.reduceByKey(lambda ratingAndFreq1, ratingAndFreq2: \
                                                  (ratingAndFreq1[0] + ratingAndFreq2[0] \
                                                  , ratingAndFreq1[1] + ratingAndFreq2[1]))

    # Map to (movieID, averageRating)
    averageRatings = ratingTotalAndFreq.mapByValues(lambda ratingSumAndFreq: \         7
                                                    ratingSumAndFreq[0] / ratingSumAndFreq[1])

    # Sort by average rating
    sortedMovies = averageRatings.sortBy(lambda x: x[1])           8

    # Take the top 10 results
    results = sortedMovies.take(10)

    # Print top 10 (i.e., bottom 10)
    for result in results:
        print(movieNames[result[0]], result[1])


if __name__ == "__main__":
    main()
```

Below is the explanation for the above code:

1. `loadMovieNames` is a function that creates an in-memory lookup table to map a movie ID to its name. This is used while printing the result.
2. `parseInput` function is used as the mapping function to map each line of the `u.data` file.
3. We create the Spark context for our job.
4. Create the in-memory lookup table by calling `loadMovieNames`.
5. Each line of the `u.data` file loaded to the RDD `lines` is mapped to a key-value pair (*movieID*, (*rating*, 1.0)), where *movieID* is the key and the tuple (*rating*, 1.0) is the value, in the output RDD `movieRatings` by the `parseInput` function. In the tuple (*rating*, 1.0) 1.0 is the frequency of *rating*.
   We do such a mapping so that we can just compute the sum of all the ratings and sum the frequencies (i.e., 1.0s)

for a particular movie and then we can just divide the sum of ratings by the sum of frequencies to get the average rating for that movie.

6 We apply the reduce operation `reduceByKey` on the RDD `movieRatings`. The `reduceByKey` operation aggregates all the *key-value* pairs of `movieRatings` RDD by their key which is *movieID* and then the lambda function is used to combine two values for a particular key. Here the combination operation is to sum the ratings and frequencies from two input values `ratingAndFreq1` and `ratingAndFreq2` which are the tuples (*rating*, 1.0) corresponding to two ratings for the key *movieID*. Therefore, the output RDD `ratingTotalAndFreq` will have key-value pairs (*movieID*, (*ratingTotal*, *ratingCount*)) where *movieID* is the key and the tuple (*ratingTotal*, *ratingCount*) is the value.

7 The `mapByValues` function invoked on the RDD `ratingTotalAndFreq` maps the value for each key-value pair (*movieID*, (*ratingTotal*, *ratingCount*)) that it contains to *averageRating* for that *movieID* using the lambda function and outputs another RDD `averageRatings` containing key-value pairs (*movieID*, *averageRating*).

8 The key-value pairs of the RDD `averageRatings` is sorted based on the value (i.e., *averageRating*). We finally take the top 10 rows from the RDD `sortedMovies` and print their names and ratings by converting *movieID* to its name using the local dictionary `movieNames`.