Course: [Taming Big Data with Apache Spark and Python - Hands On!]
Start Date: 03/01/2021

Env: *~/projects/spark-course (requires 3.6.12)*
Start spark: *docker-compose up*
Pyspark console: *pyspark --master spark://localhost:7077*
Run on cluster from console: *spark-submit ../spark-apps/ratings-counter.py*

RDD = Resilient Distributed Dataset
rdd = sc.textFile('spark-data/Vermont_Vendor_Payments.csv')

What is it?

- Fast/general engine for large scale data processing
- Distributes work across multiple workers
- Very scalable (runs in a cluster) via cluster manager
- 100x faster than hadoop with MapReduce
- Uses a DAG engine
- Can code in python, java or scala
- Built around one main concept - RDD (resilient distributed dataset)

Components of spark

- Spark Core
    - Spark Streaming
        - Allow for streaming logs into a dataset in realtime
    - Spark SQL
        - Run spark on top of a hive context / use structured data
        - Allow you to use spark as a data warehouse of sorts
    - MLLib
        - Tools for running machine learning algos
    - GraphX (deprecated) - Replaced by SparkGraph
        - Network theory sort of stuff
        - Social graph. Etc

Scala is the more popular way to write programs for spark. Spark is written in scala so is “native”. Python is fine but scala is probably better.

RDD: Resilient Distributed Dataset

This is the core of what spark does (but seems to be getting replaced by datasets??)

Dataset
It's an abstraction for a big dataset

Distributed
Can be spread out across many nodes for processing

Resilient
Worker node failures will be rerun by the manager

The Spark Context

Called *sc* in the shell - and created automatically

Transforming RDDs

- map()
  - take some data and transform it to something else given a function
  - Maps everything from the original rdd 1 to 1 to the new rdd
- flatmap()
  - Similar to map() but allows you to produce multiple values for every value in the original RDD
- filter()
  - Filter out info you don't need
- distinct()
  - Returns unique values
- sample()
  - Take a random sample - good for testing
- union()
  - Join two datasets
- intersection()
- subtract()
- cartesian()

RDD Actions

- collect()
  - Grab all values in the dataset
- count()
  - Count all values
- countByValue()
  - Breakdown by each unique value
- take()
  - Sample a few values from the dataset
- top()
  - Get the top x values
- reduce()
  - Reduce down
- map()
  - Map to a key value pair - rdd.map(lambda x: (x, 1)
  - Note: If your not going to transform keys you should use mapValues() or mapFlatValues() as it's more efficient
- flatMap()
  - Split into multiple entries for every element in an existing rdd
  - Breaking up lines into words is a good example
- reduceByKey()
  - Combine things together for the given ken in a key value pair dataset
  - rdd.reduceByKey(lambda x, y: x + y) will add up all values where the key is x
- groupByKey()
  - Group values for the same key
- sortByKey()
  - Sort rdd by keys
- keys(), values()
  - Return an rdd of just keys or values

Lazy Evaluation

Nothing actually happens in your driver program until an action is called on the dataset!

SparkSQL

- Extends RDD to DataFrame
- Data Frame
  - Can run SQL queries against them (across a cluster)
  - Contains Row objects
  - Can have a schema (for more efficient storage)
  - Allows for importing structured formats (json, JDBC/ODBC)
- Need to use SparkSession rather than SparkContext

Using in Python
spark = SparkSession.builder.appName('App Name').getOrCreate()
inputData = spark.read.json(<path to file>)
inputData.createOrReplaceTempVIew('MyTableName')
df = spark.sql('SELECT foo FROM bar ORDER BY foobar')

Can also run selects on the data frame itself
df.select('<some field name>')
df.filter(df('<some field name') > 200).mean()
df.rdd().map(<mapper function>)

Show results
df.show(<optional number of rows>)

Broadcast object allows you to share a common object (incl. functions) across all nodes in a cluster

Accumulator allows you to share a variable across all nodes in a cluster

Use .cache() or .persist() to cache a dataset if you are going to be accessing it multiple times in a script (persist caches to disk)

AWS

Spin up an **E**lastic **M**ap **R**educe service - which is actually an Hadoop cluster - and includes spark.

You will then need to spin up an EC2 instance to connect to your cluster.

Partitioning

- Spark is not magic. You need to think about how your data is partitioned
- Use partitionBy() on an RDD running a large operation (that could benefit from partitioning).
- Actions that could benefit from partitions:
    - join()
    - cogroup()
    - groupWith()
    - leftOuterJoin()
    - rightOuterJoin()
    - groupByKey()
    - reduceByKey()
    - combineByKey()
    - lookup()
- You want at least as many partitions as will fit in your cluster - maybe 100 across a cluster of 10 workers.

- Use a default SparkConf - don't supply config in the python script. This means it will automatically use the hadoop yarn cluster.
-

Troubleshooting

Spark console runs at :8080?? When running locally. CAn see thread dumps, executor/env information and more.

There is no console access on EMR

If you get errors that executors are failing to issue heartbeats - it usually means you are giving it too much work - so throw more machines at it or each executor needs more memory. Or increase the partitions to split the job up into smaller bits.

Logs
Available in the console above
In yarn logs are distributed so you need to collect them after they've run

Machine learning

- Limited to the algorithms that are built in only
- Mllib is deprecated - use ML/dataframes

Spark Streaming

- Analyse continuous streams of data (logs/twitter/files added to a directory)

The old way (D streaming)
- Data is aggregated and analysed at some given interval
- Can take data from a port
- Uses "checkpointing" to store state to disk periodically
- By default your rdd would only contain a little chunk of incoming data. "**Windowed operations**" can combine results from multiple batches over some sliding time window
  - window()
  - reduceByWindow()
  - reduceByKeyAndWindow()
- updateStateByKey()
  - Allows you to maintain state across many batches over time
  - E.g. running counts of some event

The new way (Structured streaming)
- Models streaming as a dataframe that just keeps growing over time

Window = how long to look into the past (e.g. 10 minutes)
Slide interval = how often to evaluate the window (e.g. 5 minutes)
This would mean every 5 minutes process the last 10 minutes of data

GraphX

- Only supported by scala currently - python support not likely anytime soon
- Not widely used
- Can be used for things like “connectedness”, degree distribution, average path length, triangle counts
- Can join and transform large graphs quickly
- Mainly used for network analysis