ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark Program

CHAPTER 2: INTRODUCTION TO SPARK

Chapter Objectives

In this chapter, we will :

- Review the history of Apache Spark
- Look at the architecture and components of Apache Spark
- Load files into RDD
- Process RDD using actions and transformation

Introduction to Apache Spark

- Apache Spark is a computing engine that can be used for large-scale data processing
  - Spark 2 can perform between 100X and 1000X faster than Hadoop's default computing engine (MapReduce)
  - Created by Matei Zaharia at UC Berkley in 2009
  - Donated to Apache Software Foundation in 2013
  - Apache Spark has seen immense growth over the past several years
- Spark functionality includes the ability to:
  - Perform iterative processing
  - Work with structured data via SQL
  - Support Hive Query Language (HQL)
  - Interact with it via a command-line shell
  - Support near real-time processing using in-memory data structure
- See https://spark.apache.org/

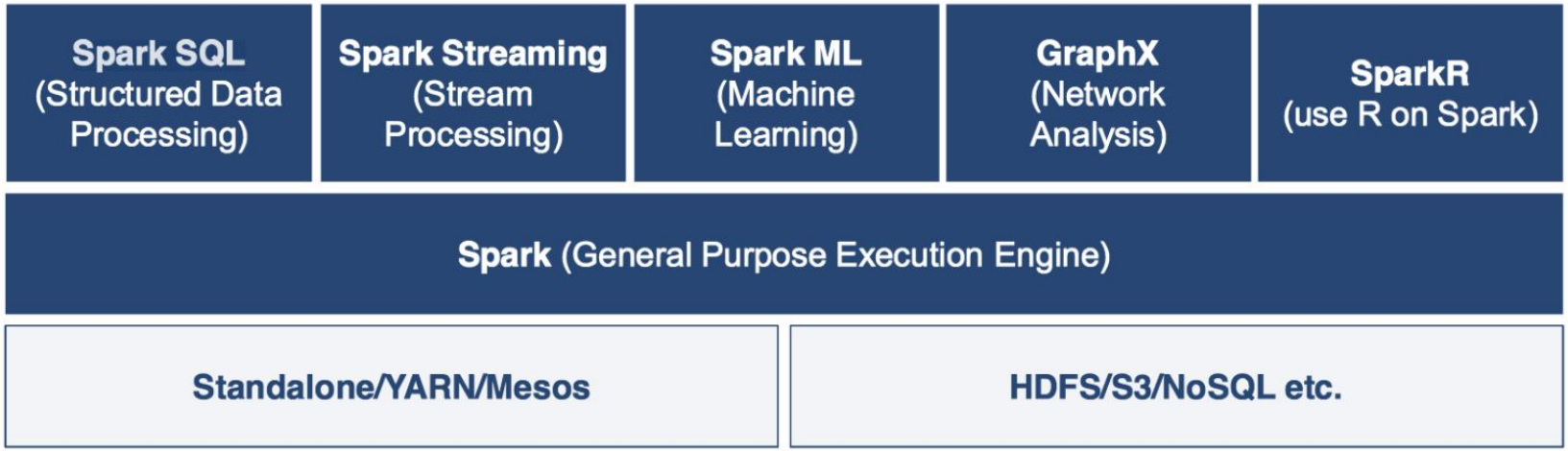
Introduction to Apache Spark (continued)

- Apache Spark provides high-level APIs in Java, Scala, Python, and R and has an optimized engine that supports general execution graphs
- Two important use cases for Apache Spark are data processing and AI
- Spark unifies data processing and AI by providing a powerful in-memory execution engine
- It also offers popular AI frameworks and libraries such as TensorFlow, R, and SciKit-Learn

Speed

- Run computations in memory
- Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing
- 100 times faster in-memory and 10 times faster even when running on a disk than MapReduce

Spark Components



Spark Core

- Spark Core is the underlying general execution engine for the Spark platform, all other functionality is built on top of it
- Provides distributed task dispatching, scheduling, and basic IO functionalities exposed through an application programming interface centered on the RDD, which is Spark's primary programming abstraction

Spark SQL

- Spark package designed for working with structured data which is built on top of Spark Core
- Provides an SQL-like interface for working with structured data
- More and more Spark workflow is moving towards Spark SQL

Spark Streaming

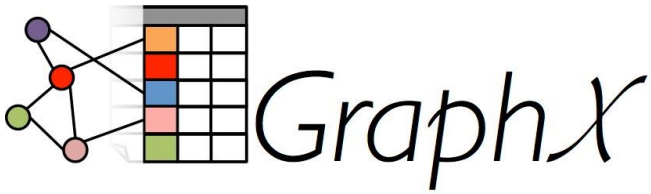
- Running on top of Spark, Spark Streaming provides an API for manipulating data streams that closely match the Spark Core's RDD API
- Enables powerful interactive and analytical applications across both streaming and historical data while inheriting Spark's ease of use and fault tolerance characteristics

Spark MLlib

- Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms and blazing speed
- Usable in Java, Scala, and Python as part of Spark applications
- Consists of common learning algorithms and utilities including classification, regression, clustering, collaborative filtering, and dimensionality reduction, etc.

GraphX

- A graph computation engine built on top of Spark that enables users to interactively create, transform, and reason about graph-structured data at scale
- Extends the Spark RDD by introducing a new graph abstraction
  - A directed multigraph with properties attached to each vertex and edge



Spark Application Top-Down View

- Spark executes *applications*
  - Either in local, stand-alone mode, or as a cluster
- Spark applications have one *driver* and one or more *executors*
  - Drivers and executors run as Java processes
  - Drivers assign *tasks* to the executors
  - Executors run tasks on *Resilient Distributed Datasets* (RDDs)
  - Executors send results to the driver
- Drivers include:
  - Spark Shell
    - PySpark—the Python shell
    - Spark Shell—the Scala shell
  - Custom program
    - Written in Python, Java, or Scala

Resilient Distributed Datasets (RDDs)

- RDDs represent the core data construct of Spark
  - RDDs are immutable
  - RDDs are fault tolerant (resilient)
    - When nodes or tasks fail, RDDs are reconstructed on other nodes
  - RDDs are split into *partitions* and can be distributed to any executor
  - RDDs can contain any kind of data
    - Prefer data that can be partitioned
- RDDs are objects that support two categories of operations
  - Transformations
    - Create new RDDs from existing RDDs
    - Always return RDDs
    - Lazily evaluated
  - Actions
    - Start computations
    - Return results to the driver
    - Save results to disk
    - Never return RDDs

Spark Application Flow

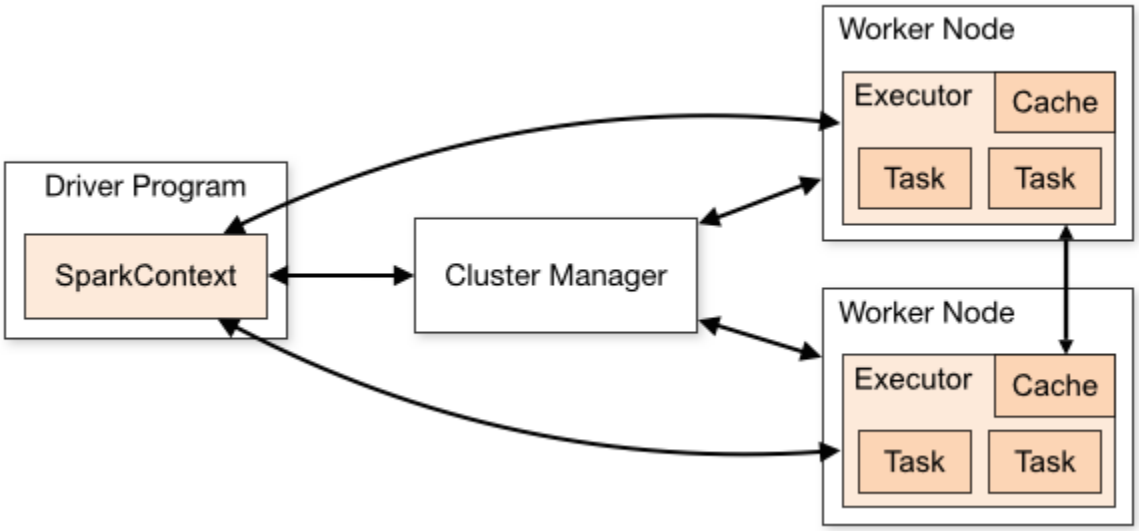
- Spark applications tend to have a similar flow
  1. Create a Spark context
     a. Automatically provided in the shells via the variable "sc"
  2. Import data as RDDs
  3. Transform and perform actions on RDDs
  4. Export results
- Spark applications are not declarative in nature, however, they are still coded at a high level of abstraction
  - In contrast, $tasks$ are at a very low level of abstraction
  - Tasks are not created by programmers, rather they are created at runtime by Spark

Drivers and Executors

- Spark applications consist of a driver process and a set of executor processes
- The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things:
  - Maintaining information about the Spark application
  - Responding to a user's program or input
  - Analyzing, distributing, and scheduling work across the executors (defined momentarily)
- The driver process is absolutely essential—it's the heart of a Spark application and maintains all relevant information during the lifetime of the application
- The executors are responsible for actually executing the work that the driver assigns them. This means, each executor is responsible for only two things:
  - Executing code assigned to it by the driver, and
  - Reporting the state of the computation, on that executor, back to the driver node

Spark's Basic Architecture

- The cluster manager controls physical machines and allocates resources to Spark applications
- This can be one of several core cluster managers: Spark's standalone cluster manager, YARN, or Mesos
- This means that there can be multiple Spark applications running on a cluster at the same time



Start PySpark

- To start PySpark on the VM:
  - Open a terminal window and type the following commands:

```
cd ~/ROI
pyspark
sc
spark
x = sc.textFile('datasets/text/shakespeare.txt')
x.count()
x.take(10)
```

- To write a Python program from scratch you have to initialize sc and Spark manually
  - initspark.py is a helper module you can copy and use in your own scripts

```
from initspark import *
sc, spark, conf = initspark()
sc, spark, conf = initspark(appname = 'appname',
servername = 'sparkservername', cassandra = 'cassandra)
```

Load Data

- The sc object is the Spark context and allows you to call methods to load and manipulate data

```
x = sc.parallelize(range(1, 11))
x.collect()
x.take(5)
sc.textFile('hdfs://localhost:9000/categories').collect()
sc.textFile('hdfsPath('categories').collect()
```

- Load a local file

```
sc.textFile('file:///home/student/ROI/datasets/northwind/
CSV/categories/categories.csv')
```

- Load a local folder

```
sc.textFile('file:///home/student/ROI/datasets/northwind/
CSV/categories/categories.csv')
```

- Load a hdfs folder

```
sc.textFile('hdfs://localhost:9000/categories')
sc.textFile(hdfsFolder('categories'))
```

Actions and Transformations

- Once you have an RDD, you can invoke methods on it
- Methods can either be:
  - An action which causes it to do some work and possibly return data back to the client
  - A transformation which is lazy evaluated and is only run when an action is called
- Transformations can be chained together to create multiple operations on the data but none are executed until an action is called. This allows the entire chain of transformations to be internally optimized by Spark before execution
- Transformations can also be either:
  - Narrow: can operate on the data in a single node
    - Like a map operation in MapReduce
  - Wide: requires data with the same key to be shuffled around to the same nodes
    - Like a reduce operation in MapReduce

Processing Data

- The data is loaded into an RDD (Resilient Distributed DataFrame)
  - Very similar to a Python list except it is spread across many nodes in the cluster
  - Has many built-in methods to process the data
- Loading data from a text file basically creates a list of strings
- Some useful actions to look at the data are:
  - `rdd.collect()` – returns the entire RDD as a Python list to the client
  - `rdd.count()` – returns a count of how many items are in the RDD
  - `rdd.take(x)` – returns `x` number of items from the RDD as a list
  - `rdd.takeOrdered(x, key=function)` – returns `x` rows of an RDD after sorting it first using a function
  - `rdd.top(x, key=function)` – returns the opposite of takeOrdered
  - `rdd.takeSample(replacement, count, seed)` - returns a sample of a larger data set
  - `rdd.foreach(function)` – executes the function once for each element of the RDD

Saving Data

- There are a lot of methods to save data to different formats
  - `rdd.saveAsTextFile()` – saves the RDD as a plain text file
  - `rdd.saveAsHadoopFile()` – saves the RDD as a key/value pair file suitable for Hadoop
  - `rdd.saveAsSequenceFile()` – saves the RDD as a Hadoop sequence file
  - `rdd.saceAsPickleFile()` – saves the RDD as a Python pickle file

Transformations

- Transformations are used to create a recipe of changes you want to make to the data
  - String parsing, data conversion, calculations
  - Filtering
  - Matching
  - Sorting
  - Aggregating
- Some useful transformations:
  - Narrow transformations
    - `rdd.map()` – applies a function to each element of the RDD
    - `rdd.flatMap()` – applies a function and flattens the elements
    - `rdd.filter()` – applies a function to determine if an element is returned
  - Wide transformations
    - `rdd.sort()` – orders the RDD
    - `rdd.groupBy()` – accumulates items with a key into a tuple of the key and list of the items
    - `rdd.reduce()` – runs a function on items for a key to return an aggregated value
    - `rdd.join()` – matches elements in one RDD to another

Lambda

- Many actions and transformations take a function as a parameter to allow customization of how the method works
- You could pass it a function name if you have one defined, but in many cases the functions are trivial
- Python allows you to create a function on the fly that can be passed as a parameter without the need to create the function in advance
- If all you need to do is create a simple function that takes one or more parameters and return a calculation that can be done in a single statement, then a lambda is a good choice

```
def isEven(x):
    return x % 2
isEven = lambda x : x % 2

rdd.filter(isEven)
rdd.filter(lambda x: x % 2)
sc.parallelize(range(1, 11)).sortBy(lambda x : (x % 2, x))
```

Project

The `creditcard.csv` dataset provides sample data on credit card transactions.

- Load the file into HDFS
- Load the file into an RDD
- Parse the file into a tuple or namedtuple or dictionary
  - Make sure to convert columns to the right data types
  - You can ignore any columns you don't need for the solution
- Filter the data to show only transactions made by women
- Calculate the amount spent in each city

Chapter Summary

In this chapter, we have:

- Reviewed the history of Apache Spark
- Looked at the architecture and components of Apache Spark
- Loaded files into RDD
- Processed RDD using actions and transformation