Spark Program

Chapter 1: Introduction to HDFS & Spark

Chapter Objectives

In this chapter, we will:

- Learn about the <u>H</u>adoop <u>D</u>istributed <u>F</u>iles <u>S</u>ystem (HDFS)
- Run a standalone instance of HDFS
- Create directories and files in HDFS
- 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

About HDFS—I

- The Hadoop Distributed File System (HDFS) is the main storage used by Hadoop MapReduce applications
  - Distributed, POSIX-like file system
    - Designed to run on commodity hardware
    - Scales to clusters composed of thousands of nodes
  - Highly fault tolerant
    - Automatically detects hardware faults
    - Supports quick recovery
  - Implemented in Java
- Can be used as a standalone general purpose file system
  - Designed for storing and reading very large files (>TB)
    - Supports high throughput read and writes
    - Write once, read many
    - Aimed at batch processing
    - Default block size is 128MB
  - Does not support random insertion or modification of data
  - Appending/truncating data is possible

About HDFS—II

- HDFS is used either directly or indirectly by many Big Data and NoSQL applications including:
  - Hadoop
  - Spark
  - HBase
  - Pig
  - Hive
  - Others

Core HDFS Services

- HDFS is implemented as several services which are usually deployed on a cluster of machines
  - Referred to as an HDFS cluster
  - Arranged in a controller/worker architecture
- Core HDFS services include:
  - **NameNode** (controller) stores file system metadata
  - **DataNode** (worker) stores file data (data blocks)
- The NameNode is the master server located on one node
  - Implements a POSIX-like hierarchical file system with `'/'` as the root directory
  - Enforces read/write permissions on files and directories
  - Tracks the location of the data blocks for each file
- The DataNode is the worker server located on each node
  - Handles read and write requests from HDFS clients
  - Performs block creation, deletion, and replication as instructed by the NameNode

Start Jupyter

Do Now

- To start Jupyter open a terminal window and type the following commands:

```
start-notebook
```

  – Once it is finished, copy the URL and paste it into a browser

Start Hadoop

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

```
start-hadoop
jps
```

- From a command line or in the Jupyter notebook, try the following commands:

```
hdfs
hdfs dfs
hdfs dfs -ls /
hdfs dfs -put /class/datasets/northwind/CSV/categories /
hadoop fs -ls /
```

- Take a look at `~/.bashrc`
  - Some handy aliases were added to make this easier

```
hls /
hput /class/datasets/northwind/CSV/regions /
```

Introduction to Apache Spark

- Apache Spark is a computing engine that can be used for large-scale data processing
  - Spark 2 can perform between 10X and 100X 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/

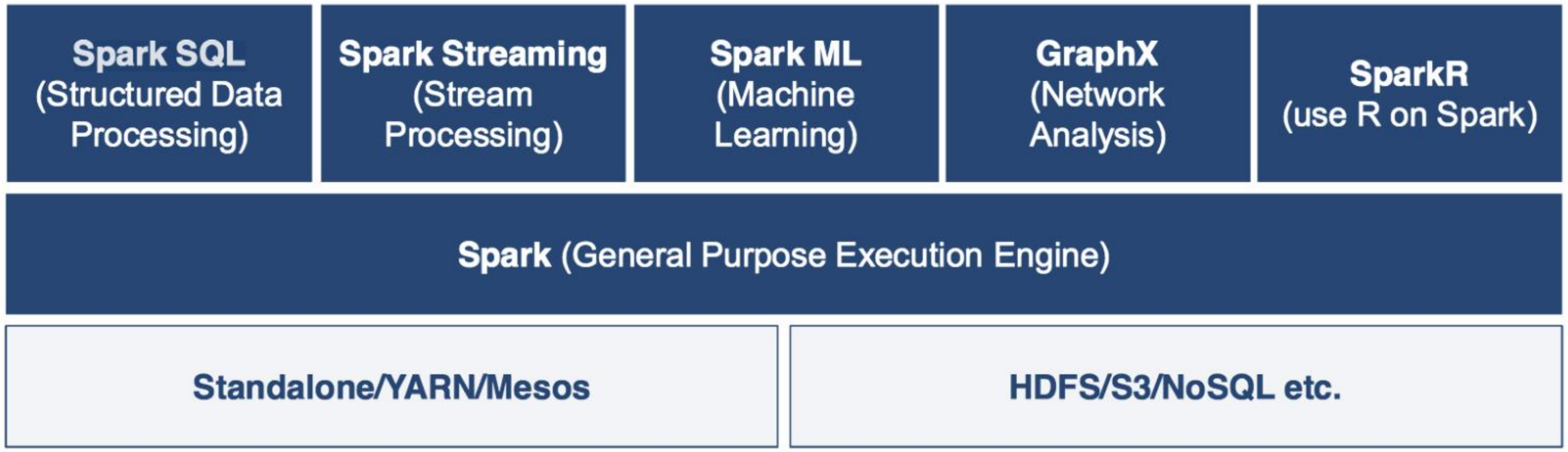
Introduction to Apache Spark (continued)

- 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 SQL (Structured Data Processing)
Spark Streaming (Stream Processing)
Spark ML (Machine Learning)
GraphX (Network Analysis)
SparkR (use R on Spark)
Spark (General Purpose Execution Engine)
Standalone/YARN/Mesos
HDFS/S3/NoSQL etc.

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, R, 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

Resilient Distributed Datasets (RDDs)

- RDDs do not hold an entire dataset in memory for the duration of processing
- They are much more like a pipeline that starts loading files into memory and immediately processing the transformation sequence
- As data is transformed and no longer needed it is released from memory making it available for more data
- It is possible to cache RDD's in memory if reuse of existing transformations is needed, but the tradeoff is that it uses more memory to save CPU and I/O costs for reprocessing
- Whether caching is a good idea or not depends on many factors

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`** but must create it manually when writing a custom program outside the shell
  2. Import data as RDDs
  3. Transform and perform actions on RDDs
  4. Export results

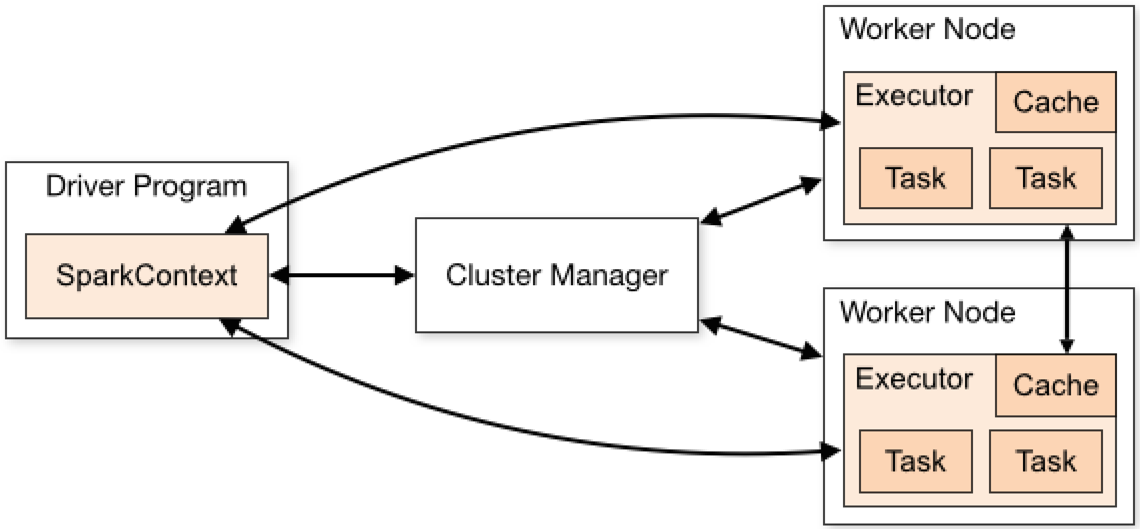
- Spark applications are not declarative in nature, however, they are still coded at a high level of abstraction
  – In contrast, *tasks* are at a very low level of abstraction
  – Tasks are not created by programmers, rather they are created at runtime by Spark

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 or the notebook and run the following commands:

```
cd /class
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` variables manually
  - `initspark.py` is a helper module written so that 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()
```

- Load a local file

```
sc.textFile('file:///class/datasets/northwind/CSV/categor
ies/categories.csv')
```

- Load a local folder

```
sc.textFile('file:///class/datasets/northwind/CSV/categor
ies')
```

- Load a HDFS folder

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

Actions and Transformations

- Once you have an RDD, you can invoke methods on it
- Methods can either be:
  - A transformation which is lazy evaluated and is only run when an action is called
  - An action which causes it to do some work and possibly return data back to the client or write it to disk
- 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
  - Once an action fires, a subsequent action redoes all the work from the beginning, no data is cached unless you tell it to
- 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.savePickleFile()` – saves the RDD as a Python pickle file
- Typically, most of these are no longer used in favor of DataFrame write options which are more rich

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:

- Learned about the Hadoop Distributed Files System (HDFS)
- Ran a standalone instance of HDFS
- Created directories and files in HDFS
- 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