Session10, 11 and 12

Framework for Python & Hadoop Streaming

Unit II Syllabus

- Data Product, Building Data Products at Scale with Hadoop, Data Science Pipeline and Hadoop Ecosystem

  Ref: (Chapter 1 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

- Operating System for Big Data: Concepts, Hadoop Architecture, Working with Distributed file system, Working with Distributed Computation

  Ref: (Chapter 2 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

- Framework for Python and Hadoop Streaming, Hadoop Streaming, MapReduce with Python, Advanced MapReduce.

  Ref: (Chapter 3 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

- In-Memory Computing with Spark, Spark Basics, Interactive Spark with PySpark, Writing Spark Applications

  Ref: (Chapter 4 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

Today's Topics

Chapter 3 Framework for Python and Hadoop Streaming

- Hadoop Streaming
- MapReduce with Python
- Advanced MapReduce.

Ref: (Chapter 2, 3 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

Overview

- The current version of Hadoop MapReduce is a software framework for composing jobs that process large amounts of data in parallel on a cluster, and is the native distributed processing framework that ships with Hadoop.
- The framework exposes a Java API that allows developers to specify input and output locations on HDFS, map and reduce functions, and other job parameters as a job configuration.
- Jobs are compiled and packaged into a JAR, which is submitted to the ResourceManager by the job client- usually via the command line.
- The ResourceManager then schedules tasks, monitors them, and provides status back to the client

Overview…

- Typically, a MapReduce application is composed of three Java classes: a Job, a Mapper, and a Reducer.
- Mappers and reducers handle the details of computation on key/value pairs and are connected through a shuffle and sort phase.
- The Job configures the input and output data format by specifying the InputFormat and OutputFormat classes of data being serialized to and from HDFS.
- All of these classes must extend abstract base classes or implement MapReduce interfaces.

Overview…

- However, Java is not the only option to use the MapReduce framework! For example, C++ developers can use Hadoop Pipes, which provides an API for using both HDFS and MapReduce.
- But what is of most interest to data scientists is Hadoop Streaming, a utility written in Java that allows the specification of any executable as the mapper and reducer.
- With Hadoop Streaming shell utilities, R, or Python, scripts can all be used to compose MapReduce jobs.
- This allows data scientists to easily integrate Map- Reduce into their workflows—particularly for routine data management tasks that don’t require extensive software development

Overview…

In this chapter, we explore the details of

how to use Hadoop Streaming,

as well as work through the creation of a small framework that will allow us to quickly write MapReduce jobs using Python

Hadoop Streaming

Hadoop Streaming

- Hadoop Streaming is a utility, packaged as a JAR file that comes with the Hadoop MapReduce distribution.

- Streaming is used as a normal Hadoop job passed to the cluster via the job client, but allows you to also specify arguments such as the input and output HDFS paths, along with the mapper and reducer executable.

- The job is then run as a normal MapReduce job, managed and monitored by the ResourceManager and the MRAppMaster as usual until the job completes

Hadoop Streaming…

- In order to perform a MapReduce job, Streaming utilizes the standard Unix streams for input and output, hence the name Streaming.
- Input to both mappers and reducers is read from stdin, which a Python process can access via the sys module.
- Hadoop expects the Python mappers and reducers to write their output key/value pairs to stdout.

Hadoop Streaming…

- Figure 3-1 demonstrates this process in a MapReduce context.
- Although Python Hadoop developers don't necessarily get access to the full MapReduce AP through this technique

  (features like partitioners or input and output formats must be written in Java),

  this is enough to express many powerful jobs and tasks that are typical in the workflow of a data scientist.

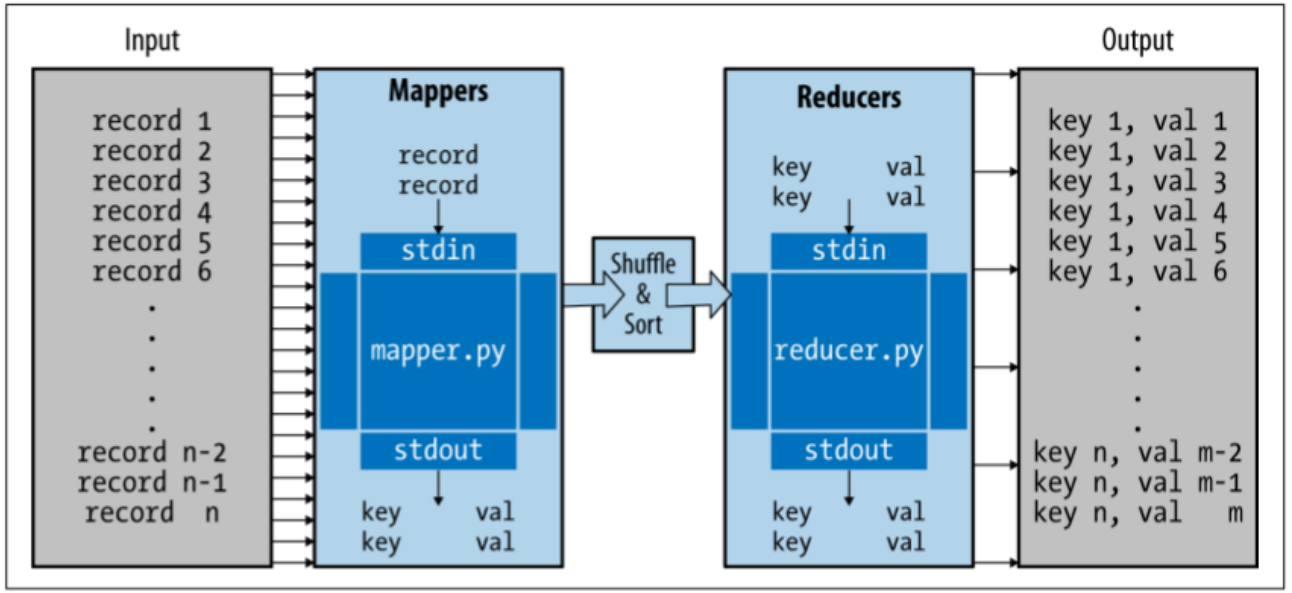


Figure 3-1. Data flow in Hadoop Streaming via Python mapper.py and reducer.py scripts

Hadoop Streaming…

- Hadoop Streaming is not to be confused with Spark Streaming or other real-time computation frameworks that utilize “unbounded streams of data,” such as Apache Storm.
- Streams in Hadoop Streaming refer to the standard Unix streams: stdin, stdout, and stderr.
- Spark Streaming and Storm do real-time analytics on an incoming stream of data by batching the data into time windows.
- They are very different things! The use of “Streaming” specifically refers to Hadoop Streaming in this chapter.

Hadoop Streaming…

- When Streaming executes a job, each mapper task will launch the supplied executable inside of its own process.
- The mapper then converts the input data into lines of text and pipes it to the stdin of the external process while simultaneously collecting out- put from stdout.
- The input conversion is usually a straightforward serialization of the value because data is being read from HDFS, where each line is a new value.
- The mapper expects output to be in a string key/value format, where the key is separated from the value by some separator character, tab (\t) by default.
- If there is no separator, then the mapper considers the output to only be a key with a null value. The separator can be customized by passing arguments to the Hadoop Streaming job.

Hadoop Streaming…

- The reducer is also launched as its own executable after the output from the mappers is shuffled and sorted to ensure that each key is sent to the same reducer.

- The key/ value output strings from the mapper are streamed to the reducer as input via stdin, matching the data output from the mapper, and guaranteed to be grouped by key.

- The output the reducer emits to stdout is expected to have the same key, separator, and value format as the mapper.

Hadoop Streaming…

- Therefore, in order to write Hadoop jobs using Python, we need to create two Python files, mapper.py and a reducer.py.
- Inside each of those files we simply need to import the sys module to get access to stdin and stdout.
- The code itself will need to deal with our input as a string, parsing and converting for each number or complex data type, and we need to serialize our output as a string as well.
- To demonstrate how this works, we will implement the WordCount example discussed in Chapter 2 in the most simple and Pythonic method as possible

mapper.py

First, we create our executable mapper in a file called *mapper.py*:

```
#!/usr/bin/env python

import sys

if __name__ == "__main__":
    for line in sys.stdin:
        for word in line.split():
            sys.stdout.write("{}\t1\n".format(word))
```

- The mapper simply reads each line from sys.stdin,
- splits on space,
- then writes each word and a 1 separated by a tab,
- line-by-line to sys.stdout

reducer.py

```
#!/usr/bin/env python

import sys

if __name__ == '__main__':
    curkey = None
    total  = 0
    for line in sys.stdin:
        key, val = line.split("\t")
        val = int(val)

        if key == curkey:
            total += val
        else:
            if curkey is not None:
                sys.stdout.write("{}\t{}\n".format(curkey, total))

            curkey = key
            total  = val
```

- The reducer is a bit more complex because we have to track which key we're on at every line of input, and only emit a completed sum when we see a new key.
- This is because, unlike the native API, individual data values are aggregated to the streaming process during shuffle and sort rather than exposed as a list or iterator.
- Keep in mind that each reducer task is guar- anteed to see all values for the same key, but may also see multiple keys.
- As the reducer iterates over each line in the input from stdin, it splits the line on the separator character and converts the value to an integer.
- It then performs a check to ensure that we're still computing the count for the same key; otherwise, it writes the output to stdout and restarts the count for the new key.

- Each Python module will be executed inside its own process—so it will have as many computing resources in terms of processing and memory as are available at the time of execution.

- It’s important to note, however, that because each mapper and reducer is treated as executable by Hadoop Streaming, every Python file should start with #!/usr/bin/env python, which alerts the shell that the code should be interpreted using Python rather than bash.

MapReduce and csv file

- Will do it in PySpark

A Framework for MapReduce with Python

- Slightly more advanced usage of Hadoop Streaming takes advantage of standard error (stderr) to update the Hadoop status as well as Hadoop counters.
- This technique essentially allows Streaming jobs to access the Reporter object, a part of the Map- Reduce Java API that tracks the global status of a job.
- By writing specially formatted strings to stderr, both mappers and reducers can update the global job status to report their progress and indicate they’re alive.
- For jobs that take a significant amount of time (especially tasks involving the loading of a large model from a pickle file that is passed with the job), this is critical to ensuring that the framework doesn’t assume a task has timed out

In Memory Computing with Spark

Unit II Syllabus

- Data Product, Building Data Products at Scale with Hadoop, Data Science Pipeline and Hadoop Ecosystem

  Ref: (Chapter 1 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

- Operating System for Big Data: Concepts, Hadoop Architecture, Working with Distributed file system, Working with Distributed Computation

  Ref: (Chapter 2 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

- Framework for Python and Hadoop Streaming, Hadoop Streaming, MapReduce with Python, Advanced MapReduce.

  Ref: (Chapter 3 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

- In-Memory Computing with Spark, Spark Basics, Interactive Spark with PySpark, Writing Spark Applications

  Ref: (Chapter 4 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

Today's Topics

Chapter 4 In-Memory Computing with Spark

- Spark Basics
  - The Spark Stack
  - Resilient Distributed Datasets
  - Programming with RDDs
- Interactive Spark with PySpark
- Writing Spark Applications

Ref: (Chapter 2, 3 - Data Analytics with Hadoop By Benjamin Bengfort & Jenny Kim)

Overview
(HDFS, MapReduce – Apache Spark)

- HDFS and MapReduce have been the foundation of and the driver for the advent of large-scale machine learning, scaling analytics, and big data appliances for the last decade.

- The maturation of Hadoop has led to a stable computing environment that is general enough to build specialist tools for tasks such as graph processing, micro-batch processing, SQL querying, data warehousing, and machine learning.
- As a result Hadoop became more widely adopted

- More specializations were required for a wider variety of new use cases, and it became clear that the batch processing model of MapReduce was not well suited to common workflows including iterative, interactive, or on-demand computations upon a single dataset.

Overview…
(HDFS, MapReduce – Apache Spark)

- The primary MapReduce abstraction (specification of computation as a mapping then a reduction) is parallelizable, easy to understand, and hides the details of distributed computing, thus allowing Hadoop to guarantee correctness.

- However, in order to achieve coordination and fault tolerance, the MapReduce model uses a pull execution model that requires intermediate writes of data back to HDFS.

- Unfortunately, the input/output (I/O) of moving data from where it's stored to where it needs to be computed upon is the largest time cost in any computing system.
- as a result, while MapReduce is incredibly safe and resilient, it is also necessarily slow on a per-task basis.

Overview…
(HDFS, MapReduce – Apache Spark)

- Worse, almost all applications must chain multiple MapReduce jobs together in multiple steps, creating a data flow toward the final required result.

- This results in huge amounts of intermediate data written to HDFS that is not required by the user, creating additional costs in terms of disk usage.

- To address these problems, Hadoop has moved to a more general resource management framework for computation: YARN.

Overview…
(HDFS, MapReduce – Apache Spark)

- Whereas previously the MapReduce application allocated resources (processors, memory) to jobs specifically for mappers and reducers, YARN provides more general resource access to Hadoop applications.

- The result is that specialized tools no longer have to be decomposed into a series of MapReduce jobs and can become more complex.

- By generalizing the management of the cluster, the programming model first imagined in MapReduce can be expanded to include new abstractions and operations.

Overview…
(HDFS, MapReduce – Apache Spark)

- Spark is the first fast, general-purpose distributed computing paradigm resulting from this shift, and is rapidly gaining popularity particularly because of its speed and adaptability.

- Spark primarily achieves this speed via a new data model called resilient distributed datasets (RDDs) that are stored in memory while being computed upon, thus eliminating expensive intermediate disk writes.

- It also takes advantage of a directed acyclic graph (DAG) execution engine that can optimize computation, particularly iterative computation, which is essential for data theoretic tasks such as optimization and machine learning.

- These speed gains allow Spark to be accessed in an interactive fashion (as though you were sitting at the Python interpreter), making the user an integral part of computation and allowing for data exploration of big datasets that was not previously possible, bringing the cluster to the data scientist.

Overview…
(HDFS, MapReduce – Apache Spark)

- In this chapter, we introduce Spark and resilient distributed datasets. This is the last chapter describing the nuts and bolts of doing analytics with Hadoop.

- Because Spark implements many applications already familiar to data scientists (e.g., DataFrames, interactive notebooks, and SQL), we propose that at least initially, Spark will be the primary method of cluster interaction for the novice Hadoop user.

Apache Spark Basics

- Apache Spark is a cluster-computing platform that provides an API for distributed programming similar to the MapReduce model, but is designed to be fast for interactive queries and iterative algorithms.

- It primarily achieves this by caching data required for computation in the memory of the nodes in the cluster.

- In-memory cluster computation enables Spark to run iterative algorithms, as programs can checkpoint data and refer back to it without reloading it from disk; in addition, it supports interactive querying and streaming data analysis at extremely fast speeds.

- Because Spark is compatible with YARN, it can run on an existing Hadoop cluster and access any Hadoop data source, including HDFS, S3, HBase, and Cassandra.

Apache Spark Basics...

- Importantly, Spark was designed from the ground up to support big data applications and data science in particular.
- Instead of a programming model that only supports map and reduce, the Spark API has many other powerful distributed abstractions similarly related to functional programming, including sample, filter, join, and collect, to name a few.
- Moreover, while Spark is implemented in Scala, programming APIs in Scala, Java, R, and Python makes Spark much more accessible to a range of data scientists who can take fast and full advantage of the Spark engine.

Apache Spark Basics...

Limitations of MapReduce

- The iterative algorithms apply the same operation many times to blocks of data until they reach a desired result.

- For example,
  - optimization algorithms like gradient descent are iterative; given some target function (like a linear model), the goal is to optimize the parameters of that function such that the error (the difference between the predicted value of the model and the actual value of the data) is minimized.
  - Here, the algorithm applies the target function with one set of parameters to the entire dataset and computes the error, afterward slightly modifying the parameters of the function according to the computed error (descending down the error curve).
  - This process is repeated (the iterative part) until the error is minimized below some threshold or until a maximum number of iterations is reached.
  - This basic technique is the foundation of many machine learning algorithms, particularly supervised learning, in which the correct answers are known ahead of time and can be used to optimize some decision space.

Apache Spark Basics...

- In order to program this type of algorithm in MapReduce, the parameters of the target function would have to be mapped to every instance in the dataset, and the error computed and reduced.
- After the reduce phase, the parameters would be updated and fed into the next MapReduce job.
- This is possible by chaining the error computation and update jobs together; however, on each job the data would have to be read from disk and the errors written back to it, causing significant I/O-related delay.

Apache Spark Basics…

- Instead, Spark keeps the dataset in memory as much as possible throughout the course of the application, preventing the reloading of data between iterations.

- Spark programmers therefore do not simply specify map and reduce steps, but rather an entire series of data flow transformations to be applied to the input data before performing some action that requires coordination like a reduction or a write to disk.

- Because data flows can be described using directed acyclic graphs (DAGs), Spark's execution engine knows ahead of time how to distribute the computation across the cluster and manages the details of the computation, similar to how MapReduce abstracts distributed computation.

Apache Spark Basics…

- By combining acyclic data flow and in-memory computing, Spark is extremely fast particularly when the cluster is large enough to hold all of the data in memory.

- In fact, by increasing the size of the cluster and therefore the amount of available memory to hold an entire, very large dataset, the speed of Spark means that it can be used *interactively*—making the user a key participant of analytical processes that are running on the cluster.

- As Spark evolved, the notion of user interaction became essential to its model of distributed computation; in fact, it is probably for this reason that so many languages are supported.

The Spark Stack…

- Spark is a general-purpose distributed computing abstraction and can run in a stand-alone mode.
- However, Spark focusse purely on *computation* rather than *data storage* and as such is typically run in a cluster that implements data warehousing and cluster management tools.
- In this book, we are primarily interested in Hadoop.
- When Spark is built with Hadoop, it utilizes YARN to allocate and manage cluster resources like processors and memory via the ResourceManager.
- Importantly, Spark can then access any Hadoop data source—for example HDFS, HBase, or Hive, to name a few

The Spark Stack…

- Spark exposes its primary programming abstraction to developers through the Spark Core module.
- This module contains basic and general functionality, including the API that defines resilient distributed datasets (RDDs).
- RDDs, which we will describe in more detail in the next section, are the essential functionality upon which all Spark computation resides.
- Spark then builds upon this core, implementing special-purpose libraries for a variety of data science tasks that interact with Hadoop, as shown in Figure 4-1

The Spark Stack…

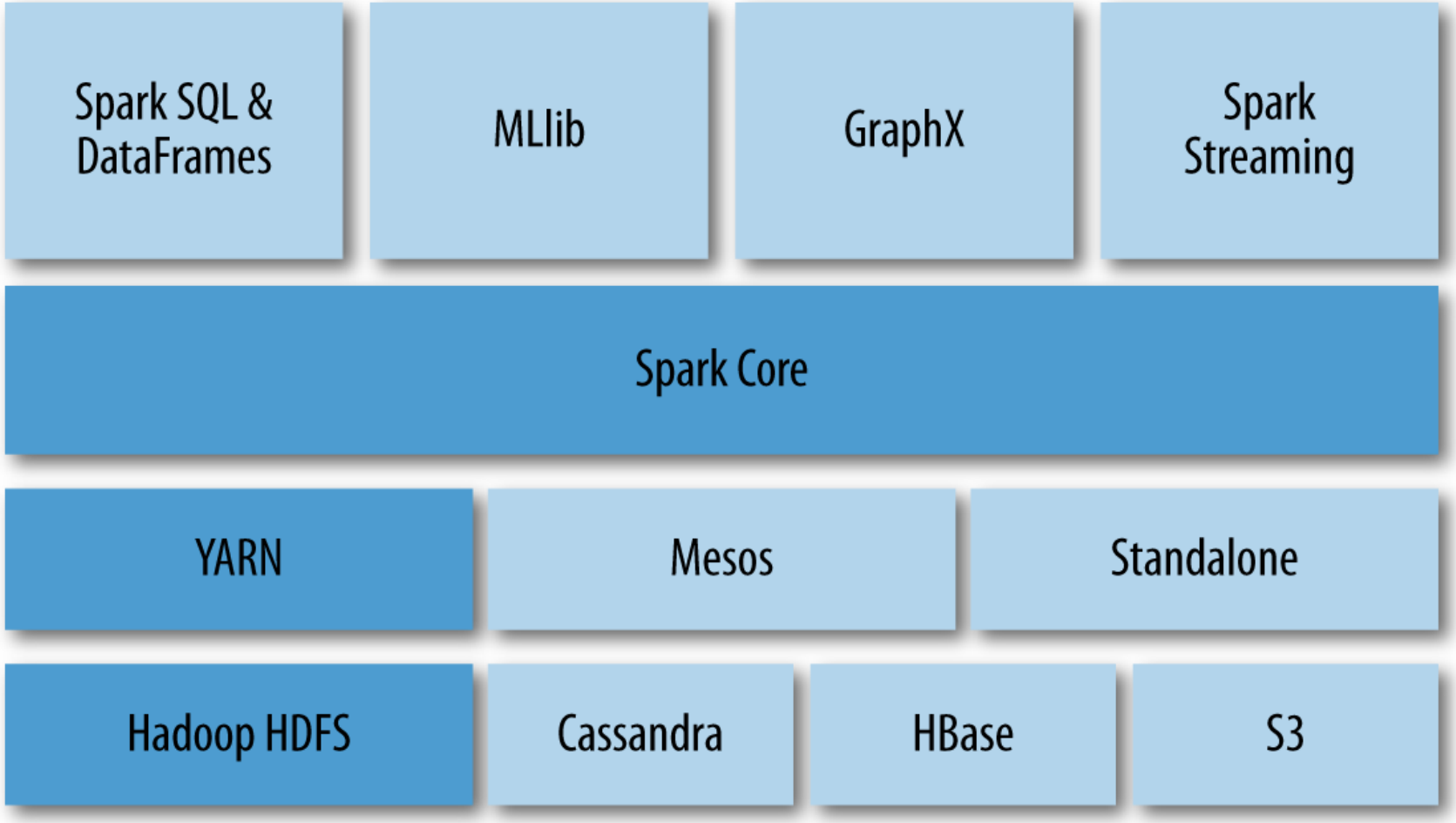
- The component libraries are not integrated into the general-purpose computing framework, making the Spark Core module extremely flexible and allowing developers to easily solve similar use cases with different approaches.

- For example,
  - Hive will be moving to Spark, allowing an easy migration path for existing users;
  - GraphX is based on the Pregel model of vertex-centric graph computation, but other graph libraries that leverage gather, apply, scatter (GAS) style computations could easily be implemented with RDDs.

- This flexibility means that specialist tools can still use Spark for development, but that new users can quickly get started with the Spark components that already exist.

The Spark Stack…

Figure 4-1. Spark is a computational framework designed to take advantage of cluster management platforms like YARN and distributed data storage like HDFS



The Spark Stack…

- Primary Components of Spark
  - Spark SQL
    - Originally provided APIs for interacting with Spark via the Apache Hive variant of SQL called HiveQL; in fact, you can still directly access Hive via this library.
    - However, this library is moving toward providing a more general, structured data-processing abstraction, DataFrames.
    - DataFrames are essentially distributed collections of data organized into columns, conceptually similar to tables in relational databases.

The Spark Stack…

- Spark Streaming
  - Enables the processing and manipulation of unbounded streams of data in real time. Many streaming data libraries (such as Apache Storm) exist for handling real-time data.
  - Spark Streaming enables programs to leverage this data similar to how you would interact with a normal RDD as data is flowing in.

The Spark Stack…

- MLLib
  - A library of common machine learning algorithms implemented as Spark operations on RDDs.
  - This library contains scalable learning algorithms (e.g., classifications, regressions, etc.). that require iterative operations across large datasets.
  - The Mahout library, formerly the big data machine learning library of choice, will move to Spark for its implementations in the future.

- GraphX
  - A collection of algorithms and tools for manipulating graphs and performing parallel graph operations and computations.
  - GraphX extends the RDD API to include operations for manipulating graphs, creating subgraphs, or accessing all vertices in a path.

The Spark Stack…

- These components combined with the Spark programming model provide a rich methodology of interacting with cluster resources.
- It is probably because of this completeness that Spark has become so immensely popular for distributed analytics.
- Instead of learning multiple tools, the basic API remains the same across components and the components themselves are easily accessed without extra installation.

Resilient Distributed Datasets

- In Chapter 2, we described Hadoop as a distributed computing framework that dealt with two primary problems:
  - how to distribute data across a cluster
    - Distributed computation intends to improve the performance (speed) of a computation by breaking a large computation or task into smaller, independent computations that can be run simultaneously (in parallel) and then aggregated to a final result.
  - how to distribute computation.
    - The distributed data storage problem deals with high availability of data (getting data to the place it needs to be processed) as well as recoverability and durability.
- Because each parallel computation is run on an individual node or computer in the cluster, a distributed computing framework needs to provide consistency, correctness, and fault-tolerant guarantees for the whole computation.
- Spark does not deal with distributed data storage, relying on Hadoop to provide this functionality, and instead focuses on reliable distributed computation through a framework called resilient distributed datasets

Resilient Distributed Datasets…

- RDDs are essentially a programming abstraction that represents a read-only collection of objects that are partitioned across a set of machines.

- RDDs can be rebuilt from a lineage (and are therefore fault tolerant), are accessed via parallel operations, can be read from and written to distributed storages (e.g., HDFS or S3), and most importantly, can be cached in the memory of worker nodes for immediate reuse.

- As mentioned earlier, it is this in-memory caching feature that allows for massive speedups and provides for *iterative computing* required for machine learning and user-centric *interactive analyses*.

Resilient Distributed Datasets…

- RDDs are operated upon with functional programming constructs that include and expand upon map and reduce.
- Programmers create new RDDs by loading data from an input source, or by transforming an existing collection to generate a new one.
- The history of applied transformations is primarily what defines the RDD's *lineage*, and because the collection is *immutable* (not directly modifiable), transformations can be reapplied to part or all of the collection in order to recover from failure.
- The Spark API is therefore essentially a collection of operations that create, transform, and export RDDs

Resilient Distributed Datasets…

- An additional benefit of Spark is that it applies transformations "lazily"—inspecting a complete sequence of transformations and an action before executing them by submitting a job to the cluster.

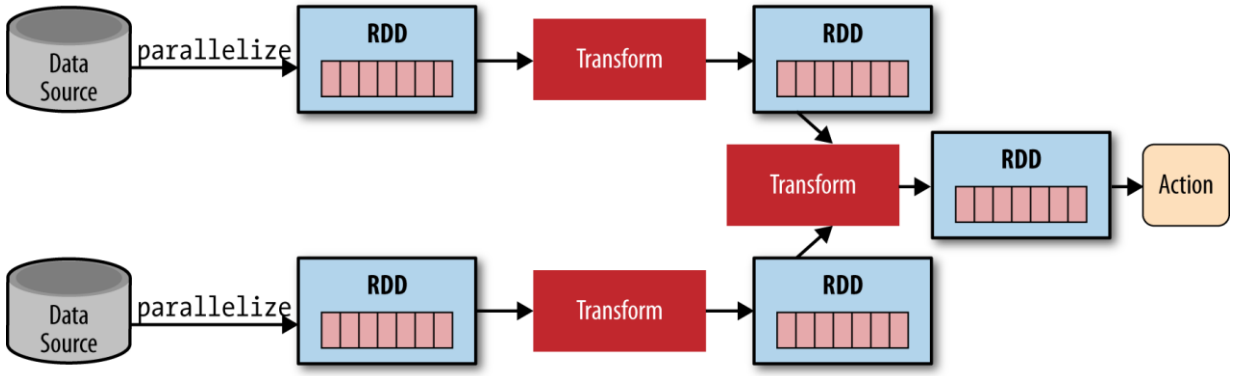
- This lazy-execution provides significant storage and computation optimizations, as it allows Spark to build up a lineage of the data and evaluate the complete transformation chain in order to compute upon only the data needed for a result;

- for example, if you run the first() action on an RDD, Spark will avoid reading the entire dataset and return just the first matching line.

Programming with RDD

- Programming Spark applications is similar to other data flow frameworks previously implemented on Hadoop.
- Code is written in a driver program that is evaluated lazily on the driver-local machine when submitted, and upon an action, the driver code is distributed across the cluster to be executed by workers on their partitions of the RDD.
- Results are then sent back to the driver for aggregation or compilation.

Programming with RDD..



- As illustrated in Figure 4-2, the driver program creates one or more RDDs by parallelizing a dataset from a Hadoop data source, applies operations to transform the RDD, then invokes some action on the transformed RDD to retrieve output.
- RDDs are partitioned collections of data that allow the programmer to apply operations to the entire collection in *parallel*.
- It is the partitions that allow the parallelization, and the partitions themselves are computed boundaries in the list where data is stored on different nodes.
- Therefore “parallelization” is the act of partitioning a dataset and sending each part of the data to the node that will perform computations upon it.

Programming with RDD…

A typical data flow sequence for programming Spark is as follows:

1. Define one or more RDDs, either through accessing data stored on disk
(e.g., HDFS, Cassandra, HBase, or S3),
parallelizing some collection,
transforming an existing RDD, or by caching.
Caching is one of the fundamental procedures in Spark—storing an RDD in the memory of a node for rapid access as the computation progresses.

2. Invoke operations on the RDD by passing closures (here, a function that does not rely on external variables or data) to each element of the RDD. Spark offers many high-level operators beyond map and reduce.

3. Use the resulting RDDs with aggregating actions (e.g., **count, collect, save**, etc.). Actions kick off the computation on the cluster because no progress can be made until the aggregation has been computed.

Programming with RDD…

Variables and Closures

- Variables and Closures can be confusing in Spark.
- When Spark runs a closure on a worker, any variables used in the closure are copied to that node, but are maintained within the local scope of that closure.
- If external data is required, Spark provides two types of shared variables that can be interacted with by all workers in a restricted fashion: *broadcast variables* and *accumulators*.

- Broadcast variables are distributed to all workers, but are read-only and are often used as lookup tables or stop word lists.

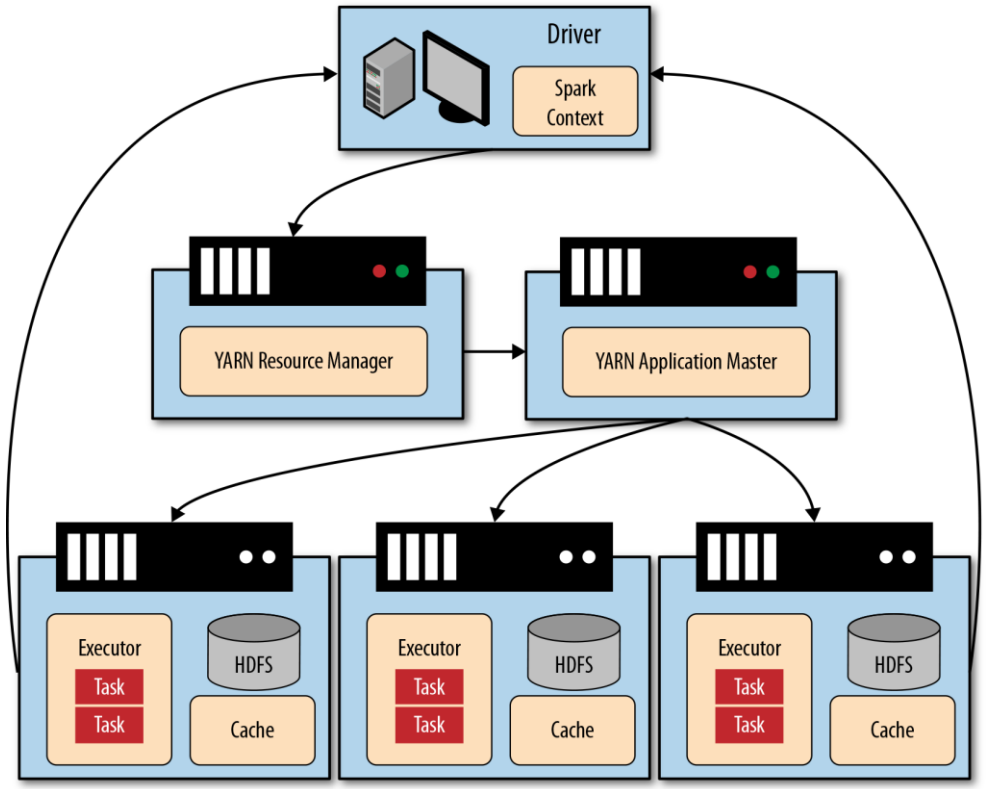
- Accumulators are variables that workers can "add" to using associative operations and are typically used as counters.

- These data structures are similar to the MapReduce distributed cache and counters, and serve a similar role.
- However, because Spark allows for general interprocess communication, these data structures are perhaps used in a wider variety of applications.

Programming with RDD…Spark Execution

- With this in mind,
- Spark applications can actually be submitted to the Hadoop cluster in two modes:
  - yarn-client
    - In yarn-client mode, the driver is run inside of the client process as described, and the ApplicationMaster simply manages the progression of the job and requests resources.
  - yarn-cluster
    - However, in yarn-cluster mode, the driver program is run inside of the ApplicationMaster process, thus releasing the client process and proceeding more like traditional MapReduce jobs.
- Programmers would use yarn-client mode to get immediate results or in an interactive mode and yarn-cluster for long-running jobs or ones that do not require user intervention.

Programming with RDD...Spark Execution



A brief note on the execution of Spark:

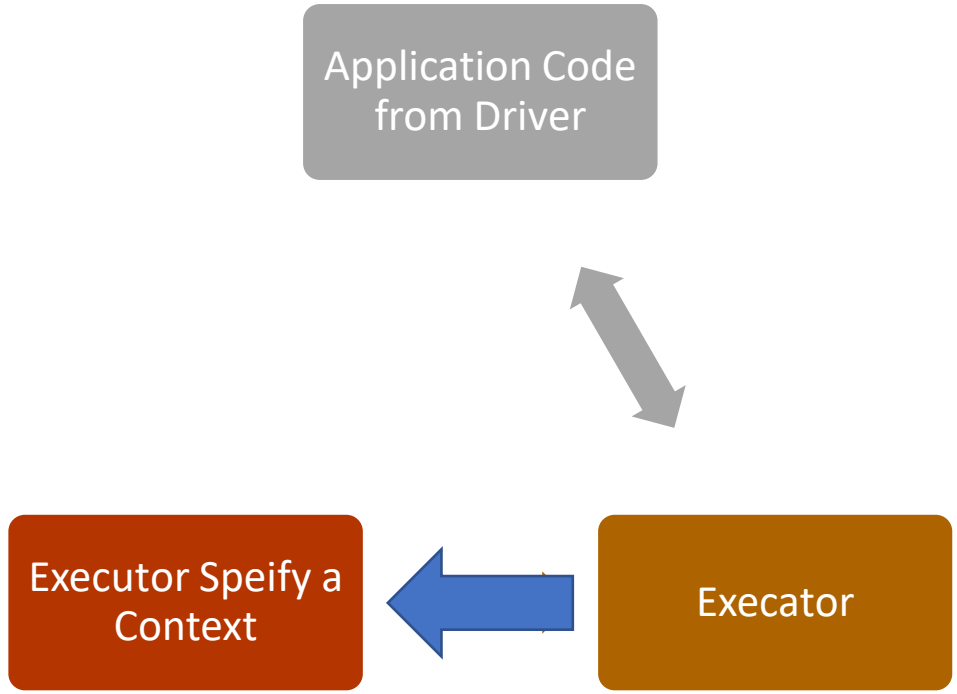
- essentially, Spark applications are run as independent sets of processes, coordinated by a SparkContext in a driver program.
- The context will connect to some cluster manager (e.g., YARN), which allocates system resources.
- Each worker in the cluster is managed by an executor, which is in turn managed by the SparkContext.
- The executor manages computation as well as storage and caching on each machine.

The interaction of the driver, YARN, and the workers is shown in Figure 4-3.

Programming with RDD...Spark Execution

- It is important to note that application code is sent from the driver to the executors, and the executors specify the context and the various tasks to be run.
- The executors communicate back and forth with the driver for data sharing or for interaction.
- Drivers are key participants in Spark jobs, and therefore, they should be on the same network as the cluster.

- This is different from Hadoop code, where you might submit a job from anywhere to the ResourceManager, which then handles the execution on the cluster.



Interactive Spark using PySpark

For datasets that fit into the memory of a cluster,

- Spark is fast enough to allow data scientists to interact and explore big data from an interactive shell that implements a Python REPL (read-evaluate-print loop) called pyspark.

- This interaction is similar to how you might interact with native Python code in the Python interpreter, writing commands on the command line and receiving output to stdout (there are also Scala and R interactive shells).

- This type of interactivity also allows the use of interactive notebooks, and setting up an iPython or Jupyter notebook with a Spark environment is very easy.

Thank You….

Revise the topics from Syllabus References…

Fill Your Attendance Form….!

Syllabus References

1. Big Data and Analytics, Subhashini Chellappan Seema Acharya, Wiley

2. Data Analytics with Hadoop *An Introduction for Data Scientists*, Benjamin Bengfort and Jenny Kim, O'Reilly

   https://github.com/oreillymedia/Data_Analytics_with_Hadoop

1. Big Data and Hadoop, V.K Jain, Khanna Publishing

   https://books.google.co.in/books?id=i6NODQAAQBAJ&pg=PA122&source=gbs_toc_r&cad=4#v=onepage&q&f=true

Advanced MapReduce

Thank You….

Revise the topics from Syllabus References…

Fill Your Attendance Form….!