Python programming and data analysis

Lecture 11

Introduction to Apache Hadoop and Spark

Robert Szmurło

e-mail: robert.szmurlo@ee.pw.edu.pl 2019L

Lecture outline

1. Big Data
2. Apache Hadoop
3. Apache Spark with Python
4. Setup (local machine and Google Colab)
5. Introduction to Apache Spark concepts
6. Few simple examples
7. K-Means Clustering with Apache Spark

Big data

Big Data includes huge volume, high velocity, and extensible variety of data. The data in it will be of three types.

- Structured data – Relational data.
- Semi Structured data – XML data.
- Unstructured data – Word, PDF, Text, Media Logs.

Big Data Challenges - The major challenges related to big data are as follows

- Capturing data (data harvesting)
- Curation (data cleaning)
- Storage (huge files)
- Searching (efficient indexing)
- Sharing (sharing blocks of data for among nodes - also intermediate results)
- Transfer (take into account network topology)
- Analysis (the results must be summarized)
- Presentation (we can't simply plot results, they must be processed)

Hadoop

Hadoop runs applications using the MapReduce programming model, where the data is processed in parallel on multiple nodes. In short, it is used to develop applications that could perform statistical analysis on huge amounts of data.

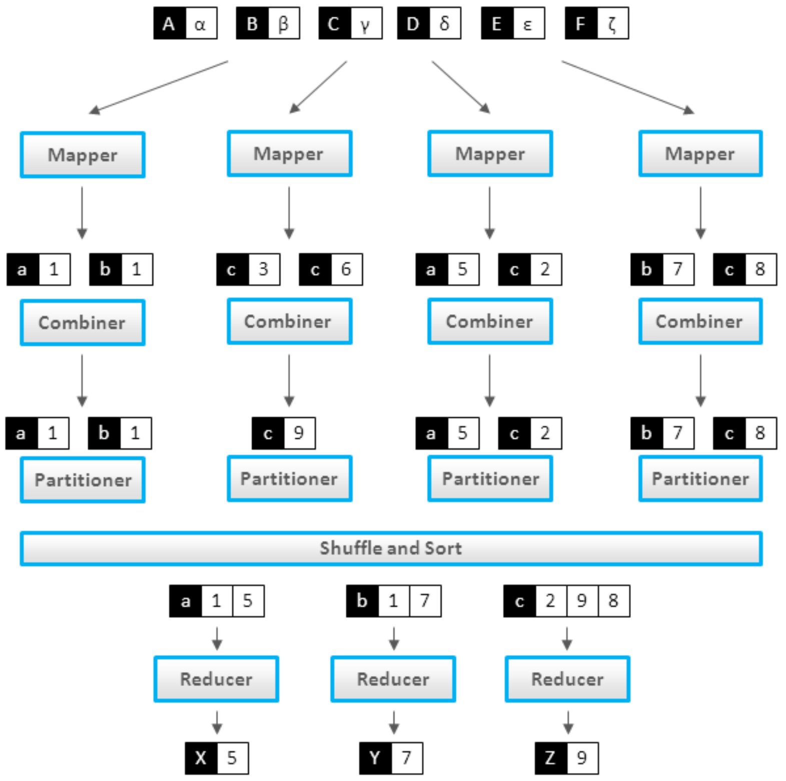
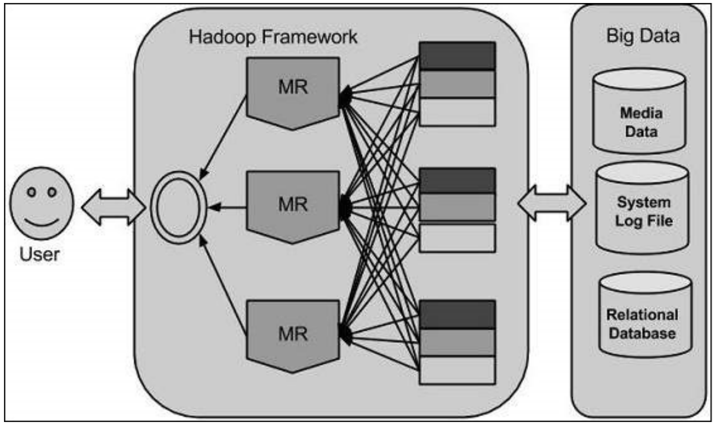


Fig. 1 MapReduce model example

Hadoop concepts

- Written in **Java**.
- Allows distributed processing of large datasets across clusters of computers using simple programming models.
- The Hadoop framework provides **distributed storage** and computation across clusters of computers.
- Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage.



At its core, Hadoop has two major layers namely:

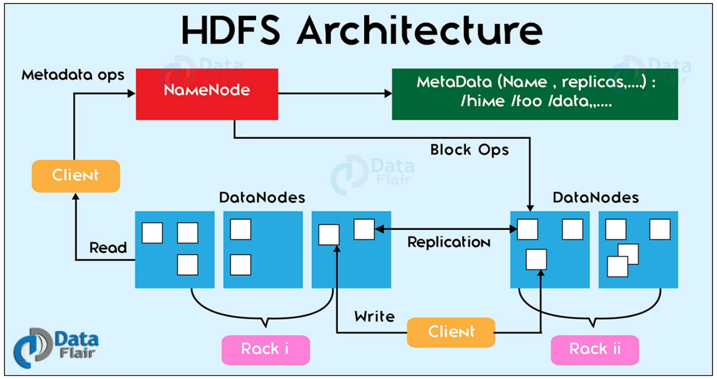
- Processing/Computation layer (**MapReduce**), and
- Storage layer (**HDFS** - Hadoop Distributed File System).

Hadoop framework also includes the following two modules:

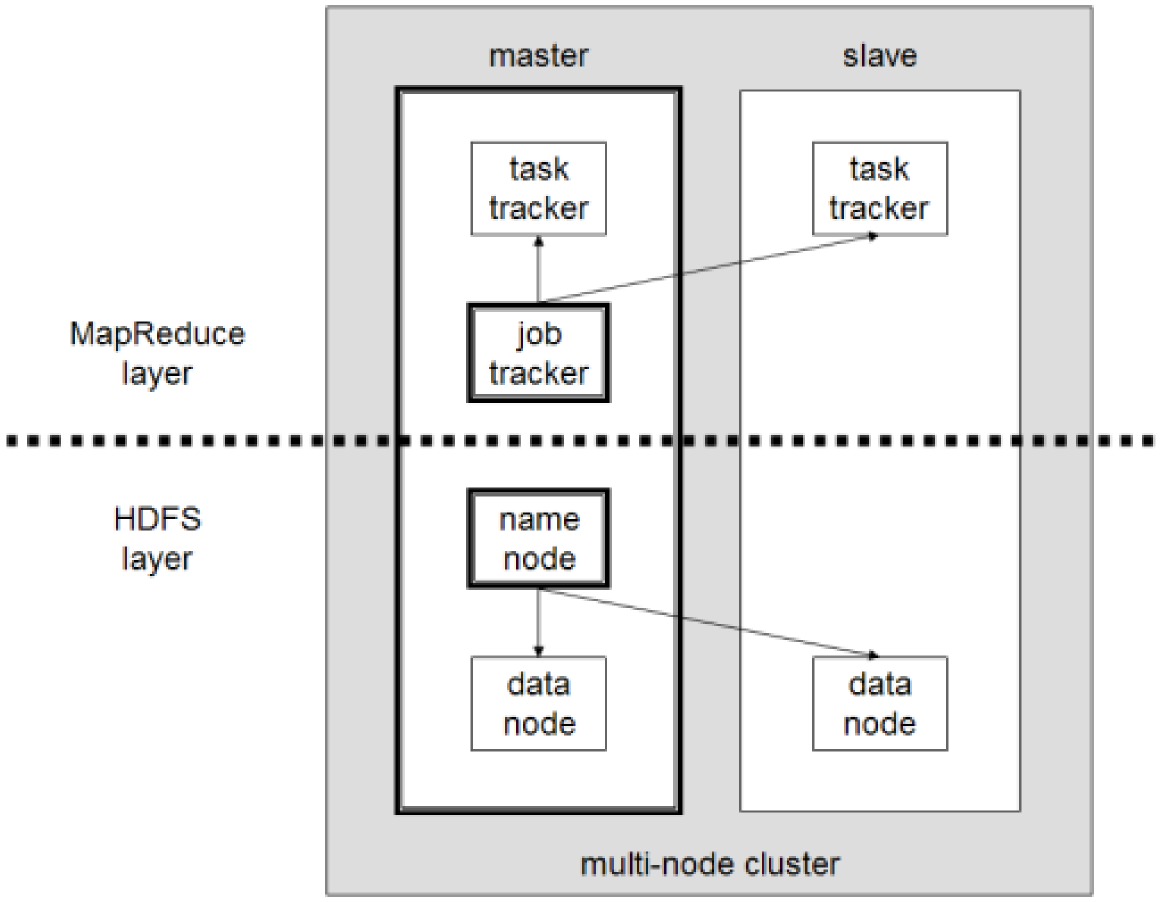
- Hadoop Common – These are Java libraries and utilities required by other Hadoop modules.
- Hadoop YARN – This is a framework for job scheduling and cluster resource management.

Hadoop Distributed File System

The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and provides a distributed file system that is designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. It is highly fault-tolerant and is designed to be deployed on low-cost hardware. It provides high throughput access to application data and is suitable for applications having large datasets.

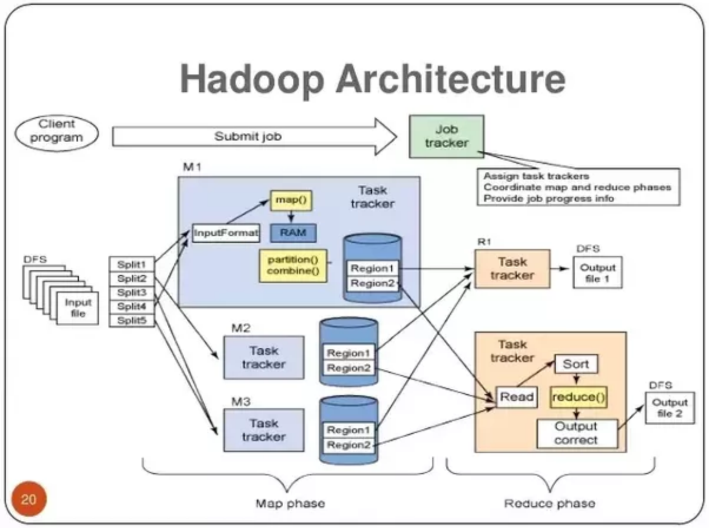


Hadoop architecture



How Does Hadoop Work?

Many commodity computers with single-CPU, are tied together as *a single functional distributed system* and practically, the clustered machines can read the dataset in parallel and provide a much higher throughput. Moreover, it is cheaper than one high-end server. So this is the first motivational factor behind using Hadoop that it runs across clustered and low-cost machines.



Hadoop runs code across a cluster of computers. This process includes the following core tasks that Hadoop performs:

- Data is initially divided into directories and files. Files are divided into uniform sized blocks of 128M and 64M (preferably 128M).
- These files are then distributed across various cluster nodes for further processing.
- HDFS, being on top of the local file system, supervises the processing.
- Blocks are replicated for handling hardware failure.
- Checking that the code was executed successfully.
- Performing the sort that takes place between the map and reduce stages.
- Sending the sorted data to a certain computer.
- Writing the debugging logs for each job.

Advantages of Hadoop

Hadoop framework allows the user to quickly write and test distributed systems. It is efficient, and it automatically distributes the data and work across the machines and in turn, utilizes the underlying parallelism of the CPU cores.

Hadoop **does not rely on hardware** to provide fault-tolerance and high availability (FTHA), rather Hadoop library itself has been designed to detect and handle failures at the application layer (much lower operational costs).

Servers can be added or removed from the cluster **dynamically** and Hadoop continues to operate without interruption.

Another big advantage of Hadoop is that apart from being open source, it is compatible on **all the platforms since it is Java based**.

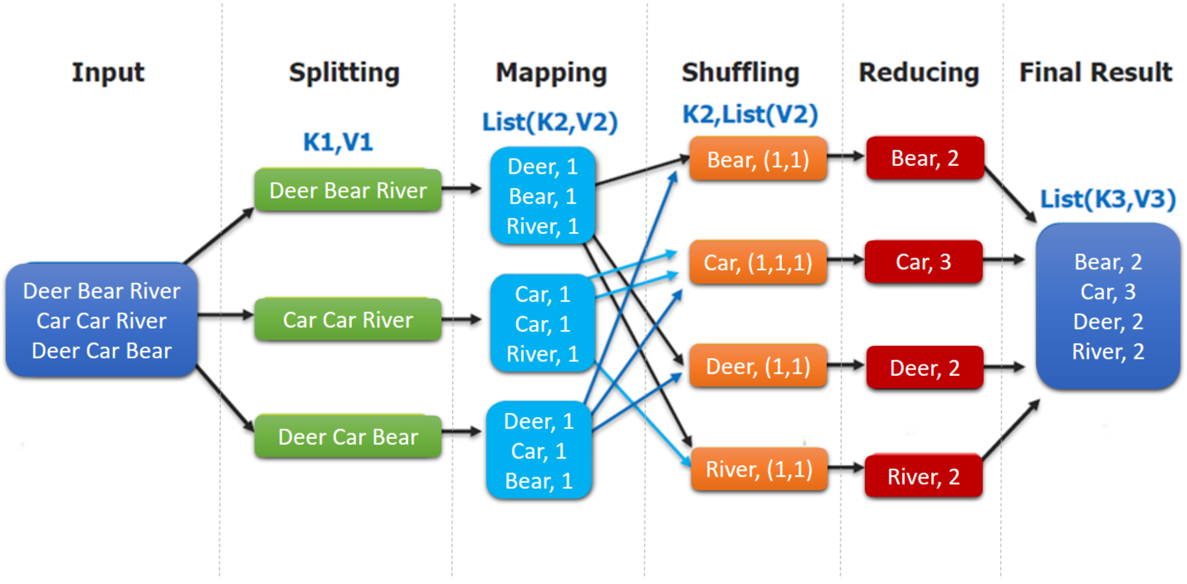
Although the Hadoop framework is implemented in Java, MapReduce **applications need not be written in Java**.

- **Hadoop Streaming** is an utility which allows users to create and run jobs with any executables (e.g. shell utilities) as the mapper and/or the reducer. We will show how to write a simple MapReduce application in Python using the Streaming module.
- **Hadoop Pipes** is a SWIG-compatible C++ API to implement MapReduce applications.

MapReduce

https://static.googleusercontent.com/media/research.google.com/pl//archive/mapreduce-osdi04.pdf

MapReduce is a parallel programming model for writing distributed applications devised at Google for efficient processing of large amounts of data (multi-terabyte data-sets), on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner. The MapReduce program runs on Hadoop which is an Apache open-source framework. Details of the model were published in 2004 in the famous paper: MapReduce: Simplified Data Processing on Large Clusters by Jeffrey Dean and Sanjay Ghemawat. (The paper has approximate 26633 number of citations!)



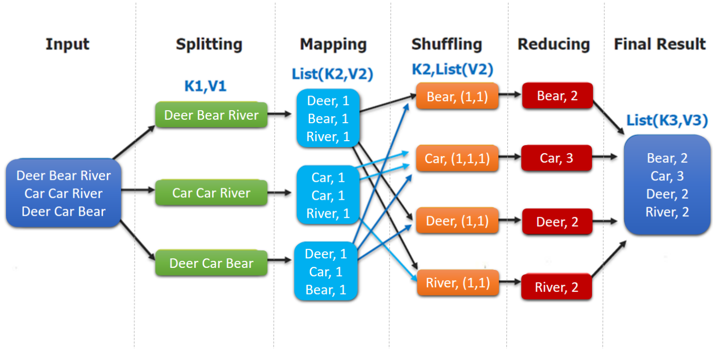
MapReduce Word Count Process

MapReduce Phases

https://datascienceguide.github.io/map-reduce

MapReduce is broken down into several steps:

1. Record Reader
2. Map
3. Combiner (Optional)
4. Partitioner
5. Shuffle and Sort
6. Reduce
7. Output Format

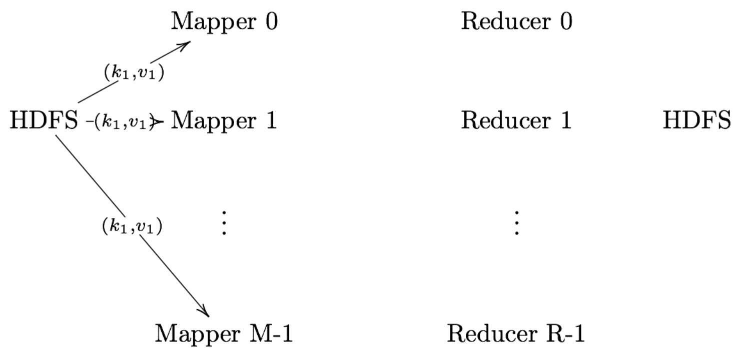


MapReduce Word Count Process

Record Reader

Record Reader **splits input** into fixed-size pieces for each mapper.

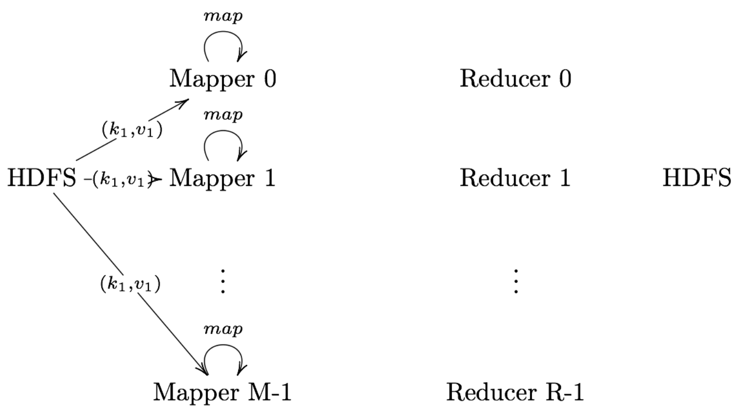
The key is positional information (the number of bytes from start of file) and **the value** is the chunk of data composing a single record.



In hadoop, each map task's is an input split which is usually simply a HDFS block Hadoop tries scheduling map tasks on nodes where that block is stored (**data locality**). If a file is broken mid-record in a block, hadoop requests the additional information from the next block in the series

Map

Map is a User defined function outputing intermediate key-value pairs. The keys allow to categorise data for future grouping, sorting or aggregating.

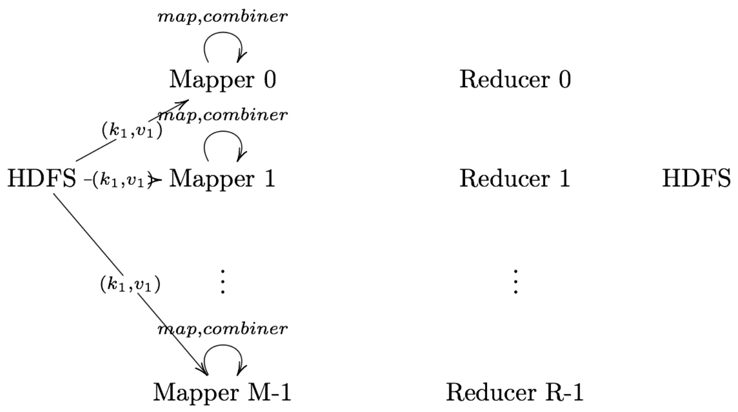


key (k1): Later, MapReduce will group and possibly aggregate data according to these keys, choosing the right keys is here is important for a good MapReduce job.

value (v1): The data to be grouped according to it’s keys.

Combiner (Optional)

Combiner UDF (*ang. user defined functions*) that aggregates data according to intermediate keys on a mapper node. (UDFs are a way to specify custom processing.)



This can usually **reduce the amount of data to be sent over the network** increasing efficiency

```
("hello world",1) \
("hello world",1)  | ⟶ combiner("hello world",3)
("hello world",1) /
```

Combiner should be written with the idea that it is executed over most but not all map tasks. ie. it will not be executed for single key-value pairs:

```
(k2,v2)↦(k2,v2)
```

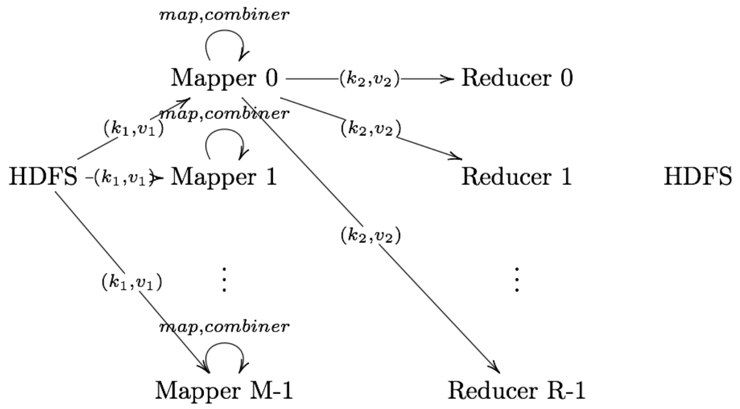
Usually the combiner is very similar or the same code as the reduce method.

Partitioner

Partitioner Sends intermediate key-value pairs (k,v) to reducer by

```
Reducer=hash(k) (mod R) % divided modulo by R
```

where R is the number of Reducer tasks



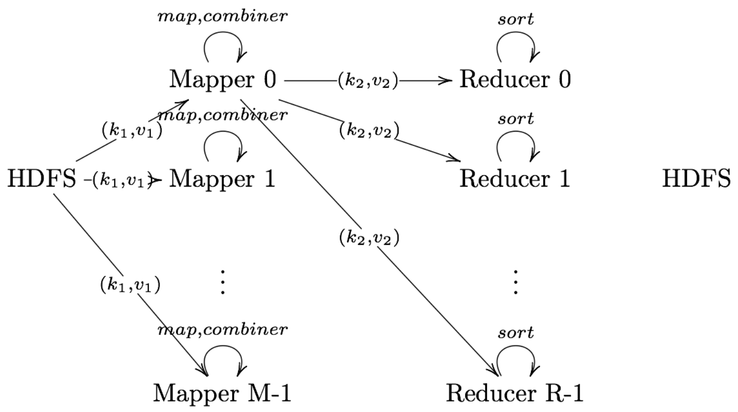
The distribution `hash(k) mod R` will usually result in a roughly balanced load accross the reducers while ensuring that all key-value pairs are grouped by their key on a single reducer.

A balancer system is in place for the cases when the key-values are too unevenly distributed.

In hadoop, the intermediate keys (k2,v2) are written to the local harddrive and grouped by reducer they will be sent to and their key.

Shuffle and Sort

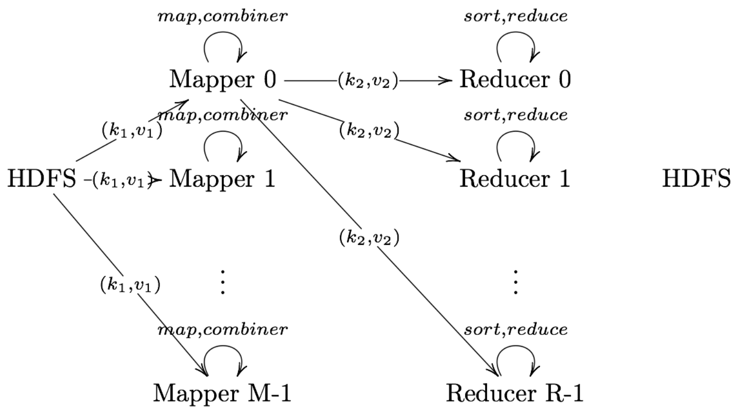
Shuffle and Sort On reducer node, sorts by key to help group equivalent keys



This option can be used for sorting large datasets on a given basis. We will have to provide the sort criterion as a propperly structured key (string value).

Reduce

Reduce is a User Defined Function that aggregates `data (v)` according to `keys (k)` to send again key-value pairs to output



The keys don't have to by unique and the same as were in the mapping phase, however it is usually the option for most cases.

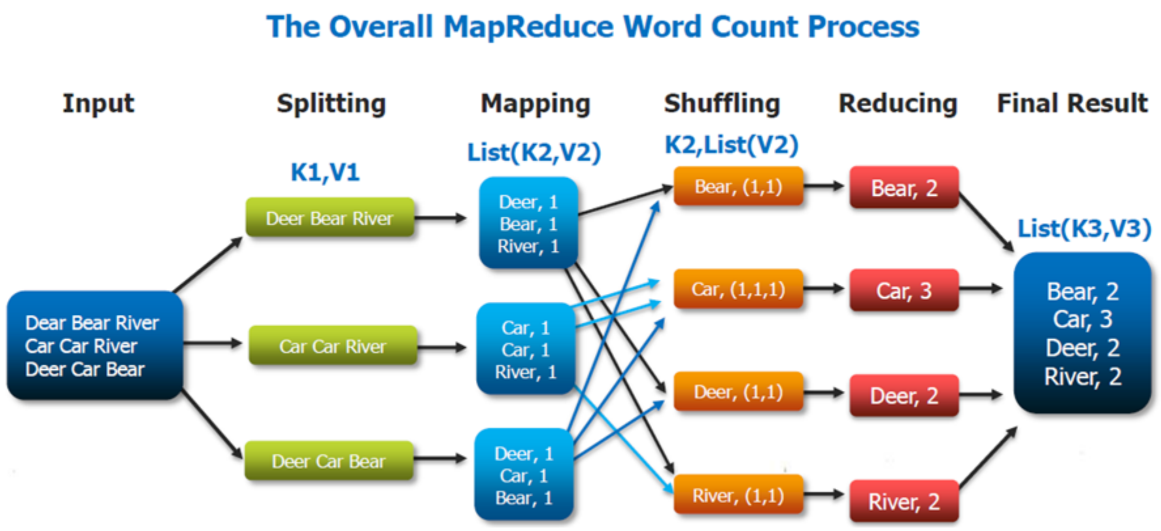
Output Format

Output Format Translates final key-value pairs to file format (tab-seperated by default).

$$\begin{tikzcd}
 & \text{Mapper 0} \arrow[loop above, "map{,}combiner"] \arrow[r, "(k_2{,}v_2)"] \arrow[dr, "(k_2{,}v_2)"] \arrow[dddr, "(k_2{,}v_2)"] & \text{Reducer 0} \arrow[loop above, "sort{,}reduce"] \arrow[dr, "(k_3{,}v_3)"] & \\
\text{HDFS} \arrow[ur, "(k_1{,}v_1)"] \arrow[r, "(k_1{,}v_1)"] \arrow[ddr, "(k_1{,}v_1)"] & \text{Mapper 1} \arrow[loop above, "map{,}combiner"] & \text{Reducer 1} \arrow[loop above, "sort{,}reduce"] \arrow[r, "(k_3{,}v_3)"] & \text{HDFS} \\
 & \vdots & \vdots & \\
 & \text{Mapper M-1} \arrow[loop above, "map{,}combiner"] & \text{Reducer R-1} \arrow[loop above, "sort{,}reduce"] \arrow[uur, "(k_3{,}v_3)"] &
\end{tikzcd}$$

Famous count words example

WordCount is a simple application that counts the number of occurrences of each word in a given input set.



WordCount - full code

```java
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

...
```

```java
  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Walk-through the code

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}
```

The Mapper implementation, via the map method, processes one line at a time, as provided by the specified TextInputFormat. It then splits the line into tokens separated by whitespaces, via the StringTokenizer, and emits a key-value pair of `< <word>, 1>`.

Given input files reminder:

```
$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World

$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

For the given sample input the first map emits:

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```

The second map emits:

```
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

Walk-trough continued - combiner

```
job.setCombinerClass(IntSumReducer.class);
```

WordCount also specifies a combiner. Hence, the output of each map is passed through the local combiner (which is same as the Reducer as per the job configuration) for local aggregation, after being sorted on the *key*s.

The output of the first map:

```
< Bye, 1>
< Hello, 1>
< World, 2>
```

The output of the second map:

```
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
```

Walthrough - reducer

```
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context
                   ) throws IOException, InterruptedException {
  int sum = 0;
  for (IntWritable val : values) {
    sum += val.get();
  }
  result.set(sum);
  context.write(key, result);
}
```

he Reducer implementation, via the reduce method just sums up the values, which are the occurrence counts for each key (i.e. words in this example).

Thus the output of the job is:

```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>
```

WordCount usage 1/2

- First we have to set up environment variables

```
export JAVA_HOME=/usr/java/default
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

- Then we need to compile WordCount.java and create a jar:

```
$ bin/hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

- Assuming that:

```
/user/joe/wordcount/input - input directory in HDFS
/user/joe/wordcount/output - output directory in HDFS
```

- Providing samlple text-files as input

```
$ bin/hadoop fs -ls /user/joe/wordcount/input/
/user/joe/wordcount/input/file01
/user/joe/wordcount/input/file02

$ bin/hadoop fs -cat /user/joe/wordcount/input/file01
Hello World Bye World

$ bin/hadoop fs -cat /user/joe/wordcount/input/file02
Hello Hadoop Goodbye Hadoop
```

WordCount usage 2/2

- Run the application:

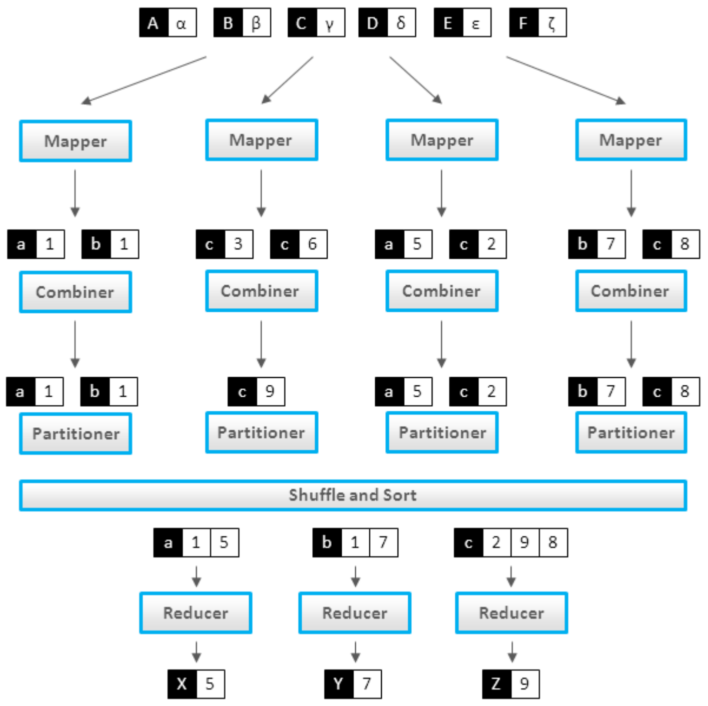
```
$ bin/hadoop jar wc.jar WordCount /user/joe/wordcount/input /user/joe/wordcount/output
```

- Output:

```
$ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

MapReduce examples

https://highlyscalable.wordpress.com/2012/02/01/mapreduce-patterns/



Basic MapReduce Patterns - Counting and Summing

The code snippet below shows Mapper that simply emit “1” for each term it processes and Reducer that goes through the lists of ones and sum them up:

```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, count 1)

class Reducer
   method Reduce(term t, counts [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
          sum = sum + c
      Emit(term t, count sum)
```

The obvious disadvantage of this approach is a high amount of dummy counters emitted by the Mapper. The Mapper can decrease a number of counters via summing counters for each document:

```
class Mapper
   method Map(docid id, doc d)
      H = new AssociativeArray
      for all term t in doc d do
          H{t} = H{t} + 1
      for all term t in H do
         Emit(term t, count H{t})
```

Basic MapReduce Patterns - Counting and Summing, cont. 1

In order to accumulate counters not only for one document, but for all documents processed by one Mapper node, it is possible to leverage Combiners:

```
class Mapper
   method Map(docid id, doc d)
      for all term t in doc d do
         Emit(term t, count 1)

class Combiner
   method Combine(term t, [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
          sum = sum + c
      Emit(term t, count sum)

class Reducer
   method Reduce(term t, counts [c1, c2,...])
      sum = 0
      for all count c in [c1, c2,...] do
          sum = sum + c
      Emit(term t, count sum)
```

MapReduce - Collating

Problem Statement: There is a set of items and some function of one item. It is required to save all items that have the same value of function into one file or perform some other computation that requires all such items to be processed as a group. The most typical example is building of inverted indexes.

Solution:

The solution is straightforward. Mapper computes a given function for each item and emits value of the function as a key and item itself as a value. Reducer obtains all items grouped by function value and process or save them. In case of inverted indexes, items are terms (words) and function is a document ID where the term was found.

Applications: Inverted Indexes, ETL (*ang. Extract, Transform and Load*)

MapReduce - Filtering (“Grepping”), Parsing, and Validation

Problem Statement: There is a set of records and it is required to collect all records that meet some condition or transform each record (independently from other records) into another representation. The later case includes such tasks as text parsing and value extraction, conversion from one format to another.

Solution: Solution is absolutely straightforward – Mapper takes records one by one and emits accepted items or their transformed versions.

Applications: Log Analysis, Data Querying, ETL, Data Validation

MapReduce - Distributed Task Execution

Problem Statement: There is a large computational problem that can be divided into multiple parts and results from all parts can be combined together to obtain a final result.

Solution: Problem description is split in a set of specifications and specifications are stored as input data for Mappers. Each Mapper takes a specification, performs corresponding computations and emits results. Reducer combines all emitted parts into the final result.

Case Study: Simulation of a Digital Communication System There is a software simulator of a digital communication system like WiMAX that passes some volume of random data through the system model and computes error probability of throughput. Each Mapper runs simulation for specified amount of data which is 1/Nth of the required sampling and emit error rate. Reducer computes average error rate.

Applications: Physical and Engineering Simulations, Numerical Analysis, Performance Testing

MapReduce - Sorting

Problem Statement: There is a set of records and it is required to sort these records by some rule or process these records in a certain order.

Solution: Simple sorting is absolutely straightforward – Mappers just emit all items as values associated with the sorting keys that are assembled as function of items. Nevertheless, in practice sorting is often used in a quite tricky way, that's why it is said to be a heart of MapReduce (and Hadoop). In particular, it is very common to use composite keys to achieve secondary sorting and grouping.

Sorting in MapReduce is originally intended for sorting of the emitted key-value pairs by key, but there exist techniques that leverage Hadoop implementation specifics to achieve sorting by values. See this blog for more details.

It is worth noting that if MapReduce is used for sorting of the original (not intermediate) data, it is often a good idea to continuously maintain data in sorted state using BigTable concepts. In other words, it can be more efficient to sort data once during insertion than sort them for each MapReduce query.

Applications: ETL, Data Analysis

Advanced MapReduce examples

- Iterative Message Passing (Graph Processing)
- Distinct Values (Unique Items Counting)
- Cross-Correlation

Distinct values example:

Problem Statement: There is a set of records. Each record has field F and arbitrary number of category labels G = {G1, G2, ...} . Count the total number of unique values of filed F for each subset of records for each value of any label. Example:

```
Record 1: F=1, G={a, b}
Record 2: F=2, G={a, d, e}
Record 3: F=1, G={b}
Record 4: F=3, G={a, b}
Result:
a -> 3   // F=1, F=2, F=3
b -> 2   // F=1, F=3
d -> 1   // F=2
e -> 1   // F=2
```

Solution:

We can solve the problem in **two stages**. At the first stage Mapper emits dummy counters for each pair of F and G; Reducer calculates a total number of occurrences for each such pair. The main goal of this phase is to guarantee uniqueness of F values. At the second phase pairs are grouped by G and the total number of items in each group is calculated.

How we can integrate in Python - Hands on example

Installation of Hadoop

https://datawookie.netlify.com/blog/2017/07/installing-hadoop-on-ubuntu/

1. Install the Java Development Kit.

```
$ sudo apt-get install default-jdk
```

2. Download the latest release of Hadoop here.
3. Unpack the archive.

```
$ tar -xvf hadoop-3.1.2.tar.gz
```

4. Move the resulting folder.

```
$ sudo mv hadoop-3.1.2 /usr/local/hadoop
```

5. Find the location of the Java package.

```
$ readlink -f /usr/bin/java | sed "s#bin/java##"
/usr/lib/jvm/java-8-openjdk-amd64/jre/
```

6. Edit the Hadoop configuration file at `/usr/local/hadoop/etc/hadoop/hadoop-env.sh` and set `JAVA_HOME`.

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre/
```

7. Test.

```
\$ /usr/local/hadoop/bin/hadoop version
```

Setting up a Single Node Cluster 1/3

https://hadoop.apache.org/docs/r3.1.2/hadoop-project-dist/hadoop-common/SingleCluster.html

By default, Hadoop is configured to run in a non-distributed mode, as a single Java process. This is useful for debugging.

- Pseudo-Distributed Operation

Hadoop can also be run on a single-node in a pseudo-distributed mode where each Hadoop daemon runs in a separate Java process.

Use the following:

.1. In `etc/hadoop/core-site.xml`:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

.2. In `etc/hadoop/hdfs-site.xml`:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

Setting up a Single Node Cluster 2/3

Now check that you can ssh to the localhost without a passphrase:

```
$ ssh localhost
```

If you cannot ssh to localhost without a passphrase, execute the following commands:

```
$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 0600 ~/.ssh/authorized_keys
```

Format the filesystem:

```
$ bin/hdfs namenode -format
```

Start NameNode daemon and DataNode daemon:

```
$ sbin/start-dfs.sh
```

The hadoop daemon log output is written to the $HADOOP_LOG_DIR directory (defaults to $HADOOP_HOME/logs).

Browse the web interface for the NameNode; by default it is available at:

```
NameNode - http://localhost:9870/
```

Setting up a Single Node Cluster 3/3

Make the HDFS directories required to execute MapReduce jobs:

```
$ bin/hdfs dfs -mkdir /myInputDirs
```

Copy the input files into the distributed filesystem:

```
  $ bin/hdfs dfs -put /etc/passwd myInputDirs
```

Run some of the examples provided:

```
./bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-3.1.2.jar grep user/szmurlor/input /user/szmurlor/output3 'a'
```

Examine the output files: Copy the output files from the distributed filesystem to the local filesystem and examine them:

```
  $ bin/hdfs dfs -get output output
  $ cat output/\*
```

or view the output files on the distributed filesystem:

```
  $ bin/hdfs dfs -cat output
```

When you’re done, stop the daemons with:

```
$ sbin/stop-dfs.sh
```

Running Python using streaming

Word count Python example - mapper

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

import sys

def main(argv):
    with sys.stdin as fin:
        for line in fin:
            line = line.strip()

            # Break the line into words
            words = line.split(":")

            # Iterate the words list
            for myword in words:
                if myword:
                    # Write the results to standard output
                    print '%s\t%s' % (myword.strip(), 1)


if __name__ == "__main__":
    main(sys.argv)
```

Word count Python example - reducer

```python
#!/usr/bin/python3
from operator import itemgetter
import sys

def main(argv):
    current_word = ""
    current_count = 0
    word = ""

    for line in sys.stdin:
        # Input takes from standard input
        # Remove whitespace either side
        line = line.strip()

        # Split the input we got from mapper.py
        word, count = line.split('\t', 1)

        # Convert count variable to integer
        try:
            count = int(count)
        except ValueError:
            # Count was not a number, so silently ignore this line continue
            pass

        if current_word == word:
            current_count += count
        else:
            if current_word:
                # Write result to standard output
                print(f'{current_word}\t{current_count}')
            current_count = count
            current_word = word

    # Do not forget to output the last word if needed!
    if current_word == word:
        print(f'{current_word}\t{current_count}')

if __name__ == "__main__":
    main(sys.argv)
```

Word count Python example - logging

```python
#!/usr/bin/python3

from operator import itemgetter
import sys
import logging

def main(argv):
    log = logging.getLogger("my-reducer")
    hdlr = logging.FileHandler("/tmp/my-reducer.log")
    log.addHandler(hdlr)
    log.setLevel(logging.DEBUG)

    current_word = ""
    current_count = 0
    word = ""

    for line in sys.stdin:
        # Input takes from standard input
        # Remove whitespace either side
        line = line.strip()

        # Log a line here
        log.info(line)
        # Split the input we got from mapper.py
        word, count = line.split('\t', 1)

(...)
```

Machine Learning and Math MapReduce Algorithms

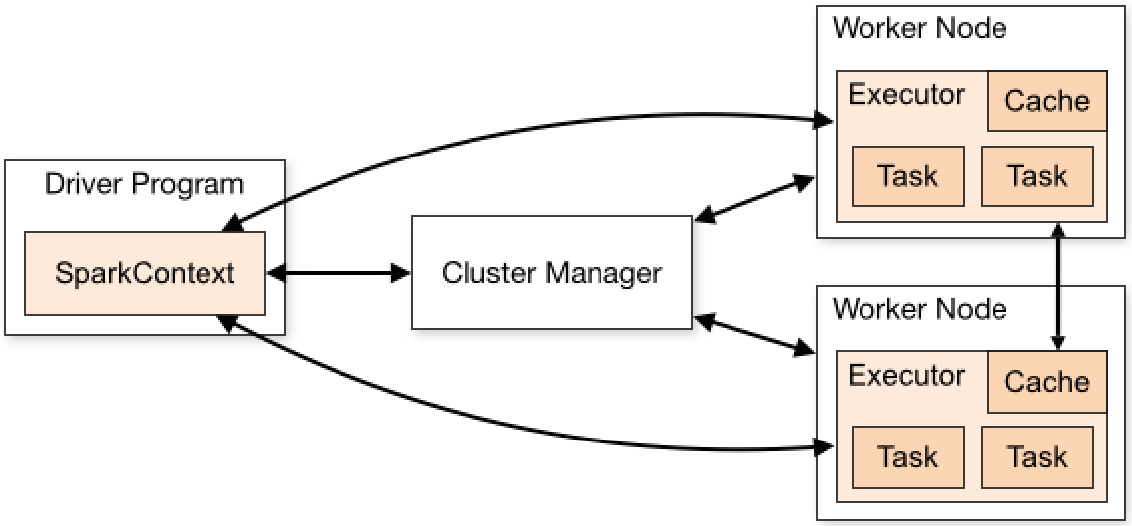
Optional reading!

C. T. Chu et al provides an excellent description of machine learning algorithms for MapReduce in the article Map-Reduce for Machine Learning on Multicore.

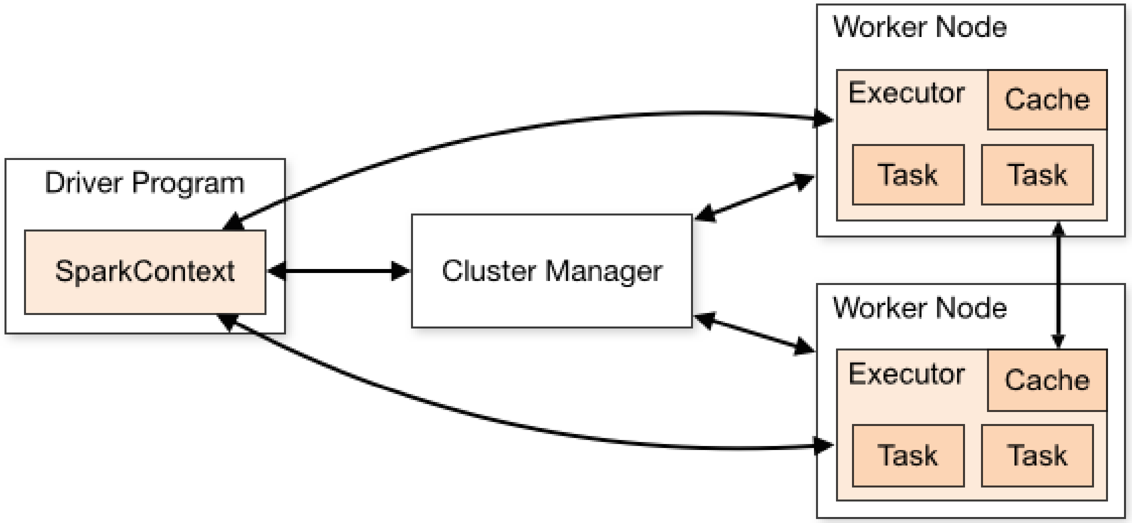
Spark

https://en.wikipedia.org/wiki/Apache_Spark

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.



Elements of Apache Spark



The `Spark context`, can be defined via a `Spark configuration` object, and a Spark URL. The `Spark context` connects to the Spark `cluster manager`, which then allocates resources across the `worker nodes` for the application. The `cluster manager` allocates executors across the cluster `worker nodes`. It copies the application jar file to the workers, and finally it allocates tasks.

Apache Spark work models

Local By specifying a Spark configuration local URL, it is possible to have the application run locally. By specifying local[n], it is possible to have Spark use threads to run the application locally. This is a useful development and test option. Standalone Standalone mode uses a basic cluster manager that is supplied with Apache Spark. The spark master URL will be as follows: Spark://:7077

Apache YARN At a larger scale when integrating with Hadoop YARN

Apache Mesos Apache Mesos is an open source system for resource sharing across a cluster. It allows multiple frameworks to share a cluster by managing and scheduling resources.

Amazon EC2 The Apache Spark release contains scripts for running Spark in the cloud against Amazon AWS EC2-based servers.

Apache Spark - comparison to Hadoop

- keeps data in memory
- intermediate results are not saved on disk
- provide interactive shell (REPL)

Introduction to Spark and Python

We shall use Spark with Python by using the pyspark library!

Creating a SparkContext

First we need to create a SparkContext. We will import this from pyspark:

```
from pyspark import SparkContext
```

Now create the SparkContext,A SparkContext represents the connection to a Spark cluster, and can be used to create an RDD and broadcast variables on that cluster.

Note! You can only have one SparkContext at a time the way you are running things.

```
sc = SparkContext()
```

Apache Spark on Google Colab

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q http://apache.osuosl.org/spark/spark-2.4.0/spark-2.4.0-bin-hadoop2.7.tgz
!tar xf spark-2.4.0-bin-hadoop2.7.tgz
!pip install -q findspark

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-2.4.0-bin-hadoop2.7"

import findspark
findspark.init()
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()

sc = spark.sparkContext
rdd = sc.parallelize([1,2,3])
rdd.collect()
```

Creating the RDD

Now we can take in some textfile using the textFile method off of the SparkContext we created. This method will read a text file from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings.

```
textFile = sc.textFile('example.txt')
```

Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop InputFormats (such as HDFS files) or by transforming other RDDs.

Actions

We have just created an RDD using the textFile method and can perform operations on this object, such as counting the rows.

RDDs have actions, which return values, and transformations, which return pointers to new RDDs. Let's start with a few actions:

```
textFile.count()
textFile.first()
```

Spark - transformations

The filter transformation will return a new RDD with a subset of items in the file.

Let's create a sample transformation using the filter() method. This method (just like Python's own filter function) will only return elements that satisfy the condition. Let's try looking for lines that contain the word 'second'. In which case, there should only be one line that has that.

```
secfind = textFile.filter(lambda line: 'second' in line)
secfind
```

Spark RDDs and transformation results are 'lazy initialized'. This means, that the transformations are exceuted when they are used. So lets collect the transformation (the filtered row) by calling:

```
secfind.collect()
```

or we can count filtered objects

```
secfind.count()
```

RDD Transformations and Actions

Definitions and important Terms:

TERM	DEFINITION
RDD	Resilient Distributed Dataset
Transformation	Spark operation that produces an RDD
Action	Spark operation that produces a local object
Spark Job	Sequence of transformations on data with a final action

Creating an RDD

There are two common ways to create an RDD:

METHOD	RESULT
`sc.parallelize(array)`	Create RDD of elements of array (or list)
`sc.textFile(path/to/file)`	Create RDD of lines from file

RDD Transformations

We can use transformations to create a set of instructions we want to preform on the RDD (before we call an action and actually execute them).

TRANSFORMATION EXAMPLE	RESULT
`filter(lambda x: x % 2 == 0)`	Discard non-even elements
`map(lambda x: x * 2)`	Multiply each RDD element by `2`
`map(lambda x: x.split())`	Split each string into words
`flatMap(lambda x: x.split())`	Split each string into words and flatten sequence
`sample(withReplacement=True,0.25)`	Create sample of 25% of elements with replacement
`union(rdd)`	Append `rdd` to existing RDD
`distinct()`	Remove duplicates in RDD
`sortBy(lambda x: x, ascending=False)`	Sort elements in descending order

RDD Actions

Once you have your 'recipe' of transformations ready, what you will do next is execute them by calling an action. Here are some common actions:

ACTION	RESULT
`collect()`	Convert RDD to in-memory list
`take(3)`	First 3 elements of RDD
`top(3)`	Top 3 elements of RDD
`takeSample(withReplacement=True,3)`	Create sample of 3 elements with replacement
`sum()`	Find element sum (assumes numeric elements)
`mean()`	Find element mean (assumes numeric elements)
`stdev()`	Find element deviation (assumes numeric elements)

Hands on examples

- Creating an RDD from a text file
- Map vs flatMap
- RDDs and Key Value Pairs
- Using Key Value Pairs for Operations

Naive K-Means in Spark - Part 1

The K-means algorithm written from scratch against PySpark. In practice, one may prefer to use the KMeans algorithm in ML, as shown in Spark examples: examples/src/main/python/ml/kmeans_example.py.

```python
from __future__ import print_function
import sys
import numpy as np
from pyspark.sql import SparkSession

# ... defs from Part 2

if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("Usage: kmeans <file> <k> <convergeDist>", file=sys.stderr)
        sys.exit(-1)

    spark = SparkSession.builder.appName("PythonKMeans").getOrCreate()

    lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
    data = lines.map(parseVector).cache()
    K = int(sys.argv[2])
    convergeDist = float(sys.argv[3])

    kPoints = data.takeSample(False, K, 1)
    tempDist = 1.0

    while tempDist > convergeDist:
        closest = data.map(
            lambda p: (closestPoint(p, kPoints), (p, 1)))
        pointStats = closest.reduceByKey(
            lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))
        newPoints = pointStats.map(
            lambda st: (st[0], st[1][0] / st[1][1])).collect()

        tempDist = sum(np.sum((kPoints[iK] - p) ** 2) for (iK, p) in newPoints)

        for (iK, p) in newPoints:
            kPoints[iK] = p
```

Naive K-Means in Spark - Part 1

```
def parseVector(line):
    return np.array([float(x) for x in line.split(' ')])


def closestPoint(p, centers):
    bestIndex = 0
    closest = float("+inf")
    for i in range(len(centers)):
        tempDist = np.sum((p - centers[i]) ** 2)
        if tempDist < closest:
            closest = tempDist
            bestIndex = i
    return bestIndex
```

PySpark magic

```
/opt/spark-2.4.0-bin-hadoop2.7
export PYSPARK_PYTHON=python3
export SPARK_LOCAL_HOSTNAME=localhost
export SPARK_HOME='/opt/spark-2.4.0-bin-hadoop2.7'
export PATH=$SPARK_HOME:$PATH
export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH
```

Spark Best practices

https://www.datacamp.com/community/tutorials/apache-spark-python

- **Use Spark DataFrames** - Spark DataFrames are optimized and therefore also faster than RDDs. Especially when you're working with structured data, you should really consider switching your RDD to a DataFrame.
- **Don't call collect() on large RDDs** - By calling collect() on any RDD, you drag data back into your applications from the nodes. Each RDD element will be copy onto the single driver program, which will run out of memory and crash. Other functions that you can use to inspect your data are take() or takeSample(), but also countByKey(), countByValue() or collectAsMap() can help you out.
- **Reduce Your RDD Before Joining** - One of the most basic rules that you can apply when you're revising the chain of operations that you have written down is to make sure that you filter or reduce your data before joining it. This way, you avoid sending too much data over the network that you'll throw away after the join, which is already a good reason, right?
- **Avoid groupByKey() on large RDDs** - On big data sets, you're better off making use of other functions, such as reduceByKey(), combineByKey() or foldByKey(). When you use groupByKey(), all key-value pairs are shuffled around in the cluster. A lot of unnecessary data is being transferred over the network.

Spark Best practices

- **Broadcast Variables** - Spark because you can reduce the cost of launching a job over the cluster.
- **Avoid flatmap(), join() and groupBy() Pattern** - When you have two datasets that are grouped by key and you want to join them, but still keep them grouped, use cogroup() instead of the above pattern.
- **Spark UI** - This web interface allows you to monitor and inspect the execution of your jobs in a web browser, which is extremely important if you want to exercise control over your jobs. The Spark UI allows you to maintain an overview off your active, completed and failed jobs. -

Resources

https://scikit-learn.org/stable/modules/clustering.html#k-means
https://www.edureka.co/blog/pyspark-tutorial/
https://www.datacamp.com/community/tutorials/apache-spark-python
https://www.edureka.co/blog/pyspark-dataframe-tutorial/

Thank you