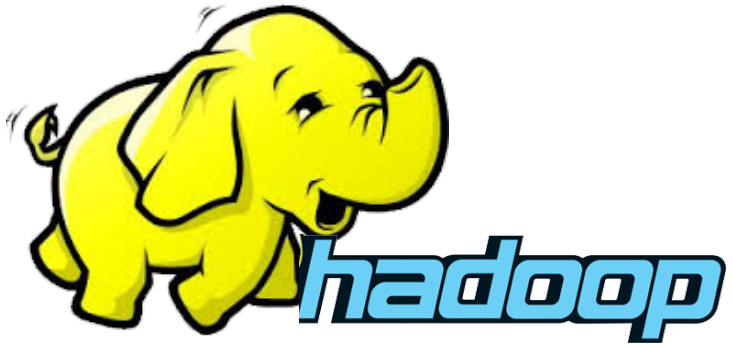
Hadoop



Adopted from slides by Nalini Venkatasubramanian,

Daniel Leblanc

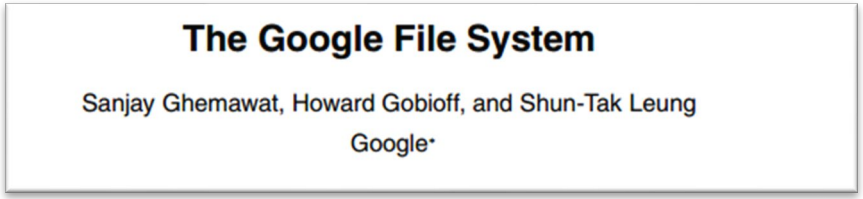
What is Apache Hadoop?

- An open-source software framework that supports data-intensive distributed applications, licensed under the Apache v2 license.
- Created by Doug Cutting and Michael Cafarella in 2005, for the Nutch search engine project funded by Yahoo.
  - Named after Cutting's son's toy elephant.
- Yahoo gave the project to Apache Software Foundation in 2006.

Google Origin

- Based on work done by Google in the early 2000s

2003



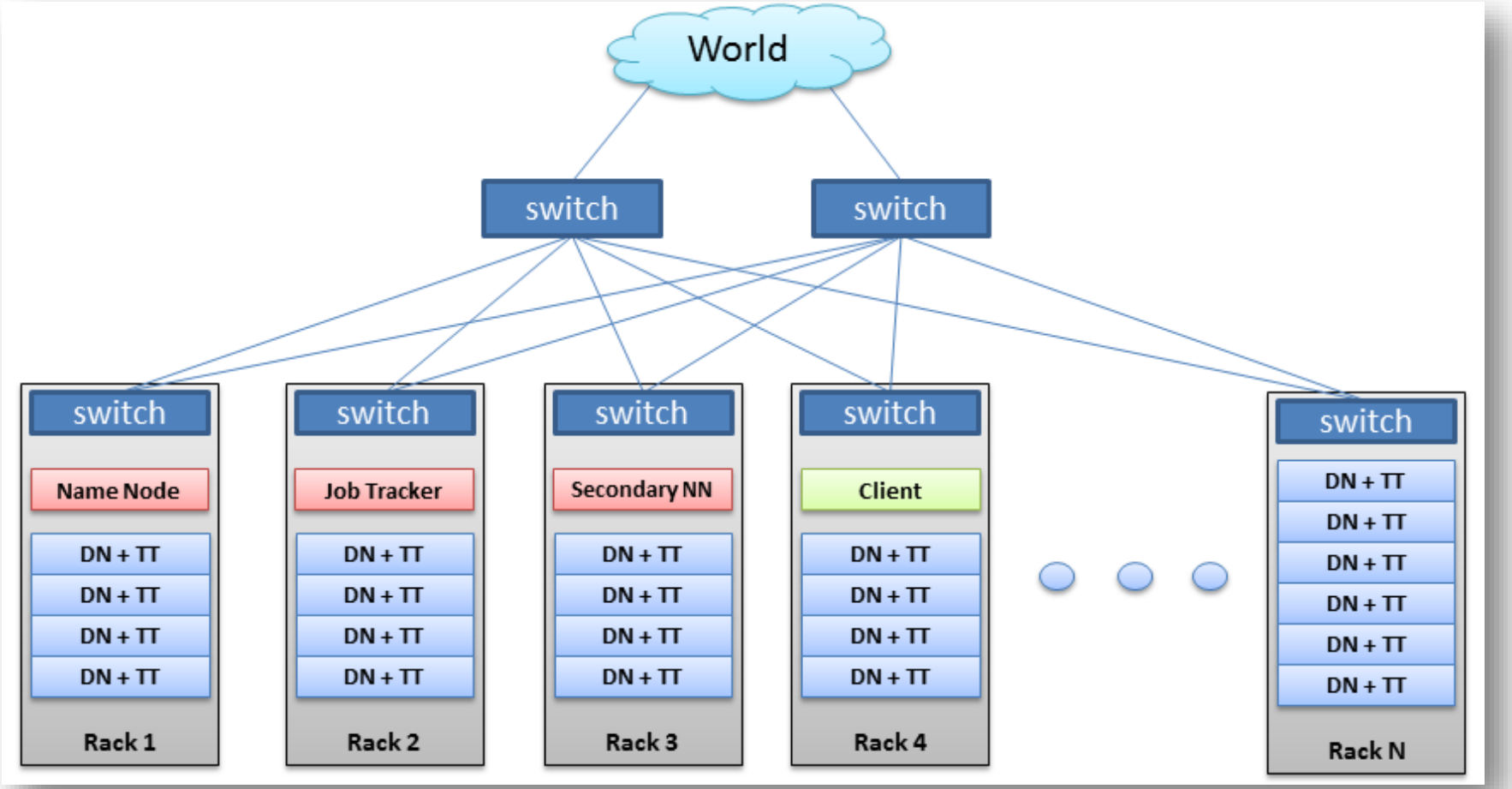
2004

Hadoop’s Architecture

- Central control nodes run NameNode to keep track of HDFS directories & files, and JobTracker to dispatch compute tasks to TaskTracker
- Main nodes run TaskTracker to accept and reply to MapReduce tasks, and also DataNode to store needed blocks closely as possible

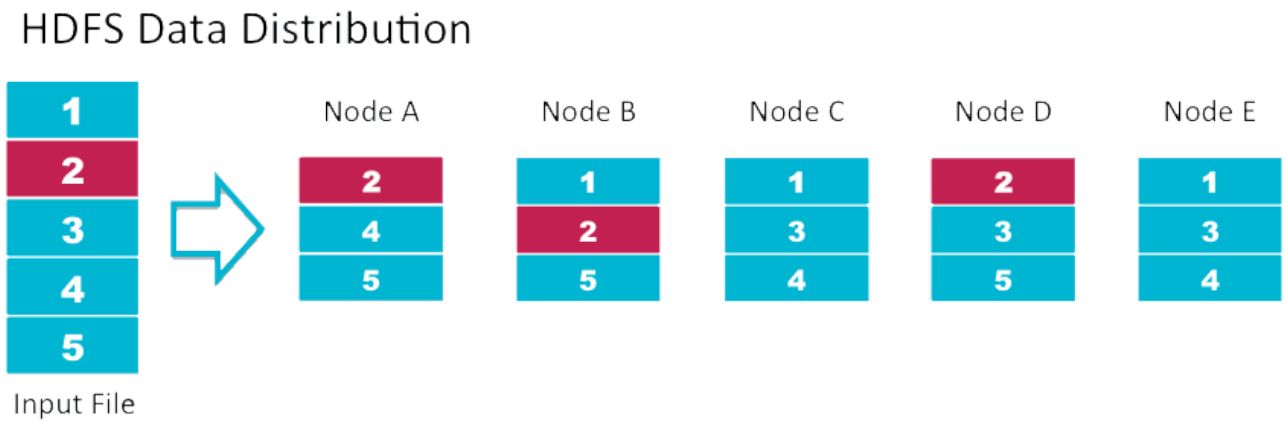
- Written in Java, also supports Python and Ruby

Hadoop's Architecture



Hadoop Distributed File System (HDFS)

- Files are divided into blocks (typically 64MB)
- Blocks are split across many machines at load time
  - Different blocks from the same file will be stored on different machines
- Blocks are replicated across multiple machines (3x by default)
  - 2x in local rack, 1x elsewhere



NameNode

- Keeps track of which blocks make up a file and where they are stored
- Stores metadata of the HDFS in a fsimage
- Updates to the file system (add/remove blocks) do not change the fsimage file
  - They are instead written to a log file
- When starting the NameNode loads the fsimage file and then applies the changes in the log file

Secondary NameNode

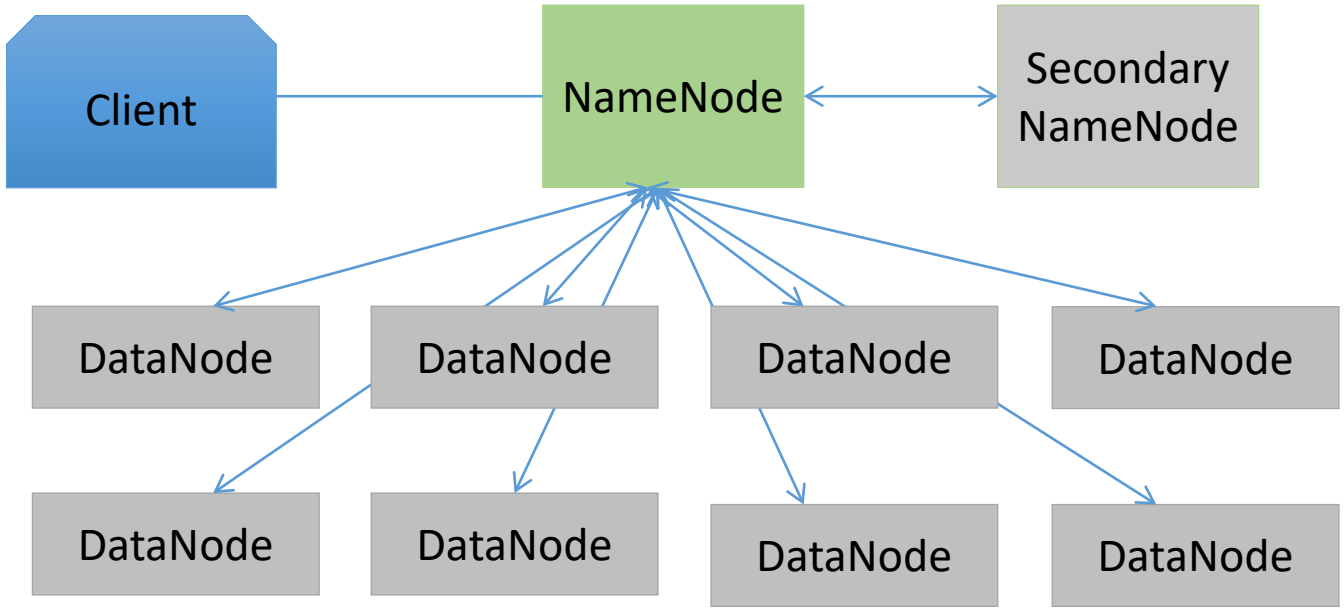
- **NOT** a backup for the NameNode
- Periodically reads the log file and applies the changes to the fsimage file bringing it up to date
- Allows the NameNode to restart faster when required

DataNode

- Stores the actual data in HDFS
- Can run on any underlying filesystem (ext3/4, NTFS, etc)
- Notifies NameNode of what blocks it has as block reports

Data Retrieval

- When a client wants to retrieve data
  - Communicates with the NameNode to determine which blocks make up a file and on which DataNodes those blocks are stored
  - Then communicate directly with the DataNodes to read the data

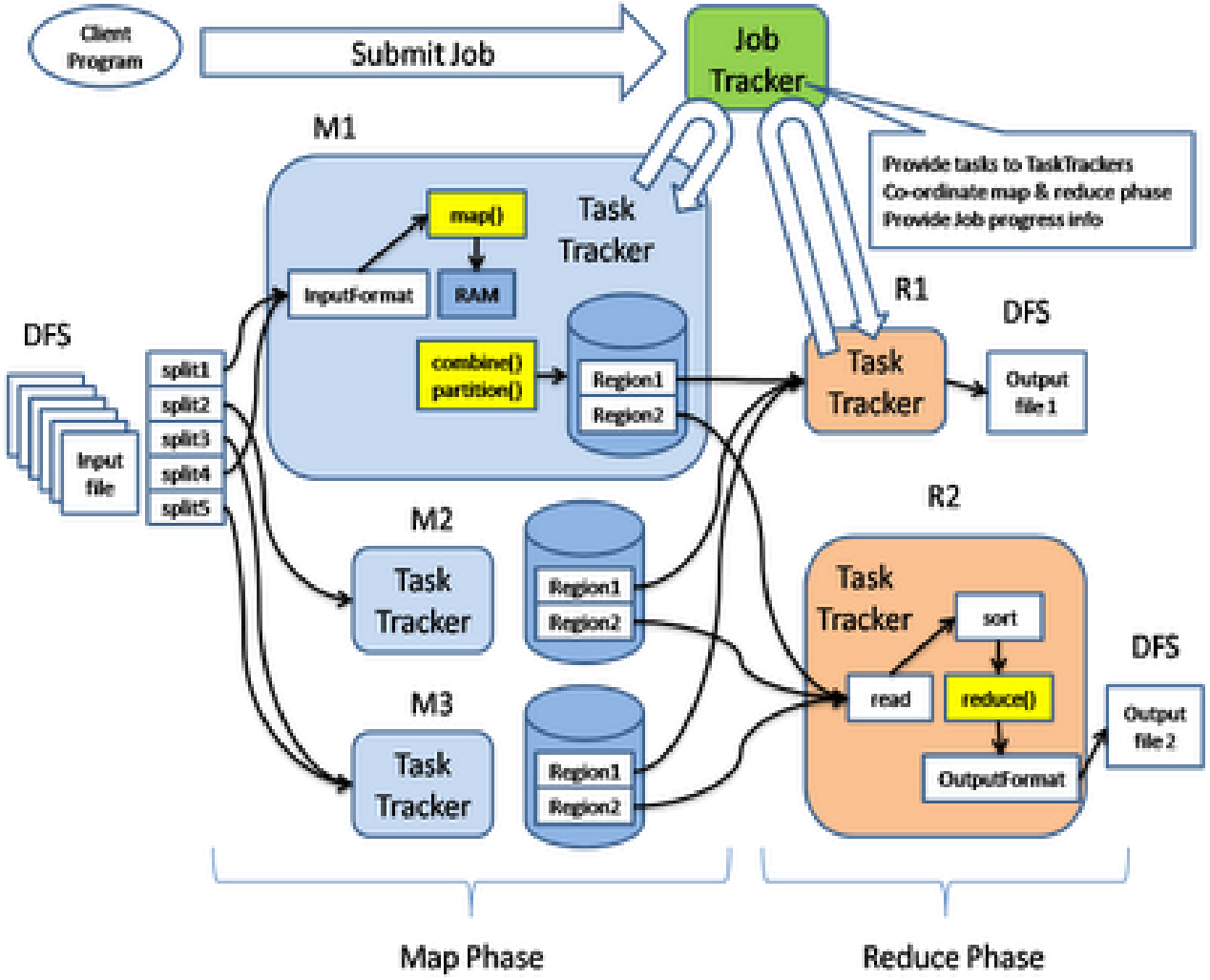


Hadoop MapReduce

- JobTracker determines the execution plan for the job and assigns individual tasks
  - Splits up data into smaller tasks (“Map”) and sends it to the TaskTracker process in each node
  - Keeps the work as close to the data as possible

- TaskTracker keeps track of the performance of an individual mapper or reducer
  - Reports back to the JobTracker node and reports on job progress, sends data (“Reduce”) or requests new jobs

MapReduce Engine

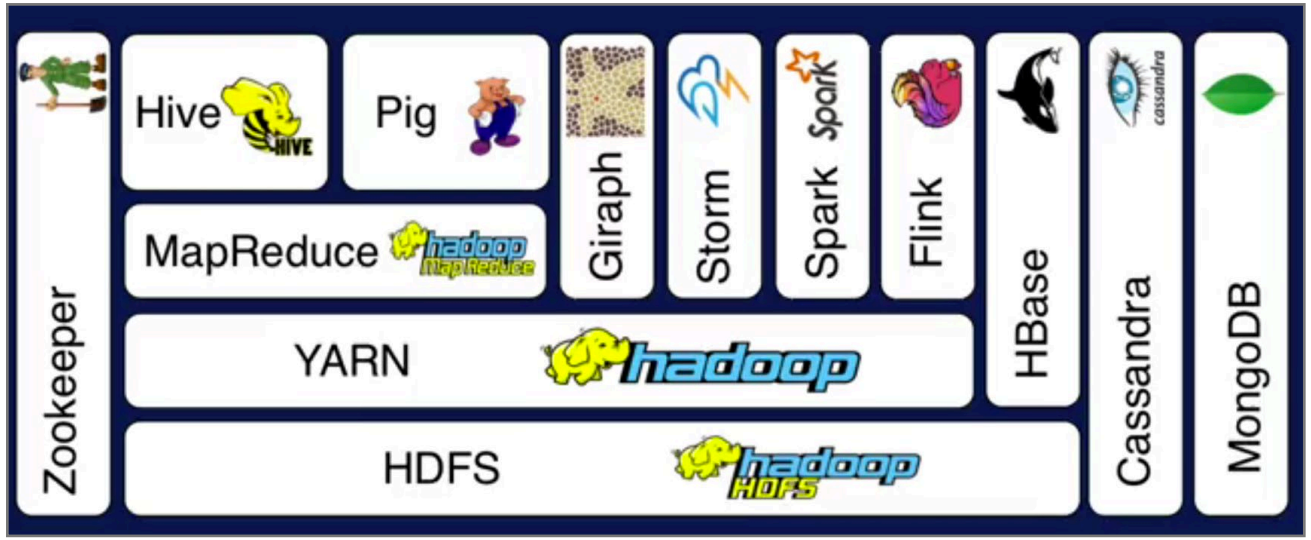


Fault Tolerance

- Failures are detected by the JobTracker which reassigns the work to a different node
- If a failed node restarts, it is added back to the system and assigned new tasks
- The JobTracker can redundantly execute the same task to avoid slow running nodes

Hadoop Ecosystem: Other Tools

- These tools allow programmers who are familiar with other programming styles to take advantage of the power of MapReduce
  - YARN: A framework for job scheduling and cluster resource management
  - Hive: Hadoop processing with SQL
  - Pig: Hadoop processing with scripting
  - Hbase: Database model built on top of Hadoop
  - …



Spark

Adopted from slides by David A. Wheeler,

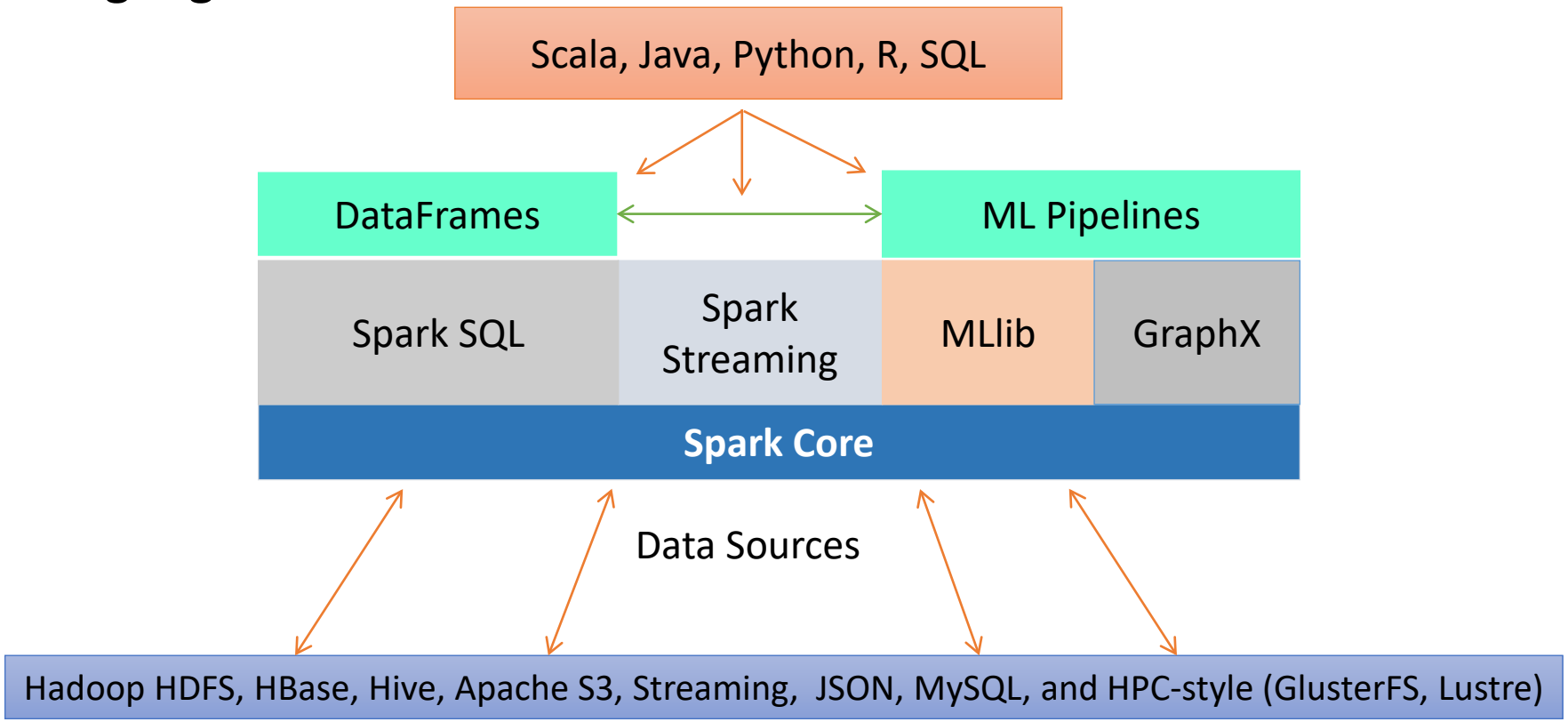
Afzal Godil

Apache Spark

- An open source software developed at AMPLab UC Berkeley, now by Databricks.com
- Supports Java, Scala, Python and R
- Capable of leveraging the Hadoop ecosystem, e.g. HDFS, YARN, Hbase, …
- A more general framework than Hadoop MapReduce, which more easily handles complicated workflows
  - Instead of just “map” and “reduce”, defines a large set of *operations* (transformations & actions)
- In-memory caching of data (for iterative, graph, and machine learning algorithms, etc.)

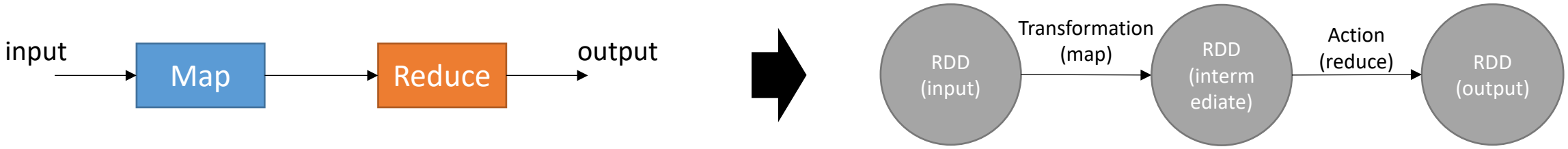
Apache Spark

- Apache Spark supports data analysis, machine learning, graphs, streaming data, etc. It can read/write from a range of data types and allows development in multiple languages.



Resilient Distributed Dataset (RDD) – key Spark construct

- RDDs represent data or transformations on data
- RDDs can be created from Hadoop InputFormats (such as HDFS files), “parallelize()” datasets, or by transforming other RDDs (you can stack RDDs)
- Actions can be applied to RDDs; actions force calculations and return values
- Lazy evaluation: Nothing computed until an action requires it
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
  - Less suitable for applications that make asynchronous fine-grained updates to shared state



Sample Spark transformations

- map(func): Return a new distributed dataset formed by passing each element of the source through a function func.
- flatMap(func): Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
- union(otherDataset): Return a new dataset that contains the union of the elements in the source dataset and the argument.
- intersection(otherDataset): Return a new RDD that contains the intersection of elements in the source dataset and the argument.
- distinct([numTasks])): Return a new dataset that contains the distinct elements of the source dataset
- groupByKey([numTasks]): When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
- join(otherDataset, [numTasks]): When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
- reduceByKey(func, [numPartitions]): When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.

Source: https://spark.apache.org/docs/latest/programming-guide.html

Sample Spark Actions

- reduce(func): Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- collect(): Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
- count(): Return the number of elements in the dataset.
- saveAsTextFile(path): Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system.

Remember: *Actions* cause calculations to be performed; *transformations* just set things up (lazy evaluation)

Source: https://spark.apache.org/docs/latest/programming-guide.html

Spark – RDD Persistence

- You can persist (cache) an RDD
- When you persist an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it)
- Allows future actions to be much faster (often > 10x).
- Mark RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes.
- Cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it
- Can choose storage level (MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, etc.)
- Can manually call unpersist()

Spark Example: Word Count

- Python

```
conf = SparkConf()
sc = SparkContext(conf=conf)
lines = sc.textFile(sys.argv[1])
words = lines.flatMap(lambda l: l.split(" "))
pairs = words.map(lambda w: (w, 1))
counts = pairs.reduceByKey(lambda n1, n2: n1 + n2)
counts.saveAsTextFile(sys.argv[2])
sc.stop()
```

- Java

```
public class WordCount {
  public static void main(String[] args) throws Exception {
    SparkConf conf = new SparkConf();
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD < String > lines = sc.textFile(args[0]);
    JavaRDD < String > words =
      lines.flatMap(l -> Arrays.asList(l.split(“ ”)).iterator());
    JavaPairRDD < String, Integer > pairs =
      words.mapToPair(w -> new Tuple2 < > (w, 1));
    JavaPairRDD < String, Integer > counts =
      pairs.reduceByKey((n1, n2) -> n1 + n2);
    counts.saveAsTextFile(args[1]);
    sc.stop();
  }
}
```

Word Count in Python

- Create a Spark context:

```
import re
import sys
from pyspark import SparkConf, SparkContext
conf = SparkConf()
sc = SparkContext(conf=conf)
```

Word Count in Python

- Next, you'll need to read the target file into an RDD:

```
lines = sc.textFile(sys.argv[1])
```

- You now have an RDD filled with strings, one per line of the file.
- Next you'll want to split the lines into individual words:

```
words = lines.flatMap(lambda l: l.split(" "))
```

- Next, you'll want to replace each word with a tuple of that word and the number 1.

```
pairs = words.map(lambda w: (w, 1))
```

Word Count in Python

- Now, to get a count of the number of instances of each word, you need only group the elements of the RDD by key (word) and add up their values:

```
counts = pairs.reduceByKey(lambda n1, n2: n1 + n2)
```

- Finally, you can store the results in a file and stop the context:

```
counts.saveAsTextFile(sys.argv[2])
sc.stop()
```

Word Count in Java

- The first step of every Java Spark application is to create a Spark context:

```
import java.util.Arrays;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import scala.Tuple2;
public class JavaWordCount {
  public static void main(String[] args) throws Exception {
    SparkConf conf = new SparkConf();
    JavaSparkContext sc = new JavaSparkContext(conf);
    ...
  }
}
```

Word Count in Java

- Next, you'll need to read the target file into an RDD:

```
JavaRDD<String> lines = sc.textFile(args[0]);
```

- You now have an RDD filled with strings, one per line of the file.
- Next you'll want to split the lines into individual words:

```
JavaRDD<String> words =
  lines.flatMap(l -> Arrays.asList(l.split(“ ”)).iterator());
```

- The flatMap() operation first converts each line into an array of words, and then makes each of the words an element in the new RDD. Note that the lambda argument to the method must return an iterator, not a list or array.

Word Count in Java

- Next, you'll want to replace each word with a tuple of that word and the number 1.

```
JavaPairRDD<String, Integer> pairs =
  words.mapToPair(w -> new Tuple2<>(w, 1));
```

- The mapToPair() operation replaces each word with a tuple of that word and the number 1. The pairs RDD is a pair RDD where the word is the key, and all of the values are the number 1. Note that the type of the RDD is now JavaPairRDD. Also note that the use of the Scala Tuple2 class is the normal and intended way to perform this operation.

Word Count in Java

- Now, to get a count of the number of instances of each word, you need only group the elements of the RDD by key (word) and add up their values:

```
JavaPairRDD<String, Integer> counts =
  pairs.reduceByKey((n1, n2) -> n1 + n2);
```

- The reduceByKey() operation keeps adding elements' values together until there are no more to add for each key (word).

Word Count in Java

- Finally, you can store the results in a file and stop the context:

```
counts.saveAsTextFile(args[1]);
sc.stop();
```

Lambda Expressions

- A lambda expression is a syntactic shortcut for defining the single abstract method of a functional interface and instantiating an anonymous class that implements the interface. The general syntax is
- Java: (*para1*, *para2*, …) -> {*method_body*}
  - If there is only one parameter, the parenthesis can be omitted.
  - If there is only one line in the method_body, the curly braces as well as the *return* keyword can be omitted.
- Python: lambda *para1*, *para2*, …: *method_body*

Spark vs. Hadoop MapReduce

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms Hadoop MapReduce, but it often needs lots of memory to do well; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - Hadoop MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: Spark is easier to program
- Data processing: Spark more general
- Maturity: Spark maturing, Hadoop MapReduce mature

"Spark vs. Hadoop MapReduce" by Saggi Neumann (November 24, 2014)
https://www.xplenty.com/blog/2014/11/apache-spark-vs-hadoop-mapreduce/

For more information

- Spark Programming Guide
  - https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html
- Spark tutorials
  - http://spark-summit.org/2014/training
- “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing” by Matei Zaharia et al
  - https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf