Lecture 03

Introduction to MapReduce and Spark

Csci E63 Big Data Analytics

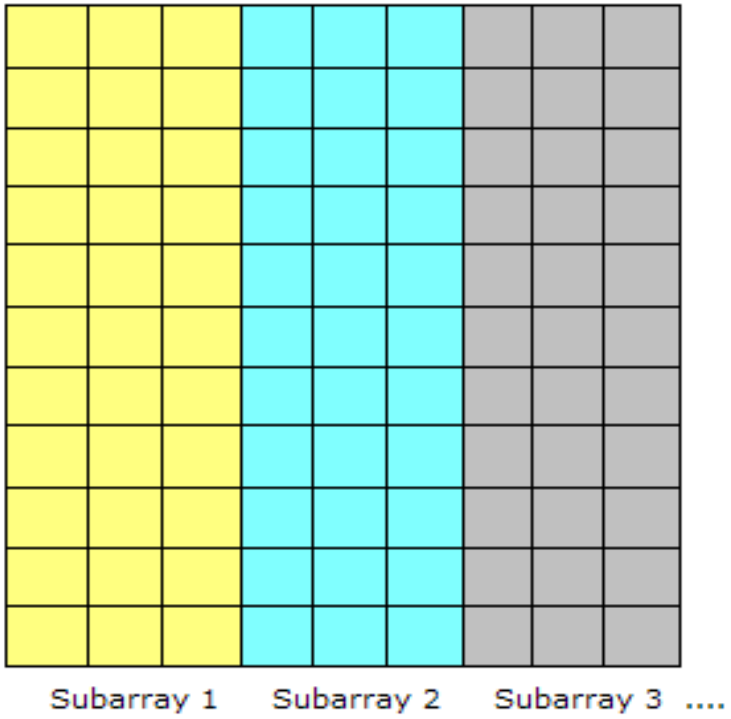
Zoran B. Djordjević

`Serial vs. Parallel` Programming Model

- Many or most of our programs are S*erial*.
  - A `Serial Program` consists of a sequence of instructions, where each instruction executes one after the other.
  - Serial programs run from start to finish on a single processor.
- *Parallel programming* developed as a means of improving performance and efficiency.
  - In a `Parallel Program`, the processing is broken up into parts, each of which could be executed concurrently on a different processor. Parallel programs could be faster.
  - Parallel Programs could also be used to solve problems involving large datasets and non-local resources.
  - Parallel Programs are usually ran on a set of computers connected on a network (a pool of CPUs), with an ability to read and write large files supported by a distributed file system.

Common Situation

- A common situation involves processing of a large amount of consistent data.
- If the data could be decomposed into equal-size partitions, we could devise a parallel solution. Consider a huge array which can be broken up into sub-arrays



If the same processing is required for each array element, with no dependencies in the computations, and no communication required between tasks, we have an ideal parallel computing opportunity, the so called *Embarrassingly Parallel problem.*
A common implementation of this approach is a technique called *Master/Worker*.

MapReduce Programming Model

- `MapReduce` programming model is derived as a technique for solving embarrassingly and not-so-embarrassingly parallel problems.
- The idea stems from the `map` and `reduce` combinators in Lisp programming language.
- In Lisp, a `map` takes as input a function and a sequence of values. It then applies the function to each value in the sequence. A `reduce` combines all the elements of a sequence using a binary operation. For example, it can use "+" to add up all the elements in the sequence.
- `MapReduce` was developed within Google as a mechanism for processing large amounts of raw data, for example, crawled documents or web request logs.

http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

- Google data is so large, it must be distributed across tens of thousands of machines in order to be processed in a reasonable time.
- The distribution implies parallel computing since the same computations are performed on each CPU, but with a different portion of data.

MapReduce Library

- `map` function, written by a user of the MapReduce library, takes `an input key/value` pair and produces `a set of intermediate key/value` pairs.
- The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the `reduce` function.
- The `reduce` function, also written by the user, accepts `an intermediate key` and `a set of values` for that key. The `reduce` function merges together these values to form a possibly smaller set of values.

Vocabulary and Number of Words in all Documents

- Consider the problem of counting the number of occurrences of each word in a large collection of documents

```
map(String documentName, String documentContent):
//key: document name, value: document content
for each word w in documentContent:
//key: word, value: number of occurances
  EmitIntermediate(w, wordCount);

reduce(String w, Iterator values):
// key: a word, // values: a list of counts
int result = 0;
for each v in values:
  result += v;
Emit(w, result));
```

- The map function emits each word plus an associated count of occurrences in a document.
- The reduce function sums all the counts for every word giving us the number of occurrances of each word in the entire set of documents.

MapReduce Programming Model, Again

- Data types: key-value *records (Keys and Values could be of different data types: ints, dates, strings, etc.)*
- Map function:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

- Intermediate keys do not have to be related to the initial keys in shape or form.
- Reduce function is fed sorted collection of intermediate values for each intermediate key.

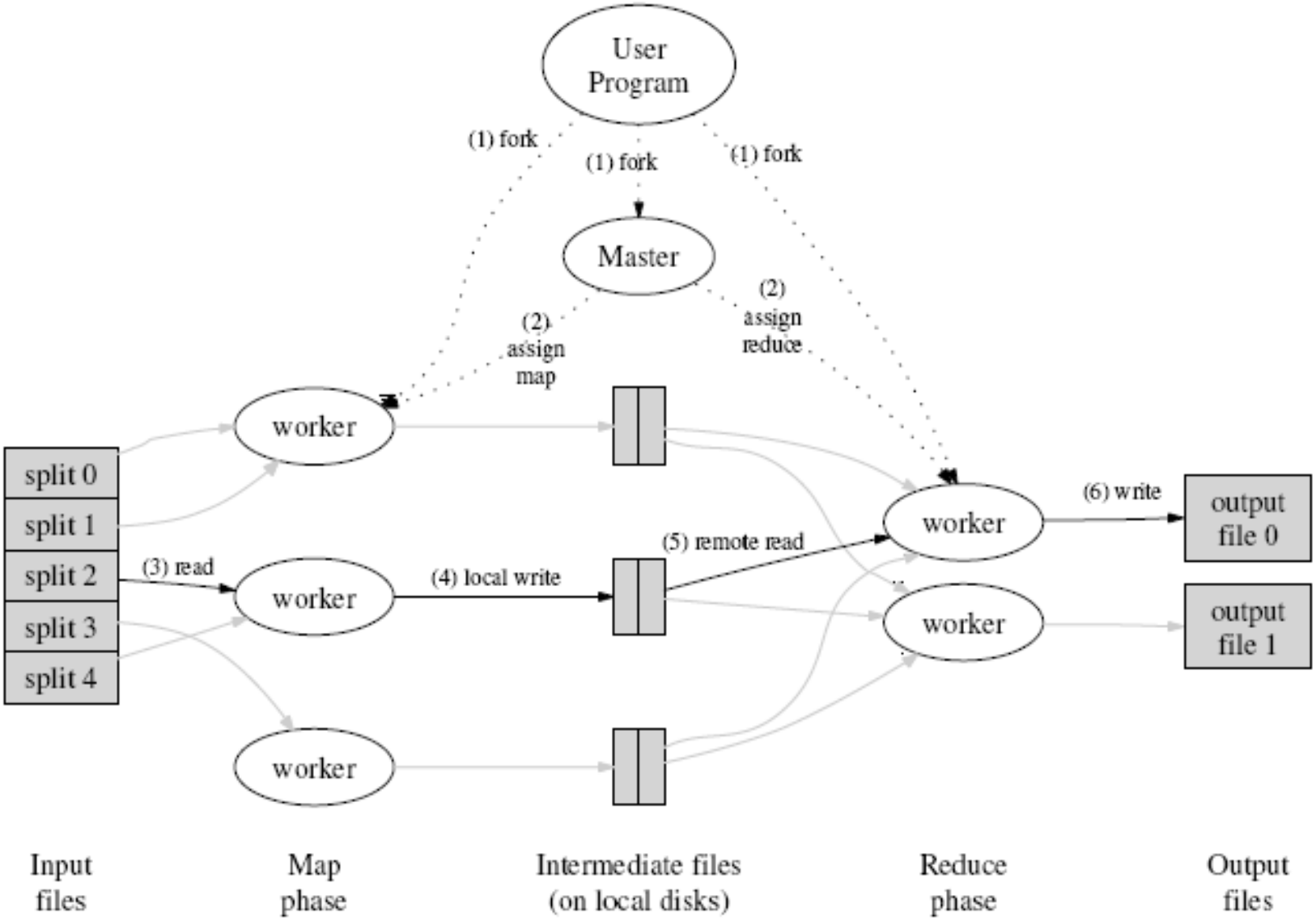
$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

- Reduce function transforms that collection into a final result, a list of key-value pairs.

MapReduce Execution

- The `Map` invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits or ***shards*.**
- The input shards can be processed in parallel on different machines.
- `reduce` invocations are distributed by partitioning the intermediate key space into `r` pieces using a partitioning function (e.g., `hash(key) mod r`).
- The number of partitions (`r`) and the partitioning function are specified by the user.

Flow of Execution



Open Source MapReduce

- We will not build MapReduce frameworks.
- We will learn to use an open source MapReduce Framework called Hadoop which is offered as Apache project, as well by commercial vendors: Cloudera, Hortonworks, MapR, IBM, and many other vendors.
- Non commercial version of Hadoop source code is available at *apache.org*.
- Each vendor improves on original, open source design and sells its improved version commercially.
- **Hadoop makes massive parallel computing on large clusters (thousands of cheap, commodity machines) possible.**

Welcome to Apache™ Hadoop®!

What Is Apache Hadoop?

The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers usi is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardwar library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of con prone to failures.

The project includes these modules:

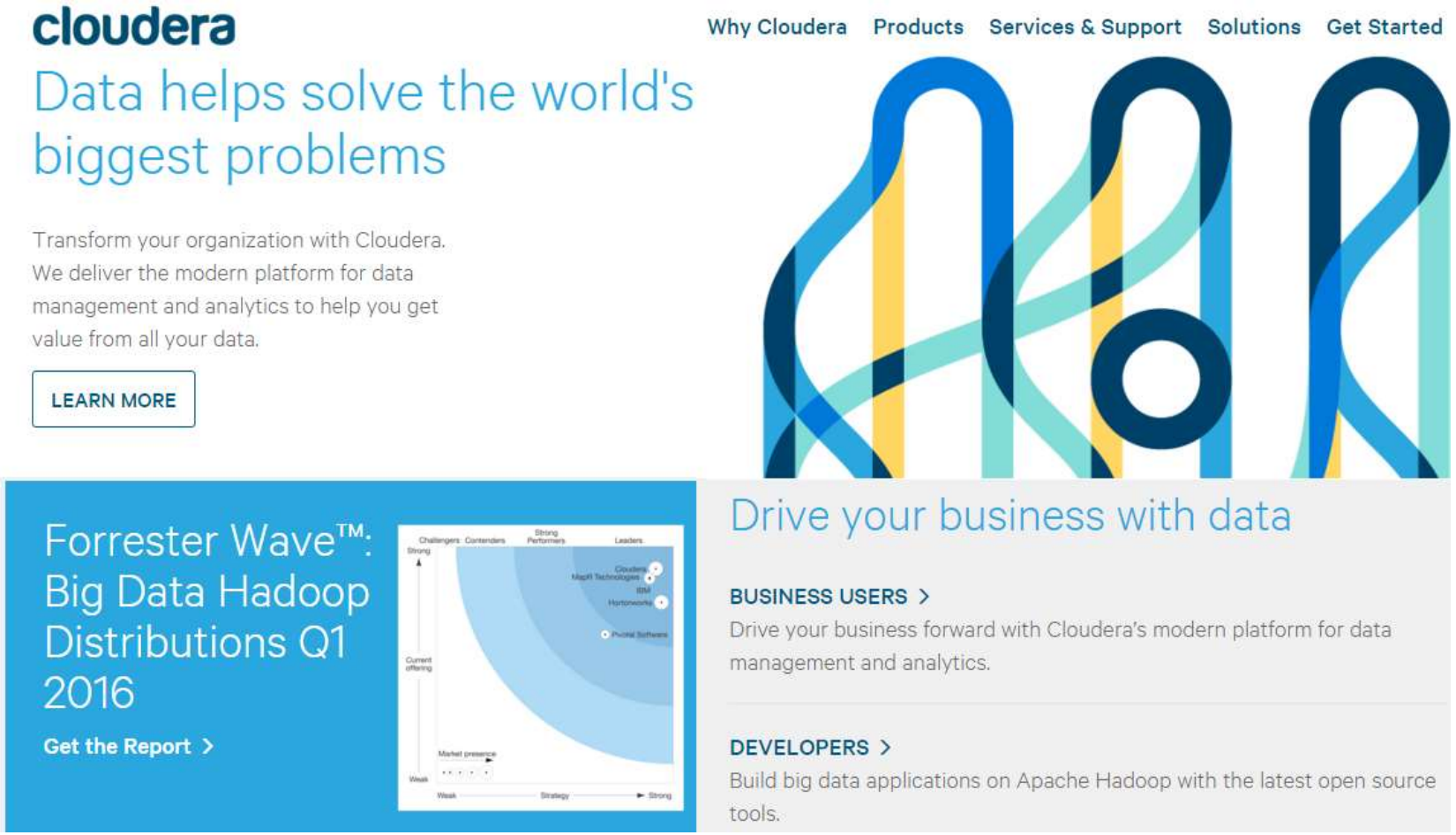
- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

Other Hadoop-related projects at Apache include:

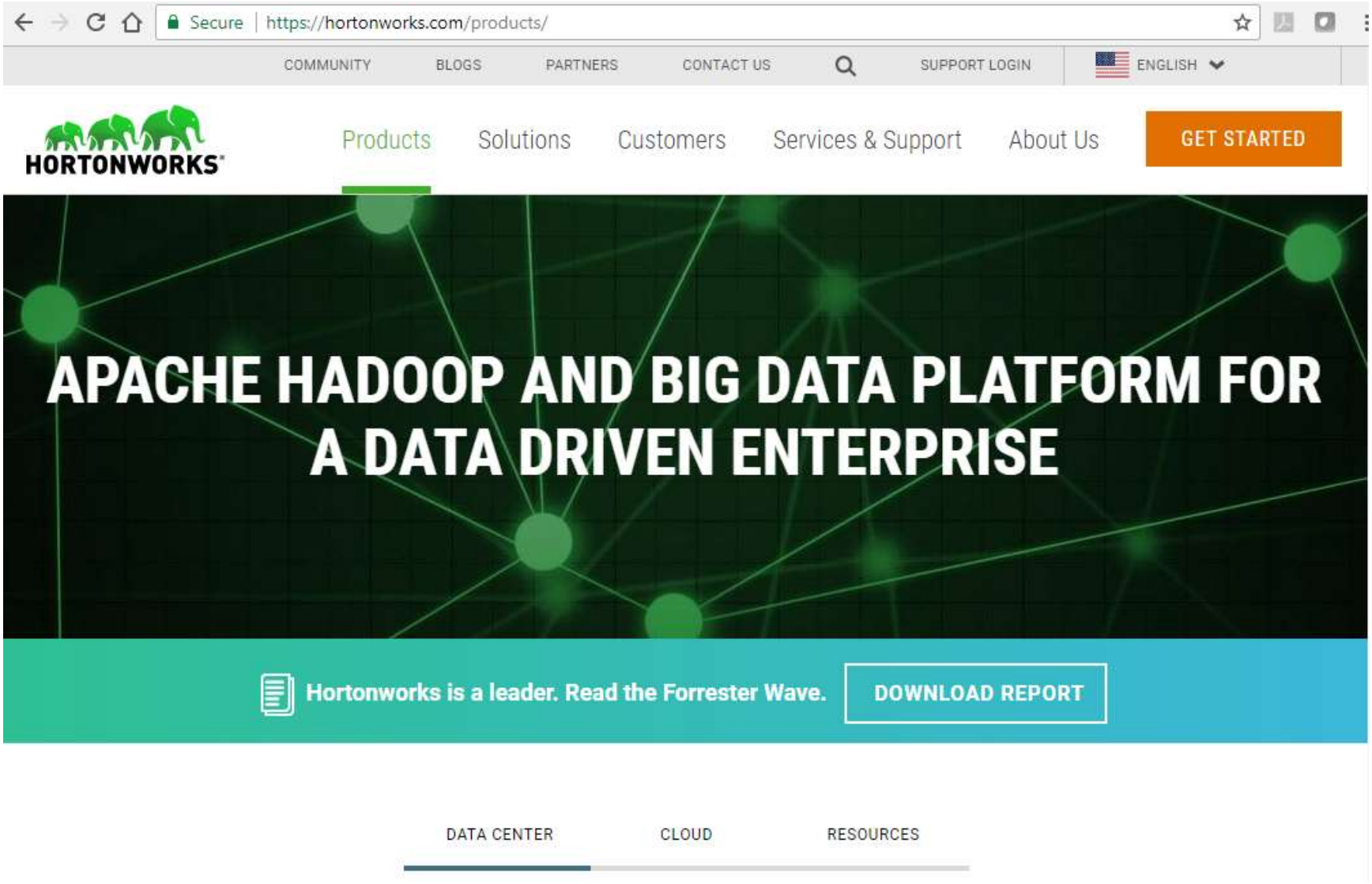
- **Ambari™**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heatmaps and ab and Hive applications visually alongwith features to diagnose their performance characteristics in a user-friendly manner.
- **Avro™**: A data serialization system.
- **Cassandra™**: A scalable multi-master database with no single points of failure.
- **Chukwa™**: A data collection system for managing large distributed systems.
- **HBase™**: A scalable, distributed database that supports structured data storage for large tables.
- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- **Mahout™**: A Scalable machine learning and data mining library.
- **Pig™**: A high-level data-flow language and execution framework for parallel computation.
- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a including ETL, machine learning, stream processing, and graph computation.
- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute a process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosyste commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.
- **ZooKeeper™**: A high-performance coordination service for distributed applications.

Cloudera

- Doug Cutting who invented Hadoop is at Cloudera.

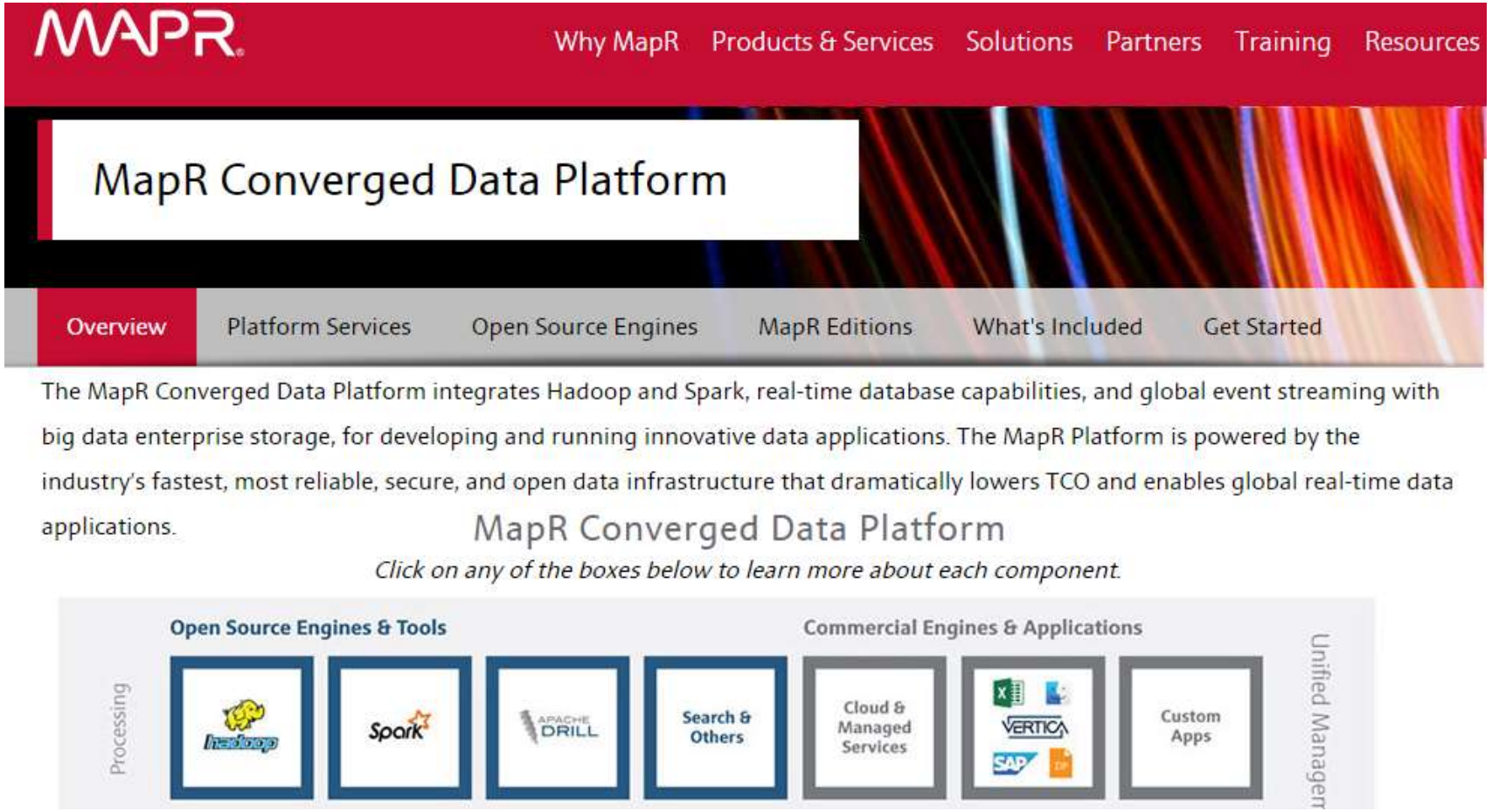


Hortonworks

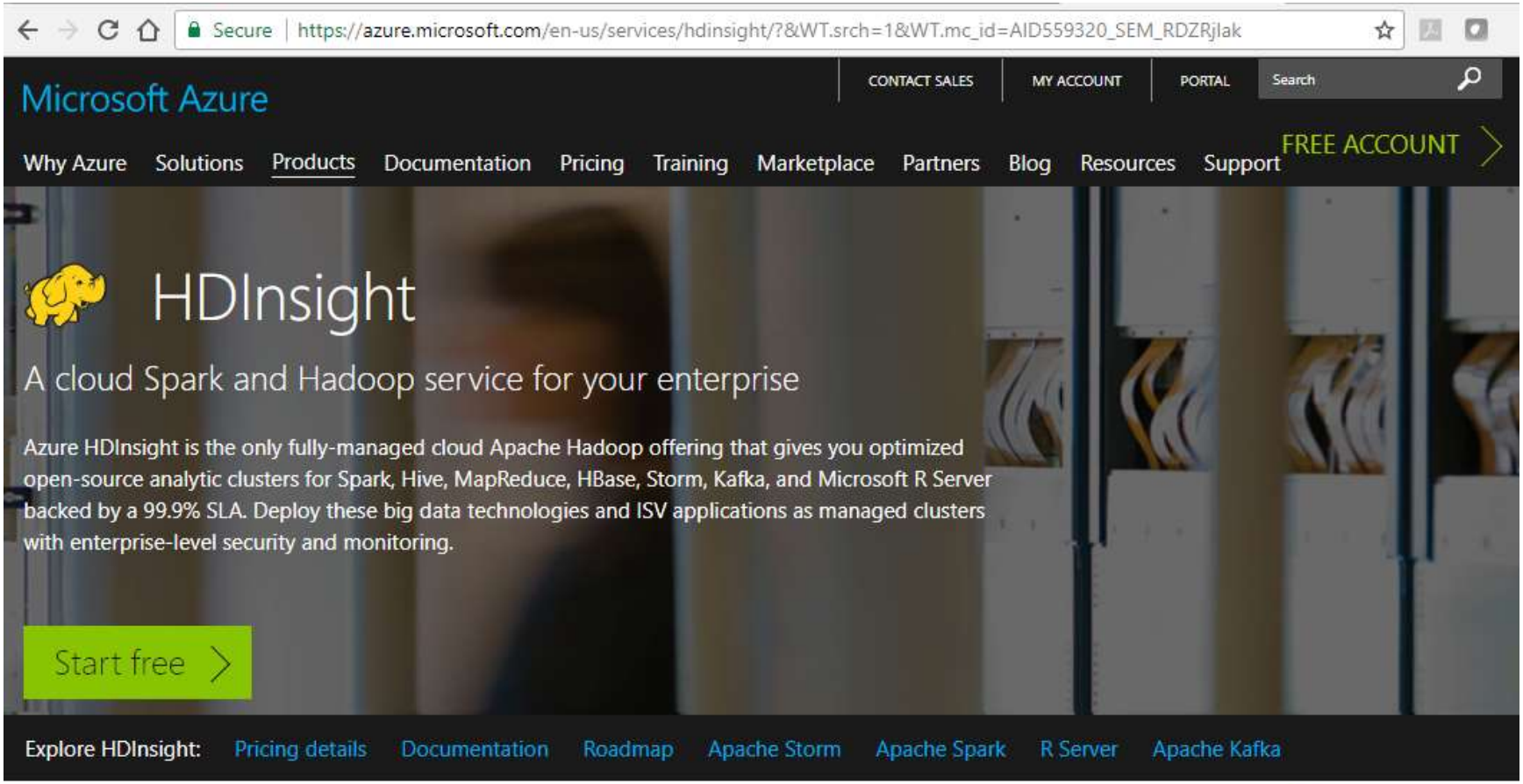


MapR

- If you are the Government of India and want to store and use biometric information of 1.2 billion people, you use MapR Hadoop



Microsoft Azure, HDInsight



Reliable open-source analytics with an industry-leading SLA

Spin up enterprise-grade, open-source cluster types with a 99.9% SLA and 24/7 support. Our SLA

IBM BigInsight

IBM's BigInsights: Smart Analytics for Big Data

References

- Hadoop the Definitive Guide, 3rd Edition, by Tom White, O'Reilly 2012
- Hadoop in Action, by Chuck Lam, Manning 2011
- Hadoop in Practice, by Alex Holmes, Manning 2012
- Cloudera, CDH5 Quick Start Guide

Lecture 03

Introduction to Spark

Zoran B. Djordjević

Reference

- These slides follow to a good measure the book

  "*Learning Spark*" by

  Holden Karau, Andy Konwinski, Patrick Wendell & Mathei Zaharia,
  O'Reilly 2015

Spark

- Spark is yet another one greatest thing ever invented.

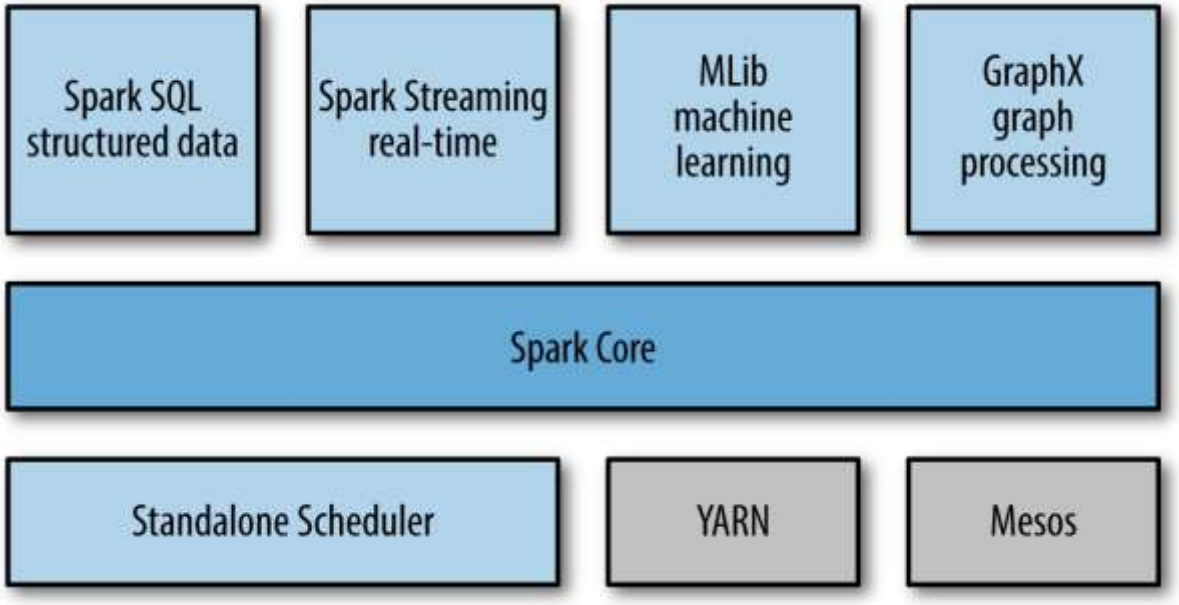
- Spark extends MapReduce model to efficiently support more types of computations, including interactive queries and stream processing.
- Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming.
- By supporting these workloads in the same engine, Spark makes it easy and inexpensive to *combine* different processing types.
- Spark is designed to be highly accessible, offering simple APIs in Python, Java, Scala, R and SQL, and rich built-in libraries.
- Spark integrates closely with other Big Data tools. Spark can run in Hadoop clusters and access any Hadoop data source, including NoSQL DBs like Cassandra.

What is Spark

- **Spark is** an *open-source* software solution that performs rapid calculations on *in-memory distributed datasets.*
- In-memory distributed datasets are referred to as RDDs
- *RDDs are Resilient Distributed Datasets*
  - RDD is the key Spark concept and the basis for what Spark does
  - RDD is a distributed collections of objects that can be cached in memory across cluster and can be manipulated in parallel.
  - RDD could be automatically recomputed on failure
  - RDD is resilient – can be recreated on the fly from known state
  - Immutable – already defined RDDs can be used as a basis to generate derivative RDDs but are never mutated
  - Distributed – the dataset is often partitioned across multiple nodes for increased scalability and parallelism

What is Spark

- Spark is a fast *general* processing engine for large scale data processing
- Spark is designed for iterative computations and interactive data mining.
- Spark supports use of well known languages: Scala, Python, Java and R
- With Spark streaming the same code could be used on data at rest and on data in motion
- Spark has several key modules:



Key Modules

- **Spark Core** contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and others.
- Spark Core is the home to the API that defines *resilient distributed datasets* (RDDs), which are Spark's main programming abstraction.
- RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. In recent releases less emphasis is placed on RDDs and more and DataFrames and Datasets.
- **Spark SQL** is Spark's package for working with structured data. I
- Spark SQL allows querying data via SQL as well as the Apache Hive variant of SQL—Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet, and JSON.
- Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs and DataFrames in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics.

Key Modules

- **Spark Streaming** is a Spark component that enables processing of live streams of data. Data streams include log files of web servers, or queues of messages.
- Spark Streaming API for manipulating data streams closely matches the Spark Core's RDD API, making it easy to move between apps that manipulate data in memory, on disk, or arriving in real time.
- Spark Streaming provides the same degree of fault tolerance, throughput, and scalability as Spark Core.
- **MLlib** is a library containing common machine learning (ML) functionality.
- MLlib provides multiple types of machine learning algorithms, including classification, regression, clustering, and collaborative filtering, as well as supporting functionality such as model evaluation and data import.
- MLlib provides some lower-level ML primitives, including a generic gradient descent optimization algorithm.
- All of ML methods are designed to scale out across a cluster.

Key Modules

- **GraphX** is a library for manipulating graphs (e.g., social network's graphs) and performing graph-parallel computations.
- GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge.
- GraphX also provides various operators for manipulating graphs (e.g., subgraph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting).

- **Cluster Managers** allow Spark to efficiently scale up from one to many thousands of compute nodes.
- Spark can run over a variety of *cluster managers*, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler.

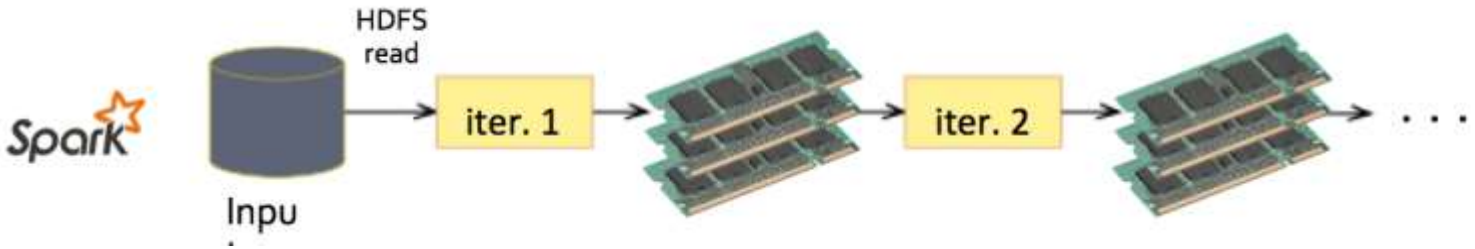
How does Spark Work?

- **RDD**
  - Your data is loaded in parallel into structured collections
- **Actions**
  - Manipulate the state of the working model by forming new RDDs and performing calculations upon them
- **Persistence**
  - Long-term storage of an RDD's state
- **Spark Application is a definition in code of**
  - RDD creation
  - Actions
  - Persistence
- Spark Application results in the creation of a DAG (Directed Acyclic Graph)
- Each DAG is compiled into stages
- Each Stage is executed as a series of Tasks
- Each Task operates in parallel on assigned partitions
- It all starts with the SparkContext 'sc'

Spark vs. Map Reduce

- Map Reduce places every result on the disk

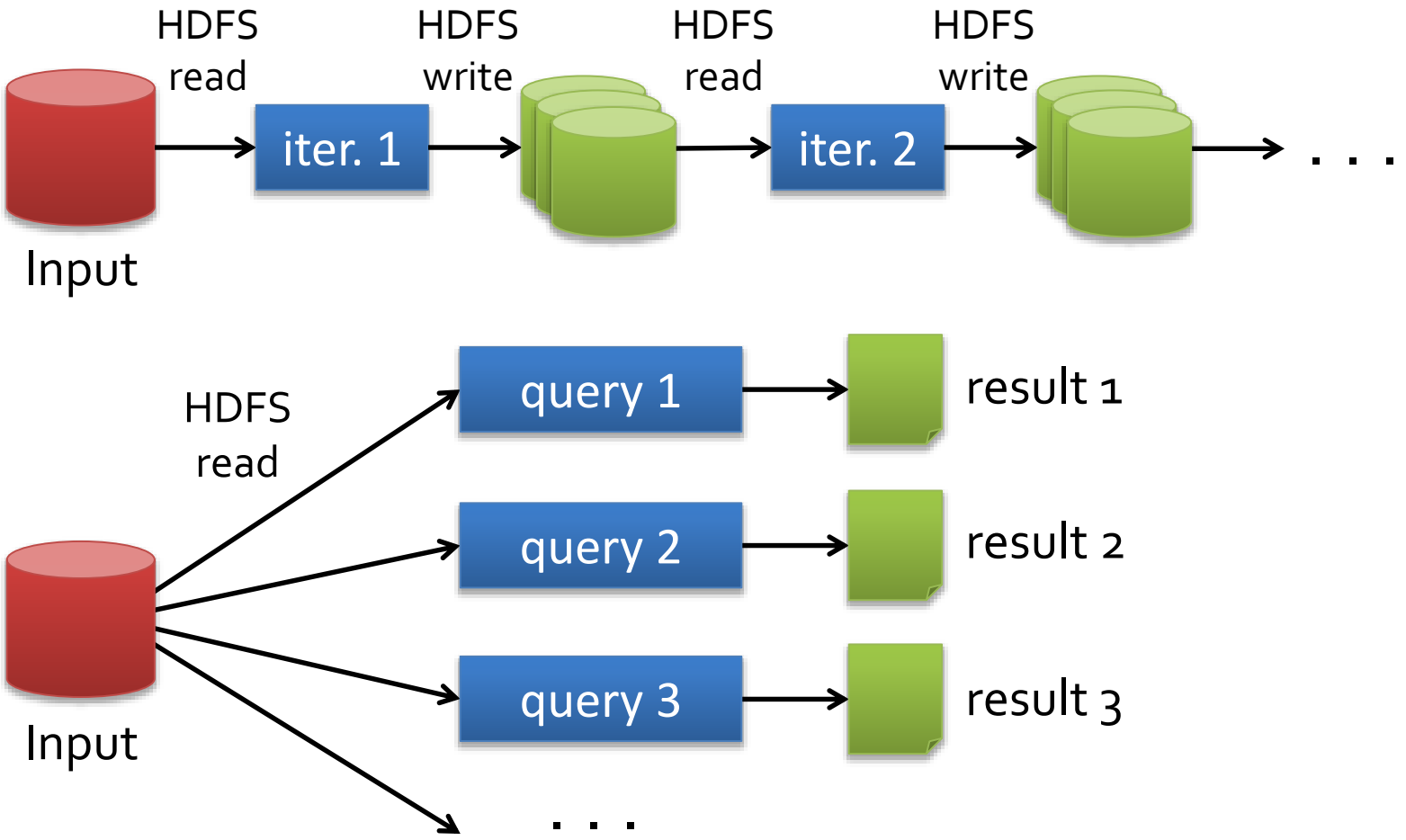
- Sp



- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - More **complex**, multi-pass analytics (e.g. ML, graph)
  - More **interactive** ad-hoc queries
  - More **real-time** stream processing
- All need faster **data sharing** across parallel jobs

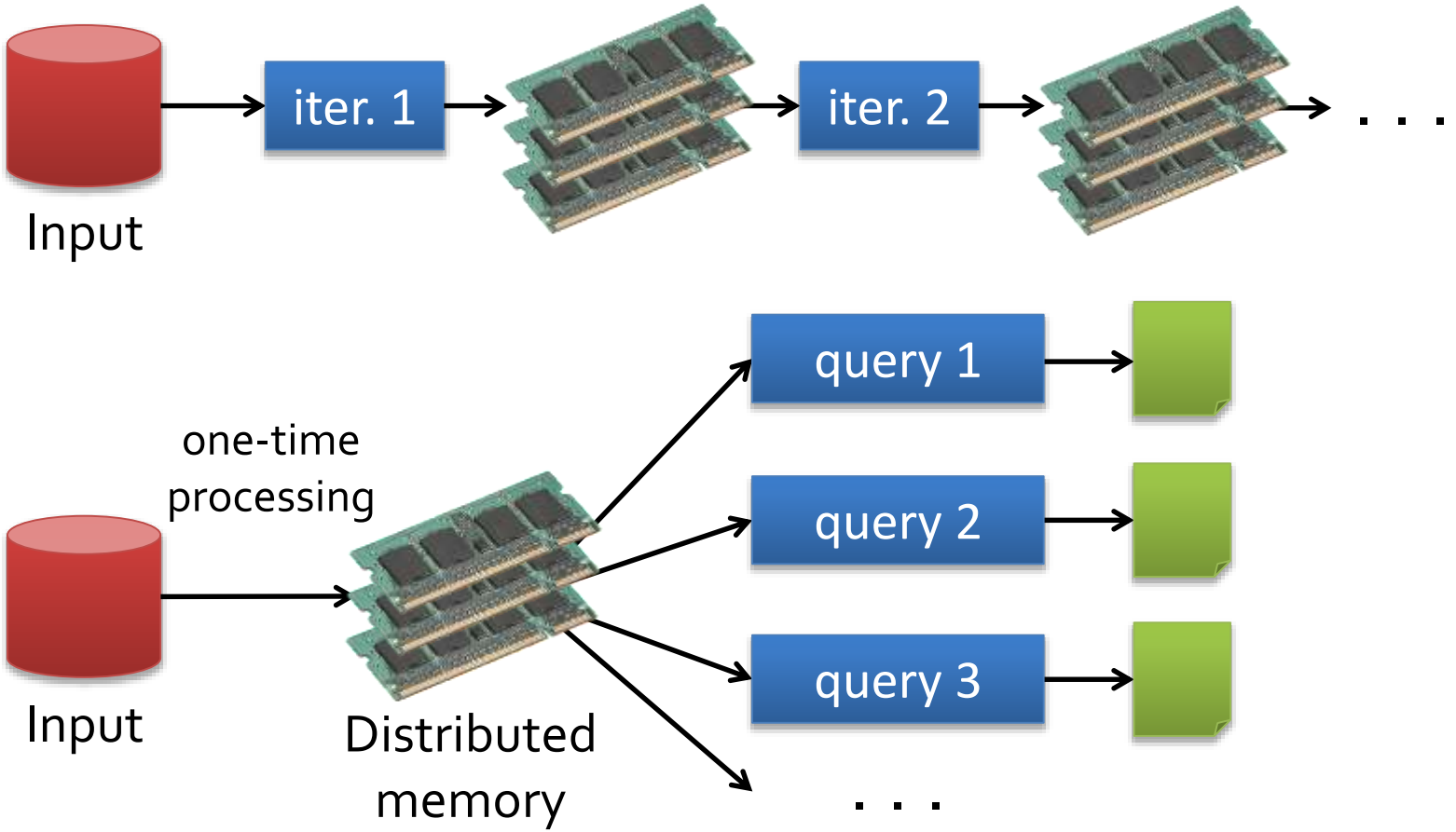
Data Sharing in Map Reduce

- Map Reduce is slow due to replication, serialization and disk IO



Data Sharing in Spark

- Distributed memory is 10-100× faster than network and disk

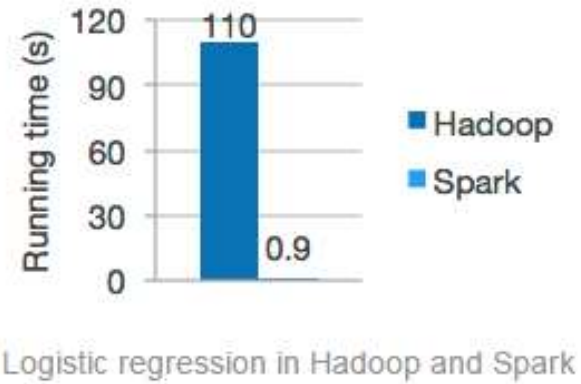


Spark vs. Map Reduce

- Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing.
- It is said that inventors of Spark noticed that Hadoop (MapReduce) was inefficient for the iterative workflows and they extended Hadoop architecture to make it more efficient.
- Speed is important in processing large datasets, as it means the difference between exploring data interactively and waiting minutes or hours.
- Spark's speed comes from its ability to run computations in memory. Spark appears to be more efficient than MapReduce for complex applications running on disk.
- Spark retains the attractive properties of MapReduce:
  - fault tolerance, data locality, scalability

Spark vs. Hadoop

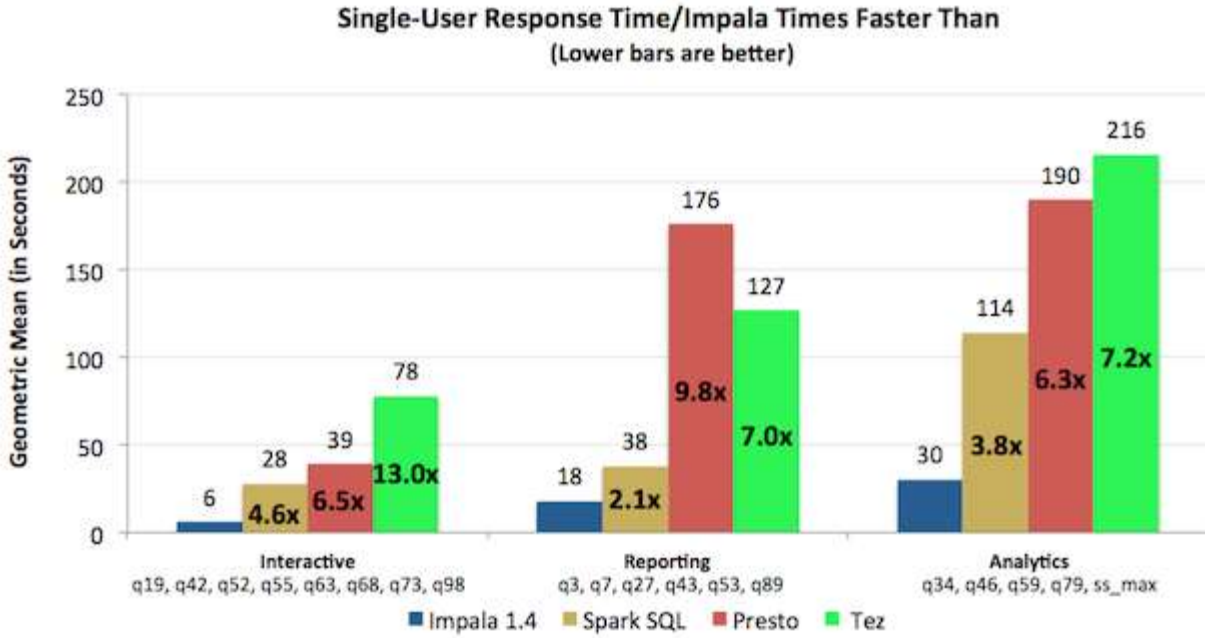
- You will frequently see diagrams like this:



Logistic regression in Hadoop and Spark

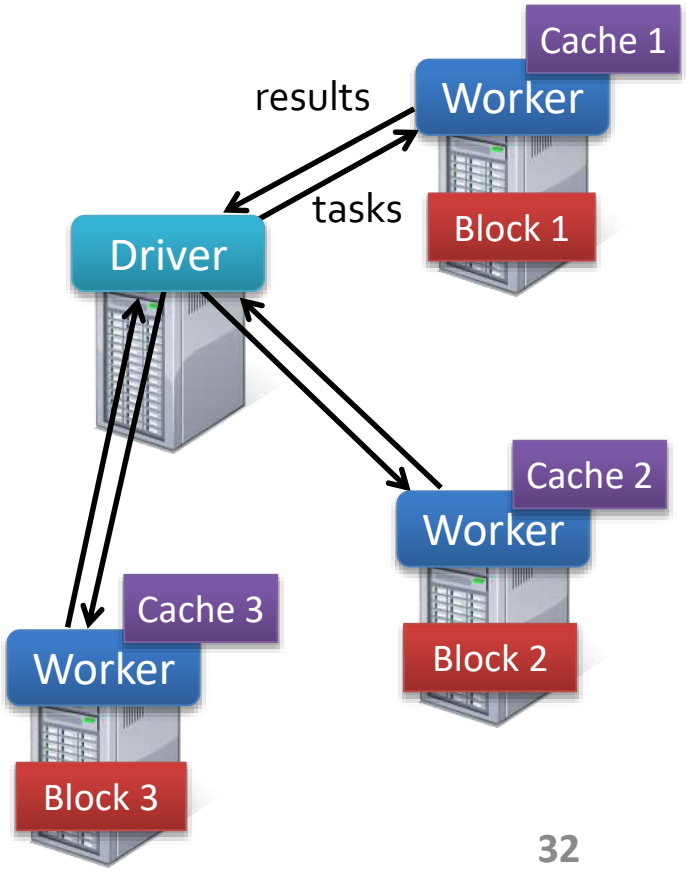
- Do not take this diagrams as a sign of inferiority of people who developed Hadoop. Hadoop people will show you the results below which compare Impala, an SQL engine they developed and Spark. Impala is 4 or 6 X faster,….

- Engineers who developed Hadoop are learning from Spark just like Spark developers learned from Hadoop.
- Cycles keep repeating



Distributed Memory Processing

- Just like MapReduce (Hadoop), Spark has a central controller, called Driver (App Master) which distributes tasks to Workers.
- Data (RDD) is split among memories of many workers.
- Data is originally sourced from the disk (HDFS, S3, regular File System). During processing is shared only between memories (if possible).
- Driver, if it detects that a worker is slacking or not working at all could make the worker redo its work or could move processing to another worker.
- Spark is fault tolerant just like Hadoop.



Installation of Spark 2.2

- Spark appears to be developed and run on Linux systems.
- From Harvard's Web Store (https://e5.onthehub.com/WebStore/Welcome.aspx?ws=4185a0dc-d0d1-e511-9416-b8ca3a5db7a1&vsro=8) download and install VMWare Workstation 12 or VMWare Fusion.
- From CentOS.org get the latest (7.4) version of that OS. Choose Everything ISO.
- Create a CentOS Virtual Machine. Besides the `root` user create one more user with administrative (`sudo`) privileges. It is convenient if that user and the root have the same password. Keep it that way, for now.
- To see which CentOS release you have, type:

```
$ cat /etc/centos-release
7.4
```

Java

- For Spark 2.2 to run you must have Java 1.8+ installed. On the command prompt of CentOS 7.4 OS, type:

```
$ java –version
1.8.0_131       # This is not the very latest but is good
```

To find where Java lives, type:

```
$ whereis java
/usr/lib/jvm ….
ls /usr/lib/jvm
java-1.7.0-openjdk-1.7.0.141-2.6.10.5.el7.x86_64
java-1.8.0-openjdk-1.8.0.131-11.b12.el7.x86_64
```

- In file `/etc/bashrc` or in file `.bash_profile` in your home directory add lines:

```
JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.131-
11.b12.el7.x86_64/jre
export $JAVA_HOME

PATH=$PATH:$JAVA_HOME/bin
export $PATH
```

Python

- Spark 2.2 needs Python 2.7+ or 3.4+. On the command prompt, type:

```
$ python
2.7.6   # this is good enough
```

- You need pip utility, if it is not already there:

```
sudo yum install python-pip
sudo pip install --upgrade pip
sudo yum install python-wheel
sudo pip install pyspark
```

Scala

- Spark 2.2 needs Scala 2.11+
- Do the following:

```
$ sudo curl https://bintray.com/sbt/rpm/rpm >
/etc/yum.repos.d/bintray-sbt-rpm.repo
$ sudo yum install sbt
$ sbt
```

Scala Programming Language (from Wikipedia)

- **Scala** (/ˈskɑːlə/ ***SKAH-lə***) is an object-functional programming language for general software applications. Scala has full support for ***functional programming*** and a very strong static type system. This allows programs written in Scala to be very concise and thus smaller in size than other general-purpose programming languages. Many of Scala's design decisions were inspired by criticism of the shortcomings of Java.
- Scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a Java virtual machine. Java libraries may be used directly in Scala code and vice versa. Like Java, Scala is object-oriented, and uses a curly-brace syntax reminiscent of the C programming language. Unlike Java, Scala has many features of functional programming languages like Scheme, Standard ML and Haskel, including currying, type inference, immutability, lazy evaluation, and pattern matching. It also has an advanced type system supporting algebraic data types, covariance and contravariance, higher-order types, and anonymous types. Other features of Scala not present in Java include operator overloading, optional parameters, named parameters, raw strings, and no checked exceptions.
- *Functional programming* is a subset of declarative programming. Programs written using this paradigm use functions, blocks of code intended to behave like mathematical functions. Functional languages discourage changes in the value of variables through assignment, making a great deal of use of recursion instead.

Install Spark 2.2

- From the site: spark.apache.org select:

- *Note: Starting version 2.0, Spark is built with Scala 2.11 by default. Scala 2.10 users should download the Spark source package and build with Scala 2.10 support.*

```
$ cd ~
$ wget https://d3kbcqa49mib13.cloudfront.net/spark-2.2.0-bin-hadoop2.7.tgz
```

- Unpack Spark in the /opt directory

```
$ sudo tar zxvf spark-2.2.0-bin-hadoop2.7.tgz -C /opt
```

- Create a symbolic link to make it easier to access

```
$ sudo ln -fs spark-2.2.0-bin-hadoop2.7 /opt/spark
```

Setup the Environment

- To complete your installation, set the SPARK_HOME environment variable so it takes effect when you login to your VM.
- Insert these lines into your `~/.bash_profile:`

```
export SPARK_HOME=/opt/spark
PATH=$PATH:$SPARK_HOME/bin
export PATH
```

- Then exit the text editor and return to the command line.
- You need to reload the environment variables (or logout and login again) so they take effect.

```
$ source ~/.bash_profile
```

- Confirm that spark-submit is now in the PATH.

```
$ spark-submit --version
```

- # (Should display a version number)

Spark Shells

- Spark comes with interactive shells that enable ad hoc data analysis.
- Unlike most other shells, which let you manipulate data using the disk and memory on a single machine, Spark's shells allow interaction with data distributed on disks or in memory across many machines.
- Spark can load data into memory on the worker nodes, and distributed computations, even ones that process large volumes of data across many machines, can run in a few seconds. This makes iterative, ad hoc, and exploratory analysis commonly done in shells a good fit for Spark.
- Spark provides both (and only) Python and Scala shells that have been augmented to support access to a cluster of machines.
- In these lecture notes, we will use Python shell, only.
- Python shell opens with `pyspark` command.
- Scala shell is very similar and opens with `spark-shell` command.
- You can program Spark from R using package `SparkR`

Shell verbosity and `log4j.properties` file

- The output is long and annoying. It would have been even longer had we not created `log4j.properties` file in the directory `$SPARK_HOME/conf`.
- You create that file by copying provided file `log4j.properties.template` and by changing line

```
log4j.rootCategory=INFO, console
```

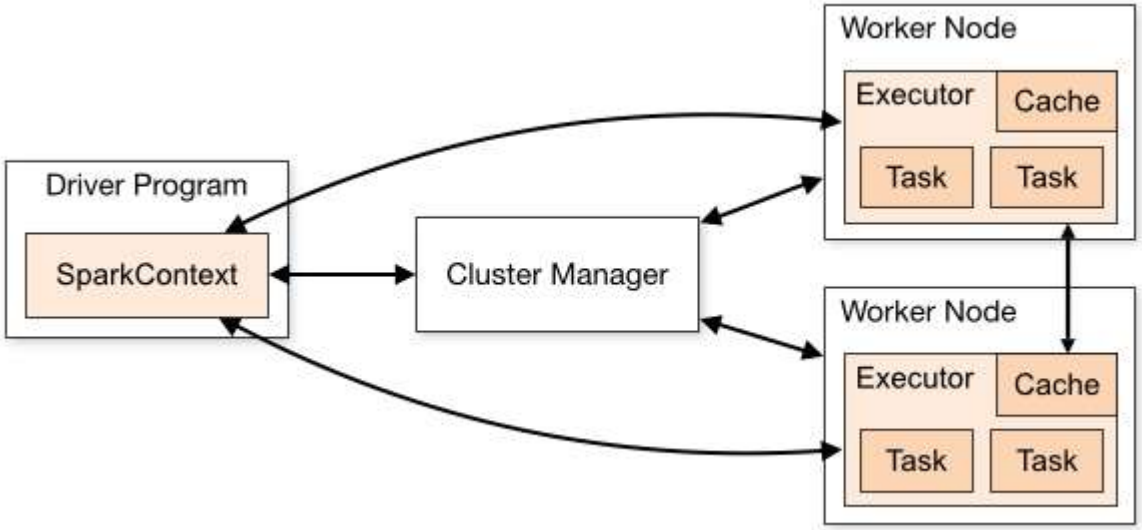
- to read:

```
log4j.rootCategory=ERROR, console
```

- That lowered the logging level so that we see only the ERROR messages, and above.
- You might want to replace all INFO levels with `WARN`, which is more verbose that ERROR but less than INFO. You might also want to replace all WORN levels with ERROR. Adjust those levels as you find convenient.pyspark

Cluster vs. Standalone

- Spark applications run as independent sets of processes on a cluster, coordinated by the `SparkContext` object in your main program (called the driver program).
- Spark clusters could have many thousands of nodes. Initially, we will run a single node standalone "cluster".
- and need to be managed by special resource (cluster) managers such as
- Specifically, to run on a cluster, the SparkContext connects to a cluster manager, such as Spark's own standalone cluster manager, YARN or Mesos. Once connected, Spark acquires *executors* on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends *tasks* to the executors to run.



Cluster Managers

Spark currently supports three cluster managers:

- Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications.
- Hadoop YARN – the resource manager in Hadoop 2.
- Kubernetes (experimental) – In addition to the above, there is experimental support for Kubernetes. Kubernetes is an open-source platform for providing container-centric infrastructure.

- Applications can be submitted to a cluster of any type using the `spark-submit` script.
- Each driver program has a web UI, typically on port 4040, that displays information about running tasks, executors, and storage usage. Simply go to http://<driver-node>:4040 in a web browser to access this UI.

Start your local Cluster of 1

- To start your local cluster with the single machine go to `/opt/spark/sbin` and type:

```
$ sudo ./start-master.sh
```

- The `.` and `/`, `i.e. ./` are significant.

pyspark

- If you type `pyspark` on the Linux command prompt, you will see the following:

```
[centos@localhost ~]$ pyspark
Python 2.7.5 (default, Aug  4 2017, 00:39:18)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
17/09/16 10:55:40 WARN NativeCodeLoader: Unable to load native-hadoop library for
your platform... using builtin-java classes where applicable
17/09/16 10:55:40 WARN Utils: Your hostname, localhost.localdomain resolves to a
loopback address: 127.0.0.1; using 192.168.135.128 instead (on interface ens33)
17/09/16 10:55:40 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
17/09/16 10:55:49 WARN ObjectStore: Failed to get database global_temp, returning
NoSuchObjectException
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Python version 2.7.5 (default, Aug  4 2017 00:39:18)
SparkSession available as 'spark'.
>>>
>>> quit()
$
```

spark-shell

- If your master is started type on the command prompt:

```
$ spark-shell
[centos@localhost ~]$ spark-shell
Spark context Web UI available at http://192.168.135.128:4040
Spark context available as 'sc' (master = local[*], app id = local-
1505573432881).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala> :quit
$
```

Load Data (RDD), Spark 1.6

- In Spark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster.
- These collections are called *resilient distributed datasets*, or RDDs.
- When we load some data, i.e. a file into a shell variable, we are creating an RDD, like

```
>>> lines = sc.textFile("/home/centos/ulysses10.txt")
>>> lines.count()
32742
```

- `ulysses10.txt` is a text file residing in the home directory of user `centos`.
- What we've just done is create and populate an RDD named `lines` using a mysterious object "`sc`" and its method `textFile()`. We populated lines2 = that RDD with data in file `ulysses10.txt`.
- "`sc`" stands for an implicit `SparkContext`.
- `SparkContext` communicates with the execution environment.
- It appears that RDD-s are also (Object Oriented) objects and have methods, such as `count()` which gave use the exact number of lines in file `ulysses10.txt`.
- We could have run the above load process using command:

```
lines = sc.textFile("file:///home/centos/ulysses10.txt")
```

- Emphasizing that the file resides in a regular file system

Load Data (Dataset), Spark 2.2

- In Spark 2.2 RDDs are still around but you are advised to use a different type of data objects called Datasets. Also, in Spark 2.2 SparkContext (object "sc") is implicit and you rather use spark session object, denoted by "spark". Previous commands could read:

```
>>> lines = spark.read.text("/home/centos/ulysses10.txt")
>>> lines.count()
32742
```

- In `spark-shell` which lets you use Scala, you would use a very similar command:

```
scala> val textFile =
spark.read.textFile("file:///opt/spark/README.md")
textFile: org.apache.spark.sql.Dataset[String] = [value: string]
```

- Spark 2.x Dataset is strongly-typed like an RDD, but with richer optimizations under the hood. Dataset objects have better performance than RDDs.

Load Data (Dataset) from HDFS

- We could load the same data from the Hadoops Distributed File System (HDFS)
- We will learn how to use HDFS in the next lecture.
- If we happen to have the `ulysses10.txt` file in `centos` home directory in HDFS, we could do the following:

```
>>> blines = spark.read.text ("hdfs:///user/centos/ulysses10.txt")
>>> blines.count()
32742
>>> blines.first()
u'The Project Gutenberg EBook of Ulysses, by James Joyce'
```

- What we did above was create an RDD named `blines` and populate that RDD with data from HDFS resident file `ulysses10.txt`.
- We also see in action another method of RDD-s, `first()`, which tells us that the first line in RDD `blines` is some uninterested collection of characters (**`u''`**) .
- Please note that we are not terminating our commands with a semi-colon (";") or anything else aside from the carriage return. Savings on typing all those semi-colons is one of the greatest contributions of Python to the computer science.
- By the way, our commands are in Python

Load Data from the Cloud (AWS S3 Bucket)

- I uploaded the same `ulysses10.txt` file to the Amazon's AWS S3 bucket called `zoran001`.
- On my Linux box (Cloudera VM) I created two new environmental variables in file `.bash_profile`.

```
AWS_ACCESS_KEY_ID=ADTSDYSDUIOSIDOI and
AWS_SECRET_ACCESS_KEY=dsfsfuierfsdfuiorfarifaifaopopa
```

- I source the file:

```
$ source .bash_profile
```

- Then, after reopening Python Spark shell, I issued the command:

```
>>> s3lines = sc.textFile("s3n://zoran001/ulysses10.txt")
>>> s3lines.count()
32742
```

- We did not loose a single line of text while brining the text down from the Cloud.

RDD filter() Method

- RDD method `filter()` takes a function returning `True` or `False` and applies it to a sequence (list) and returns only those members of the sequence for which the function returned `True`.
- So, in the Python code :

```
heavens =lines.filter(lambda line: "Heaven" in line)
```

- Method `filter()` acts on the collection `lines`, and passes every element of that collection as the variable `line` as the argument to the anonymous function created using lambda construct. That anonymous function uses a simple regular expression to test whether string "`Heaven`" exists in variable `line`.
- If the regular expression returns `True` for a particular `line`, an element of collection `lines`, the anonymous function will return `True` and for that particular `line`, `filter()` will return/add variable `line` building up a new collection called `heavens`.
- You can accomplish the same in Java 1.7 and older with several lines of code. In Java 1.8 you have similarly efficient lambda constructs.

RDD `filter()` method

```
>>> lines = sc.textFile("/home/centos/ulysses10.txt")
heaven = lines.filter(lambda line: "heaven" in line)
>>> heaven.count()
50
```

- We see in action a method of RDD-s, `filter()`, which apparently let us inquire how many times is `heaven` mentioned in the Ulysses.
- Heaven is mentioned 52 time, i.e. some 0.15% of the time (> Bible)

Dataset filtering

- If we load data into a Dataset, like

```
>>> dset = spark.read.text("/home/centos/ulysses10.txt")
```

- and want to extract all lines with word `heaven` we would use slightly different syntax:

```
>>> dset.count()
32742
>>> heavens = dset.filter(dset.value.contains('heaven'))
>>> heavens.count()
47
```

Python `lambda` Syntax

- Python supports the creation of anonymous inline functions (i.e. functions that are not bound to a name) at runtime, using a construct called "lambda".
- The following code shows the difference between a normal function definition ("f") and a lambda function ("g"):

```
>>> def f(x): return x**2
>>> print f(8)
64
>>> g = lambda x: x**2
>>> print g(8)
64
```

- As you can see, `f()` and `g()` do exactly the same thing. The lambda definition does not include a "return" statement. The last expression is returned.
- You can put a lambda definition anywhere a function is expected, and you don't have to assign it to a variable at all.

Passing Function to Spark in Python

- We want to convince ourselves that rather than using lambda constructs we could define functions and then pass their names to Spark. For example:

```
>>> def hasLife(line):
. . .       return "life" in line    # At the beginning hit a tab
. . .
>>> lines = sc.textFile("file:///home/centos/ulysses10.txt")
>>> lifeLines = lines.filter(hasLife)
>>> lifeLines.count()
203
>>> print lifeLines.first()
u'whom Mulligan was one, and Arius, warring his life long upon
the'
>>>
```

- Function `hasLife()` returns `True` if the line of text in its argument contains string "`life`". We successfully passed that function's name to method `filter()`.
- Above, we have seen a few more crucial features of Python. Python accepted the second line of function definition only when we properly indented `'return "life"…` statement.
- Also, to terminate function definition, we had to hit Carriage Return (Enter) twice.
- Most importantly, there are no semicolons in sight.

Passing Function in Java

- In Java Spark API it is possible to pass functions as well. In that case, functions are defined as Java classes, implementing a Spark interface:

  `org.apache.spark.api.java.function.Function.`

- For example:

```
JavaRDD<String> lifeLines = lines.filter(
    new Function<String, Boolean>() {
     Boolean call(String line){return line.contains("life");}
    }
);
```

- Java 8 introduces shorthand syntax called *`lambdas`* that looks similar to Python.
- The above Java code in new lambda syntax would look like:

```
JavaRDD<String> lifeLines = lines.filter(line ->
        line.contains("life"));
```

- A lot of Spark's magic is in the fact that function-based operations like `filter()` *also* parallelize across the cluster. That is, Spark automatically takes your function (e.g., `line.contains("Python")`) and ships it to executor nodes.
- You write code in a single driver program and Spark automatically has parts of it running on multiple nodes

Spark Sources and Destinations of Data

- Spark supports a wide range of input and output sources, partly because it builds on the ecosystem made available by Hadoop.
- In particular, Spark can access data through the `InputFormat` and `OutputFormat` interfaces used by Hadoop MapReduce, which are available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.).
- For data stored in a local or distributed file system, such as NFS, HDFS, or Amazon S3, Spark can access a variety of file formats including `Text`, `JSON`, `SequenceFiles`, and Google's `Protocol Buffers`.
- The Spark SQL module, provides an efficient API for structured data sources, including JSON and Apache Hive.
- Spark could also use third-party libraries for connecting to Cassandra, HBase, Elasticsearch, and JDBC databases.
- We will analyze some of the above techniques a bit later.

File Formats

- Spark makes it very simple to load and save data in a large number of file formats. Spark transparently handles compressed formats based on the file extension.
- We can use both Hadoop's new and old file APIs for keyed (or paired) data. We can use those only with key/value data.

Format Name	Structured	Comments
Text File	No	Plain old text files. Records assumed to be one per line.
JSON	Semi	Common text-based format, semi-structured; most libraries require one record per line.
CSV	YES	Very common text-based format, often used with spreadsheet applications.
SequenceFile	YES	A common Hadoop file format used for key/value data
Protocol buffer	YES	A fast, space-efficient multi-language format
Object file	YES	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

Standalone Applications

- Spark can be linked into standalone applications in either Java, Scala, or Python. The main difference from using it in a shell is that you need to initialize your own `SparkContext`. After that, the API is the same.
- The process of linking to Spark varies by language. In Java and Scala, you give your application a Maven dependency on the spark-core artifact.
- Maven is a popular package management tool for Java-based languages that lets you link to libraries in public repositories. You can use Maven itself to build your project, or use other tools that can talk to the Maven repositories, including Scala's `sbt` tool or `Gradle`.
- Eclipse also allows you to directly add a Maven dependency to a project.
- In Python, you simply write applications as Python scripts, but you must run them using the `bin/spark-submit` script included in Spark. The `spark-submit` script includes the Spark Python dependencies.
- We simply run your script with the

```
$SPARK_HOME/bin/spark-submit  your_script.py
```

SparkSession in Spark 2, Scala

- In Spark 1.6, you have to create a `SparkConf` and `SparkContext` to interact with Spark, as shown here in Scala snippets:

```
//set up the spark configuration and create contexts
val sparkConf = new
SparkConf().setAppName("SparkSessionZipsExample").setMaster("local")
// your handle to SparkContext to access other context like SQLContext
val sc = new SparkContext(sparkConf).set("spark.some.config.option", "some-
value") val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

- In Spark 2.0 the same effect will be achieved through `SparkSession`, without explicitly creating `SparkConf, SparkContext or SQLContext`, as they're encapsulated within the `SparkSession`.
- Using a builder design pattern, we instantiate a `SparkSession` object along with its associated underlying contexts.

```
// Create a SparkSession. No need to create SparkContext
// You automatically get it as part of the SparkSession
val warehouseLocation = "file:${system:user.dir}/spark-warehouse"
val spark = SparkSession .builder() .appName("SparkSessionZipsExample")
.config("spark.sql.warehouse.dir", warehouseLocation) .enableHiveSupport()
.getOrCreate()
```

Scala Standalone App, Spark 2.x

```
/* SimpleApp.scala */
import org.apache.spark.sql.SparkSession
object SimpleApp {
  def main(args: Array[String]) {
    val logFile = "somefile.txt" // some file on your system
    val spark = SparkSession.builder.appName("Simple Application").getOrCreate()
    val logData = spark.read.textFile(logFile).cache()
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println(s"Lines with a: $numAs, Lines with b: $numBs")
    spark.stop()
  }
}
```

- Note that applications defines a `main()` method.
- This program just counts the number of lines containing ‘a’ and the number containing ‘b’ in your file.

Standalone Application in Python 1.6

- To create an application (Python script) we need to import some Python classes and create `SparkContext` object.
- The rest of the application is coded as if you are writing code in PySpark shell

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("MyApp")
sc = SparkContext(conf = conf)
lines = sc.textFile("ulysses10.txt")
lifeLines = lines.filter(lambda line: "life" in line)
print lifeLines.first()
```

- If we invoke the above with:

```
$ spark-submit my_script.py
```

- We get:

```
py4j.protocol.Py4JJavaError: An error occurred while calling
o25.partitions.
: org.apache.hadoop.mapred.InvalidInputException: Input path
does not exist: hdfs://localhost:8020/ulysses/4300.txt
```

Standalone App in Python, Spark 2.2

- Applications can be submitted to a cluster of any type using the `spark-submit` script.

```
from pyspark.sql import SparkSession

logFile = "somefile.txt"  # Should be some file on your system
spark =
SparkSession.builder().appName(appName).master(master).getOrCreate()
logData = spark.read.text(logFile).cache()

numAs = logData.filter(logData.value.contains('a')).count()
numBs = logData.filter(logData.value.contains('b')).count()

print("Lines with a: %i, lines with b: %i" % (numAs, numBs))

spark.stop()
```

- This program just counts the number of lines containing ‘a’ and the number containing ‘b’ in a text file.

```
$ spark-submit your_script.py
```