Introduction to Spark

Apache Hadoop Vs Spark

Dr. Zeenat Tariq

Outline

- Growth of big datasets
- Introduction to Apache Hadoop and Spark for developing applications
- Components of Hadoop, HDFS and MapReduce
- Working Example using Hadoop
- Shortcomings of MapReduce
- Introduction to Spark Programming
- Capabilities of Spark and the differences from a typical MapReduce solution
- MapReduce vs Spark Summary
- Conclusion

Growth of Big Datasets

- The Large Hadron Collider produces about 30 petabytes of data per year
- Facebook’s data is growing at 8 petabytes per month
- The New York stock exchange generates about 4 terabyte of data per day
- YouTube had around 80 petabytes of storage in 2012
- Internet Archive stores around 19 petabytes of data

What is Apache Hadoop

- Large scale open-source software framework
- Dedicated to scalable, distributed and data-intensive computing
- Handles thousands of nodes and petabytes of data
- Supports application under a free license
- Consists of
  - HDFS: Hadoop Distributed File System
  - MapReduce execution engine: Schedule tasks on HDFS

Hadoop Architecture

- Hadoop Distributed File System (HDFS)
  - Single name node, many data nodes
  - Files stored as large, fixed-size (e.g., 64MB) blocks
  - HDFS typically holds map input and reduce output
  - Data is distributed and replicated over multiple machines
- Hadoop MapReduce
  - Single master node, many worker nodes
  - Client submits a *job* to master node
  - Master splits each job into *tasks* (MapReduce), and assigns tasks to worker nodes

Hadoop HDFS

- Hadoop distributed File System (based on Google File System (GFS) paper, 2004)
  - Serves as the distributed file system for most tools in the Hadoop ecosystem
  - Scalability for large data sets
  - Reliability to cope with hardware failures
- HDFS good for:
  - Large files
  - Streaming data
- Not good for:
  - Lots of small files
  - Random access to files
  - Low latency access

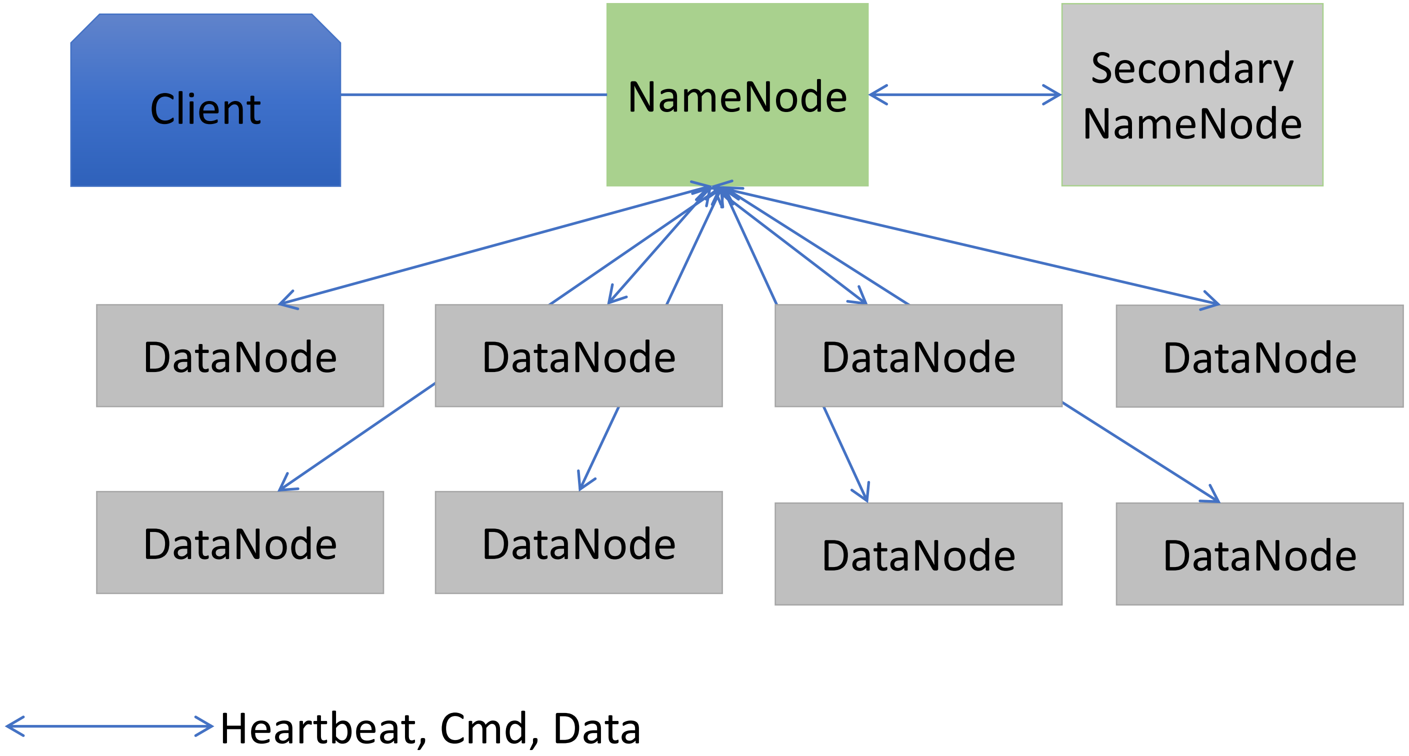
Single Hadoop cluster with 5000 servers and 250 petabytes of data

Design of Hadoop Distributed File System (HDFS)

- Master-Slave design
- Master Node
  - Single NameNode for managing metadata
- Slave Nodes
  - Multiple DataNodes for storing data
- Other
  - Secondary NameNode as a backup

HDFS Architecture

NameNode keeps the metadata, the name, location and directory
DataNode provide storage for blocks of data

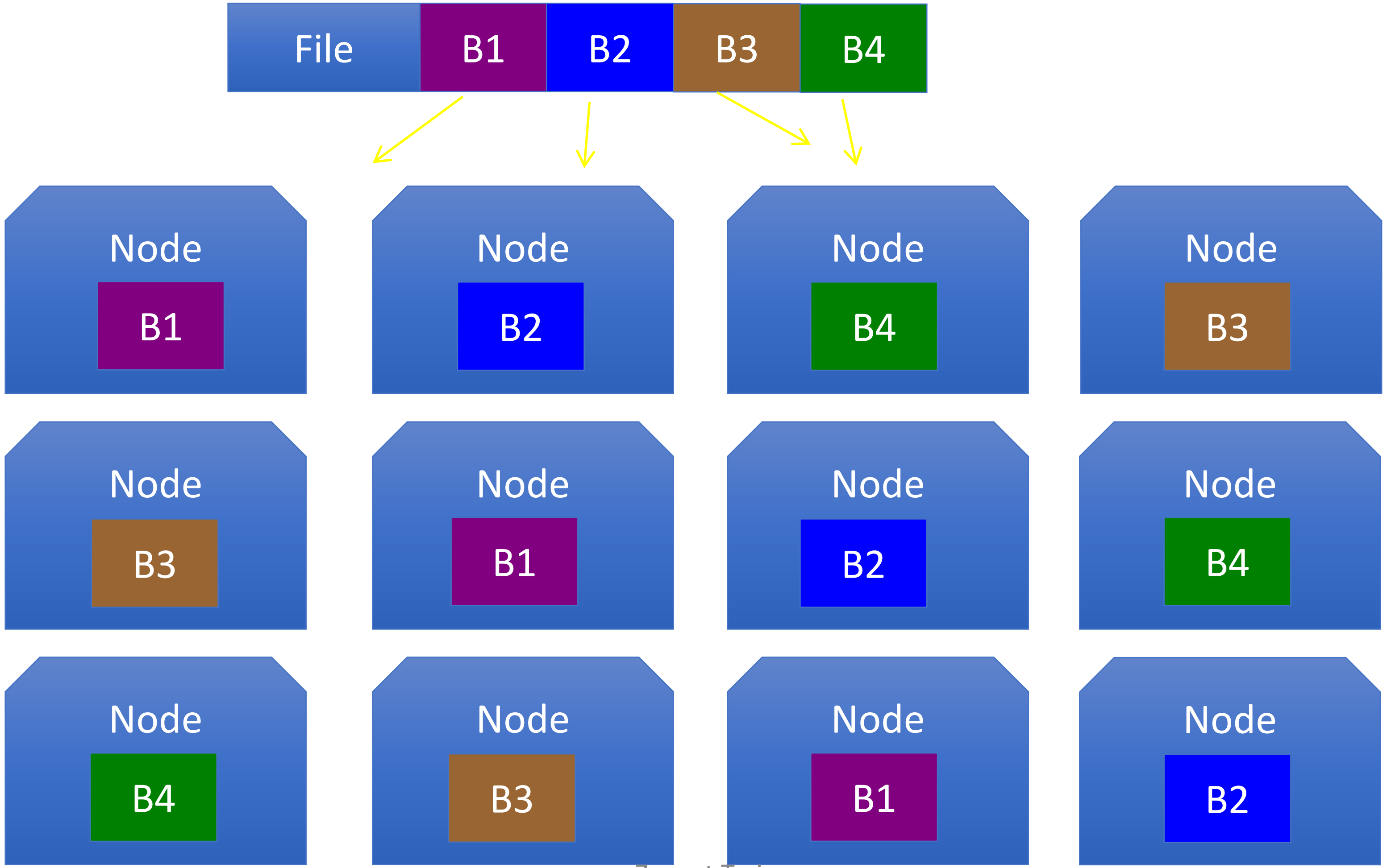


HDFS

- HDFS files are divided into blocks
  - It’s the basic unit of read/write
  - Default size is 64MB, could be larger (128MB)
  - Hence makes HDFS good for storing larger files
- HDFS blocks are replicated multiple times
  - One block stored at multiple location, also at different racks (usually 3 times)
  - This makes HDFS storage fault tolerant and faster to read

HDFS

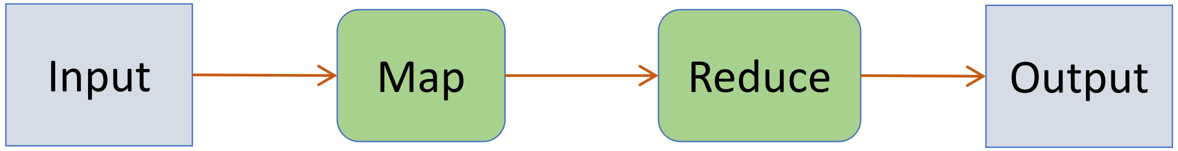
What happens; if node(s) fail?
Replication of Blocks for fault tolerance



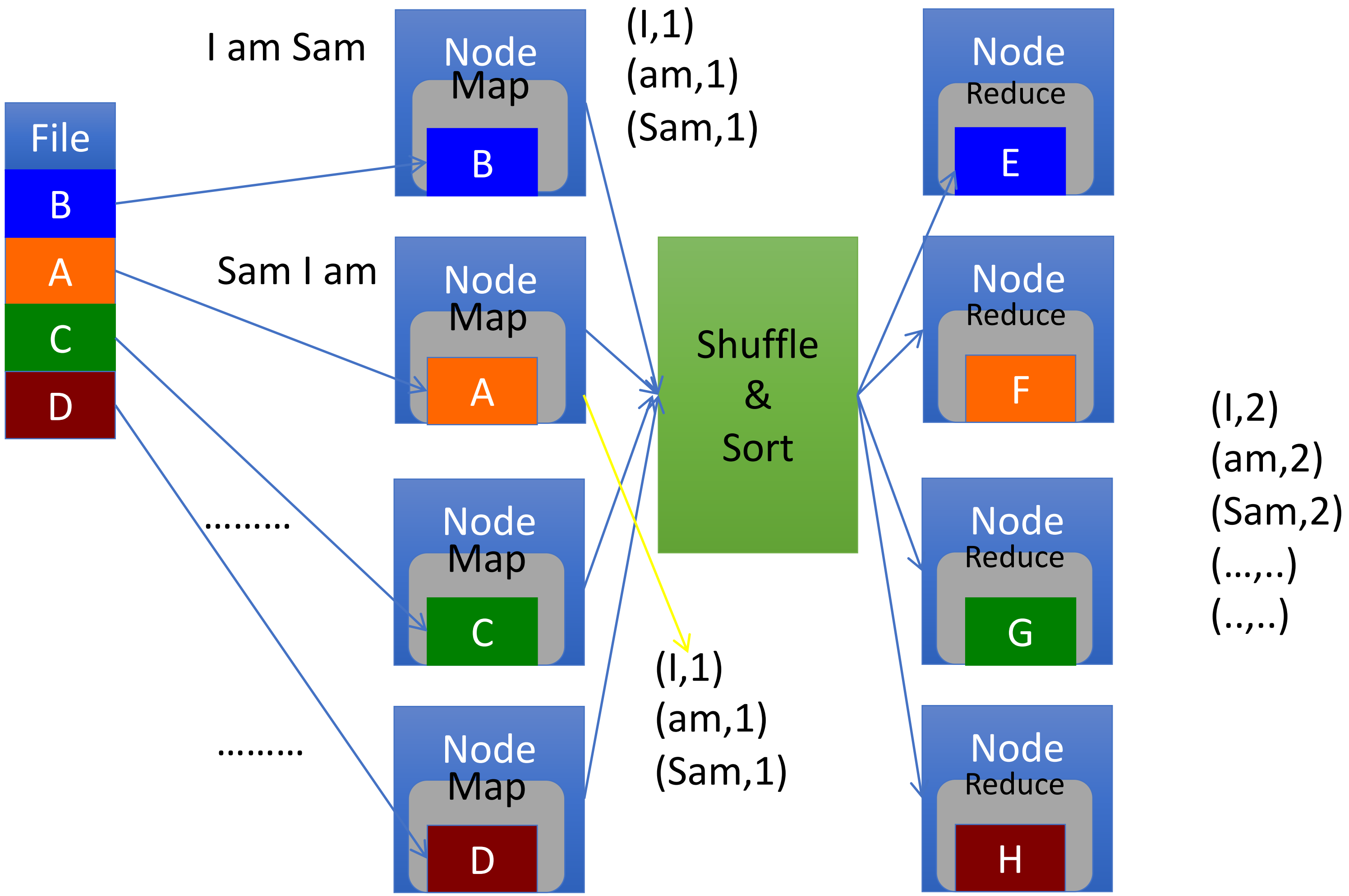
Map Reduce Paradigm

- Map and Reduce are based on functional programming

Map:	**Reduce:**
Apply a function to all the elements of List list1=[1,2,3,4,5]; square x = x * x list2=Map square(list1) print list2 -> [1,4,9,16,25]	Combine all the elements of list for a summary list1 = [1,2,3,4,5]; A = reduce (+) list1 Print A -> 15



MapReduce Word Count Example



Shortcoming of MapReduce

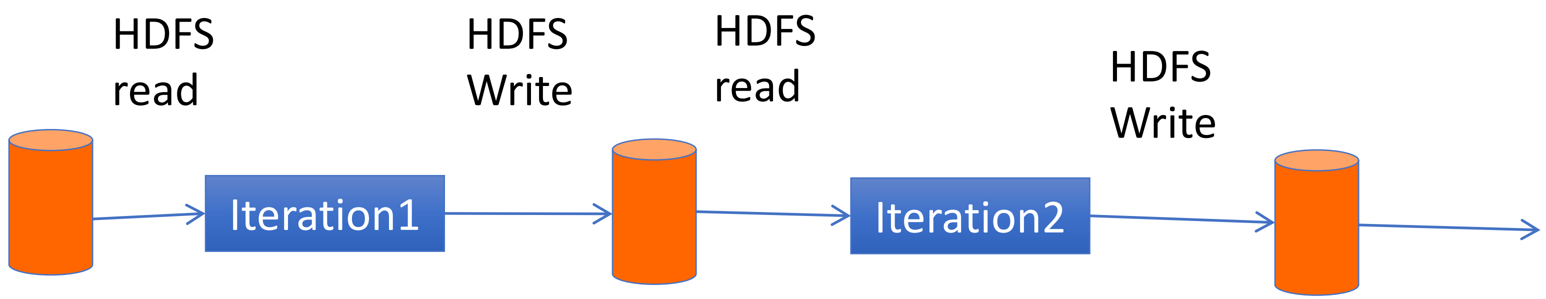
- Forces your data processing into Map and Reduce
  - Other workflows missing include join, filter, flatMap, groupByKey, union, intersection, …
- Based on “Acyclic Data Flow” from Disk to Disk (HDFS)
- Read and write to Disk before and after Map and Reduce (stateless machine)
  - Not efficient for iterative tasks, i.e., Machine Learning
- Only Java natively supported
  - Support for others languages needed
- Only for Batch processing
  - Interactivity, streaming data

One Solution is Apache Spark

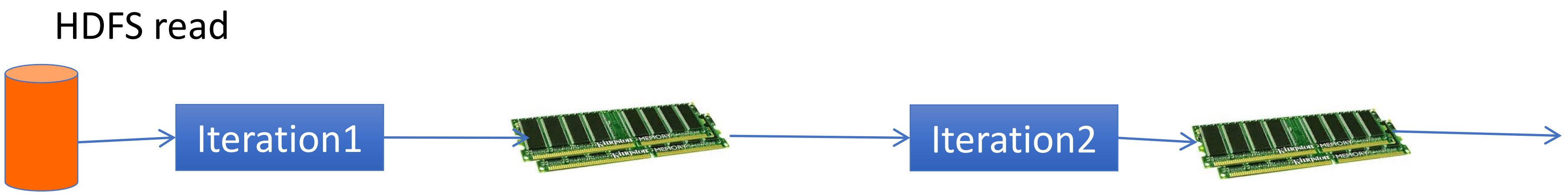
- A new general framework, which solves many of the short comings of MapReduce
- It capable of leveraging the Hadoop ecosystem, e.g. HDFS, YARN, HBase, S3, …
- Has many other workflows, i.e. join, filter, flatMapdistinct, groupByKey, reduceByKey, sortByKey, collect, count, first…
  - (around 30 efficient distributed operations)
- In-memory caching of data (for iterative, graph, and machine learning algorithms, etc.)
- Native Scala, Java, Python, and R support
- Supports interactive shells for exploratory data analysis
- Spark API is extremely simple to use
- Developed at AMPLab UC Berkeley, now by Databricks.com

Spark Uses Memory instead of Disk

Hadoop: Use Disk for Data Sharing



Spark: In-Memory Data Sharing



Sort competition

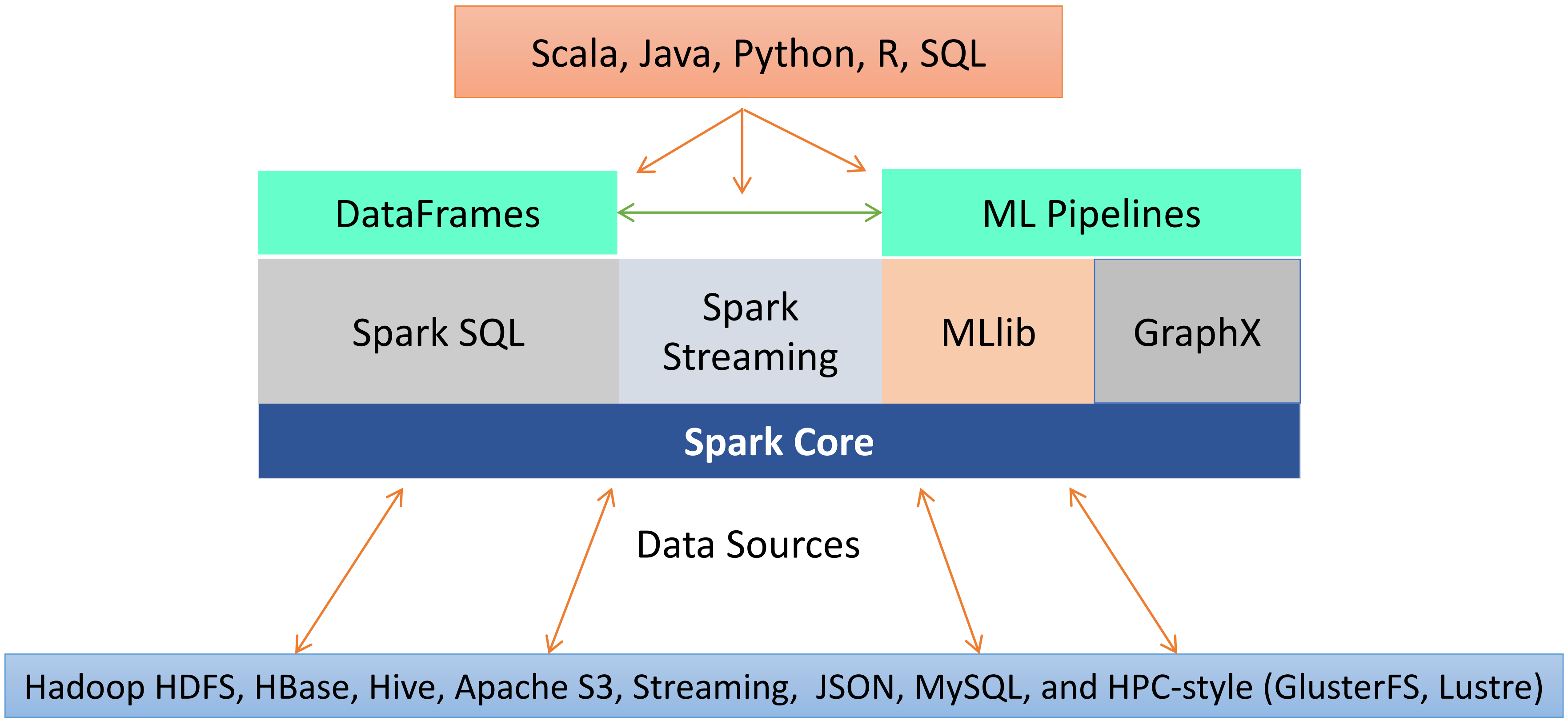
	Hadoop MR Record (2013)	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**

Spark, 3x faster with 1/10 the nodes

Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)
http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

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.



Spark Basics

Spark: Flexible, in-memory data processing framework written in Scala

Goals:

- Simplicity (Easier to use):
  - Rich APIs for Scala, Java, and Python
- Generality: APIs for different types of workloads
  - Batch, Streaming, Machine Learning, Graph
- Low Latency (Performance) : In-memory processing and caching
- Fault-tolerance: Faults shouldn’t be special case

Spark Fundamentals

Example of an application:

```
val sc = new SparkContext("spark://...", "MyJob", home,
   jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
   an RDD

errors.cache()

errors.count() // This is an action
```

- **Spark Context**
- **Resilient Distributed Data**
- **Transformations**
- **Actions**

Spark: Fundamentals

- Spark Context
- Resilient Distributed Datasets (RDDs)
- Transformations
- Actions

Spark Context

- Every Spark application requires a spark context: the main entry point to the Spark API
- Spark Shell provides a preconfigured Spark Context called “sc”

Python

```
Using Python version 2.7.8 (default, Aug 27 2015 05:23:36)
SparkContext available as sc, HiveContext available as sqlCtx.

>>> sc.appName
u'PySparkShell'
```

Scala

```
...
Spark context available as sc.
SQL context available as sqlContext.

scala> sc.appName
res0: String = Spark shell
```

Spark Fundamentals

Example of an application:

```
val sc = new SparkContext("spark://...", "MyJob", home,
   jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
   an RDD

errors.cache()

errors.count() // This is an action
```

- Spark Context
- **Resilient Distributed Data**
- Transformations
- Actions

Resilient Distributed Dataset

RDD (Resilient Distributed Dataset) is the fundamental unit of data in Spark: An *Immutable* collection of objects (or records, or elements) that can be operated on “in parallel” (spread across a cluster)

Resilient -- if data in memory is lost, it can be recreated

- Recover from node failures
- An RDD keeps its lineage information → it can be recreated from parent RDDs

Distributed -- processed across the cluster

- Each RDD is composed of one or more partitions → (more partitions – more parallelism)

Dataset -- initial data can come from a file or be created

RDDs

Key Idea: Write applications in terms of transformations on distributed datasets

- Collections of objects spread across a Memory caching layer(cluster) that stores data in a distributed, fault-tolerant cache
- Can fall back to disk when dataset does not fit in memory
- Built through parallel transformations (map, filter, group-by, join, etc)
- Automatically rebuilt on failure
- Controllable persistence (e.g., caching in RAM)

Creating a RDD

Three ways to create a RDD

- From a file or set of files
- From data in memory
- From another RDD

Example: A File-based RDD

```
> val mydata = sc.textFile("purplecow.txt")
…
15/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()

…
15/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```

File: purplecow.txt

```
I've never seen a purple
cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

RDD: mydata

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

Spark Fundamentals

Example of an application:

```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

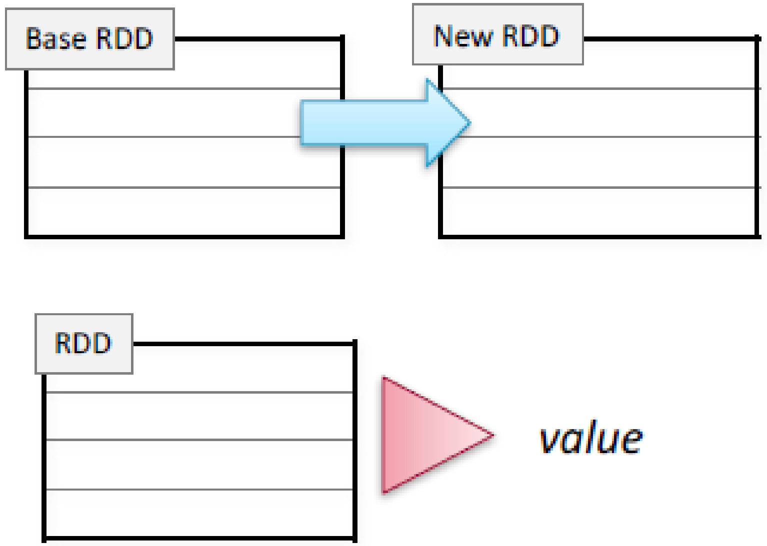
- Spark Context
- **Resilient Distributed Data**
- **Transformations**
- **Actions**

RDD Operations

Two types of operations

Transformations: Define a new RDD based on current RDD(s)

Actions: return values




```
val sc = new SparkContext("spark://...", "MyJob", home,
    jars)

val file = sc.textFile("hdfs://...") // This is an RDD

val errors = file.filter(_.contains("ERROR")) // This is
    an RDD

errors.cache()

errors.count() // This is an action
```

RDD Transformations

- Set of operations on a RDD that define how they should be transformed
- As in relational algebra, the application of a transformation to an RDD yields a new RDD (because RDD are immutable)
- Transformations are lazily evaluated, which allow for optimizations to take place before execution
- Examples: map(), filter(), groupByKey(), sortByKey(), etc.

RDD Actions

- Apply transformation chains on RDDs, eventually performing some additional operations (e.g., counting)
- Some actions only store data to an external data source (e.g. HDFS), others fetch data from the RDD (and its transformation chain) upon which the action is applied, and convey it to the driver
- Some common actions
  - count() – return the number of elements
  - take(*n*) – return an array of the first *n* elements
  - collect()– return an array of all elements
  - saveAsTextFile(*file*) – save to text file(s)

Lazy Execution of RDDs (1)

Data in RDDs is not processed until an action is performed

File: purplecow.txt

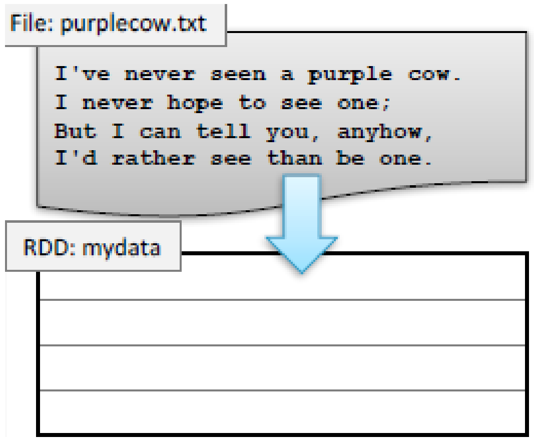
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

Lazy Execution of RDDs (2)

Data in RDDs is not processed until an action is performed

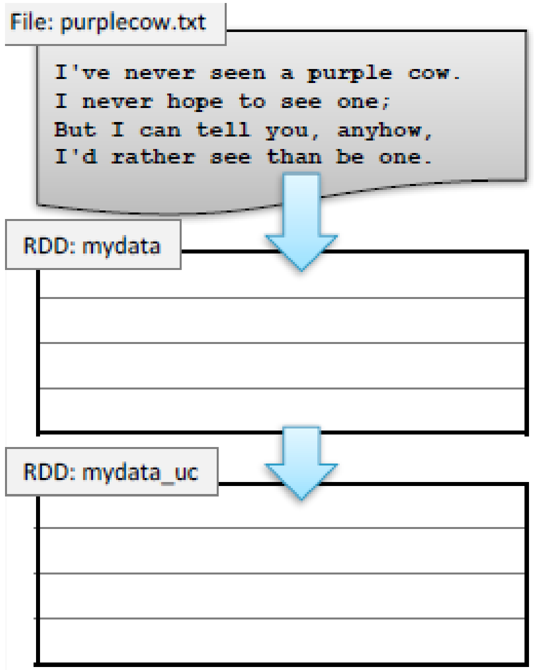
```
> val mydata = sc.textFile("purplecow.txt")
```



Lazy Execution of RDDs (3)

Data in RDDs is not processed until an action is performed

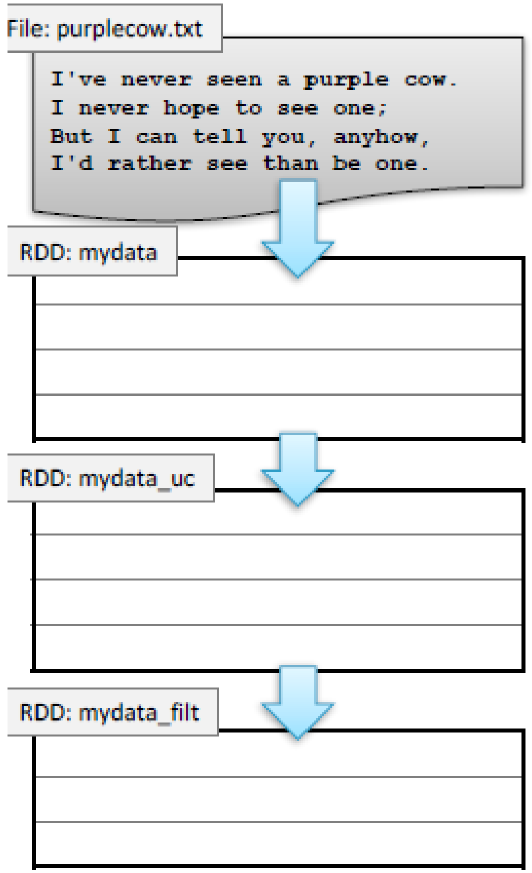
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



Lazy Execution of RDDs (4)

Data in RDDs is not processed until an action is performed

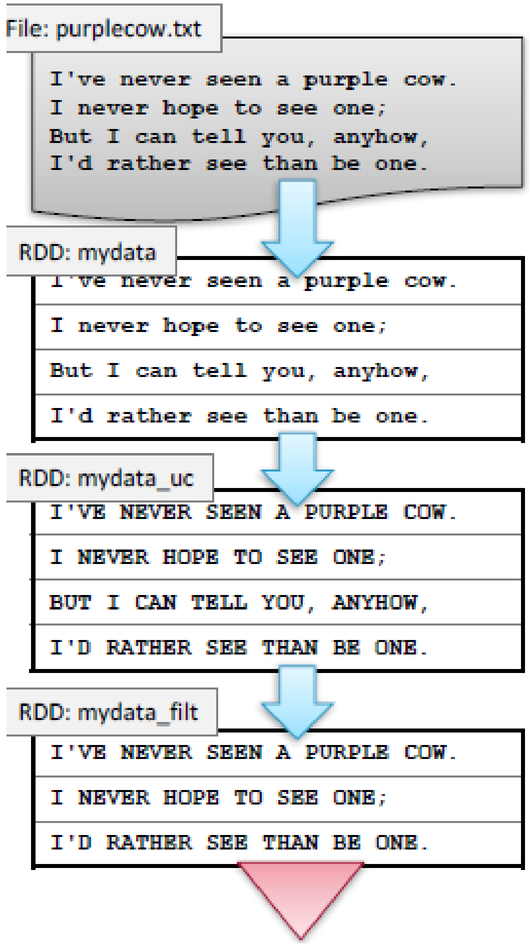
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy Execution of RDDs (5)

Data in RDDs is not processed until an action is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



Spark example (Scala)

```
// "sc" is a "Spark context" – this transforms the file into an RDD
val textFile = sc.textFile("README.md")
// Return number of items (lines) in this RDD; count() is an action
textFile.count()
// Demo filtering.  Filter is a tranform.  By itself this does no real work
val linesWithSpark = textFile.filter(line => line.contains("Spark"))
// Demo chaining – how many lines contain "Spark"?  count() is an action.
textFile.filter(line => line.contains("Spark")).count()
// Length of line with most words.  Reduce is an action.
textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
// Word count – traditional map-reduce.  collect() is an action
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
wordCounts.collect()
```

Source: https://spark.apache.org/docs/latest/quick-start.html

MapReduce vs Spark (Summary)

- Performance:
  - While Spark performs better when all the data fits in the main memory (especially on dedicated clusters), MapReduce is designed for data that doesn’t fit in the memory
- Ease of Use:
  - Spark is easier to use compared to Hadoop MapReduce as it comes with user-friendly APIs for Scala (its native language), Java, Python, and Spark SQL.
- Fault-tolerance:
  - Batch processing: Spark → HDFS replication
  - Stream processing: Spark RDDs replicated

MapReduce vs. Spark for Large Scale Data Analysis

- MapReduce and Spark are two very popular open-source cluster computing frameworks for large scale data analytics
- These frameworks hide the complexity of task parallelism and fault-tolerance, by exposing a simple programming API to users

Conclusion

- Hadoop (HDFS, MapReduce)
  - Provides an easy solution for processing of Big Data
  - Brings a paradigm shift in programming distributed system
- Spark
  - Has extended MapReduce for in memory computations
  - for streaming, interactive, iterative and machine learning tasks
- Changing the World
  - Made data processing cheaper and more efficient and scalable
  - Is the foundation of many other tools and software