Apache Hadoop and Spark

Outline

- Growth of big datasets
- Introduction to Apache Hadoop and Spark for developing applications
- Components of Hadoop, HDFS, MapReduce and HBase
- Capabilities of Spark and the differences from a typical MapReduce solution
- Some Spark use cases for data analysis

Growth of Big Datasets

- Internet/Online Data
  - Clicks
  - Searches
  - Server requests
  - Web logs
  - Cell phone logs
  - Mobile GPS locations
  - User generated content
  - Entertainment (YouTube, Netflix, Spotify, …)
- Healthcare and Scientific Computations
  - Genomics, medical images, healthcare data, billing data
- Graph data
  - Telecommunications network
  - Social networks (Facebook, Twitter, LinkedIn, …)
  - Computer networks
- Internet of Things
  - RFID
  - Sensors
- Financial data

Data

- 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, now around 10-15 exabuytes
- Internet Archive stores around 19 petabytes of data

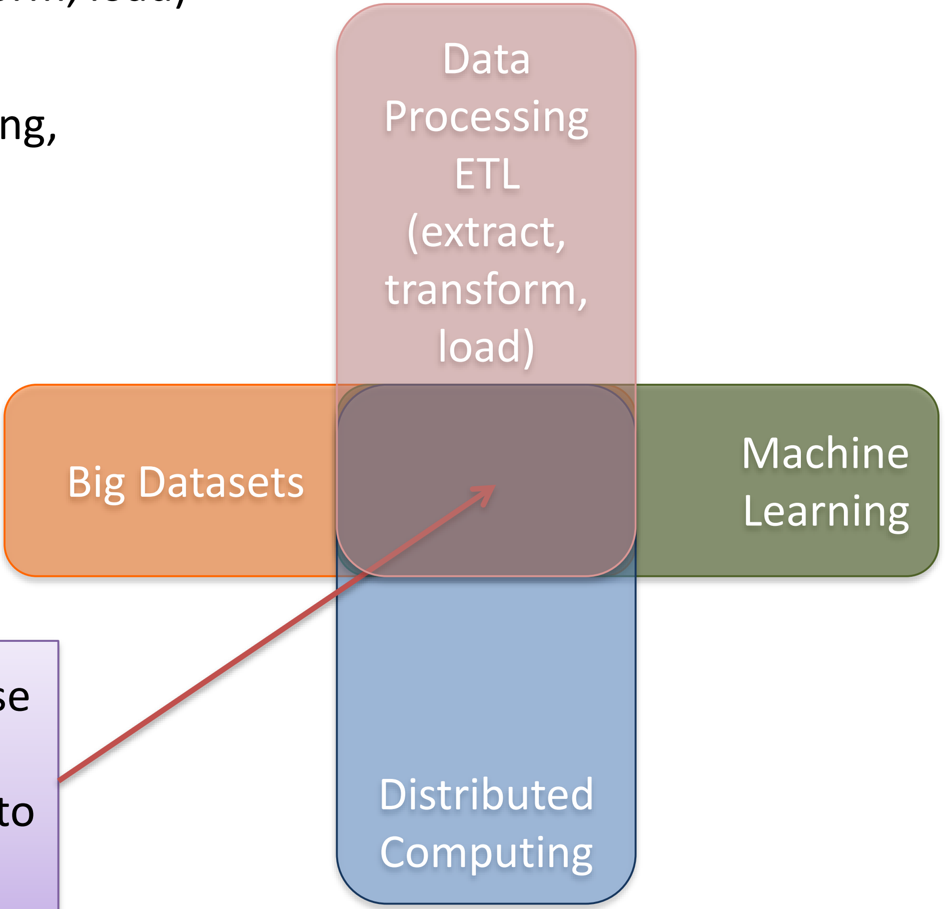
Cloud and Distributed Computing

- The second trend is pervasiveness of cloud based storage and computational resources
  - For processing of these big datasets
- Cloud characteristics
  - Provide a scalable standard environment
  - On-demand computing
  - Pay as you need
  - Dynamically scalable
  - Cheaper

Data Processing and Machine learning Methods

- Data processing (third trend)
  - Traditional ETL (extract, transform, load)
  - Data Stores (HBase, ……..)
  - Tools for processing of streaming, multimedia & batch data
- Machine Learning (fourth trend)
  - Classification
  - Regression
  - Clustering
  - Collaborative filtering

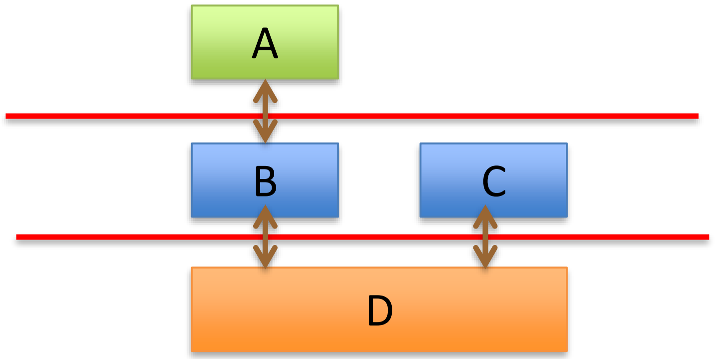
Working at the Intersection of these four trends is very exciting and challenging and require new ways to store and process **Big Data**



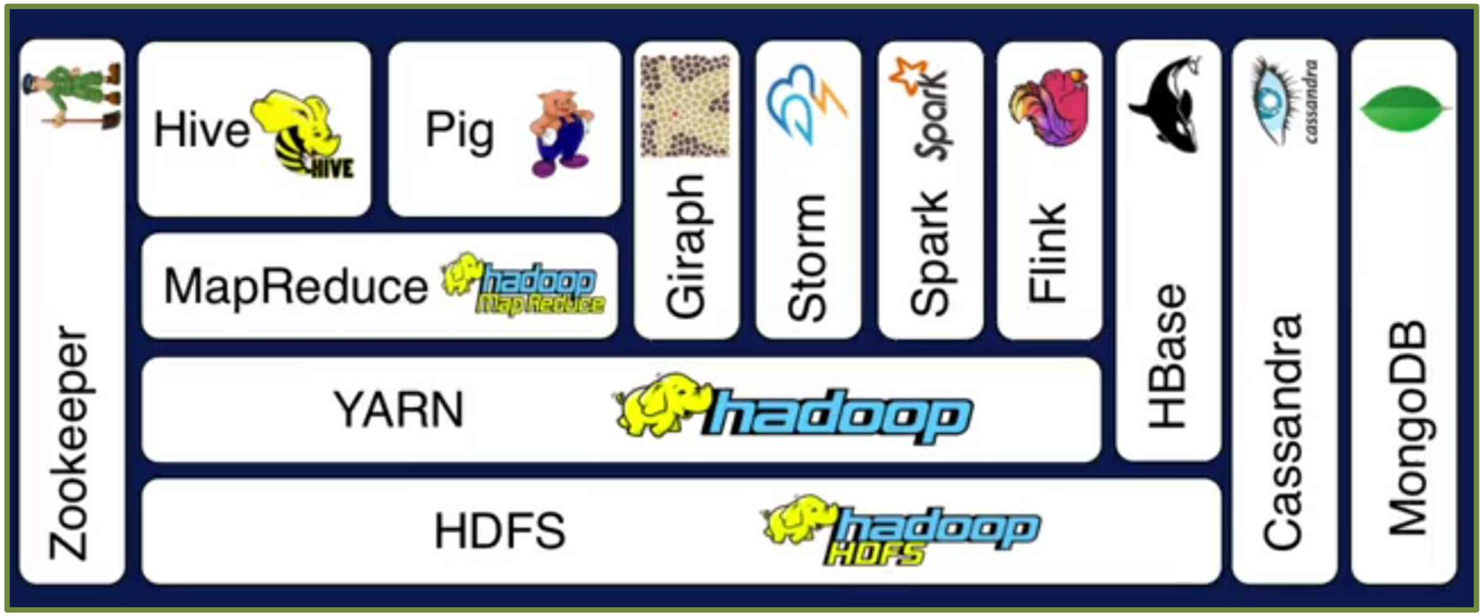
Hadoop Ecosystem

- Enable Scalability
  - on commodity hardware
- Handle Fault Tolerance
- Can Handle a Variety of Data type
  - Text, Graph, Streaming Data, Images,…
- Shared Environment
- Provides Value
  - Cost

Hadoop Ecosystem

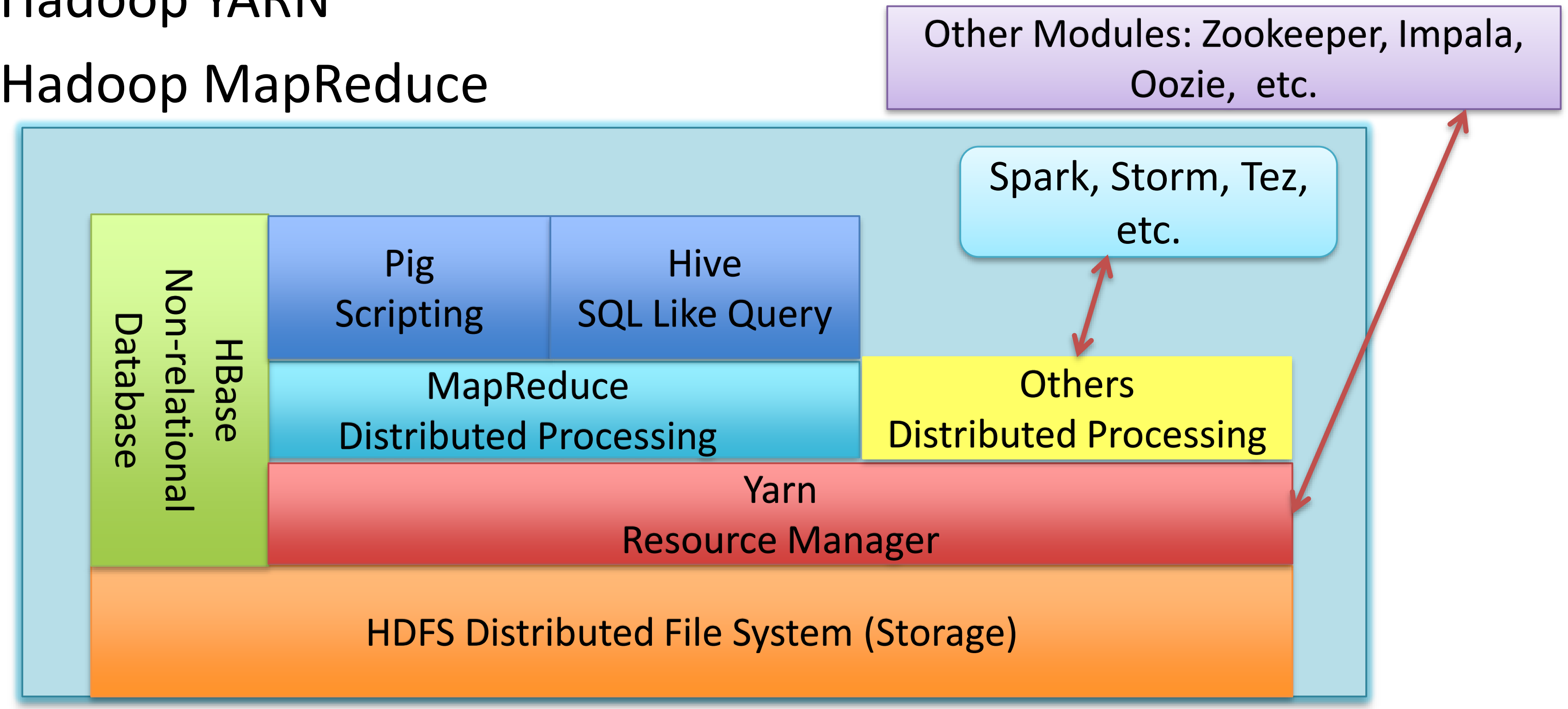


Layer Diagram



Apache Hadoop Basic Modules

- Hadoop Common
- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- Hadoop MapReduce


Other Modules: Zookeeper, Impala, Oozie, etc.
Spark, Storm, Tez, etc.
HBase Non-relational Database
Pig Scripting
Hive SQL Like Query
MapReduce Distributed Processing
Others Distributed Processing
Yarn Resource Manager
HDFS Distributed File System (Storage)

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

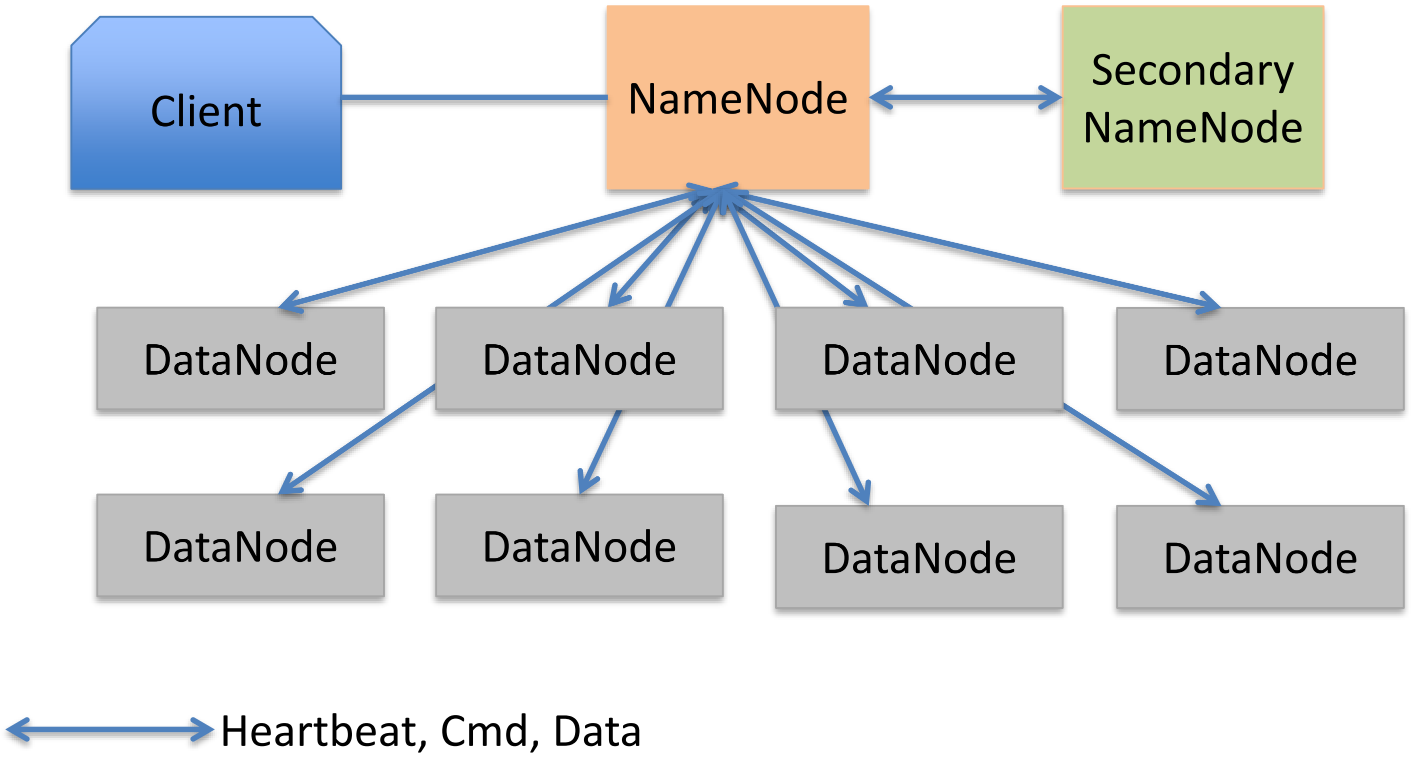
Single Hadoop cluster with 5000 servers and 250 petabytes of data

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

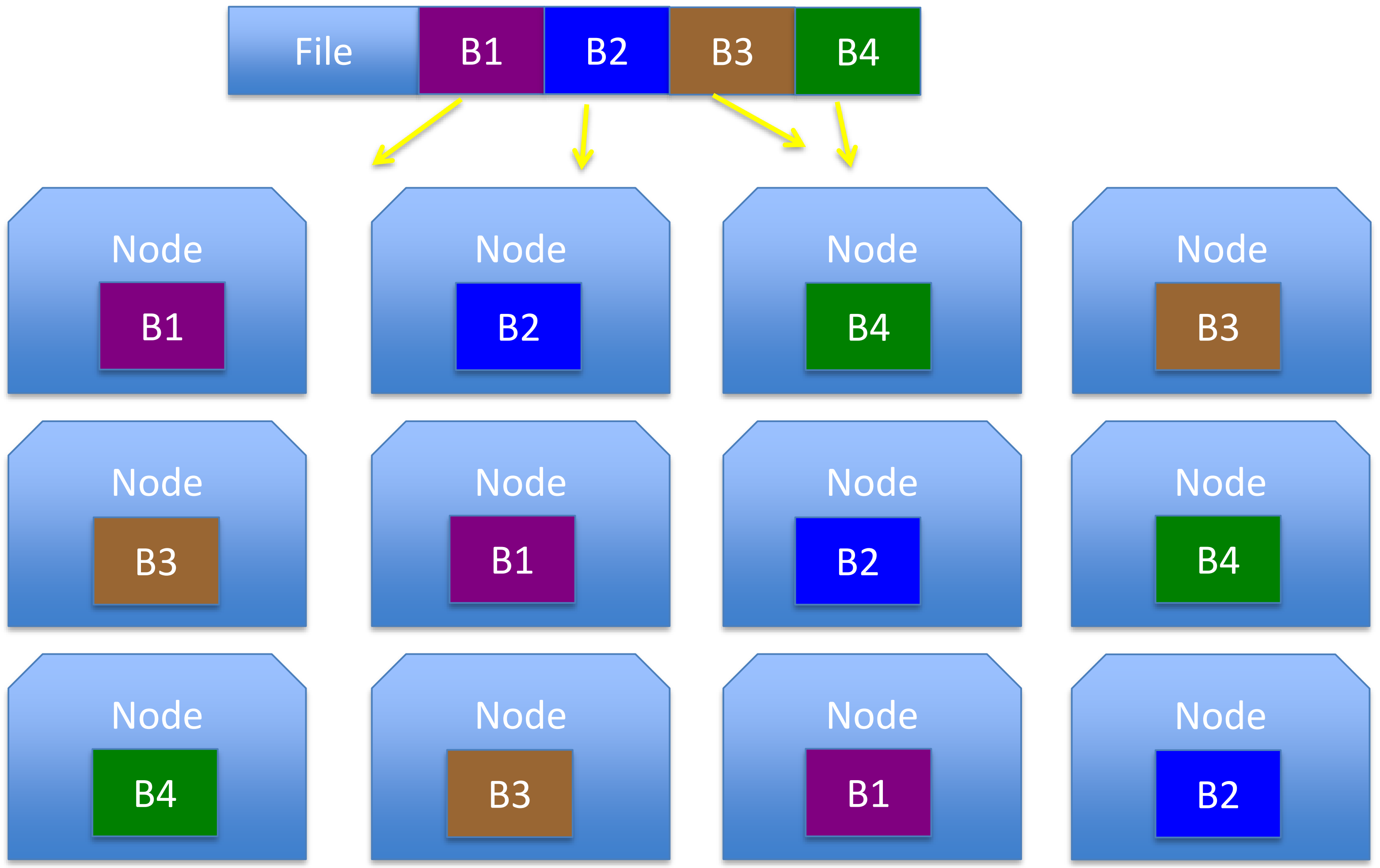
HDFS Architecture

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



HDFS

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



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

HBase

- NoSQL data store build on top of HDFS
- Based on the Google BigTable paper (2006)
- Can handle various types of data
- Stores large amount of data (TB,PB)
- Column-Oriented data store
- Big Data with random read and writes
- Horizontally scalable

HBase, not to use for

- Not good as a traditional RDBMs (Relational Database Model)
  - Transactional applications
  - Data Analytics

- Not efficient for text searching and processing

MapReduce: Simple Programming for Big Data

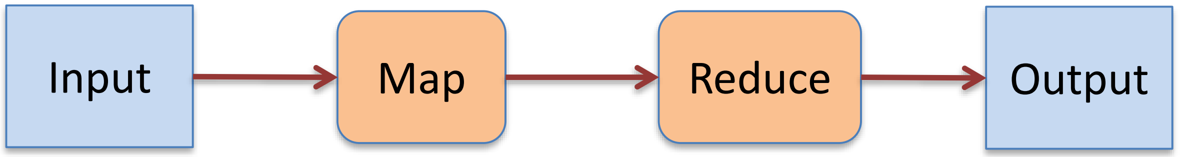
Based on Google's MR paper (2004)

- MapReduce is simple programming paradigm for the Hadoop ecosystem
- Traditional parallel programming requires expertise of different computing/systems concepts
  - examples: multithreads, synchronization mechanisms (locks, semaphores, and monitors)
  - incorrect use: can crash your program, get incorrect results, or severely impact performance
  - Usually not fault tolerant to hardware failure
- The MapReduce programming model greatly simplifies running code in parallel
  - you don't have to deal with any of above issues
  - only need to create, map and reduce functions

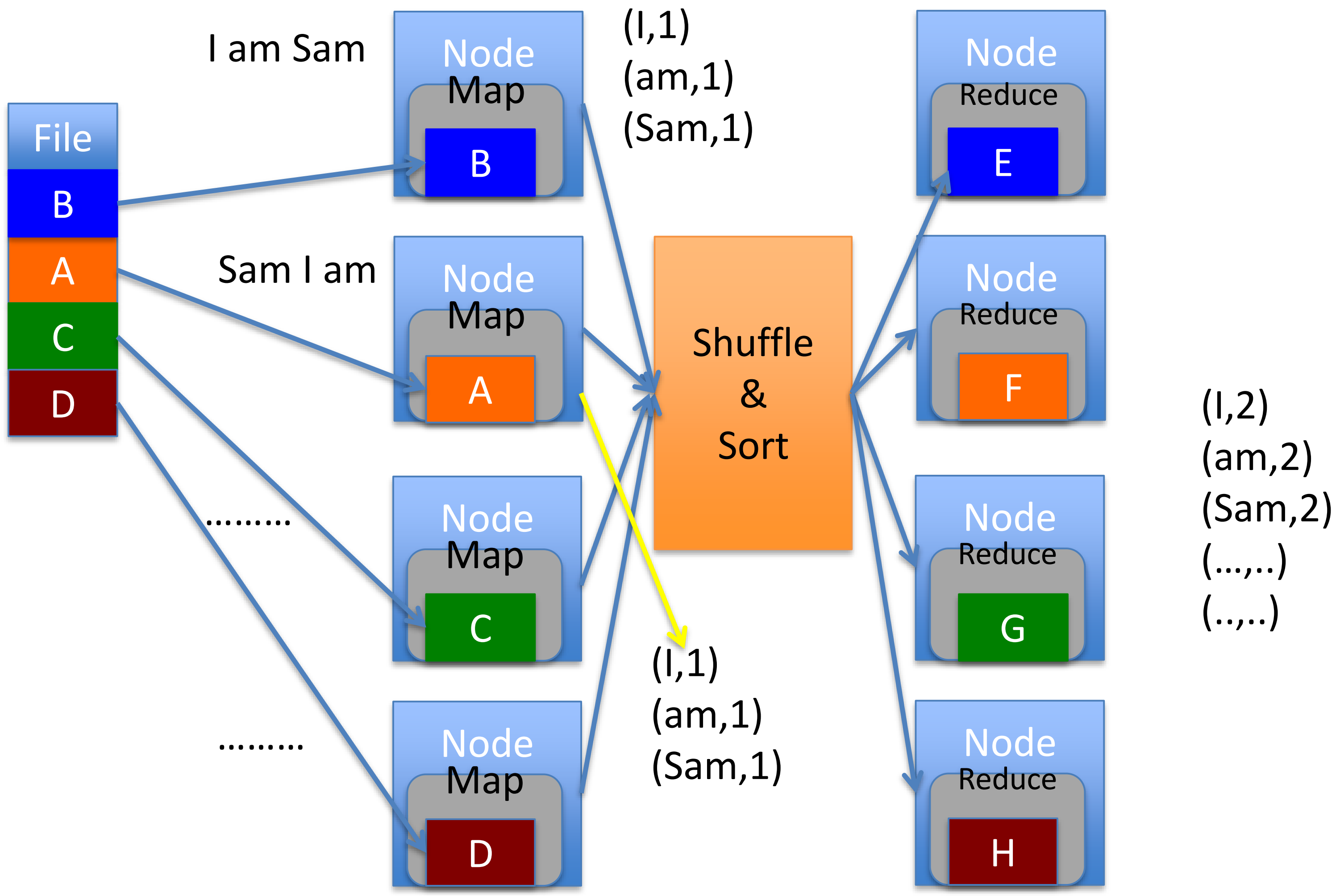
Map Reduce Paradigm

- 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



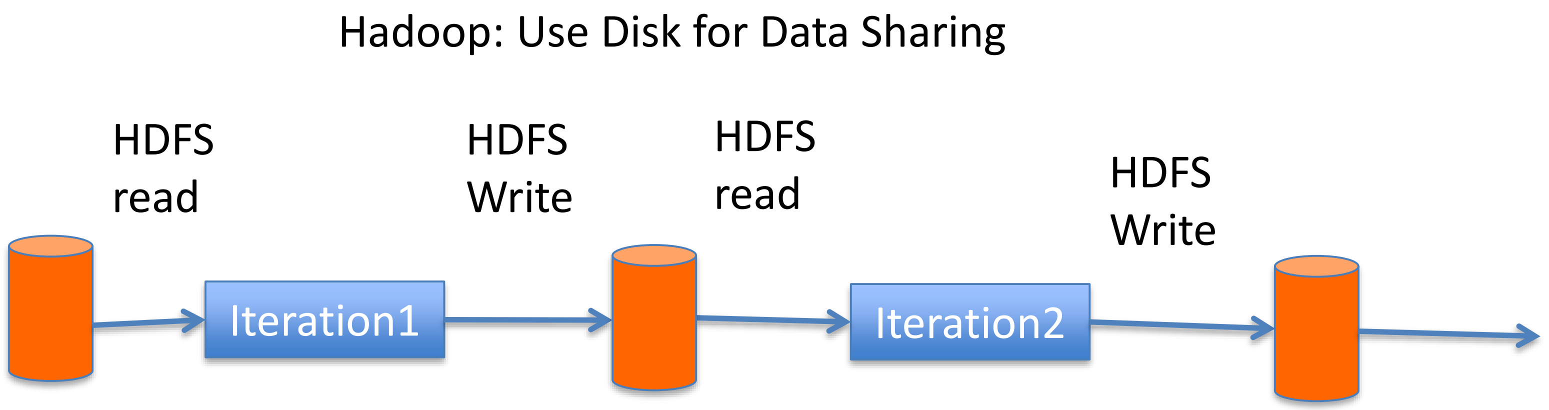
Shortcoming of MapReduce

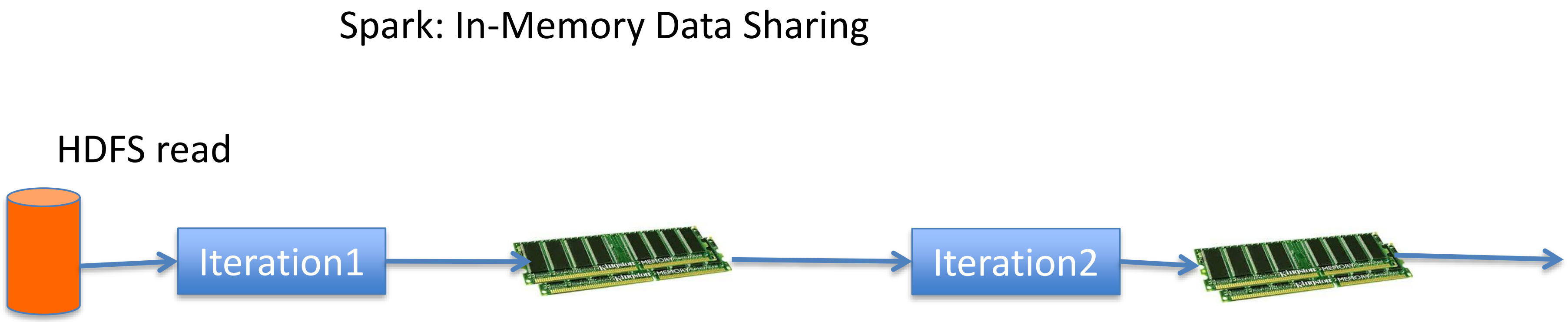
- Forces your data processing into Map and Reduce
  - Other workflows missing include join, filter, flatMap, groupByKey, union, intersection, …
- Based on “Acyclic Data Flow” from Disk to Disk (HDFS)
- Read and write to Disk before and after Map and Reduce (stateless machine)
  - Not efficient for iterative 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





Sort competition

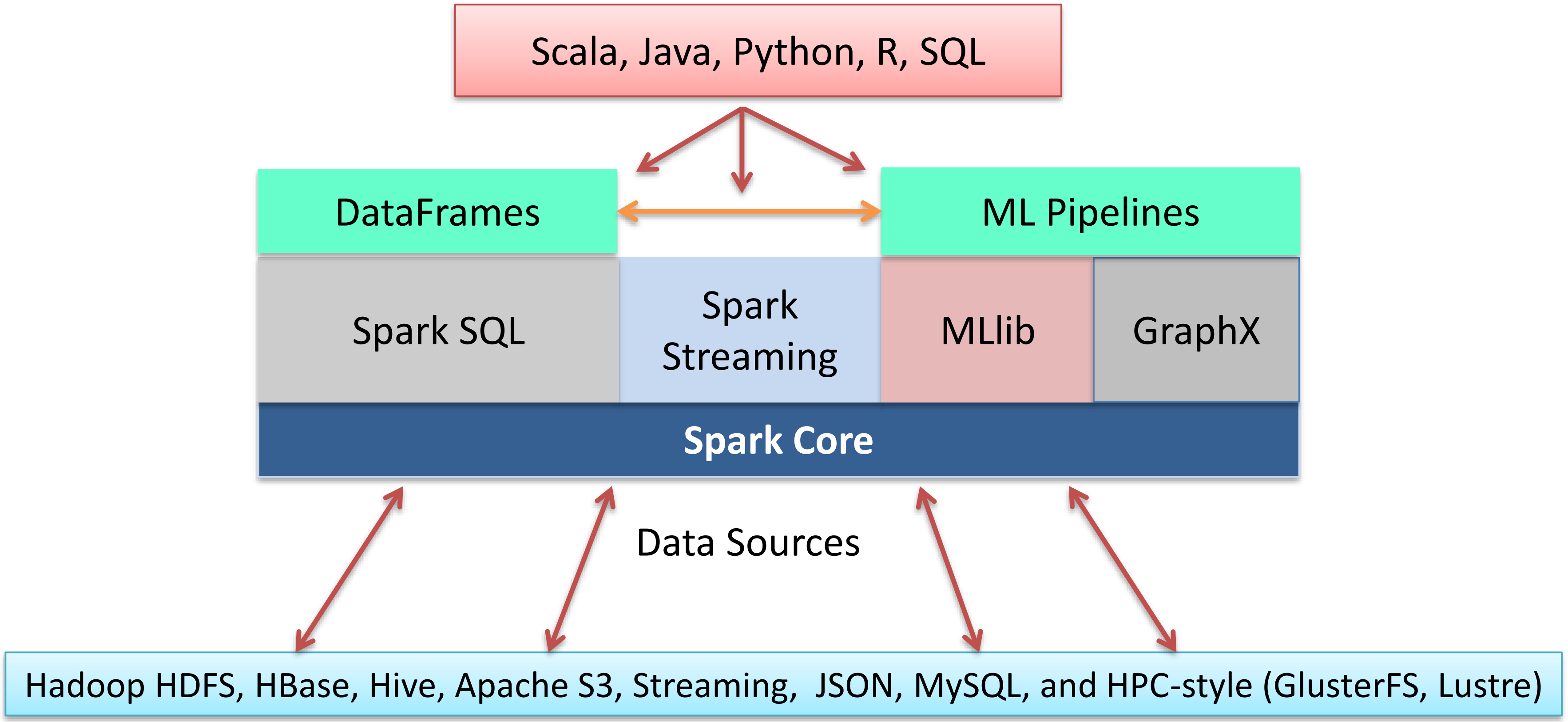
	Hadoop MR Record (2013)	Spark Record (2014)
Data Size	102.5 TB	100 TB
Elapsed Time	72 mins	23 mins
# Nodes	2100	206
# Cores	50400 physical	6592 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**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

Spark, 3x faster with 1/10 the nodes

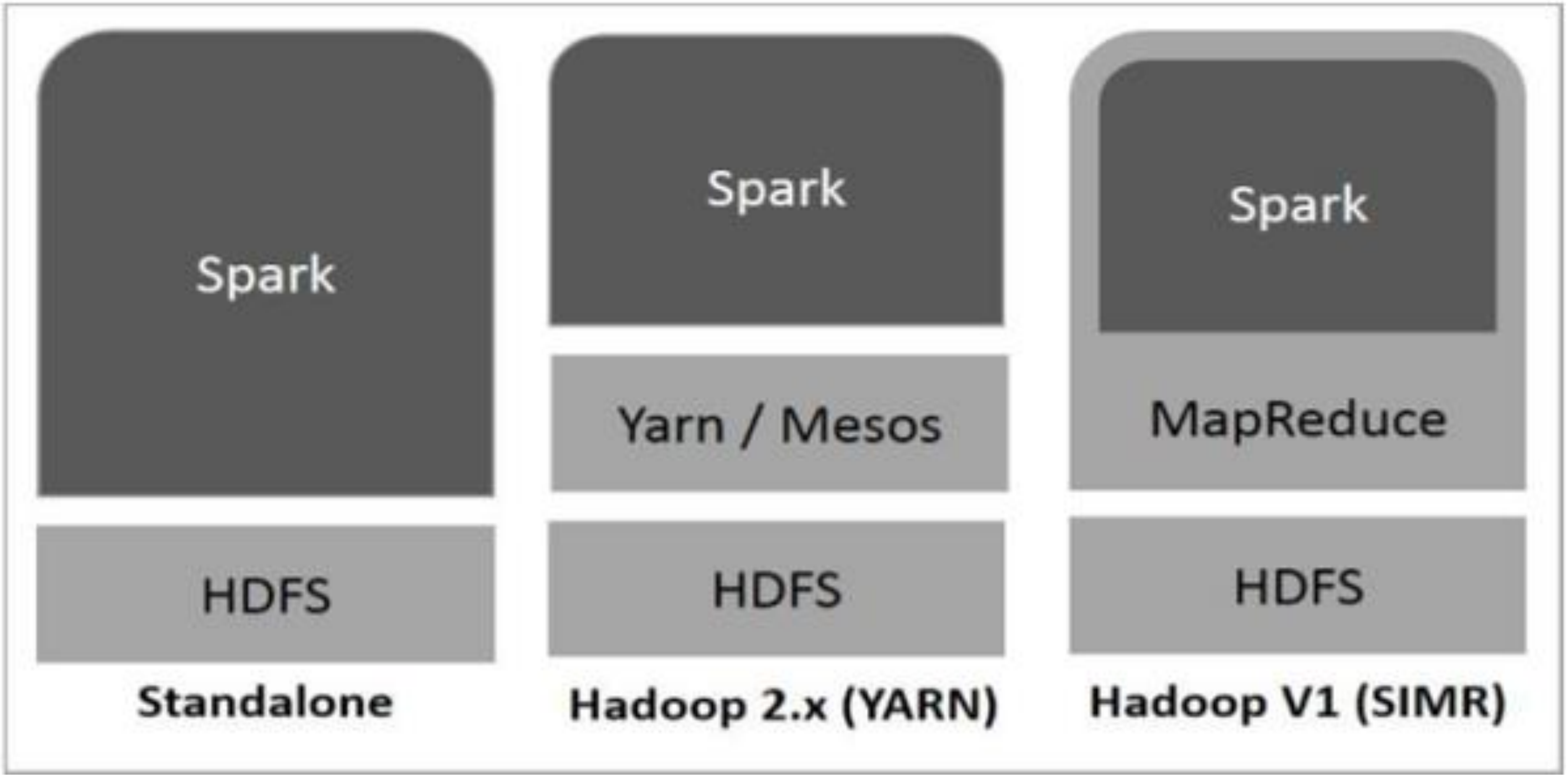
Sort benchmark, Daytona Gray: sort of 100 TB of data (1 trillion records)
http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

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.


Scala, Java, Python, R, SQL
DataFrames
ML Pipelines
Spark SQL
Spark Streaming
MLlib
GraphX
Spark Core
Data Sources
Hadoop HDFS, HBase, Hive, Apache S3, Streaming, JSON, MySQL, and HPC-style (GlusterFS, Lustre)

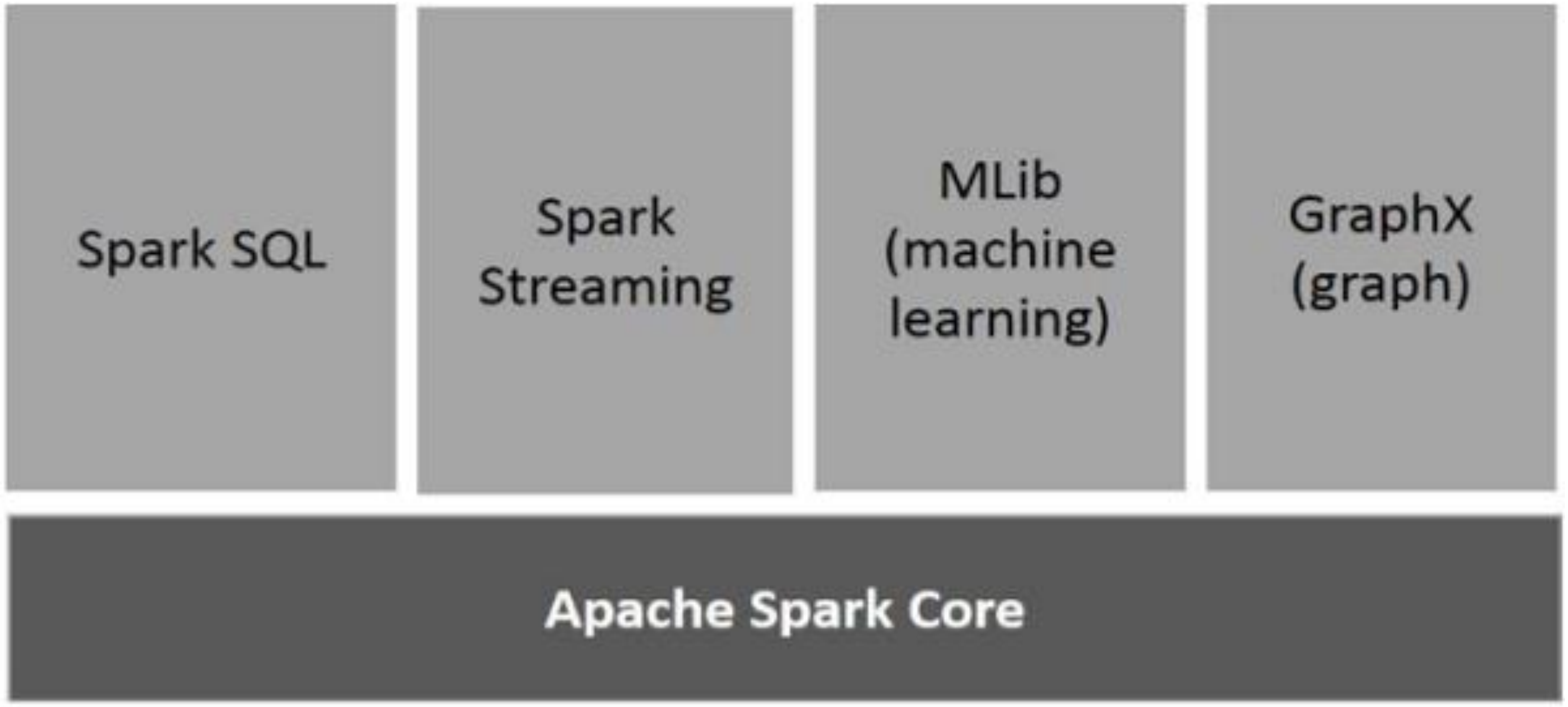
Spark Scenarios with Hadoop



Spark Scenarios with Hadoop

- **Standalone** − Spark Standalone deployment means Spark occupies the place on top of HDFS(Hadoop Distributed File System) and space is allocated for HDFS, explicitly. Here, Spark and MapReduce will run side by side to cover all spark jobs on cluster.
- **Hadoop Yarn** − Hadoop Yarn deployment means, simply, spark runs on Yarn without any pre-installation or root access required. It helps to integrate Spark into Hadoop ecosystem or Hadoop stack. It allows other components to run on top of stack.
- **Spark in MapReduce (SIMR)** − Spark in MapReduce is used to launch spark job in addition to standalone deployment. With SIMR, user can start Spark and uses its shell without any administrative access.

Spark Components



Components

- **Apache Spark Core** is the underlying general execution engine for spark platform that all other functionality is built upon. It provides In-Memory computing and referencing datasets in external storage systems.

- **Spark SQL** is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.

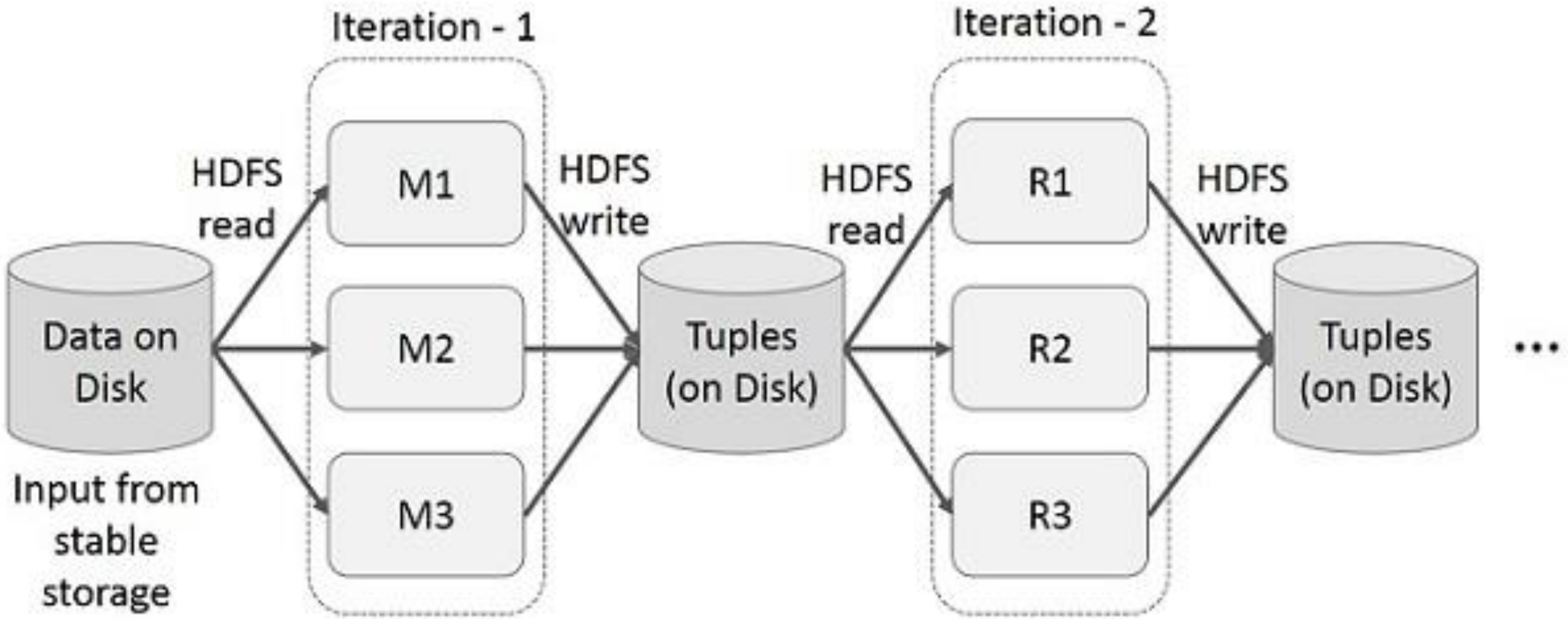
- **Spark Streaming** leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.

Components

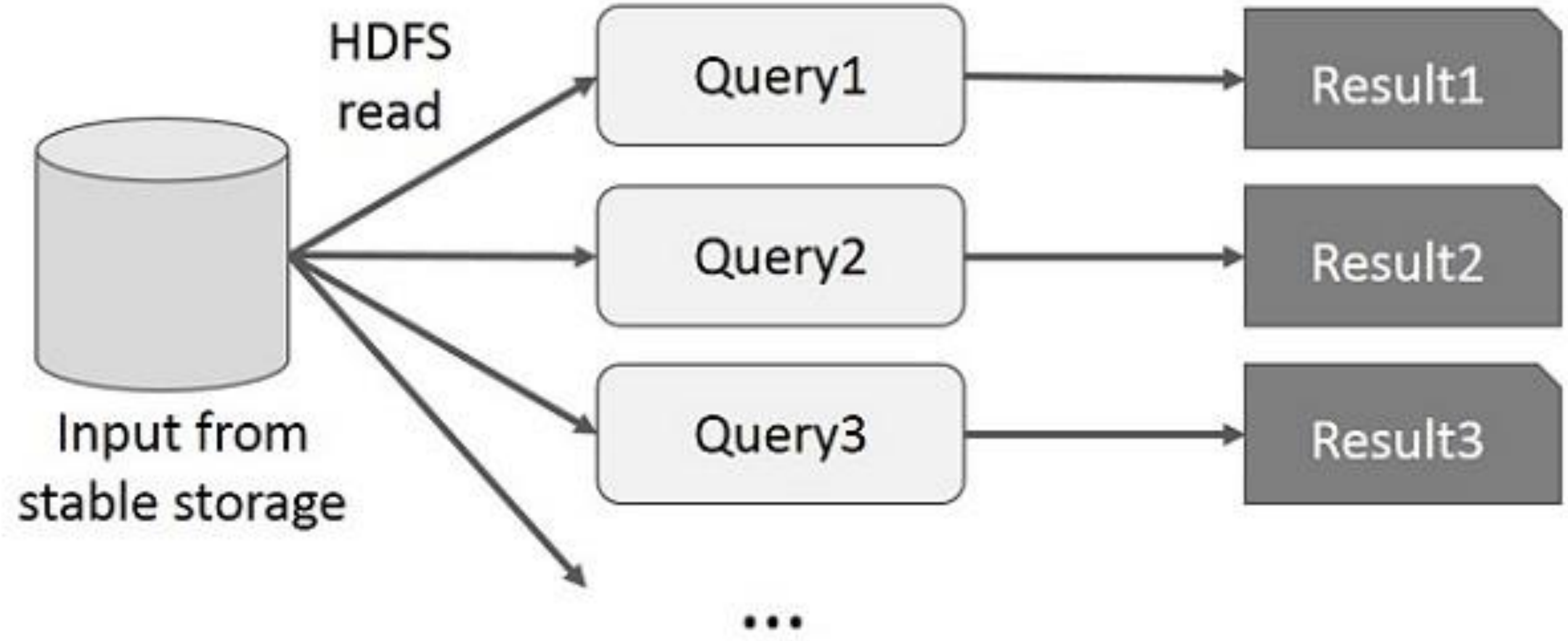
- **MLlib (Machine Learning Library)** is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture. It is, according to benchmarks, done by the MLlib developers against the Alternating Least Squares (ALS) implementations. Spark MLlib is nine times as fast as the Hadoop disk-based version of **Apache Mahout** (before Mahout gained a Spark interface).

- **GraphX** is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API. It also provides an optimized runtime for this abstraction.

Iterative Job in MapReduce



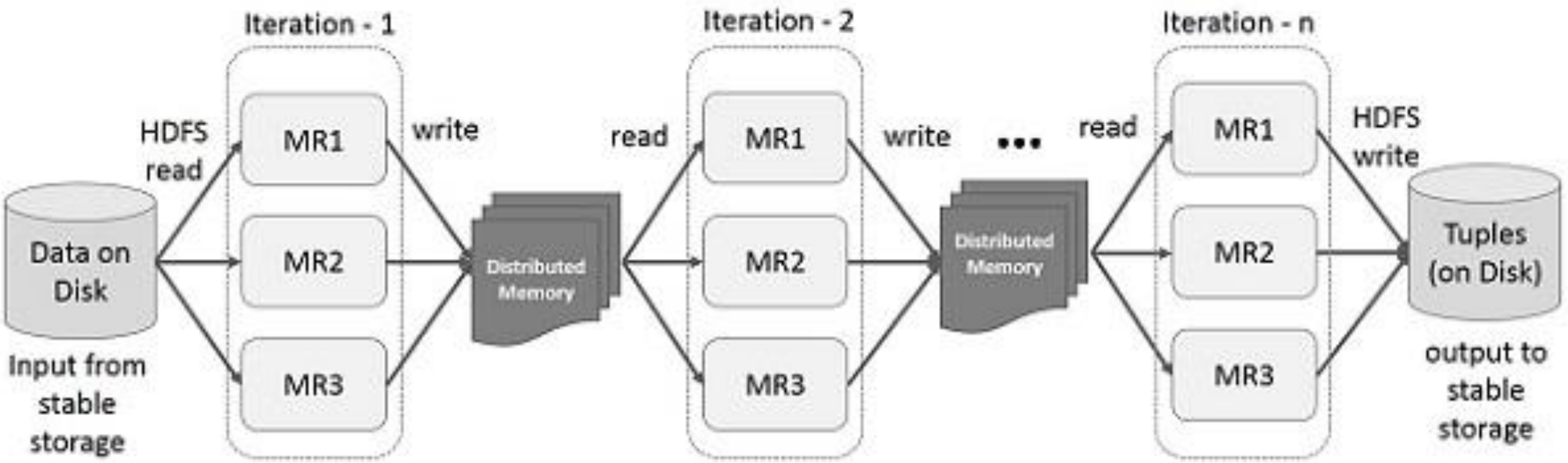
Interactive Job in MapReduce



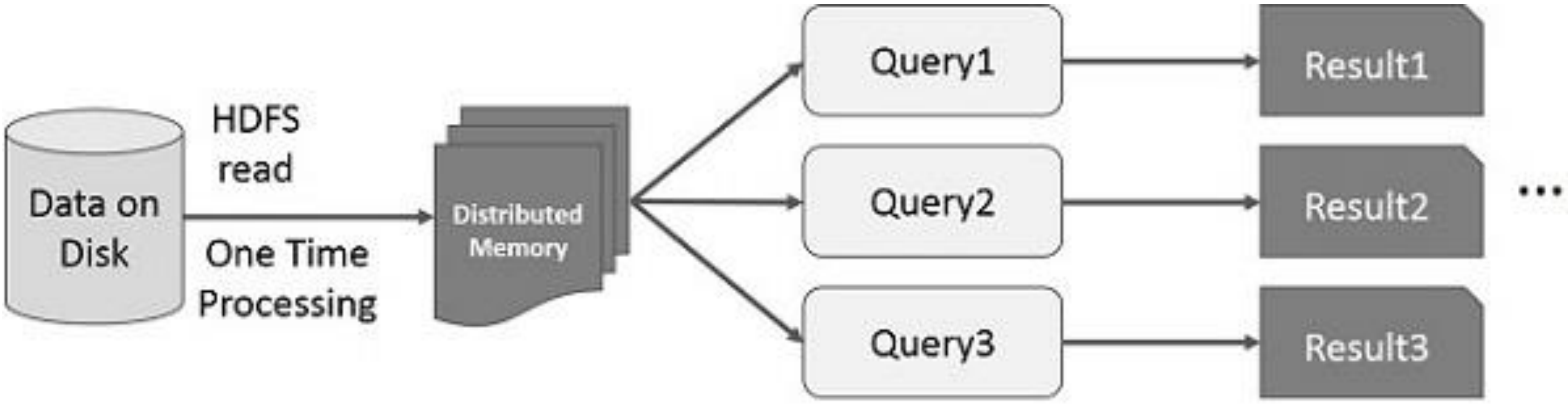
Resilient Distributed Datasets (RDDs)

- RDDs (Resilient Distributed Datasets) is **Data Containers**
- All the different processing components in Spark share the **same abstraction called RDD**
- As applications share the RDD abstraction, you can mix different kind of transformations to create new RDDs
- Created by parallelizing a collection or reading a file
- Fault tolerant

Iterative Operations on Spark RDD



Interactive Operations on Spark RDD



Transformations & Meaning

S.No	Transformations & Meaning
1	**map(func)** Returns a new distributed dataset, formed by passing each element of the source through a function **func**.
2	**filter(func)** Returns a new dataset formed by selecting those elements of the source on which **func** returns true.
3	**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).
4	**mapPartitions(func)** Similar to map, but runs separately on each partition (block) of the RDD, so **func** must be of type Iterator<T> ⇒ Iterator<U> when running on an RDD of type T.

Transformations & Meaning

5	**mapPartitionsWithIndex(func)** Similar to map Partitions, but also provides **func** with an integer value representing the index of the partition, so **func** must be of type (Int, Iterator<T>) ⇒ Iterator<U> when running on an RDD of type T.
6	**sample(withReplacement, fraction, seed)** Sample a **fraction** of the data, with or without replacement, using a given random number generator seed.
7	**union(otherDataset)** Returns a new dataset that contains the union of the elements in the source dataset and the argument.
8	**intersection(otherDataset)** Returns a new RDD that contains the intersection of elements in the source dataset and the argument.
9	**distinct([numTasks])** Returns a new dataset that contains the distinct elements of the source dataset.

10	**groupByKey([numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note** − If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.
11	**reduceByKey(func, [numTasks])** 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. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
12	**aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])** When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different from the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

13	**sortByKey([ascending], [numTasks])** When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.
14	**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. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.
15	**cogroup(otherDataset, [numTasks])** When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called group With.
16	**cartesian(otherDataset)** When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).

17	**pipe(command, [envVars])** Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
18	**coalesce(numPartitions)** Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
19	**repartition(numPartitions)** Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
20	**repartitionAndSortWithinPartitions(partitioner)** Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

DataFrames & SparkSQL

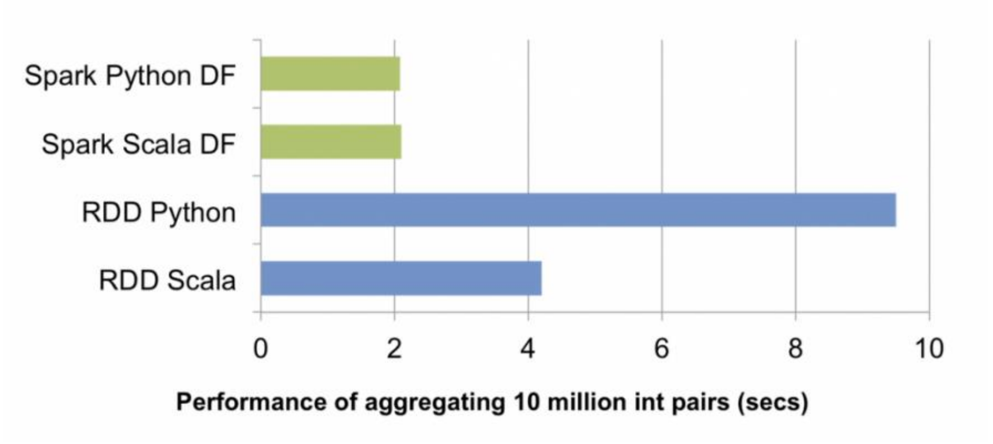
- **DataFrames (DFs)** is one of the other distributed datasets organized in named columns
- Similar to a relational database, Python Pandas Dataframe or R's DataTables
  - Immutable once constructed
  - Track lineage
  - Enable distributed computations
- How to construct Dataframes
  - Read from file(s)
  - Transforming an existing DFs(Spark or Pandas)
  - Parallelizing a python collection list
  - Apply transformations and actions

DataFrame example

```
// Create a new DataFrame that contains “students”
students = users.filter(users.age < 21)

//Alternatively, using Pandas-like syntax
students = users[users.age < 21]

//Count the number of students users by gender
students.groupBy("gender").count()

// Join young students with another DataFrame called
logs
students.join(logs, logs.userId == users.userId,
“left_outer")
```

RDDs vs. DataFrames

- RDDs provide a low level interface into Spark
- DataFrames have a schema
- DataFrames are cached and optimized by Spark
- DataFrames are built on top of the RDDs and the core Spark API

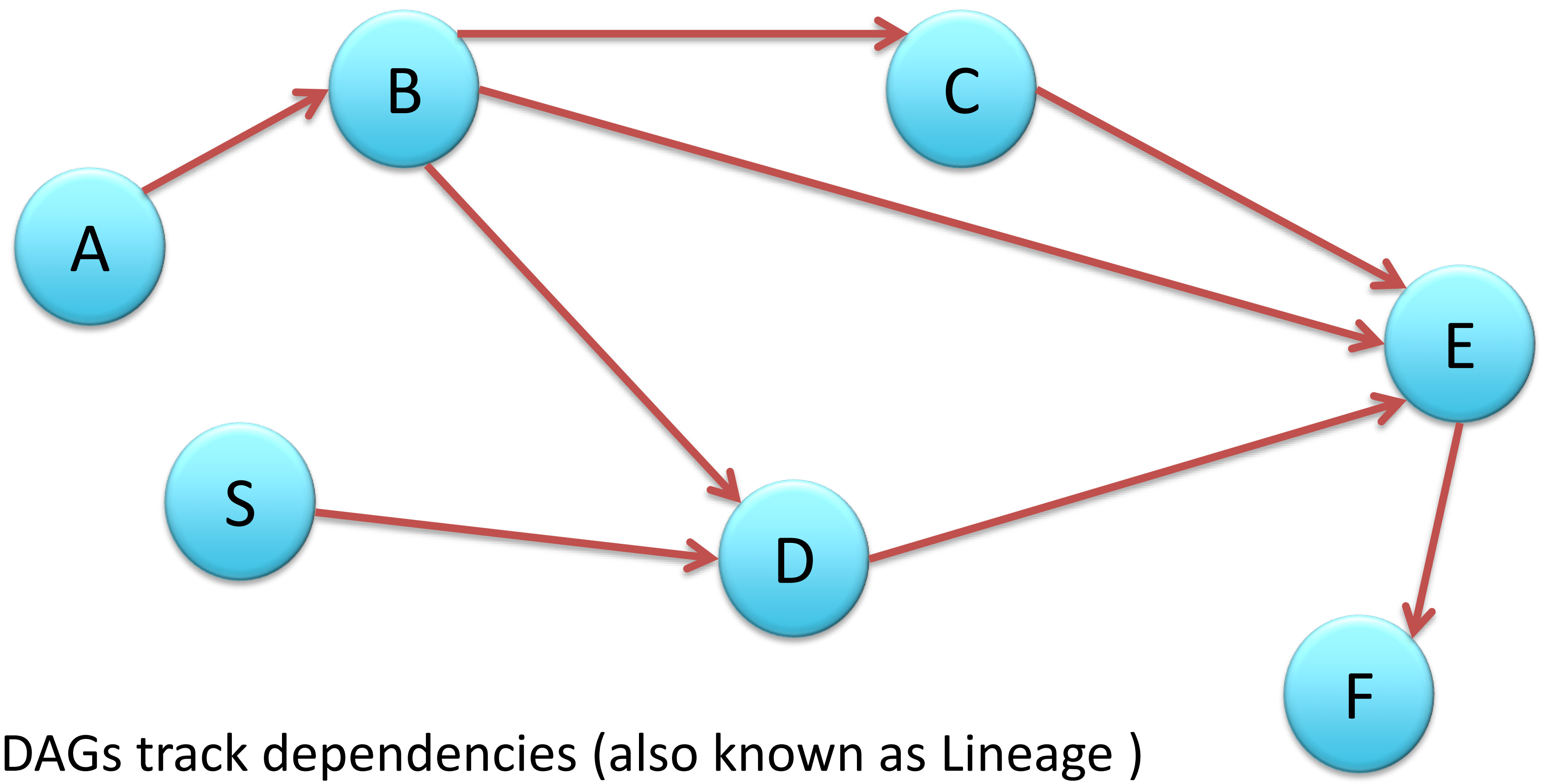


Example: performance

Spark Operations

Transformations (create a new RDD)	map	flatMap
	filter	union
	sample	join
	groupByKey	cogroup
	reduceByKey	cross
	sortByKey	mapValues
	intersection	reduceByKey
Actions (return results to driver program)	collect	first
	Reduce	take
	Count	takeOrdered
	takeSample	countByKey
	take	save
	lookupKey	foreach

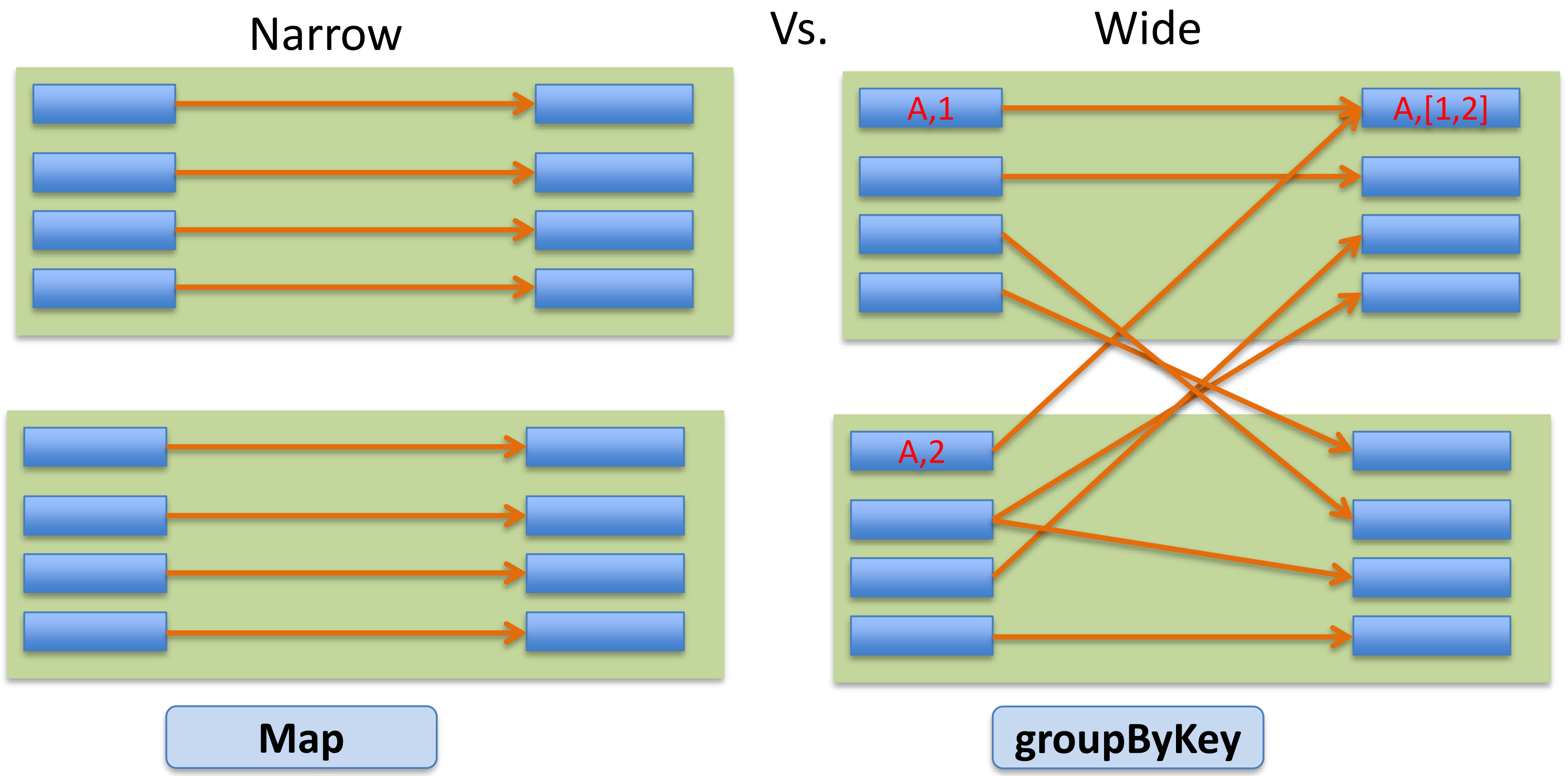
Directed Acyclic Graphs (DAG)



DAGs track dependencies (also known as Lineage)

- nodes are RDDs
- arrows are Transformations

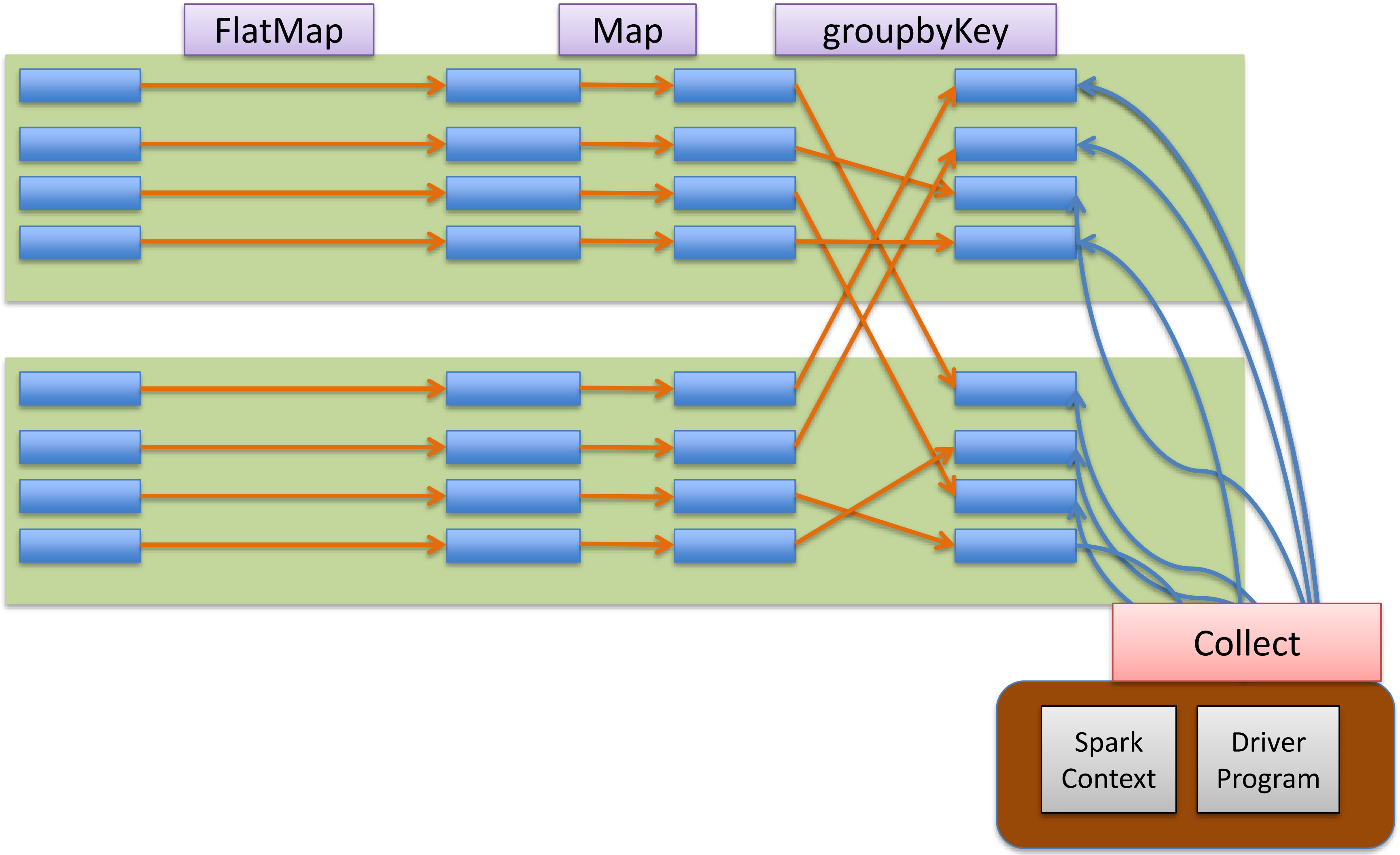
Narrow Vs. Wide transformation



Actions

- What is an action
  - The final stage of the workflow
  - Triggers the execution of the DAG
  - Returns the results to the driver
  - Or writes the data to HDFS or to a file

Spark Workflow



Python RDD API Examples

- Word count

```
text_file = sc.textFile("hdfs:\\usr\godil\text\book.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
          .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://usr/godil/output/wordCount.txt")
```

- Logistic Regression

```
# Every record of this DataFrame contains the label and
# features represented by a vector.
df = sqlContext.createDataFrame(data, ["label", "features"])
# Set parameters for the algorithm.
# Here, we limit the number of iterations to 10.
lr = LogisticRegression(maxIter=10)
# Fit the model to the data.
model = lr.fit(df)
# Given a dataset, predict each point's label, and show the results.
model.transform(df).show()
```

Examples from http://spark.apache.org/

RDD Persistence and Removal

- RDD Persistence
  - RDD.persist()
  - Storage level:
    - MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, DISK_ONLY,…….

- RDD Removal
  - RDD.unpersist()

Broadcast Variables and Accumulators (Shared Variables)

- Broadcast variables allow the programmer to keep a read-only variable cached on each node, rather than sending a copy of it with tasks

```
>broadcastV1 = sc.broadcast([1, 2, 3,4,5,6])
>broadcastV1.value
[1,2,3,4,5,6]
```

- Accumulators are variables that are only “added” to through an associative operation and can be efficiently supported in parallel

```
accum = sc.accumulator(0)
accum.add(x)
accum.value
```

Spark’s Main Use Cases

- Streaming Data
- Machine Learning
- Interactive Analysis
- Data Warehousing
- Batch Processing
- Exploratory Data Analysis
- Graph Data Analysis
- Spatial (GIS) Data Analysis
- And many more

Exercise

- Write Python/Java code to cluster the data on at least 3 attributes (for city, age, contracted from) using COVID 19 data set and run it in Spark environment.
- Twitter Sentiment Analysis
  - Developed a Spark based Sentiment Analysis code for a Twitter dataset

Spark in the Real World (I)

- Uber – the online taxi company gathers terabytes of event data from its mobile users every day.
  - By using Kafka, Spark Streaming, and HDFS, to build a continuous ETL pipeline
  - Convert raw unstructured event data into structured data as it is collected
  - Uses it further for more complex analytics and optimization of operations

- Pinterest – Uses a Spark ETL pipeline
  - Leverages Spark Streaming to gain immediate insight into how users all over the world are engaging with Pins—in real time.
  - Can make more relevant recommendations as people navigate the site
  - Recommends related Pins
  - Determine which products to buy, or destinations to visit

Spark in the Real World (II)

Here are Few other Real World Use Cases:

- Conviva – 4 million video feeds per month
  - This streaming video company is second only to YouTube.
  - Uses Spark to reduce customer churn by optimizing video streams and managing live video traffic
  - Maintains a consistently smooth, high quality viewing experience.
- Capital One – is using Spark and data science algorithms to understand customers in a better way.
  - Developing next generation of financial products and services
  - Find attributes and patterns of increased probability for fraud
- Netflix – leveraging Spark for insights of user viewing habits and then recommends movies to them.
  - User data is also used for content creation

Spark: when not to use

- Even though Spark is versatile, that doesn’t mean Spark’s in-memory capabilities are the best fit for all use cases:
  - For many simple use cases Apache MapReduce and Hive might be a more appropriate choice
  - Spark was not designed as a multi-user environment
  - Spark users are required to know that memory they have is sufficient for a dataset
  - Adding more users adds complications, since the users will have to coordinate memory usage to run code

HPC and Big Data Convergence

- Clouds and supercomputers are collections of computers networked together in a datacenter
- Clouds have different networking, I/O, CPU and cost trade-offs than supercomputers
- Cloud workloads are data oriented vs. computation oriented and are less closely coupled than supercomputers
- Principles of parallel computing same on both
- Apache Hadoop and Spark vs. Open MPI