

HDFS and Spark



Rajiv Misra

Dept. of Computer Science &
Engineering

Indian Institute of Technology Patna

rajivm@iitp.ac.in

The Hadoop Distributed File System (HDFS)

Introduction

- **Hadoop provides a distributed file system** and a framework for the analysis and transformation of very large data sets using the **MapReduce paradigm**.
- An important characteristic of **Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application computations in parallel close to their data**.
- A Hadoop cluster **scales computation capacity, storage capacity and IO bandwidth** by simply adding commodity servers. Hadoop clusters at Yahoo! span 25,000 servers, and store 25 petabytes of application data, with the largest cluster being 3500 servers. One hundred other organizations worldwide report using Hadoop.

Contd...

- Hadoop is an **Apache project**; all components are available via the Apache open source license.
- **Yahoo!** has developed and contributed to 80% of the core of Hadoop (**HDFS and MapReduce**).
- **HBase** was originally developed at **Powerset**, now a department at Microsoft.
- **Hive** was originated and developed at **Facebook**.
- **Pig, ZooKeeper, and Chukwa** were originated and developed at **Yahoo!**
- **Avro** was originated at **Yahoo!** and is being co-developed with **Cloudera**.

Hadoop Project Components

HDFS	Distributed file system
MapReduce	Distributed computation framework
HBase	Column-oriented table service
Pig	Dataflow language and parallel execution framework
Hive	Data warehouse infrastructure
ZooKeeper	Distributed coordination service
Chukwa	System for collecting management data
Avro	Data serialization system

Table 1. Hadoop project components

Contd...

- **HDFS is the file system component of Hadoop.** While the interface to HDFS is patterned after the UNIX file system, faithfulness to standards was sacrificed in favour of improved performance for the applications at hand.
- **HDFS stores file system metadata and application data separately.**
- As in other distributed file systems, like PVFS, Lustre and, HDFS **stores metadata on a dedicated server, called the NameNode.**
- **Application data are stored on other servers called DataNodes. All servers are fully connected** and communicate with each other using **TCP-based protocols.**

HDFS Design Assumptions

- Single machines tend to fail
 - Hard disk, power supply,
- More machines = increased failure probability
- Data doesn't fit on a single node
- **Desired:**
 - Commodity hardware
 - Built-in backup and failover

Architecture

Namenode and Datanodes

- **Namenode (*Master*)**

- Metadata:

- Where file blocks are stored (namespace image)

- Edit (*Operation*) log

- Secondary namenode (*Shadow master*)

- **Datanode (*Chunkserver*)**

- Stores and retrieves blocks

- by client or namenode.

- Reports to namenode with list of blocks they are storing

Noticeable Differences from GFS

- Only single-writers per file.
 - **No record append operation.**
- **Open source**
 - Provides **many interfaces and libraries** for different file systems.
 - S3, KFS, etc.
 - Thrift (C++, Python, ...), libhdfs (C), FUSE

A) Namenode

- **The HDFS namespace is a hierarchy of files and directories.** Files and directories are represented on the NameNode by **inodes**, which record attributes like permissions, modification and access times, namespace and disk space quotas.
- The file content is split into large blocks (**typically 128 megabytes**, but user selectable file-by-file) and each block of the file is independently replicated at multiple **DataNodes** (typically three, but user selectable file-by-file). The **NameNode** maintains the namespace tree and the mapping of file blocks to DataNodes (the physical location of file data).
- **An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the client.**

Contd...

- When writing data, the client requests the **NameNode** to nominate a suite of **three DataNodes** to host the block replicas.
- The client then **writes data to the DataNodes in a pipeline fashion.**
- The current design has a **single NameNode** for each cluster.
- The cluster can have **thousands of DataNodes** and **tens of thousands of HDFS clients per cluster**, as each **DataNode** may execute multiple application tasks concurrently.

Contd...

- **HDFS keeps the entire namespace in RAM.** The inode data and the list of blocks belonging to each file comprise the metadata of the name system called the **image**. The persistent record of the image stored in the local host's native files system is called a **checkpoint**.
- **The NameNode** also stores the modification log of the image called the **journal** in the local host's native file system. For improved durability, redundant copies of the checkpoint and journal can be made at other servers.
- **During restarts the NameNode restores the namespace** by reading the namespace and replaying the journal. The locations of block replicas may change over time and are not part of the persistent checkpoint.

B) Datanode

- **Each block replica on a DataNode is represented by two files in the local host's native file system.** The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp.
- **The size of the data file equals the actual length of the block** and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.
- **During startup each DataNode connects to the NameNode and performs a handshake.** The purpose of the handshake is to **verify the namespace ID** and **the software version** of the DataNode. If either does not match that of the NameNode the DataNode automatically shuts down.

Contd...

- The **namespace ID** is assigned to the file system instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace ID will not be able to join the cluster, thus preserving the integrity of the file system.
- The **consistency of software versions is important** because **incompatible version may cause data corruption or loss**, and on large clusters of thousands of machines it is easy to overlook nodes that did not shut down properly prior to the software upgrade or were not available during the upgrade.
- A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID.

Contd...

- **After the handshake the DataNode registers with the NameNode.** DataNodes persistently store their unique storage IDs. **The storage ID is an internal identifier of the DataNode,** which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that.
- A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block id, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-to-date view of where block replicas are located on the cluster.

Contd...

- **During normal operation DataNodes send heartbeats to the NameNode** to confirm that the DataNode is operating and the block replicas it hosts are available. The **default heartbeat interval is three seconds.**
- If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable.
- The NameNode then schedules creation of new replicas of those blocks on other DataNodes.

Contd...

- **Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress.** These statistics are used for the NameNode's space allocation and load balancing decisions.
- **The NameNode does not directly call DataNodes. It uses replies to heartbeats to send instructions to the DataNodes.** The instructions include commands to:
 - Replicate blocks to other nodes;
 - Remove local block replicas;
 - Re-register or to shut down the node;
 - Send an immediate block report.
- These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The **NameNode can process thousands of heartbeats per second** without affecting other NameNode operations.

C) HDFS Client

- User applications access the file system using the **HDFS client, a code library that exports the HDFS file system interface.**
- Similar to most conventional file systems, **HDFS supports operations to read, write and delete files, and operations to create and delete directories.**
- The user references files and directories by paths in the namespace.
- The user application generally does not need to know that file system metadata and storage are on different servers, or that blocks have multiple replicas.

Contd...

- **When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes** that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block.
- When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data.
- When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of DataNodes is likely to be different.
- The interactions among the client, the NameNode and the DataNodes are illustrated in **Figure 1**.

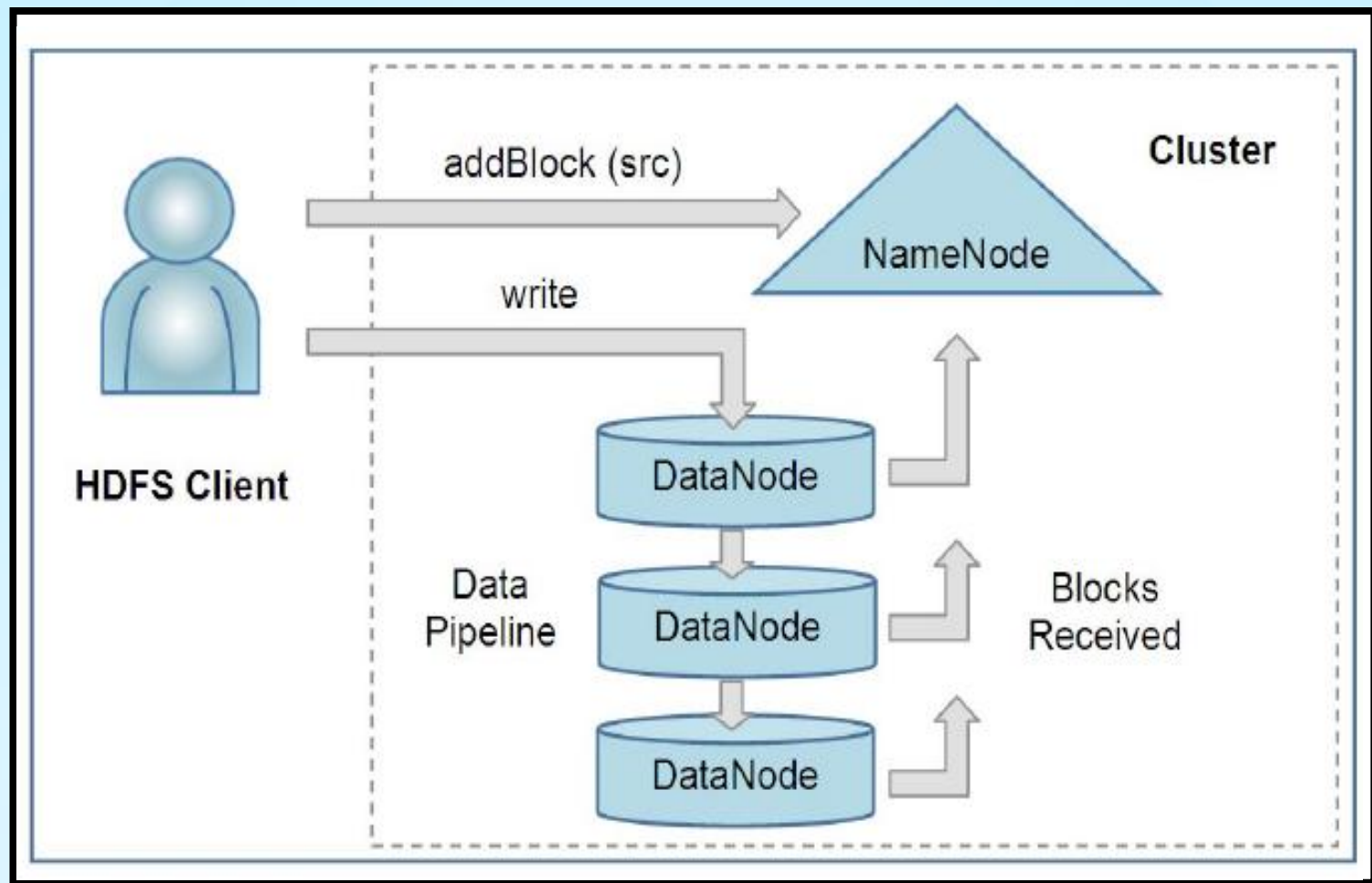
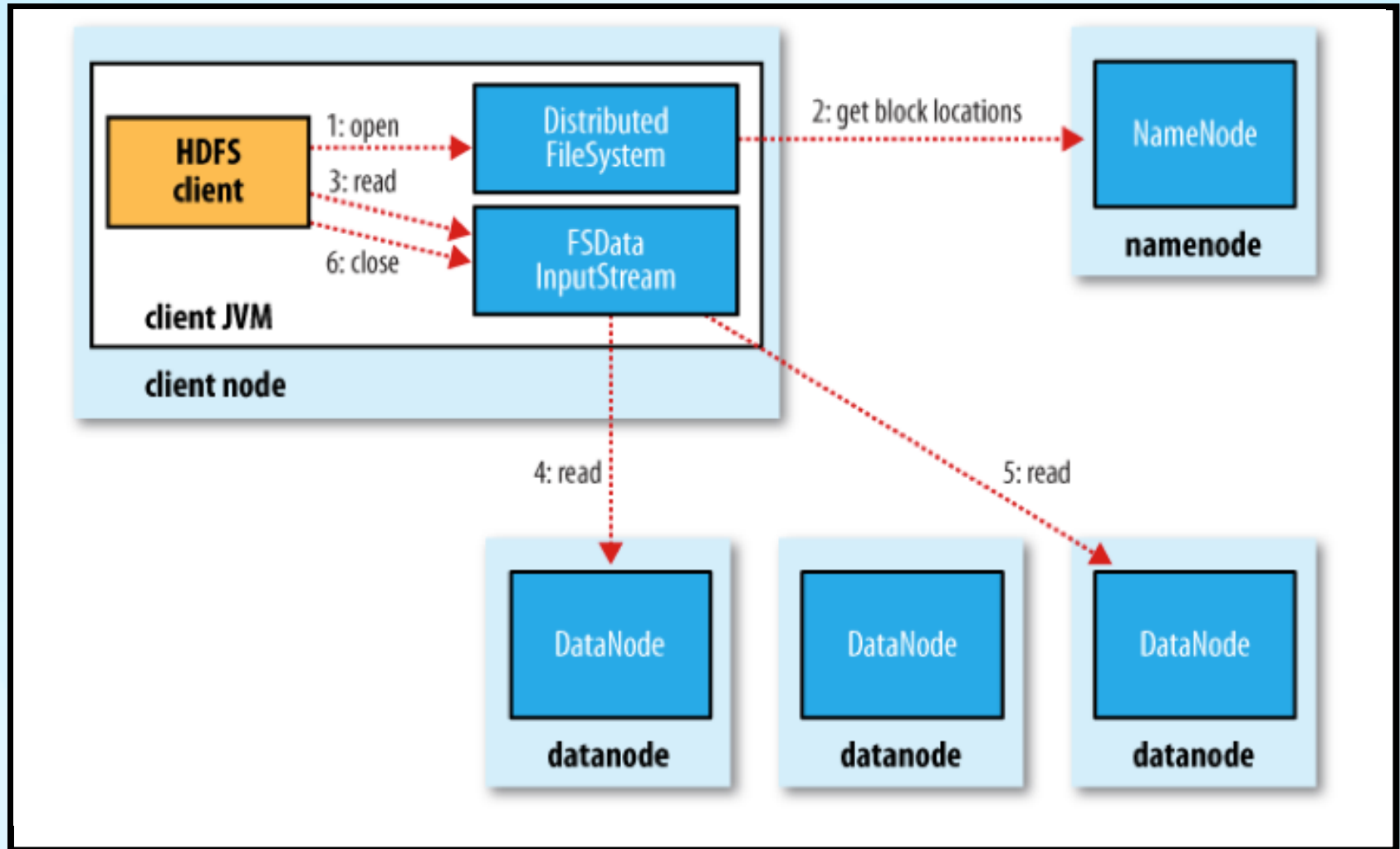


Figure 1. An HDFS client creates a new file by giving its path to the NameNode. For each block of the file, the NameNode returns a list of DataNodes to host its replicas. The client then pipelines data to the chosen DataNodes, which eventually confirm the creation of the block replicas to the NameNode.

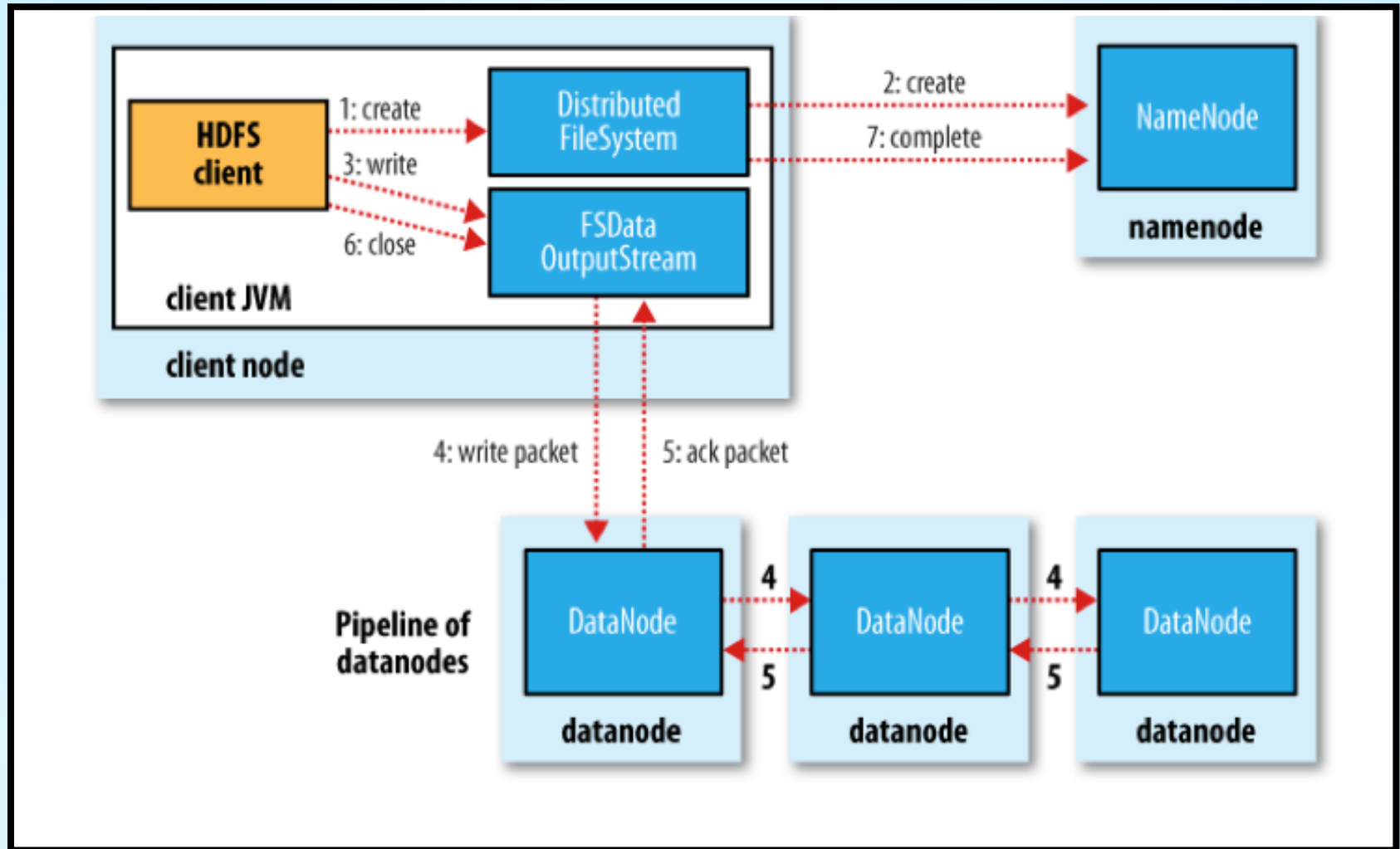
Contd...

- Unlike conventional file systems, **HDFS provides an API** that exposes the locations of a file blocks.
- This allows applications like the **MapReduce framework to schedule a task** to where the data are located, thus improving the read performance.
- It also allows an application to set the replication factor of a file. By default a file's replication factor is three.
- For critical files or files which are accessed very often, having a higher replication factor improves their tolerance against faults and increase their read bandwidth.

Anatomy of a File Read



Anatomy of a File Write



Additional Topics

- **Replica placements:**

- Different node, rack, and center

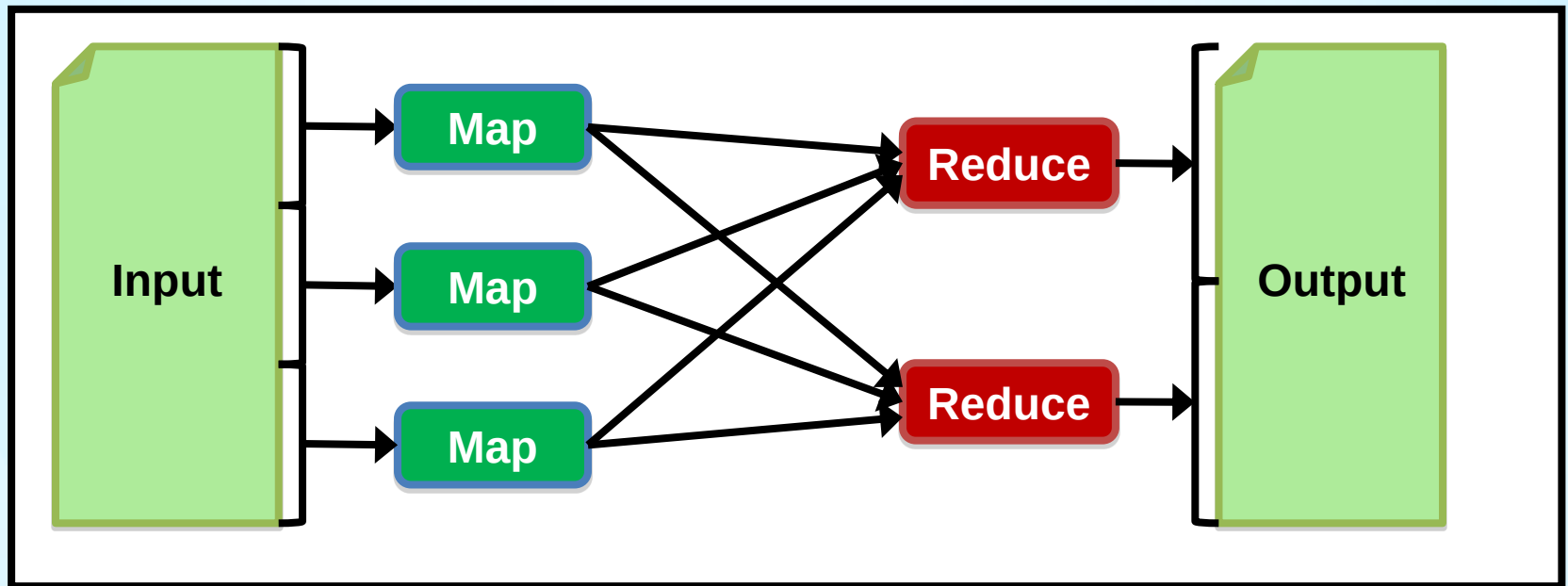
- **Coherency model:**

- Describes data visibility
- Current block being written may not be visible to other readers

Spark

Motivation

- **MapReduce** and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters.



Contd...

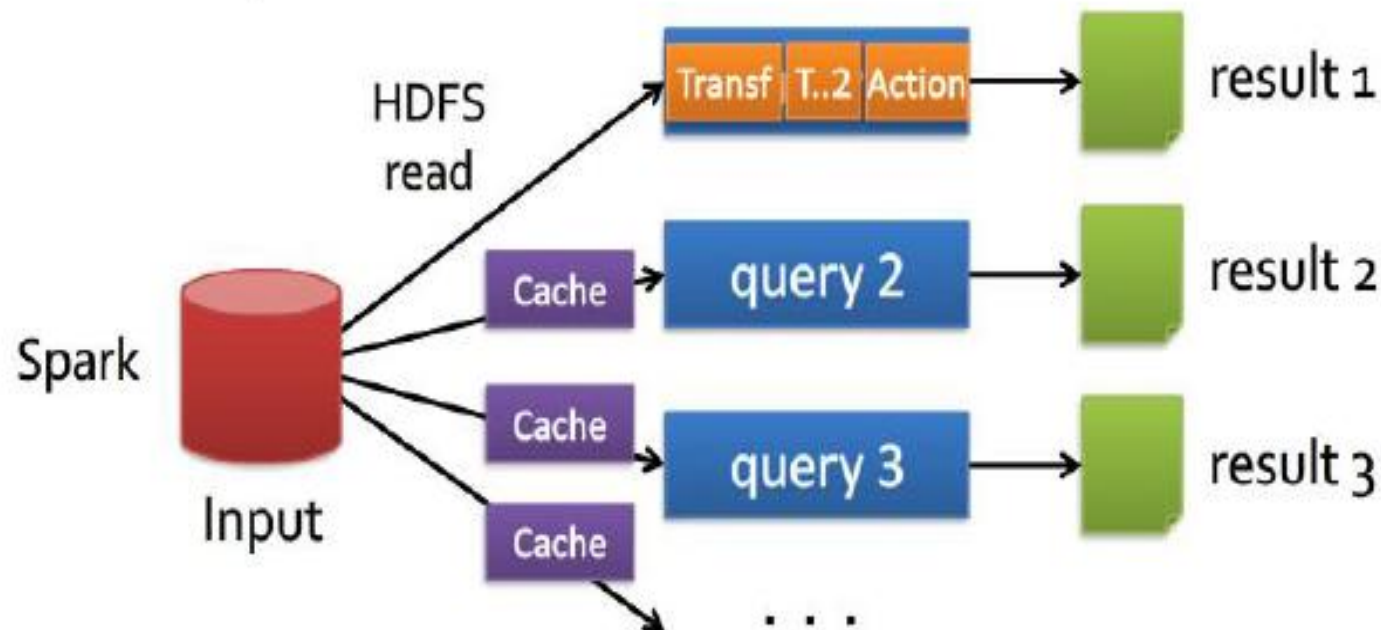
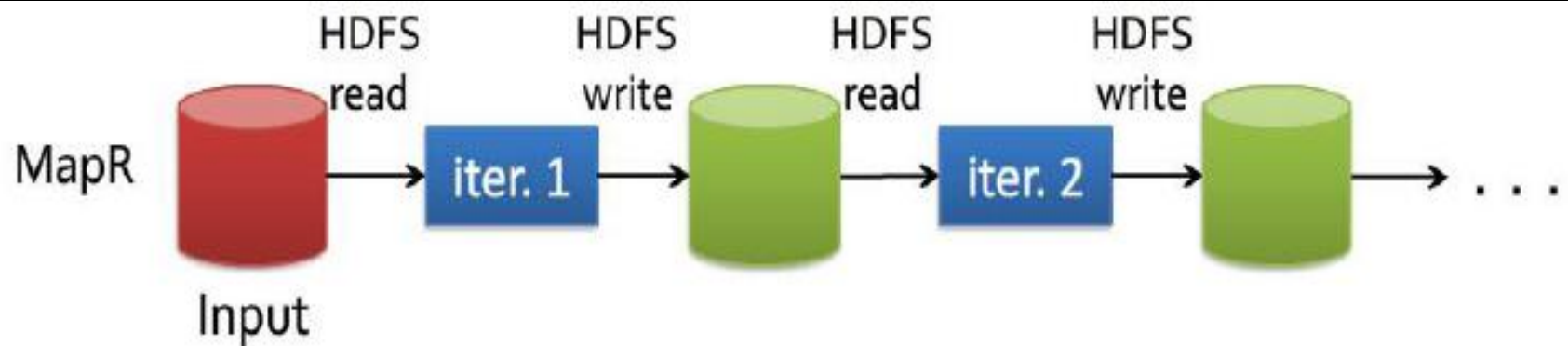
- However, most of these systems are built around an **acyclic data flow model** that is not suitable for other popular applications.
- In this part of the lecture, we will focus on one such class of applications, that **reuse a working set of data across multiple parallel operations**.
- This includes many **iterative machine learning algorithms**, as well as **interactive data analysis tools**.
- A new framework called **Spark** supports such applications **while retaining the scalability and fault tolerance** of MapReduce.

Contd...

- To achieve these goals, Spark introduces an abstraction called **resilient distributed datasets (RDDs)**.
- An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.
- **Spark can outperform Hadoop by 10x in iterative machine learning jobs**, and can be used to interactively query a 39 GB dataset with sub-second response time.

Difference Between Hadoop MapReduce vs. Apache Spark

Hadoop MapReduce	Apache Spark
Fast	100x faster than MapReduce
Batch Processing	Real-time Processing
Stores Data on Disk	Stores Data in Memory
Written in Java	Written in Scala



I/O and serialization can take **90%** of the time

Introduction

- A **new model of cluster computing** has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide **locality-aware scheduling, fault tolerance, and load balancing**.
- MapReduce pioneered this model, while systems like **Dryad** and **Map-Reduce-Merge** generalized the types of data flows supported.
- These systems achieve their **scalability and fault tolerance** by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators.
- This allows the underlying system to manage scheduling and to react to faults without user intervention.

Contd...

- While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows.
- Here, we focus on one such class of applications, that reuse a working set of data across multiple parallel operations. This includes two use cases where we have seen Hadoop users report that MapReduce is deficient:
 - (i) **Iterative jobs:** Many common machine learning algorithms **apply a function repeatedly to the same dataset** to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a **MapReduce/Dryad job**, each job must reload the data from disk, incurring a significant performance penalty.

Contd...

- (ii) **Interactive analytics:** Hadoop is often **used to run ad-hoc exploratory queries on large datasets**, through SQL interfaces such as Pig and Hive.
- Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly.
 - However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.
 - A **new cluster computing framework called Spark** supports applications with working sets while providing similar **scalability and fault tolerance properties** to MapReduce.

Contd...

- The main abstraction in Spark is that of a **resilient distributed dataset (RDD)**, which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.
- **Users can explicitly cache an RDD in memory across machines** and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition.
- Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and found well-suited for a variety of applications.

Contd...

- **Spark is implemented in Scala**, a statically typed high-level programming language for the **Java VM**, and exposes a functional programming interface similar to **DryadLINQ**.
- In addition, Spark can be used interactively from a modified version of the **Scala interpreter**, **which allows the user to define RDDs, functions, variables and classes** and use them in parallel operations on a cluster.
- Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.
- Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

Programming Model

- To use Spark, developers write a **driver program** that implements the high-level control flow of their application and launches various operations in parallel.
- Spark provides two main abstractions for parallel programming:
- **Resilient distributed datasets** and **parallel operations** on these datasets (invoked by passing a function to apply on a dataset).
- In addition, Spark supports two restricted types of shared variables that can be used in functions running on the cluster.

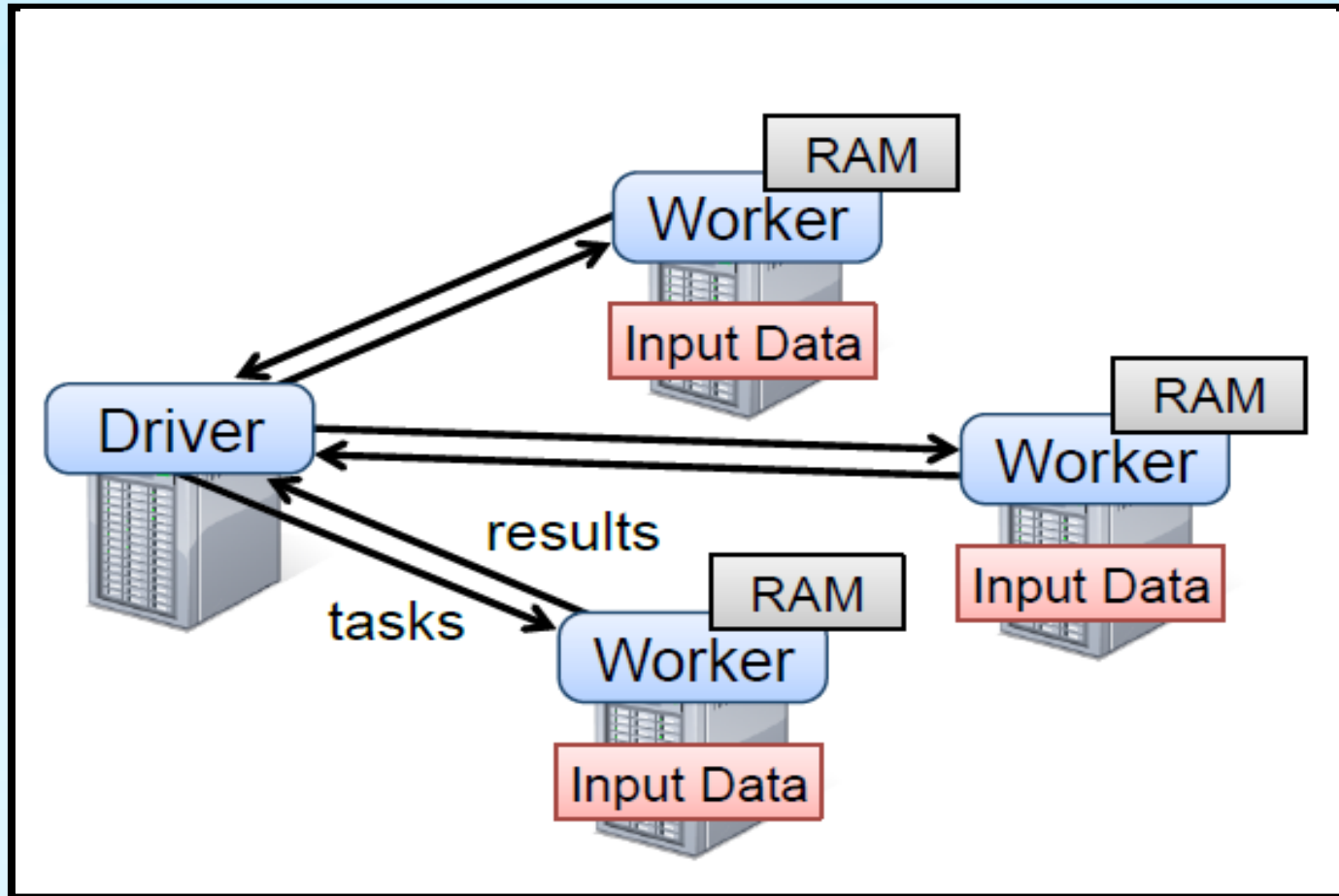
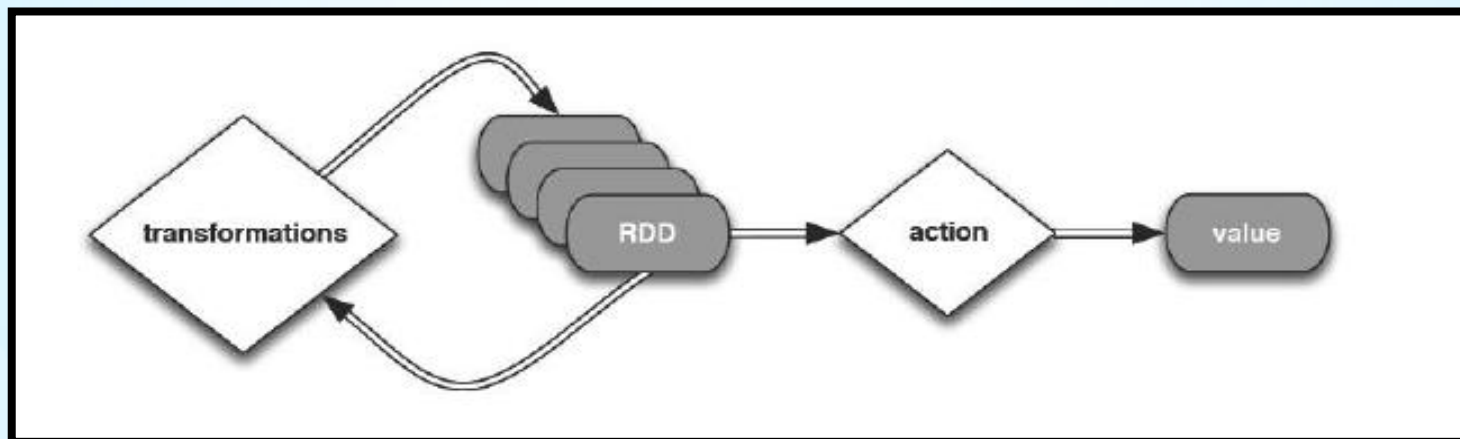


Figure: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

Resilient Distributed Datasets (RDDs)

- A **resilient distributed dataset (RDD)** is a **read-only collection** of objects partitioned across a set of machines that can be rebuilt if a partition is lost.
- The elements of an RDD need not exist in physical storage; instead, a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage.
- This means that RDDs can always be reconstructed if nodes fail.



Contd...

- **In Spark, each RDD is represented by a Scala object.** Spark lets programmers construct RDDs in four ways:
- **From a file in a shared file system**, such as the Hadoop Distributed File System (HDFS).
- **By “parallelizing” a Scala collection (e.g., an array)** in the driver program, which means dividing it into a number of slices that will be sent to multiple nodes.
- **By transforming an existing RDD.** A dataset with elements of **type A** can be transformed into a dataset with elements of **type B** using an operation called **flatMap**, which passes each element through a user-provided function of type $A \Rightarrow \text{List}[B]$.
- **Other transformations can be expressed using flatMap**, including map (pass elements through a function of type $A \Rightarrow B$ and filter (pick elements matching a predicate).

Contd...

- By changing the ***persistence*** of an existing RDD. By default, RDDs are lazy and ephemeral.
- That is, partitions of a dataset are materialized on demand when they are used in a parallel operation (e.g., by passing a block of a file through a map function), and are discarded from memory after use. However, a user can alter the persistence of an RDD through two actions:
- The ***cache action*** leaves the dataset lazy, but hints that it should be kept in memory after the first time it is computed, because it will be reused.
- The ***save action*** evaluates the dataset and writes it to a distributed filesystem such as HDFS. The saved version is used in future operations on it.

Contd...

- If there is not enough memory in the cluster to cache all partitions of a dataset, Spark will recompute them when they are used.
- Spark programs keep working (at reduced performance) if nodes fail or if a dataset is too big. This idea is loosely analogous to *virtual memory*.
- Spark can also be extended to support other levels of persistence (e.g., in-memory replication across multiple nodes).
- The goal is to let users trade off between the **cost of storing** an RDD, the **speed of accessing** it, the **probability of losing part** of it, and the **cost of recomputing** it.

Parallel Operations

- Several parallel operations can be performed on RDDs:
- **Reduce:** Combines dataset elements using an associative function to produce a result at the driver program.
- **Collect:** Sends all elements of the dataset to the driver program. For example, an easy way to update an array in parallel is to parallelize, map and collect the array.
- **Foreach:** Passes each element through a user provided function. This is only done for the side effects of the function.

Shared Variables

- Programmers invoke operations like *map, filter and reduce* by passing closures (functions) to Spark.
- As is typical in functional programming, these closures can refer to variables in the scope where they are created.
- Normally, when Spark runs a closure on a worker node, these variables are copied to the worker.
- However, Spark also lets programmers create two restricted types of shared variables to support two simple but common usage patterns such as **Broadcast variables and Accumulators**.

Broadcast variables

- **Broadcast variables:** If a large read-only piece of data (e.g., a **lookup table**) is used in multiple parallel operations, it is preferable to distribute it to the workers only once instead of packaging it with every closure.
- Spark lets the programmer create a **“broadcast variable”** object that wraps the value and ensures that it is only copied to each worker once.

Accumulators

- **Accumulators:** These are variables that workers can only “add” to using an **associative operation**, and that only the driver can read.
- They can be used to implement counters as in MapReduce and to provide a more imperative syntax for parallel sums.
- Accumulators can be defined for any type that has an “add” operation and a “zero” value.
- Due to their “add-only” semantics, they are easy to make fault-tolerant.

RDD Operations

Transformations (define a new RDD)

map
filter
sample
union
groupByKey
reduceByKey
join
cache
...

Parallel operations (Actions) (return a result to driver)

reduce
collect
count
save
lookupKey
...

Transformations

Transformation	Description
map (func)	Return a new distributed dataset formed by passing each element of the source through a function func
filter (func)	Return a new dataset formed by selecting those element of the source on which func returns true
flatMap (func)	Similar to map, but each input can be mapped to 0 or more output items (so func should return a Seq rather than a single item)
sample (withReplacement, fraction, seed)	Sample a fraction of the data, with or without replacements, using a given random number generator seed
union (otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument
distinct ([numtasks]))	Return a new dataset that contains the distinct elements of the source dataset

Contd...

Transformation	Description
groupByKey ([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
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
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
join (otherDataset, [numTasks])	When called on a dataset of type (K, V) and (K, W) returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
cogroup (otherDataset, [numTasks])	When called on a dataset of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples-also called groupWith
cartesian (otherDataset)	When called on a dataset of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

Actions

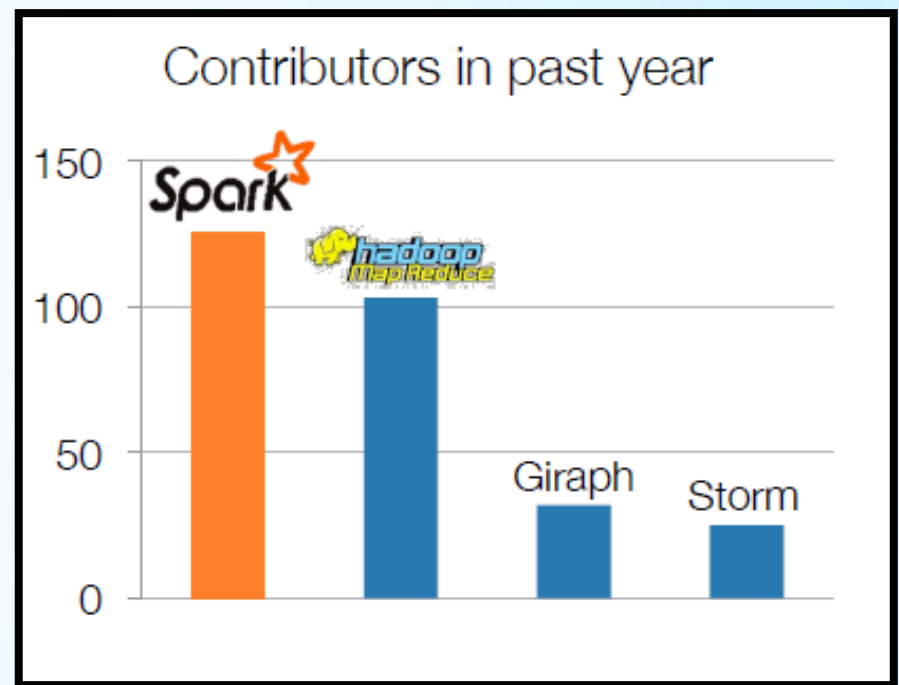
Action	Description
reduce (func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect ()	Return all the elements of the dataset as an array at the driver program- 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
first ()	Return the first element of the dataset-similar to take (1)
take (n)	Return an array with the first n elements of the dataset -currently not executed in parallel, instead the driver program computes all the elements
takeSample (withReplacement, fraction, seed)	Return an array with a random sample of num elements of the dataset, with or without replacement, using the given random number generator seed.

Contd...

Action	Description
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, Spark will call toString on each element to convert it to a line of text in the file
saveAsSequenceFile (path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem. HDFS or any other Hadoop-supported file system, Only available on RDDs of key-value pairs that either implement, Hadoop's writable interface or are implicitly Hadoop's Writable interface or are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
countByKey ()	Only available of RDD of type (K, V) Returns a 'Map' of (K, Int) pairs with the count of each key.
foreach (func)	Run a function func on each element of the dataset- usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Spark Community

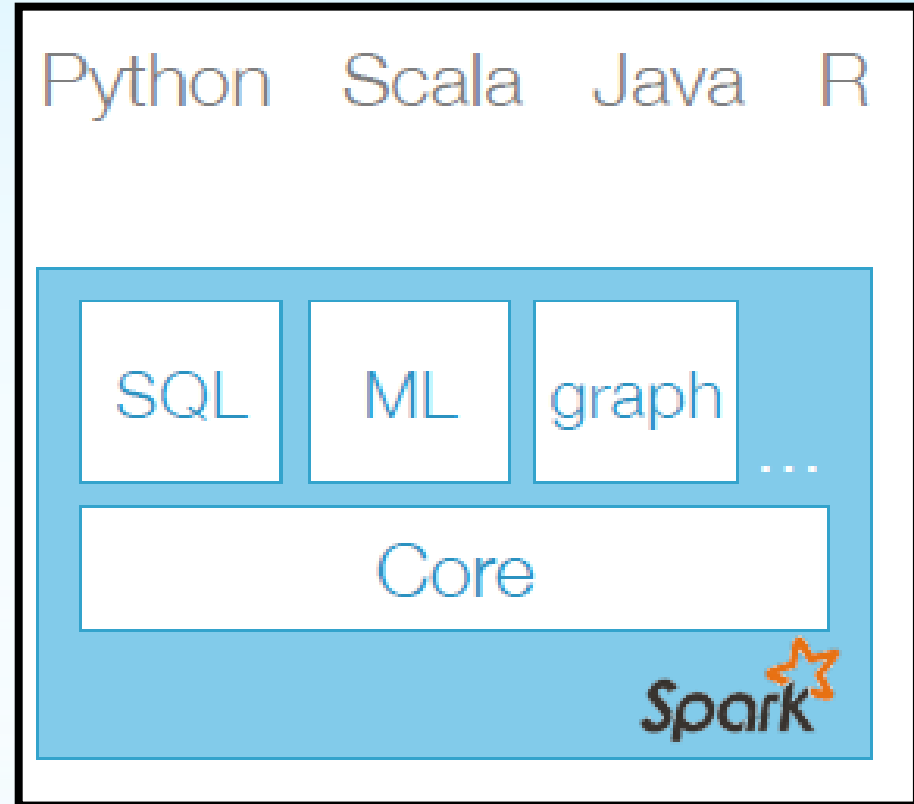
- Most active open source community in big data
- 200+ developers, 50+ companies contributing



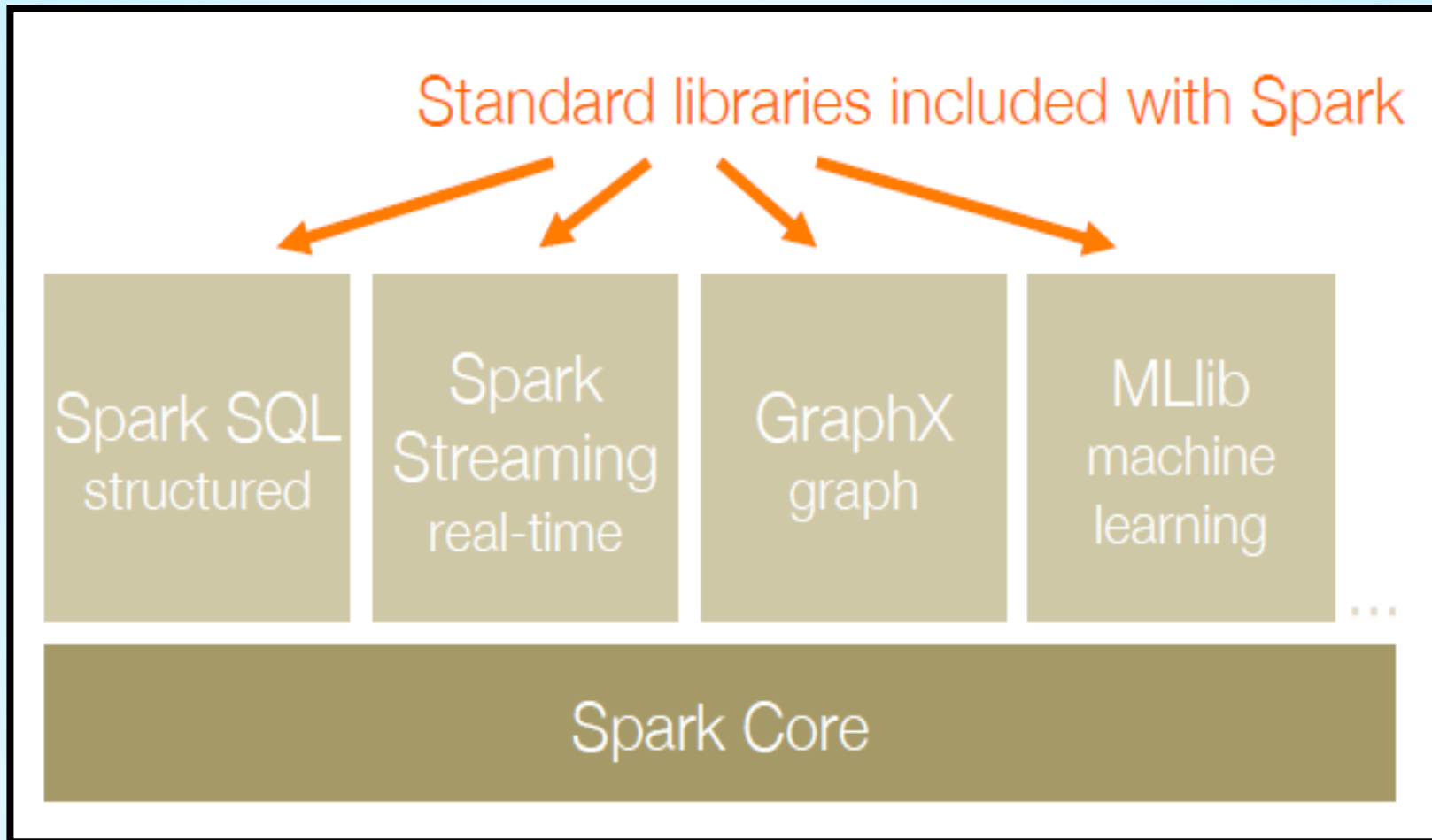
Built-in Libraries

Standard Library for Big Data

- Big data apps lack libraries“ of common algorithms
- Spark’s generality + support“ for multiple languages make it“ suitable to offer this
- Much of future activity will be in these libraries



A General Platform



Machine Learning Library (MLlib)

MLlib algorithms:

(i) Classification: logistic regression, linear SVM, “naïve Bayes, classification tree

(ii) Regression: generalized linear models (GLMs), regression tree

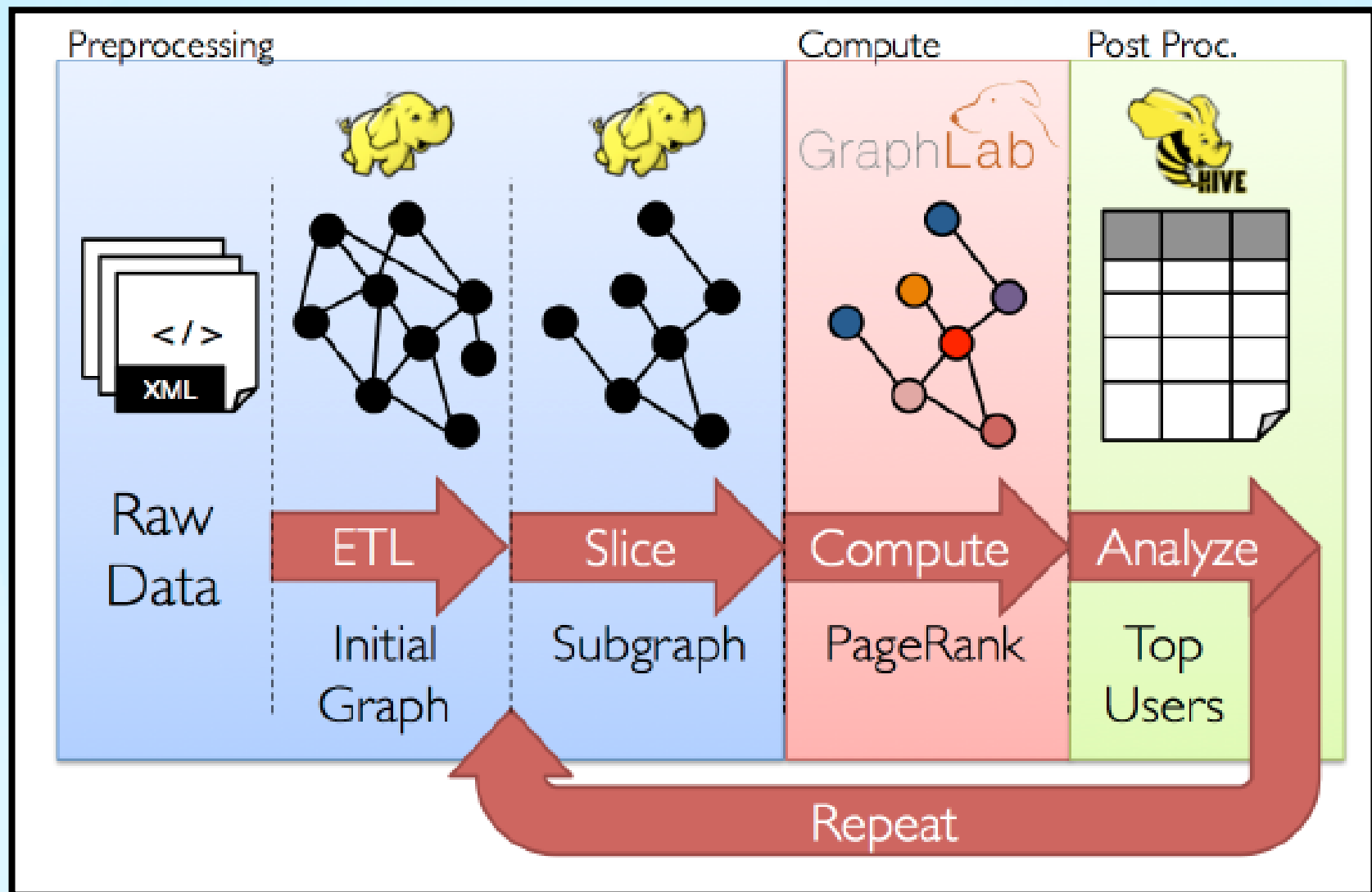
(iii) Collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)

(iv) Clustering: k-means

(v) Decomposition: SVD, PCA

(vi) Optimization: stochastic gradient descent, L-BFGS

GraphX



GraphX

- General graph processing library
- Build graph using RDDs of nodes and edges
- Large library of graph algorithms with composable steps

GraphX Algorithms

(i) Collaborative Filtering

Alternating Least Squares
Stochastic Gradient Descent
Tensor Factorization

(ii) Structured Prediction

Loopy Belief Propagation
Max-Product Linear Programs
Gibbs Sampling

(iii) Semi-supervised ML

Graph SSL
CoEM

(iv) Community Detection

Triangle-Counting
K-core Decomposition
K-Truss

(v) Graph Analytics

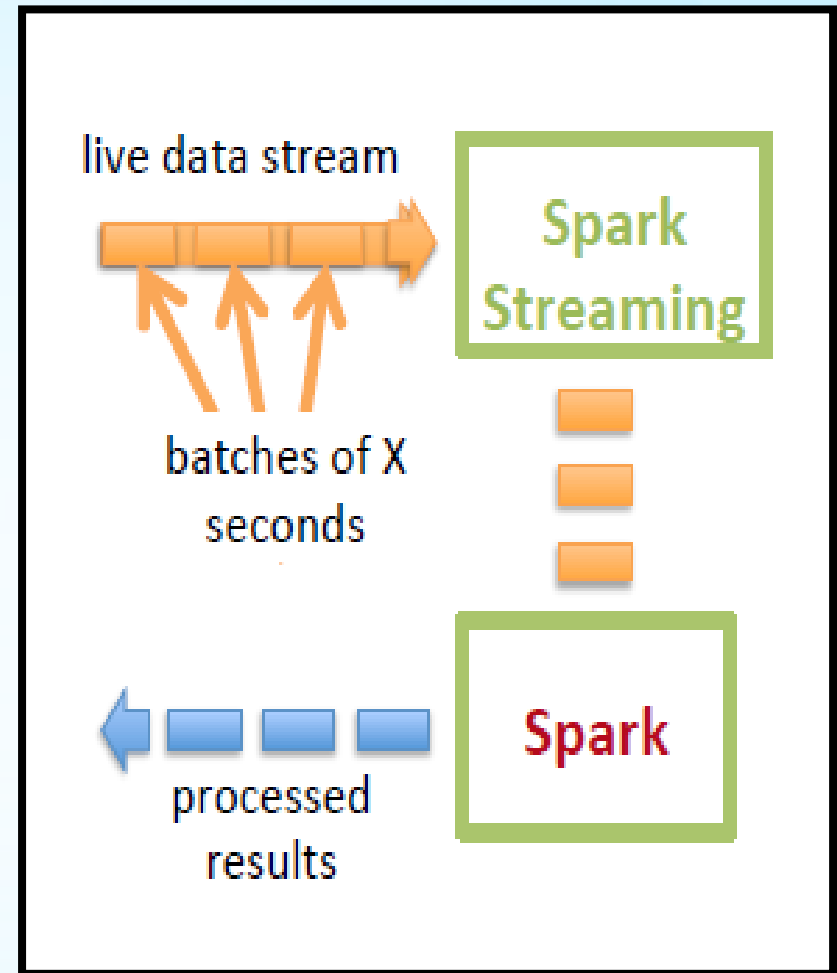
PageRank
Personalized PageRank
Shortest Path
Graph Coloring

(vi) Classification

Neural Networks

Spark Streaming

- Large scale streaming computation
- Ensure exactly one semantics
- Integrated with Spark → unifies batch, interactive, and streaming computations!



Spark SQL

Enables loading & querying structured data in Spark

From Hive:

```
c = HiveContext(sc)
rows = c.sql("select text, year from hivetable")
rows.filter(lambda r: r.year > 2013).collect()
```

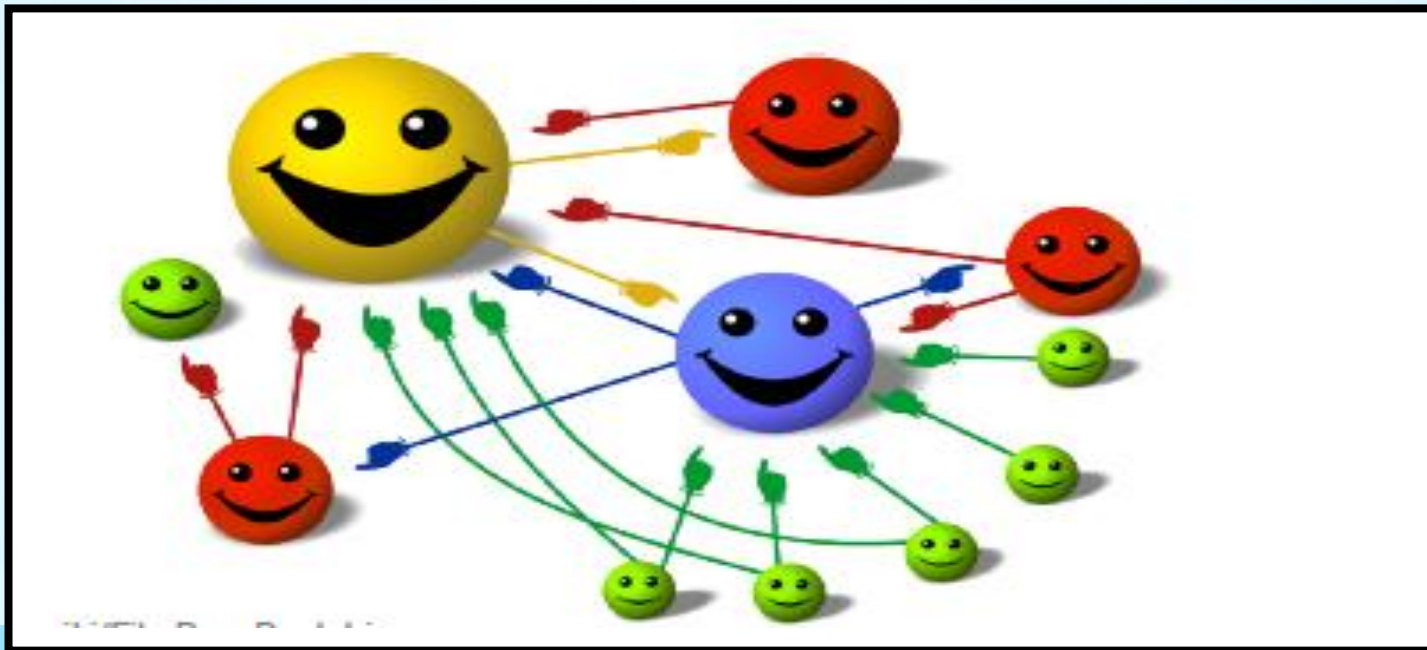
From JSON:

```
c.jsonFile("tweets.json").registerAsTable("tweets")
c.sql("select text, user.name from tweets")
```

Examples

Example 1: PageRank

- Give pages ranks (scores) based on links to them
- Links from many pages  high rank
- Links from a high-rank page  high rank

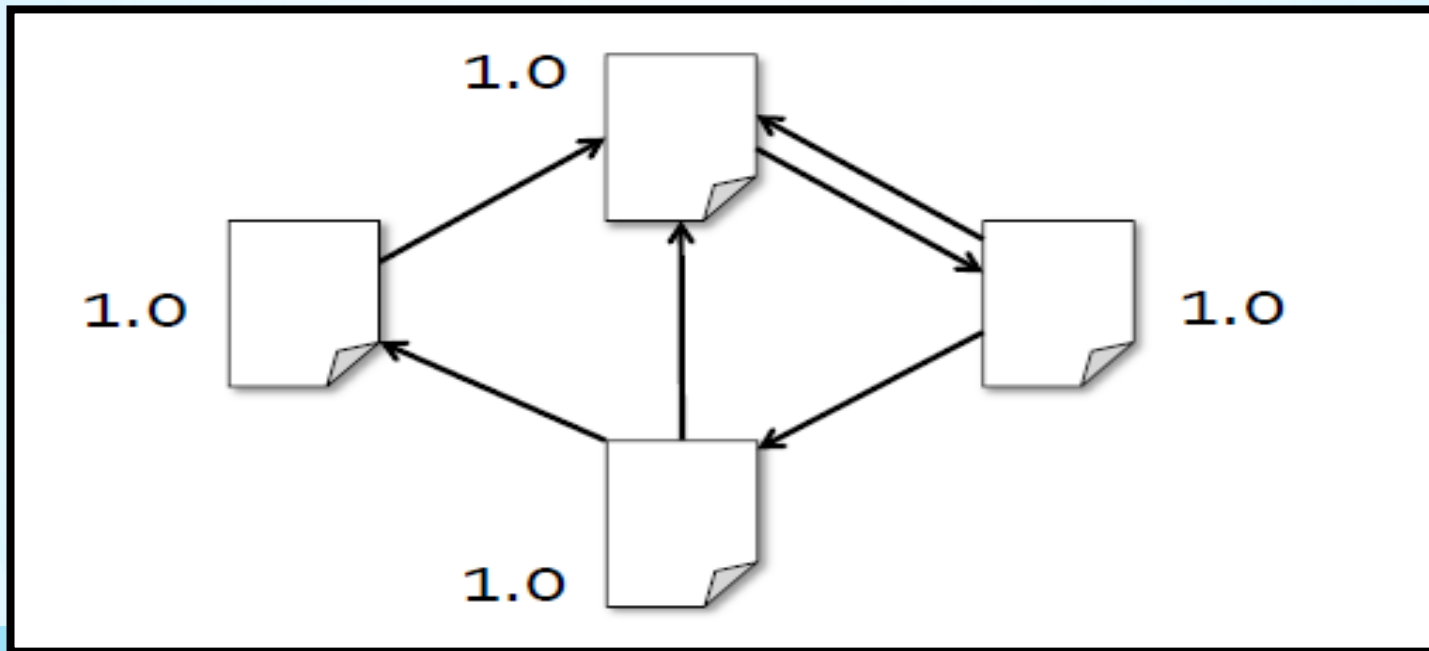


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Step-3 Set each page's rank to $0.15 + 0.85 \times \text{contributions}$

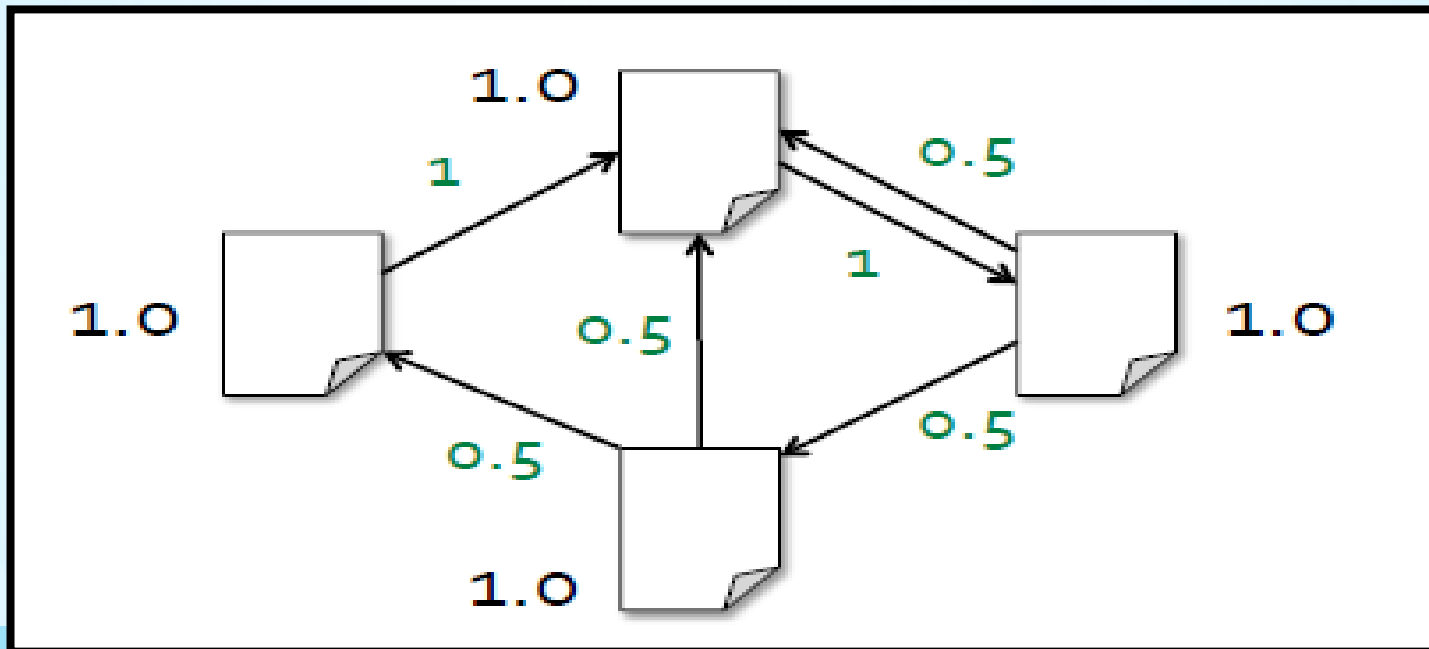


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Step-3 Set each page's rank to $0.15 + 0.85 \times \text{contributions}$

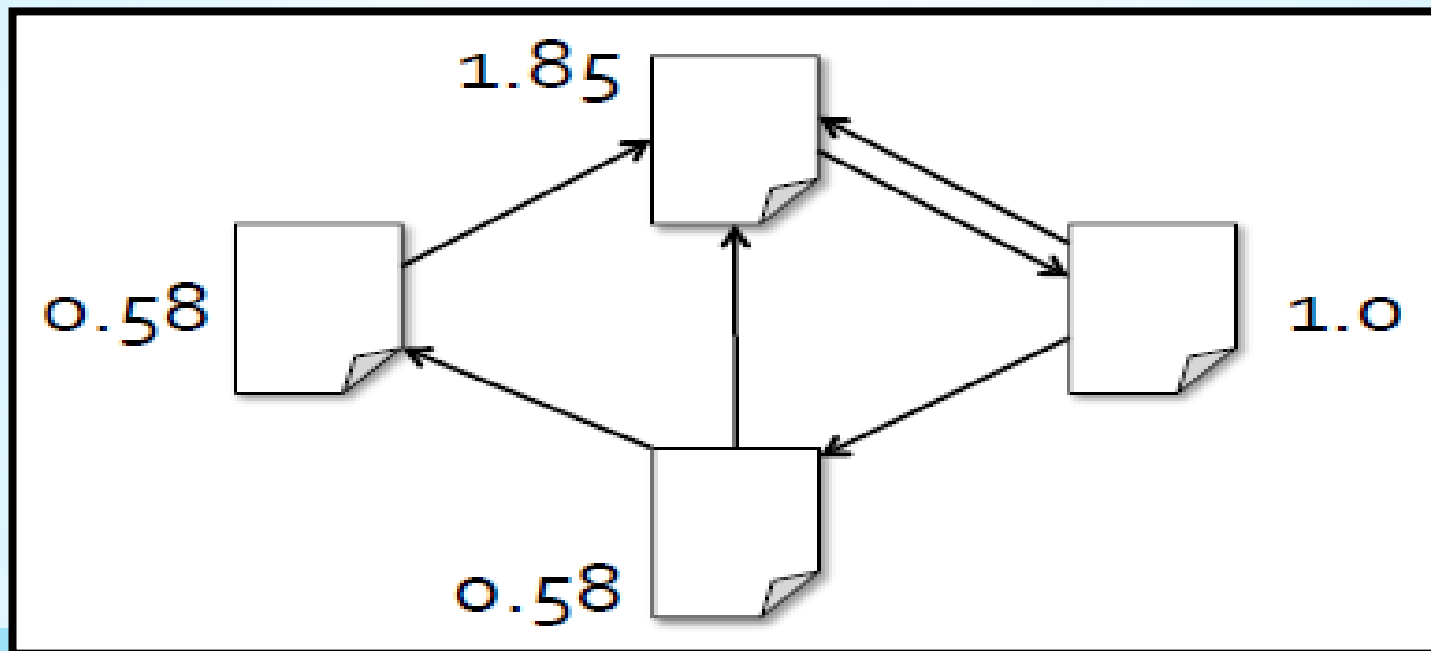


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Step-3 Set each page's rank to $0.15 + 0.85 \times \text{contributions}$

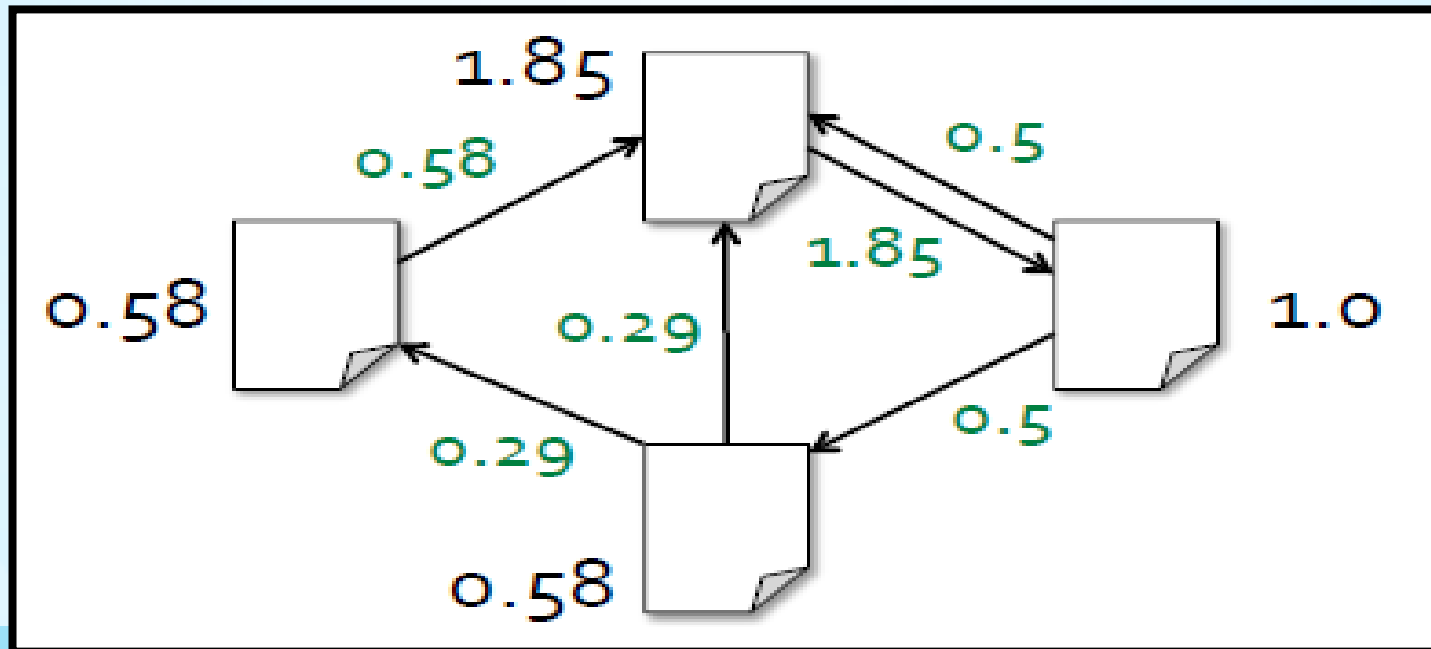


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Step-3 Set each page's rank to $0.15 + 0.85 \times \text{contributions}$

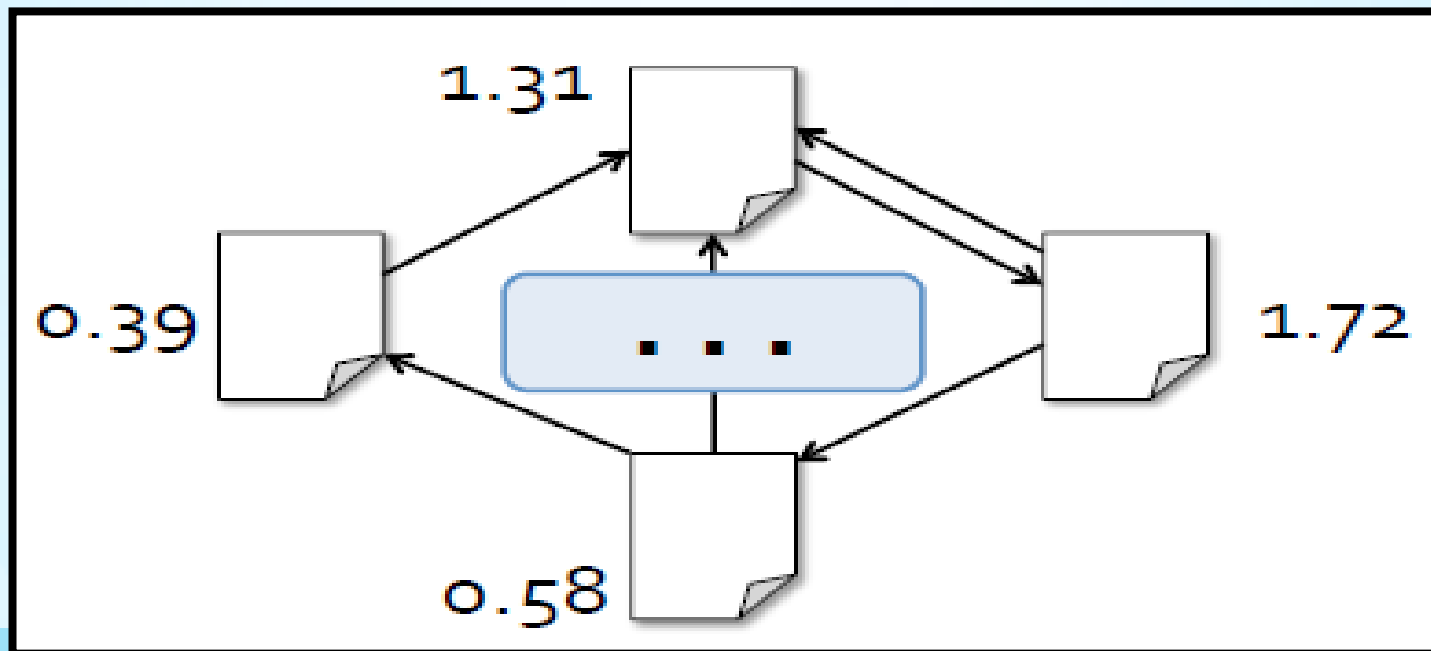


Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Step-3 Set each page's rank to $0.15 + 0.85 \times \text{contributions}$



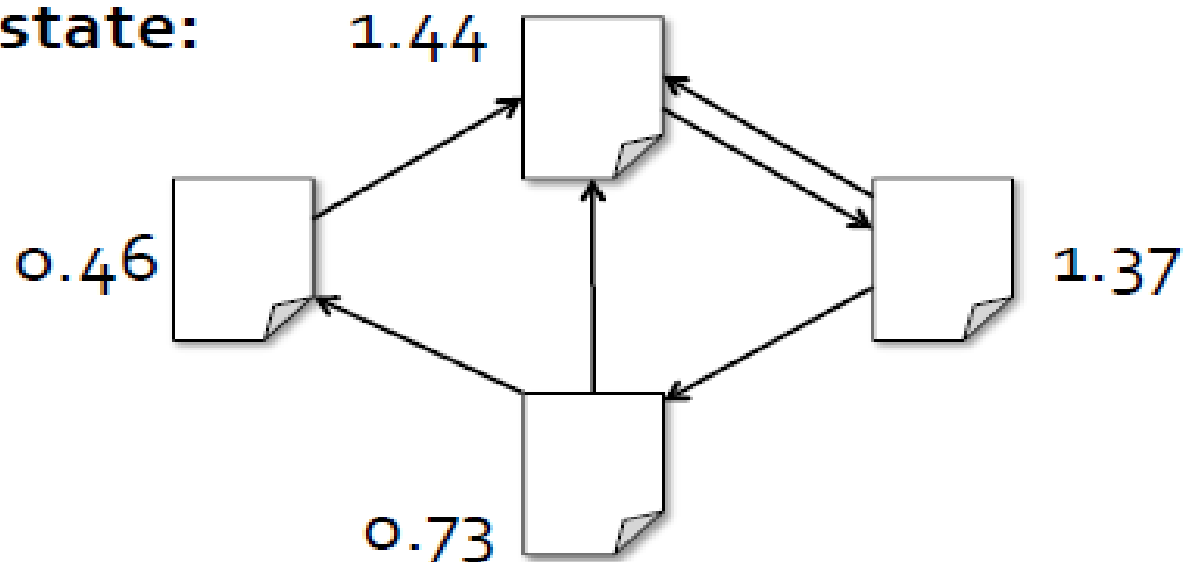
Algorithm

Step-1 Start each page at a rank of 1

Step-2 On each iteration, have page p contribute $\text{rank}_p / |\text{neighbors}_p|$ to its neighbors

Step-3 Set each page's rank to $0.15 + 0.85 \times \text{contributions}$

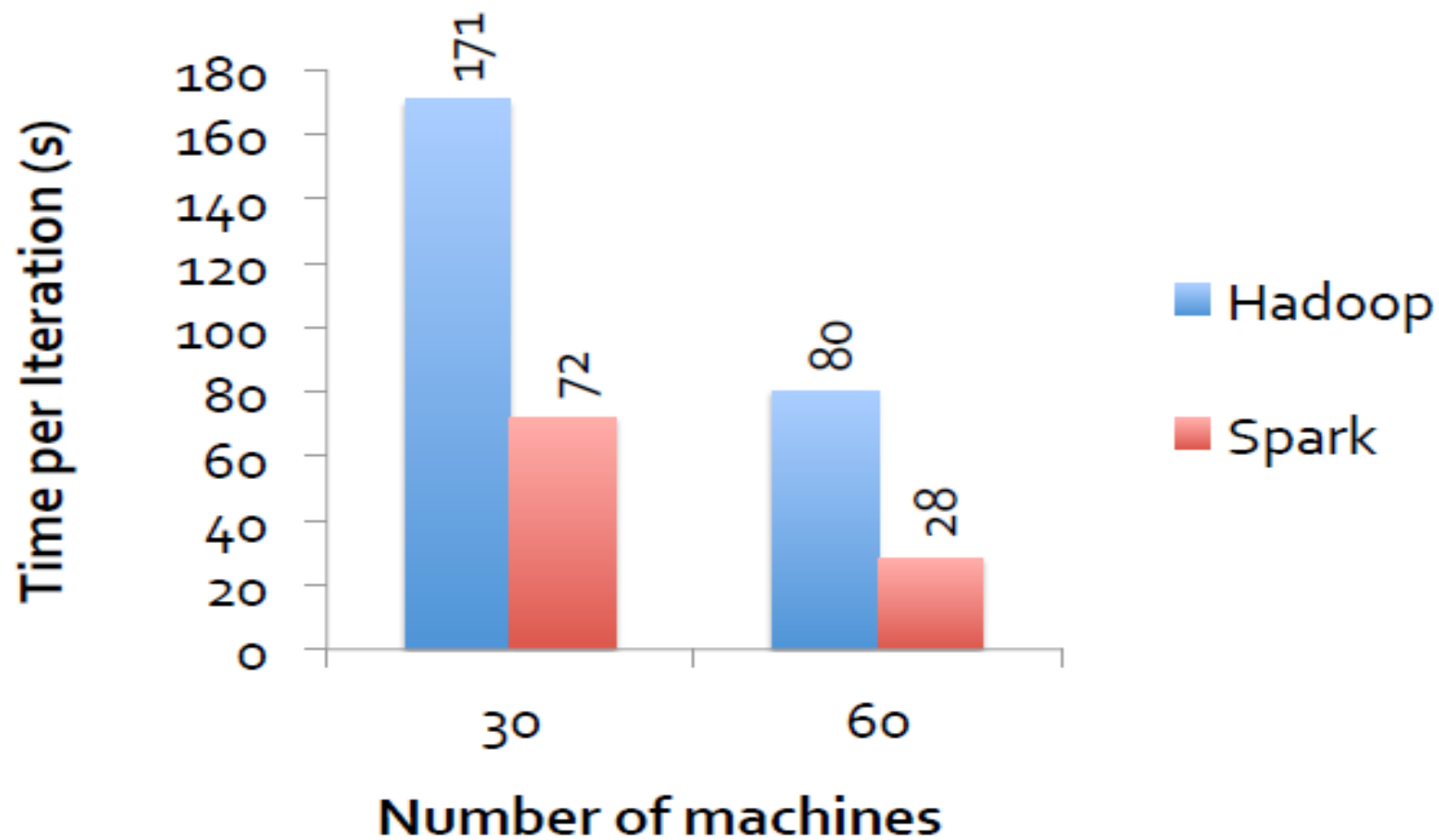
Final state:



Spark Program

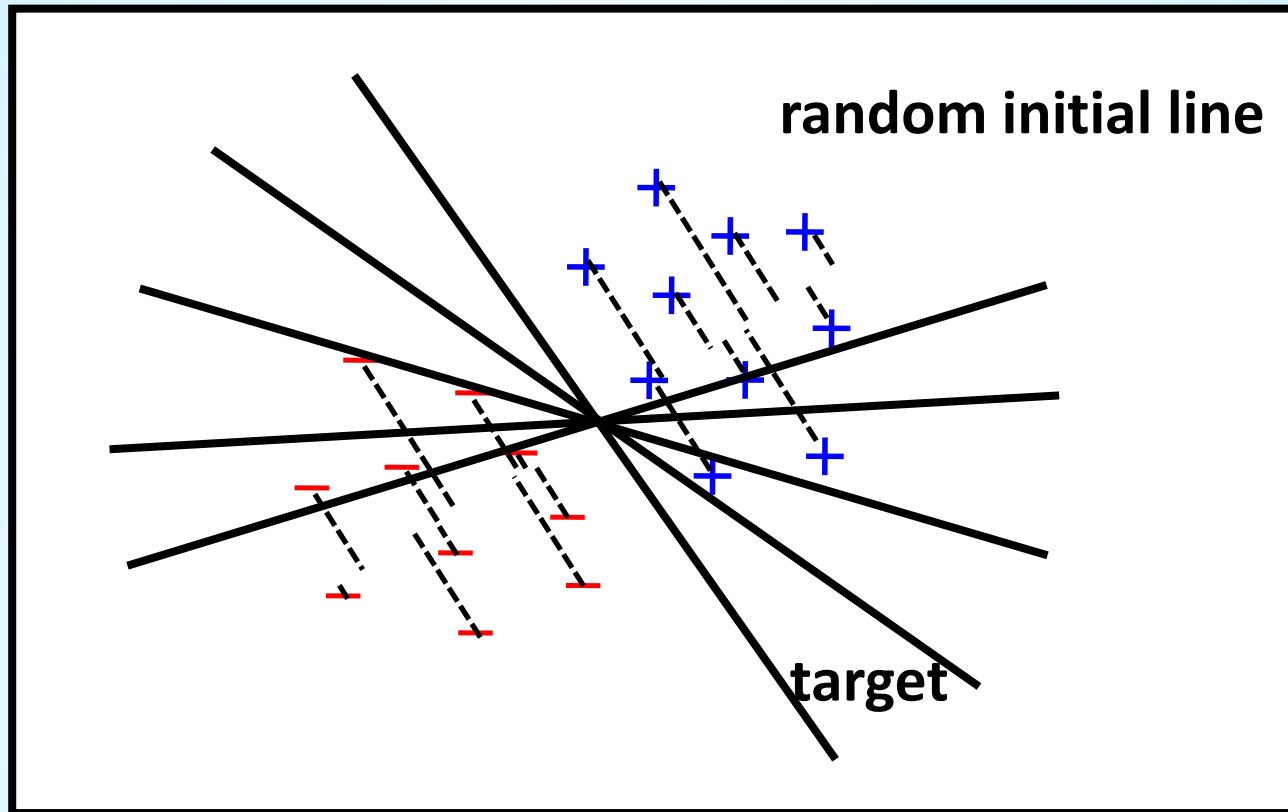
```
val links = // RDD of (url, neighbors) pairs
var ranks = // RDD of (url, rank) pairs
for (i <- 1 to ITERATIONS) {
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  ranks = contribs.reduceByKey(_ + _)
                      .mapValues(0.15 + 0.85 * _)
}
ranks.saveAsTextFile(...)
```

PageRank Performance



Example 2: Logistic Regression

- **Goal:** find best line separating two sets of points



Logistic Regression Code

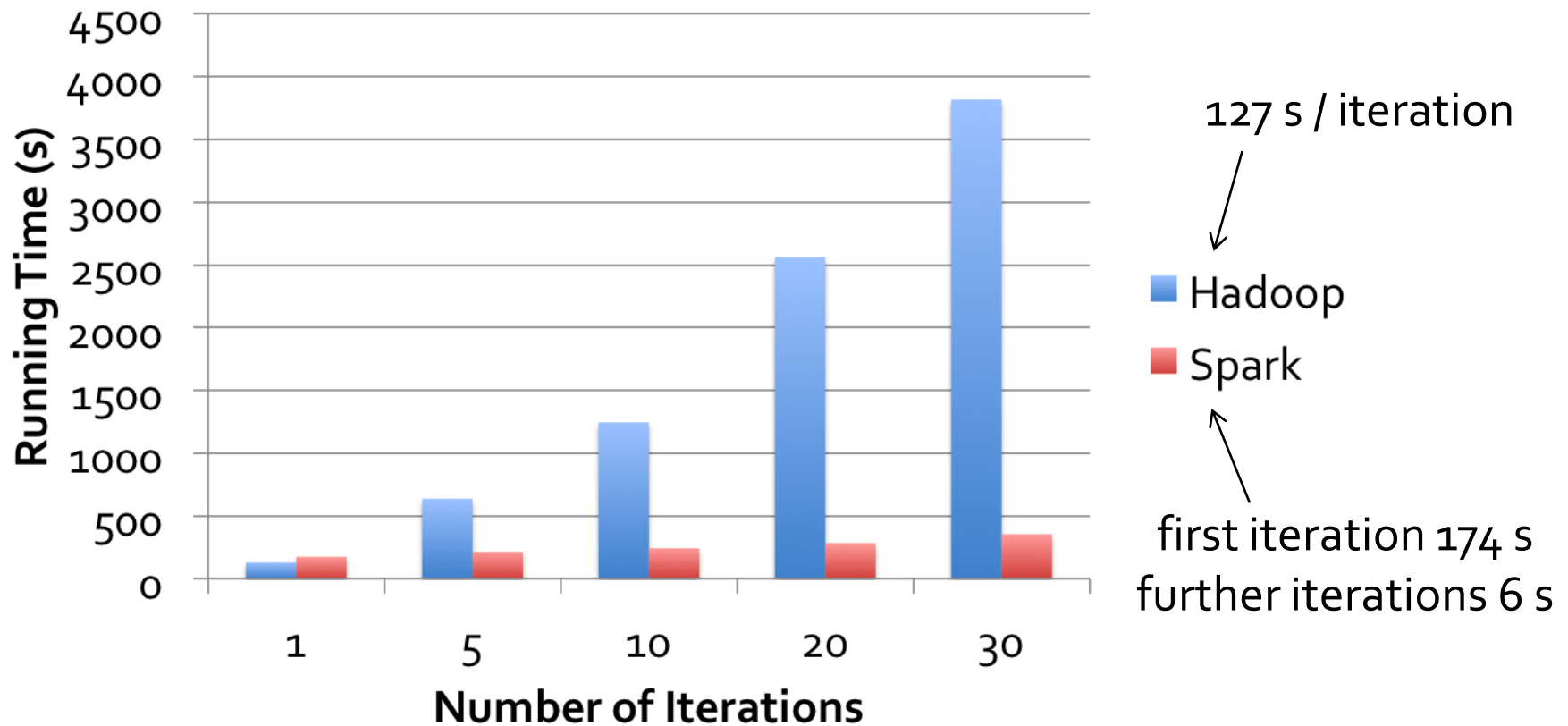
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Logistic Regression Performance



Example 3: MapReduce

- MapReduce data flow can be expressed using RDD transformations

```
res = data.flatMap(rec => myMapFunc(rec))  
           .groupByKey()  
           .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))  
           .reduceByKey(myCombiner)  
           .map((key, val) => myReduceFunc(key, val))
```

Example 4: WordCount

Definition: Count how often each word appears in a collection of text documents

This simple program provides a good test case for parallel processing, since it:

- Requires a minimal amount of code
- Demonstrates use of both symbolic and numeric values
- Isn't many steps away from search indexing
- Serves as a “Hello World” for Big Data apps

A distributed computing framework that can run WordCount **efficiently in parallel at scale** can likely handle much larger and more interesting computer problems.

WordCount Program

Scala:

```
val f = sc.textFile ( "README.md" )  
val wc = f.flatMap (l => l.split(" ")).map (word=> (word, 1)).reduceByKey(_ + _ )  
wc.saveAsTextFile ( "wc_out" )
```

Python:

```
from operator import add  
f = sc.textFile ( "README.md" )  
wc = f.flatMap (lamda x: x.split(' ' ) ).map (lamda x: (x, 1)).reduceByKey(add )  
wc.saveAsTextFile ( "wc_out" )
```

Other Spark Applications

- i. Twitter spam classification
- ii. EM algorithm for traffic prediction
- iii. K-means clustering
- iv. Alternating Least Squares matrix factorization
- v. In-memory OLAP aggregation on Hive data
- vi. SQL on Spark

Reading Material

- Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica

“Spark: Cluster Computing with Working Sets”

- Matei Zaharia, Mosharaf Chowdhury et al.

“Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”

- Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler

“The Hadoop Distributed File System”

Conclusion

- Spark provides **three simple data abstractions for programming clusters: resilient distributed datasets (RDDs), and two restricted types of shared variables: broadcast variables and accumulators.** While these abstractions are limited, It is found that they are powerful enough to express several applications that pose challenges for existing cluster computing frameworks, including iterative and interactive computations.
- Furthermore, It is believed that the core idea behind RDDs, of a dataset handle that has enough information to (re)construct the dataset from data available in reliable storage, may prove useful in developing other abstractions for programming clusters.
- In this lecture, we have discussed the **HDFS components, its architecture and framework of spark with its applications.**