BIG DATA PROCESSING WITH SPARK

SOURANGSHU BHATTACHARYA

CSE, IIT KHARAGPUR

WEB: HTTPS://CSE.IITKGP.AC.IN/~SOURANGSHU/

EMAIL: SOURANGSHU@CSE.IITKGP.AC.IN

BIG DATA PROCESSING

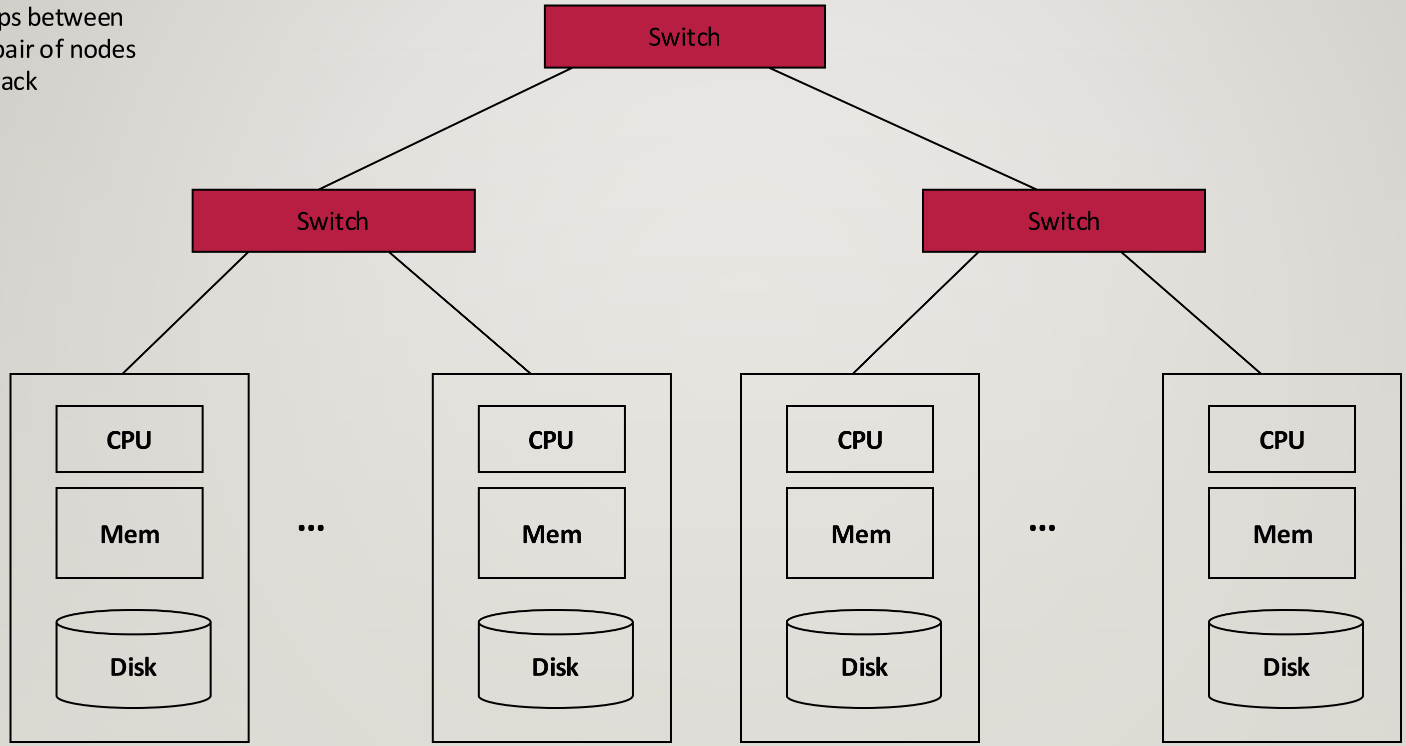
MOTIVATION: GOOGLE EXAMPLE

- 20+ billion web pages x 20KB = 400+ TB
- A computer reads 30-35 MB/sec from disk
  - ~4 months to read the data
- ~ 400 hard drives to store the data
- Takes even more to **do** something useful with the data!
- **Today, a standard architecture for such problems is used:**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

CLUSTER ARCHITECTURE

2-10 Gbps backbone between racks

1 Gbps between
any pair of nodes
in a rack



Each rack contains 16-64 nodes

LARGE-SCALE COMPUTING

- **Large-scale computing** for **data mining** problems on **commodity hardware**

- **Challenges:**
  - **How do you distribute computation?**
  - **How can we make it easy to write distributed programs?**
  - **Machines fail:**
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to loose 1/day
    - People estimated Google has ~1M machines
      - 1,000 machines fail every day!

BIG DATA CHALLENGES

- Scalability: processing should scale with increase in data.
- Fault Tolerance: function in presence of hardware failure
- Cost Effective: should run on commodity hardware
- Ease of use: programmers do not write additional code for communication, fault tolerance, etc.
- Flexibility: able to process unstructured data

- Solution: Map Reduce !

IDEA AND SOLUTION

- **Issue: Copying data over a network takes time**
- **Ideas:**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Map-reduce** addresses these problems
  - Elegant way to work with big data
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - Map-Reduce

WHAT IS HADOOP ?

- A scalable fault-tolerant distributed system for data storage and processing.

- Core Hadoop:
  - Hadoop Distributed File System (HDFS)
  - Hadoop YARN: Job Scheduling and Cluster Resource Management
  - Hadoop Map Reduce: Framework for distributed data processing.

- Open Source system with large community support.

  https://hadoop.apache.org/

WHAT IS MAP REDUCE ?

- Programming paradigm for seamlessly distributing a task across multiple servers.
- Proposed by Dean and Ghemawat, 2004.
- Consists of two developer created phases:
  - Map
  - Reduce
- In between Map and Reduce is the Shuffle and Sort phase.
- User is responsible for casting the algorithm into map – reduce framework.

PROGRAMMING MODEL: MAPREDUCE

Warm-up task:

- We have a huge text document
- Count the number of times each distinct word appears in the file
- **Sample application:**
  - Analyze web server logs to find popular URLs

TASK: WORD COUNT

Case 1:

- File too large for memory, but all <word, count> pairs fit in memory
- Use a hashmap.

Case 2:

- Count occurrences of words:
  - `words(doc.txt) | sort | uniq -c`
    - where **`words`** takes a file and outputs the words in it, one per a line
- Case 2 captures the essence of **MapReduce**

MAPREDUCE: OVERVIEW

- Both sequentially read and write a lot of data records
- **Map:**
  - Extract something you care about
  - Output is (key, value) pair
- **Group by key:** Sort and Shuffle
- **Reduce:**
  - Process records with the same key value: e.g. Aggregate, summarize, etc.
- Write the result

Outline stays the same, **Map** and **Reduce** change to fit the problem

MORE SPECIFICALLY

- **Input:** a set of key-value pairs
- Programmer specifies two methods:
  - **Map(k, v)** $\rightarrow$ <k', v'>*
    - Takes a key-value pair and outputs a set of key-value pairs
      - E.g., key is the filename, value is a single line in the file
    - There is one Map call for every *(k,v)* pair
  - **Reduce(k', <v'>*)** $\rightarrow$ <k', v''>*
    - **All values *v'* with same key *k'* are reduced together and processed in *v'* order**
    - There is one Reduce function call per unique key *k'*

MAPREDUCE: WORD COUNTING

Provided by the programmer (over MAP) | **Provided by the programmer** (over Reduce)

MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key

Reduce:
Collect all values belonging to the key and output

The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/mache partnership. '"The work we're doing now -- the robotics we're doing -- is what we're going to need ……………..…………..

Big document

(The, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
….

(key, value)

(crew, 1)
(crew, 1)
(space, 1)
(the, 1)
(the, 1)
(the, 1)
(shuttle, 1)
(recently, 1)
…

(key, value)

(crew, 2)
(space, 1)
(the, 3)
(shuttle, 1)
(recently, 1)
…

(key, value)

Only sequential reads

WORD COUNT USING MAPREDUCE

```
map(key, value):

// key: document name; value: text of the document

  for each word w in value:

      emit(w, 1)

reduce(key, values):
// key: a word; value: an iterator over
counts
     result = 0
     for each count v in values:
          result += v
     emit(key, result)
```

HADOOP MAP REDUCE

- Provides:
  - Automatic parallelization and Distribution
  - Fault Tolerance
  - Methods for interfacing with HDFS for colocation of computation and storage of output.
  - Status and Monitoring tools
  - API in Java
  - Ability to define the mapper and reducer in many languages through Hadoop streaming.

HDFS

WHAT'S HDFS

- HDFS is a distributed file system that is fault tolerant, scalable and extremely easy to expand.
- HDFS is the primary distributed storage for Hadoop applications.
- HDFS provides interfaces for applications to move themselves closer to data.
- HDFS is designed to 'just work', however a working knowledge helps in diagnostics and improvements.

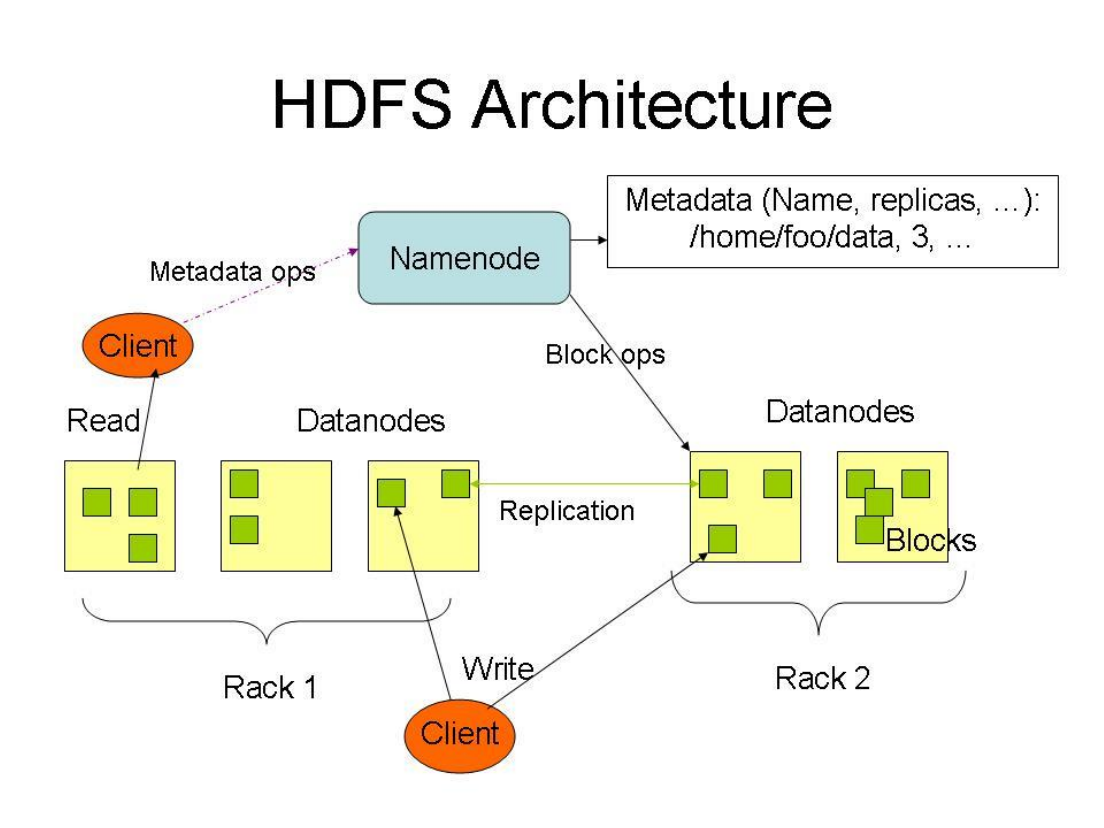
COMPONENTS OF HDFS

There are two (*and a half*) types of machines in a HDFS cluster

- NameNode :– is the heart of an HDFS filesystem, it maintains and manages the file system metadata. E.g; what blocks make up a file, and on which datanodes those blocks are stored.

- DataNode :- where HDFS stores the actual data, there are usually quite a few of these.

HDFS ARCHITECTURE



HDFS

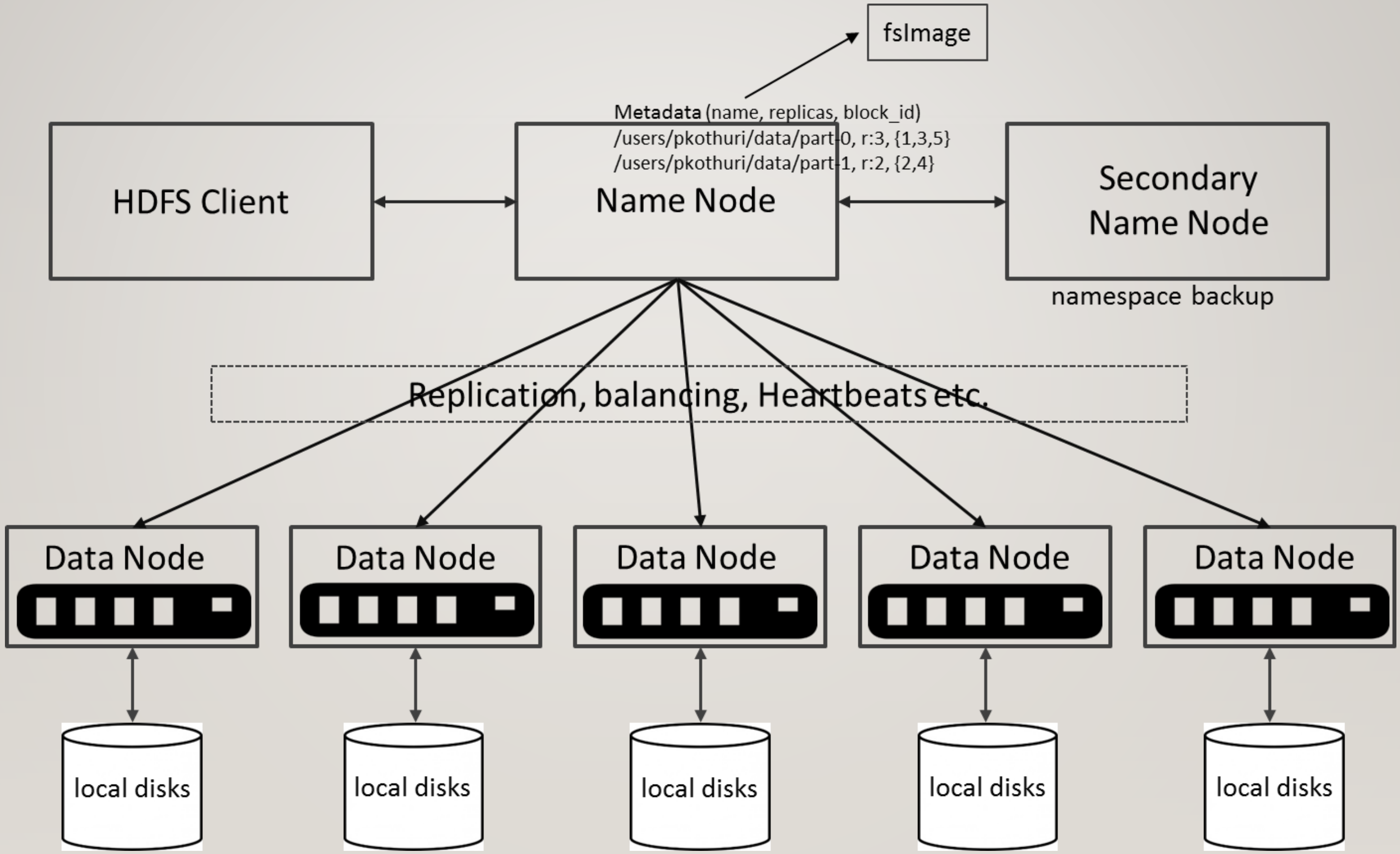
- Design Assumptions
  - Hardware failure is the norm.
  - Streaming data access.
  - Write once, read many times.
  - High throughput, not low latency.
  - Large files.

- Characteristics:
  - Performs best with modest number of large files
  - Optimized for streaming reads
  - Layer on top of native file system.

HDFS

- Data is organized into file and directories.
- Files are divided into blocks and distributed to nodes.
- Block placement is known at the time of read
  - Computation moved to same node.
- Replication is used for:
  - Speed
  - Fault tolerance
  - Self healing.

HDFS ARCHITECTURE



DATANODE

- **A Block Server**
  - – Stores data in the local file system (e.g. ext3)
  - – Stores meta-data of a block (e.g. CRC)
  - – Serves data and meta-data to Clients
- **Block Report**
  - – Periodically sends a report of all existing blocks to the NameNode
- **Facilitates Pipelining of Data**
  - – Forwards data to other specified DataNodes

NAMENODE METADATA

- **Meta-data in Memory**
  – The entire metadata is in main memory
  – No demand paging of meta-data
- **Types of Metadata**
  – List of files
  – List of Blocks for each file
  – List of DataNodes for each block
  – File attributes, e.g creation time, replication factor
- **A Transaction Log**
  – Records file creations, file deletions. etc

HDFS – USER COMMANDS (DFS)

List directory contents

```
hdfs dfs –ls
hdfs dfs -ls /
hdfs dfs -ls -R /var
```

Display the disk space used by files

```
hdfs dfs -du /hbase/data/hbase/namespace/
hdfs dfs -du -h /hbase/data/hbase/namespace/
hdfs dfs -du -s /hbase/data/hbase/namespace/
```

HDFS – USER COMMANDS (DFS)

Copy data to HDFS

```
hdfs dfs -mkdir tdata
hdfs dfs -ls
hdfs dfs -copyFromLocal tutorials/data/geneva.csv tdata
hdfs dfs -ls –R
```

Copy the file back to local filesystem

```
cd tutorials/data/
hdfs dfs –copyToLocal tdata/geneva.csv geneva.csv.hdfs
md5sum geneva.csv geneva.csv.hdfs
```

HDFS – USER COMMANDS (ACLS)

List acl for a file

```
hdfs dfs -getfacl tdata/geneva.csv
```

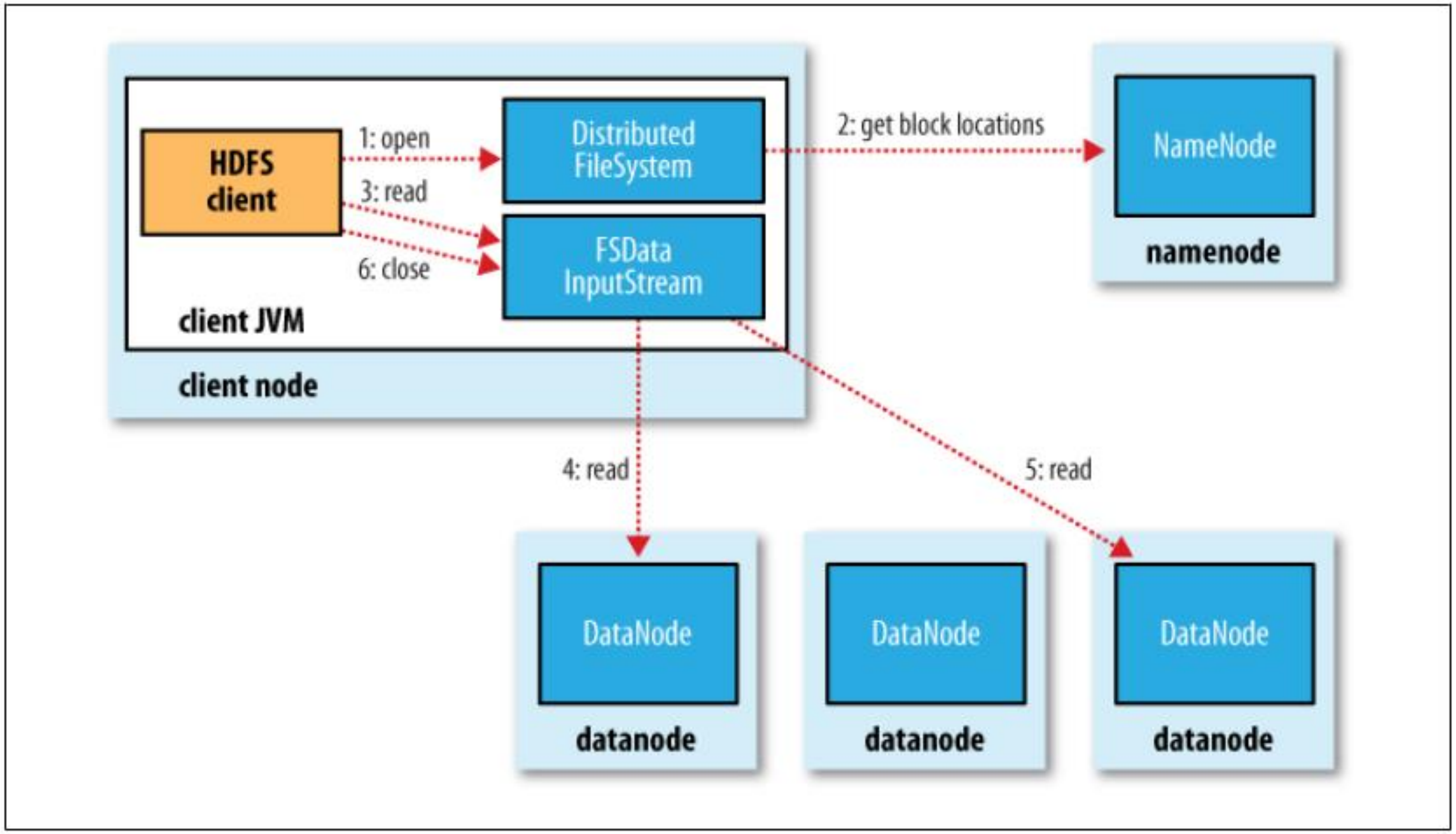
List the file statistics – (%r – replication factor)

```
hdfs dfs -stat "%r" tdata/geneva.csv
```

Write to hdfs reading from stdin

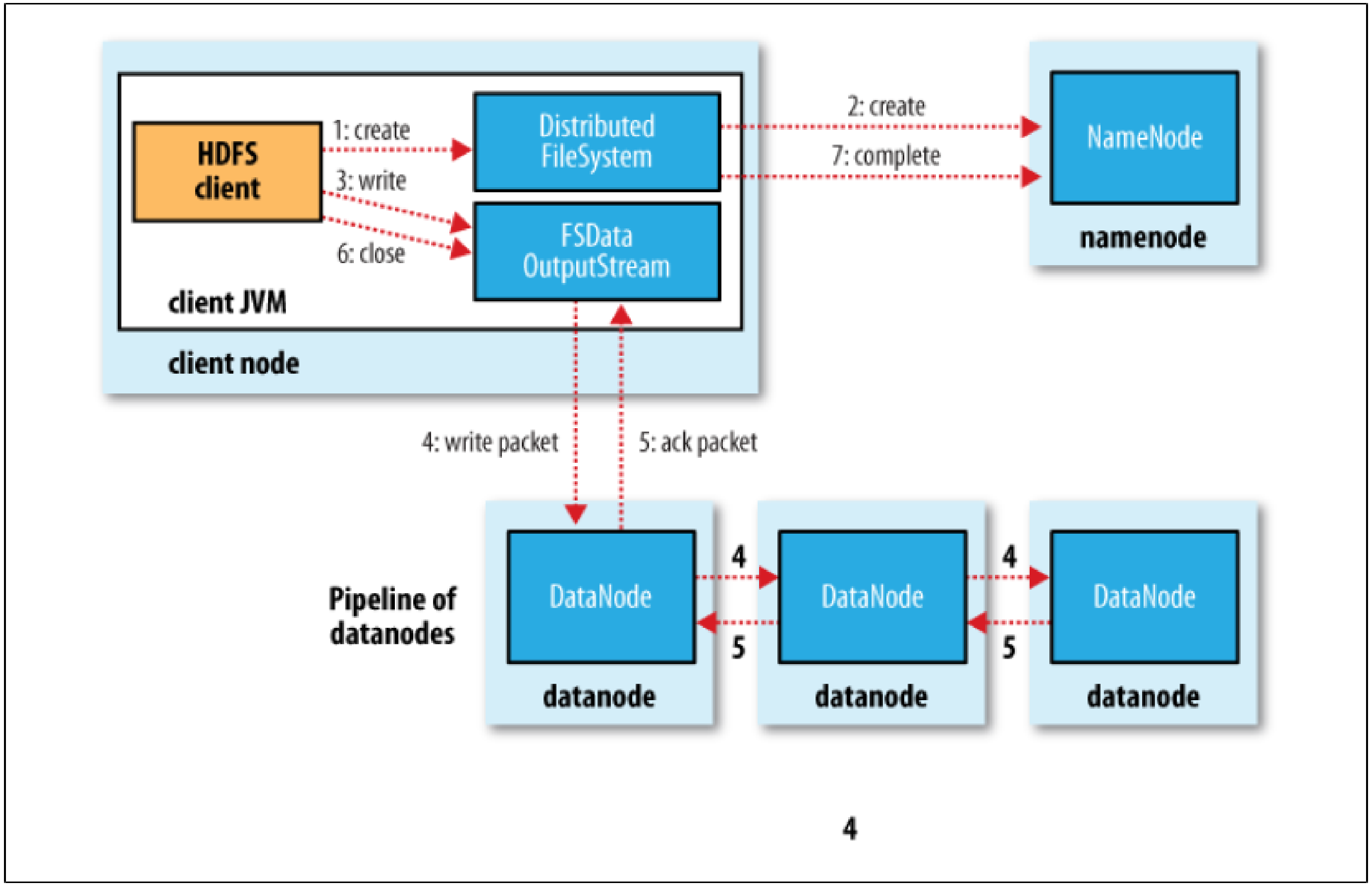
```
echo "blah blah blah" | hdfs dfs -put - tdataset/tfile.txt
hdfs dfs -ls -R
hdfs dfs -cat tdataset/tfile.txt
```

HDFS READ CLIENT



Source: Hadoop: The Definitive Guide

HDFS WRITE CLIENT



Source: Hadoop: The Definitive Guide

BLOCK PLACEMENT

- **Current Strategy**

  -- One replica on local node

  -- Second replica on a remote rack

  -- Third replica on same remote rack

  -- Additional replicas are randomly placed

- **Clients read from nearest replica**

- **Would like to make this policy pluggable**

NAMENODE FAILURE

- **A single point of failure**

- **Transaction Log stored in multiple directories**
  – A directory on the local file system
  – A directory on a remote file system (NFS/CIFS)

DATA PIPELINING

- Client retrieves a list of DataNodes on which to place replicas of a block
- Client writes block to the first DataNode
- The first DataNode forwards the data to the next DataNode in the Pipeline
- Usually, when all replicas are written, the Client moves on to write the next block in file

Conclusion:

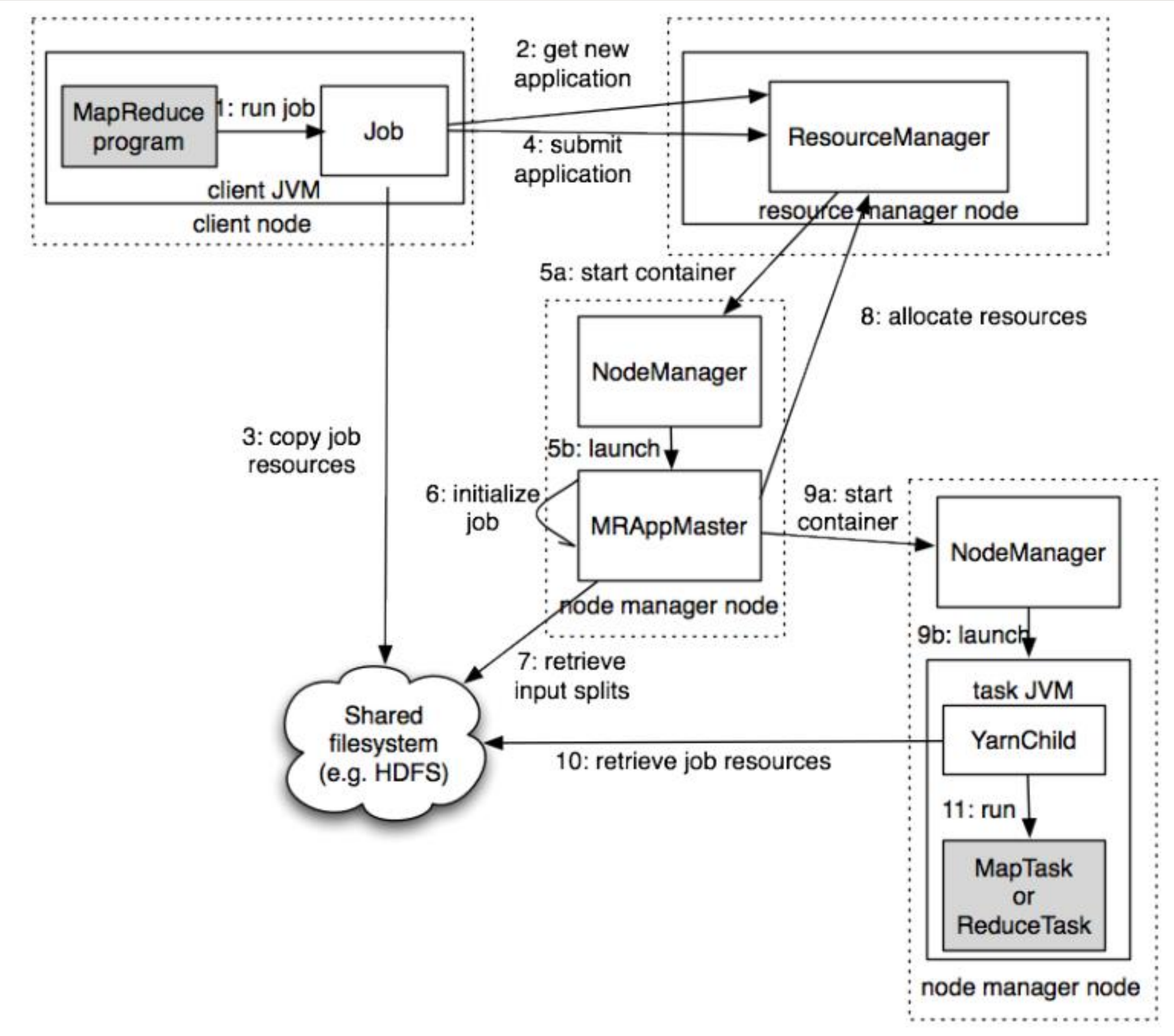
- We have seen:
  - The structure of HDFS.
  - The shell commands.
  - The architecture of HDFS system.
  - Internal functioning of HDFS.

MAPREDUCE INTERNALS

HADOOP MAP REDUCE

- Provides:
  - Automatic parallelization and Distribution
  - Fault Tolerance
  - Methods for interfacing with HDFS for colocation of computation and storage of output.
  - Status and Monitoring tools
  - API in Java
  - Ability to define the mapper and reducer in many languages through Hadoop streaming.

HADOOP(V2) MR JOB



Source: Hadoop: The Definitive Guide

WORDCOUNT PROGRAM

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

WORDCOUNT PROGRAM - MAIN

```
public class WordCount {
public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "word count");

job.setJarByClass(WordCount.class);

job.setMapperClass(TokenizerMapper.class);

job.setCombinerClass(IntSumReducer.class);

job.setReducerClass(IntSumReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0]));

FileOutputFormat.setOutputPath(job, new Path(args[1]));

System.exit(job.waitForCompletion(true) ? 0 : 1);

} }
```

WORDCOUNT PROGRAM - MAPPER

```
public static class TokenizerMapper extends Mapper<Object, Text, Text,
IntWritable>{

private final static IntWritable one = new IntWritable(1);

private Text word = new Text();


public void map(Object key, Text value, Context context )

throws IOException, InterruptedException {

      StringTokenizer itr = new StringTokenizer(value.toString());

      while (itr.hasMoreTokens()) {

            word.set(itr.nextToken()); context.write(word, one);

      }

}

}
```

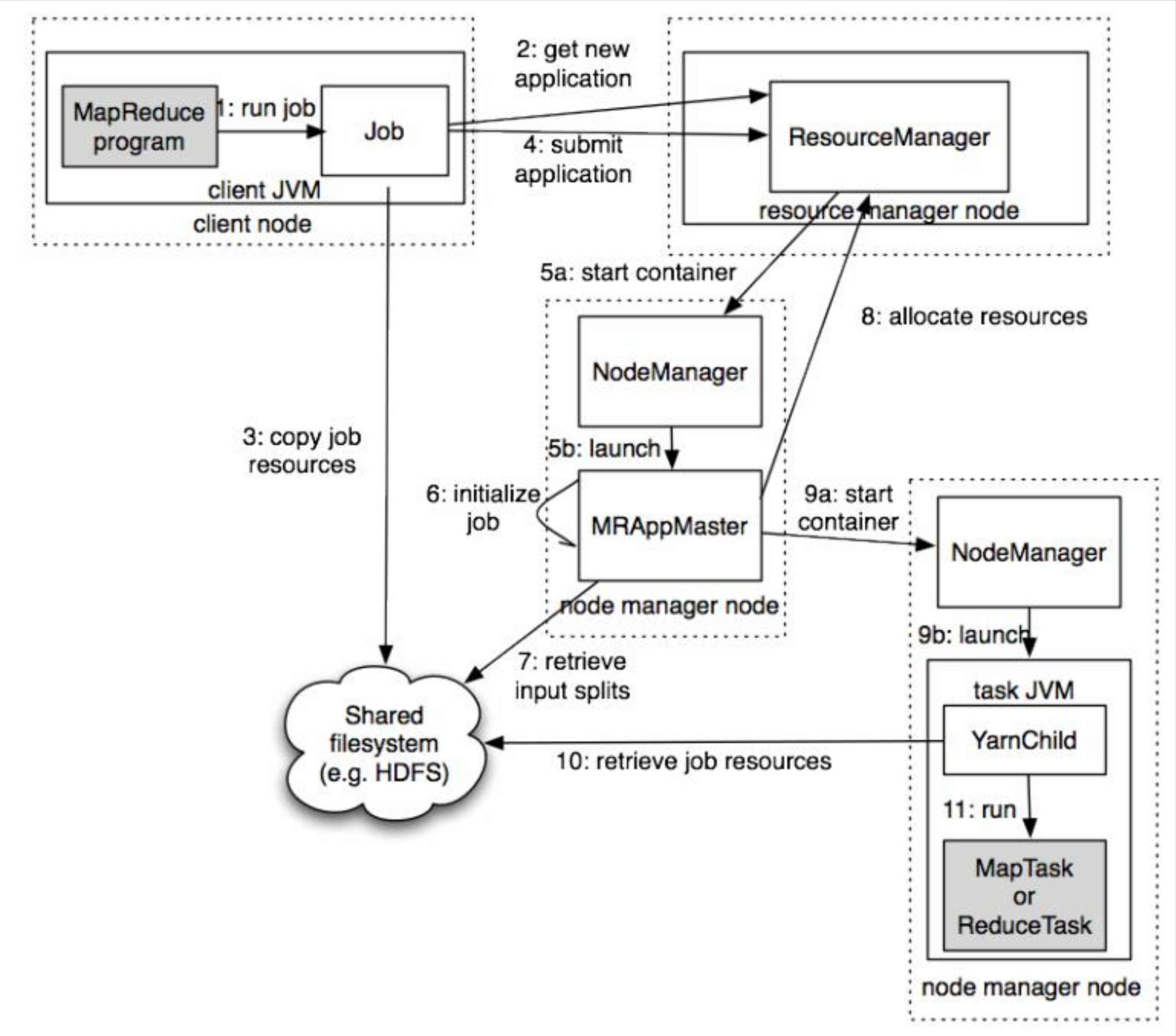
WORDCOUNT PROGRAM - REDUCER

```
public static class IntSumReducer extends
Reducer<Text,IntWritable,Text,IntWritable> {
private IntWritable result = new IntWritable();


public void reduce(Text key, Iterable<IntWritable> values, Context context )
throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
                sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
}
}
```

WORDCOUNT PROGRAM - RUNNING

```
export JAVA_HOME=[ Java home directory ]

bin/hadoop com.sun.tools.javac.Main WordCount.java

jar cf wc.jar WordCount*.class

bin/hadoop jar wc.jar WordCount [Input path] [Output path]
```

HADOOP(V2) MR JOB



Source: Hadoop: The Definitive Guide

WORDCOUNT IN PYTHON

Mapper.py

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

WORDCOUNT IN PYTHON

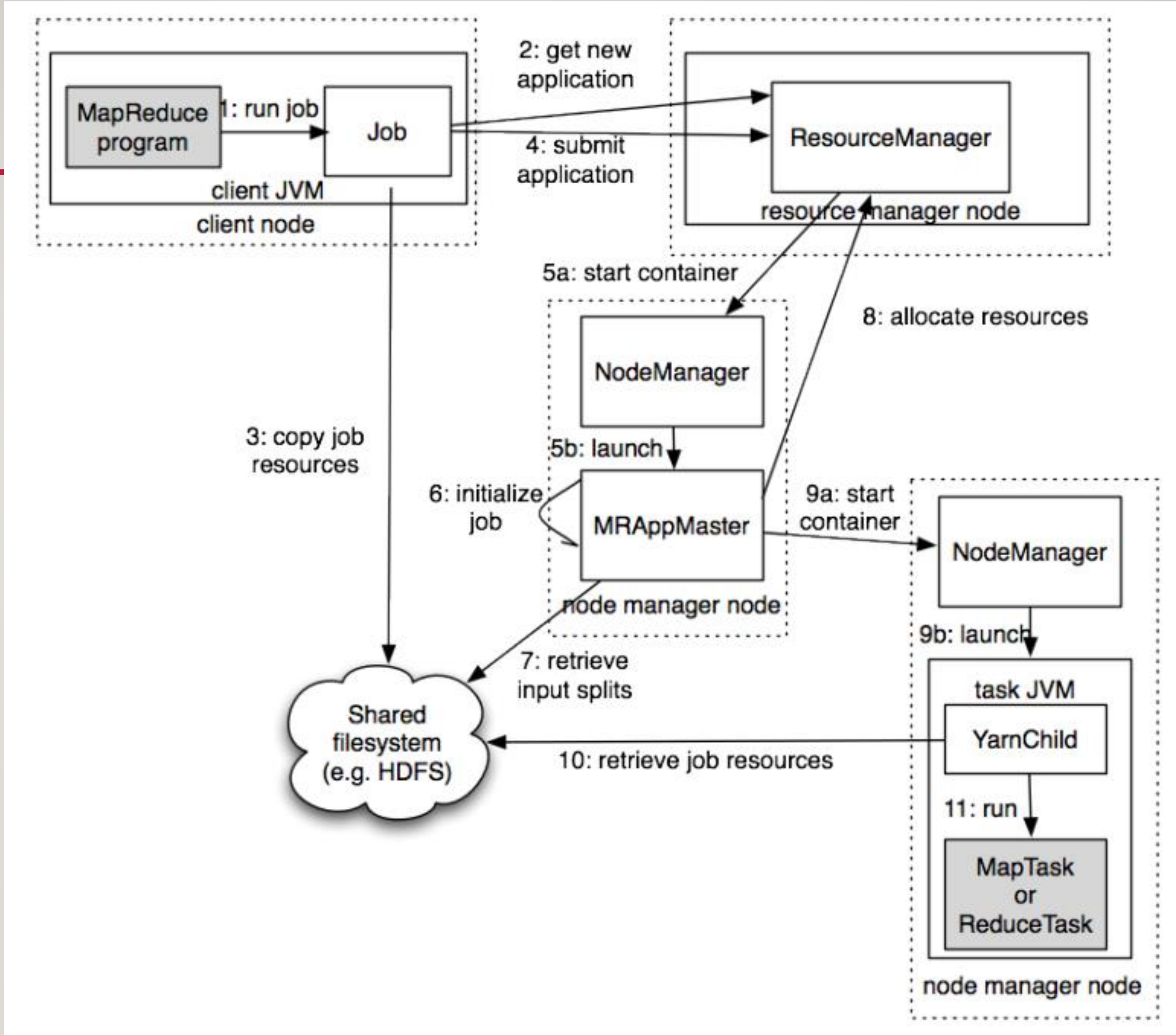
Reducer.py

```
#!/usr/bin/env python

from operator import itemgetter
import sys

# maps words to their counts
word2count = {}

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    # convert count (currently a string) to int
    try:
        count = int(count)
        word2count[word] = word2count.get(word, 0) + count
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        pass

# sort the words lexigraphically;
#
# this step is NOT required, we just do it so that our
# final output will look more like the official Hadoop
# word count examples
sorted_word2count = sorted(word2count.items(), key=itemgetter(0))

# write the results to STDOUT (standard output)
for word, count in sorted_word2count:
    print '%s\t%s'% (word, count)
```

EXECUTION CODE

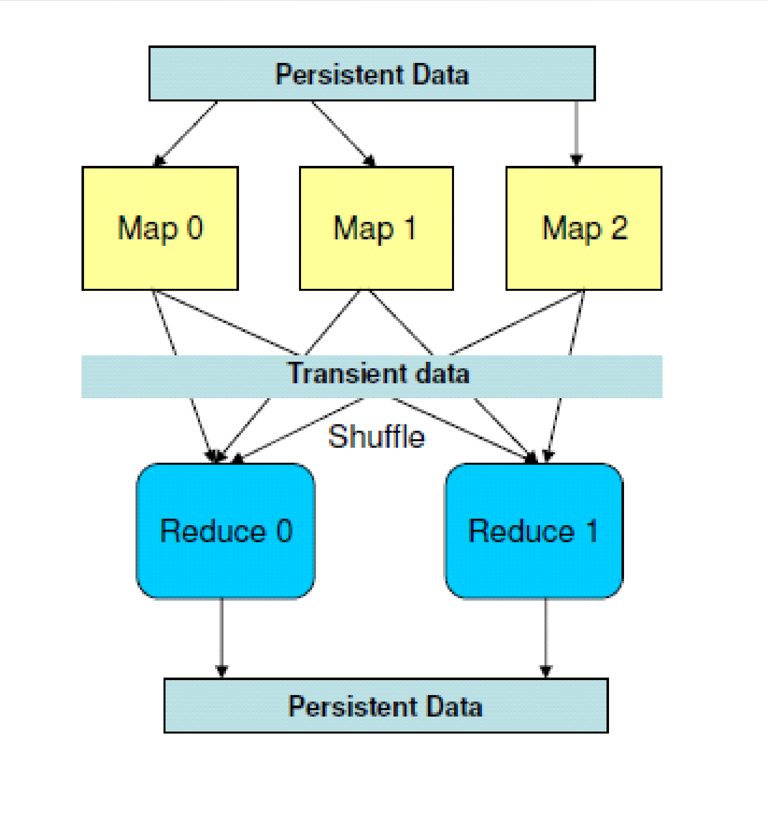
```
bin/hadoop dfs -ls

bin/hadoop dfs -copyFromLocal example example

bin/hadoop jar contrib/streaming/hadoop-0.19.2-streaming.jar -file
wordcount-py.example/mapper.py -mapper wordcount-py.example/mapper.py
-file wordcount-py.example/reducer.py -reducer wordcount-
py.example/reducer.py -input example -output java-output

bin/hadoop dfs -cat java-output/part-00000

bin/hadoop dfs -copyToLocal java-output/part-00000 java-output-local
```

HADOOP(V2) MR JOB

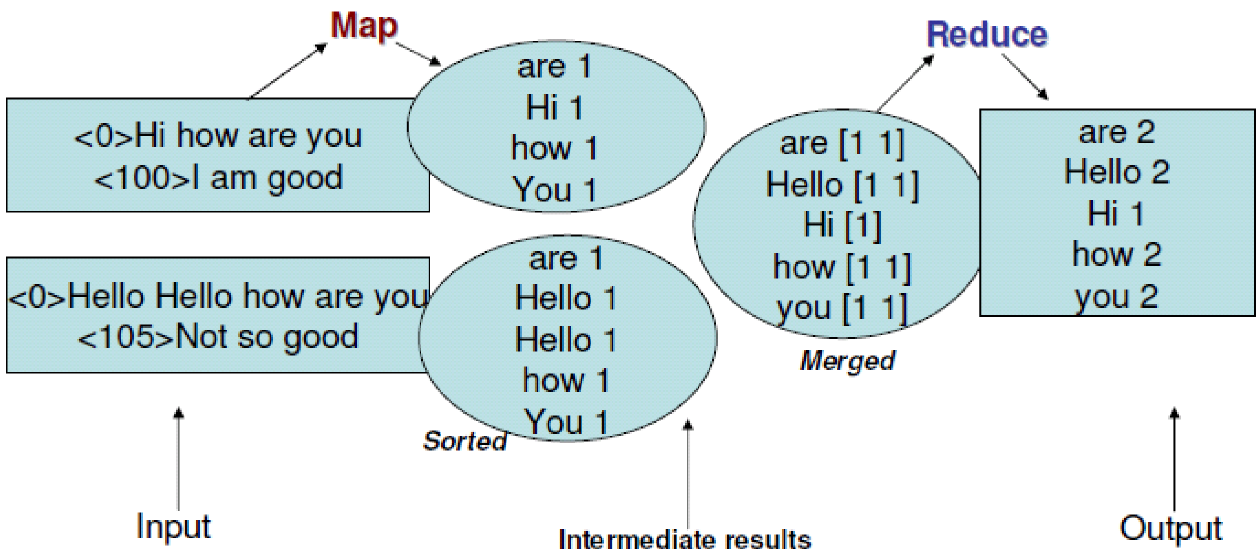


Source: Hadoop: The Definitive Guide

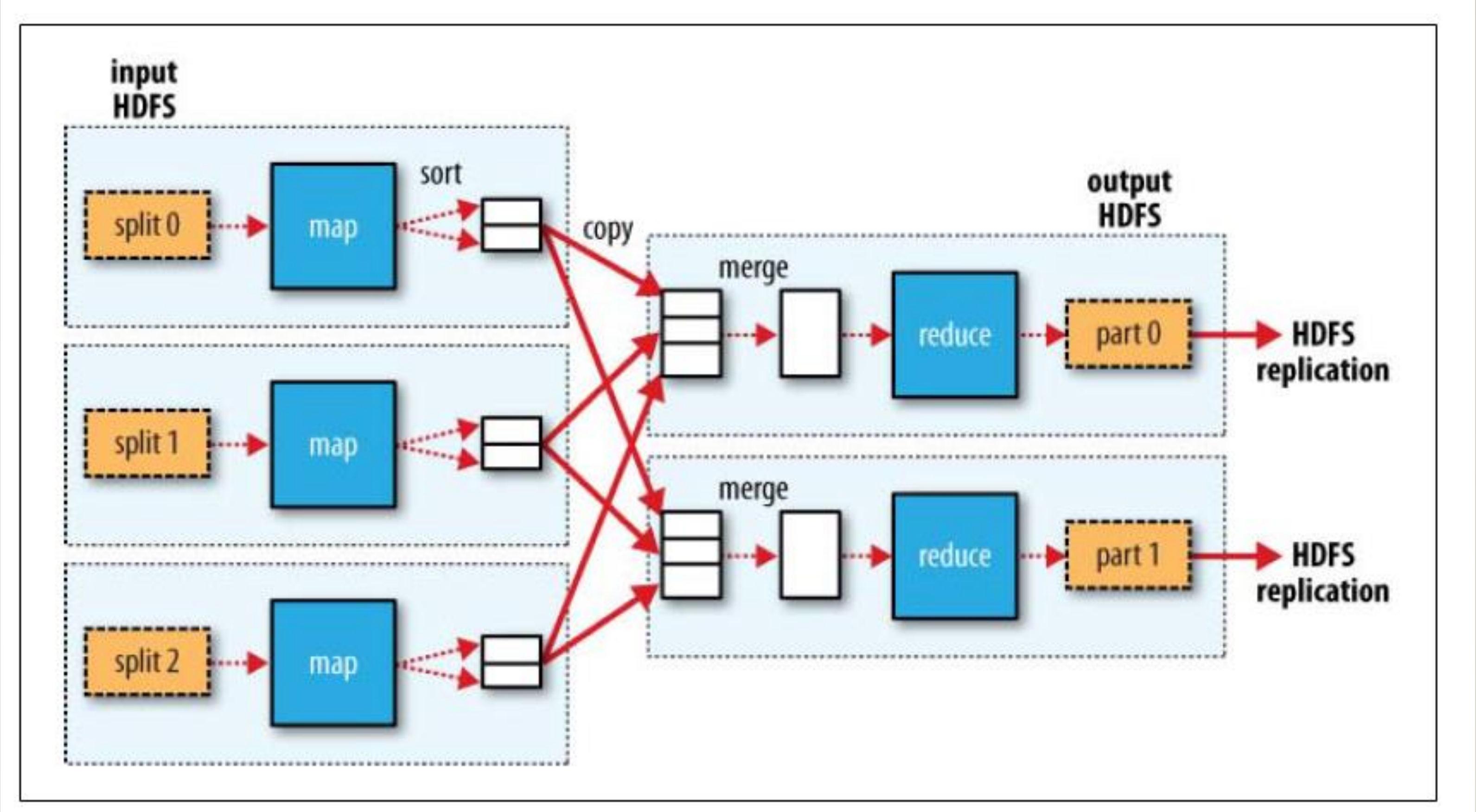
MAP REDUCE DATA FLOW



Data: Stream of keys and values

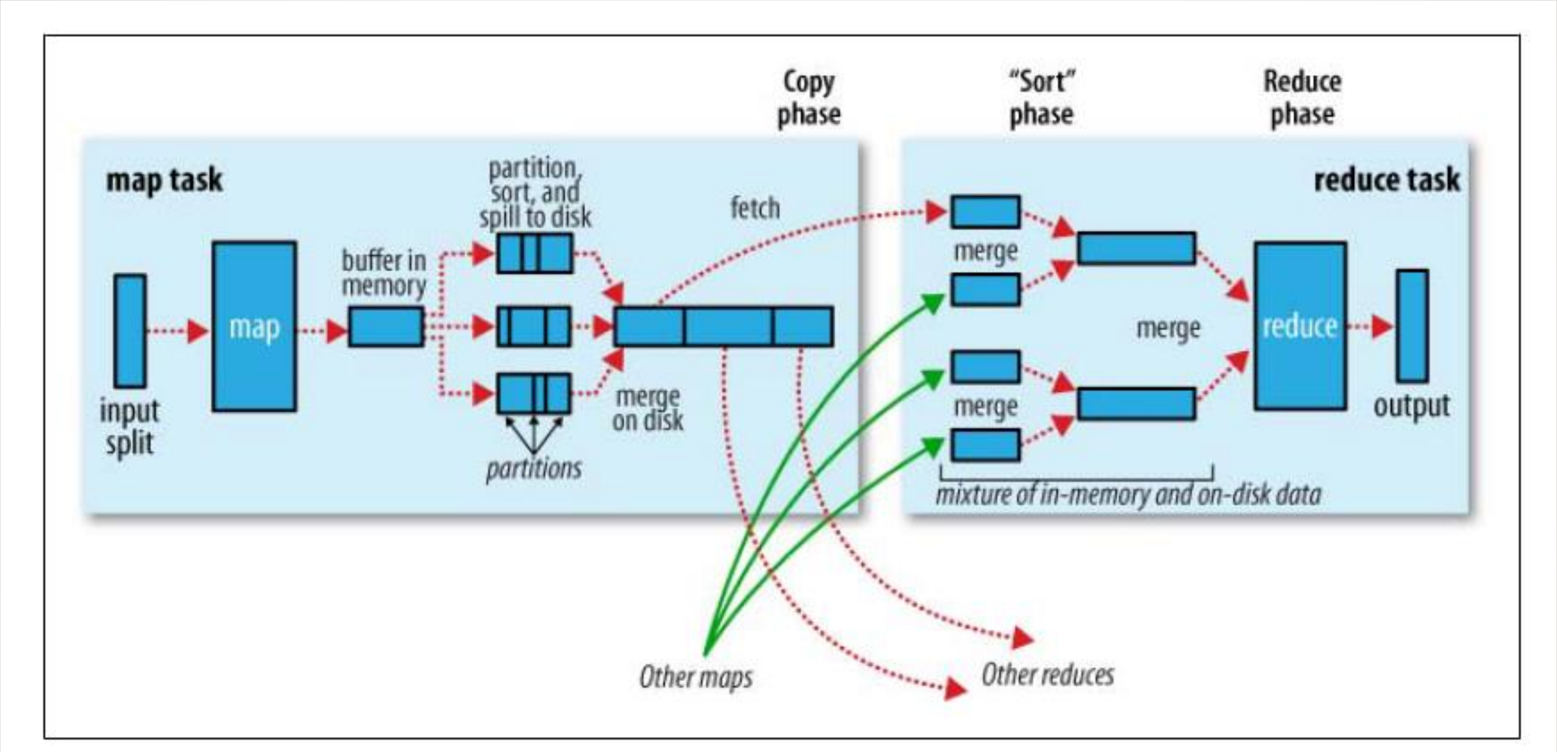


HADOOP MR DATA FLOW



Source: Hadoop: The Definitive Guide

SHUFFLE AND SORT



Source: Hadoop: The Definitive Guide

DATA FLOW

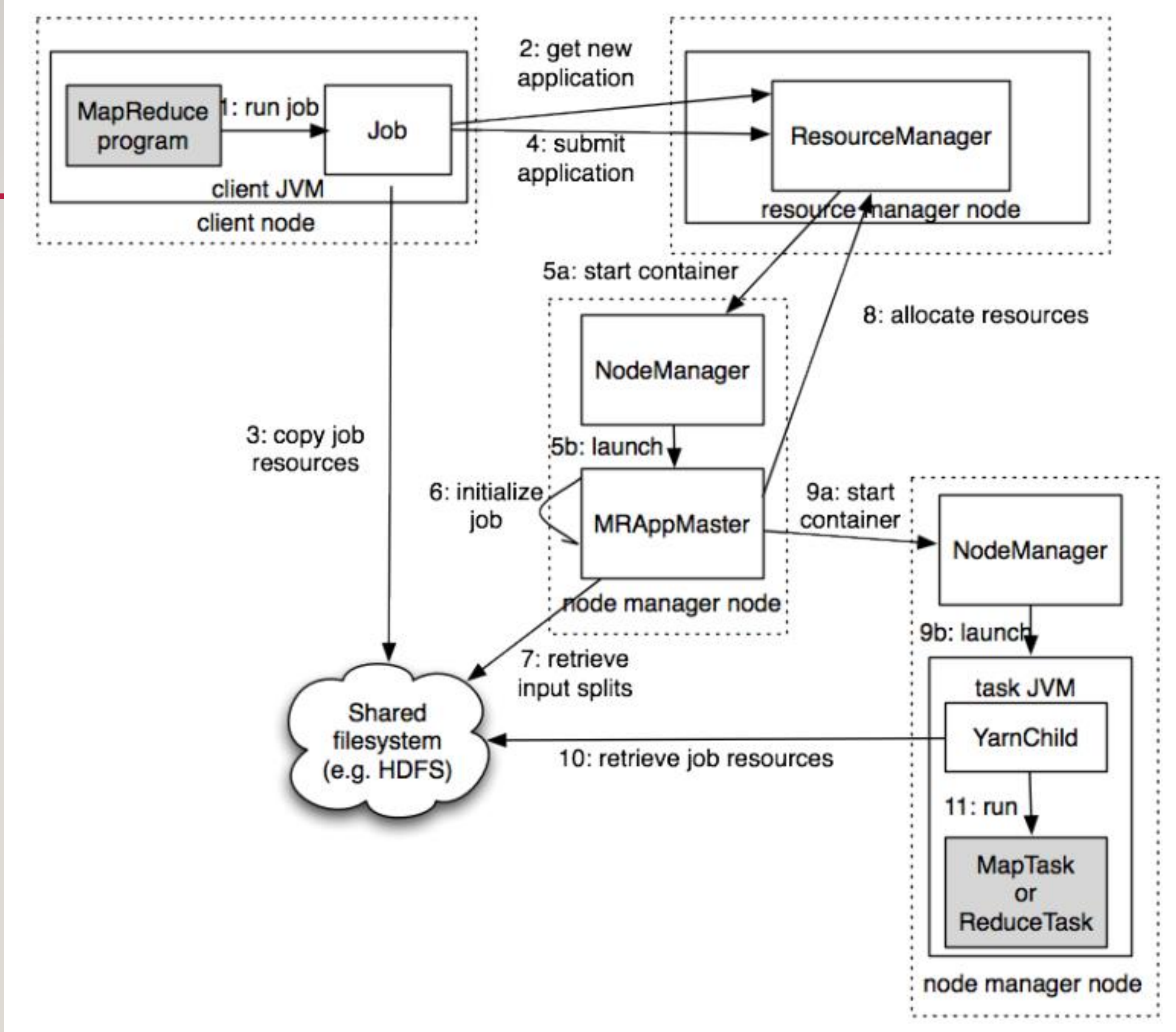
- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data

- **Intermediate results are stored on local FS of Map workers.**

- **Output** of **Reduce** workers are stored on a **distributed file system**.

- **Output is often input to another MapReduce task**

HADOOP(V2) MR JOB



Source: Hadoop: The Definitive Guide

FAULT TOLERANCE

- Comes from scalability and cost effectiveness
- HDFS:
  - Replication
- Map Reduce
  - Restarting failed tasks: map and reduce
  - Writing map output to FS
  - Minimizes re-computation

COORDINATION: MASTER

- Master node takes care of coordination:
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its *R* intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

FAILURES

- Task failure
  - Task has failed – report error to node manager, appmaster, client.
  - Task not responsive, JVM failure – Node manager restarts tasks.
- Application Master failure
  - Application master sends heartbeats to resource manager.
  - If not received, the resource manager retrieves job history of the run tasks.
- Node manager failure

DEALING WITH FAILURES

- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker

- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted

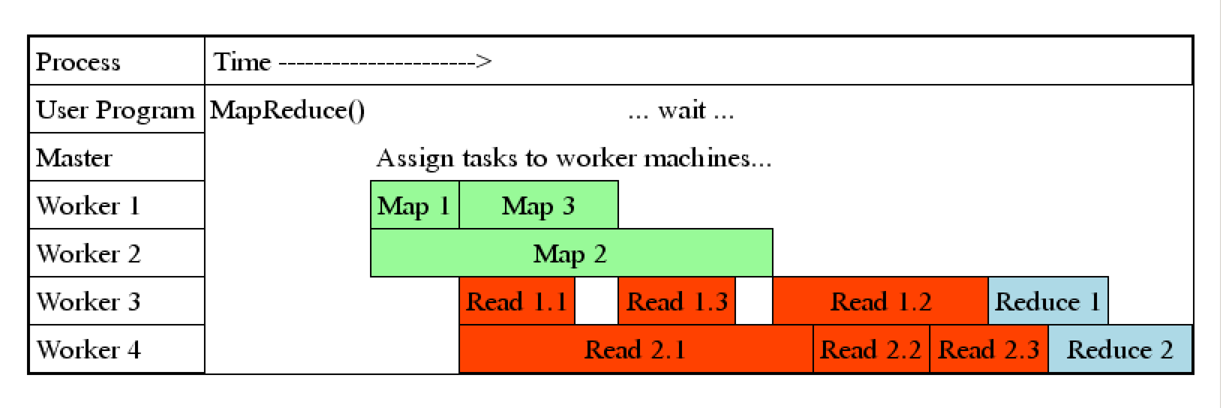
- **Master failure**
  - MapReduce task is aborted and client is notified

HOW MANY MAP AND REDUCE JOBS?

- *M* map tasks, *R* reduce tasks
- **Rule of a thumb:**
  - Make *M* much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually *R* is smaller than *M***
  - Because output is spread across *R* files

TASK GRANULARITY & PIPELINING

- **Fine granularity tasks:** map tasks >> machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing



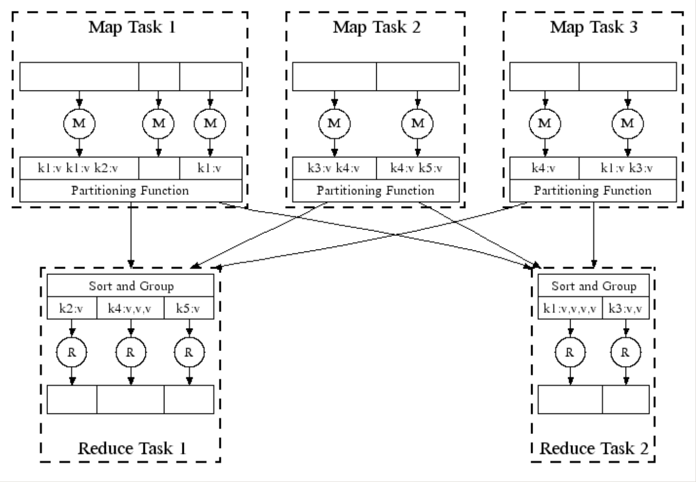
REFINEMENTS: BACKUP TASKS

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first "wins"
- **Effect**
  - Dramatically shortens job completion time

REFINEMENT: COMBINERS

- Often a Map task will produce many pairs of the form *(k,v_1), (k,v_2)*, … for the same key *k*
  - E.g., popular words in the word count example

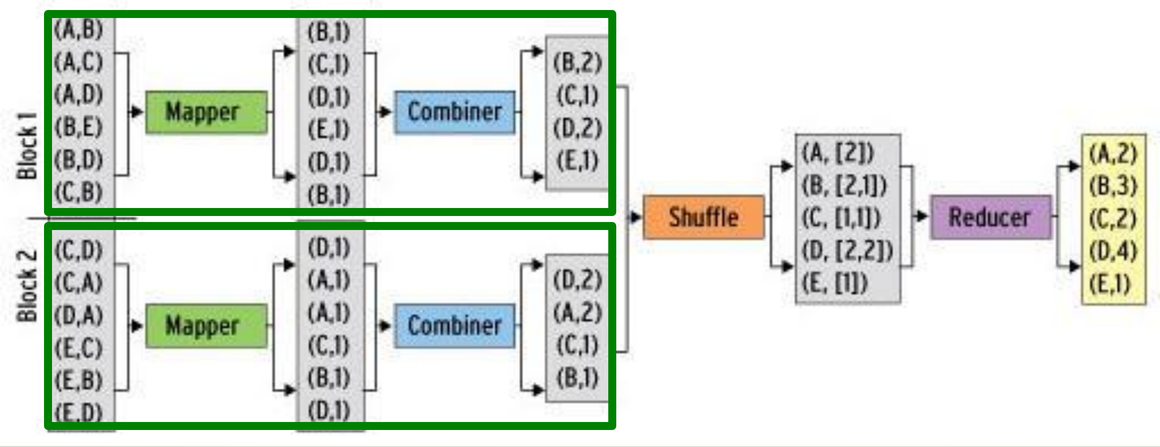
- **Can save network time by pre-aggregating values in the mapper:**
  - combine(k, list(v_1)) → v_2
  - Combiner is usually same as the reduce function



- Works only if reduce function is commutative and associative

REFINEMENT: COMBINERS

- **Back to our word counting example:**
  – Combiner combines the values of all keys of a single mapper (single machine):



  – Much less data needs to be copied and shuffled!

REFINEMENT: PARTITION FUNCTION

- **Want to control how keys get partitioned**
  - Inputs to map tasks are created by contiguous splits of input file
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- **System uses a default partition function:**
  - **hash(key) mod *R***
- **Sometimes useful to override the hash function:**
  - E.g., **hash(hostname(URL)) mod *R*** ensures URLs from a host end up in the same output file

SPARK

SPARK

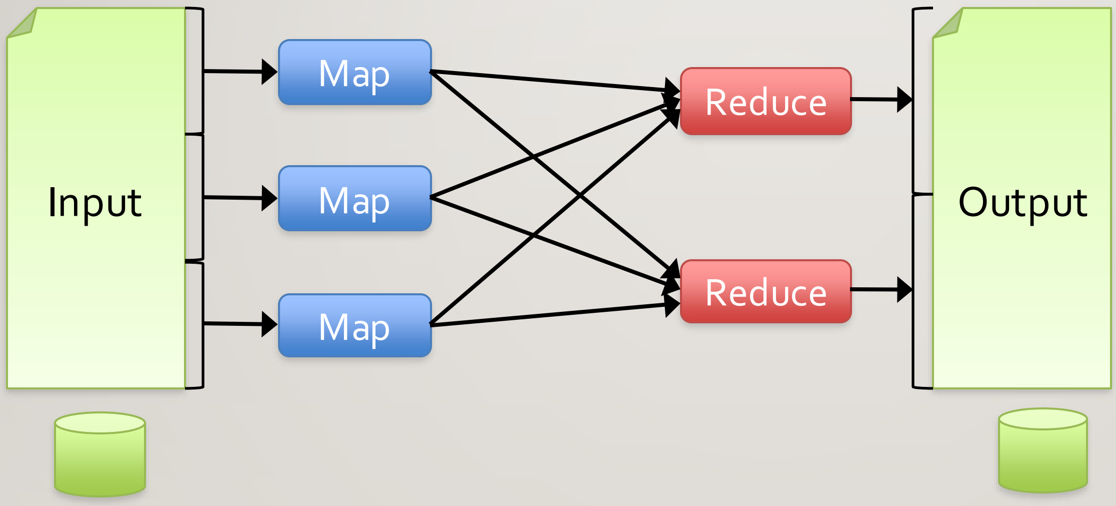
Spark is an In-Memory Cluster Computing platform for Iterative and Interactive Applications.

- Started in AMPLab at UC Berkeley.
- Resilient Distributed Datasets.
- Data and/or Computation Intensive.
- Scalable – fault tolerant.
- Integrated with SCALA.
- Straggler handling.
- Data locality.
- Easy to use.

MAP REDUCE DATA FLOW

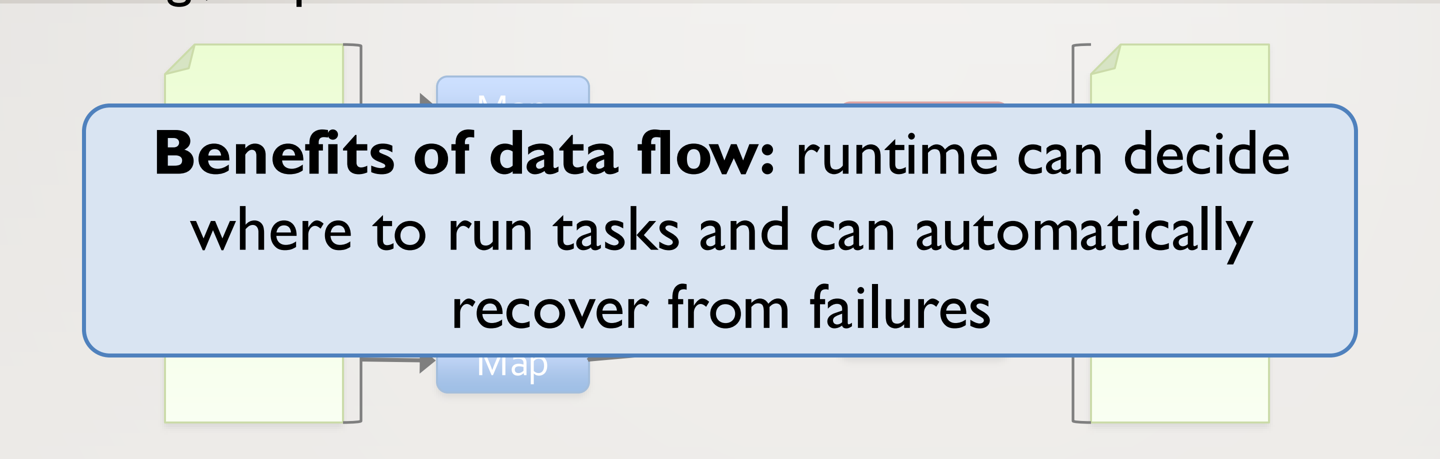
Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:



MAP REDUCE DATA FLOW

- Current popular programming models for clusters transform data flowing from stable storage to stable storage
- E.g., MapReduce:



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

MOTIVATION

- Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (many in machine learning)
  - **Interactive** data mining tools (R, Excel, Python)

- Spark makes working sets a first-class concept to efficiently support these apps

SPARK OBJECTIVE

- Provide distributed memory abstractions for clusters to support apps with working sets
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability

Solution: augment data flow model with "resilient distributed datasets" (RDDs)

RESILIENT DISTRIBUTED DATASETS

- Immutable distributed data collections inspired by Scala.
  - Array, List, Map, Set, etc.
- Transformations on RDDs create new RDDs.
  - Map, ReducebyKey, Filter, Join, etc.
- Actions on RDD return values.
  - Reduce, collect, count, take, etc.
- RDDs are materialized when needed – lazy execution.
- RDDs are can be cached to disk – graceful degradation with memory size.
- Spark framework re-computes lost splits of RDDs – fault tolerance.

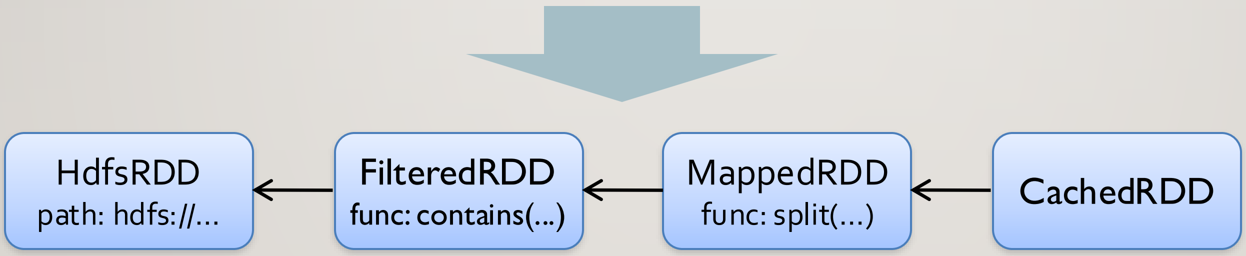
RDD OPERATIONS

Transformations (define a new RDD)	Actions (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	
…	

RDD LAZY EXECUTION

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions
- Ex:

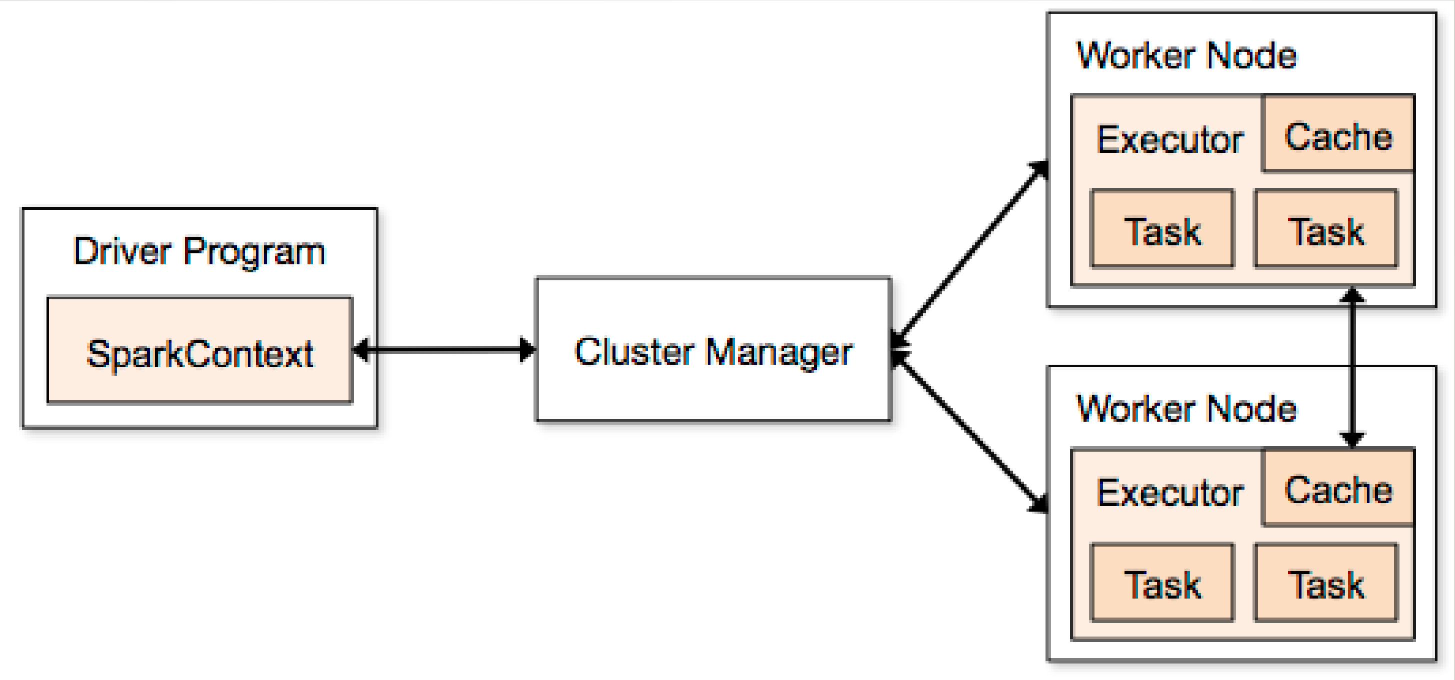
```
cachedMsgs = textFile(...).filter(_.contains(“error”))
                          .map(_.split(‘\t’)(2))
                          .cache()
```



BENEFITS OF RDD MODEL

- Consistency due to immutability
  - New RDDs take input from old RDDs which cannot be changed once created.
- Inexpensive fault tolerance
  - Lineage rather than replicating/checkpointing data
  - Only lost partitions are recomputed.
- Locality-aware scheduling of tasks on partitions
- Despite being restricted, model seems applicable to a broad variety of applications

SPARK CLUSTER ARCHITECTURE



WORD COUNT IN SPARK

```
val lines = spark.textFile("hdfs://...")

val counts = lines.flatMap(_.split("\\s"))
                  .reduceByKey(_ + _)

counts.save("hdfs://...")
```

EXAMPLE: 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))
```

SPARK PI

```scala
val slices = if (args.length > 0) args(0).toInt else 2

val n = math.min(100000L * slices, Int.MaxValue).toInt // avoid overflow

val count = spark.sparkContext.parallelize(1 until n, slices).map { i =>
      val x = random * 2 - 1
      val y = random * 2 - 1
      if (x*x + y*y <= 1) 1 else 0
    }.reduce(_ + _)

println(s"Pi is roughly ${4.0 * count / (n)}")
```

EXAMPLE: LOGISTIC REGRESSION

LOGISTIC REGRESSION

- Binary Classification. $y \, \varepsilon \, \{+1, -1\}$
- Probability of classes given by linear model:

$$p(y \mid x, w) = \frac{1}{1 + e^{(-yw^T x)}}$$

- Regularized ML estimate of w given dataset (x_i, y_i) is obtained by minimizing:

$$l(w) = \sum_i \log(1 + \exp(-y_i w^T x_i)) + \frac{\lambda}{2} w^T w$$

LOGISTIC REGRESSION

- Gradient of the objective is given by:

$$\nabla l(w) = \sum_i (1 - \sigma(y_i w^T x_i)) y_i x_i - \lambda w$$

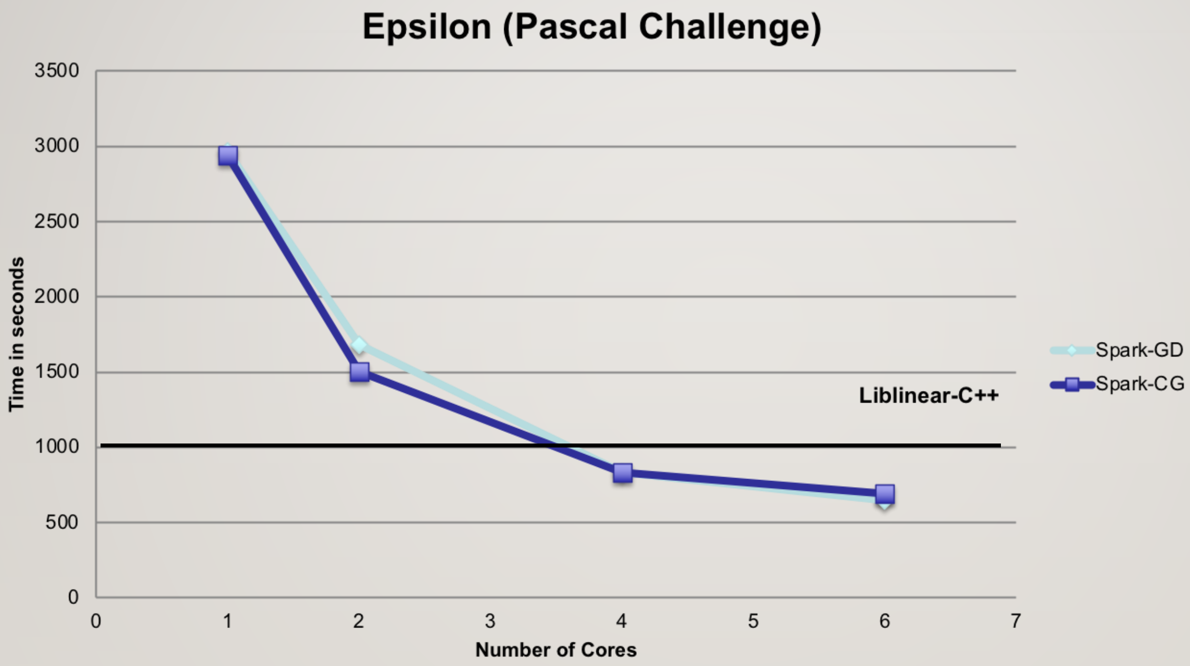
- Gradient Descent updates are:

$$w^{t+1} = w^t - s \nabla l(w^t)$$

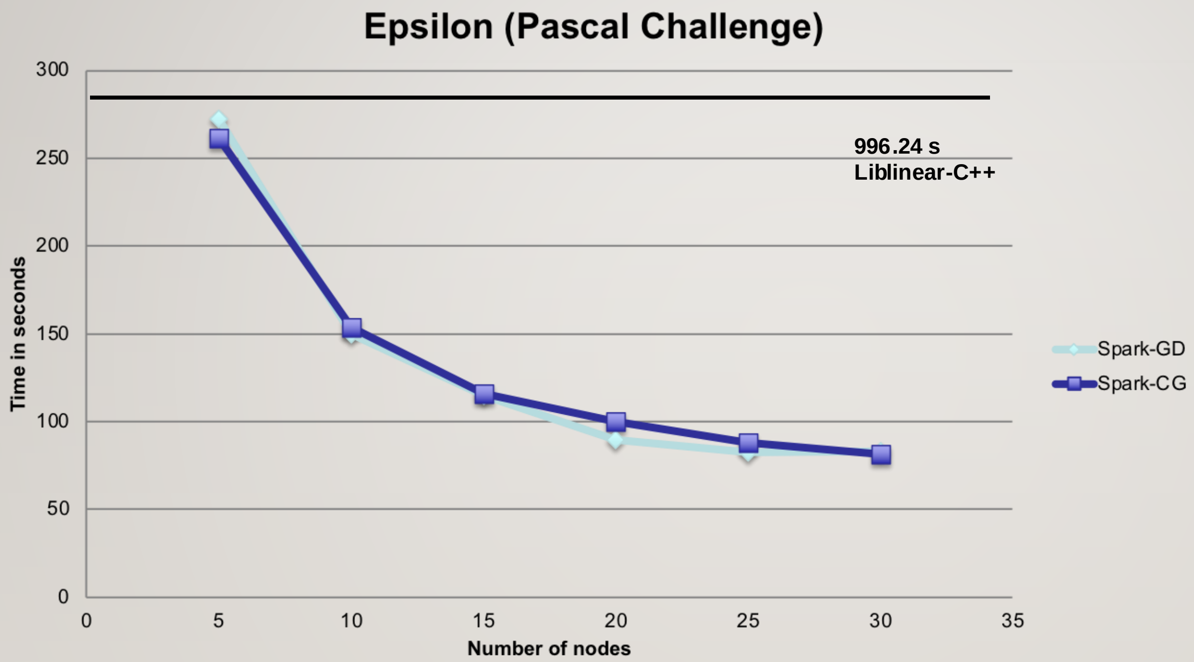
SPARK IMPLEMENTATION

```
val x = loadData(file) //creates RDD
var w = 0
do {
//creates RDD
val g = x.map(a => grad(w,a)).reduce(_+_)
s = linesearch(x,w,g)
w = w - s * g
}while(norm(g) > e)
```

SCALEUP WITH CORES



SCALEUP WITH NODES



EXAMPLE: MATRIX MULTIPLICATION

MATRIX MULTIPLICATION

- Representation of Matrix:
  - List <matrix id, Row index, Col index, Value>
  - Size of matrices: First matrix (A): m*k, Second matrix (B): k*n
- Scheme: For each record
  - if matrix 1: emit i=1,…,n records <(row ind, i), (col ind, value)>
  - Else: emit i=1,…,m records <(i , col ind), (row ind, value)>
- GroupByKey: so that there are m*n groups, each with 2*k records:
  - (col ind, value) for first matrix or (row ind,value) for second matrix
- Foreach group and for each record (i, value):
  - Find another record (j, value) such that i=j
  - Multiply the corresponding values and add to sum

EXAMPLE: PAGERANK

BASIC IDEA

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

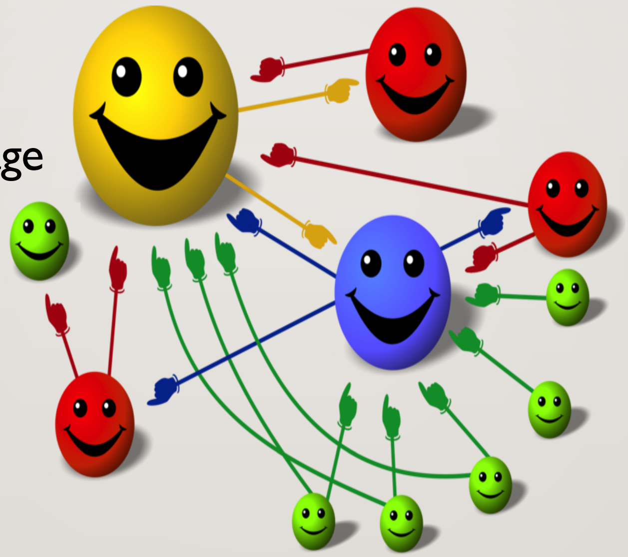
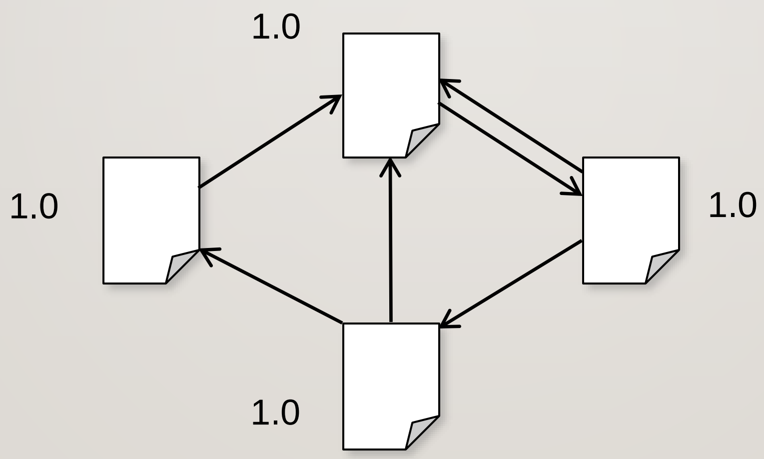


Image: en.wikipedia.org/wiki/File:PageRank-hi-res-2.png

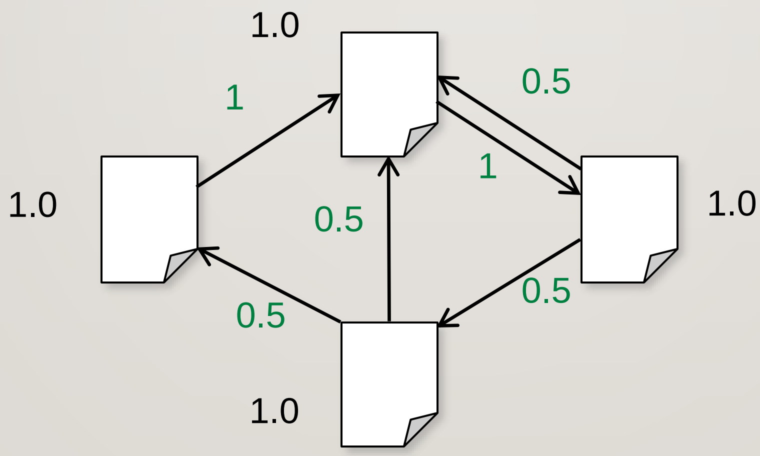
ALGORITHM

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times contribs$



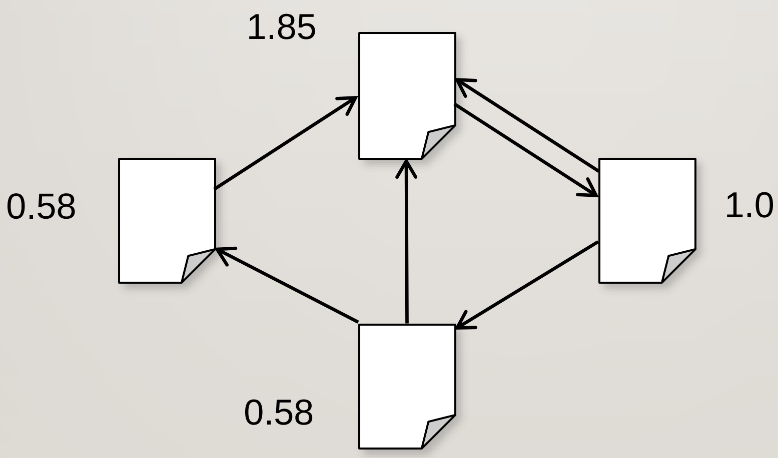
ALGORITHM

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times$ contribs



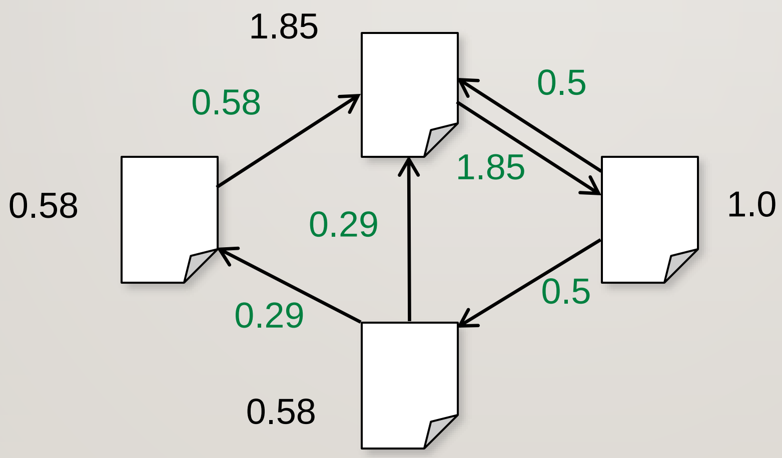
ALGORITHM

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to 0.15 + 0.85 × contribs



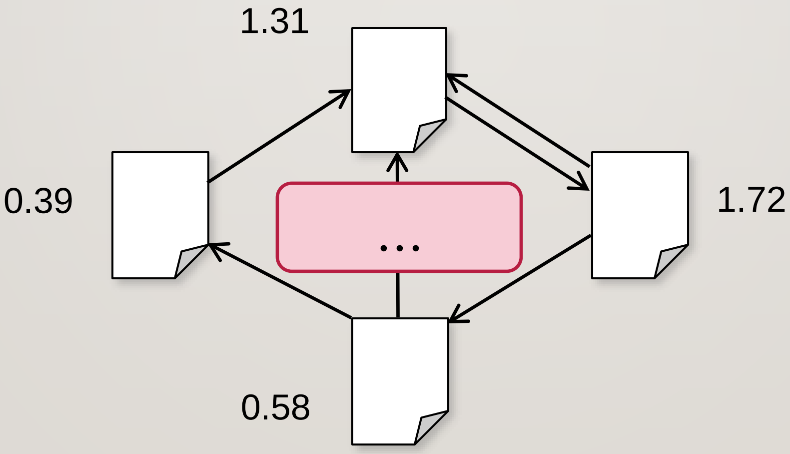
ALGORITHM

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to 0.15 + 0.85 × contribs



ALGORITHM

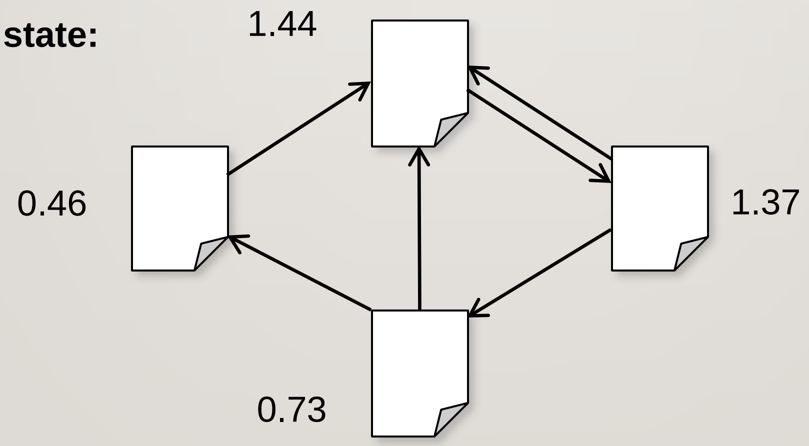
1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times$ contribs



ALGORITHM

1. Start each page at a rank of 1
2. On each iteration, have page p contribute $rank_p$ / $|neighbors_p|$ to its neighbors
3. Set each page's rank to $0.15 + 0.85 \times$ contribs

Final state:



SPARK IMPLEMENTATION

```
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 {
    (url, (nhb, rank)) =>
      nhb.flatMap(dest => (dest, rank/nhb.size))
  }
  ranks = contribs.reduceByKey(_ + _)
                  .mapValues(0.15 + 0.85 * _)
}

ranks.saveAsTextFile(...)
```

EXAMPLE: ALTERNATING LEAST SQUARES

COLLABORATIVE FILTERING

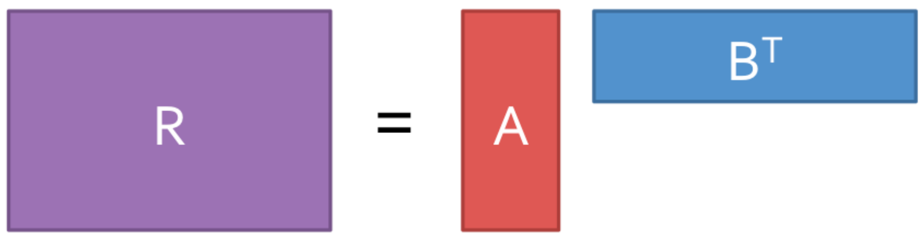
Predict movie ratings for a set of users based on their past ratings of other movies

$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix}$$

Users (rows) — Movies (columns)

MATRIX FACTORIZATION

Model R as product of user and movie matrices A and B of dimensions U×K and M×K



Problem: given subset of R, optimize A and B

ALTERNATING LEAST SQUARES

Start with random A and B

Repeat:

1. Fixing B, optimize A to minimize error on scores in R
2. Fixing A, optimize B to minimize error on scores in R

NAÏVE SPARK ALS

```
val R = readRatingsMatrix(...)

var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
           .map(i => updateUser(i, B, R))
           .toArray()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateMovie(i, A, R))
           .toArray()
}
```

EFFICIENT SPARK ALS

```
val R = spark.broadcast(readRatingsMatrix(...))

var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
           .map(i => updateUser(i, B, R.value))
           .toArray()
  B = spark.parallelize(0 until M, numSlices)
           .map(i => updateMovie(i, A, R.value))
           .toArray()
}
```

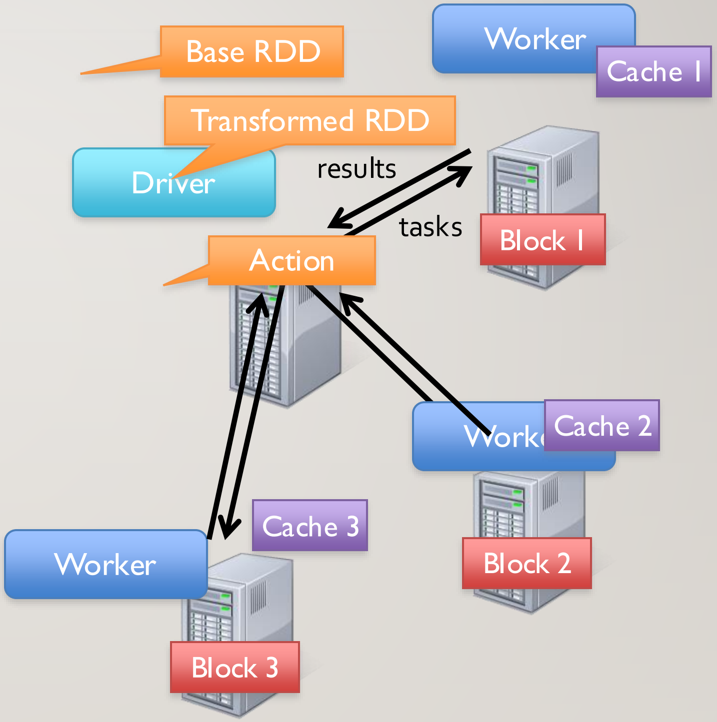
Solution: mark R as "broadcast variable"

CACHING OF RDDS

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count

cachedMsgs.filter(_.contains("bar")).count

. . .
```

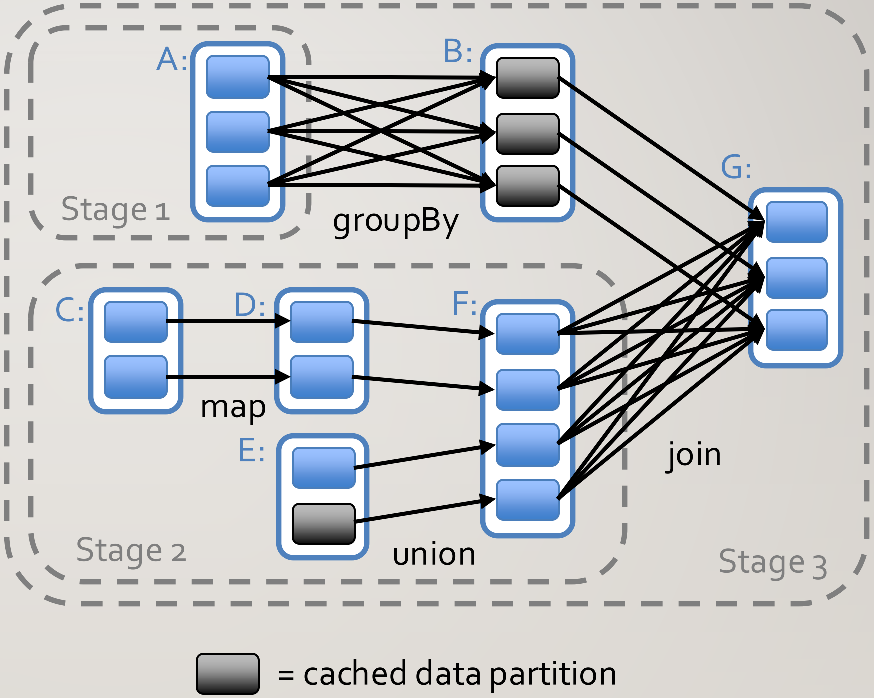
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)



SPARK INTERNALS

SPARK SCHEDULER

- Input: DAGs
- Creates the physical execution plan
- Breaks job into stages.
- Stages are DAG subgraphs with fat dependencies.
- DAG Scheduler:
  - Pipelines functions within a stage
  - Cache-aware work reuse & locality
  - Partitioning-awareto avoid shuffles



PHYSICAL EXECUTION PLAN

- User code defines a DAG (directed acyclic graph) of RDDs
  - Operations on RDDs create new RDDs that refer back to their parents, thereby creating a graph.
- Actions force translation of the DAG to an execution plan
  - When you call an action on an RDD, it's parents must be computed. That job will have one or more stages, with tasks for each partition. Each stage will correspond to one or more RDDs in the DAG. A single stage can correspond to multiple RDDs due to pipelining.
- Tasks are scheduled and executed on a cluster
  - Stages are processed in order, with individual tasks launching to compute segments of the RDD. Once the final stage is finished in a job, the action is complete.

TASKS

- Each task internally performs the following steps:
  - Fetching its input, either from data storage (if the RDD is an input RDD), an existing RDD (if the stage is based on already cached data), or shuffle outputs.
  - Performing the operation necessary to compute RDD(s) that it represents. For instance, executing filter() or map() functions on the input data, or performing grouping or reduction.
  - Writing output to a shuffle, to external storage, or back to the driver (if it is the final RDD of an action such as count()).

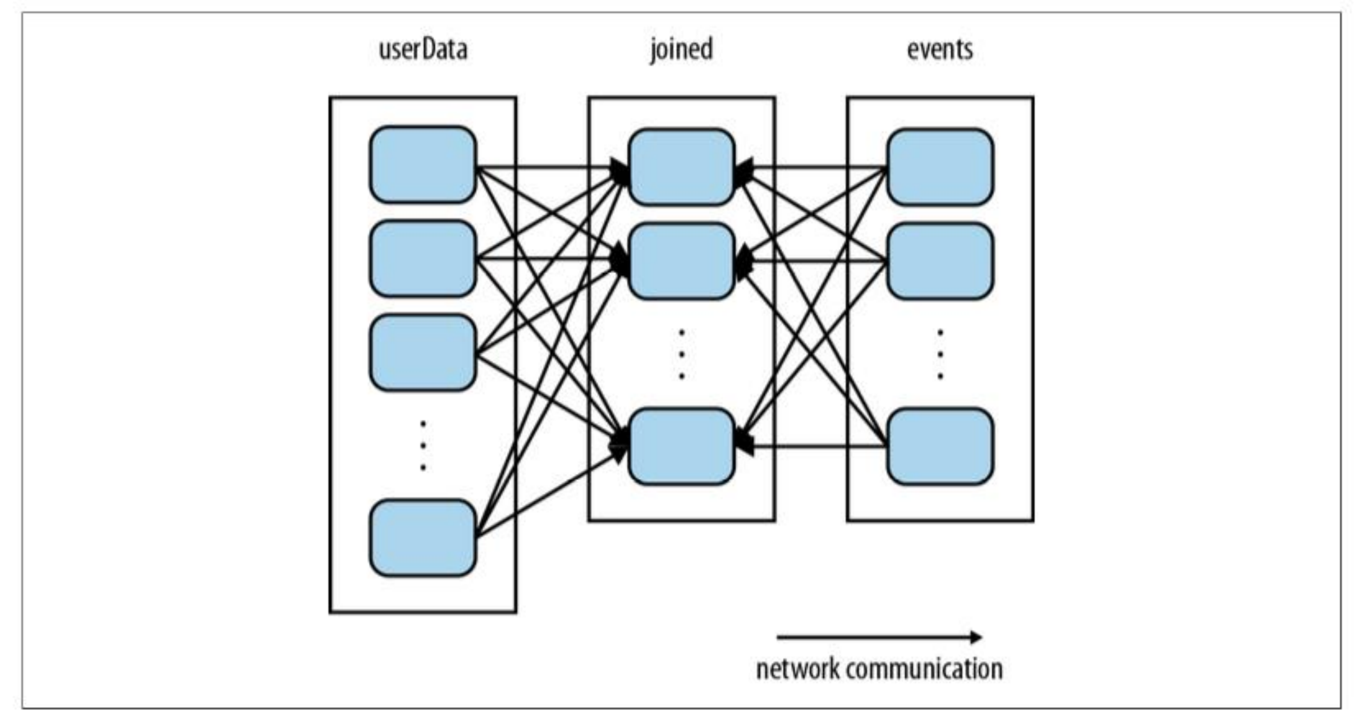
ADVANCED EXAMPLES

USER LOG MINING

Calculate the number of off-topic visits for a user.

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

def processNewLogs(logFileName: String) {
    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
    val joined = userData.join(events) // RDD of (UserID, (UserInfo, LinkInfo))
    pairs
    val offTopicVisits = joined.filter {// Expand the tuple into its components
    case (userId, (userInfo, linkInfo)) =>
      userInfo.topics.contains(linkInfo.topic)
    }.count()
    println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

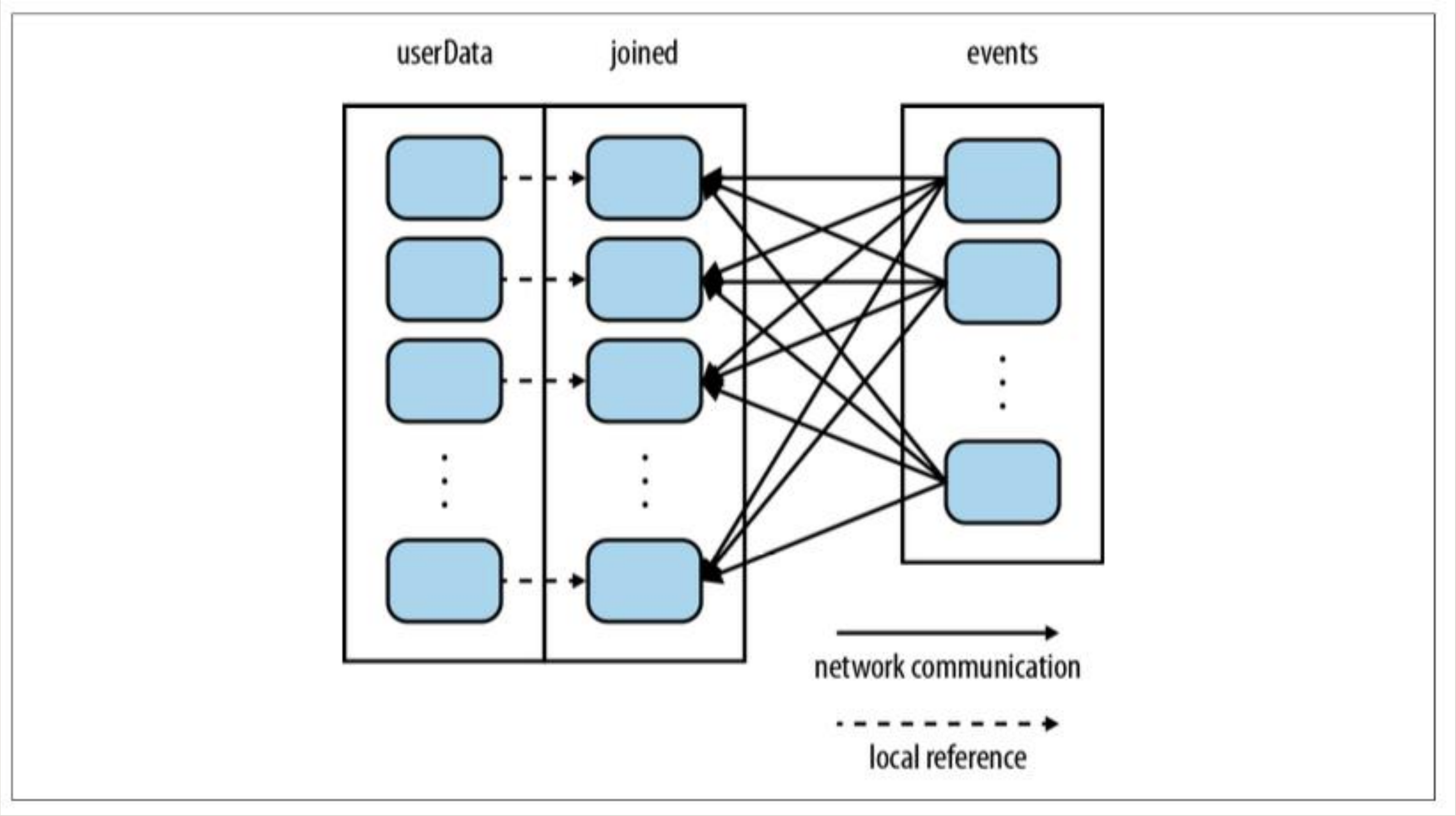
USER LOG MINING



USER LOG MINING

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
.partitionBy(new HashPartitioner(100)) // Create 100 partitions
.persist()
def processNewLogs(logFileName: String) {
    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
    val joined = userData.join(events) // RDD of (UserID, (UserInfo, LinkInfo))
    pairs
    val offTopicVisits = joined.filter {
     // Expand the tuple into its components
     (userId, (userInfo, linkInfo)) => userInfo.topics.contains(linkInfo.topic)
    }.count()
println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

USER LOG MINING



PARTITIONING

- RDD splits are created using a common hash function for related RDDs.
- Records with same keys are mapped to same split / partition.
  - Reduces network communication.
- Operations **benefitting** from partitioning:

`cogroup(), groupWith(), join(), leftOuterJoin(), rightOuter Join(), groupByKey(), reduceByKey(), combineByKey(), and lookup().`

- Operations **affecting** partitioning:

`cogroup(), groupWith(), join(), leftOuterJoin(), rightOuter Join(), groupByKey(), reduceByKey(), combineByKey(), partitionBy(), sort()`

```
mapValues() (if the parent RDD has a partitioner),
flatMapValues() (if parent has a partitioner)
filter() (if parent has a partitioner).
```

PAGE RANK (REVISITED)

```
val links = sc.objectFile[(String, Seq[String])]("links") .
partitionBy(new HashPartitioner(100)).persist()


var ranks = links.mapValues(v => 1.0)


for(i<-0 until 10) {
val contributions = links.join(ranks).flatMap {

case (pageId, (nbh, rank)) => nbh.map(dest => (dest, rank / nbh.size))

}
ranks = contributions.reduceByKey((x, y) => x + y).
mapValues(v => 0.15 + 0.85*v)

 }

ranks.saveAsTextFile("ranks")
```

ACCUMULATORS

```
val sc = new SparkContext(...) val file = sc.textFile("file.txt")


val blankLines = sc.accumulator(0)
// Create an Accumulator[Int] initialized to 0

val callSigns = file.flatMap(

line => {    if (line == "") {
blankLines += 1                     // Add to the accumulator

}
line.split(" ") })


callSigns.saveAsTextFile("output.txt”)


println("Blank lines: " + blankLines.value)
```

References:

- Learning Spark: Lightning-Fast Big Data Analysis. Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia. O Reilly Press 2015.
- Any book on scala and spark.
- Jure Leskovec, Anand Rajaraman, Jeff Ullman. Mining of Massive Datasets. 2nd edition. - Cambridge University Press. http://www.mmds.org/
- Tom White. Hadoop: The definitive Guide. Oreilly Press.

THANKS

QUESTIONS?

Email: sourangshu@cse.iitkgp.ac.in