

# BIG DATA ANALYTICS



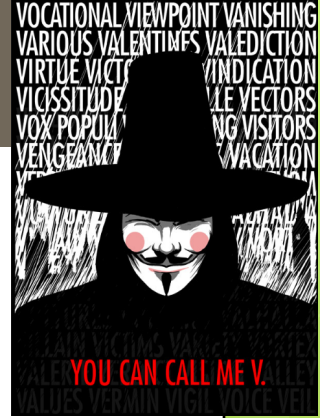
Dr. Buğra Gedik  
Department of Computer  
Engineering, Bilkent University

# Overview

- Motivation
- Big Data Processing Frameworks
  - Map/Reduce (M/R)
  - Resilient Distributed Datasets (RDD)
  - Bulk-synchronous Processing (BSP)
- Data Mining on Big Data with Mahout
  - Using the CLI
  - Email classification via Naive Bayes
  - News clustering via k-Means
- Graph Mining on Big Data
  - Map/Reduce-based Algorithms: Degree computation, PageRank
  - BSP-style Algorithms: Connected Components, PageRank
  - Asynchronous Systems and Algorithms: PageRank on GraphLab

# BIG DATA

- The increase in the **V**olume, **V**elocity, and **V**ariety of data has passed a threshold such that existing data management and mining technologies are insufficient in managing and extracting actionable insight from this data
- Big Data technologies are new technologies that represent a paradigm shift, in the areas of platform, analytics, and applications
- Key features



*Scalability in managing and mining of data*



*Analysis and mining with low-latency and high throughput*



*Non-traditional data, including semi-structured and unstructured*

# Big Data Processing Frameworks

- Map/Reduce (M/R)
- Resilient Distributed Datasets (RDD)
- Bulk-Synchronous Processing (BSP)

# Map/Reduce

- Express computations as a series of *map* and *reduce* steps
  - map*: transform data items into (key, value) pairs
  - reduce*: aggregate values of items that share the same key
- Simple Example: Word count
  - Input: A bunch of documents
  - Output: The number of occurrences of each word
  - Map: Convert each document into a series of (key, value) pairs, where the key is a word, value is the number of times it appears in the doc
    - "Happy new years everyone. Happy 2015."  
=> ("Happy",2),("new",1),("years",1),("everyone",1),("2015",1)
    - "A new year, a new hope"  
=> ("A",2),("new",2),("year",1),("hope",1)
  - Reduce: Sum up the counts for each word to get totals
    - ("new",1),("new",2) => ("new",3)
    - ...

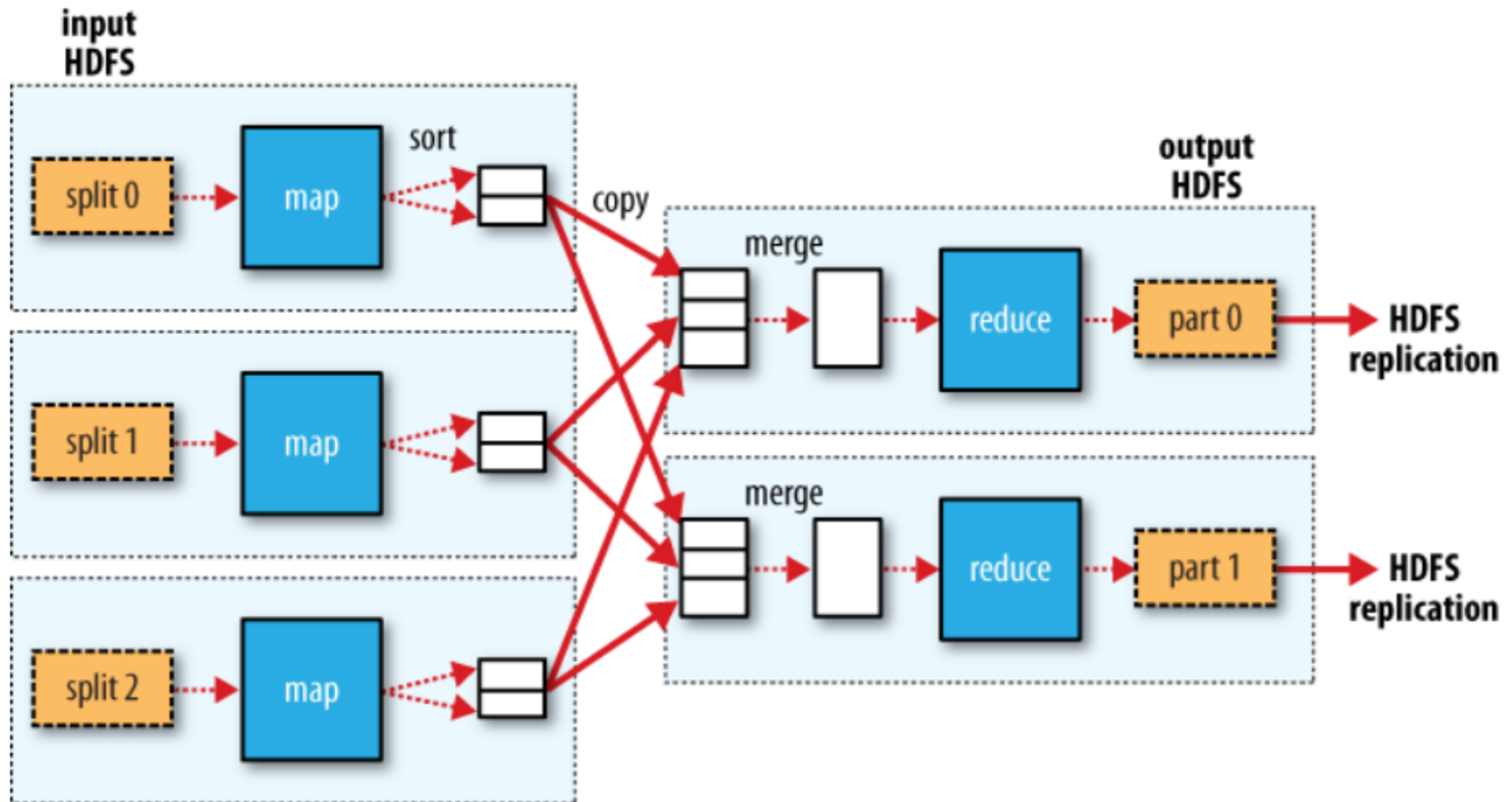
# M/R: What's So Special?

- Not much, other than:
  - Surprisingly many algorithms can be expressed as Map/Reduce jobs
  - Both the Map step, and the Reduce step are highly parallelizable
  - Map/Reduce lends itself to a scalable distributed implementation
    - **Apache Hadoop** is the popular open-source implementation

# Distributed M/R: A Giant Sort Machine

- A distributed file system stores the input data (a bunch of files)
  - The data is distributed over machines for scalability
  - Replication is used for fault-tolerance
- Map phase: Each machine executes a number of Map tasks (using preferably local input data)
  - The output of the Map tasks are buffered in memory and are spilled to disk as the buffer fills up.
  - The output is stored on disk in a partitioned way, each partition corresponds to a reducer (key is hashed to the reducers)
  - When mappers are done, partitions are sorted by the key on the disk
- Reduce phase: Each machine executes a bunch of reduce tasks
  - The mapper output prepared for the reducer is fetched
  - A disk-based merge takes place to order the mapper output from different mappers into a single sorted output
  - The reducer is then executed by streaming the sorted key/value pairs

# Distributed Map/Reduce Process





# Distributed M/R: End-to-end Example

- ◉ Assume we are counting characters
- ◉ Assume there are 3 mappers and 2 reducers
- ◉ Mapper 0 gets the following lines:  
“abadbb”, “acbcaa”, “bcccbb”
- ◉ Mapper 1 gets the following lines:  
“dada”, “acdc”, “cddc”, “dd”
- ◉ Mapper 2 gets the following lines:  
“aba”, “bdbd”, “baab”, “bdb”

## Example continued: Map phase (1)

- Mappers generate key/value pairs
- Key value pairs generated by Mapper 0  
[(a,2),(b,3),(d,1),(a,3),(c,2),(b,1),(b,3),(c,3)]
- Key value pairs generated by Mapper 1  
[(d,2),(a,2),(a,1),(c,2),(d,1),(c,2),(d,2),(d,2)]
- Key value pairs generated by Mapper 2  
[(a,2),(b,1),(b,2),(d,2),(b,2),(a,2),(b,2),(d,1)]

## Example continued: Map phase (2)

- Assume we have a spill buffer size of 4
- Assume a hash function  $H$ , where  $H(a) = 0, H(b)=1, H(c)=0, H(d)=1$

- List of buffers and their contents for Mapper 0

SpillBuffer0		SpillBuffer1	
[ (a,2),(a,3)   (b,3),(d,1) ]	for Reducer0	[ (c,2),(c,3)   (b,1),(b,3) ]	for Reducer1

- List of buffers and their contents for Mapper 1

[ (a,2),(a,1),(c,2)   (d,2) ]	[ (c,2)   (d,1),(d,2),(d,2) ]
-------------------------------	-------------------------------

- List of buffers and their contents for Mapper 2

[ (a,2)   (b,1),(b,2),(d,2) ]	[ (a,2)   (b,2),(b,2),(d,1) ]
-------------------------------	-------------------------------

## Example continued: Map phase (3)

- Mappers merge their spill buffers and sort each partition (using the key)

- The final output from Mapper 0:

[ (a,2),(a,3),(c,2),(c,3)		(b,3),(b,1),(b,3),(d,1)	]
For Reducer0		For Reducer1	

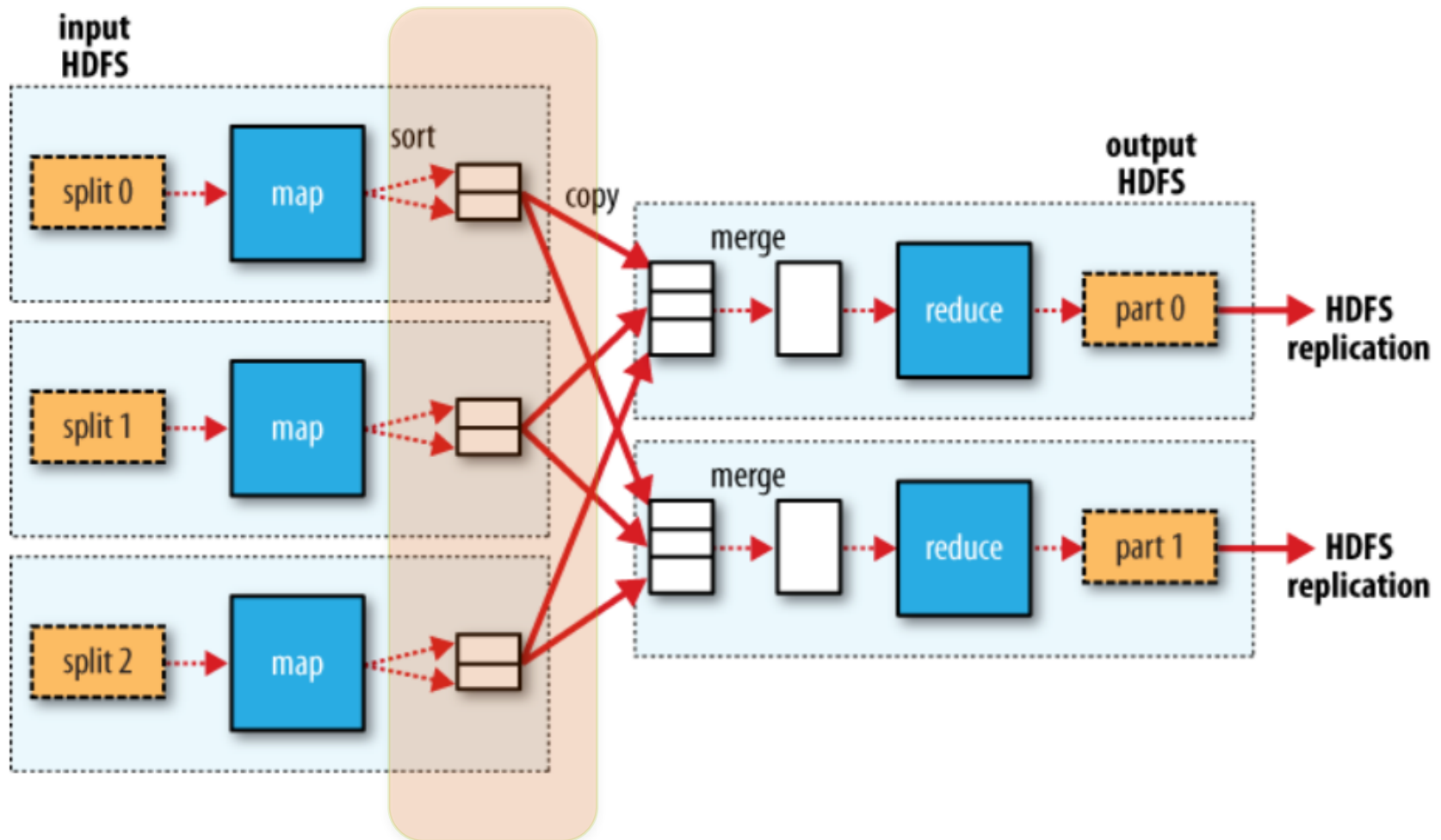
- The final output from Mapper 1:

[ (a,2),(a,1),(c,2),(c,2)		(d,2),(d,1),(d,2),(d,2)	]
---------------------------	--	-------------------------	---

- The final output from Mapper 2:

[ (a,2),(a,2)		(b,1),(b,2),(b,2),(b,2),(d,2),(d,1)	]
---------------	--	-------------------------------------	---

# Distributed Map/Reduce Process



# Example continued: Reduce (1)

- Reducers fetch data from mappers
- Reducer 0:
  - From Mapper 0: [(a,2),(a,3),(c,2),(c,3)]
  - From Mapper 1: [(a,2),(a,1),(c,2),(c,2)]
  - From Mapper 2: [(a,2),(a,2)]
- Reducer 1:
  - From Mapper 0: [(b,3),(b,1),(b,3),(d,1)]
  - From Mapper 1: [(d,2),(d,1),(d,2),(d,2)]
  - From Mapper 2: [(b,1),(b,2),(b,2),(b,2),(d,2),(d,1)]

## Example continued: Reduce (2)

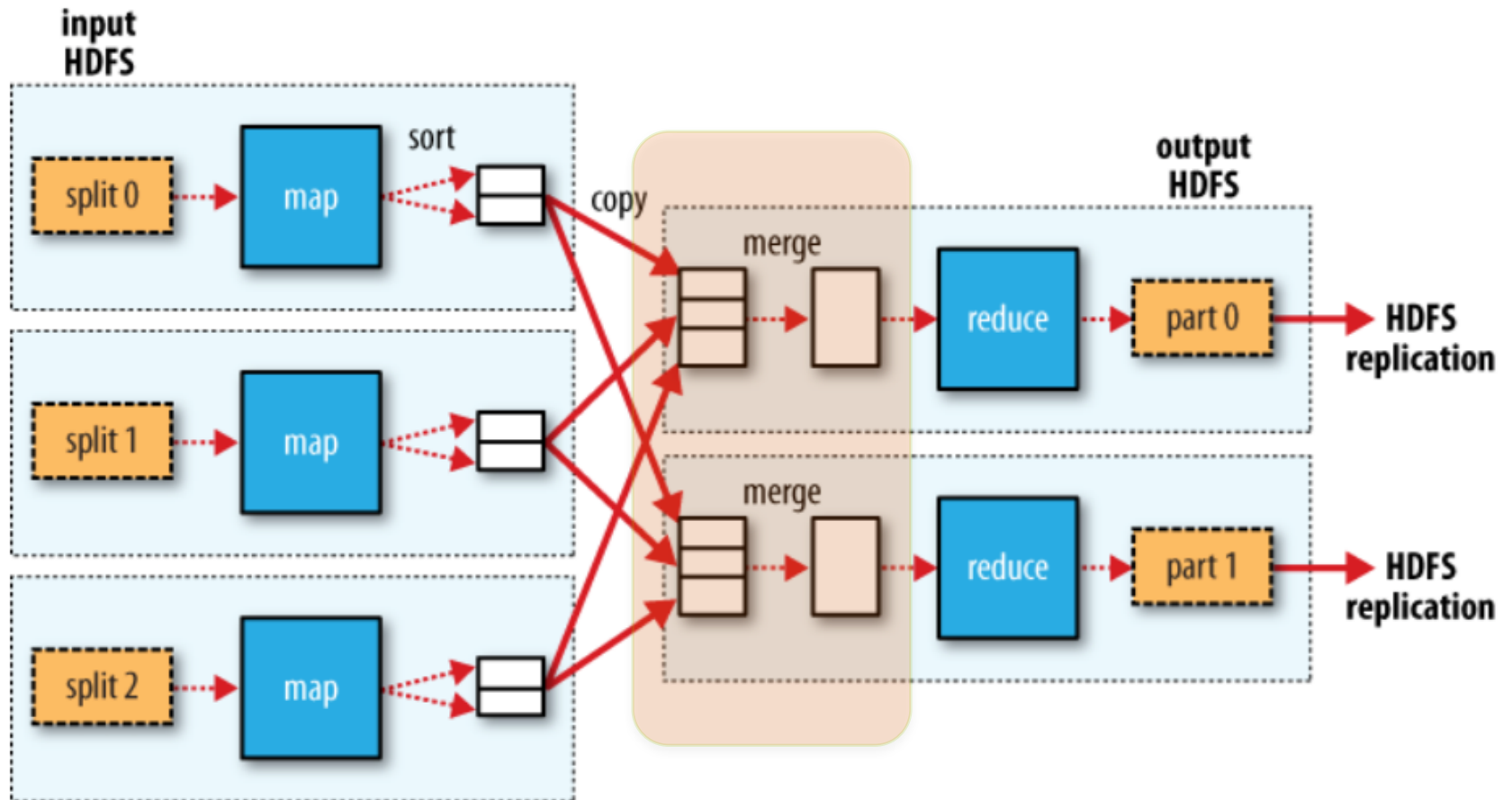
- Reducers merge their sorted input data from different Mappers into a single sorted list
- The sorted input file for Reducer 0:  
[(a,2),(a,3),(a,2),(a,1),(a,2),(a,2),(c,2),(c,3),(c,2),(c,2)]
- The final input file for Reducer 1:  
[(b,3),(b,1),(b,3),(b,1),(b,2),(b,2),(b,2),(d,1),(d,2),(d,1),(d,2),(d,2),(d,2),(d,1)]
- Now we are ready to apply the reduction

## Example continued: Reduce (3)

- ◉ The output for the Reducer 0:  
[(a,12),(c,9)]
- ◉ The output for the Reducer 1:  
[(b,14),(d,11)]
- ◉ A lot of work happen behind the scenes
- ◉ Important to note that the disk is involved
- ◉ While there is a lot of overhead, the overall process **scales** as the number of machines increases
- ◉ Important:
  - ◉ High-performance and scalability are different things



# Distributed Map/Reduce Process



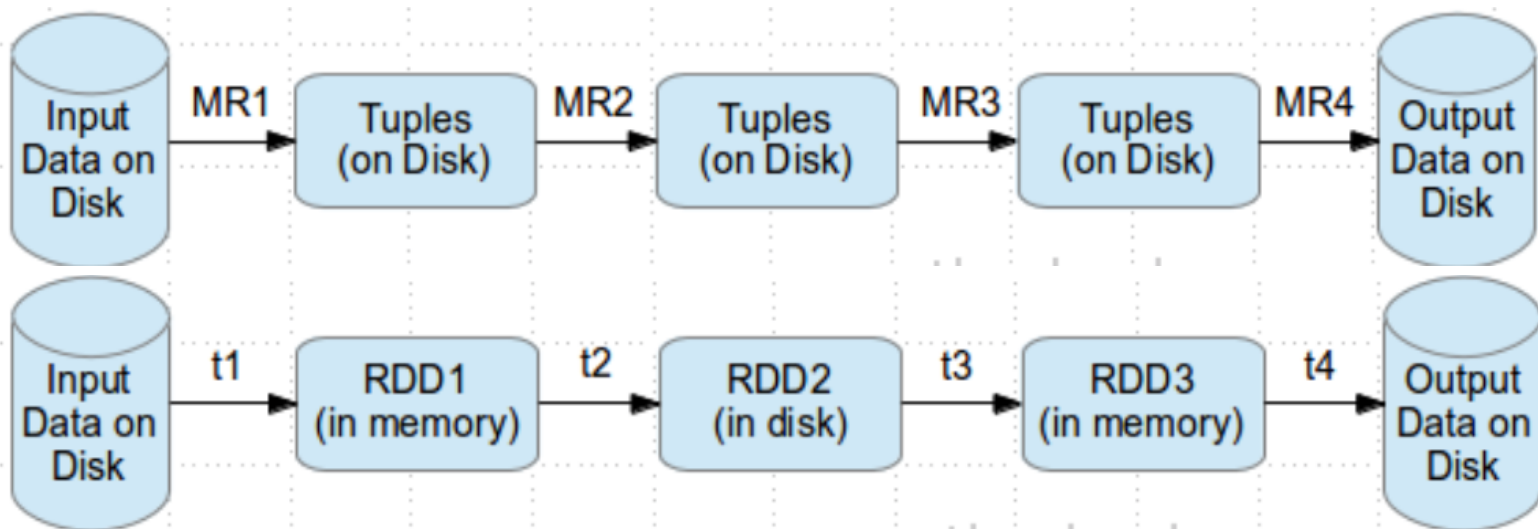
# Apache HADOOP

- Provides HDFS
  - A distributed file system (not a posix file system)
  - Files are replicated across nodes for fault-tolerance
- Provides an M/R runtime
  - M/R jobs are developed using Java APIs
    - Implement a Mapper and a Reducer
  - The runtime handles distribution, execution, fault-tolerance, monitoring, etc.
- Now considered a mature technology



# Resilient Distributed Datasets

- RDD: Read-only, partitioned collection of records
  - Distributed over a set of nodes, replicated for fault-tolerance
  - An RDD is either created from input data on disk, or by applying a transformation over existing RDDs
  - If the RDD fits into memory of multiple nodes, no disk processing is involved
- Comparison of M/R and RDDs



# Apache Spark

- RDD transformations are typically implemented via M/R-like techniques, but without the disk being involved (as long as there is enough memory)
- Apache Spark provides the RDD abstraction
  - Works within the Hadoop ecosystem: HDFS, YARN
  - Supports Scala, Python, Java
  - Supports interactive exploration
    - Consider the word count example:



```
file = spark.textFile("hdfs://...")
counts = file.flatMap(lambda line: line.split(" ")) \
               .map(lambda word: (word, 1)) \
               .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

# A Simple Example: Tf-Idf

- Let's say we have a bunch of lines of text, and we want to compute the tf-idf scores of the words
- Here, a line corresponds to a document
  - tf** of a word in a line: # of times it appears in the line
  - idf** of a word:  $\log(\# \text{ of lines} / \# \text{ of lines word appears})$
  - tf-idf** of a word in a line:  $\text{tf} * \text{idf}$

# Tf-idf in Spark

- Let us compute tf-idf's using Spark
  - Compute idf:
    - For each unique word in a line, output (word, 1) as a pair
    - Reduce pairs across all lines via sum to get the raw idf
    - Map raw idf to idf by dividing to # of lines and taking log
  - Compute tf:
    - For each word in a line, output (word, (id, tf))
      - Here id is the line/doc id (we need to remember it to recombine words of a line with their tf-idfs)
  - Compute tf-idf:
    - Join the words with idfs from before as (word, ((id, tf), idf))
    - Map to compute tf-idfs as (id, (word, tf-idf))
    - Group by key to get (id, [(word, tf-idf)])

# Spark tf-idf Code in Python (1)

- Compute idf:
  - For each unique word in a line, output (word, 1) as a pair
  - Reduce pairs across all lines via sum to compute the idf
  - Map raw idf to idf by dividing to # of lines and taking the log

```
from pyspark import SparkContext
sc = SparkContext("local", "TfIdf App")
```

```
dataFile = sc.textFile("hdfs://...")
N = dataFile.count()
```

```
def idfPreMapper(line):
    wordMap = {}
    for word in line.split():
        wordMap[word] = 1
    return wordMap.iteritems()
```

```
sumReducer = lambda x, y: x + y
```

```
def idfPostMapper(word_count):
    (word, count) = word_count
    return (word, math.log(N/count))
```

```
idf = dataFile.flatMap(idfPreMapper)\
    .reduceByKey(sumReducer)\
    .map(idfPostMapper)
```

# Spark tf-idf Code in Python (2)

- Compute tf:
  - For each word in a line, output (word, (id, tf))
    - Here id is the line/doc id (we need to remember it to re-combine words of a line with their tf-idfs)

```
def tfMapper(line):  
    wordMap = defaultdict(int)  
    for word in line.split():  
        wordMap[word] += 1  
    result = []  
    doc = uuid.uuid1()  
    for (word, tf) in wordMap.iteritems():  
        wordTf = (word, (doc, tf))  
        result.append(wordTf)  
    return result  
  
tf = dataFile.flatMap(tfMapper)
```



# Spark tf-idf Code in Python (3)

- Compute tf-idf:
  - Join the words with idfs from before to get (word, ((id, tf), idf))
  - Map to compute tf-idfs as (id, (word, tf-idf))
  - Group by key to get (id, [(word, tf-idf)])

```
def tfIdfJoiner(word_docTfAndIdf):  
    word = word_docTfAndIdf[0]  
    ((doc, tf), idf) = word_docTfAndIdf[1]  
    return (doc, (word, tf * idf))  
  
def tfIdfMapper(doc_wordAndTfIdfSeq):  
    return list(doc_wordAndTfIdfSeq[1])  
  
tfIdf = tf.join(idf)\  
    .map(tfIdfJoiner)\  
    .groupByKey().map(tfIdfMapper)  
  
tfIdf.saveAsTextFile("hdfs://...")
```

# Bulk Synchronous Parallel

- BSP is a parallel computational model that consists of a series of supersteps
- A superstep consists of three ordered stages:
  - Concurrent computation: Computation on locally stored data
  - Communication: Send and receive/messages in a point-to-point manner
  - Barrier synchronization: Wait and synchronize all processors at the end of superstep
- A BSP system consists of a number of networked computers with both local memory and disk

# Apache HAMA & Giraph

- Apache HAMA is a general purpose BSP framework on top of Hadoop HDFS
  - There are some machine learning/data mining algorithms implemented on top of it
- Apache Giraph is a graph mining framework using the BSP model
- We will cover BSP-style graph processing in more details later in the presentation



# Mining Big Data with



- Mahout is a Data Mining tool that works on large-scale data
  - Data is stored on a set of distributed hosts
    - HDFS is used
  - Processing is done in a scalable and distributed manner
    - There are different processing back-ends
- Back-ends supported in version 0.9
  - Single-machine
  - Hadoop Map/Reduce
- Back-ends supported in version 1.0 (unreleased)
  - **Spark** – considered as the the main driving fore
  - H2O – these are at a very early stage
  - Flink – these are at a very early stage

# Algorithms and Back-ends (1.0)

	Single Machine	MapReduce	Spark
<b>Collaborative Filtering</b> <i>with CLI Drivers</i>			
User-Based Collaborative Filtering	x		x
Item-Based Collaborative Filtering	x	x	x
Matrix Factorization with ALS	x	x	
Matrix Factorization with ALS on Implicit Feedback	x	x	
Weighted Matrix Factorization, SVD++	x		
<b>Classification</b> <i>with CLI Drivers</i>			
Logistic Regression - trained via SGD	x		
Naive Bayes / Complementary Naive Bayes		x	x
Random Forest		x	
Hidden Markov Models	x		
Multilayer Perceptron	x		
<b>Clustering</b> <i>with CLI Drivers</i>			
Canopy Clustering	deprecated	deprecated	
k-Means Clustering	x	x	
Fuzzy k-Means	x	x	
Streaming k-Means	x	x	
Spectral Clustering		x	

Enables running these algorithms from the command line, rather than using programming APIs

We will look at these algorithm in more detail

# Algorithms and Back-ends (1.0)

	Single Machine	MapReduce	Spark
<b>Dimensionality Reduction with CLI Drivers</b>			
Singular Value Decomposition	x	x	
Lanczos Algorithm	x	x	
Stochastic SVD	x	x	
PCA (via Stochastic SVD)	x	x	
QR Decomposition	x	x	
<b>Topic Models</b>			
Latent Dirichlet Allocation	x	x	
<b>Miscellaneous</b>			
RowSimilarityJob		x	x
ConcatMatrices		x	
Collocations		x	
Sparse TF-IDF Vectors from Text		x	
XML Parsing		x	
Email Archive Parsing		x	
Lucene Integration		x	
Evolutionary Processes	x		

# Using the CLI Interface

- Let us do an example Naive Bayes classification
- Problem:
  - Input: A large set of emails that have labels identifying their topics
  - Output: A model that can classify emails by assigning them a topic
- Data is structured as follows on disk/HDFS:

The screenshot shows a file system view of the '20news-all' directory. It contains a list of subdirectories representing different topics (labels) and a corresponding list of file IDs. A green arrow points from the word 'Label' to the 'rec.sport.baseball' directory. Another green arrow points from the file ID '100521' to the text 'file name represents data item id'. A third green arrow points from the text 'content is the data item' to the file ID '102587'.

Label	File ID
comp.graphics	98657
comp.os.ms-windows.misc	99971
comp.sys.ibm.pc.hardware	100521
comp.sys.mac.hardware	101666
comp.windows.x	102151
misc.forsale	102584
rec.autos	102585
rec.motorcycles	102586
rec.sport.baseball	102587
rec.sport.hockey	102588
sci.crypt	102589
sci.electronics	102590
sci.med	102591
sci.space	102592
soc.religion.christian	102593

file name represents data item id

content is the data item

# Email Classification

- An example data item:

From: admiral@jhunix.hcf.jhu.edu (Steve C Liu)  
Subject: spring records  
Organization: Homewood Academic Computing, Johns Hopkins University, Baltimore, Md, USA  
Lines: 26  
Distribution: world  
Expires: 5/9/93  
NNTP-Posting-Host: jhunix.hcf.jhu.edu  
Summary: Earl Weaver Commandments

The Orioles' pitching staff again is having a fine exhibition season. Four shutouts, low team ERA, (Well, I haven't gotten any baseball news since March 14 but anyways) Could they contend, yes. Could they win it all? Maybe.

But for all those fans of teams with bad spring records, remember Earl Weaver's first law of baseball (From his book on managing)

No one gives a damn in July if you lost a game in March. :)

BTW, anyone have any idea on the contenders for the O's fifth starter? It's pretty much set that Sutcliffe, Mussina, McDonald and Rhodes are the first four in the rotation.

Here at Johns Hopkins University where the mascot is the Blue Jay :(, their baseball team logo was the Toronto club's logo. Now it's a anatomically correct blue jay. God, can't they think of an original idea? It's even in the same pose as the baltimore oriole on the O's hats. How many people realize that the bird is really called a baltimore oriole?

---

|Admiral Steve C. Liu            Internet Address: admiral@jhunix.hcf.jhu.edu|  
|"Committee for the Liberation and Intergration of Terrifying Organisms |  
|and their Rehabilitation Into Society" from Red Dwarf - "Polymorph" |  
|\*\*\*The Bangles are the greatest female rock band that ever existed!\*\*\*|  
|    This sig has been brought to you by... Frungy! The Sport of Kings! |

---



# Naive Bayes via CLI Interface

- First step: Convert data into *sequence files* so that Mahout will be able to process them efficiently

```
mahout seqdirectory
```

```
-i 20news-all # input directory  
-o 20news-seq # output directory  
-ow # overwrite if exists
```

- Second step: Convert sequence files into sparse vector form

```
mahout seq2sparse
```

```
-i 20news-seq # input directory  
-o 20news-vectors # output directory  
-wt tfidf # use tf-idf as weighted  
-lnorm # log normalize the vectors ( $\log_2(1+b)/L_2(v)$ )  
-nv # create named vectors
```

- Other interesting options:

- use `-ng <k>` for creating n-grams
- use `-n <k>` to normalize vectors using `L<k>-norm`

# Naive Bayes via CLI Interface

- Third step: Create training and holdout set with a random 80-20 split of the vector dataset

```
mahout split
```

```
-i 20news-vectors/tfidf-vectors # input dir
--trainingOutput 20news-train-vectors # training set
--testOutput 20news-test-vectors # test set
--randomSelectionPct 20 # 80-20 split
--overwrite # overwrite if existing
--sequenceFiles # generate sequence files
-xm sequential # do not run this as an M/R job
```

- Fourth step: Train the Naive Bayes Model

```
mahout trainnb
```

```
-i 20news-train-vectors # input directory
-el # extract labels from the input
-o model # the directory for the model
-li labelindex # index2label mapping file
-overwrite # overwrite the model if exists
```

# Naive Bayes via CLI Interface

- Final step: Testing the accuracy of the classifier

mahout testnb

- i 20news-test-vectors # input test vectors
- m model # model file
- l labelindex # index2label mapping
- o 20news-testing # results file
- overwrite # overwrite results if there

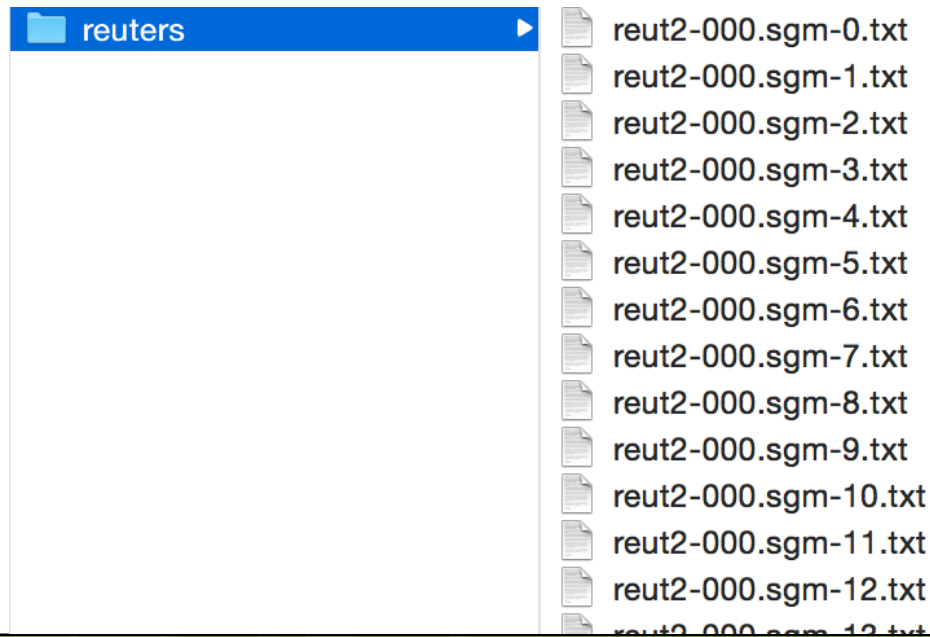
- Test results:

```
=====
Summary
-----
Correctly Classified Instances      :    3341      92.2673%
Incorrectly Classified Instances   :      280      7.7327%
Total Classified Instances        :    3621

=====
Confusion Matrix
-----
a      b      c      d      e      f      g      h      i      j      k      l      m      n      o      p      q      r      s      <--Classified as
196    3        5        1        6        0        0        0        1        0        0        2        1        0        0        0        0        0        0      |    215      a      = comp.graphics
21    141    29    10        9        3        0        0        0        0        0        0        0        0        0        0        0        0        0      |    213      b      = comp.os.ms-windows.misc
5        0    168        4        0        4        0        0        0        0        0        2        0        0        0        0        0        0        1      |    184      c      = comp.sys.ibm.pc.hardware
3        0        6    180        0        2        0        0        0        0        0        2        1        0        0        0        0        0        0      |    194      d      = comp.sys.mac.hardware
11    1        1        2    186        0        1        0        1        0        0        0        1        0        0        0        0        0        0      |    204      e      = comp.windows.x
0        0    10        2        0    157        4        0        1        0        1        3        0        1        0        0        0        0        0      |    179      f      = misc.forsale
0        0        0        0        1        4    209        2        0        0        0        1        0        1        0        1        0        1        0      |    220      g      = rec.autos
0        0        0        0        0        2        4    165        0        0        0        1        1        0        0        0        0        0        0      |    173      h      = rec.motorcycles
0        0        0        0        0        0        0        1    182        2        0        1        1        0        0        0        0        0        0      |    187      i      = rec.sport.baseball
0        0        0        0        0        0        0        0        2    217        0        0        0        0        1        0        0        1        0      |    221      j      = rec.sport.hockey
1        1        0        1        0        0        0        0        0        0    188        0        2        0        0        0        0        0        1      |    194      k      = sci.crypt
2        0        5        3        1        3        3        1        0        0        1    195        0        1        0        0        0        0        0      |    215      l      = sci.electronics
0        0        3        2        0        0        1        0        0        0        0        3    157        1        0        0        1        0        1      |    169      m      = sci.med
1        0        0        2        0        0        1        0        0        0        0        0        1    189        0        0        1        0      |    195      n      = sci.space
1        0        0        0        0        1        0        0        0        0        0        0        2        0    182        1        1        0        2      |    190      o      = soc.religion.christian
0        0        0        0        0        0        0        0        0        0        1        0        1        1        0    186        0        3        0      |    192      p      = talk.politics.guns
0        0        0        0        0        0        0        1        0        0        0        0        0        0        1        0    195        0        0      |    197      q      = talk.politics.mideast
1        0        0        1        0        0        0        0        0        1        1        0        0        2        0        4        2    136        2      |    150      r      = talk.politics.misc
0        0        0        0        0        0        1        1        0        0        0        0        0        1        7        1        2        4    112      |    129      s      = talk.religion.misc
```

# k-Means Clustering Example

- Problem:
  - Input: A large set of text files
  - Output: Clustering of the text files into  $k$  groups, and for each group, the cluster centroid (that is a sparse word vector with term weights)
- Input data is structured as a list of files on disk/HDFS:



# News Clustering

- An example data item:

26-FEB-1987 16:57:27.21

## IVACO SEES MINIMAL FIRST QUARTER EARNINGS

(Ivaco Inc) said price pressure on steel products, particularly in the U.S., and the recent increase in the value of the Canadian dollar is expected to result in "minimal" first quarter earnings. It said subsequent quarters should show substantial improvement from first quarter levels but 1987 earnings will not reach 1986 levels as long as those conditions continue. Ivaco earlier reported 1986 profit rose to 44.1 mln dlrs, after a one mln dlr extraordinary gain, from 35.1 mln dlrs the previous year. It said demand for the company's products are continuing at high levels and sales are expected to show further growth. Revenues last year rose to 1.94 billion dlrs from 1.34 billion dlrs in 1985. Reuter &#3;

# k-Means Clustering via CLI

- First step: Convert data into *sequence files* so that Mahout will be able to process them efficiently

```
mahout seqdirectory
```

```
-i reuters # input directory  
-o reuters-seqdir # output directory  
-c UTF-8 # character encoding  
-chunk 64 # chunk size in MBs
```

- Second step: Convert sequence files into sparse vector form

```
mahout seq2sparse
```

```
-i reuters-seqdir # input directory  
-o reuters-sparse # output directory  
--maxDFPercent 85 # ignore frequent words  
--namedVector # use named vectors
```

# k-Means Clustering via CLI

- Third step: Perform the clustering

```
mahout kmeans
```

```
-i reuters-sparse/tfidf-vectors # input directory  
-c reuters-centroids # initial centroids (populated)  
-o reuters-kmeans # output directory  
-dm org.apache.mahout.common.distance.CosineDistanceMeasure # distance measure  
-x 10 # number of iterations  
-k 20 # number of clusters  
-ow # overwrite if existing  
--clustering # cluster after the iterations
```

- Fourth step: View the results

```
mahout clusterdump
```

```
-i reuters-kmeans/clusters-*-final # input clusters  
-o reuters-kmeans/clusterdump # output file  
-d reuters-sparse/dictionary.file-0 # dictionary  
-dt sequencefile # dictionary format  
-b 100 # limit the length of string representation of clusters  
-n 20 # number of top terms to print  
--evaluate # evaluate how good the clustering is  
-dm org.apache.mahout.common.distance.CosineDistanceMeasure # distance measure  
-sp 0 # sample points to show in each cluster  
-pointsDir reuters-kmeans/clusteredPoints # points directory
```

# Sample clusterdump output

```
:VL-5762{n=2259 c=[0:0.121, 0.006913:0.004, 0.007050:0.009, 0.01:0.004, 0.02:0.004, 0.05:0.004, 0.1:0
```

```
Top Terms:
```

he	=> 2.793883439976328
said	=> 2.7000598058388157
u.s	=> 2.438783091430698
would	=> 2.4318358077048834
trade	=> 2.2482740508607657
have	=> 1.922344003329503
japan	=> 1.6248975570563882
has	=> 1.6227564276610396
told	=> 1.6091087686640861
which	=> 1.5353790651965638
we	=> 1.5125704430119555
had	=> 1.4770049531674059
japanese	=> 1.3788979858780077
minister	=> 1.3669306590209866
from	=> 1.3585378379619137
officials	=> 1.3572178405172375
countries	=> 1.345225612797427
foreign	=> 1.307425490096904
been	=> 1.3069350678290663
its	=> 1.3047775451774986

```
Weight : [props - optional]: Point:
```

```
:VL-15211{n=507 c=[0:0.136, 0.01:0.024, 0.02:0.018, 0.07:0.029, 0.1:0.052, 0.10:0.048, 0.11:0.093, 0.
```

```
Top Terms:
```

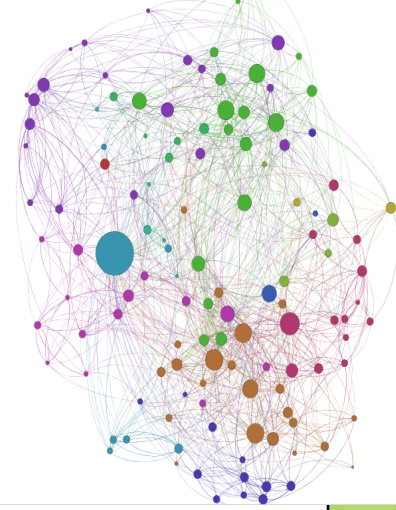
department	=> 2.943879099287225
usda	=> 2.4856224549122348
u.s	=> 2.361959061914177
agriculture	=> 2.3341514485827566
said	=> 1.9690223873484534
program	=> 1.447121717990973
corn	=> 1.431495368363119
mln	=> 1.1333308436226328
from	=> 1.0858395052614531
has	=> 1.081101337128137
ccc	=> 1.0708439862704606
under	=> 0.9977508564672527

```
reuters-kmeans/clusterdump
```

```
Inter-Cluster Density: 0.5804128317901368  
Intra-Cluster Density: 0.7013755618650461  
CDBw Inter-Cluster Density: 0.0  
CDBw Intra-Cluster Density: 19.912664116551753  
CDBw Separation: 217.5813153713576
```



# Graph Processing



- Graph data is everywhere
  - *Relationships* between people, systems, and the nature
  - *Interactions* between people, systems, and the nature
- Relationship graphs
  - Social media: Twitter follower-followee graph, Facebook friendship graph, etc.
  - The web: The link graph of web pages
  - Transportation networks, biological networks, etc.
- Interaction graphs
  - Social media: Mention graph of twitter
  - Telecommunications: Call Detail Records (CDRs)
- Interaction graphs can be summarized to form relationship graphs

# Applications

- ◉ Finding influence for ranking
  - ◉ Pages that are influential within the web graph (PageRank)
  - ◉ Users that are influential within the Twitter graph (TweetRank)
- ◉ Community detection for recommendations, churn prediction
  - ◉ If X is in the same community with Y, they may have similar interests
  - ◉ If X has churned, Y might be likely to churn as well
- ◉ Diffusion for targeted advertising
  - ◉ Start with known users in the graph with known interests, diffuse to others
- ◉ Regular path queries and graph matching for surveillance

# Graph Processing vs Management

- Graphs pose challenges in processing and management
- RDBMS are inadequate for graph analytics
  - Traditional graph algorithms require *traversals* (e.g., BFS, DFS)
  - Traversals require recursive SQL: difficult to write, costly to execute
- Large-scale graphs require distributed systems for scalability
- Management vs Processing
  - Management: CRUD operations (Create, Read, Update, Delete)
  - Processing: Graph analytics (BFS, Connected Components, Community Detection, Clustering Coefficient, PageRank, etc.)
- Systems may support one or both
  - This talk focus on *graph processing systems*, with a focus on distributed ones

# Distributed Graph Processing Systems

- Graph data stays on the disk, typically in a distributed file system
  - E.g., graph data is on HDFS, in the form of *list of edges*
- To perform a graph analytic, the graph is loaded from the disk to the memory of a set of processing nodes
- The graph analytic is performed in-memory, using multiple nodes, typically requiring communication between them
- The graph could be potentially morphed during the processing
- The results (which could be a graph as well) are written back to disk
- Overall, it is a batch process
  - E.g., Compute the PageRank over the current snapshot of the web graph
- *Advantages*: Fast due to in-memory processing, scalable with increasing number of processing nodes

# Some Approaches

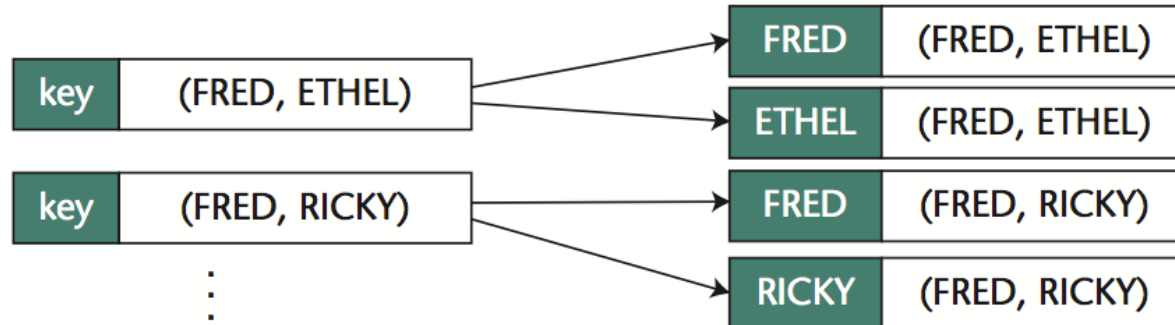
- ◉ Apache Hadoop & Map/Reduce
  - ◉ Use Map/Reduce framework for executing graph analytics
- ◉ Vertex Programming
  - ◉ A new model of processing specifically designed for graphs
  - ◉ Synchronous model
    - ◉ Foundational work: Pregel from Google
    - ◉ Pregel clones: Apache Giraph and HAMA (more general)
  - ◉ Asynchronous model
    - ◉ GraphLab, PowerGraph
    - ◉ Disk-based variety: GraphChi

# Example: Degree Computation

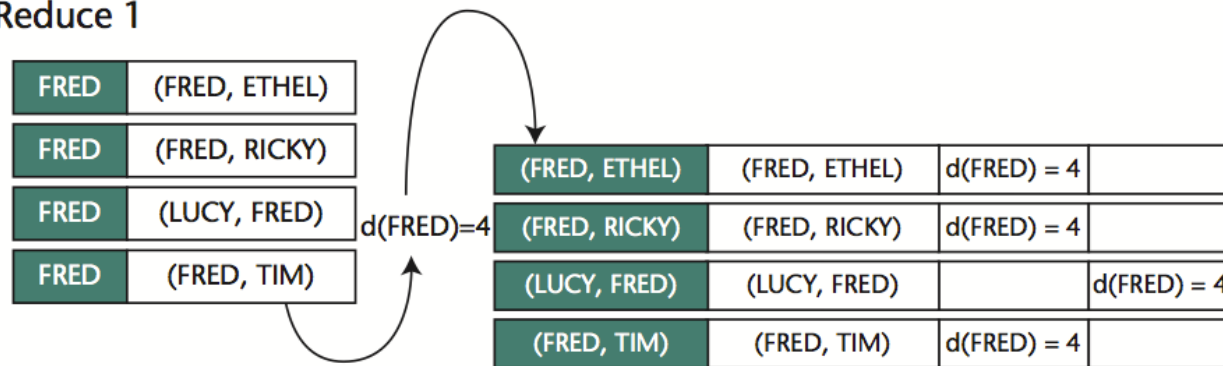
- Out-degree computation
  - Source Data: (from\_vertex, to\_vertex)
  - Mapper: (from\_vertex, to\_vertex)  $\Rightarrow$  key: from\_vertex, value: 1
  - Reducer: key: vertex, values: [1, 1, ...]  $\Rightarrow$  (vertex, vertex\_degree)
- Degree computation
  - Source Data: (from\_vertex, to\_vertex)
  - Mapper: (from\_vertex, to\_vertex)  $\Rightarrow$  key: from\_vertex, value: 1  
key: to\_vertex, value: 1
  - Reducer: key: vertex, values: [1, 1, ...]  $\Rightarrow$  (vertex, vertex\_degree)
- What if you want to augment each edge with the degrees of the vertices involved:  $(u, v) \Rightarrow (u, v, d(u), d(v))$ 
  - We can add one job to add the  $d(u)$ , another to add  $d(v)$
  - Can we do this using less number of jobs?

# Example: Degree Augmentation (1)

Map 1



Reduce 1



# Example: Degree Augmentation (2)

Map 2 (Identity)

(FRED, ETHEL)	(FRED, ETHEL)	d(FRED) = 4	
---------------	---------------	-------------	--



(FRED, ETHEL)	(FRED, ETHEL)	d(FRED) = 4	
---------------	---------------	-------------	--

⋮

Reduce 2

(FRED, ETHEL)	(FRED, ETHEL)	d(FRED) = 4	
---------------	---------------	-------------	--

(FRED, ETHEL)	(FRED, ETHEL)		d(ETHEL) = 2
---------------	---------------	--	--------------



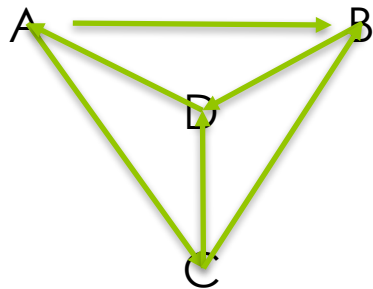
(FRED, ETHEL)	(FRED, ETHEL)	d(FRED) = 4	d(ETHEL) = 2
---------------	---------------	-------------	--------------



# Example: PageRank

- Probability of a web surfer being at a particular page under the *random surfer model*
- Random surfer model:
  - The surfer starts from a random page
  - With probability  $d$ , she continues surfing by following one of the outlinks on the page at random
  - With probability  $(1-d)$ , she jumps to random page
- Let  $p_i$  be the PageRank of page  $i$ ,  $N$  be the total number of pages,  $M(i)$  be the pages that link to page  $i$ , and  $L(i)$  be the out-degree of page  $i$ 
  - $p_i = (1-d) / N + d * \sum_{j \in M(i)} p_j / L(j)$
- Iterative implementation
  - Start with all pages having a PageRank of  $1/N$
  - Apply the formula above to update it each page's PageRank using page rank values from the last step
  - Repeat fixed number of times or until convergence
  - Note: pages with no outgoing links need special handling (assumed as if they link to all other pages)

# PageRank M/R Style



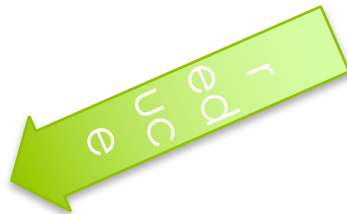
one iteration



$v$	$PR(v), Neig(v)$
A	$PR(A), [B, C]$
B	$PR(B), [D]$
C	$PR(C), [B, D]$
D	$PR(D), [A]$

source data

A	$PR(A), [B, C]$	B	$PR(B), [D]$	C	$PR(C), [B, D]$	D	$PR(D), [A]$
B	$PR(A)/2$	D	$PR(B)$	B	$PR(C)/2$	A	$PR(D)$
C	$PR(A)/2$			D	$PR(C)/2$		



A	$PR(A), [B, C]$	B	$PR(B), [D]$	C	$PR(C), [B, D]$	D	$PR(D), [A]$
A	$PR(D)$	B	$PR(A)/2$	C	$PR(A)/2$	D	$PR(B)$
...		B	$PR(C)/2$	...		D	$PR(C)/2$
		B	$(1-d)/N +$			...	
			$d * (PR(A)/2 + PR(C)/2), [D]$				

# Vertex Programming (sync.)

- Graph analytics are written from the perspective of a vertex
  - You are programming a single vertex
  - The vertex program is executed for all of the vertices
- There are a few basic principles governing the execution
  - Each vertex maintains its own data
  - The execution proceeds in *supersteps*
  - At each superstep
    - the vertex program is executed for all vertices
  - Between two supersteps
    - Messages sent during the previous superstep are delivered

# Super Steps

- ◉ During a superstep, the vertex program can do the following:
  - ◉ Access the list of messages sent to it during the last superstep
  - ◉ Update the state of the vertex
  - ◉ Send messages to other vertices
    - ◉ these will be delivered in the next superstep
  - ◉ Vote to halt, if done
- ◉ Each vertex has access to vertex ids of its neighbors
- ◉ Vertex ids are used for addressing messages
- ◉ Messages can be sent to neighbor vertices or any other vertex (as long as the vertex id is learnt by some means, such as through messages exchanged earlier)
- ◉ The execution continues until no more supersteps can be performed, which happens when:
  - ◉ There are no pending messages
  - ◉ There are no non-halted vertices

# BSP & Pregel

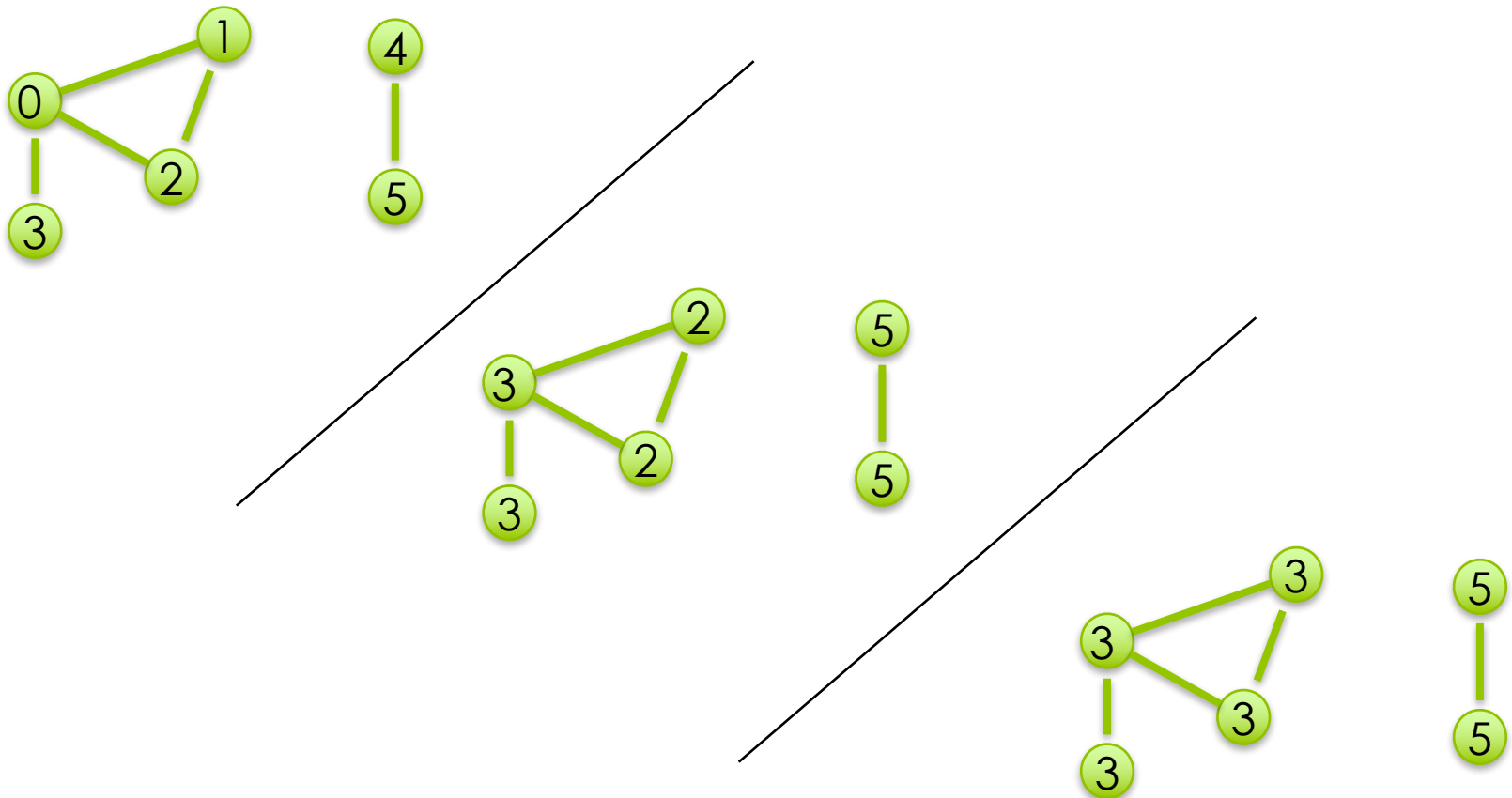
- Vertex programs can be executed in a scalable manner using the Bulk Synchronous Processing (BSP) paradigm
- Pregel system by Google (research paper, code not available) does that
- Vertices are distributed to machines using some partitioning
  - The default is a hash based partitioning (on the vertex id)
- At each superstep, each machine executes the vertex program for the vertices it hosts (keeps the state for those vertices as well)
- At the end of the superstep, messages that need to cross machine boundaries are transported
- Pregel also supports additional abstractions
  - Aggregations: Reduction functions that can be applied on vertex values
  - Combiners: Reduction functions that can be applied to messages destined to the same vertex from different vertices
  - Ability to remove vertices (morphing the graph)

# Example: Connected Components

- Vertex state
  - Just an int value, representing the id of the connected component the vertex belongs
- The vertex program

```
    if (superstep() == 0)
        getVertexValue() = getVertexId();
        sendMessageToAllNeighbors(getVertexId());
    } else {
        int mid = getVertexValue();
        for (msg in getReceivedMessages())
            mid = max(mid, msg.getValue());
        if (mid != getVertexValue()) {
            getVertexValue() = mid;
            sendMessageToAllNeighbors(mid);
        }
        VoteToHalt();
    }
}
```

# Example: Execution



# Example: PageRank

- Vertex state
  - Just a double value, representing the PageRank
- The vertex program

```
if (superstep() >= 1) {  
    double sum = 0;  
    for (msg in getReceivedMessages())  
        sum += msg.getValue();  
    getVertexValue() = 0.15 /  
        getNumVertices() + 0.85 * sum;  
}  
if (superstep() < 30) {  
    int64 n = getNumOutEdges();  
    sendMessageToOutNeighbors(getVertexValue() / n);  
} else {  
    VoteToHalt();  
}
```

Surprisingly simple

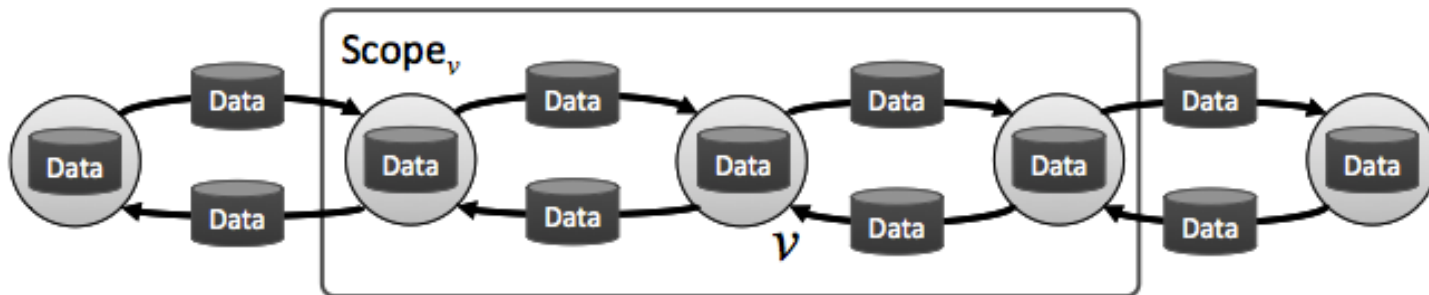


# Systems Issues

- Scalability
  - Better than M/R for most graph analytics
  - Minimizing communication is key (many papers on improved partitioning to take advantage of high locality present in graph analytics)
  - Skew could be an issue for power graphs where there are a few number of very high degree vertices (results in imbalance)
- Fault tolerance
  - Checkpointing between supersteps, say every x supersteps, or every y seconds
- Example Open Source Systems
  - Apache Giraph: Pregel-like
  - Apache HAMA: More general BSP framework for data mining/machine learning

# Asynchronous VP & GraphLab

- GraphLab targets not just graph processing, but also iterative Machine Learning and Data Mining algorithms
- Similar to Pregel but with important differences
  - It is *asynchronous* and supports *dynamic execution schedules*
- Vertex programs in GraphLab
  - Access vertex data, adjacent edge data, adjacent vertex data
    - These are called the *scope*
    - No messages as in Pregel, but similar in nature
  - Update any of the things in scope as a result of execution
  - Return a list of vertices, that will be scheduled for execution in the future



# GraphLab Continued

- Asynchronous
  - Pregel works in supersteps, with synchronization in-between
  - Graphlab works asynchronously
    - It is shown that this improves convergence for many iterative data mining algorithms
- Dynamic execution schedule
  - Pregel executes each vertex at each superstep
  - In GraphLab, new vertices to be executed are determined as a result of previous vertex executions
- GraphLab also supports
  - Configuring the *consistency model*: determines the extent to which computation can overlap
  - Configuring the *scheduling*: determines the order in which vertices are scheduled (synchronous, round-robin, etc.)
- GraphLab has multi-core (single machine) and distributed versions

- Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, Joseph M. Hellerstein: GraphLab: A New Framework For Parallel Machine Learning. UAI 2010: 340-349
- Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, Joseph M. Hellerstein: Distributed GraphLab: A Framework for Machine Learning in the Cloud. PVLDB 5(8): 716-727 (2012)

# Example: PageRank

PageRank

scope

**Input:** Vertex data  $\mathbf{R}(v)$  from  $\mathcal{S}_v$

**Input:** Edge data  $\{w_{u,v} : u \in \mathbf{N}[v]\}$  from  $\mathcal{S}_v$

**Input:** Neighbor vertex data  $\{\mathbf{R}(u) : u \in \mathbf{N}[v]\}$  from  $\mathcal{S}_v$

$\mathbf{R}_{old}(v) \leftarrow \mathbf{R}(v)$  // Save old PageRank

$\mathbf{R}(v) \leftarrow \alpha/n$

**foreach**  $u \in \mathbf{N}[v]$  **do** // Loop over neighbors

$\mathbf{R}(v) \leftarrow \mathbf{R}(v) + (1 - \alpha) * w_{u,v} * \mathbf{R}(u)$

// If the PageRank changes sufficiently

**if**  $|\mathbf{R}(v) - \mathbf{R}_{old}(v)| > \epsilon$  **then**

    // Schedule neighbors to be updated

**return**  $\{u : u \in \mathbf{N}[v]\}$

**Output:** Modified scope  $\mathcal{S}_v$  with new  $\mathbf{R}(v)$

The list of vertices to be added to the scheduler's list

```
graph TD
    PR[PageRank] --> S[scope]
    S --> I3[Input: Neighbor vertex data {R(u) : u in N[v]} from S_v]
    S --> R_stmt[return {u : u in N[v]}]
    R_stmt --> O[The list of vertices to be added to the scheduler's list]
```

# Understanding Dynamic Scheduling

- A very high level view of the execution model

**Input:** Data Graph  $G = (V, E, D)$

**Input:** Initial vertex set  $\mathcal{T} = \{v_1, v_2, \dots\}$

**while**  $\mathcal{T}$  is not Empty **do**

Vertex to  
be  
executed

$v \leftarrow \text{RemoveNext}(\mathcal{T})$

Apply the  
vertex  
program

$(\mathcal{T}', \mathcal{S}_v) \leftarrow f(v, \mathcal{S}_v)$

$\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}'$

**Output:** Modified Data Graph  $G = (V, E, D')$

Vertices to be  
scheduled

Updated scope

- Consistency model adjusts how the execution is performed in parallel
- Scheduling adjusts how the RemoveNext method is implemented

# More on GraphLab

- GraphLab also supports
  - Global aggregations over vertex values, which are read-only accessible to vertex programs
  - Unlike Pregel, these are computed continuously in the background
- PowerGraph is a GraphLab variant
  - Specially designed for scale-free graphs
    - Degree distribution follows a power law
    - $P(k) \sim k^{-\gamma}$        $k$ : degree,  $\gamma$ : typically in the range  $2 < \gamma < 3$
  - Main idea is to decompose the vertex program into 3 steps
    - Gather: Collect data from the scope
    - Apply: Compute the value of the central vertex
    - Scatter: Update the data on adjacent vertices
  - This way a single vertex with very high-degree can be distributed over multiple nodes  
(partition edges not vertices)
- GraphChi is another GraphLab variant
  - It is designed for disk-based, single machine processing
  - The main idea is a disk layout technique that can be used to execute vertex programs by doing mostly sequential I/O (potentially parallel)

- Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, Carlos Guestrin: PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. OSDI 2012: 17-30
- Aapo Kyrola, Guy E. Blelloch, Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. OSDI 2012: 31-46

# Other Systems

- GraphX
  - Built on Spark RDD
  - Supports Graph ETL tasks, such as graph creation and transformation
  - Supports interactive data analysis (kind of like PigLatin of the graph world)
  - Low-level, can be used to implement Pregel and GraphLab
- Boost Parallel BGP
  - SPMD approach with support for distributed data structures
  - Many graph algorithms are already implemented
  - No fault-tolerance

- Douglas Gregor and Andrew Lumsdaine. Lifting Sequential Graph Algorithms for Distributed-Memory Parallel Computation. In Proceedings of the 2005 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05), October 2005.
- Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica: GraphX: A resilient distributed graph system on Spark. GRADES 2013: 2

# Questions

- ???