Transition

Cluster analysis

- Provides a quick and meaningful overview of data
- Improves efficiency of data mining by combining data with similar characteristics so that
  - a generalization can be derived for each cluster and
  - hence processing is done batch wise rather than individually
- • Gives a good understanding of the unusual similarities that may occur once the clustering is complete
- Provides a really good base for
  - nearest neighbouring
  - ordination of deeper relations

Parallel K-means: practical experience

Genoveva Vargas-Solar
French Council of Scientific Research, LIG & LAFMIA Labs
Javier Espinosa
Barcelona Supercomputing Centre & LAFMIA Lab

Montevideo, 22nd November – 4th December, 2015
http://www.vargas-solar.com/big-data-analytics

K-means

P = (x1(P), x2(P), x3(P) .....) and Q = (x1(Q), x2(Q), x3(Q) .....).

- Compute the distance d(P,Q)
- Compute cluster **centroid**: the point whose coordinates corresponds to the mean of the coordinates of all the points in the cluster
- The data set will have certain items that may not be related to any cluster and that cannot be classified under them,
  - Such points are referred to as **outliers**
  - Often correspond to the extremes of the data set depending on whether their values or extremely high or low
- **Objective**: obtain a minimal squared difference between the centroid of the cluster and the item in the dataset

K-means

- **Objective**: obtain a minimal squared difference between the centroid of the cluster and the item in the dataset

$$|\mathbf{x}_i^{(j)} - \mathbf{c}_j|^2$$

- Where x_i is the value of the item and c_j is the value of the centroid of the cluster

K-means steps

- The required number of cluster must be chosen: ‘K’
- Choose distant and distinct centroids for each of the chosen set of K clusters
- Consider each element of the given set and compare its distance to all the centroids of the K clusters.
  - Based on the calculated distance the element is added to the cluster whose centroid is nearest to the element
- The cluster centroids are re-calculated after each assignment or a set of assignments
- Iterative method and continuously updated

Map reduced K-means

Prajesh P Anchalia, Anjan K Koundinya, Srinath N K, MapReduce Design of K-Means Clustering Algorithm, IEEE, 2013

Design steps

- Define and handle the input and output of the implementation.
  - The input is given as a <key, value> pair

General principle

- In the map step
  - Read the cluster centers into memory from a sequencefile
  - Iterate over each cluster center for each input key/value pair.
  - Measure the distances and save the nearest center which has the lowest distance to the vector
  - Write the clustercenter with its vector to the filesystem.
- In the reduce step (we get associated vectors for each center)
  - Iterate over each value vector and calculate the average vector. (Sum each vector and devide each part by the number of vectors we received).
  - This is the new center, save it into a SequenceFile.
  - Check the convergence between the clustercenter that is stored in the key object and the new center.
    - If it they are not equal, increment an update counter
- Run this whole thing until nothing was updated anymore.

Implementation 1

Pre-requisite

- Two files:
  - F1: houses the clusters with their centroids
  - F2: houses the vectors to be clustered

- The initial set of centres is stored in the input directory of HDFS
  - they form the ‘key’ field in the `<key,value>`

Map & Reduce routines

- Mapper:
  - Computes the distance between the given data set and cluster centre fed as a `<key,value>`
  - Keeps track of the cluster to which the given vector is closest
  - Assign the vector to the nearest cluster, once the computation of distances is complete
- Reducer:
  - Recalculates the centroid
  - Restructures the cluster to prevent creations of clusters with extreme sizes i.e. cluster having too less data vectors or a cluster having too many data vectors
  - Re-writes the new set of vectors and clusters to the disk
- Ready for the next iteration

Algorithm 1 Mapper design for K-Means Clustering

0: **procedure** KMEANMAPDESIGN
0: LOAD Cluster file
0: $fp = Mapclusterfile$
0: Create two list
0: $listnew = listold$
0: CALL read (Mapclusterfile)
0: newfp = MapCluster()
0: dv =0
0: Assign correct centeroid
0: read(dv)
0: calculate centerorid
0: dv = minCenter()
0: CALL KmeansReduce()
0: **end procedure**=0

Algorithm 2 Reducer design for K-Means Clustering

0: **procedure** KMEANREDUCEDESIGN
0: NEW ListofClusters
0: COMBINE resultant clusters from MAP CLASS.
0: **if** cluster size too high or too low **then**
RESIZE the cluster
0: $C_{Max} = findMaxSize(ListofClusters)$
0: $C_{min} = findMinSize(ListofClusters)$
0: **if** $C_{max} > \frac{1}{20} totalSize$ **then** Resize(cluster)
0: WRITE cluster FILE to output DIRECTORY.
0:

Algorithm 3 Implementing KMeans Function

```
0: procedure KMEANS FUNCTION
0:   if Initial Iteration then LOAD cluster file from DIREC-
   TORY
0:   elseREAD cluster file from previous iteration
0:     Create new JOB
0:     SET MAPPER to map class defined
0:     SET REDUCER to reduce class define
0:     paths for output directory
0:     SUBMIT JOB
0:
```

Implementation 2

http://codingwiththomas.blogspot.kr/2011/05/k-means-clustering-with-mapreduce.html

Implementation 3: Spark platform

Spark

Fast, Interactive, Language-Integrated Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das,
Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin,
Scott Shenker, Ion Stoica

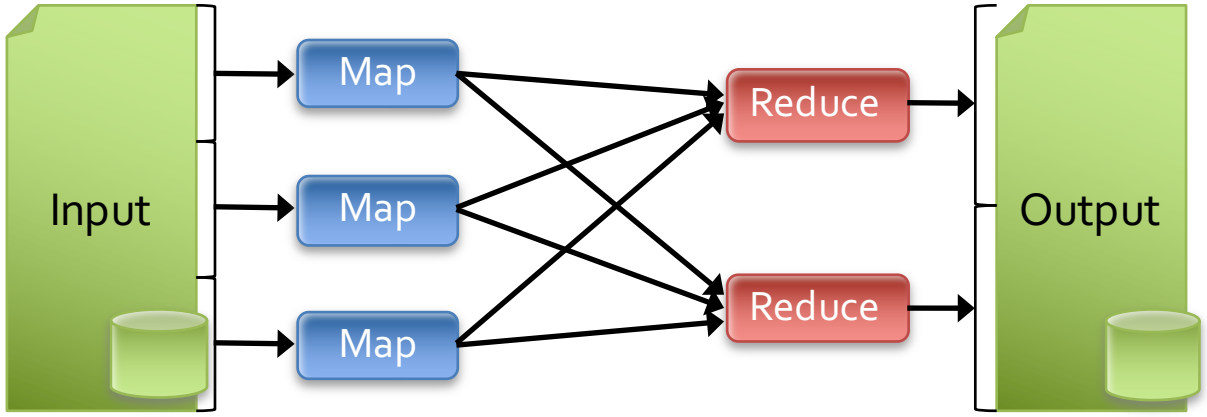
www.spark-project.org

Project Goals

- Extend the MapReduce model to better support two common classes of analytics apps:
  - **Iterative** algorithms (machine learning, graphs)
  - **Interactive** data mining
- Enhance programmability:
  - Integrate into Scala programming language
  - Allow interactive use from Scala interpreter

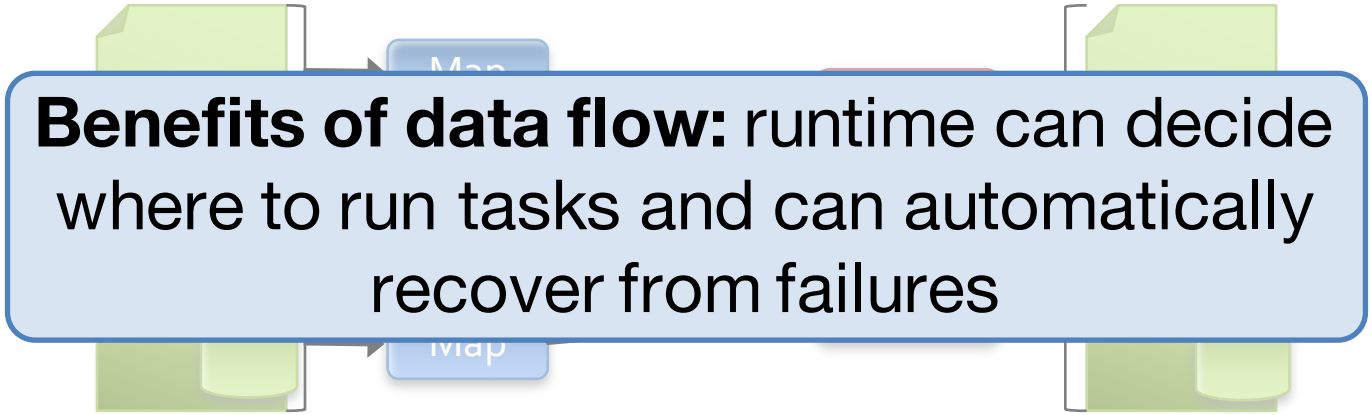
Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



Motivation

- Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



Motivation

- Acyclic data flow is inefficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (machine learning, graphs)
  - **Interactive** data mining tools (R, Excel, Python)
- With current frameworks, apps reload data from stable storage on each query

Solution: Resilient Distributed Datasets (RDDs)

- Allow apps to keep working sets in memory for efficient reuse
- Retain the attractive properties of MapReduce
  - Fault tolerance, data locality, scalability
- Support a wide range of applications

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Outline

Spark programming model

Implementation

User applications

Programming Model

Resilient distributed datasets (RDDs)

- Immutable, partitioned collections of objects
- Created through parallel *transformations* (map, filter, groupBy, join, …) on data in stable storage
- Can be *cached* for efficient reuse

Actions on RDDs

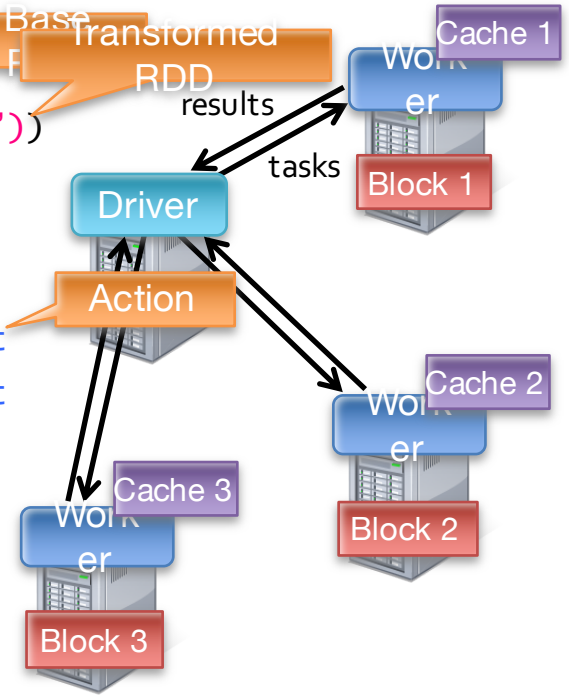
- Count, reduce, collect, save, …

Example: Log Mining

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
. . .
```



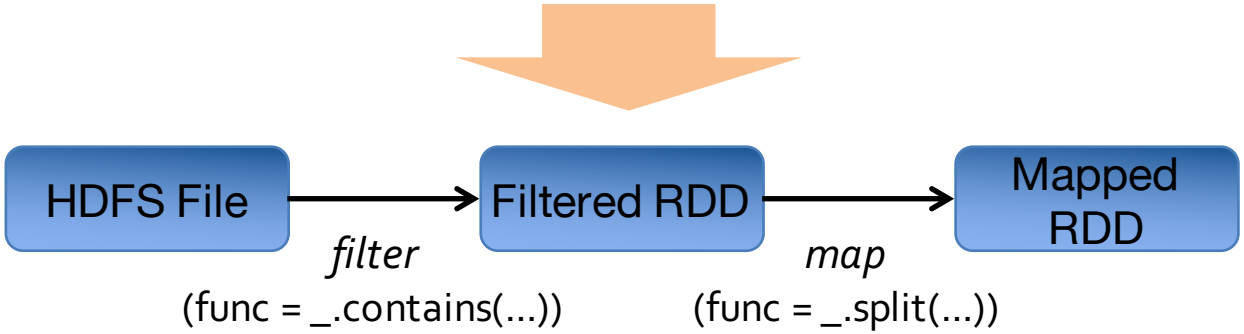
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

RDD Fault Tolerance

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

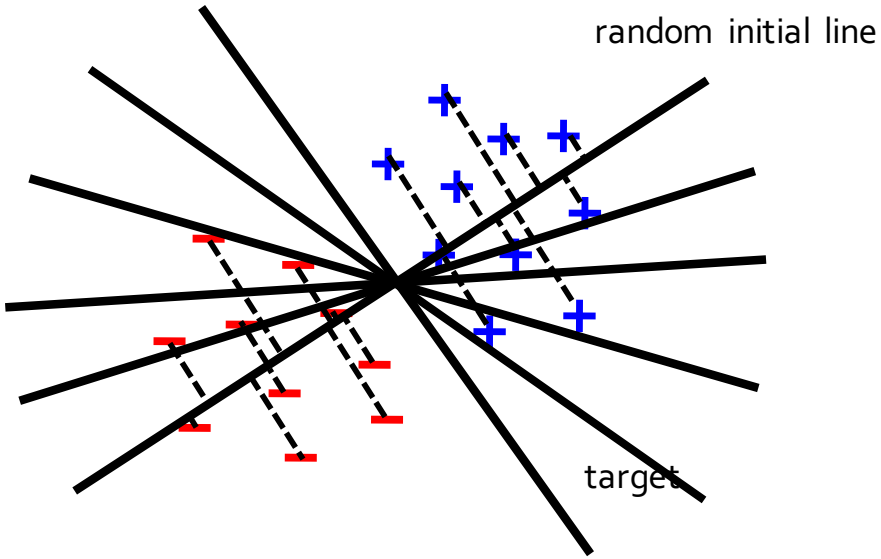
Ex:

```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



Example: Logistic Regression

Goal: find best line separating two sets of points



Example: Logistic Regression

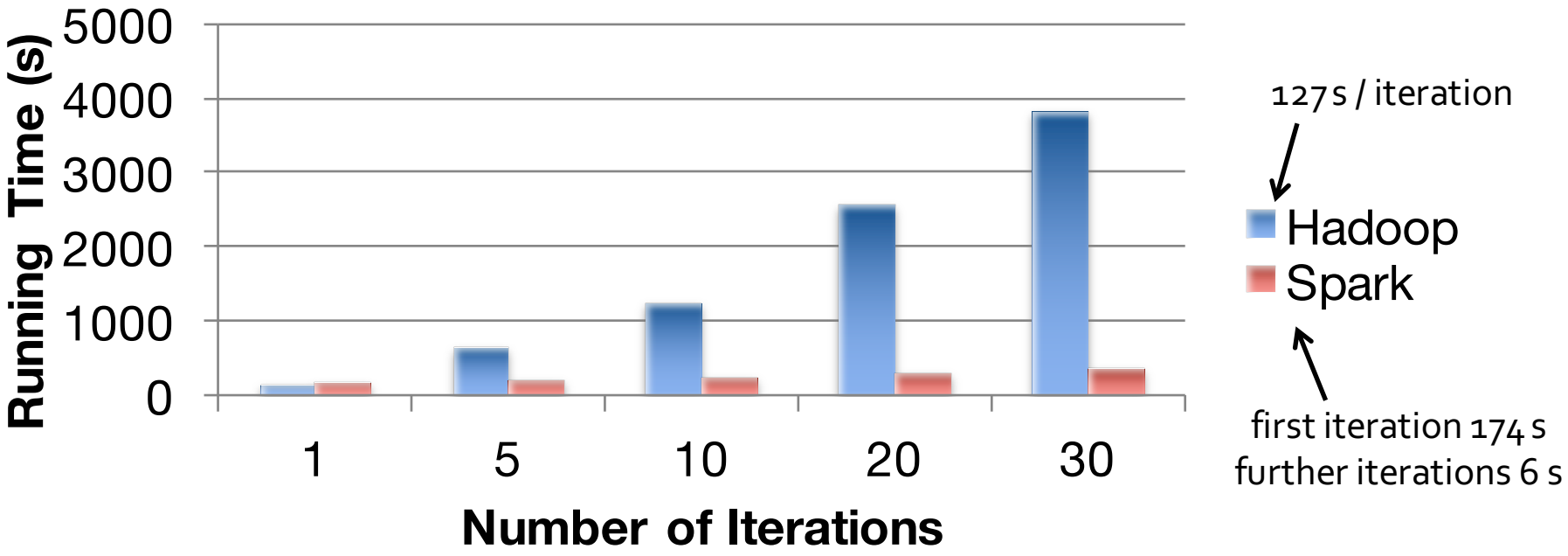

```
val data = spark.textFile(...).map(readPoint).cache()

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



This is for a 29 GB dataset on 20 EC2 m1.xlarge machines (4 cores each)

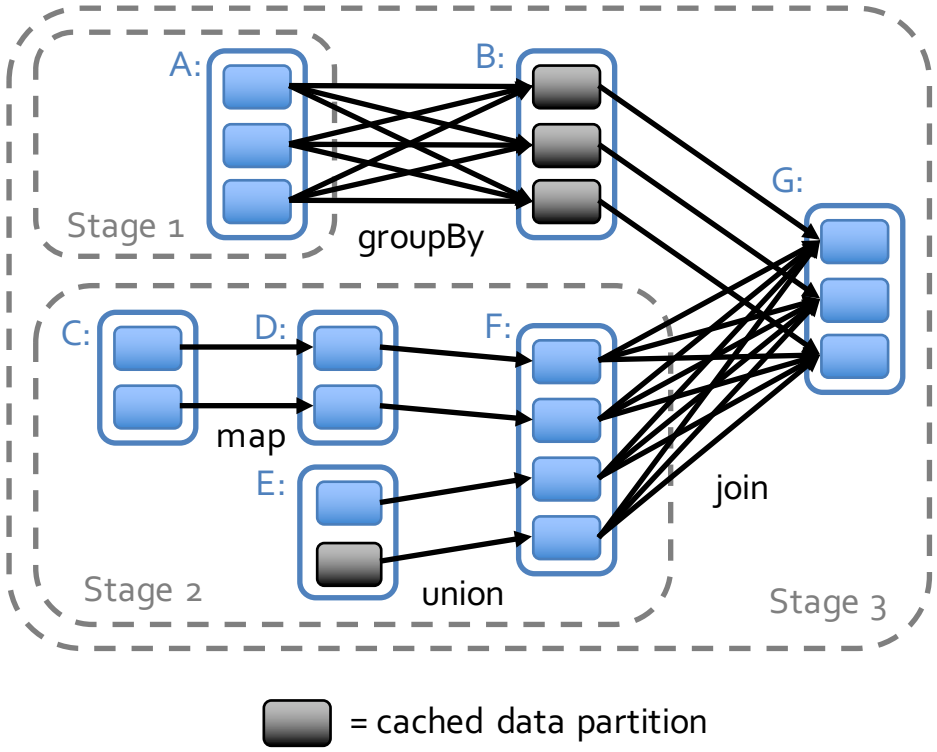
Spark Scheduler

Dryad-like DAGs

Pipelines functions within a stage

Cache-aware work reuse & locality

Partitioning-aware to avoid shuffles



Conclusion

- Spark provides a simple, efficient, and powerful programming model for a wide range of apps
- Download our open source release:
  - **www.spark-project.org**

matei@berkeley.edu

Related Work

DryadLINQ, FlumeJava

- Similar "distributed collection" API, but cannot reuse datasets efficiently *across* queries

- Relational databases
  - Lineage/provenance, logical logging, materialized views

GraphLab, Piccolo, BigTable, RAMCloud

- Fine-grained writes similar to distributed shared memory

- Iterative MapReduce (e.g. Twister, HaLoop)
  - Implicit data sharing for a fixed computation pattern

- Caching systems (e.g. Nectar)
  - Store data in files, no explicit control over what is cached

Let's dive on Spark for executing and analyzing K-Means

https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html