



MapReduce

& Pig & Spark

Ioanna Miliou

Giuseppe Attardi

Advanced Programming

Università di Pisa

Hadoop

- The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.
- Framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.
- It is designed to **scale up** from single servers to thousands of machines, each offering local computation and storage.
- It is designed to detect and **handle failures** at the application layer.

The core of Apache Hadoop consists of a storage part, known as **Hadoop Distributed File System (HDFS)**, and a processing part called **MapReduce**.

Hadoop

- The project includes these modules:
 - **Hadoop Common**: The common utilities that support the other Hadoop modules.
 - **Hadoop Distributed File System (HDFS)**: A distributed file system that provides high-throughput access to application data.
 - **Hadoop YARN**: A framework for job scheduling and cluster resource management.
 - **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets.

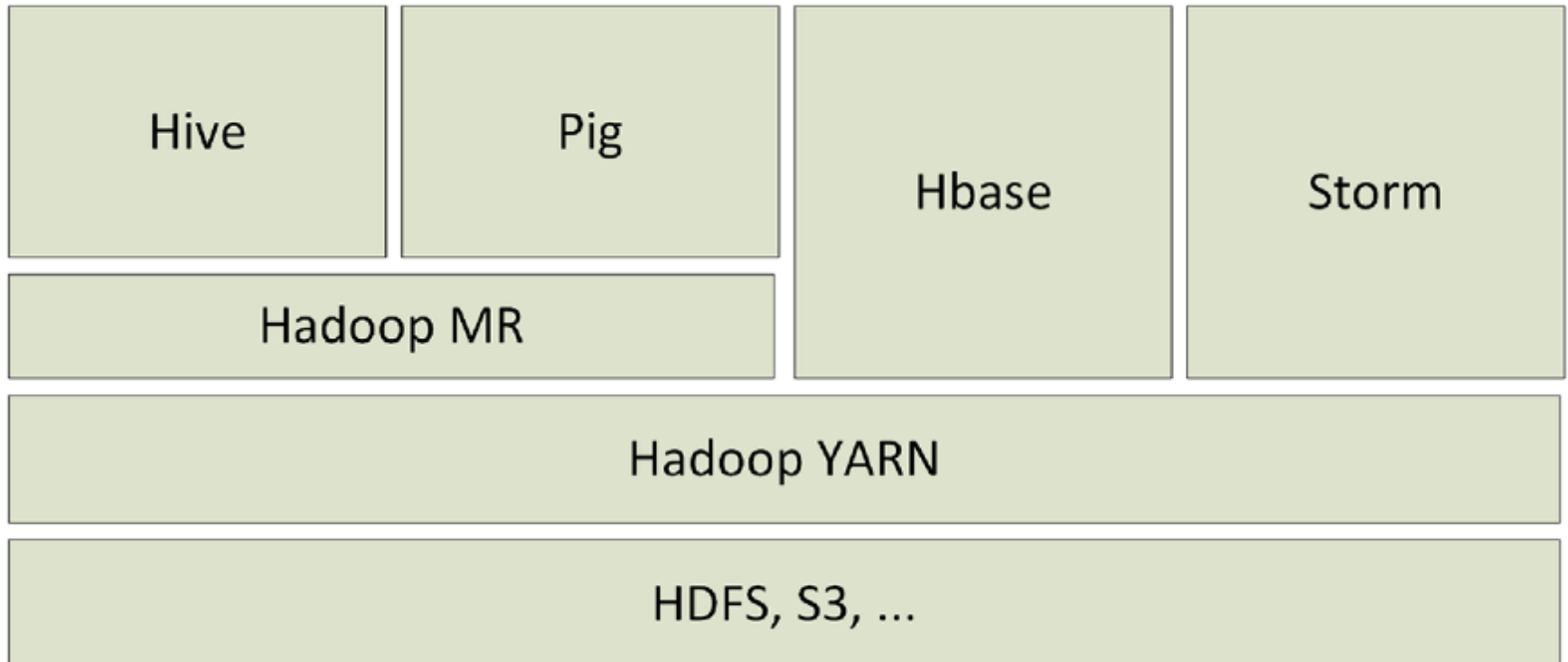
Hadoop

- Other Hadoop-related projects at Apache include:
 - **Ambari**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop.
 - **Avro**: A data serialization system.
 - **Cassandra**: A scalable multi-master database with no single points of failure.
 - **Chukwa**: A data collection system for managing large distributed systems.
 - **HBase**: A scalable, distributed database that supports structured data storage for large tables.
 - **Hive**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
 - **Mahout**: A Scalable machine learning and data mining library.
 - **Tez**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases.
 - **ZooKeeper**: A high-performance coordination service for distributed applications.

Hadoop

- **Pig** : A high-level data-flow language and execution framework for parallel computation.
- **Spark** : A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.

Hadoop Stack





What is MapReduce?





- MapReduce is the heart of Hadoop®
- Programming paradigm that allows for massive scalability across hundreds or thousands of servers in a Hadoop cluster.

Proposed by Dean and Ghemawat at Google

What is it?

- Processing engine of Hadoop
- Used for big data batch processing
- Parallel processing of huge data volumes
- Fault tolerant
- Scalable

Why use it?

- Your data in Terabyte / Petabyte range
- You have huge I/O
- Hadoop framework takes care of
 - Job and task management
 - Failures
 - Storage
 - Replication

You just write Map and Reduce jobs



Big Users

- Users

- Facebook
- Yahoo
- Amazon
- Ebay

- Providers

- Amazon
 - Cloudera
 - HortonWorks
 - MapR
- 

Map & Reduce

The term MapReduce actually refers to two separate and distinct tasks that Hadoop programs perform.

1. The **map** job, which takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs).

$$\text{map } (k1,v1) \rightarrow \text{list}(k2,v2)$$

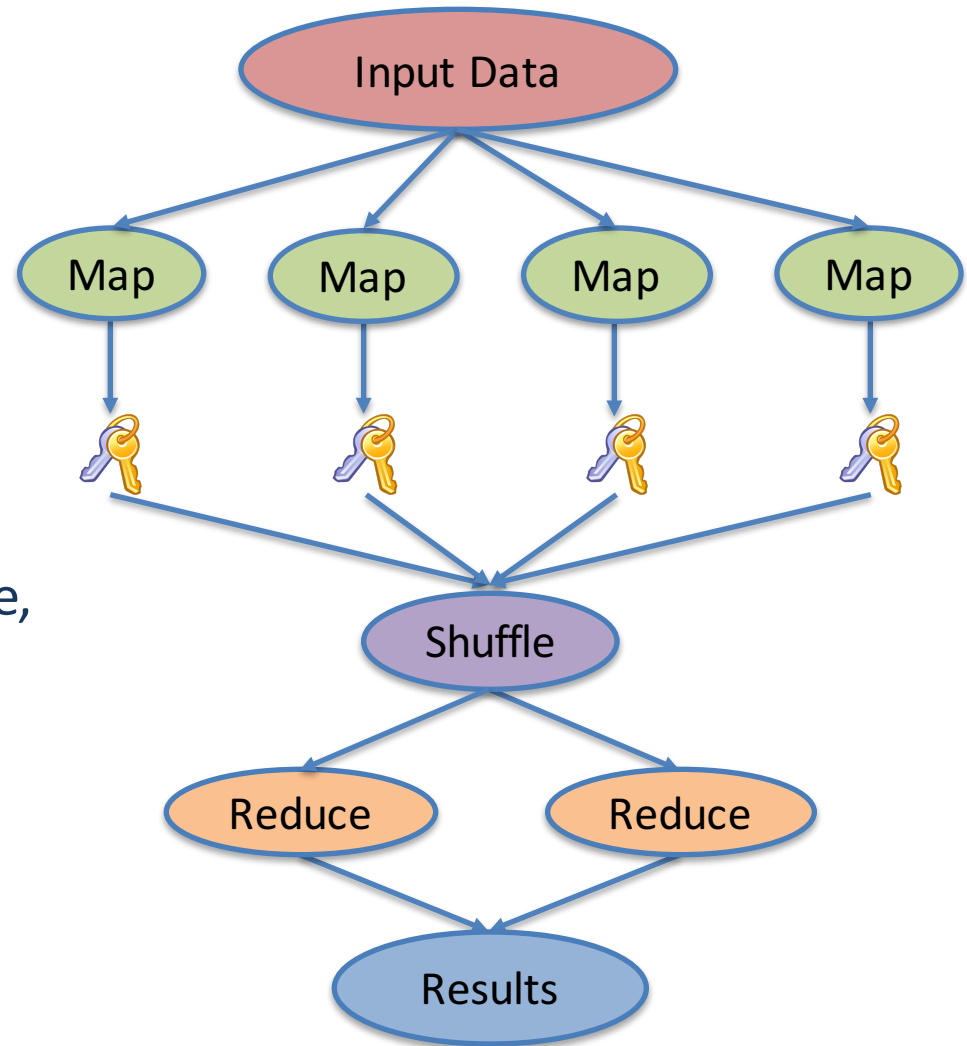
1. The **reduce** job takes the output from a map as input and combines those data tuples into a smaller set of tuples.

$$\text{reduce } (k2,\text{list}(v2)) \rightarrow \text{list}(v2)$$

As the sequence of the name MapReduce implies, the reduce job is always performed after the map job.

Typical Problem solved by MapReduce

- Read a lot of data
- *Map* : extract something you care about from each record
- Shuffle and Sort
- *Reduce* : aggregate, summarize, filter, or transform
- Write the results



Example : Word Count in Web Pages

A typical exercise for a new engineer in his or her first week

- Input is files with one document per record
- Specify a *map* function that takes a key/value pair
key = document URL
value = document contents
- Output of map function is (potentially many) key/value pairs.
In our case, output (word, "1") once per word in the document

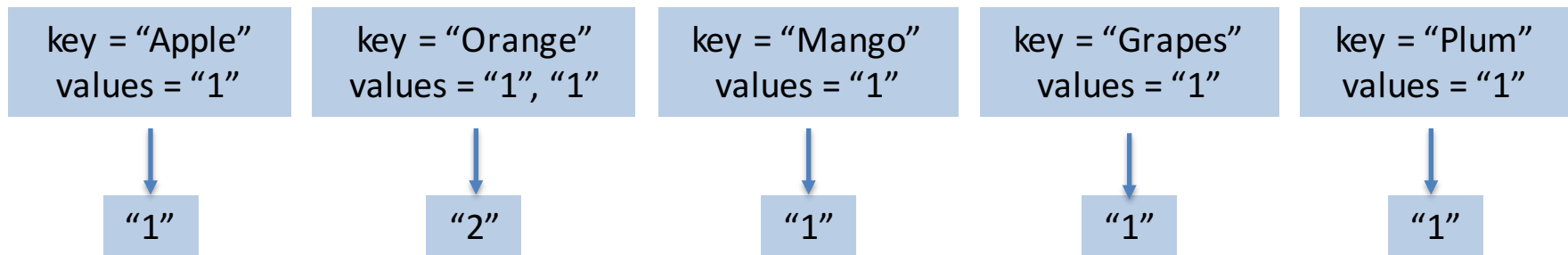
"document1", "Apple Orange Mango Orange Grapes Plum"

"Apple", "1"
"Orange", "1"
"Mango", "1"
...

Example continued :

Word Count in Web Pages

- MapReduce library gathers together all pairs with the same key (shuffle/sort)
- The reduce function combines the values for a key
In our case, compute the sum

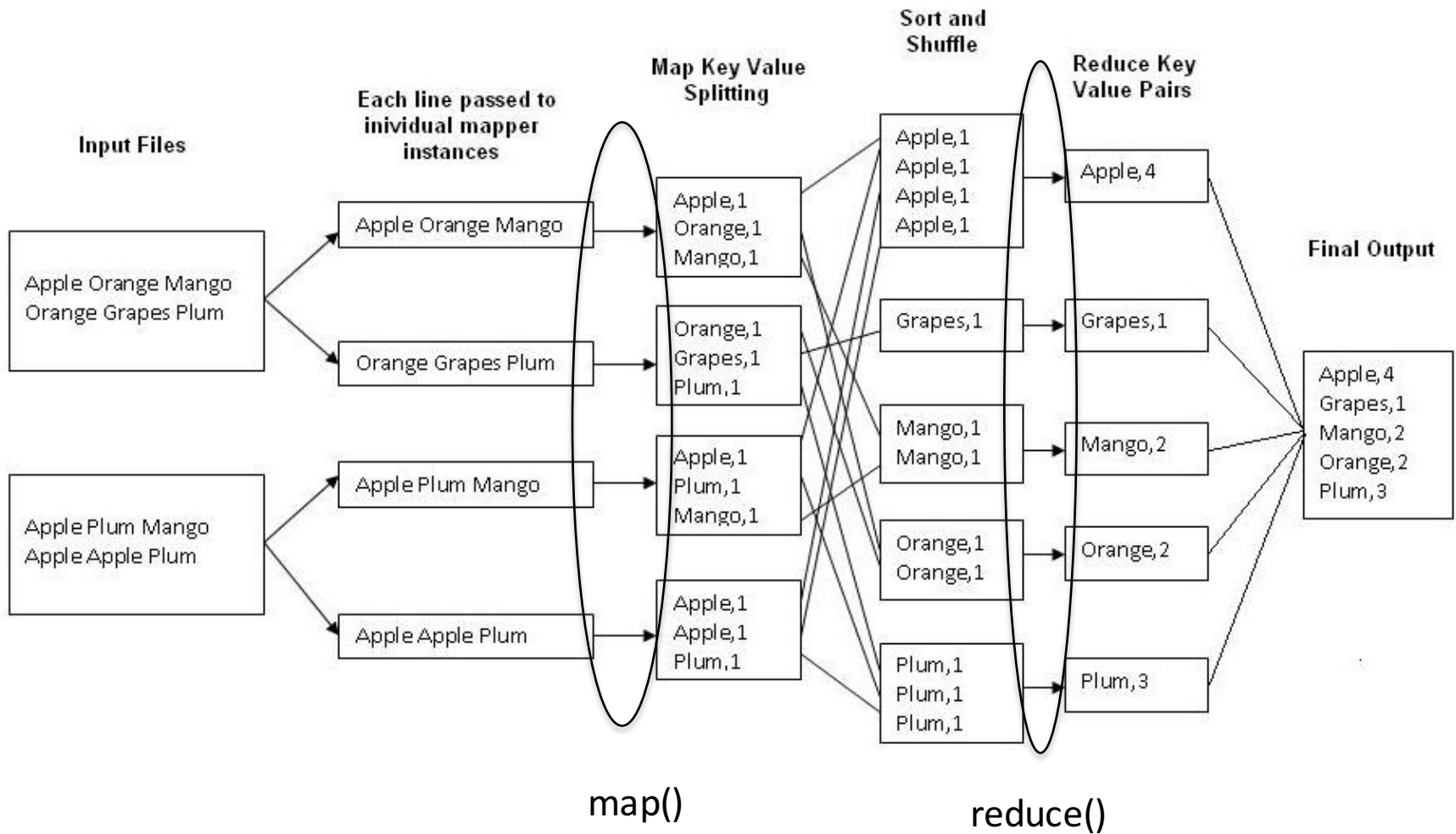


- Output of reduce paired with key and saved

"Apple", "1"
"Orange", "2"
"Mango", "1"
"Grapes", "1"
"Plum", "1"

Example Pseudo-code

```
map(String input_key, String input_value):  
  // input_key : document name  
  // input_value : document contents  
  for each word w in input_value:  
    EmitIntermediate(w, "1") ;  
reduce(String key, Iterator intermediate_values):  
  // key : a word, same for input and output  
  // intermediate_values : a list of counts  
  int result = 0 ;  
  for each v in intermediate_values:  
    result += ParseInt(v) ;  
  Emit(AsString(result)) ;
```

MapReduce wrappers

Wrappers have been developed in order to:

- provide a better control over the MapReduce code
- aid in the source code development

Some well-known example:

- Sawzall (Google)
- Pig (originally Yahoo, now Apache)
- Hive (Facebook)
- DryadLINQ (Microsoft)

Widely applicable at Google

- Implemented as a C++ library linked to user programs
- Can read and write many different data types

Example uses:


distributed grep
distributed sort
term-vector per host
document clustering
machine learning
...

web access log stats
web link-graph reversal
inverted index construction
statistical machine translation
...



Example:

Generating Language Model Statistics

- Used in the statistical machine translation system
 - need to count # of times every 5-word sequence occurs in large corpus of documents (and keep all those where count ≥ 4)
 - Easy with MapReduce:
 - **map** : extract 5-word sequences => count from document
 - **reduce** : combine counts, and keep if count large enough
- 

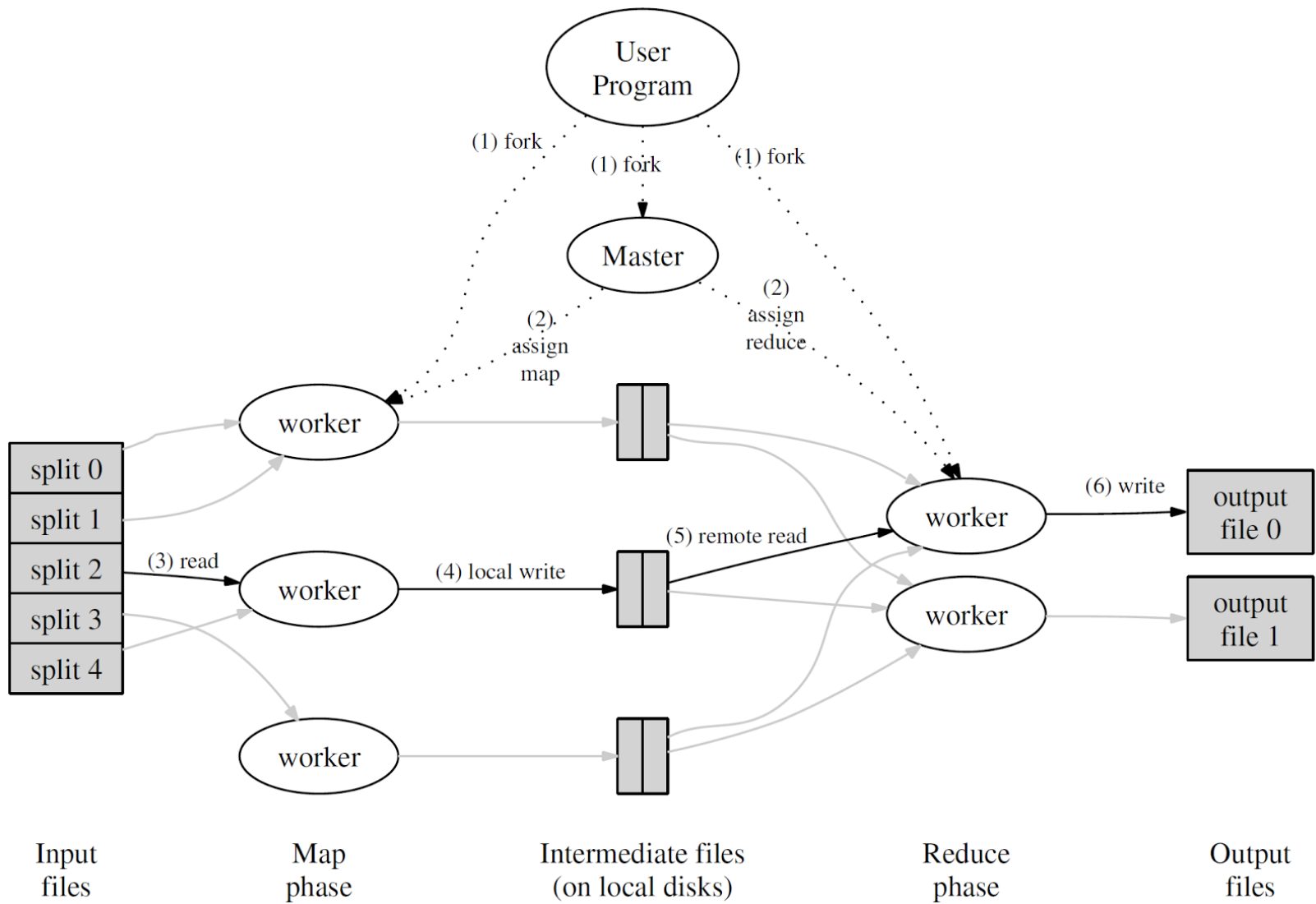
Example :

Joining with Other Data

- Example : generate per-doc summary, but include per-host information (e.g. # of pages on host, important terms on host)
 - per-host information might be in per-process data structure, or might involve RPC to a set of machines containing data for all sites
- Easy with MapReduce:
 - **map** : extract host name from URL, lookup per-host info, combine with per-doc data and emit
 - **reduce** : identity function (just emit key/value directly)

MapReduce: Scheduling

- **One master, many workers**
 - Input data split into M map tasks (typically 64 MB in size)
 - Reduce phase partitioned into R reduce tasks
 - Tasks are assigned to workers dynamically
 - Often: M=200000, R=4000, workers=2000
- **Master assigns each map task to a free worker**
 - Considers locality of data to worker when assigning task
 - Worker reads task input (often from local disk)
 - Worker produces R local files containing intermediate k/v pairs
- **Master assigns each reduce task to a free worker**
 - Worker reads intermediate k/v pairs from map workers
 - Worker sorts & applies user's Reduce op to produce the output

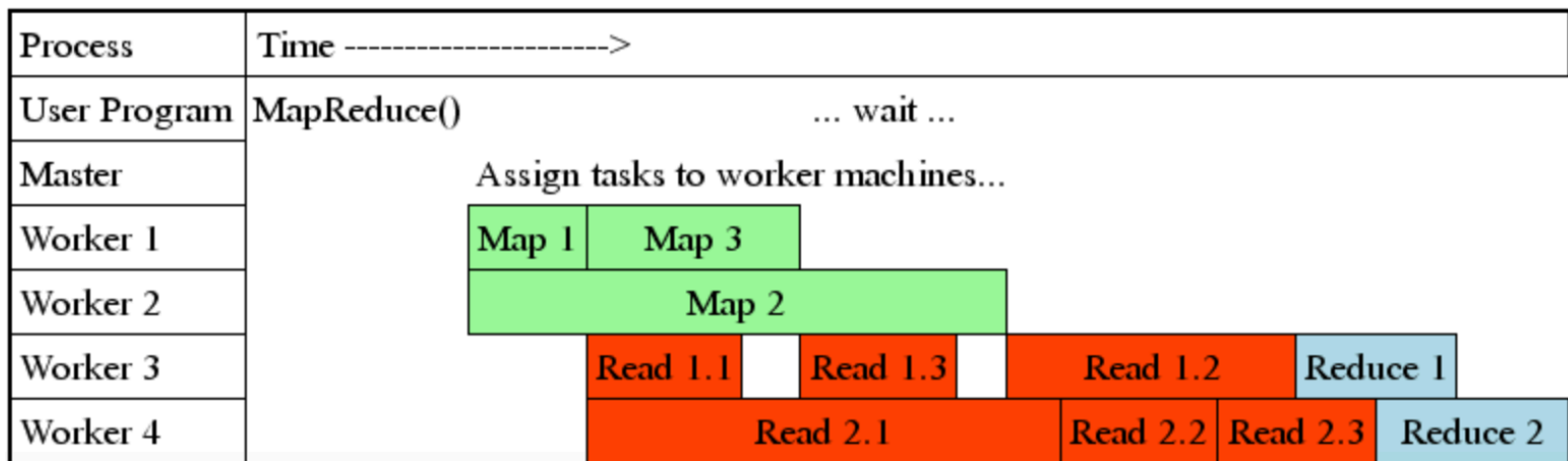


Task Granularity and Pipelining

Fine granularity tasks: many more map tasks than machines

- Minimizes time for fault recovery
- Can pipeline shuffling with map execution
- Better dynamic load balancing

Often use 200,000 map/5000 reduce tasks w/ 2000 machines



Fault tolerance: Handled via re-execution

- On worker failure:
 - Detect failure via periodic heartbeats
 - Re-execute completed and in-progress *map* tasks
 - Re-execute in progress *reduce* tasks
 - Task completion committed through master
- Master failure:
 - State is checkpointed : new master recovers & continues

Robust: Once Google lost 1600 of 1800 machines, but finished fine

Refinement: Backup tasks

- Slow workers significantly lengthen completion time
 - Other jobs consuming resources on machine
 - Bad disks with soft errors transfer data very slowly
 - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup copies of tasks
 - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time

Refinement: Locality Optimization

Master scheduling policy:

- Asks for locations of replicas of input file blocks
- Map tasks typically split into 64MB
- Map tasks scheduled so input block replica are on same machine or same rack

Effect: Thousands of machines read input at local disk speed

- Without this, rack switches limit read rate

Refinement: Skipping Bad Records

Map/Reduce functions sometimes fail for particular inputs

- Best solution is to debug & fix, but not always possible

On seg fault:

- Send UDP packet to master from signal handler
- Include sequence number of record being processed


If master sees K failures for same record (typically K set to 2 or 3) :

- Next worker is told to skip the record

Effect: Can work around bugs in third-party libraries



Other Refinements

- Optional secondary keys for ordering
 - Compression of intermediate data
 - Combiner: useful for saving network bandwidth
 - Local execution for debugging/testing
 - User-defined counters
- 

“Play around”

- Amazon Elastic MapReduce (Amazon EMR)
- Hortonworks Sandbox
- MapR Sandbox for Hadoop
- Qubole
- Microsoft Azure HDInsight
- Cloudera



MapReduce examples in Java



Serializable vs Writable

- Serializable stores the class name and the object representation to the stream; other instances of the class are referred to by an handle to the class name: this approach is not usable with random access
- For the same reason, the sorting needed for the shuffle and sort phase can not be used with Serializable
- The deserialization process creates an new instance of the object, while Hadoop needs to reuse object to minimize computation
- Hadoop introduces the two interfaces **Writable** and **WritableComparable** that solve these problem

Writable wrappers

Java primitive	Writable implementation
boolean	BooleanWritable
byte	ByteWritable
short	ShortWritable
int	IntWritable VIntWritable
float	FloatWritable
long	LongWritable VLongWritable
double	DoubleWritable

Java class	Writable implementation
String	Text
byte[]	BytesWritable
Object	ObjectWritable
<i>null</i>	NullWritable

Java collection	Writable implementation
<i>array</i>	ArrayWritable ArrayPrimitiveWritable TwoDArrayWritable
Map	MapWritable
SortedMap	SortedMapWritable
<i>enum</i>	EnumSetWritable

Implementing Writable: the SumCount class

```
public class SumCount implements WritableComparable<SumCount> {

    DoubleWritable sum;
    IntWritable count;

    public SumCount() {
        set(new DoubleWritable(0), new IntWritable(0));
    }

    public SumCount(Double sum, Integer count) {
        set(new DoubleWritable(sum), new IntWritable(count));
    }

    @Override
    public void write(DataOutput dataOutput) throws IOException{
        sum.write(dataOutput);
        count.write(dataOutput);
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException{
        sum.readFields(dataInput);
        count.readFields(dataInput);
    }
    // getters, setters and Comparable override methods are omitted
}
```

Glossary

Term	Meaning
Job	The whole process to execute: the input data, the mapper and reducers execution and the output data
Task	Every job is divided among the several mappers and reducers; a task is the job portion that goes to every single mapper and reducer
Split	The input file is split into several splits (the suggested size is the HDFS block size, 64Mb)
Record	The split is read from mapper by default a line at the time: each line is a record. Using a class extending <code>FileInputFormat</code> , the record can be composed by more than one line
Partition	The set of all the key-value pairs that will be sent to a single reducer. The default partitioner uses an hash function on the key to determine to which reducer send the data

WordCount

- <http://www.gutenberg.org/cache/epub/201/pg201.txt>
- Input Data :
The text of the book “Flatland” by Edwin Abbott

WordCount mapper

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {  
  
    private final static IntWritable one = new IntWritable(1);  
    private Text word = new Text();  
  
    @Override  
    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().trim());  
            context.write(word, one);  
        }  
    }  
}
```

WordCount reducer

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

Word Count results

a	936
ab	6
abbot	3
abbott	2
abbreviated	1
abide	1
ability	1
able	9
ablest	2
abolished	1
abolition	1
about	40
above	22
abroad	1
abrogation	1
abrupt	1
abruptly	1
absence	4
absent	1
absolute	2
...	

TopN : We want to find the top-n used words of a text file

- <http://www.gutenberg.org/cache/epub/201/pg201.txt>
- Input Data :
The text of the book “Flatland” by Edwin Abbott

TopN mapper

```
public static class TopNMapper extends Mapper<Object, Text, Text, IntWritable> {  
  
    private final static IntWritable one = new IntWritable(1);  
    private Text word = new Text();  
    private String tokens = "[_!$#<>\\^=\\[\\]\\\\*^\\\\\\\\,;,.\\-:()?!\\\"'"]";  
  
    @Override  
    public void map(Object key, Text value, Context context)  
        throws IOException, InterruptedException {  
  
        String cleanLine = value.toString().toLowerCase().replaceAll(tokens, " ");  
        StringTokenizer itr = new StringTokenizer(cleanLine);  
        while (itr.hasMoreTokens()) {  
            word.set(itr.nextToken().trim());  
            context.write(word, one);  
        }  
    }  
}
```

TopN reducer

```
public static class TopNReducer extends Reducer<Text, IntWritable, Text, IntWritable> {  
  
    private Map<Text, IntWritable> countMap = new HashMap<>();  
  
    @Override  
    public void reduce(Text key, Iterable<IntWritable> values, Context context)  
        throws IOException, InterruptedException {  
  
        int sum = 0;  
        for (IntWritable val : values) {  
            sum += val.get();  
        }  
        countMap.put(new Text(key), new IntWritable(sum));  
    }  
  
    @Override  
    public void cleanup(Context context) throws IOException, InterruptedException {  
  
        Map<Text, IntWritable> sortedMap = sortByValues(countMap);  
  
        int counter = 0;  
        for (Text key : sortedMap.keySet()) {  
            if (counter ++ == 20) {  
                break;  
            }  
            context.write(key, sortedMap.get(key));  
        }  
    }  
}
```

TopN results

the	2286
of	1634
and	1098
to	1088
a	936
i	735
in	713
that	499
is	429
you	419
my	334
it	330
as	322
by	317
not	317
or	299
but	279
with	273
for	267
be	252
...	

MEAN : We want to find the mean max temperature for every month

- <http://archivio-meteo.distile.it/tabelle-dati-archivio-meteo/>
- Input Data :
Temperature in Milan (DD/MM/YYYY, MIN, MAX)
02012015, -2, 7
03012015, -1, 8
04012015, 1, 16
...
29012015, 0, 5
30012015, 0, 9
31012015, -3, 6

Mean mapper

```
private Map<Text, List<Double>> maxMap = new HashMap<>();

@Override
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {

    String[] values = value.toString().split(",");
    if (values.length != 3) return;

    String date = values[DATE];
    Text month = new Text(date.substring(2));
    Double max = Double.parseDouble(values[MAX]);

    if (!maxMap.containsKey(month)) {
        maxMap.put(month, new ArrayList<Double>());
    }
    maxMap.get(month).add(max);
}

@Override
protected void cleanup(Context context) throws InterruptedException {

    for (Text month : maxMap.keySet()) {
        List<Double> temperatures = maxMap.get(month);
        Double sum = 0d;
        for (Double max : temperatures)
            sum += max;
        context.write(month, new SumCount(sum, temperatures.size()));
    }
}
```

Mean reducer

```
private Map<Text, SumCount> sumCountMap = new HashMap<>();

@Override
public void reduce(Text key, Iterable<SumCount> values, Context context)
    throws IOException, InterruptedException {

    SumCount totalSumCount = new SumCount();
    for (SumCount sumCount : values) {
        totalSumCount.addSumCount(sumCount);
    }
    sumCountMap.put(new Text(key), totalSumCount);
}

@Override
protected void cleanup(Context context) throws InterruptedException {

    for (Text month : sumCountMap.keySet()) {
        double sum = sumCountMap.get(month).getSum().get();
        int count = sumCountMap.get(month).getCount().get();
        context.write(month, new DoubleWritable(sum/count));
    }
}
```

Mean results

022012	7.230769230769231
022013	7.2
022010	7.851851851851852
022011	9.785714285714286
032013	10.741935483870968
032010	13.133333333333333
032012	18.548387096774192
032011	13.741935483870968
022003	9.278571428571428
022004	10.41034482758621
022005	9.146428571428572
022006	8.903571428571428
022000	12.344444444444441
022001	12.164285714285715
022002	11.839285714285717
...	

TODO : k-means clustering algorithm

- We want to aggregate 2D points in clusters using k-means algorithm

- Input data :

A random set of points

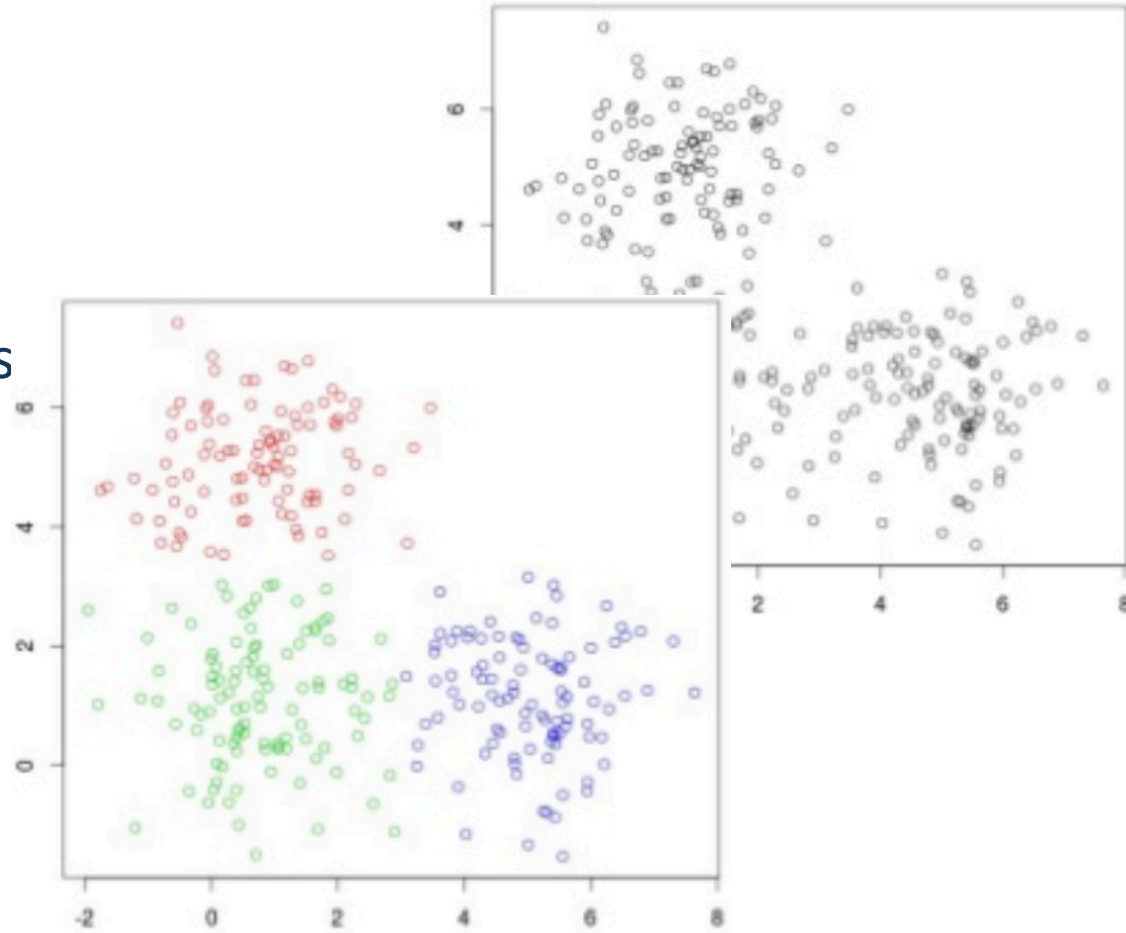
2.2705 0.9178

1.8600 2.1002

2.0915 1.3679

-0.1612 0.8481

...



k-means algorithm

Input: data points D , number of cluster k

1. initialize k centroids randomly
2. associate each data point in D with the nearest centroid. This will divide the data points into k clusters.
3. recalculate the position of centroids.

Repeat steps 2 and 3 until there are no more changes in the membership of the data points.

Output: data points with cluster memberships



MapReduce examples in Python



Word Count using mrjob

```
def mapper(self, key, line):  
    for word in line.split():  
        yield word, 1
```


```
def reducer(self, key, occurrences):  
    yield word, sum(occurrences)
```



```
"a", 936  
"ab", 6  
"abbot", 3  
"abbott", 2  
"abbreviated", 1  
...
```



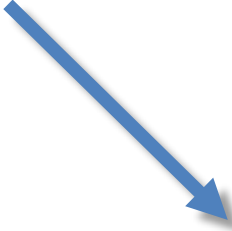
Product Recommendations

- Goal : For each product a client buys, generate a 'people who bought this also bought this' recommendation
 - Input Data : product_id_1, product_id_2
- 

Coincident Purchase Frequency

```
def mapper(self, key, line):  
    purchases = set(line.split(','))  
    for p1, p2 in permutations(purchases, 2):  
        yield (p1, p2), 1
```

```
def reducer(self, pair, occurrences):  
    p1, p2 = pair  
    yield p1, (p2, sum(occurrences))
```



```
"8" ["5", 11]  
"8" ["6", 19]  
"8" ["7", 14]  
"8" ["9", 11]  
"9" ["1", 20]  
"9" ["10", 22]  
"9" ["11", 21]  
"9" ["12", 13]
```

Top Recommendations

```
def reducer(self, purchase_pair, occurrences):
```

```
    p1, p2 = purchase_pair
```

```
    yield p1, (sum(occurrences), p2)
```

```
def reducer_find_best_recos(self, p1, p2_occurrences):
```

```
    top_products = sorted(p2_occurrences, reverse=True) [:5]
```

```
    top_products = [p2 for occurrences, p2 in top_products]
```

```
    yield p1, top_products
```

```
def steps(self) :
```

```
    return [self.mr(mapper=self.mapper, reducer=self.reducer),
```

```
            self.mr(reducer=self.reducer_find_best_recos)]
```



```
"7"  ["15", "18", "17", "16", "3"]
```

```
"8"  ["14", "15", "20", "6", "3"]
```

```
"9"  ["15", "17", "19", "6", "3"]
```

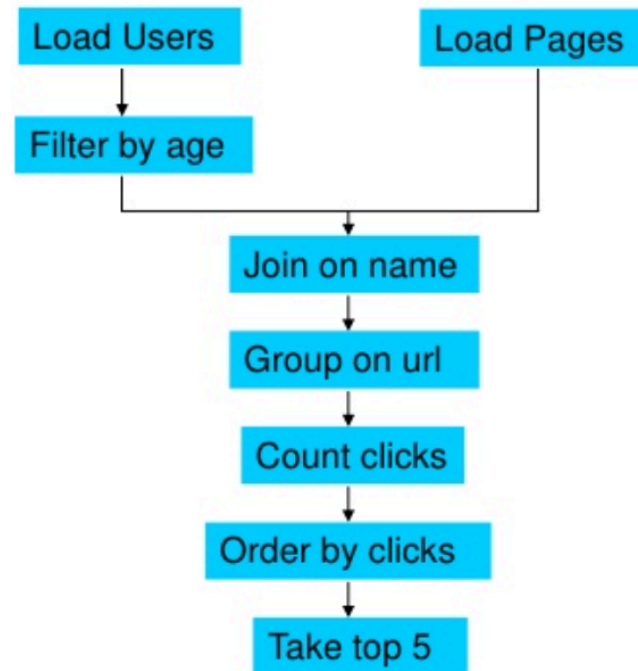
But...

Suppose you have :

- **user data** in one file,
- **website data** in another,

and you need to find

- the **top 5** most visited pages by users aged 18-25.



In Pig Latin

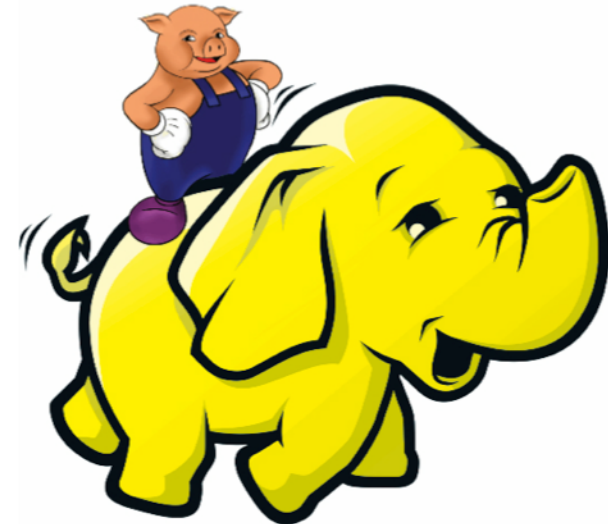
```
Users = load 'users' as (name, age);
Fltrd = filter Users by
    age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
COUNT(Jnd) as clicks;
Srted = order Smmd by clicks desc;
Top5 = limit Srted 5;
store Top5 into 'top5sites';
```

What is Apache Pig?



Idea: a MapReduce program essentially performs a group-by-aggregation in parallel over a cluster of machines.

- **Pig** is a high-level platform for creating MapReduce programs used with Hadoop.
- The language for this platform is called **Pig Latin**. It combines high-level declarative querying in the spirit of SQL, and low-level, procedural programming à la MapReduce.



Developed at Yahoo

Pig

- **Apache Pig** is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. The salient property of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets.
- At the present time, Pig's infrastructure layer consists of a compiler that produces sequences of Map-Reduce programs, for which large-scale parallel implementations already exist (e.g., the Hadoop subproject).

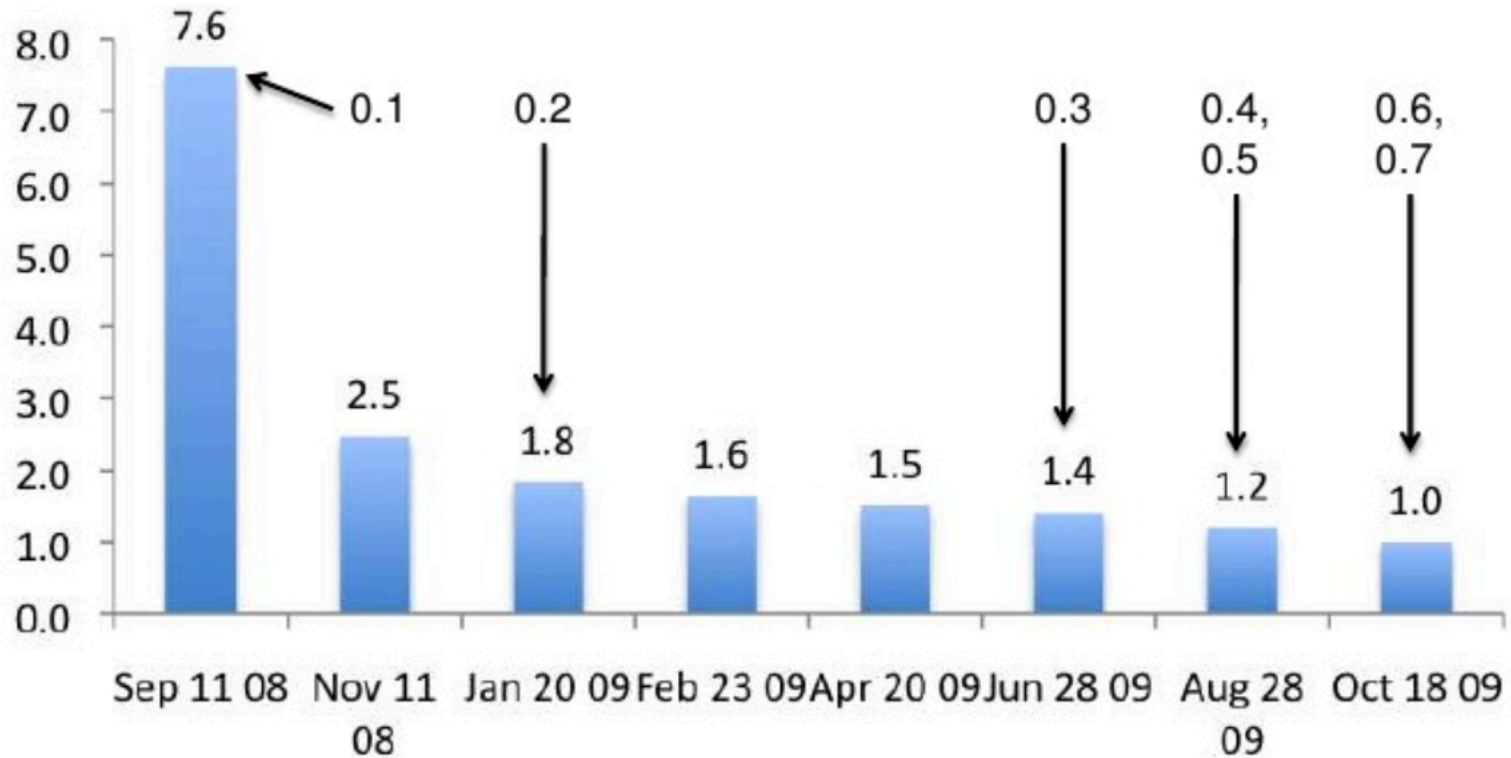
Pig Latin

Pig Latin has the following key properties:

- **Ease of programming.** It is trivial to achieve parallel execution of simple, "embarrassingly parallel" data analysis tasks. Complex tasks comprised of multiple interrelated data transformations are explicitly encoded as data flow sequences, making them easy to write, understand, and maintain.
- **Optimization opportunities.** The way in which tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.
- **Extensibility.** Users can create their own functions to do special-purpose processing.

Performance

Pig Performance vs Map-Reduce



Pig Highlights

- User defined functions (UDFs) can be written for column transformation (TOUPPER), or aggregation (SUM)
- UDFs can be written to take advantage of the combiner
- Four join implementations built in : hash, fragment-replicate, merge, skewed
- Multi-query : Pig will combine certain types of operations together in a single pipeline to reduce the number of times data is scanned
- Order by provides total ordering across reducers in a balanced way
- Writing load and store functions is easy once an InputFormat and OutputFormat exist
- Piggybank, a collection of users contributed UDFs

Who uses Pig for what?

- 70% of production jobs at *Yahoo* (10ks per day)
- Also used by *Twitter, LinkedIn, Ebay, AOL, ...*
- Used to
 - Process web logs
 - Build user behavior models
 - Process images
 - Build maps of the web
 - Do research on raw data sets

Components

Job executes on cluster

Pig resides on user machine



User machine




No need to install anything extra on your Hadoop cluster.

So, why Pig?

- Faster development
 - Fewer lines of code
 - Don't re-invent the wheel
- Flexible
 - Metadata is optional
 - Extensible
 - Procedural programming



But...

- Do you need your program to run faster?
 - Does your analytic job runs for hours?
- 

Limitations of MapReduce

One of the major drawbacks of MapReduce is its inefficiency in running **iterative algorithms**.

MapReduce is not designed for iterative processes: after each iteration, the results have to be written to the disk to pass them onto the next iteration.



degradation of performance



Limitations of Pig

Pig uses batch oriented frameworks, which means your analytics jobs will run for many minutes or hours.

Spark is faster!



What is Apache Spark?

- A fast and general compute engine for large-scale data processing.
- The major feature: the ability to perform **in-memory** computation (the data can be cached in memory).
- Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.




Developed at the University of California at Berkeley

Spark

- It provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.
- For certain tasks, it is tested to be up to 100x faster (data in memory) or 10x (data in disk) faster than Hadoop MapReduce
- It can run on Hadoop YARN manager and can read data from HDFS.



Spark

- Designed to be used with a **range of programming languages** and on a **variety of architectures**.
 - Increasingly popular with a wide range of developers, thanks to **speed, simplicity**, and broad **support** for existing development environments and storage systems.
 - Relatively **accessible** to those learning to work with it for the first time.
 - One of Apache's largest and most vibrant, with over 500 contributors from more than 200 organizations responsible for code in the software release.
- 

Why?

- Spark is basically developed to overcome MapReduce's shortcoming that it is not optimized for **iterative algorithms** and **interactive data analysis** which performs cyclic operations on same set of data.
- Spark overcomes this problem by providing a new storage primitive called **Resilient Distributed Datasets (RDDs)**.




Resilient Distributed Datasets (RDDs)

The Resilient Distributed Dataset is a concept at the heart of Spark. It is designed to support in-memory data storage, distributed across a cluster in a manner that is demonstrably both fault-tolerant and efficient.

- **Fault-tolerance** is achieved, in part, by tracking the lineage of transformations applied to coarse-grained sets of data.
- **Efficiency** is achieved through parallelization of processing across multiple nodes in the cluster, and minimization of data replication between those nodes.

Once data is loaded into an RDD, two basic types of operation can be carried out:

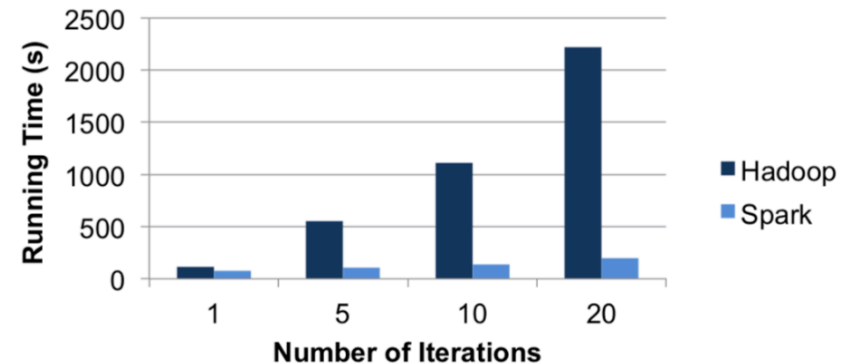
- **Transformations**, which create a new RDD by changing the original through processes such as mapping, filtering, and more;
 - **Actions**, such as counts, which measure but do not change the original data.
- 

Word Count in Spark

```
val counts = file.flatMap(line => line.split(" "))  
                  .map(word => (word, 1))  
                  .reduceByKey((a, b) => a + b)
```

Another example : logistic regression

A common machine learning algorithm for classifying objects such as, say, spam vs. non-spam emails.



```
val points = spark.textFile(...).map(parsePoint).persist()
var w = Vector.random(D) // Current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  }.reduce((a, b) => a + b)
  w -= gradient
}
println("Final separating plane: " + w)
```

Pig vs Spark

- **Pig**

- This is the best data loading tool available inside hadoop.
- It uses a scripting language called Pig Latin, which is more workflow driven.
- Don't need to be an expert Java programmer but need a few coding skills.
- Is also an abstraction layer on top of map-reduce.
- Simple to write and control.

- **Spark**

- Pretty much the successor to map-reduce in Hadoop, with an emphasis on in-memory computing.
- You'll need to be a pretty good Java programmer to use this.
- Much lower level.

How to choose a platform?


- The decision to choose a particular platform for a certain application usually depends on the following important factors:
 - data size
 - speed or throughput optimization
 - model development (Training/Applying a model)



Example: k-means clustering algorithm

The k-means algorithm is used for providing more insight into the analytics algorithms on different platforms.

Characteristics:

- popular and widely used
 - iterative nature
 - compute-intensive task (calculating the centroids)
 - aggregation of the local results to obtain a global solution
- 

k-means algorithm

Input: data points D , number of cluster k

1. initialize k centroids randomly
2. associate each data point in D with the nearest centroid. This will divide the data points into k clusters.
3. recalculate the position of centroids.

Repeat steps 2 and 3 until there are no more changes in the membership of the data points.

Output: data points with cluster memberships

k-means on MapReduce

Input: data points D , number of cluster k and centroids

1. for each data point $d \in D$ do
2. assign d to the closest centroid

Map

Output: centroids with associated data points

Input: centroids with associated data points

1. compute the new centroids by calculating the average of data points in cluster
2. write the global centroids to the disk

Reduce

Output: new centroids

k-means on PigLatin

```
REGISTER udf.jar
DEFINE find_centroid FindCentroid('$centroids');
points = LOAD 'points.txt' as (id:int, pos:double);
centroided = FOREACH points GENERATE pos, find_centroid(pos)
as centroid;
grouped = GROUP centroided BY centroid;
result = FOREACH grouped GENERATE group,
AVG(centroided.pos);
STORE result INTO 'output';
```

k-means on Spark

Similar to MapReduce-based implementation

- Instead of writing the global centroids to the disk, they are written to memory which speeds up the processing and reduces the disk I/O overhead.
- The data will be loaded into the system memory in order to provide faster access.

References

- Dean, J. and Ghemawat, S. ***MapReduce: Simplified data processing on large clusters***. In Proceedings of Operating Systems Design and Implementation (OSDI). San Francisco, CA. 137-150. 2004
- Hadoop: Open source implementation of MapReduce. <http://lucene.apache.org/hadoop/>
- C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins. ***Pig Latin: A Not-so-foreign Language for Data Processing***. In Proceedings SIGMOD '08. 2008
- D. Singh and C. K. Reddy. ***A survey on platforms for big data analytics***, Journal of Big Data. 2014
- M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. ***Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing***. USENIX NSDI. 2012