Peta-Bytes Data Processing

Google processes 20 PB a day (2008).

eBay has 9 PB of user data + >50 TB/day (11/2011).

Facebook reports 36 PB of user data + 80-90 TB/day (6/2010).

Amazon Web Services reports 53449 objects, peak 290k request/second (7/2011).

JPMorganChase indicates 150 PB on 50k+ servers running 15k apps.


CERN: (LHC) ~15 PB a year (at full capacity).

LSST: 6-10 PB a year (~2010).

640K ought to be enough for anybody.
What is MapReduce?

- **MapReduce**: programming model and associated implementation for batch processing of massive data sets
  - MapReduce could be implemented on different architectures, but Google proposed it for large clusters of commodity PCs

- Functional programming meets distributed computing:
  - A clean abstraction for programmers by factoring out many reliability concerns from application logic
  - Automatic parallelization & distribution
  - Fault-tolerance
  - Status and monitoring tools

- MapReduce implementations such as Hadoop differ in details, but main principles are the same

---

Overview

- Clever abstraction that is a good fit for many real-world problems
- Programmer focuses on algorithm itself
- Runtime system takes care of all messy details
  - Partitioning of input data
  - Scheduling program execution
  - Managing inter-machine communication
  - Handling machine failures

---

Divide and Conquer

```
"Problem" -> W1 -> W2 -> W3
"worker" -> T1 -> T2 -> T3
"Result"
```

Partition

Combine
MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python

- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix but also A9.com, AOL, The New York Times, Last.fm, Baidu.com, Joost, Veoh, etc.
  - The de facto big data processing platform
  - Rapidly expanding software ecosystem

- Lots of custom research implementations
  - For GPUs, cell processors, etc.

Hadoop: Storage & Compute on One Platform

The Traditional Way
- Expensive, Special purpose, “Reliable” Servers
  - Expensive Licensed Software
  - Hard to scale
  - Network is a bottleneck
  - Only handles relational data
  - Difficult to add new fields & data types
- Commodity “Unreliable” Servers
  - Hybrid Open Source Software
  - Scales out forever
  - No bottlenecks
  - Easy to ingest any data
  - Agile data access

Expensive & Unattainable
$30,000+ per TB

The Hadoop Way
- Storage (Disk)
- Memory
- Compute (CPU)
- Data Storage (SAN, NAS)

Affordable & Attainable
$300-$1,000 per TB

Evolution from Apache Hadoop to the Enterprise Data Hub A. Awadallah Co-Founder & CTO of Cloudera SMDB 2014
Big Data Requires A New Programming Approach

What we do
Copy Data to Applications

What we should do
Bring Applications to Data

Process-centric businesses use:
- Structured data mainly
- Internal data only
- “Important” data only
- Multiple copies of data

Evolution from Apache Hadoop to the Enterprise Data Hub A. Awadallah Co-Founder & CTO of Cloudera SMDB 2014

Map Reduce Foundations
Map and Reduce: The Origins

- The programming idea of Map, and Reduce is 40+ year old
  - Present in all Functional Programming Languages e.g., APL, Lisp and ML
  - Alternate Map names: Apply-All

- Higher Order Functions
  - take function definitions as arguments,
  - or return a function as output

- Map and Reduce are higher-order functions
  - Map processes each record individually
  - Reduce processes (combines) set of all records in a batch

### Map: A Higher Order Function

F(x: int) returns r: int  
Let V be an array of integers  
W = map(F, V)  
W[i] = F(V[i]) for all I  
i.e., apply F to every element of V

Examples in Haskell

map (+1) [1,2,3,4,5]  
== [2, 3, 4, 5, 6]  
map (toLowerCase) "abcDEFG12@#“  
== "abcdefg12@#“  
map (\mod\ 3) [1..10]  
== [1, 2, 0, 1, 2, 0, 1, 2, 0, 1]
Reduce: A Higher Order Function

- `reduce` also known as `fold`, accumulate, compress or inject

- Reduce/fold takes in a function and folds it in between the elements of a list

Fold-Left in Haskell

- Definition
  - `foldl f z [] = z`
  - `foldl f z (x:xs) = foldl f (f z x) xs`

- Examples
  - `foldl (+) 0 [1 .. 5] == 15`
  - `foldl (+) 10 [1 .. 5] == 25`
  - `foldl (div) 7 [34, 56, 12, 4, 23] == 0`
Fold-Right in Haskell

- **Definition**
  - \( \text{foldr } f z \left[ \right] = z \)
  - \( \text{foldr } f z \left(x:xs\right) = f x \left( \text{foldr } f z xs \right) \)

- **Example**
  - \( \text{foldr } \left( \text{div} \right) 7 \left[34, 56, 12, 4, 23\right] = 8 \)

Computing in Map Reduce
What Can I do in MapReduce?

- Three main functions:
  1. **Querying**
     - Filtering (distributed grep, etc.)
     - Relational-based (join, selection, projection, etc.)
  2. **Summarizing**
     - Computing Aggregates (word/record count, Min/Max/Average/Median/Standard deviation, etc.)
     - Data Organization (sort, indexing, etc.)
  3. **Analyzing**
     - Iterative Message Passing (graph processing)

... large datasets in offline mode for boosting other on-line processes

Grease Example

- Search input files for a given pattern: e.g., Given a list of tweets (username, date, text) determine the tweets that contain a word
  - **Map:** emits a (filename, line) if pattern is matched
  - **Reduce:** Copies results to output
Word Count Example

- Read text files and count how often words occur
  - The input is text files
  - The output is a text file
    - each line: word, tab, count
- Map: Produce pairs of (word, count = 1) from files
- Reduce: For each word, sum up the counts (i.e., fold) and copies result to the output

Inverted Index Example

- Generate an inverted index of words from a given set of files
- Map: parses a document and emits <word, docId> pairs
- Reduce: takes all pairs for a given word, sorts the docId values, and copies a <word, list(docId)> pair to the output
Real World Applications in MapReduce

<table>
<thead>
<tr>
<th>Organizations</th>
<th>MapReduce Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>Wide-range applications, grep/sorting, machine learning, clustering, report extraction, graph computation</td>
</tr>
<tr>
<td>Yahoo</td>
<td>Data model training, Web map construction, Web log processing using Pig, and much, much more</td>
</tr>
<tr>
<td>Amazon</td>
<td>Build product search indices</td>
</tr>
<tr>
<td>Facebook</td>
<td>Web log processing via both MapReduce and Hive</td>
</tr>
<tr>
<td>PowerSet(Microsoft)</td>
<td>HBase for natural language search</td>
</tr>
<tr>
<td>Twitter</td>
<td>Web log processing using Pig</td>
</tr>
<tr>
<td>New York Times</td>
<td>Large-scale image conversion</td>
</tr>
<tr>
<td>Others (&gt;74)</td>
<td>Details in <a href="http://wiki.apache.org/hadoop/PoweredBy">http://wiki.apache.org/hadoop/PoweredBy</a> (so far, the longest list of applications for MapReduce)</td>
</tr>
</tbody>
</table>

MapReduce Principle applied to BigData
Always maps/reduces on list of key/value pairs
- Records from the data source (lines out of files, rows of a database, etc.) are fed into the map function as input key=value pairs
  - map() produces one or more intermediate values along with an output key from the input
- After the map phase is over, all the intermediate values for a given output key are combined together into a list
  - reduce() accepts and combines those intermediate values into one or more final values for that same output key
  - in practice, only one final value per key

Automatic Parallelization:
- Depending on the size of RAW INPUT DATA ➔ instantiate multiple map() tasks
- Depending upon the number of intermediate <key, value> partitions ➔ instantiate multiple reduce() tasks
- Master program divvies up tasks based on data location
  - tries to have map() tasks on same machine as physical file data, or at least same rack
MapReduce for BigData

- Map/Reduce tasks are executed in parallel on a cluster
- Fault tolerance is built in the framework
- Specific systems/implementation aspects matters
  - How is data partitioned as input to map
  - How is data serialized between processes
- Cloud specific improvements:
  - Handle elasticity
  - Take cluster topology (e.g., node proximity, node size) into account

Source: Google Developers

Execution on Clusters

1. Input files split (M splits)
2. Assign Master & Workers
3. Map tasks
4. Writing intermediate data to disk (R regions)
5. Intermediate data read & sort
6. Reduce tasks
7. Return
MapReduce Data Flow in Hadoop with no Reduce Tasks

MapReduce Data Flow in Hadoop with a Single Reduce Task
MapReduce Data Flow in Hadoop with Multiple Reduce Tasks

- Programmers must specify:
  *map* \((k, v) \rightarrow <k', v'>\)*
  *reduce* \((k', v') \rightarrow <k', v'>\)*
  - All values with the same key are reduced together
- Optionally, also:
  *partition* \((k', \text{number of partitions}) \rightarrow \text{partition for } k'\)
  - Often a simple hash of the key, e.g., hash\((k')\) mod n
  - Divides up key space for parallel reduce operations
  *combine* \((k', v') \rightarrow <k', v'>\)*
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

MapReduce Programming Constructs in Hadoop

- **Input Format**
  - data \(\rightarrow K_1, V_1\)
- **Mapper**
  - \(K_1, V_1 \rightarrow K_2, V_2\)
- **Combiner**
  - \(K_2, \text{iter}(V_2) \rightarrow K_2, V_2\)
- **Partitioner**
  - \(K_2, V_2 \rightarrow \text{int}\)
- **Reducer**
  - \(K_2, \text{iter}(V_2) \rightarrow K_3, V_3\)
- **Out. Format**
  - \(K_3, V_3 \rightarrow \text{data}\)
The Execution Framework Handles “Everything Else”

- The execution framework handles “everything else”
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
  - All algorithms must expressed in $m$, $r$, $c$, $p$
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

Execution Phases

Map Phase

<table>
<thead>
<tr>
<th>Input</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>

Intermediate: $k1:v, k1:v, k2:v, k1:v, k3:v, k4:v, k4:v, k3:v, k4:v, k1:v, k3:v$

Shuffle Phase

Grouped: $k1:v, k2:v, k3:v, k4:v, k5:v$

Reduce Phase

<table>
<thead>
<tr>
<th>Output</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
Automatic Parallel Execution in MapReduce

- Usually many more map tasks than machines
- E.g.
  - 200K map tasks
  - 5K reduce tasks
  - 2K machines

Map/Reduce Execution in Clusters

- Several map or reduce tasks can run on a single computer
- Each intermediate file is divided into $R$ partitions, by partitioning function
- Each reduce task corresponds to one partition
Moving Data From Mappers to Reducers

- Mappers need to partition their output for different reducers
- Reducers need to collect their input from all mappers and group it by key
  - Keys at each reducer are processed in order!

- **Shuffle & sort** phase—synchronization barrier between map & reduce phase
  - Divides the map output into chunks
  - Collects the output from all map executions
  - Transforms the map output into the reduce input

- Often one of the **most expensive parts** of a MapReduce execution!

---

Shuffle and Sort in Map Reduce

- Spilled to a new disk file when almost full
- Spill files merged into single output file
- Spill files on disk: partitioned by reduce task, each partition sorted by key
- Merge happens in memory if data fits, otherwise also on disk
- Reduce task starts copying data from map task as soon as it completes. Reduce cannot start working on the data until all mappers have finished and their data has arrived.
Tools for Synchronization

- **Cleverly-constructed data structures**
  - Bring partial results together

- **Sort order of intermediate keys**
  - Control the order in which reducers process keys

- **Partitioner**
  - Control which reducer processes which keys

- **Preserving state in mappers and reducers**
  - Capture dependencies across multiple keys and values

Component Overview
Google’s MapReduce inspired Yahoo’s Hadoop

Now as an open source (Java) Project of Apache hadoop.apache.org

13 December, 2017: Release 3.0.0 available

Distribute processing of petabytes data across thousands of commodity machines

In 2008 about 100k MapReduce jobs per day in Google

- over 20 petabytes of data processed per day
- each job occupies about 400 servers

Easy to use since run-time complexity hidden from the users

Hadoop Key Components

- Hadoop Distributed File System (HDFS) - distributes data
  - Store big files across machines
  - Store each file as a sequence of blocks
  - Blocks of a file are replicated for fault tolerance
- Hadoop MapReduce - distributes applications
How do we get Data to the Workers?

- What's the problem here?

Distributed File System

- Don't move data to workers… Move workers to the data!
  - Store data on the local disks for nodes in the cluster
  - Start up the workers on the node that has the data local!

- Why?
  - Not enough RAM to hold all the data in memory
  - Common local-area network (LAN) speeds go up to 100 Mbit/s, which is about 12.5MB/s
  - Today's hard disks provide a lot of storage and transfer speeds around 40-60MB/s

- A distributed file system is the answer
  - GFS (Google File System)
  - DFS for Hadoop (= GFS clone)
Distributed File System Principles

- **Single Namespace** for entire cluster
- **Data Coherency**
  - Write-once-read-many access model
  - Client can only append to existing files
- **Files are broken up into blocks**
  - Typically 128 MB block size
  - Each block replicated on multiple DataNodes
- **Intelligent Client**
  - Client can find location of blocks
  - Client accesses data directly from DataNode

Hadoop Distributed File System (HDFS)

- **Very Large Distributed File System**
  - 10K nodes, 100 million files, 10 PB
- **Assumes Commodity Hardware**
  - Files are replicated (3 times by default) to handle hardware failure
  - Detect failures and recovers from them
- **Optimized for Batch Processing**
  - Data locations exposed so that computations can move to where data resides
  - Provides *very high aggregate bandwidth*
- **User Space**, runs on top of the file systems of the underlying OS
Hadoop Distributed File System (HDFS)

- **Master** manages the file system and access to files by clients
  - Runs on master node of the HDFS cluster
  - Directs DataNodes to perform their low-level I/O tasks

- **Slave** manages the storages attached to the nodes running on
  - Runs on each slave machine in the HDFS cluster
  - Does the low-level I/O work

- **Files stored in many blocks**
  - Each block has a block Id
  - Block Id associated with several nodes hostname:port (depending on level of replication)

---

**HDFS Architecture**

1. **filename**
2. **BlockId, DataNodes**
3. **Read data**

**NameNode**: Maps a file to a file-id and list of DataNodes

**DataNode**: Maps a block-id to a physical location on disk

**SecondaryNameNode**: Periodic merge of Transaction log
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** stored in multiple directories (on a local/remote FS)
  - Records file creations, file deletions, etc.

---

WordCount

A Simple Hadoop Example

Word Counting with MapReduce

- How can we do word count over a given set of documents in parallel?

Before reduce functions are called, for each distinct key, the list of its values is generated.
WordCount Mapper

public class TokenCounterMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}

WordCount Reducer

public static class TokenCounterReduce extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
Job Configuration

```java
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
}
```

Hadoop Job Tuning

- Choose appropriate number of mappers and reducers
- Define combiners whenever possible
- Consider Map output compression
- Optimize the expensive shuffle phase (between mappers and reducers) by setting its configuration parameters
- Profiling distributed MapReduce jobs is challenging
Mechanics of Programming
Hadoop Jobs

Anatomy of a Hadoop Job in MR v1.0

- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of a running task is called a task attempt
  - Multiple jobs can be composed into a workflow
Creating the Mapper

- One instance of your Mapper is initialized by the `MapTaskRunner` for a `TaskInProgress`
  - Exists in separate process from all other instances of Mapper – no data sharing!

```java
public void map (WritableComparable key, Writable value, Context context)
```

Data Types in Hadoop

- **Writable**: Defines a de/serialization protocol. Every data type in Hadoop is a `Writable`.
- **WritableComparable**: Defines a sort order. All keys must be of this type (but not values).
- **IntWritable**, **LongWritable**, **Text**: Concrete classes for different data types.
- **SequenceFiles**: Binary encoded of a sequence of key/value pairs.
What is Writable?

- Hadoop defines its own “box” classes for strings (Text), integers (IntWritable), etc.
  - All values are instances of Writable
  - All keys are instances of WritableComparable

- Writing for cache coherency

```java
while (more input exists) {
    myIntermediate = new intermediate(input);
    myIntermediate.process();
    export outputs;
}
```

Getting Data to the Mapper

- Data sets are specified by InputFormats
  - Defines input data (e.g., a directory)
  - Identifies partitions of the data that form an InputSplit
  - Factory for RecordReader objects to extract (k,v) records from the input source
FileInputFormat and Friends

- **TextInputFormat**
  - Treats each ‘\n’-terminated line of a file as a value

- **KeyValueTextInputFormat**
  - Maps ‘\n’-terminated text lines of “k SEP v”

- **SequenceFileInputFormat**
  - Binary file of (k, v) pairs with some additional metadata

- **SequenceFileAsTextInputFormat**
  - Same, but maps (k.toString(), v.toString())

- **FileInputFormat** will read all files out of a specified directory and send them to the mapper
  - Delegates filtering this file list to a method
    - subclasses may override e.g., create your own “xyzFileInputFormat” to read *.xyz from directory list

Record Readers

- Each InputFormat provides its own RecordReader implementation
  - Provides (unused?) capability multiplexing

- **LineRecordReader**
  - Reads a line from a text file

- **KeyValueRecordReader**
  - Used by KeyValueTextInputFormat
Input Split Size

FileInputFormat will divide large files into chunks
- Exact size controlled by mapreduce.input.fileinputformat.split.minsize

RecordReaders receive file, offset, and length of chunk

Custom InputFormat implementations may override split size
- e.g., “NeverChunkFile”

WritableComparator

- Compares writableComparable data
  - Will call writableComparable.compare()
  - Can provide fast path for serialized data
Sending Data ToReducers: The Context Class

- Both Mapper and Reducer define an inner class called Context which implements the JobContext interface
  - Job also implements JobContext
    - when you create a new Job, you also set the context for the Mapper and Reducer

- Some methods of Context:
  - write: generate an output key/value pair
  - progress and setStatus: report progress or set the status of a task
  - getCounter: get access (read/write) to the value of a Counter
  - getConfiguration: return the configuration for the job
  - getCacheFiles: get cache files set in the Configuration

Partitioning

- Which reducer will receive the intermediate output keys and values?
  - (key, value) pairs with the same key end up at the same partition

- Mappers partition data independently
  - they never exchange information with one another

- Hadoop uses an interface called Partitioner to determine which partition a (key, value) pair will go to

- A single partition refers to all (key, value) pairs which will be sent to a single reduce task: #partitions = #reduce tasks
  - each Reducer can process multiple reduce tasks

- The Partitioner determines the load balancing of the reducers
  - JobConf sets Partitioner implementation
The Partitioner interface

- The Partitioner interface defines the getPartition() method
  - **Input**: a key, a value and the number of partitions
  - **Output**: a partition id for the given key, value pair
- The default Partitioner is the HashPartitioner:
  ```java
  int getPartition(K key, V value, int numPartitions) {
    return key.hashCode() % numPartitions;
  }
  ```

<table>
<thead>
<tr>
<th>Key</th>
<th>key.hashCode()</th>
<th>partitionId (0-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>12847</td>
<td>1</td>
</tr>
<tr>
<td>world</td>
<td>23874</td>
<td>0</td>
</tr>
<tr>
<td>map</td>
<td>82375</td>
<td>1</td>
</tr>
<tr>
<td>reduce</td>
<td>12839</td>
<td>2</td>
</tr>
</tbody>
</table>

numPartitions = 3

Partition and Shuffle
Creating the Reducers

- One instance of your Reducer is initialized by the ReduceTaskRunner for a TaskInProgress
  - Exists in separate process from all other instances of Reducer – no data sharing!

```java
public void reduce(WritableComparable key, Iterator values, Context context)
```

- Keys & values sent to one partition all go to the same reduce task
- Calls are sorted by key:
  - “earlier” keys are reduced and output before “later” keys

Finally: Writing The Output
OutputFormat

- The OutputFormat and RecordWriter interfaces dictate how to write the results of a job back to the underlying permanent storage.
- The default format (TextOutputFormat) will write \((key, value)\) pairs as strings to individual lines of an output file.
  - Writes “key\nval” strings using the toString() methods of the keys and values.
- The SequenceFileOutputFormat will keep the data in binary, so it can be later read quickly by the SequenceFileInputFormat.
  - Uses a binary format to pack \((k, v)\) pairs.
- NullOutputFormat
  - Discards output.

Additional Languages & Components
Every subproject relies on HDFS for input/output data and in Map/Reduce for processing it in different ways.

Hadoop Subprojects Dependencies

Hadoop Levels of Abstraction
MapReduce Cloud Service

- Providing MapReduce frameworks as a service in clouds becomes an attractive usage model for enterprises.

- A MapReduce cloud service allows users to cost-effectively access a large amount of computing resources without creating their own cluster.

- Users are able to adjust the scale of MapReduce clusters in response to the change of the resource demand of applications.

Takeaway

- MapReduce’s data-parallel programming model hides complexity of distribution and fault tolerance.

- Principal philosophies:
  - Make it scale, so you can throw hardware at problems
  - Make it cheap, saving hardware, programmer and administration costs (but requiring fault tolerance)

- MapReduce is not suitable for all problems, but when it works, it may save you a lot of time.
References

- We rely on the new org.apache.hadoop.mapreduce API, but…
  - Many existing programs might be written using the old API org.apache.hadoop.mapred
  - Some old libraries might only support the old API
- Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. Usenix SDI '04, 2004.
- David DeWitt, Michael Stonebraker, "MapReduce: A major step backwards", craig-henderson.blogspot.com
  - http://scienceblogs.com/goodmath/2008/01/databases_are_hammers_mapreduc.php
- CS9223 – Massive Data Analysis J. Freire & J. Simeon New York University Course 2013
- INFM 718G / CMSC 828G Data-Intensive Computing with MapReduce J. Lin University of Maryland 2013
- The MapReduce Programming Model: Introduction and Examples Jose Maria Alvarez-Rodriguez FP7 RELATE-ITN April, 2013