Introduction to Cloud Computing

MapReduce and Hadoop

15-319, spring 2010

17th Lecture, Mar 16th

Majd F. Sakr

Lecture Goals

- Transition to MapReduce from Functional Programming
- Understand the origins of MapReduce
- Explore the Hadoop API and Interface
- One hands-on program to dive into Hadoop!

Lecture Outline

- **Functional Programming Review and MapReduce**
- **Hadoop**
  - Components
  - MapReduce
  - Programming Model
  - Map and Reduce Functions
  - Terminology – Jobs and Tasks
- **Java Implementations and Classes**
  - Mapper and Reducer Classes
  - Writing a MapReduce Driver
  - Input, Output and Other handler classes
- **Simple Program (Hands-On)**
- **Details of Hadoop**
  - Handling Input and Output
  - Job Execution and Data Flow
  - Hadoop Optimizations

Functional Programming Review

- Functional operations do not modify data structures: They always create new ones
- Original data still exists in unmodified form
- Data flows are implicit in program design
- Order of operations does not matter

Functional Programming

- **Process lists of data very frequently**
  - Iterators: operators over lists
- **Map**
- **Fold**
- **Map operation can be assigned to each element of a list independently**
- **If the operation performed in a fold operation is commutative, it can be parallelized**

Implicit Parallelism in Functional Programming

- In a purely functional setting, elements of a list being computed by map cannot see the effects of the computations on other elements
- If order of application of *f* to elements in list is *commutative (and associative)*, we can reorder or parallelize execution of *f*
- This is the "secret" that MapReduce exploits

Enter MapReduce

- Functional programming on top of distributed processing
- Motivated by need to process large amounts of data using hundreds (or thousands) of processor nodes
- Data is central to the computation, place the program closest to the data it will be working on.
- Provide a clean abstraction to programmers similar to functional programming
- The Interface deals with all the messy details

MapReduce History

- **Developed by Google to simplify their data processing jobs on large data**
  - Details emerged from two published papers:
    - James Dean, Sanjay Ghemawat, ***MapReduce : Simplified Data Processing on Large Clusters,*** Proceedings of *OSDI '04,* 2004
    - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, ***Google File System,*** Proceedings of *Symposium of Operating Systems Principles,* ACM SIGOPS, 2004
- **Since Google's implementation is proprietary and not available to public, an Apache Project called Hadoop emerged as an open source implementation**
  - Primary Contributors: Yahoo! , Facebook

Motivating Example

- **Processing Web Data on a Single Machine**
  - 20+ billion web pages x 20KB = 400+ terabytes
  - One computer can read 30-35 MB/sec from disk
  - ~ four months to read the web
  - ~1,000 hard drives just to store the web
  - Even more to *do something with the data*
- **Takes too long on a single machine, but with 1000 machines?**
  - < 3 hours to perform on 1000 machines
  - But how long to program? What about the overheads?
    - Communication, coordination, recovery from machine failure
    - Status Reporting, Debugging, Optimization, Locality
    - Reinventing the Wheel: This has to be done for every program!

MapReduce Features

- Automatic Parallelization and Distribution of Work
- Fault-Tolerant
- Status and Monitoring Tools
- Clean Abstraction for Programmers

Typical Problem Solved by MapReduce

1. Read a lot of Data
2. MAP: extract something you need from each record
3. Shuffle and Sort
4. REDUCE: aggregate, summarize, filter or transform
5. Write the results

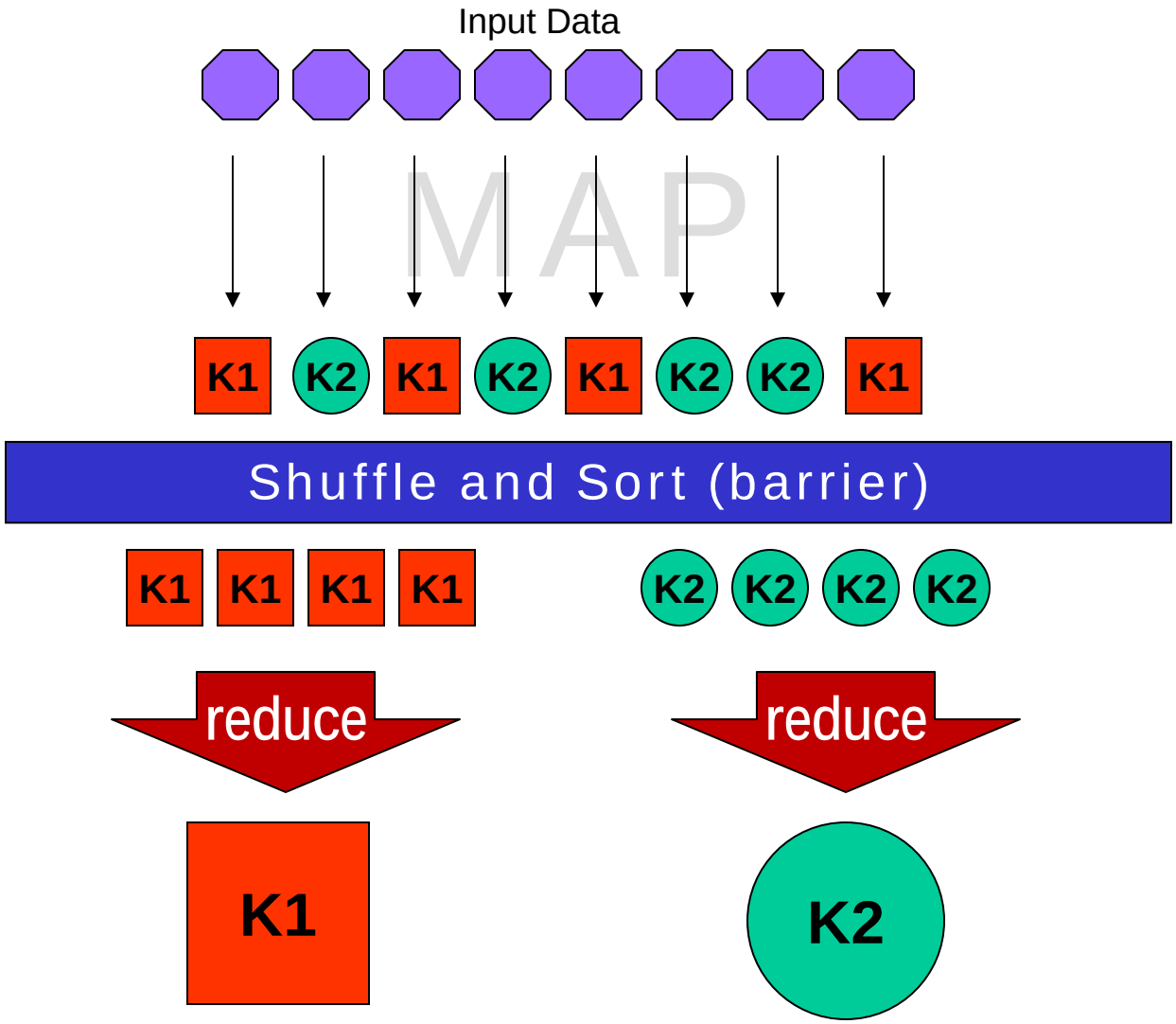
Outline stays the same,
Map and Reduce change to fit the Problem

Model seems restrictive but it is Turing Complete.
Multiple maps and reduces needed to solve a complex problem.

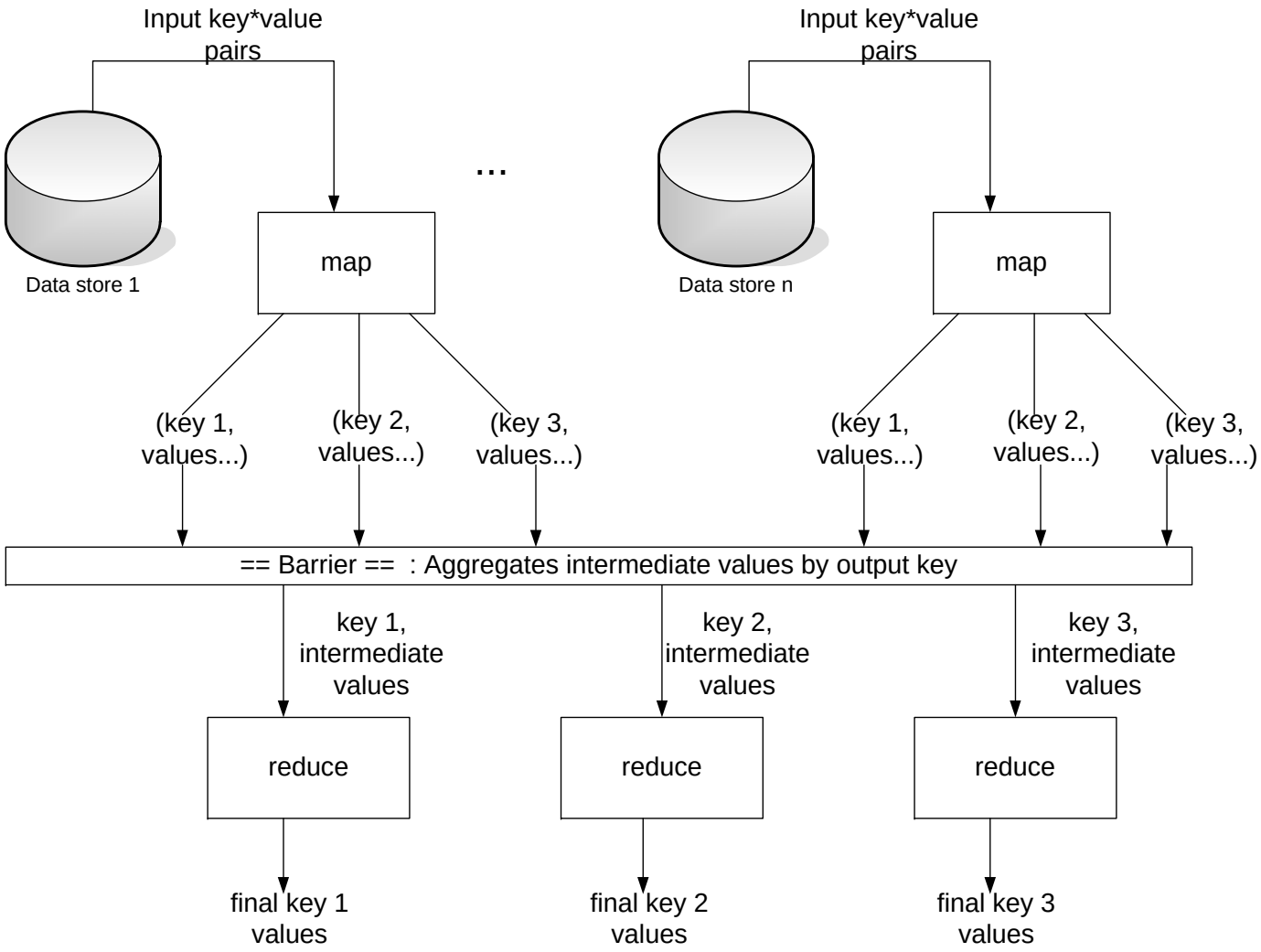
Programming Model

- Fundamentally similar to Functional Programming
- Users implement interfaces to following two functions:
  - `map (in_key, in_value) -> (out_key, intermediate_value) list`
  - `reduce (out_key, intermediate_value list) -> out_value list`

A Typical MapReduce Program - I



A Typical MapReduce Program - II



Parallelism

- **Map functions run in parallel, create intermediate values from each input data set**
  - The programmer must specify a proper input split (chunk) between mappers to enable parallelism
- **Reduce functions also run in parallel, each will work on different output keys**
  - Number of reducers is a key parameter which determines map-reduce performance
- **All values are processed independently**
  - Reduce phase cannot start until the map phase is completely finished

Data Locality

- Master program creates *tasks* based on the location of the data; tries to send map() tasks to the same machine or at least same rack
- Map() task inputs are divided into 64 MB blocks (same as HDFS/GFS) or at file boundaries.
- Minimizes communication at the network level

Fault Tolerance

- **Master keeps track of progress of each task and worker nodes**
  - If a node fails, it re-executes the completed as well as in-progress map tasks on other nodes that are alive
  - It also executes in-progress reduce tasks.
- **If particular input key/value pairs keep crashing**
  - Master blacklists them and skips them from re-execution
- **Tolerate small failures, allow the job to run in best-effort basis**
  - For large datasets containing potentially millions of records, we don't want to stop computation for a few records not processing correctly
  - User can set the failure tolerance level

Optimizations

- **Reduce tasks will start only after all the map operations are complete - bottleneck**
  - **Use a combiner** (a mini-reducer on the output of a map) to reduce the output and the amount of communication during sort and shuffle
  - **Speculative execution** – New feature in Hadoop, enables multiple task attempts to run in parallel and use the results of the first one to finish
- **MapReduce programs are bound by the slowest element – the last mapper to finish or the time taken for the Reduce operations to finish.**

Time to Dive into Hadoop!

What is Hadoop?

- Open Source Implementation of Google's Distributed Computing Projects
- Includes open source versions of MapReduce, BigTable, GFS etc.
- Supported by the Apache Foundation
- Primary Contributors – Yahoo!, Facebook
- Work in Progress – lots of features are unstable
- Used by Yahoo to run a 2000+ node distributed cluster, and many other web companies as well.

Quick Look at Hadoop Components

"Google Calls It"	Hadoop Equivalent	Description
MapReduce	MapReduce	Java Implementation of the MapReduce Programming Model
GFS	HDFS	Distributed File System
Sawzall	Pig Hive	Data Flow Language Distributed Data Warehouse and Query Engine
BigTable	HBase	Distributed Database
Chubby	ZooKeeper	Distributed Co-ordination Service

Hadoop MapReduce

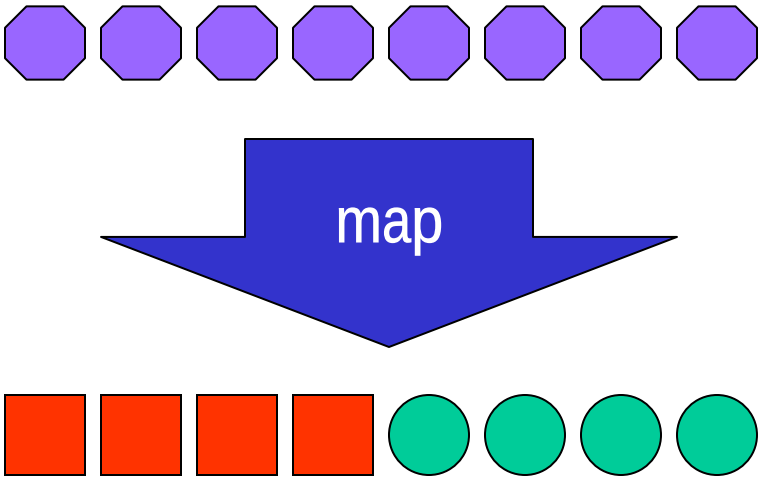
- **The "Meat" of Hadoop. Implementation of Google's MapReduce Programming Model**
- **Express complex tasks in the form of Maps and Reduces to enable large-scale processing of data**
- **Functional programming meets distributed computing using batch processing of data**
- **Benefits:**
  - Automatic Parallelization and Distribution of Data
  - Fault Tolerance
  - Status and Monitoring Tools
  - Clean Programming Abstraction to help focus on the solving the problem

MapReduce Programming Model

- Functional Programming concepts of map and fold
- Users Implement Map and Reduce Functions
  - `map (in_key, in_value) -> (out_key, intermediate_value) list`
  - `reduce (out_key, intermediate_value) list -> out_value list`

The Map Function

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., *(filename, line).*
- map() produces (emits) one or more intermediate values along with an output key from the input.



Example – UpperCase Mapper

```
let map(k,v) =
  emit(k.toUpper(),v.toUpper())
```

("abcd", "efgh")→("ABCD", "EFGH")

("CMUq", "hadoop")→("CMUQ", "HADOOP")

("foo", "bar")→("FOO", "BAR")

Example – Filter Mapper

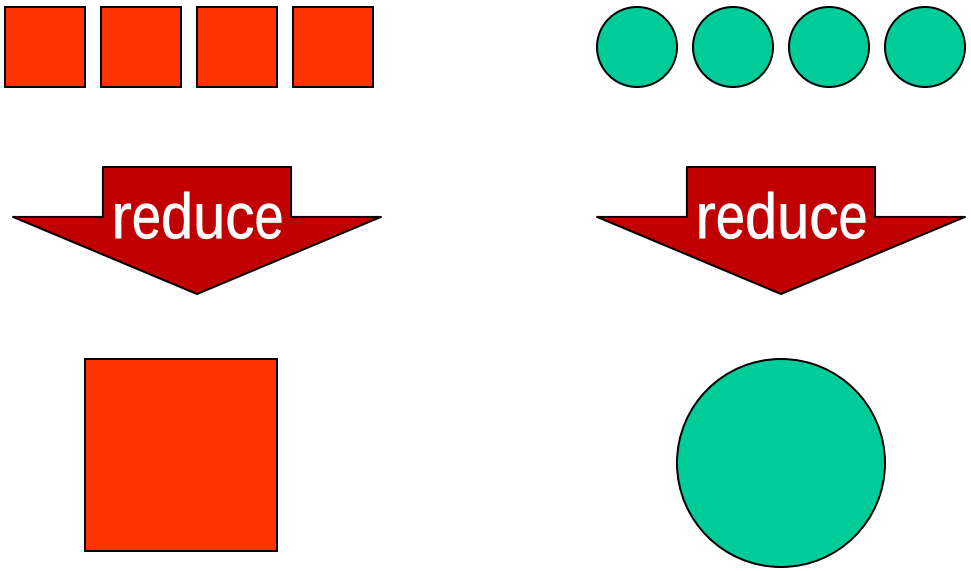
```
let map(k, v) =
      if (v%2==0) then emit(k, v)
```

("foo", 7)→(nothing)

("test", 10)→("test",10)

The Reduce Function

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more final values for that same output key



Example – Sum Reducer

```
let reduce(k, vals) =
{
sum = 0
  foreach int v in vals:
  sum += v
emit(k, sum)
}
```

("A", [42, 100, 312])→("A", 454)

("B", [12, 6, -2])→("B", 16)

Hadoop MapReduce Terminology

- **Job**
  - Execution of a “complete” program (mapper and reducer) across an input data set
- **Task**
  - Execution of a single mapper or reducer on a slice of data (task in progress)
- **Task Attempt**
  - A particular instance of an attempt to execute a task on a machine or node

Example of Terminology

- Wordcount on 20 files: One *Job*
- The Job implies 20 map *tasks* for each file plus a few reducer tasks
- There will be at least 20+ *task attempts* total. More task attempts if a node crashes, etc.

MapReduce Java Package Structure

- **`org.apache.hadoop.*`**
  - **`mapred`** – Legacy MapReduce API (upto 0.19)
  - **`mapreduce`** – New MapReduce API (0.20+)
  - **`conf`** – Job Configuration Utilities
  - **`io`** – Input/Output formats
  - **`fs`** – File System Handlers
  - **`util`** – Utilities for hadoop

The `Mapper` Java Class

- To write your own mapper, you extend the `MapReduceBase` Class, and override the `map()` function in the `Mapper` Class, inside `org.apache.mapred`

```
public class NewMapper extends MapReduceBase implements Mapper
{
  //User Defined Class Variables and Functions

  public void map(WritableComparable key, Writable values,
                  OutputCollector output, Reporter reporter) throws IOException
  {
    //Map Function Goes Here
    output.collect(key,value);
  }
}
```

Input/Output Datatypes

- **`Writable` Class**
  - Hadoop type for writing data as a byte stream (Serialization)
  - *IntWritable*, *Text* etc
  - All values must be *Writable*
  - You can create your own custom writable for your own input format.
- **`WritableComparable` Class**
  - Subclass of **`Writable`**
  - Required for sorting and shuffling for reducers
  - All keys must be type *WriteableComparable*

Java Primitive	Writeable Implementation
`boolean`	`booleanWritable`
`byte`	`ByteWritable`
`int`	`IntWritable`
`float`	`FloatWritable`
`long`	`LongWritable`

Java Type	Writeable Implementation
`String`	`Text`
`Bytes`	`BytesWritable`

The Reducer Java Class

- To write your own reducer, you extend the MapReduceBase Class, and override the reduce() function in the Reducer Class.

```
public class ExampleReducer extends MapReduceBase implements Reducer
{
        public void reduce(WritableComparable _key, Iterator values,
                           OutputCollector output, Reporter reporter)
        throws IOException
        {
                while (values.hasNext())
                {
                // process value
                }
        }
}
```

The `Reducer` Java Class Continued…

- The reducer class is executed for every key emitted from the mapper, we use the *Iterator* class to iterate through the values of each key
- You can perform the required reduction operation on the values for each key and emit the key,value pairs as necessary

The `OutputCollector` and `Reporter` Classes

- The `OutputCollector` Class is used to handle output key-value pairs emitted by mappers and reducers
  - `output.collect(k,v)`
- The `Reporter` class is used to update counters and status messages you see onscreen when hadoop executes.
- The functionality of both classes have now been merged in Hadoop 0.20 using `context` objects

Putting it all together in the Driver Class

- **Once you have your mappers and reducers set, you can write a driver class**
- **You configure a `JobConf` object with the following information**
  - Input/Output File formats
  - Input/Output Key-Value Formats
  - Mapper and Reducer Classes

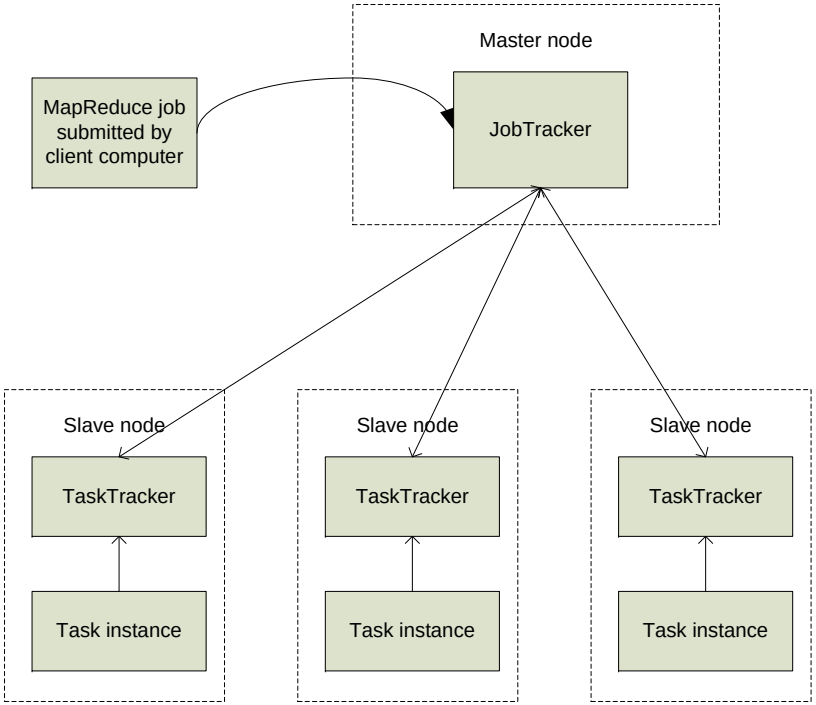
```
public class Example {
    public static void main(String[] args)
    {
    JobClient client = new JobClient();
    JobConf conf = new JobConf(Example.class);

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setInputPath(new Path("src"));
    conf.setOutputPath(new Path("out"));
    conf.setMapperClass(ExampleMapper.class);
    conf.setReducerClass(ExampleReducer.class);
    client.setConf(conf);
    try {
        JobClient.runJob(conf);
        } catch (Exception e) {
                e.printStackTrace();
        }
    }
}
```

A Simple Program (Hands-On)

- **Lets try to write a Hadoop program that counts the number of even and odd numbers in a list.**
- **Input: Text File with one integer value per line**
- **Output: Text File Containing the number of even and Number of Odd numbers in the Input file**
- **Mapper**
  - Read each line, parse for integer and emit(“even/odd”,1);
- **Reducer**
  - Sum up all the values of each key (even/odd)
- **That’s It!**

How does a Job Execute in Hadoop?

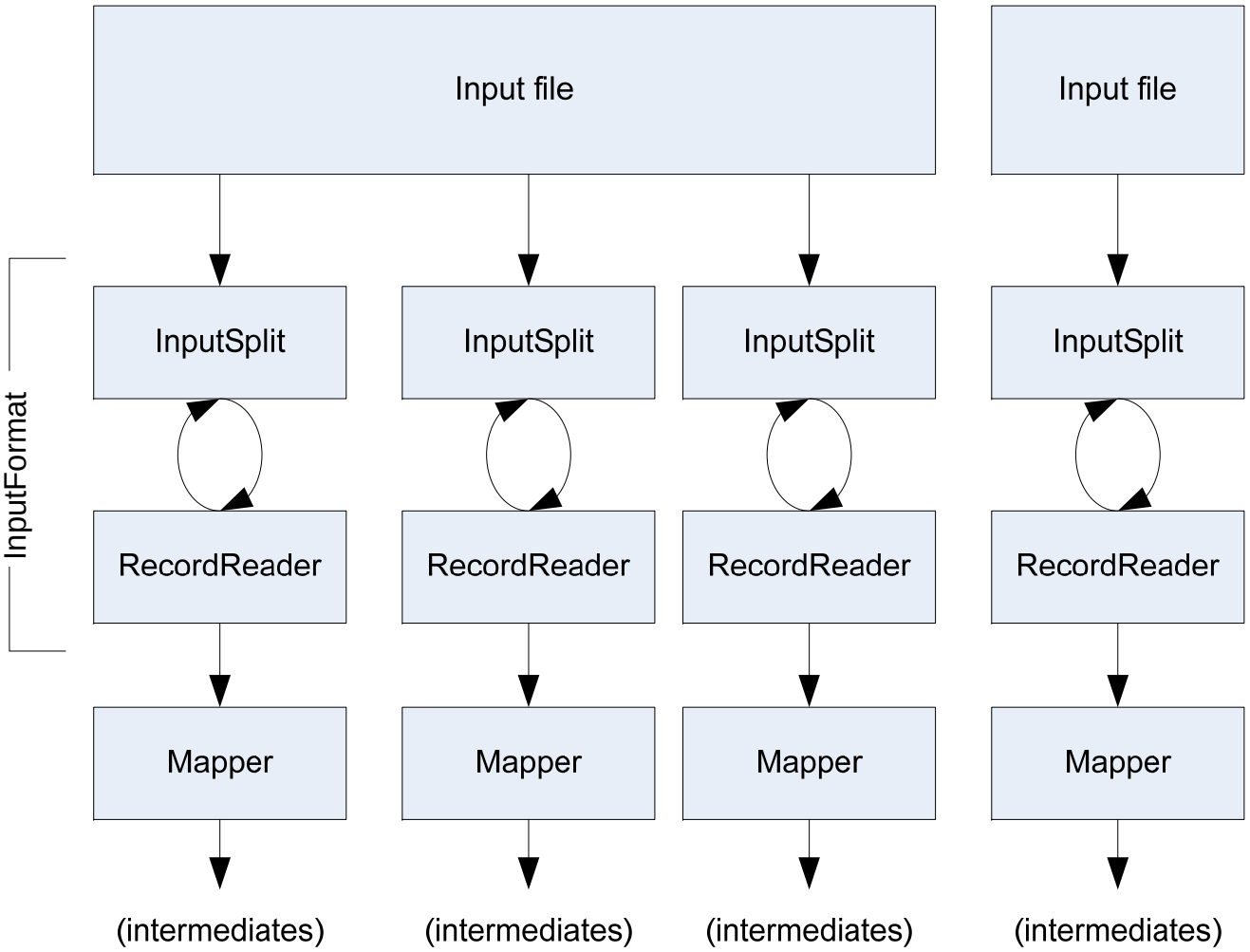
- MapReduce Job is sent to the masternode JobTracker
- JobTracker creates the tasks and sends them to the individual slave node TaskTrackers
- The TaskTracker creates task instances for each task and runs them, reports status and results back to the JobTracker



Job Configuration in Hadoop

- **Via the JobConf Object**
  - Creates a JobConfigration XML file that will packaged along with the Hadoop JAR file containing your program
  - You specify the Mapper, Reducer, Input and Output Files/Formats etc.
  - There are additional attributes that can be specified, each as XML attributes in the conf file.
- **The Hadoop system uses the job XML file along with other configuration parameters in the Framework (core-site.xml etc.)**
  - Some parameters can be defined as *final* to make sure they are not overridden.

Data Flow from Input to Mappers



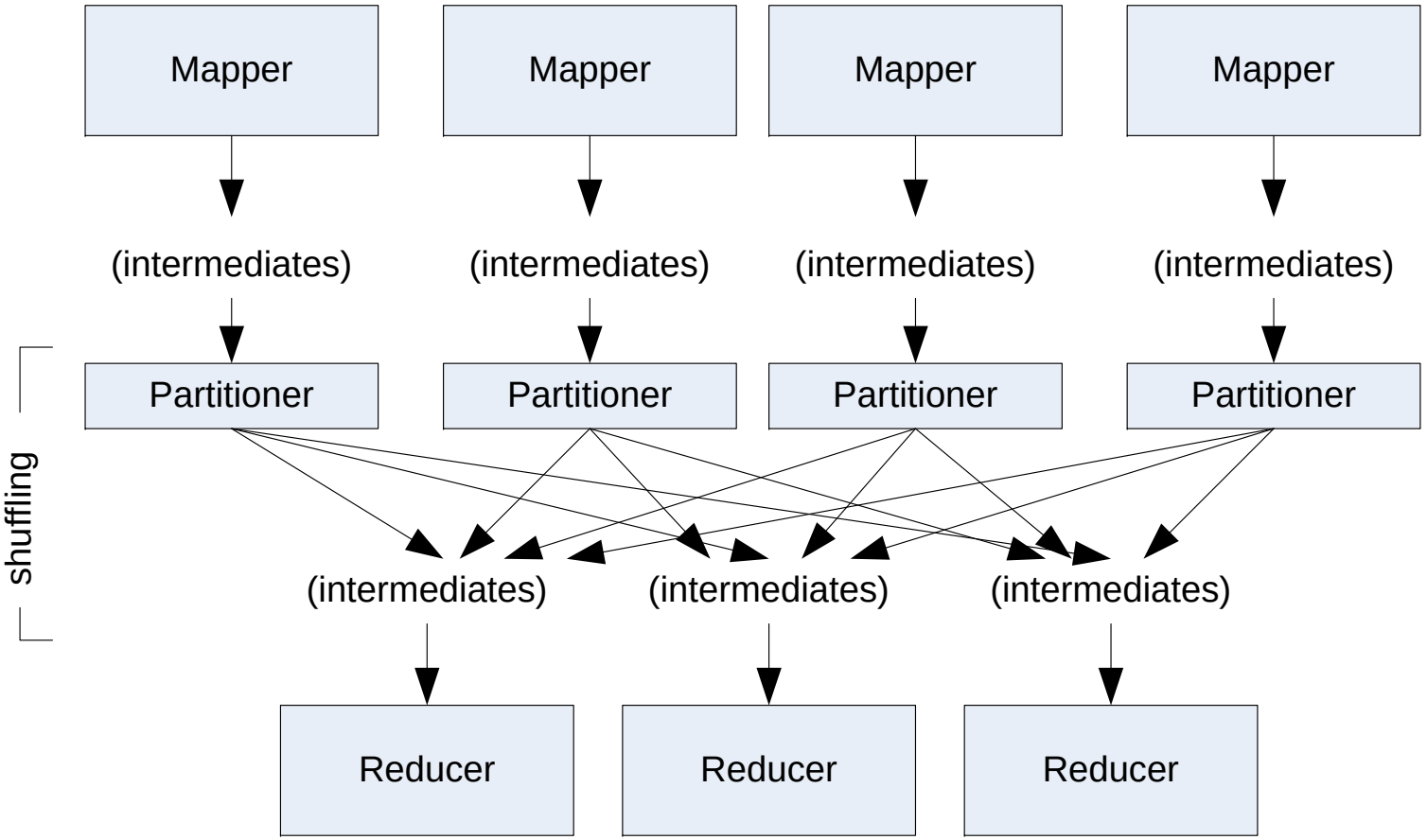
Handling Input

- **First we must specify an input file type**
  - **Text Files** – the basic file type in Hadoop, reads text files and provides input (key,value) pairs to the map tasks.
  - **Sequence Files** – Binary format to store sets of key/value pairs in binary form. Supports compression as well.
  - **Map Files** - pairs of sorted input SequenceFiles, one file (**`data`**) contains key/value pairs as records and the other (**`index`**) contains key/location pairs where location is the location of the key in the file **`data`**

Handling Input

- **Input file is split into chunks using an `InputSplit`**
  - Default split occurs at 64MB and file boundaries
  - You can specify your own **`InputSplit`** for your Input Formats
  - One Map task for each **`InputSplit`**
- **A record is a (key,value) pair**
  - Read using the **`RecordReader`** class
  - You can specify your own **`RecordReader`** for your own input formats
  - Each **`InputFormat`** provides its own **`RecordReader`** implementation
    - **`LineRecordReader`** – Reads a line from a text file
    - **`KeyValueRecordReader`** – Used by **`KeyValueTextInputFormat`**

Data Flow from Mappers to Reducers



Reducer in Detail

- **Reduces a set of intermediate values which share a key to a smaller set of values**

- **Shuffle**
  - Copies the sorted output from each mapper using HTTP across the network

- **Sort**
  - Sorts reduce inputs by keys
  - Shuffle and sort phases occur simultaneously
  - Secondary Sort using custom functions

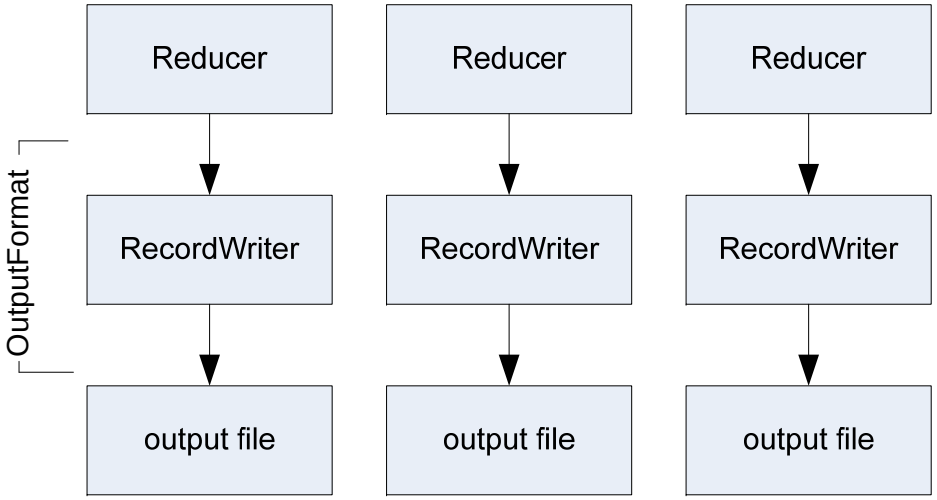
- **Reduce**
  - Framework class Reduce for each key and collection of values

The `Partitioner` Class

- **Responsible for creating the partitions for the Sort and Shuffle Phase**
  - Dictates what each reducer receives from each mapper
  - Can be used to create a "Global Order" of keys
  - Total partitions is same as number of map tasks
- **`HashPartitioner` used by default**
  - Uses **`key.hashCode()`** to return partition num
- **`JobConf` sets `Partitioner` implementation**

Handling Output

- **Each Reducer writes it's own output file**
- **Uses a RecordWriter that is part of a OutputFormat object**
  - **`TextOutputFormat`** – Writes "key val\n" strings to output file
  - **`SequenceFileOutputFormat`** – Uses a binary format to pack (k, v) pairs
  - **`NullOutputFormat`** – Discards output



Optimizations for a MapReduce Job

- **Use a Combiner Function**
  - Essentially a mini-reducer that runs on a mapper's output
  - Executed locally on each node after a map task
  - Helps in reducing data to be transferred during the shuffle phase

```
conf.setCombinerClass(ExampleReducer.class);
```

Conclusions

- **Basic overview of Hadoop and MapReduce**
- **Introduction to the Hadoop framework to get you started on writing programs and code**
- **For More Information**
  - Tom White : Hadoop : The Defnitive Guide, O'Reilly Press
  - Pro Hadoop : Jason Venner

References

- http://code.google.com/edu/submissions/mapreduce-minilecture/listing.html
- http://www.cloudera.com/wp-content/uploads/2010/01/2-MapReduceAndHDFS.pdf
- http://www.cloudera.com/wp-content/uploads/2010/01/4-ProgrammingWithHadoop.pdf
- http://search.iiit.ac.in/cloud/lectures/MapReduce.ppt
- http://search.iiit.ac.in/cloud/lectures/HadoopDetailedView.ppt