# CSE6331: Cloud Computing

Leonidas Fegaras

University of Texas at Arlington


## Map-Reduce Fundamentals

Based on:
J. Simeon: Introduction to MapReduce
P. Michiardi: Tutorial on MapReduce
J. Leskovec, A. Rajaraman, J. Ullman: Map-Reduce and the New Software Stack, http://www.mmds.org

# Map-Reduce

"Simplified Data Processing on Large Clusters", by Dean and Ghermawat, Google Inc:

> *A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs*

- 2004: Google Map-Reduce based on GFS (Google File System)
- 2006: Apache Hadoop
- 2007: HDFS, Pig
- 2008: Cloudera founded
- 2009: MapR founded
- 2010: HBase, Hive, Zookeeper
- 2013: Yarn
- 2014: Spark

# Motivation

- Large-Scale Data Processing (petabytes of data)
- Want to use thousands of CPU cores on clusters of commodity servers
- Want seamless scalability: scale "out", not "up"
  - Small cluster of high-end servers vs large cluster of commodity PCs
  - Communication latency: 100ns vs 100$\mu$s
- Move processing to the data
- Process data sequentially, avoid random access

  disk seeks are expensive but disk throughput is high
- Don't want the hassle of managing things

  such as, distributed computing, fault tolerance, recovery, etc

# Motivation (cont.)

- The power of scaling out:
  - 1 HDD: 80 MB/sec
  - 1000 HDDs: 80 GB/sec
- The Web: 20 billion web pages x 20KB $\approx$ 400 TB
  - 1 HDD: 2 months to read the web
  - 1000 HDDs: 1.5 hours to read the web
- Distributed filesystems (DFS) are necessary
- Failures:
  - One server may stay up 3 years (1,000 days)
  - For 1,000 servers: 1 fail per day
  - Google had  1M machines in 2011: 1,000 servers fail per day!

# Motivation (cont.)

- Sharing a global state is very hard: synchronization, deadlocks
- Need a shared nothing architecture
  - independent nodes
  - no common state
  - communicate through network and DFS only
- HPC: shared-memory approach (eg, MPI)
  - a programmer needs to take care of many details: synchronization, concurrency, deadlocks, resource allocation, ...
  - distinction between processing nodes and storage nodes
- Map-Reduce: *data locality* - processing and storage nodes are colocated
- Organize computation for sequential reads (full scans), not random access
- Failures are very common due to the scale

# Map-Reduce

- Map-Reduce provides:
  - Batch processing
  - Automatic parallelization and distribution
  - Full scans of datasets stored in DFS
    - Huge files (100s of GB to TB)
    - Data is rarely updated in place
    - Reads and appends are common
  - Fault tolerance
  - Monitoring and status updates
- A programming model inspired by functional programming languages
- Many data parallel problems can be phrased this way

  "embarrassingly" parallel problems
- Designed for large-scale data processing
- Makes easy to distribute computations across nodes
- Designed to run on clusters of commodity hardware
- Nice retry/failure semantic

# Map and Reduce in Functional Programming (Haskell)

- Map is **map** f s: apply the function f to every element of the list s
  types: s: $[\alpha]$, f: $\alpha \rightarrow \beta$, **map** f s: $[\beta]$

  **map** (\x −> x+1) [1,2,3]
  = [2,3,4]

- **concatMap** f s: apply the function f to every element of the list s and concatenate the results
  types: s: $[\alpha]$, f: $\alpha \rightarrow [\beta]$, **concatMap** f s: $[\beta]$

  **concatMap**(\x −> **if** (x>1) **then** [x,x∗2] **else** [ ]) [1,2,3]
  = [2,4,3,6]

- Reduce is **foldr** acc zero s: accumulate the elements in s using acc
  if s = [x1,x2,x3,...] then it returns acc(x1,acc(x2,acc(x3,...zero)))
  types: s: $[\alpha]$, acc: $\alpha \rightarrow \beta \rightarrow \beta$, zero: $\beta$, **foldr** acc zero s: $[\beta]$

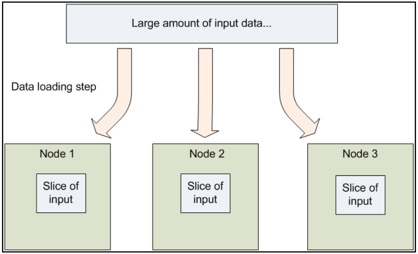
  **foldr**(+) 0 [1,2,3,4]
  = 10

# Apache Hadoop

- An open-source project for reliable, scalable, distributed computing, developed by Apache: `http://hadoop.apache.org/`
- Hadoop includes these subprojects:
  - Hadoop Common: The common utilities that support the other Hadoop subprojects.
  - HDFS: A distributed file system that provides high throughput access to application data.
  - Map-Reduce: A software framework for distributed processing of large data sets on compute clusters.
  - Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying.
  - Pig: A high-level data-flow language and execution framework for parallel computation.

# The Hadoop Distributed File System (HDFS)

- In a Hadoop cluster, data is distributed to all the nodes of the cluster as it is being loaded in
- The Hadoop Distributed File System (HDFS) splits large data files into chunks which are managed by different nodes in the cluster
  - each chunk is replicated across several machines
  - data is conceptually record-oriented
  - which data operated on by a compute node is chosen based on its locality to the node
  - moving computation to the data, instead of moving the data to the computation



# HDFS Architecture

- It is based on the Google File System (GFS)
- HDFS is a block-structured file system
  - individual files are broken into blocks of a fixed size (default is 128MB)
  - eg, 1GB file is 8 blocks, 128MB each
- A file in HDFS smaller than a single block does not occupy a full block
- Blocks are stored across a cluster of one or more machines
- Individual machines in the cluster are referred to as **DataNodes**
- A file can be made of several blocks, and they are not necessarily stored on the same machine
  - the target machines which hold each block are chosen randomly on a block-by-block basis
  - files do not become unavailable by the loss of any data node
  - each block is replicated across a number of machines (3, by default)
- Metadata are stored reliably and synchronized by a single machine, called the **NameNode**
  - keeps metadata in memory
  - there is also a **SecondaryNameNode**, which merges editlogs with the NameNode snapshot

# HDFS Architecture (cont.)

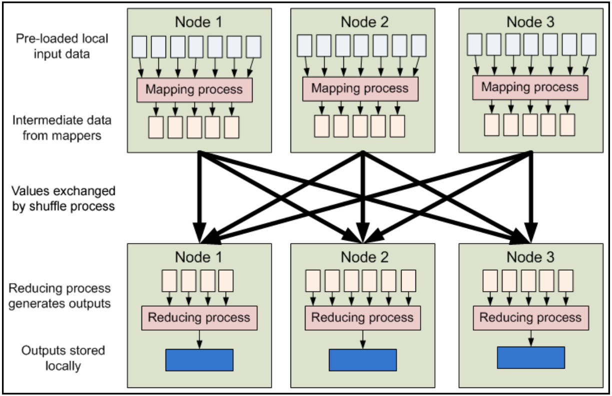
- The NameNode is only used to get block location, not data blocks
- For each requested block, the NameNode returns a set of DataNodes holding a copy of the block
- DataNodes are sorted according to their proximity to the client (nearest replica)
- For Map-Reduce clients, the NameNode returns the DataNode that is colocated with the TaskTracker (same node)
- ... if not possible, it returns the closest node (same rack)
- For block write, it uses a default placement:
  - First copy on the same node as the client
  - Second replica is off-rack
  - Third replica is on the same rack as the second but on a different node
  - Additional replicas are randomly placed
  - This can be customized
- Block contents may not be visible after a write is finished
- sync() forces synchronization

# HDFS Architecture (cont.)

- Writing blocks is done using pipelining:
  - The client retrieves a list of DataNodes to place replicas of a block
  - The client writes the block to the first DataNode
  - The first DataNode forwards the data to the next DataNode in the pipeline, etc
- Data Correctness:
  - DataNodes store checksums for each block (1% overhead)
  - A client uses checksums to validate data
  - If validation fails, the client tries other replicas
- Rebalancer:
  - Makes sure that every DataNode has about the same number of blocks
  - Usually run when new DataNodes are added
  - The cluster is kept online when Rebalancer is active
  - The Rebalancer is throttled to avoid network congestion
  - Can also be called manually using a command
- Note: the NameNode is the single point of failure

# Map-Reduce: Isolated Computation Tasks

- Records are processed in isolation by tasks called **Mappers**
- The output from the Mappers is then brought together into a second set of tasks called **Reducers**, where results from different mappers can be merged together



# Map-Reduce Data

- Key-value pairs are the basic data structure in Map-Reduce
  - Keys and values can be integers, float, strings, or any arbitrary data structures
  - Keys may uniquely identify a record or may be completely ignored
  - Keys can be combined in complex ways to design various algorithms
- A Map-Reduce job consists of:
  - An input dataset stored on HDFS
  - The mapper is applied to every input key-value pair to generate intermediate key-value pairs
  - The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs
- The output dataset is also stored on HDFS
  - The output may consist of a number of distinct files, equal to the number of reducers
- Intermediate data between the mapper and reducer are not stored on HDFS
  - They are spilled to the local disk of each compute node

# Serialization

- Need to be able to transform any object into a byte stream
  - to transmit data over the network (RPC)
  - to store data on HDFS
- Hadoop uses its own serialization interface
- Values must implement Writable

```
class MyClass implements Writable {
    public void write ( DataOutput out ) throws IOException { ... }
    public void readFields ( DataInput in ) throws IOException { ... }
}
```

- Provides a default read:

```
public static MyClass read ( DataInput in ) throws IOException {
    MyClass w = new MyClass();
    w.readFields(in);
    return w;
}
```

# Serialization (cont.)

- Keys must implement WritableComparable
- It is a Writable with the additional method:

  **public int** compareTo ( MyClass o ) { ... }

- Why? because must be able to sort data by the key
- Hadoop provides a number of WritableComparable classes: IntWritable, DoubleWritable, Text, etc

  Text is a Writable String

- If you want to use a custom serializer, you may specify your own implementation of org.apache.hadoop.io.serializer.Serialization and set io.serialization in the Hadoop configuration

# Serialization Example

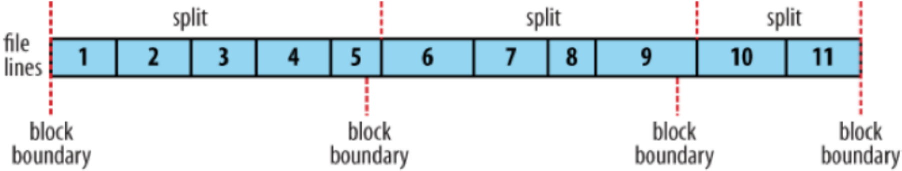
```
class Employee implements Writable {
    public String name;
    public int dno;
    public String address;

    public void write ( DataOutput out ) throws IOException {
        out.writeInt(dno);
        out.writeUTF(name);
        out.writeUTF(address);
    }

    public void readFields ( DataInput in ) throws IOException {
        dno = in.readInt();
        name = in.readUTF();
        address = in.readUTF();
    }
}
```

# InputFormat

- Describes the input-specification for a Map-Reduce job
- The Map-Reduce framework relies on the InputFormat of the job to:
  - Validate the input-specification of the job
  - Split-up the input files into InputSplits
  - Provide the **RecordReader** implementation to be used to get input records from the InputSplit for processing by the mapper
  - upper bound for input splits = HDFS blocksize
  - no lower bound, but can be set by programmer: mapred.min.split.size



# InputFormat (cont.)

- **FileInputFormat** provides a generic implementation of getSplits
- **TextInputFormat** for plain text files
  - Files are broken into lines
  - Either linefeed or carriage-return are used to signal end of line
  - Keys are the position in the file and values are the line of text
- **SequenceFileInputFormat**
  - Provides a persistent data structure for binary key-value pairs
  - Also works well as containers for smaller files
  - It comes with the sync() method to introduce sync points to help managing InputSplits for MapReduce
- Custom InputFormat implementations may override split size

# OutputFormat

- For writing the reducer results to the output (HDFS)
- Analogous to InputFormat
  - TextOutputFormat writes "key value", followed by a newline, to the output file
  - SequenceFileOutputFormat uses a binary format to pack key-value pairs
  - NullOutputFormat discards output

# The Map-Reduce Algorithm

- Input: a set of key-value pairs
- A programmer needs to specify two methods:
  1. A mapper $\boxed{map: \ (k, v) \ \rightarrow \ list(< k', v' >)}$
     - For each a key-value pair $< k, v >$, it returns a sequence of key-value pairs $< k', v' >$
     - Can be executed in parallel for each pair
  2. A reducer $\boxed{reduce: \ (k', list(v')) \ \rightarrow \ list(< k'', v'' >)}$
     - All values $v'$ with the same key $k'$ are reduced together
     - There is one reduce function call per unique key $k'$
     - Can be executed in parallel for each distinct key
- Output: a set of key-value pairs

# The Map-Reduce Algorithm (cont.)

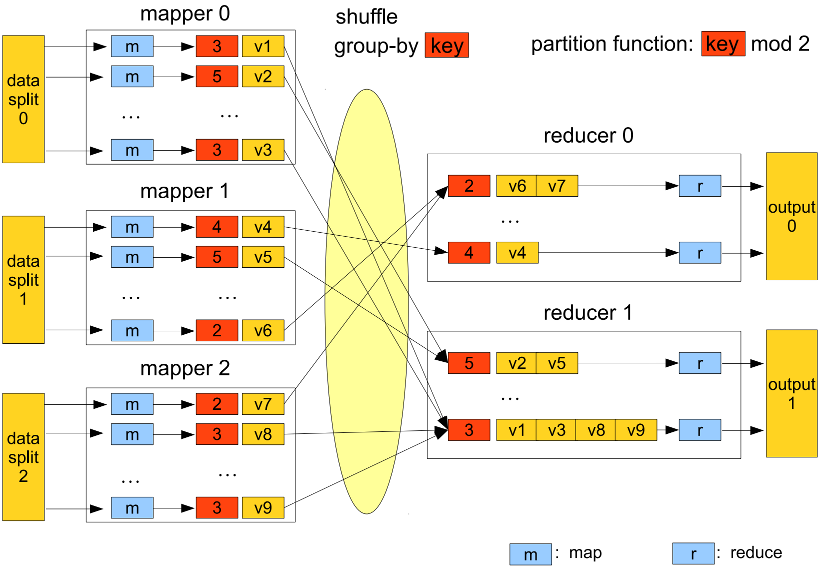
- Shuffling:
  - implicit stage between map and reduce
  - groups the map results by key $k'$
  - can be controlled by the programmer implicitly

    by specifying custom partitioning, grouping, and sorting functions
- The input is split into a number of **input splits** (block-sized chunks)
  - One mapper for each input split
  - An input dataset may be an HDFS directory with multiple files
  - A file may be split into one or more input splits

    last split may be smaller than a block
  - An input split can be from one file only
  - A small number of large files is better than a large number of small files
- How many map and reduce tasks?
  - Hadoop takes care of the map tasks automatically:
    - one task for each input split
    - if the number of splits is more than the available workers

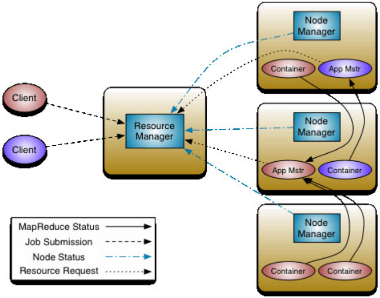
      $\Rightarrow$ a worker gets more than one splits
  - The programmer can specify the number of reducers using the job.setNumReduceTasks method (default is 1)

# An Example with 3 Mappers and 2 Reducers



# Resource Allocation: Hadoop Yarn

- The master is called the **ResourceManager**
  It has two main components:
  1. The **Scheduler** is responsible for allocating resources to the various running applications
  2. The **ApplicationManager** accepts job-submissions and provides the service for restarting the ApplicationMaster container on failure
- Each worker (compute node) is called a **NodeManager**
- Old Hadoop (hadoop-1.*) used JobTracker and TaskTrackers



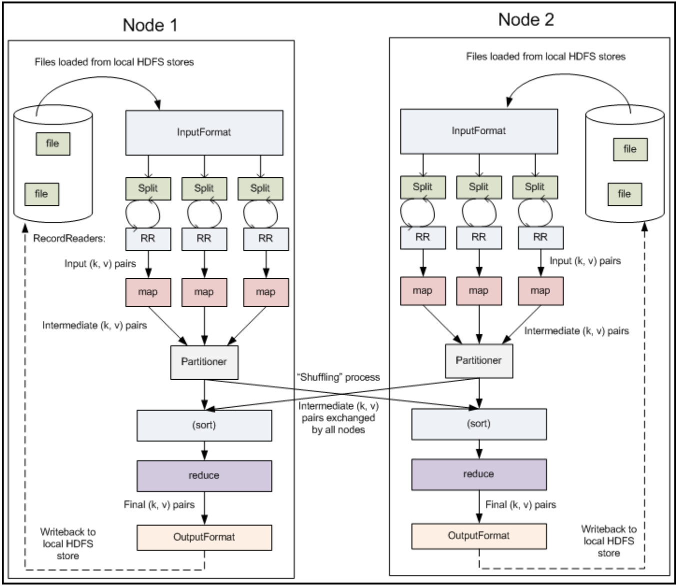
# Fault Recovery

- Workers are pinged by master periodically
  - Non-responsive workers are marked as failed
  - All tasks in-progress or completed by failed worker become eligible for rescheduling
- The master could periodically checkpoint
  - Current implementations abort on master failure

# Stages of a Map-Reduce Program

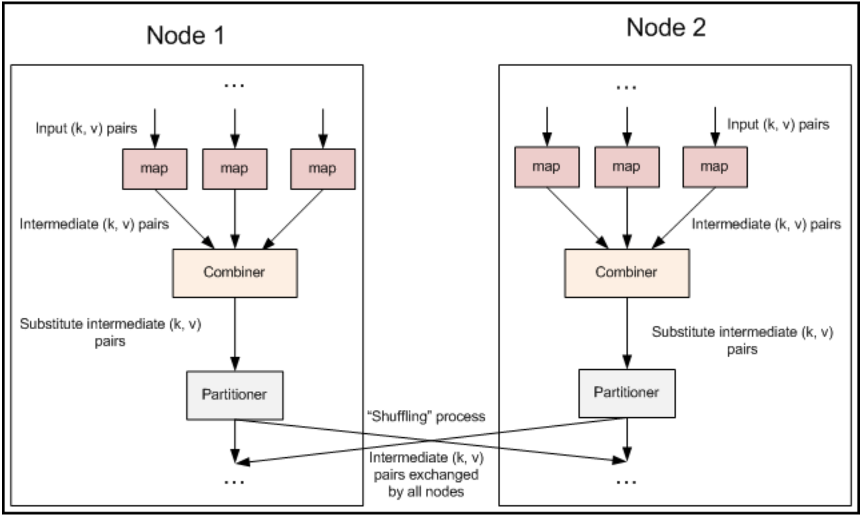
- **InputFormat:** describes how input files are split up and read
  - Selects the files or other objects that should be used for input
  - Defines the InputSplits that break a file into tasks
  - Provides a factory for RecordReader objects that read the file
- **Mapper:** Given a key and a value, the map() method emits (key, value) pair(s) which are forwarded to the Reducers
- **Shuffle:** moving map outputs to the reducers
- Each reducer is associated with a different key space (a partition)
  - all values for the same key are sent to the same partition
  - default partitioner: key-hash-value % number-of-reducers
- **Sort:** The set of intermediate keys in a partition is sorted before they are presented to the Reducer
- **Reduce:** For each different key in the partition assigned to a Reducer, the Reducer's reduce() method is called once
- **OutputFormat:** The (key, value) pairs are written to output files

# A Closer Look



# The Optional Combiner

- Local aggregation after mapping before shuffling
- Possible performance gains: reduces amount of shuffling



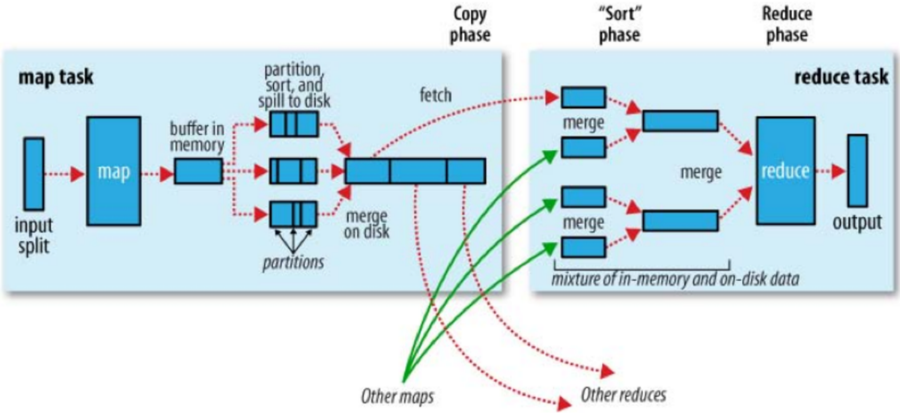
# DataFlow of Map-Reduce

- An InputFormat that maps an HDFS dataset to a sequence of $< k, v >$ pairs
- A mapper $\quad map(k, v) \;\rightarrow\; list(< k', v' >)$
- An optional combiner $\quad combine(k', list(v')) \;\rightarrow\; list(< k', v' >)$
- A reducer $\quad reduce(k', list(v')) \;\rightarrow\; list(< k'', v'' >)$
- An OutputFormat that dumps $< k'', v'' >$ pairs to HDFS

# Shuffling Implementation

- Let say that there are $m$ mappers and $n$ reducers
- At the mapper side:
  - Each mapper creates $n$ partitions
  - Each $< k', v' >$ pair produced by the mapper goes to partitioner$(k')\,\%\,n$
  - Each partition is sorted locally using the sorting function
  - If there is a combiner function, each partition is reduced by combining consecutive pairs with the same key
- At the reducer side:
  - Each reducer fetches one partition from each of the $m$ mappers
  - These $m$ partitions are merged in stages
  - The reducer scans the merged data: consecutive pairs with the same key belong to the same group
- Number of copying operations: $m \times n$

# DataFlow of Map-Reduce (from Tom White: Hadoop the Definitive Guide)



# The org.apache.hadoop.mapreduce.Mapper Class

- Need to define the map method:

  **public void** map ( KEYIN key, VALUEIN value, Context context )
  **throws** IOException, InterruptedException { ... }

- Called once for each key/value pair in the input split
- Use context.write(k,v) to write the (k,v) pair to the map output
- Optionally, overwrite these methods:
  - Called once at the beginning of the task:

    **public void** setup ( Context context )

  - Called once at the end of the task:

    **public void** cleanup ( Context context )

# The org.apache.hadoop.mapreduce.Reducer Class

- Need to define the reduce method:

```
public void reduce ( KEYIN key, Iterable<VALUEIN> values,
                     Context context )
    throws IOException, InterruptedException { ... }
```

- The Iterable values contain all values associated with the same key
- When you access values, you get the *same object* but with different values (Hadoop uses readFields to get the next value)
- ... so this is wrong:

```
Vector<VALUEIN> v = new Vector<VALUEIN>();
for (VALUEIN a: values)
    v.add(a);
```

- You can also overide setup and cleanup

# org.apache.hadoop.mapreduce.Job Class

- The job submitter's view of the Job
- It allows the user to configure the job, submit it, control its execution, and query the state
- Normally the user creates the application, describes various facets of the job via Job and then submits the job and monitor its progress
- Here is an example on how to submit a job:

```java
// Create a new Job
Job job = Job.getInstance();
job.setJarByClass(MyJob.class);
// Specify various job-specific parameters
job.setJobName("myjob");
job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));
job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);
// Submit the job, then poll for progress until the job is complete
job.waitForCompletion(true);
```

# A Simple Map-Reduce Example

- We have a CSV file with int-double pairs:

```
1, 2.3
2, 3.4
1, 4.5
3, 6.6
2, 3.0
```

- We want to group data by the first column and, for each group, we want to calculate the average value of the second column:

```
select s.X, avg(s.Y)
from csv_file as s
group by s.X
```

# The Pseudo-Code for the Simple Map-Reduce Example

Assuming that you use TextInputFormat to read lines

```
map( loc, line ):
    parse the line into a long key and a double value
    emit(key,value)

reduce( key, values ):
    sum = 0.0
    count = 0
    for each v in values:
        count++
        sum += v
    emit(key,sum/count)
```

# Simple Map-Reduce: Mapper

```
class MyMapper extends Mapper<Object,Text,IntWritable,DoubleWritable> {
    public void map ( Object key, Text value, Context context )
                throws IOException, InterruptedException {
        Scanner s = new Scanner(value.toString()).useDelimiter(",");
        int x = s.nextInt();
        double y = s.nextDouble();
        context.write(new IntWritable(x),new DoubleWritable(y));
        s.close();
    }
}
```

# Simple Map-Reduce: Reducer

```
class MyReducer extends Reducer<IntWritable,DoubleWritable,
                                 IntWritable,DoubleWritable> {
  public void reduce ( IntWritable key, Iterable<DoubleWritable> values,
                       Context context )
           throws IOException, InterruptedException {
    double sum = 0.0;
    long count = 0;
    for (DoubleWritable v: values) {
      sum += v.get();
      count++;
    };
    context.write(key,new DoubleWritable(sum/count));
  }
}
```

# Simple Map-Reduce: Main Program

```
package edu.uta.cse6331;
public class Simple extends Configured implements Tool {
...
    @Override
    public int run ( String[] args ) throws Exception {
        Configuration conf = getConf();
        Job job = Job.getInstance(conf);
        job.setJobName("MyJob");
        job.setJarByClass(Simple.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(DoubleWritable.class);
        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(DoubleWritable.class);
        job.setMapperClass(MyMapper.class);
        job.setReducerClass(MyReducer.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.setInputPaths(job,new Path(args[0]));
        FileOutputFormat.setOutputPath(job,new Path(args[1]));
        return job.waitForCompletion(true) ? 0 : 1;
    }
    public static void main ( String[] args ) throws Exception {
        ToolRunner.run(new Configuration(),new Simple(),args);
    } }
```

# Simple Map-Reduce: Run in Standalone (Local) Mode

Source is available at:
`http://lambda.uta.edu/cse6331/tests/Simple.java`
input.txt:

```
1,2.3
2,3.4
1,4.5
3,6.6
2,3.0
```

Linux shell commands:

```
javac -d classes -cp classes:`hadoop classpath` Simple.java
jar cf simple.jar -C classes .
hadoop jar simple.jar edu.uta.cse6331.Simple input.txt output
cat output/part-r-00000
```