Hadoop/MapReduce

Object-oriented framework presentation
CSCI 5448
Casey McTaggart

What is Apache Hadoop?

- Large scale, open source software framework
  - Yahoo! has been the largest contributor to date
- Dedicated to scalable, distributed, data-intensive computing
- Handles thousands of nodes and petabytes of data
- Supports applications under a free license
- 3 Hadoop subprojects:
  - Hadoop Common: common utilities package
  - HFDS: Hadoop Distributed File System with high throughput access to application data
  - MapReduce: A software framework for distributed processing of large data sets on computer clusters

Hadoop MapReduce

- MapReduce is a programming model and software framework first developed by Google (Google's MapReduce paper submitted in 2004)
- Intended to facilitate and simplify the processing of vast amounts of data in parallel on large clusters of commodity hardware in a reliable, fault-tolerant manner
  - Petabytes of data
  - Thousands of nodes
- Computational processing occurs on both:
  - Unstructured data : filesystem
  - Structured data : database

Hadoop Distributed File System (HFDS)

- Inspired by Google File System
- Scalable, distributed, portable filesystem written in Java for Hadoop framework
  - Primary distributed storage used by Hadoop applications
- HFDS can be part of a Hadoop cluster or can be a stand-alone general purpose distributed file system
- An HFDS cluster primarily consists of
  - NameNode that manages file system metadata
  - DataNode that stores actual data
- Stores very large files in blocks across machines in a large cluster
  - Reliability and fault tolerance ensured by replicating data across multiple hosts
- Has data awareness between nodes
- Designed to be deployed on low-cost hardware

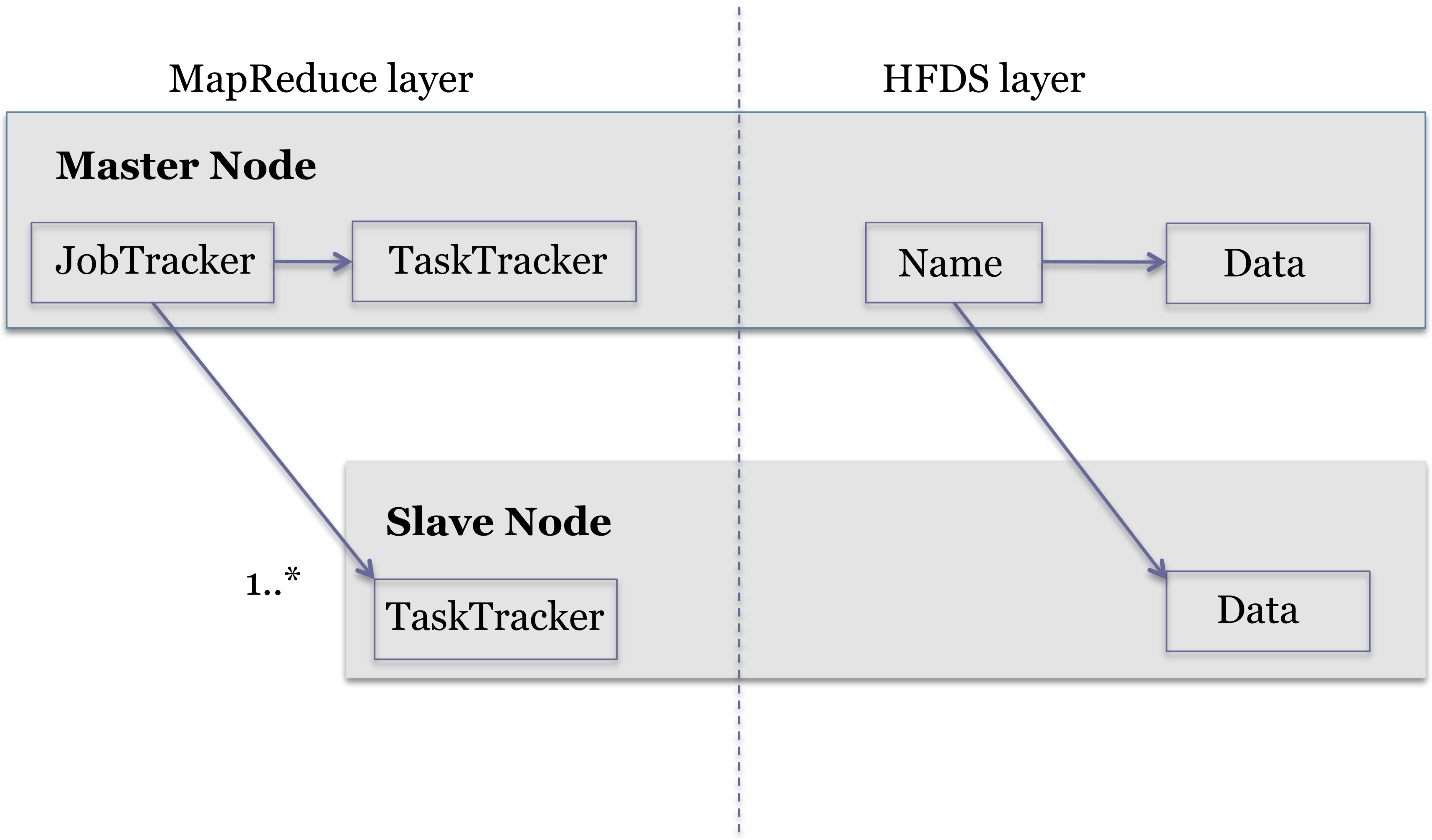
More on Hadoop file systems

- Hadoop can work directly with any distributed file system which can be mounted by the underlying OS
- However, doing this means a loss of locality as Hadoop needs to know which servers are closest to the data
- Hadoop-specific file systems like HFDS are developed for locality, speed, fault tolerance, integration with Hadoop, and reliability

Typical Hadoop cluster integrates MapReduce and HFDS

- Master/slave architecture
- Master node contains
  - Job tracker node (MapReduce layer)
  - Task tracker node (MapReduce layer)
  - Name node (HFDS layer)
  - Data node (HFDS layer)
- Multiple slave nodes contain
  - Task tracker node (MapReduce layer)
  - Data node (HFDS layer)
- MapReduce layer has job and task tracker nodes
- HFDS layer has name and data nodes

Hadoop simple cluster graphic



MapReduce framework

- Per cluster node:
  - Single JobTracker per master
    - Responsible for scheduling the jobs' component tasks on the slaves
    - Monitors slave progress
    - Re-executing failed tasks
  - Single TaskTracker per slave
    - Execute the tasks as directed by the master

MapReduce core functionality

- Code usually written in Java- though it can be written in other languages with the Hadoop Streaming API
- Two fundamental pieces:
  - Map step
    - Master node takes large problem input and slices it into smaller sub problems; distributes these to worker nodes.
    - Worker node may do this again; leads to a multi-level tree structure
    - Worker processes smaller problem and hands back to master
  - Reduce step
    - Master node takes the answers to the sub problems and combines them in a predefined way to get the output/answer to original problem

MapReduce core functionality (II)

- Data flow beyond the two key pieces (map and reduce):
  - Input reader – divides input into appropriate size splits which get assigned to a Map function
  - Map function – maps file data to smaller, intermediate <key, value> pairs
  - Partition function – finds the correct reducer: given the key and number of reducers, returns the desired Reduce node
  - Compare function – input for Reduce is pulled from the Map intermediate output and sorted according to ths compare function
  - Reduce function – takes intermediate values and reduces to a smaller solution handed back to the framework
  - Output writer – writes file output

MapReduce core functionality (III)

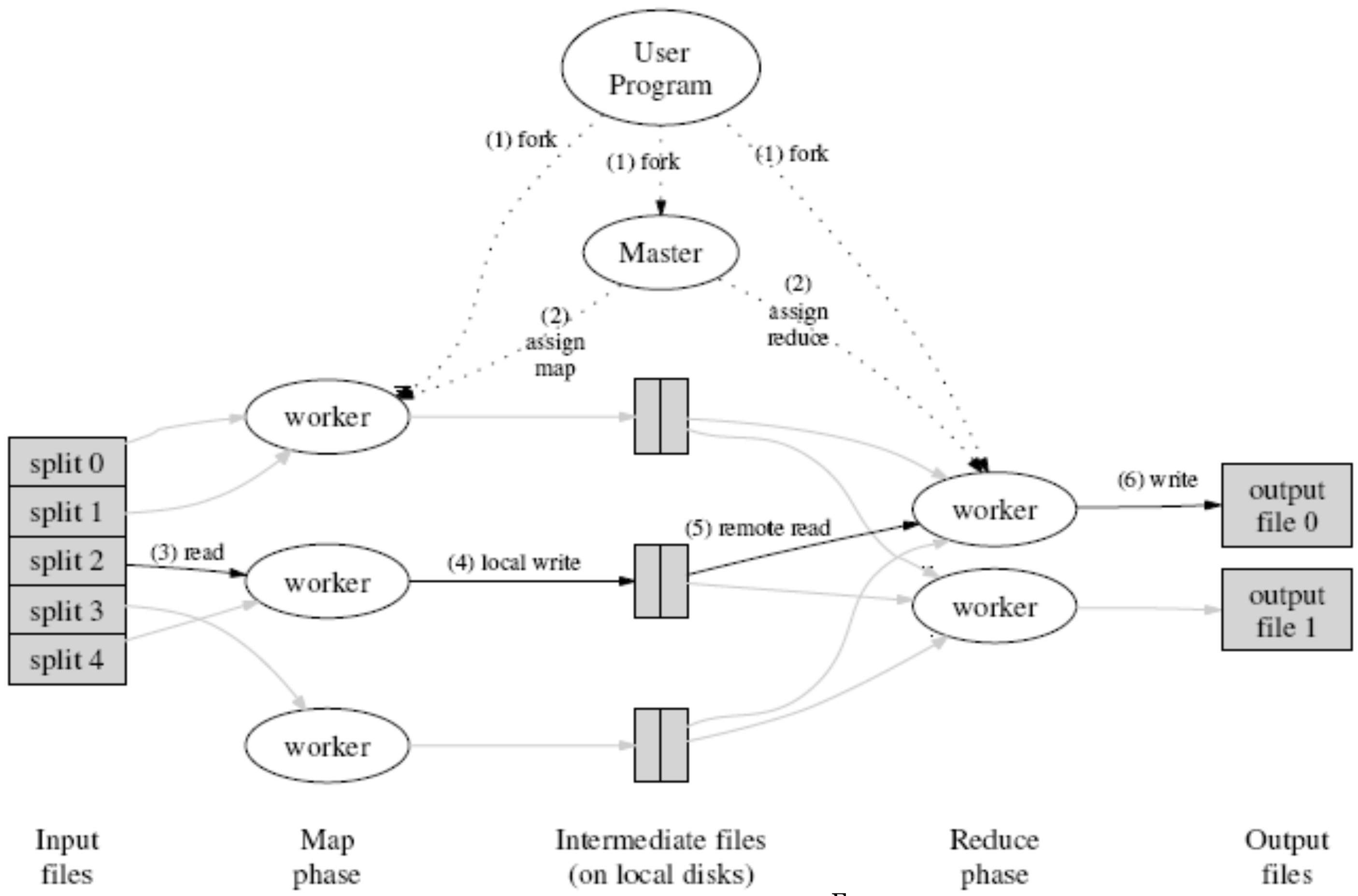
- A MapReduce *Job* controls the execution
  - Splits the input dataset into independent chunks
  - Processed by the map tasks in parallel
- The framework sorts the outputs of the maps
- A MapReduce *Task* is sent the output of the framework to reduce and combine
- Both the input and output of the job are stored in a filesystem
- Framework handles scheduling
  - Monitors and re-executes failed tasks

MapReduce input and output

- MapReduce operates exclusively on `<key, value>` pairs
- Job Input: `<key, value>` pairs
- Job Output: `<key, value>` pairs
  - Conceivably of different types
- Key and value classes have to be serializable by the framework.
  - Default serialization requires keys and values to implement Writable
  - Key classes must facilitate sorting by the framework

Input and Output (II)

$$\begin{tikzcd}
\text{Input} & & & \text{Output} \\
\boxed{\texttt{<k1, v1>}} \arrow[r, "\textbf{map}"] & \boxed{\texttt{<k2, v2>}} \arrow[r, "\textbf{combine*}"] & \boxed{\texttt{<k2, v2>}} \arrow[r, "\textbf{reduce}"] & \boxed{\texttt{<k3, v3>}}
\end{tikzcd}$$



From
http://code.google.com/edu/parallel/mapreduce-tutorial.html

To explain in detail, we'll use a code example: WordCount
Count occurrences of each word across different files

Two input files:

file1: "hello world hello moon"

file2: "goodbye world goodnight moon"

Three operations:

map

combine

reduce

What is the output per step?

MAP

First map:
```
< hello, 1 >
< world, 1 >
< hello, 1 >
< moon, 1 >
```

Second map:
```
< goodbye, 1 >
< world, 1 >
< goodnight, 1 >
< moon, 1 >
```

COMBINE

First map:
```
< moon, 1 >
< world, 1 >
< hello, 2 >
```

Second map:
```
< goodbye, 1 >
< world, 1 >
< goodnight, 1 >
< moon, 1 >
```

REDUCE
```
< goodbye, 1 >
< goodnight, 1 >
< moon, 2 >
< world, 2 >
< hello, 2 >
```

Main run method – the engine

```
public int run(String[] args) {
   Job job = new Job(getConf());
   job.setJarByClass(WordCount.class);
   job.setJobName(“wordcount”);

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   job.setMapperClass(Map.class);
   job.setCombinerClass(Reduce.class);
   job.setReducerClass(Reduce.class);

   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);

   FileInputFormat.setInputPaths(job, new Path(args[0]));
   FileOutputFormat.setOutputPaths(job, new Path(args[1]));
   boolean success = job.waitForCompletion(true);
   return success ? 0 : 1;
}
```

Main run method: pseudocode

```
public int run(String[] args) {
   - Create a new job with the given configuration

   - Set Job Output <key.class, value.class> as
      <Text, IntWritable>

   - Set Job Input <key.class, value.class> as
      <TextInputFormat, TextOutputFormat>

   - Tell Job to use our Map as Mapper class
   - Tell Job to use our Reduce as Combiner class
   - Tell Job to use our Reduce as Reducer class

   - Set file input paths
   - Set file output paths in the Job
   - Wait until Job is done
   - Return success if successful
}
```

Job details

- Job sets the overall MapReduce job configuration
- Job is specified client-side
- Primary interface for a user to describe a MapReduce job to the Hadoop framework for execution
- Used to specify
  - Mapper
  - Combiner (if any)
  - Partitioner (to partition key space)
  - Reducer
  - InputFormat
  - OutputFormat
  - Many user options; high customizability

Job details (II)

- Jobs can be monitored by users
- Users can chain MapReduce jobs together to accomplish complex tasks which cannot be done with a single MapReduce job
  - make use of Job.waitForCompletion()
  - and Job.submit()

Map class for WordCount

```
public static class Map extends Mapper {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context
  context) {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
      }
  }
}
```

Map class implements a public map method, that processes one line at a time and splits each line into tokens separated by whitespaces. It emits a key-value pair of `< <word>, 1>`, written to the Context.

Map class (II)

Remember our input files:

file1: “hello world hello moon”

file2: “goodbye world goodnight moon”

Two maps are generated (1 per file)

First map emits:

```
< hello, 1 >
< world, 1 >
< hello, 1 >
< moon, 1 >
```

Second map emits:

```
< goodbye, 1 >
< world, 1 >
< goodnight, 1 >
< moon, 1 >
```

Mapper

- Mapper maps input key/value pairs to a set of intermediate key/value pairs
- Implementing classes extend Mapper and override map()
  - Main Mapper engine: Mapper.run()
    - setup()
    - map() for each input record
    - cleanup()
- Mapper implementations are specified in the Job
- Mapper instantiated in the Job
- Output data is emitted from Mapper via the Context object
- Hadoop MapReduce framework spawns one map task for each logical representation of a unit of input work for a map task
  - E.g. a filename and a byte range within that file

How many maps?

- The number of maps is driven by the total size of the inputs
- Hadooop has found the right level of parallelism for maps is between 10-100 maps/node
- If you expect 10TB of input data and have a block size of 128MB, you will have 82,000 maps
- Number of tasks controlled by number of splits returned and can be user overridden

Context object details

Recall Mapper code:

```
while (tokenizer.hasMoreTokens()) {
    word.set(tokenizer.nextToken());
    context.write(word, one);
}
```

- Context object: allows the Mapper to interact with the rest of the Hadoop system
- Includes configuration data for the job as well as interfaces which allow it to emit output
- Applications can use the Context
  - to report progress
  - to set application-level status messages
  - update Counters
  - indicate they are alive

Combiner class

- Specifies how to combine the maps for local aggregation
- In this example, it is the same as the Reduce class
- Output after running combiner:

First map:

```
< moon, 1 >
< world, 1 >
< hello, 2 >
```

Second map:

```
< goodbye, 1 >
< world, 1 >
< goodnight, 1 >
< moon, 1 >
```

Details on Combiner class and intermediate outputs

- Framework groups all intermediate values associated with a given output key
- Passed to the Reducer class to get final output
- User-specified Comparator can be used to control grouping
- Combiner class can be user specified to perform local aggregation of the intermediate outputs
- Intermediate, sorted outputs always stored in a simple format
  - Applications can control if (and how) intermediate outputs are to be compressed (and the CompressionCode) in the Job

Reduce class for WordCount

```
public static class Reduce extends Reducer {
  public void reduce(Text key, Iterable<IntWritable>
  values, Context  context) {

      int sum = 0;
      for (IntWritable val : values) {
         sum += val.get();
      }
      context.write(key, new IntWritable(sum));
  }
}
```

The framework puts together all the pairs with the same key and feeds them to the reduce function, that then sums the values to give occurrence counts.

Reduce class (II)

Recall the output of the job: a count of occurrences.

```
< goodbye, 1 >
< goodnight, 1 >
< moon, 2 >
< world, 2 >
< hello, 2 >
```

Reducer (III)

- Reduces a set of intermediate values which share a key to a (usually smaller) set of values
- Sorts and partitions Mapper outputs
- Number of reduces for the job set by user via Job.setNumReduceTasks(int)
- Reduce engine
  - receives a Context containing job's configuration as well as interfacing methods that return data back to the framework
  - Reducer.run()
    - setup()
    - reduce() per key associated with reduce task
    - cleanup()

Reducer (IV)

- Reducer.reduce()
  - Called once per key
  - Passed in an Iterable which returns all values associated with that key
  - Emits output with Context.write()
  - Output is **not** sorted.
  - 3 primary phases
    - Shuffle: the framework fetches relevant partitions of the output of all mappers via HTTP
    - Sort: framework groups Reducer inputs by keys
    - Reduce: reduce called on each <key, (value list) >

How many reduces?

- 0.95 or 1.75 multiplied by (numberOfNodes * mapreduce.tasktracker.reduce.tasks.maximum
- 0.95 : all of the reduces can launch immediately and start transferring map outputs as the maps finish
- 1.75: the faster nodes will finish their first round of reduces and launch a second wave of reduces, doing a better job of load balancing
- Increasing number of reduces increases framework overhead; and increases load balancing and lowers cost of failures

Task Execution and Environment

- TaskTracker executes Mapper/Reducer task as a child process in a separate jvm
- Child task inherits the environment of the parent TaskTracker
- User can specify environmental variables controlling memory, parallel computation settings, segment size, and more

Scheduling

- By default, Hadoop uses FIFO to schedule jobs. Alternate scheduler options: capacity and fair
- Capacity scheduler
    - Developed by Yahoo
    - Jobs are submitted to queues
    - Jobs can be prioritized
    - Queues are allocated a fraction of the total resource capacity
    - Free resources are allocated to queues beyond their total capacity
    - No preemption once a job is running

- Fair scheduler
  - Developed by Facebook
  - Provides fast response times for small jobs
  - Jobs are grouped into Pools
  - Each pool assigned a guaranteed minimum share
  - Excess capacity split between jobs
  - By default, jobs that are uncategorized go into a default pool. Pools have to specify the minimum number of map slots, reduce slots, and a limit on the number of running jobs

Requirements of applications using MapReduce

- Specify the Job configuration
  - Specify input/output locations
  - Supply *map* and *reduce* functions via implementations of appropriate interfaces and/or abstract classes
- Job client then submits the job (jar/executables etc) and the configuration to the JobTracker

What about bad input?

- Hadoop provides an option to skip bad records:
  - SkipBadRecords class
- Used when map tasks crash deterministically on certain input
  - Usually a result of bugs in the map function
  - May be in 3rd party libraries
  - Tasks never complete successfully even after multiple attempts
- Framework goes into ‘skipping mode’ after a certain number of map failures
- Number of records skipped depends on how frequently the processed record counter is incremented by the application

What are Hadoop/MapReduce limitations?

- Cannot control the order in which the maps or reductions are run
- For maximum parallelism, you need Maps and Reduces to not depend on data generated in the same MapReduce job (i.e. stateless)
- A database with an index will always be faster than a MapReduce job on unindexed data
- Reduce operations do not take place until all Maps are complete (or have failed then been skipped)
- General assumption that the output of Reduce is smaller than the input to Map; large datasource used to generate smaller final values

Who’s using it?

- Lots of companies!
    - Yahoo!, AOL, eBay, Facebook, IBM, Last.fm, LinkedIn, The New York Times, Ning, Twitter, and more
- In 2007 IBM and Google announced an initiative to use Hadoop to support university courses in distributed computer programming
- In 2008 this collaboration and the Academic Cloud Computing Initiative were funded by the NSF and produced the Cluster Exploratory Program (CLuE)

Summary and Conclusion

- Hadoop MapReduce is a large scale, open source software framework dedicated to scalable, distributed, data-intensive computing
- The framework breaks up large data into smaller parallelizable chunks and handles scheduling
    - Maps each piece to an intermediate value
    - Reduces intermediate values to a solution
    - User-specified partition and combiner options
- Fault tolerant, reliable, and supports thousands of nodes and petabytes of data
- If you can rewrite algorithms into Maps and Reduces, and your problem can be broken up into small pieces solvable in parallel, then Hadoop's MapReduce is the way to go for a distributed problem solving approach to large datasets
- Tried and tested in production
- Many implementation options