What's this hullabaloo about an elephant?  
No, not the one named Horton  
Who has fun in the Jungle of Nool  
This one's named Hadoop, and is just as cool  
Crunching through data and having fun

Frequently asked questions from the previous class survey
- How does a Map know that it must produce <key, value> pairs?
- Why does a Mapper produce R intermediate outputs?
- Why can't you perform intermediate reductions?
- Can a reducer be located so that it can pull data from a Mapper?
- Do GFS and HDFS perform caching?
- If each reducer produces an output, how are you actually able to combine and produce one output file?

Topics covered in this lecture
- Wrap-up of the MapReduce Paper  
  - Refinements
- Hadoop  
  - Application development  
  - API

Ordering Guarantees
- Intermediate key/pairs are processed in *increasing* key order
- Easy to generate sorted output file

The Combiner function
- There is significant *repetition* in intermediate keys produced by each map task
  - For word-frequencies
    - Each map may produce 100s or 1000s of <the, “1”>
  - All of these counts are sent over the network
  - Combiner: Does *partial merging* of this data
    - Before it is sent to the reducer
Combiner function

- Executed on each machine that performs map task
- Code implementing combiner & reduce function
- Usually the same ... [We will see an example where this is not true.]
- Difference?
  - Combiner: Output written to intermediate file
  - Reduce: Output written to final output

Input/Output Types: Support for reading input data in different formats

- Text mode treats every line as a <key, value> pair
  - Key: Offset in the file
  - Value: Contents of the line
- <key, value> pairs are sorted by key
- Each input type knows how to split itself for
  - Processing as separate map tasks
  - Text mode splitting occurs only at line boundaries

Side-effects

- Besides intermediate files, other auxiliary files may be produced
- Side effects
- No atomic commits for multiple auxiliary files that are produced

Skipping Bad Records [1/3]

- Bugs in user code cause Map or Reduce functions to crash
  - Deterministically: On certain records
  - Fix the bug?
    - Yes, but not always feasible
  - Acceptable to ignore a few records

Skipping Bad Records [2/3]

- Optional mode of operation
  1. Detected records that cause deterministic crashes
  2. Skip them
- Each worker installs a signal handler to catch segmentation violations and bus errors

Skipping Bad Records [3/3]

- Signal handler sends last gasp UDP packet to the Master
  - Contains sequence number
  - When Master sees more than 1 failure at that record
    - Indicates record should be skipped during next execution
Local Execution
- Support for **sequential execution** of MapReduce operation on a single machine
  - Helps with debugging, profiling, and testing
- Controls to limit computation to a particular map
- Invoke programs with a special flag
  - Use debugging and testing tools

Status Information
- Master runs an internal HTTP Server
- Exports pages for viewing
- Show the progress of a computation
  - Number of tasks in progress
  - Number of tasks that completed
  - Bytes of input
  - Bytes of intermediate data
  - Processing rate

Hadoop
- Java-based open-source implementation of MapReduce
- Created by Doug Cutting
- Origins of the name Hadoop
  - Stuffed yellow elephant
- Includes HDFS (Hadoop Distributed File System)

Hadoop timelines
- Feb 2006
  - Apache Hadoop project officially started
  - Adoption of Hadoop by Yahoo! Grid team
- Feb 2008
  - Yahoo! Announced its search index was generated by a 10,000-core Hadoop cluster
- May 2009
  - 17 clusters with 24,000 nodes

Hadoop Releases
- There are four active releases at the moment
  - 2.7.x
  - 2.8.x
  - 3.1.x
  - 3.2.x
- Last release from the 2.7.x branch (v2.7.7) was on May 31, 2018
  - 2.7.x branch is in maintenance mode
- All 3.x.x branches had releases in 2019
Hadoop Evolution

- 0.20.x series became 1.x series
- 0.23.x was forked from 0.20.x to include some major features
- 0.23 series later became 2.x series
- 2.8.0 branched off from 2.7.3
- 2.9.0 branched off from 2.8.2
- 3.0.0 series branched off from 2.7.0
- 3.1.0 series branched off from 3.0.0
- 3.2.0 branched off from 3.1.0

0.23 included several major features

- New MapReduce runtime, called MapReduce 2, implemented on a new system called YARN
  - YARN: Yet Another Resource Negotiator
  - Replaces the "classic" runtime in previous releases
- HDFS federation
  - HDFS namespace can be dispersed across multiple name nodes
- HDFS high-availability
  - Removes name node as a single point of failure; supports standby nodes for failover

3.2.0 includes major features

- Hadoop Submarine support
  - Hadoop Submarine is a new project that orchestrates Tensorflow programs without modifications on Yarn and provide access to data stored on HDFS
  - Support for GPUs and Docker images
- New/Improved storage connectors
  - ADLS (Azure DataLake Generation 2), Amazon S3, and Amazon DynamoDB
- HDFS storage policies
  - Hierarchical storage – Archive, Disk (default), SSD, and RamDisk
  - Users can define the type of storage when storing data
  - Blocks can be moved between different storage types

Latest Release

- February 6, 2019
  - v3.1.2 released [We will use this for HW3]
- January 16, 2019
  - v3.2.0 released
    - This version is considered stable, but not production ready yet.
  - v2.9.2, v2.8.5, and v2.7.7 were released in 2018.

The Hadoop Ecosystem

MapReduce Jobs

- A MapReduce Job is a unit of work
- Consists of:
  - Input Data
  - MapReduce program
  - Configuration information
- Hadoop runs the jobs by dividing it into tasks
  - Map tasks
  - Reduce tasks
Types of nodes that control the job execution process
[Older Versions]
- Job tracker
  - Coordinates all jobs by scheduling tasks to run on task trackers
  - Records overall progress of each job
  - If task fails, reschedule on a different task tracker
- Task tracker
  - Run tasks and reports progress to job tracker

[Newer Versions]
- Resource Manager
- Application Manager
- Node manager

Processing a weather dataset
- The dataset is from NOAA
- Stored using a line-oriented format
  - Each line is a record
- Lots of elements being recorded
  - We focus on temperature
    - Always present with a fixed width

Format of a record in the dataset
- 0057 332130           # USAF weather station identifier
- 99999            # WBAN weather station identifier
- 19500101         # Observation date
- 300 # Observation time
- +51317           # latitude (degrees x 1000)
- +028783          # longitude (degrees x 1000)
- +0171            # elevation (meters)
- 99999
- V020
- 320 # wind direction (degrees)
- 1 # quality code
- 1 # air temperature (degrees Celsius x 10)
- 1 # quality code
- -0128           # air temperature (degrees Celsius x 10)

Analyzing the dataset
- What's the highest recorded temperature for each year in the dataset?
- See how programs are written
  - Using Unix tools
  - Using MapReduce

Using awk
Tool for processing line-oriented data
#!/usr/bin/env bash
for year in all/*
do
echo -ne 'basename $year .gz' "| awk '{
  temp=substr($0, 88, 5) + 0;
  q=substr($0, 93, 1);
  if (temp !=9999 && q ~ /[01459]/ &&
  temp > max) max = temp }
END {print max}'
done
Sample output that is produced

```bash
% ./max_temperature.sh
1901  317
1902  244
1903  289
1904  256
1905  283
```

To speed things up, we need to be able to do this processing on multiple machines

- **STEP 1:** Divide the work and execute concurrently on multiple machines
- **STEP 2:** Combine results from independent processes
- **STEP 3:** Deal with failures that might take place in the system

The Hollywood principle

**Don't call us, we'll call you.**

- Useful software development technique
- Object's (or component's) initial condition and ongoing life cycle is handled by its environment, rather than by the object itself
- Typically used for implementing a class/component that must fit into the constraints of an existing framework

Doing the analysis with Hadoop

- Break the processing into two phases
  - Map and Reduce
  - Each phase has `<key, value>` pairs as input and output
- Specify two functions
  - Map
  - Reduce

The map phase

- Choose a Text input format
  - Each line in the dataset is given as a text value
  - key is the offset of the beginning of the line from the beginning of the file
- Our map function
  - Pulls out year and the air temperature
  - Think of this as a data preparation phase
  - Reducer will work on data generated by the maps

How the data is represented in the actual file

```
0067011990999991950051507004 ...9999999N9+00001+99999999999...
0043011990999991950051512004 ...9999999N9+00221+99999999999...
0043011990999991950051518004 ...9999999N9-00111+99999999999...
0043012650999991949032412004 ...0500001N9+01111+99999999999...
0043012650999991949032418004 ...0500001N9+00781+99999999999...
```
How the lines in the file are presented to the map function by the framework

- **Keys**: Line offsets within the file
- **Values**: Data associated with the key

The lines are presented to the map function as key-value pairs.

Map function

- Extract year and temperature from each record and emit output

<table>
<thead>
<tr>
<th>Year</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>0</td>
</tr>
<tr>
<td>1959</td>
<td>22</td>
</tr>
<tr>
<td>1949</td>
<td>-111</td>
</tr>
<tr>
<td>1949</td>
<td>78</td>
</tr>
</tbody>
</table>

The output from the map function

- Processed by the MapReduce framework before being sent to the reduce function
- Sorted and grouped key-value pairs by key
- In our example, each year appears with a list of all its temperature readings

<table>
<thead>
<tr>
<th>Year</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1949</td>
<td>111, 78</td>
</tr>
<tr>
<td>1959</td>
<td>0, 22</td>
</tr>
</tbody>
</table>

What about the reduce function?

- All it has to do now is iterate through the list supplied by the maps and pick the max reading
- Example output at the reducer?

<table>
<thead>
<tr>
<th>Year</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1949</td>
<td>111</td>
</tr>
<tr>
<td>1959</td>
<td>22</td>
</tr>
</tbody>
</table>

What does the actual code to do all of this look like?

1. Map functionality
2. Reduce functionality
3. Code to run the job

The map function is represented by an abstract Mapper class

- Declares an abstract map() method
- Mapper class is a generic type
  - 4 formal type parameters
  - Specifies input key, input value, output key, and output value
The Mapper for our example

```
public class MaxTemperatureMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
  private final int MISSING = 9999;
  public void map(LongWritable key, Text value, Context context)
  throws IOException, InterruptedException {
    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') {
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]") {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

Rather than use built-in Java types, Hadoop uses its own set of basic types

- Optimized for network serialization
- These are in the org.apache.hadoop.io package
- LongWritable corresponds to Java Long
- Text corresponds to Java String
- IntWritable corresponds to Java Integer

But the map() method also had Context

- You use this to write the output
- In our example
  - Year was written as a Text object
  - Temperature was wrapped as an IntWritable

More about Context

- A context object is available at any point of the MapReduce execution
- Provides a convenient mechanism for exchanging required system and job-wide information
- Context coordination happens only when an appropriate phase (driver, map, reduce) of a MapReduce job starts.
- Values set by one mapper are not available in another mapper but is available in any reducer.

The reduce function is represented by an abstract Reducer class

- Declares an abstract reduce() method
- Reducer class is a generic type
  - 4 formal type parameters
  - Used to specify the input and output types of the reduce function
  - The input types should match the output types of the map function
  - In the example, Text and IntWritable

The Reducer

```
public class MaxTemperatureReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
  throws IOException, InterruptedException {
    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```
The code to run the MapReduce job

```java
public class MaxTemperature {
    public static main(String[] args) throws Exception {
        Job job = Job.getInstance();
        job.setJarByClass( MaxTemperature.class);
        job.setJobName("Max temperature");
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(MaxTemperatureMapper.class );
        job.setReducerClass(MaxTemperatureReducer.class );
        job.setOutputKeyClass( Text.class);
        job.setOutputValueClass(IntWritable.class );
        System.exit(job.waitForCompletion(true ) ? 0: 1);
    }
}
```

Details about the Job submission

- Code must be packaged in a JAR file for Hadoop to distribute over the cluster
  - `setJarByClass()` causes Hadoop to locate relevant JAR file by looking for JAR that contains this class
- Input and output paths must be specified next
  - `addInputPath()` can be called more than once
  - `setOutputPath()` specifies the output directory
    - Directory should not exist before running the job
    - Precaution to prevent data loss
- The methods `setOutputKeyClass()` and `setOutputValueClass()`
  - Control the output types of the map and reduce functions
  - `setMapOutputKeyClass()` and `setMapOutputValueClass()`
  - Control the map output types
  - If they are different?
  - Map output types can be set using `setMapOutputKeyClass()` and `setMapOutputValueClass()`
- The `waitForCompletion()` method submits the job and waits for it to complete
  - The boolean argument is a `verbose` flag; if set, progress information is printed on the console
  - `waitForCompletion()` indicates success (true) or failure (false)
  - In the example this is the program's exit code (0 or 1)

Contents of this slide set are based on the following references