Programming with Hadoop
MapReduce

Kostas Solomos
Computer Science Department
University of Crete, Greece
What we will cover...

- Diving deeper into Hadoop architecture
- Using a shared input for the mappers/reducers
  - Distributed Cache, Variables, Accessing HDFS programmatically
- Counters
  - Default, Custom
- Reporting progress/status
- Tuning a Job
  - Setting the number of mappers & reducers
  - Load Balancing
  - Data Compression (Intermediate/Final output, available codecs)
- Multiple Inputs/Outputs
- Chaining Jobs
  - ChainMapper, ChainReducer
A DEEPER LOOK INTO HADOOP ARCHITECTURE
High Level MRv1 Data Flow

JobTracker handles all scheduling & data flow between TaskTrackers.

TaskTracker handles all worker tasks on a node.
Hadoop v.1 vs Hadoop v.2

**HADOOP 1.0**
- **MapReduce**
  - (cluster resource management & data processing)
- **HDFS**
  - (redundant, reliable storage)

**HADOOP 2.0**
- **MapReduce**
  - (data processing)
- **Others**
  - (data processing)
- **YARN**
  - (cluster resource management)
- **HDFS**
  - (redundant, reliable storage)

**Applications Run Natively IN Hadoop**
- BATCH (MapReduce)
- INTERACTIVE (Tez)
- ONLINE (HBase)
- STREAMING (Storm, S4,...)
- GRAPH (Giraph)
- IN-MEMORY (Spark)
- HPC MPI (OpenMPI)
- OTHER (Search) (Weave...)

**YARN** (Cluster Resource Management)

**HDFS2** (Redundant, Reliable Storage)
A deeper look into MR(v2)
Resource Manager

- The ultimate authority that arbitrates resources among all the applications in the system
- The Resource Manager has two main components:
  - **Applications Manager:**
    - Responsible for accepting job-submissions and negotiating the first container for executing the application-specific ApplicationMaster
    - Provides the service for restarting the ApplicationMaster container on failure
  - **Scheduler:**
    - Responsible for allocating resources to the various running applications
Application Master

- An application master is
  - started to manage each MapReduce job and
  - terminated when the job completes
- The per-application Application Master has the responsibility of:
  - negotiating appropriate resource containers from the Scheduler,
  - tracking their status and monitoring the progress
- Application Master key characteristics:
  - **Scale**: The Application Master provides much of the functionality of the traditional ResourceManager so that the entire system can scale more dramatically
  - **Open**: Moving all application framework specific code into the ApplicationMaster generalizes the system so that we can now support multiple frameworks such as MapReduce, MPI and GraphX
Node Manager

- The Node Manager is the per-machine framework agent who is responsible for containers
  - monitoring their resource usage (cpu, memory, disk, network)
  - reporting the same to the Scheduler
Container

- A resource allocation, incorporating elements such as memory, cpu, disk, network etc., on a specific host

- An application can ask for specific resource requests via the Application Master to satisfy its resource needs. The Scheduler responds to a resource request by granting a Container, which satisfies the requirements laid out by the Application Master in the initial resource request

- Can house a map task, a reduce task, or an Application Master
MRv2 Walkthrough
MRv2 Walkthrough

1. A client program submits the application, including the necessary specifications to launch the application-specific ApplicationMaster itself.
2. The ResourceManager assumes the responsibility to negotiate a specified container in which to start the ApplicationMaster and then launches the ApplicationMaster.
3. The ApplicationMaster, on boot-up, registers with the ResourceManager – the registration allows the client program to query the ResourceManager for details, which allow it to directly communicate with its own ApplicationMaster.
MRv2 Walkthrough

4. During normal operation the ApplicationMaster negotiates appropriate resource containers via the resource-request protocol.
MRv2 Walkthrough

5. The ApplicationMaster launches the container by providing the launch specification to the NodeManager. The launch specification, typically, includes the necessary information to allow the container to communicate with the ApplicationMaster itself.
6. The application code executing within the container then provides necessary information (progress, status etc.) to its ApplicationMaster via an application-specific protocol.
MRv2 Walkthrough

7. During the application execution, the client that submitted the program communicates directly with the ApplicationMaster to get status, progress updates etc. via an application-specific protocol.
MRv2 Walkthrough

8. Once the application is complete, and all necessary work has been finished, the ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed.
SHARED INPUT
Setting and Getting Variables

- Setting variables:
  Configuration conf = getConf();
  
  ...  
  Job job = Job.getInstance(conf);
  
  ...  
  conf.set("myParameterName", "StringValue");
  conf.setInt("myIntParameter", 10);

- Getting variables from Mapper/Reducer (e.g., in setup()):
  Configuration conf = context.getConfiguration();
  String sValue = conf.get("myParameterName");
  int iValue = conf.getInt("myIntParameter");
Distributed Cache

Applications specify the files to be cached via the JobConf.

The DistributedCache assumes that the files are already present on HDFS and are accessible by every machine in the cluster.

The framework will copy the necessary files on to the slave node before any tasks for the job are executed on that node.

- Efficiency: files are only copied once per job.
Using the Distributed Cache

1. Copy the requisite files to HDFS:
   
   ```bash
   bin/hadoop dfs –copyFromLocal myFile.txt /path/on/HDFS
   ```

2. Setup the application's Job:
   
   ```java
   Configuration conf = getConf();
   ...
   Job job = Job.getInstance(conf);
   ...
   job.addCacheFile(new Path(filename).toUri());
   ```

3. Use the cached files in the Mapper or Reducer (e.g., inside setup())
   
   ```java
   URI[] localPaths = context.getCacheFiles();
   ```
Accessing HDFS Programmatically

```java
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);

Path filenamePath = new Path(theFilename);
try {
    if (fs.exists(filenamePath)) { // to remove a file
        fs.delete(filenamePath);
    }
}

FSDataOutputStream out = fs.create(filenamePath); //to create a file
    out.writeUTF("message");
out.close();
    FSDataInputStream in = fs.open(filenamePath); //to read a file
    String messageIn = in.readUTF();
    System.out.println(messageIn);
in.close();
} catch (IOException ioe) { ... }
```
Example usages of shared input

- **stop words**
  - a list of words to be excluded from the computations
  - e.g., can be used in a word counting program

- **parameter values**
  - e.g., $k$ for a top-$k$ computation, threshold values, user input

- **the output of previous jobs**
  - e.g., the rank of the input words, based on their frequency, an inverted index
  - for example, find the cosine similarity of two documents, based on their TF*IDF scores (information retrieval)
HADOOP COUNTERS
Hadoop Counters

The Hadoop system records a set of metric counters for each job that it runs.

- For example, the number of input records mapped, the number of bytes it reads from or writes to HDFS, etc.

To profile your applications, you may wish to record other values as well.

- For example, if the records sent into your mappers fall into two categories (call them "A" and "B"), you may wish to count the total number of A-records seen vs. the total number of B-records.
## Default Counters

<table>
<thead>
<tr>
<th>Group Name</th>
<th>Counter Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>MAP_INPUT_RECORDS</td>
</tr>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>MAP_OUTPUT_RECORDS</td>
</tr>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>MAP_INPUT_BYTES</td>
</tr>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>MAP_OUTPUT_BYTES</td>
</tr>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>COMBINE_INPUT_RECORDS</td>
</tr>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>COMBINE_OUTPUT_RECORDS</td>
</tr>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>REDUCE_INPUT_GROUPS</td>
</tr>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>REDUCE_INPUT_RECORDS</td>
</tr>
<tr>
<td>org.apache.hadoop.mapred.Task$Counter</td>
<td>REDUCE_OUTPUT_RECORDS</td>
</tr>
</tbody>
</table>

To get the value of a Counter:

```java
context.getCounter(groupName, counterName);
```

may already have the answer to “how many (distinct)... ?”, or “what percentage of... ?”
Custom Counters

enum MyCounters{NO-CHAR, SINGLE-CHAR, DOUBLE-CHAR};

... if (word.isEmpty()) {
    // counts words without characters → error
    context.getCounter(MyCounters.NO-CHAR).increment(1);
    return; // or print an error (error logs can be huge…)
} else if(word.length() == 1) {
    // counts single-character words
    context.getCounter(MyCounters.SINGLE-CHAR).increment(1);
} else if(word.length() == 2) {
    // counts double-character words
    context.getCounter(MyCounters.DOUBLE-CHAR).increment(1);
} else ...
Reporting Status/Progress

Prevent your map/reduce tasks from being killed (as inactive), by reporting progress/status.

The tasks are not killed when they report status/progress, or receive/emit a new record, or increment a counter.

Example:
for (int i = 0; i < VERY_BIG_NUMBER; ++i) {
    ... //something very slow to finish
    context.progress(); //or better...
    context.setStatus("on iteration " + i + " out of " + VERY_BIG_NUMBER);
}

context.write(key, value);
The Context

- 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`

- The most useful methods of `Context`:
  - `write`: Generate an output key/value pair
  - `progress` and `setStatus`: report progress or set the status of the 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
JOB TUNING
Make it faster!

- Set the number of mappers:
  - Rule of thumb: each mapper should take around a minute
    - have fewer mappers, if they run too fast
- Set the number of reducers:
  - Two choices (rules of thumb again):
    - have almost as many reducers as the number of reduce slots
    - finish in one wave of reducers, more work for each node
    - have almost twice as many reducers as the number of reduce slots
      - better load balancing, bigger overhead
- Use a combiner
  - it reduces data transfer
- Use a custom Partitioner
- Compress the intermediate output
  - smaller data → faster data transfer
Under the hood...
Load Balancing

- A single slow task may slow-down the whole job
- A custom Partitioner can be used to assign the work load evenly to all reducers
  - map-side load-balancing is trickier
Data Compression

- MapReduce jobs are usually I/O bound

- Data compression:
  - speeds up I/O operations
  - saves storage space
  - speeds up data transfers
  - increases CPU utilization and processing time during compression and decompression

adapted from http://www.slideshare.net/ydn/hug-compression-talk
# Compression Options in Hadoop

<table>
<thead>
<tr>
<th>Format</th>
<th>Strategy</th>
<th>Codec</th>
<th>Splittable*</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>Dictionary-based</td>
<td>org.apache.hadoop.io.compress.GzipCodec</td>
<td>N</td>
</tr>
<tr>
<td>bzip2</td>
<td>Transform-based, block-oriented</td>
<td>org.apache.hadoop.io.compress.BZip2Codec</td>
<td>Y</td>
</tr>
<tr>
<td>LZO</td>
<td>Dictionary-based, block-oriented</td>
<td>com.hadoop.compression.lzo.LzooCodec</td>
<td>N</td>
</tr>
<tr>
<td>LZ4</td>
<td>Fast scan</td>
<td>org.apache.hadoop.io.compress.Lz4Codec</td>
<td>N</td>
</tr>
<tr>
<td>Snappy</td>
<td>Block-oriented</td>
<td>org.apache.hadoop.io.compress.SnappyCodec</td>
<td>N</td>
</tr>
</tbody>
</table>

*Splittable*: can be decompressed in parallel by multiple MapReduce tasks.

adapted from http://www.slideshare.net/ydn/hug-compression-talk
Space-Time Tradeoff of Compression Options

Codec Performance on the Wikipedia Text Corpus

CPU Time in Sec. (Compress + Decompress)

High Compression Speed

High Compression Ratio

71%, 60.0

Bzip2

64%, 32.3

Zlib
(Deflate, Gzip)

42%, 4.0
44%, 2.4
47%, 4.8

Snappy
LZ4
LZO

Space Savings

adapted from http://www.slideshare.net/ydn/hug-compression-talk
Using compression in Hadoop

- To compress the final output (using Snappy):

```java
conf.setOutputFormat(SequenceFileOutputFormat.class);
SequenceFileOutputFormat.setOutputCompressionType(conf,
    CompressionType.BLOCK);
SequenceFileOutputFormat.setCompressOutput(conf, true);
conf.set("mapred.output.compression.codec",
    "org.apache.hadoop.io.compress.SnappyCodec");
```

- To compress the map output (using Snappy):

```java
Configuration conf = new Configuration();
conf.setBoolean("mapred.compress.map.output", true);
conf.set("mapred.map.output.compression.codec",
    "org.apache.hadoop.io.compress.SnappyCodec");
```
MULTIPLE INPUTS/OUTPUTS
Multiple Inputs/Outputs

- **Multiple Inputs:**
  - you can assign different mappers for each input data, in the same job, that will be handled by the same reducer
  - handle input files with different data formats
  - useful for joins!

- **Multiple Outputs:**
  - generate additional outputs to the usual output
    - each additional output, may be configured with its own OutputFormat, with its own key class and value class
Multiple Inputs Usage

Example application:
A program that outputs, for each English word, its frequency in a document corpus and its translation to other languages

```java
MultipleInputs.addInputPath(job, new Path(documentsSource), SequenceFileInputFormat.class, WordCountMapper.class);
MultipleInputs.addInputPath(job, new Path(translationsPath), TextInputFormat, TranslateMapper.class);
job.setReducerClass(CommonReducer.class);
```
Multiple Outputs (Job Submission)

Job job = new Job();
FileInputFormat.setInputPath(job, inDir);
FileOutputFormat.setOutputPath(job, outDir);
job.setMapperClass(MOMap.class);
job.setReducerClass(MOReduce.class);
...

// Defines additional single text based output 'text' for the job
MultipleOutputs.addNamedOutput(job, "text", TextOutputFormat.class,
                                 LongWritable.class, Text.class);

// Defines additional sequence-file based output 'sequence' for the job
MultipleOutputs.addNamedOutput(job, "seq",
                                 SequenceFileOutputFormat.class, LongWritable.class, Text.class);
Multiple Outpus (in Reducer)

```java
public class MOReduce extends Reducer... {
    private MultipleOutputs mos;

    public void setup(Context context) {
        mos = new MultipleOutputs(context);
    }
    public void reduce(...) throws IOException {
        ...
        mos.write("text", key, new Text("Hello"));
        mos.write("seq", new LongWritable(1), new Text("Bye"), "seq_a");
        mos.write("seq", new LongWritable(2), key, new Text("Chau"), "seq_b");
    }
    public void cleanup(Context) throws IOException {
        mos.close();
    }
}
```
CHAINING JOBS
Chaining Jobs

- Many problems which at first seem impossible in MapReduce, can be accomplished by multiple jobs.
  - Map1 -> Reduce1 -> Map2 -> Reduce2 -> Map3...

Example:
Use the output of wordcount as input of a new job that classifies words as frequent, normal, or rare

Two main options to implement job chaining:
- run one job after another finishes
- use an external workflow scheduler system (Oozie)
  - not covered in this talk
ChainMapper and ChainReducer

- The ChainMapper class allows to use multiple Mapper classes within a single Map task
  - The Mapper classes are invoked in a chained (or piped) fashion, the output of the first becomes the input of the second, and so on until the last Mapper, the output of the last Mapper will be written to the task’s output.

- The ChainReducer class allows to chain multiple Mapper classes after a Reducer within the Reducer task
  - For each record output by the Reducer, the Mapper classes are invoked in a chained (or piped) fashion, the output of the first becomes the input of the second, and so on until the last Mapper, the output of the last Mapper will be written to the task’s output.

- Using the ChainMapper and the ChainReducer classes is possible to compose Map/Reduce jobs that look like `[MAP+ / REDUCE MAP*]`.
  - An immediate benefit of this pattern is a reduction in disk IO
Resources

- Large part of this tutorial was adapted from https://developer.yahoo.com/hadoop/tutorial/index.html, under a Creative Commons Attribution 3.0 Unported License.
- The slides for compressing data were adapted from http://www.slideshare.net/ydn/hug-compression-talk.
- The first part of this tutorial was adapted from
  - http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html,
  - http://blog.cloudera.com/blog/2013/11/migrating-to-mapreduce-2-on-yarn-for-users/ and
MR VERSION 1
Job Submission

1. Get new job ID from JobTracker
2. Determine input splits for job
3. Copy job resources (job JAR file, configuration file, computed input splits) to HDFS into directory named after the job ID
4. Informs JobTracker that job is ready for execution
Job Initialization

- JobTracker puts ready job into internal queue
- Job scheduler picks job from queue
  - Initializes it by creating job object
  - Creates list of tasks
    - One map task for each input split
    - Number of reduce tasks determined by mapred.reduce.tasks property in Job, which is set by setNumReduceTasks()
- Tasks need to be assigned to worker nodes
Task Scheduling

- **TaskTrackers** send heartbeat to **JobTracker**
  - Indicate if ready to run new tasks
  - Number of “slots” for tasks depends on number of cores and memory size

- **JobTracker** replies with new task
  - Chooses task from first job in priority-queue
    - Chooses map tasks before reduce tasks
    - Chooses map task whose input split location is closest to machine running the TaskTracker instance
    - Ideal case: data-local task
  - Could also use other scheduling policy
Task Execution

- TaskTracker copies job JAR and other configuration data (e.g., distributed cache) from HDFS to local disk
- Creates local working directory
- Creates TaskRunner instance
- TaskRunner launches new JVM (or reuses one from another task) to execute the JAR

- Monitoring Job Progress
  - Tasks report progress to TaskTracker
  - TaskTracker includes task progress in heartbeat message to JobTracker
  - JobTracker computes global status of job progress
  - JobClient polls JobTracker regularly for status
  - Visible on console and Web UI
Task Execution

1. JobTracker
   - Assign task for Execution

2. Task Tracker
   - Upto MAX_MAP_SLOTS
     - Map Task JVMs Concurrently
   - Upto MAX_REDUCE_SLOTS
     - Reduce Task JVMs Concurrently
   - Read into Local Disk
     - Job.xml, Job.jar

3. DFS
Handling Failures

Task
- Error reported to TaskTracker and logged
- Hanging task detected through timeout
- JobTracker will automatically re-schedule failed tasks
  - Tries up to mapred.map.max.attempts many times (similar for reduce)
  - Job is aborted when task failure rate exceeds mapred.max.map.failures.percent (similar for reduce)

TaskTracker and JobTracker
- TaskTracker failure detected by JobTracker from missing heartbeat messages
  - JobTracker re-schedules map tasks and not completed reduce tasks from that TaskTracker
- Hadoop cannot deal with JobTracker failure
  - Could use Google’s proposed JobTracker take-over idea, using ZooKeeper to make sure there is at most one JobTracker
Handling Failures

- **NameNode Death**
  - No new requests can be served while NameNode is down
  - Secondary will not fail over as new primary

- **So why have a secondary at all?**
  - If NameNode dies from software glitch, just reboot
  - But if machine is hosed, metadata for cluster is irretrievable!

- If original NameNode can be restored, secondary can re-establish the most current metadata snapshot
  - If not, create a new NameNode, use secondary to copy metadata to new primary, restart whole cluster
Further Reliability Measures

- Problem: DataNodes “fix” the address of the NameNode in memory, can’t switch in flight
- Solution: Bring new NameNode up, but use DNS to make cluster believe it’s the original one

- Namenode can output multiple copies of metadata files to different directories
  - Including an NFS mounted one
- May degrade performance; watch for NFS locks
Making Hadoop Work

- Basic configuration involves pointing nodes at master machines
  - `mapred.job.tracker`
  - `fs.default.name`
  - `dfs.data.dir`, `dfs.name.dir`
  - `hadoop.tmp.dir`
  - `mapred.system.dir`

- See “Hadoop Quickstart” in online documentation
Configuring for Performance

- Configuring Hadoop performed in “base JobConf” in conf/hadoop-site.xml

- Contains three different categories of settings
  - Settings that make Hadoop work
  - Settings for performance
  - Optional flags/bells & whistles
Number of Tasks

- Controlled by two parameters:
  - mapred.tasktracker.map.tasks.maximum
  - mapred.tasktracker.reduce.tasks.maximum

- Two degrees of freedom in mapper run time: Number of tasks/node, and size of InputSplits

- Current conventional wisdom: 2 map tasks/core, less for reducers
  - See http://wiki.apache.org/lucene-hadoop/HowManyMapsAndReduces
Working With the Scheduler

- **Remember:** Hadoop has a FIFO job scheduler
  - No notion of fairness, round-robin

- **Design your tasks to “play well” with one another**
  - Decompose long tasks into several smaller ones which can be interleaved at Job level