Lecture 3 – Hadoop Technical Introduction

CSE 490H
Announcements

- My office hours: M 2:30—3:30 in CSE 212
- Cluster is operational; instructions in assignment 1 heavily rewritten
- Eclipse plugin is “deprecated”
- Students who already created accounts: let me know if you have trouble
Breaking news!

- Hadoop tested on 4,000 node cluster
  - 32K cores (8 / node)
  - 16 PB raw storage (4 x 1 TB disk / node)
    (about 5 PB usable storage)

You Say, “tomato…”

<table>
<thead>
<tr>
<th>Google calls it:</th>
<th>Hadoop equivalent:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MapReduce</td>
<td>Hadoop</td>
</tr>
<tr>
<td>GFS</td>
<td>HDFS</td>
</tr>
<tr>
<td>Bigtable</td>
<td>HBase</td>
</tr>
<tr>
<td>Chubby</td>
<td>Zookeeper</td>
</tr>
</tbody>
</table>
Some MapReduce Terminology

- **Job** – A “full program” - an execution of a Mapper and Reducer across a data set
- **Task** – An execution of a Mapper or a Reducer on a slice of data
  - a.k.a. Task-In-Progress (TIP)
- Task Attempt – A particular instance of an attempt to execute a task on a machine
Terminology Example

- Running “Word Count” across 20 files is one job
- 20 files to be mapped imply 20 map tasks + some number of reduce tasks
- At least 20 map task attempts will be performed… more if a machine crashes, etc.
Task Attempts

- A particular task will be attempted at least once, possibly more times if it crashes
  - If the same input causes crashes over and over, that input will eventually be abandoned
- Multiple attempts at one task may occur in parallel with speculative execution turned on
  - Task ID from TaskInProgress is not a unique identifier; don’t use it that way
MapReduce: High Level

Master node

MapReduce job submitted by client computer

JobTracker

Slave node

TaskTracker

Task instance

Slave node

TaskTracker

Task instance

Slave node

TaskTracker

Task instance
Node-to-Node Communication

- Hadoop uses its own RPC protocol
- All communication begins in slave nodes
  - Prevents circular-wait deadlock
  - Slaves periodically poll for “status” message
- Classes must provide explicit serialization
Nodes, Trackers, Tasks

- Master node runs *JobTracker* instance, which accepts *Job* requests from clients

- *TaskTracker* instances run on slave nodes

- TaskTracker forks separate Java process for task instances
Job Distribution

- MapReduce programs are contained in a Java “jar” file + an XML file containing serialized program configuration options
- Running a MapReduce job places these files into the HDFS and notifies TaskTrackers where to retrieve the relevant program code

... Where’s the data distribution?
Data Distribution

- Implicit in design of MapReduce!
  - All mappers are equivalent; so map whatever data is local to a particular node in HDFS

- If lots of data does happen to pile up on the same node, nearby nodes will map instead
  - Data transfer is handled implicitly by HDFS
Configuring With JobConf

- MR Programs have many configurable options
- `JobConf` objects hold (key, value) components mapping `String → 'a`
  - e.g., “mapred.map.tasks” → 20
  - JobConf is serialized and distributed before running the job
- Objects implementing `JobConfigurable` can retrieve elements from a JobConf
What Happens In MapReduce? Depth First
Client program creates a `JobConf`

- Identify classes implementing `Mapper` and `Reducer` interfaces
  - `JobConf.setMapperClass()`, `setReducerClass()`

- Specify inputs, outputs
  - `FileInputFormat.addInputPath()`,
  - `FileOutputFormat.setOutputPath()`

- Optionally, other options too:
  - `JobConf.setNumReduceTasks()`,
    - `JobConf.setOutputFormat()`
Job Launch Process: *JobClient*

- Pass JobConf to JobClient.runJob() or submitJob()
  - runJob() blocks, submitJob() does not

*JobClient*:
- Determines proper division of input into *InputSplits*
- Sends job data to master *JobTracker* server
Job Launch Process: *JobTracker*

- **JobTracker:**
  - Inserts jar and JobConf (serialized to XML) in shared location
  - Posts a *JobInProgress* to its run queue
Job Launch Process: *TaskTracker*

- *TaskTrackers* running on slave nodes periodically query *JobTracker* for work
- Retrieve job-specific jar and config
- Launch task in separate instance of Java
  - `main()` is provided by Hadoop
Job Launch Process: Task

- TaskTracker.Child.main():
  - Sets up the child TaskInProgress attempt
  - Reads XML configuration
  - Connects back to necessary MapReduce components via RPC
  - Uses TaskRunner to launch user process
Job Launch Process: *TaskRunner*

- *TaskRunner, MapTaskRunner, MapRunner* work in a daisy-chain to launch your *Mapper*
  - Task knows ahead of time which *InputSplits* it should be mapping
  - Calls *Mapper* once for each record retrieved from the *InputSplit*
- Running the *Reducer* is much the same
Creating the Mapper

- You provide the instance of Mapper
  - Should extend MapReduceBase
- One instance of your Mapper is initialized by the MapTaskRunner for a TaskInProgress
  - Exists in separate process from all other instances of Mapper – no data sharing!
Mapper

- void map(K1 key,
  V1 value,
  OutputCollector<K2, V2> output,
  Reporter reporter)

- $K$ types implement WritableComparable
- $V$ types implement Writable
What is Writable?

- Hadoop defines its own “box” classes for strings \((\text{Text})\), integers \((\text{IntWritable})\), etc.
- All values are instances of \(\text{Writable}\)
- All keys are instances of \(\text{WritableComparable}\)
Getting Data To The Mapper
Reading Data

- Data sets are specified by *InputFormats*
  - Defines input data (e.g., a directory)
  - Identifies partitions of the data that form an *InputSplit*
  - Factory for *RecordReader* objects to extract \((k, v)\) records from the input source
**FileInputFormat** and Friends

- **TextInputFormat** – Treats each ‘\n’-terminated line of a file as a value
- **KeyValueTextInputFormat** – Maps ‘\n’-terminated text lines of “k SEP v”
- **SequenceFileInputFormat** – Binary file of (k, v) pairs with some add’l metadata
- **SequenceFileAsTextInputFormat** – Same, but maps (k.toString(), v.toString())
Filtering File Inputs

- `FileInputFormat` will read all files out of a specified directory and send them to the mapper

- Delegates filtering this file list to a method subclasses may override
  - *e.g.*, Create your own “xyzFileInputFormat” to read *.xyz* from directory list
Record Readers

- Each *InputFormat* provides its own *RecordReader* implementation
  - Provides (unused?) capability multiplexing
- *LineRecordReader* – Reads a line from a text file
- *KeyValueRecordReader* – Used by *KeyValueTextInputFormat*
Input Split Size

- `FileInputFormat` will divide large files into chunks
  - Exact size controlled by `mapred.min.split.size`
- RecordReaders receive file, offset, and length of chunk
- Custom `InputFormat` implementations may override split size – e.g., “NeverChunkFile”
Sending Data ToReducers

- Map function receives `OutputCollector` object
  - `OutputCollector.collect()` takes (k, v) elements
- Any `(WritableComparable, Writable)` can be used
- By default, mapper output type assumed to be same as reducer output type
**WritableComparator**

- Compares WritableComparable data
  - Will call WritableComparable.compare()
  - Can provide fast path for serialized data
- `JobConf.setOutputValueGroupingComparator()`
Sending Data To The Client

- *Reporter* object sent to Mapper allows simple asynchronous feedback
  - `incrCounter(Enum key, long amount)`
  - `setStatus(String msg)`
- Allows self-identification of input
  - `InputSplit getInputSplit()`
Partition And Shuffle

Mapper → (intermediates) → Partitioner → (intermediates) → Reducer

Mapper → (intermediates) → Partitioner → (intermediates) → Reducer

Mapper → (intermediates) → Partitioner → (intermediates) → Reducer
Partitioner

- int getPartition(key, val, numPartitions)
  - Outputs the partition number for a given key
  - One partition == values sent to one Reduce task
- HashPartitioner used by default
  - Uses key.hashCode() to return partition num
- JobConf sets Partitioner implementation
Reduction

- `reduce(K2 key, Iterator<V2> values, OutputCollector<K3, V3> output, Reporter reporter)`

- Keys & values sent to one partition all go to the same reduce task

- Calls are sorted by key – “earlier” keys are reduced and output before “later” keys
Finally: Writing The Output

Reducer

RecordWriter

output file

Reducer

RecordWriter

output file

Reducer

RecordWriter

output file

OutputFormat
OutputFormat

- Analogous to InputFormat
- TextOutputFormat – Writes “key val\n” strings to output file
- SequenceFileOutputFormat – Uses a binary format to pack (k, v) pairs
- NullOutputFormat – Discards output
Questions?