Big Data 4

Week 3: MapReduce

Dr. Nikos Ntarmos

Room M111, Lilybank Gardens
<nikos.ntarmos@glasgow.ac.uk>

School of Computing Science
University of Glasgow

15 October, 2013

Outline

MapReduce basics :: Introduction

Motivation

- Need to process **petabytes** of data ...
- ... on your **multi-thousand commodity PC** cluster ...
- ... and make it **easy**!

$\Rightarrow$ Enter **Map/Reduce**

[1] J. Dean and S. Ghemawat. ``MapReduce: Simplified Data Processing on Large Clusters´´. In *Proc. OSDI '04.*

What is MapReduce?

1. MapReduce -- The programming model
   - Functional-like programming $\rightarrow$ Map/reduce functions
2. MapReduce -- The execution framework
   - Input splitting, intermediate result sort/shuffling, task scheduling, ...
3. MapReduce -- The software implementation
   - Google MapReduce, Hadoop, Storm/Kafka, AcB/Cell, ...

MapReduce premises

- Scale *out*, not *up*.
- Design for fault tolerance (assume failures are the norm).
- Move processing to the data.
- Optimize for **high throughput batch jobs** -- i.e., sequential reads, no random accesses.
- Hide system-related stuff from the app developers ⇒ provide automated distribution, parallelization, scheduling, . . .
- Provide mechanisms for progress report and monitoring.
- ``Always'' operates on key-value pairs

Hadoop Timeline

2002: Apache Nutch (open-source web search engine).

2003: Google File System (GFS) paper appears in SOSP.

2004: Nutch Distributed File System (NDFS) airs.

Google MapReduce paper appears in OSDI.

2005: MapReduce is implemented for NDFS.

2006: **Hadoop is born!**

2008: Hadoop gets top-level Apache project status

... and breaks the record for fastest TB sort.

2009: Hadoop breaks two more records ($<1'$ to sort 500GB, $<3'$ to sort 100 TB).

2012: YARN is born!

2013: . . .

Example: Word Count

- *``Million lyrics dataset''*

```
Hear the rime of the Ancient Mariner, see his eye as he stops one of three.
Mesmerises one of the wedding guests. Stay here and listen to the nightmares
     of the Sea.
And the music plays on, as the bride passes by. Caught by his spell and the
     Mariner tells his tale.

Driven south to the land of the snow and ice, to a place where nobody's been.
Through the snow fog flies on the albatross. Hailed in God's name, hoping
     good luck it brings.

And the ship sails on, back to the North. Through the fog and ice and the
     albatross follows on.

The mariner kills the bird of good omen. His shipmates cry against what he's
     done,
but when the fog clears, they justify him and make themselves a part of the
     crime.
[...]
```

Q: Compute all unique words in the stream and their number of appearances (*Word Count*).

Example: Word Count

Pfft... We can do that with simple shell scripts...

- Using shell programming, take #1:

```
cat input.txt | tr '[:space:]' '\n' | sort | uniq -c | \
   awk '{print $2" "$1}'
```

  Now what? $\longrightarrow$ **Must sort entire input and go over it twice!**

- Using shell programming, take #2:

```
cat input.txt | awk '\
   {for (i = 1; i <= NF; i++) freqs[$i]++;} END \
   {for (word in freqs) print word" "freqs[word];}'
```

  What's wrong with this? $\longrightarrow$ **Out of RAM as freqs grows huge! Plus serial execution...**

Let's combine these two and go all-out distributed!

MapReduce basics :: The MapReduce paradigm

The MapReduce paradigm

$$\textbf{map} : (\mathbf{k_1}, \mathbf{v_1}) \rightarrow [(\mathbf{k_2}, \mathbf{v_2})]$$
$$\textbf{reduce} : (\mathbf{k_2}, [\mathbf{v_2}]) \rightarrow [(\mathbf{k_3}, \mathbf{v_3})]$$

```
void Map(int lineNo, String line) {
   foreach (Word w in line)
      emit(tolower(w), 1);
}
```

```
void Reduce(Word w, int[] counts) {
   int sum = 0;
   foreach (int i in counts)
      sum += i;
   emit(w, sum);
}
```

‘‘And the ship sails on, back to the North, through the fog and ice and the albatross follows on.’’

```
and     1
the     1
ship    1
sails   1
on      1
back    1
to      1
the     1
north   1
through 1
the     1
fog     1
...
```

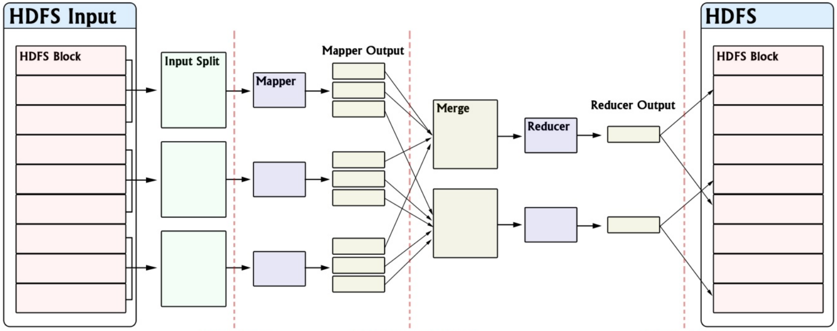
⇒

```
albatross 1
and       1,1,1
back      1
fog       1
follows   1
ice       1
north     1
on        1,1,1
sails     1
ship      1
the       1,1,1,1
through   1
to        1
```

⇒

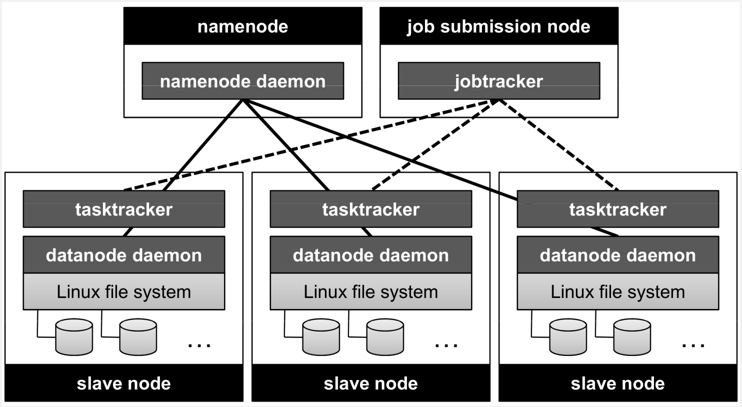
```
albatross 1
and       3
back      1
fog       1
follows   1
ice       1
north     1
on        2
sails     1
ship      1
the       4
through   1
to        1
```

MapReduce processing steps



- Input parsing & splitting $\rightarrow$ *InputFormat*
- Map function execution $\rightarrow$ *Mapper, Combiner*
- Intermediate result partitioning, sorting, and storing $\rightarrow$ *Partitioner, Merge*
- Intermediate data transfer to reducers $\rightarrow$ *Shuffle*
- Reduce function execution $\rightarrow$ *Reducer*
- Final output storage $\rightarrow$ *OutputFormat*

MapReduce infrastructure components



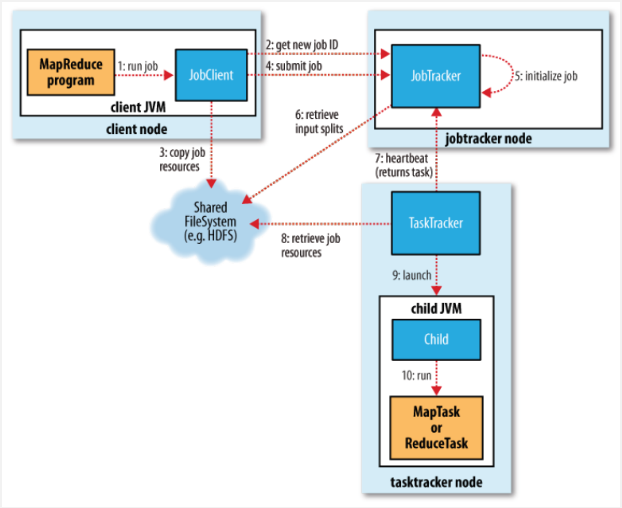
- Jobtracker-tasktrackers communication over **heartbeats**
- **Predefined slots** for mappers & reducers on tasktrackers

Anatomy of a MapReduce job :: Execution stages

Execution stages

- Execution stages:
  1. Job submission
  2. Job initialization
  3. Map tasks scheduling
  4. Execution of mappers
  5. Execution of combiners
  6. Intermediate output sorting
  7. Reduce task scheduling
  8. Execution of reducers
  9. Job completion/cleanup

Execution stages



1

[1] Source: *Hadoop: The definitive guide, 3rd Edition.*

Anatomy of a MapReduce job :: Job submission

Job submission

1. Contact the jobtracker and get a new job ID
2. Configure all client-writable job settings
3. Define the class names implementing the mappers, reducers, combiners, ...
4. Define the types of keys, values, intermediate results, ...
5. Compute input splits
6. Upload class files (jar), conf files, split keys, etc. to HDFS
7. Let the jobtracker know everything is ready (i.e., ``submit´´ the job)

Input handling

Goal: High throughput, parallel, fault tolerant data crunching

⇒ Allow for as much parallelism as possible

⇒ Minimize the need for synchronization across mappers

⇒ Lose only a small part of the computation when a node goes down

Solution: Split up the input; let each mapper get its own part

Input splitting

- Default strategy: split files into **as many equally-sized parts** as there are mappers
- Splits should (by default) be at most one HDFS block large **(lower limit on # splits)**
- Splits are precomputed & stored along with the data files on HDFS
- Splits are **data agnostic**

What happens if a word is split in half?

Let the InputFormat decide!

Input splitting

`abstract class org.apache.hadoop.mapreduce.InputFormat`
(subclasses under `org.apache.hadoop.mapreduce.lib.input`)

- → `FileInputFormat`: Read data from files on HDFS; **byte-oriented operations**
  - → `TextInputFormat`: Treat input as text files; line-oriented operation
  - → `KeyValueTextInputFormat`: Treat input as files storing key-value pairs, one per line; key-values separated by a special character (default: tab)
  - → `SequenceFileInputFormat`: Treat input as binary files storing key-value pairs (i.e., `org.apache.hadoop.io.SequenceFile`'s)
    - → `SequenceFileAsTextInputFormat`: Converts keys/values to Strings by means of their `toString()` method
    - → `SequenceFileAsBinaryInputFormat`: Reads keys/values in their raw format
  - → `CombineFileInputFormat`: Process multiple small files as a single large file
    - → `CombineTextInputFormat`: Combine multiple text files
    - → `CombineSequenceFileInputFormat`: Combine multiple `SequenceFile`'s
  - → `NLineInputFormat`: Treat input as text files, create a split every N lines
- → `DBInputFormat`: Read data from a SQL table (over JDBC)
- → `CompositeInputFormat`: ``Join'' multiple data sources

Anatomy of a MapReduce job :: Job initialization

Job initialization

- The client has now submitted the new job to the Jobtracker.

→ The job is added to the scheduler queue

→ The scheduler picks it up and *initializes it*

1. Retrieves the precomputed input splits
2. Creates one *map* task per split
3. Creates as many *reduce* tasks as requested by the client
4. Creates a *job setup* task for each tasktracker
   (→ creates intermediate and final output directories, if needed)
5. Creates a *job cleanup* task for each tasktracker
   (→ removes intermediate output files and directories)

Anatomy of a MapReduce job :: Task assignment

Task assignment

- Next step: assign tasks to tasktrackers
- The jobtracker selects which tasktracker will take on the next task

? **How does it know which tasktracker is idle?**

→ Through jobtracker-tasktracker **heartbeats**

? **How does it know which task to run where?**

→ That's a work for the **Scheduler**

  - FIFO / Fair / Capacity Scheduler
  - Data-local, rack-local, ``remote'' tasks

- ... plus **Speculative Execution** ...

Anatomy of a MapReduce job :: Task execution

Task execution

- Now tasktrackers take the lead:
  1. Create local working dir for the task
  2. Transfer/unpack the job files to the local dir
  3. Launch a new JVM to execute the task code
  4. Monitor the progress of children JVM ...
     ... where progress is:
     - Reading/writing an input/output record
     - Setting the status description
     - Incrementing a counter
     - Explicitly calling the Reporter's `progress()` method
  5. Aggregate progress reports and send them to the jobtracker
  6. JVMs optionally reused -- but **not** across jobs

Anatomy of a MapReduce job :: Map tasks

Map tasks

- Map tasks execute the map function in a loop
- Separate invocation for each input record ... but this can be overriden (through `TaskInputOutputContext.nextKeyValue()`)
- Output is *emitted*, through `Context.write()`
    - Initially stored in memory
    - Periodically written out to disk
      ... or wherever requested ...

Mapper output handling

- Mappers strive for data locality. What about reducers?

→ Each reducer may need data from **all** mappers!

→ Mappers pre-partition their output to separate files

- Partitions defined by the Partitioner
    - **HashPartitioner**, `BinaryPartitioner`, `KeyFieldBasedPartitioner`, `TotalOrderPartitioner`, ...
- Each partition sorted in-memory before being written to disk
- Finally, all partitions merged/sorted into a single file on disk

Anatomy of a MapReduce job :: Combiners

Combiners

- **Combiner $\approx$ (mini) Reducer**
  - Executed prior to storing map outputs on disk (maybe also later)
  - Works with data already in main memory $\rightarrow$ **fast** data access
  - Pre-processes output data just like a reducer would
  - Speeds up later phases (sorting/shuffling/transferring/reducing)
    - $\Rightarrow$ Lower intermediate results' disk space usage
    - $\Rightarrow$ Lower shuffling phase network usage
    - $\Rightarrow$ ... and more ...

Anatomy of a MapReduce job :: Reduce tasks

Reduce tasks

1. As map tasks finish, the jobtracker informs tasktrackers to launch reducers
2. Each reducer first fetches relevant map output partitions

   ... in parallel, to increase throughput
3. Partitions are merge/sorted, in rounds
4. The final round output is directly fed to the reducer function...

   ... and its output directly written to HDFS (or wherever instructed by the job's `OutputFormat`)

Anatomy of a MapReduce job :: Job output

Job output

`abstract class org.apache.hadoop.mapreduce.OutputFormat`
(subclasses under `org.apache.hadoop.mapreduce.lib.output`)

- → `FileOutputFormat`: Write data to files on HDFS
  - → `TextOutputFormat`: Write plain text files
  - → `MapFileOutputFormat`: Write indexed key-value files
  - → `SequenceFileOutputFormat`: Write output in `org.apache.hadoop.io.SequenceFile`'s
    - → `SequenceFileAsBinaryOutputFormat`: Write output in raw format
- → `NullOutputFormat`: Send all output to `/dev/null`
- → `LazyOutputFormat`: Writes data out in a *lazy* manner
- → `MultipleOutputs`: Writes data to multiple files, possibly using a different `OutputFormat` for each

Extras :: Compression

Compression

- Input/(intermediate) output data can/should be compressed
  Why?
  - More bytes per HDFS block/partition $\Rightarrow$ Lower storage/network overhead
  - Less I/Os across the dataset $\Rightarrow$ Higher throughput
  - Higher CPU overhead...
    ... but it pays out for I/O-intensive workloads
- Multiple codecs already available: ZLib, GZip, BZip2, LZO, Snappy, LZ4, ...
  ... but what about splitting???
  - Only Bzip2 splittable by default
  - Can be worked around, but not for the faint at heart
    (e.g., see `https://github.com/kevinweil/hadoop-lzo`)

Extras :: Distributed cache

Distributed cache

- Allows to distribute (read-only) files to tasktrackers
  - Can be used for jars, text, archives, conf files, etc.
  - Compressed files unarchived automatically
  - Jars/libs (can be) added to the task's classpath
- Files are copied only once per job to each tasktracker
- Can be *private* or *public*
- Interesting ``hack'' to share state across tasks!

Extras :: Counters

Counters

- Hadoop/MapReduce further supplies ``Counters''
- Built-in counters for:
  - HDFS and FileInputFormat/FileOutputFormat profiling
  - Map/reduce tasks' progress
  - Job progress
- Also support for user-defined counters
- All counters automatically aggregated at the jobtracker

Extras :: Streaming/pipes

Streaming/pipes

- Hadoop is written in Java...
- ...but can ``interface´´ with other languages as well
- Pipes
  - $\longrightarrow$ Use C++ for mapper/reducer implementations
- Streaming
  - Use any language able of I/O on stdin/stdout!

MapReduce failure handling :: Task failure

Task failure

- Task failure detected by tasktracker
  - Runtime exception/other user code bug $\rightarrow$ error reported to tasktracker by child JVM
  - Child JVM dies $\rightarrow$ tasktracker informed by the OS
  - Task is hung $\rightarrow$ detected through timeouts in reporting
- The tasktracker informs the jobtracker of the failure
- The jobtracker reschedules the failed task (to a different tasktracker, if possible)

$\rightarrow$ Multiple attempts per task

- Jobs with many failed tasks are eventually terminated
- ... or bad input records are marked and skipped

MapReduce failure handling :: Tasktracker failure

Tasktracker failure

- Tasktracker failure detected by jobtracker
  - Crashed or slow tasktracker $\longrightarrow$ missing heartbeats
  - Failing or misconfigured tasktracker $\longrightarrow$ multiple task failures
- Tasktracker removed from pool of ``live'' tasktrackers
- Tasks initially assigned to the failed tasktracker are rescheduled
- Failed/blacklisted tasktrackers reconsidered after some time

MapReduce failure handling :: Jobtracker failure

Jobtracker failure

The death of all!!!

(... well, not really, but close enough ...)

... but wait, YARN is on its way!!!

Putting it all together :: Word Count galore

Word Count v0 :: Built-in mappers/reducers

Word Count v0

```
public class WordCount extends Configured implements Tool {
   public int run(String[] args) throws Exception {
      Job job = new Job();
      job.setJobName("WordCount-v0");
      job.setJarByClass(WordCount.class);

      job.setMapperClass(TokenCounterMapper.class);
      job.setReducerClass(IntSumReducer.class);

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

      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));

      job.submit();
      return (job.waitForCompletion(true) ? 0 : 1);
   }

   public static void main(String[] args) throws Exception {
      System.exit(ToolRunner.run(new Configuration(), new WordCount(), args));
   }
}
```

Word Count v1 :: User-defined mappers/reducers

Word Count v1

```
public class WordCount extends Configured implements Tool {
   static class Map extends org.apache.hadoop.mapreduce.Mapper<LongWritable,
        Text, Text, IntWritable> {
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

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

   public static class Reduce extends Reducer<Text, IntWritable, Text,
        IntWritable> {
      public void reduce(Text key, Iterable<IntWritable> values, Context
           context) throws IOException, InterruptedException {
         int sum = 0;
         for (IntWritable value: values)
            sum += value.get();
         context.write(key, new IntWritable(sum));
      }
   }
```

Word Count v1 (cont.)

```
    public int run(String[] args) throws Exception {
        Job job = new Job();
        job.setJobName("WordCount-v1");
        job.setJarByClass(WordCount.class);

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

        job.setInputFormatClass(TextInputFormat.class);
        FileInputFormat.setInputPaths(job, new Path(args[0]));

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.submit();
        return (job.waitForCompletion(true) ? 0 : 1);
    }

    public static void main(String[] args) throws Exception {
        System.exit(ToolRunner.run(new Configuration(), new WordCount(), args));
    }
}
```

Word Count v2 ::

Distributed cache + configuration + counters + status messages + progress report

Word Count v2

- Count occurrences of words in the input stream , but also:
- Allow user to define patterns/words to be skipped
- Allow user to turn case-sensitivity on/off
- Count total number of words processed
- Report progress and update status messages as we go

Word Count v2

```java
public class WordCount extends Configured implements Tool {
   public static class Map extends Mapper<LongWritable, Text, Text, IntWritable>{
      static enum Counters { INPUT_WORDS }
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();
      private boolean caseSensitive = true;
      private Set<String> patternsToSkip = new HashSet<String>();
      private long numRecords = 0;
      private String inputFile;

      public void setup(Context context) {
         Configuration conf = context.getConfiguration();
         caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
         inputFile = conf.get("map.input.file");

         if (conf.getBoolean("wordcount.skip.patterns", false)) {
            Path[] patternsFiles = new Path[0];
            try {
               patternsFiles = DistributedCache.getLocalCacheFiles(conf);
            } catch (IOException ioe) {
               System.err.println("Caught exception while getting cached files:
                    " + StringUtils.stringifyException(ioe));
            }
            for (Path patternsFile : patternsFiles)
               parseSkipFile(patternsFile);
         }
      }
```

Word Count v2 (cont.)

```
      public void map(LongWritable key, Text value, Context context) throws
           IOException, InterruptedException {
         String line = caseSensitive ? value.toString() :
              value.toString().toLowerCase();

         for (String pattern : patternsToSkip)
            line = line.replaceAll(pattern, "");

         StringTokenizer tokenizer = new StringTokenizer(line);
         while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
            context.getCounter(Counters.INPUT_WORDS).increment(1);
         }

         if ((++numRecords % 100) == 0)
            context.setStatus("Finished processing " + numRecords + " records "
                 + "from the input file: " + inputFile);
      }
```

Word Count v2 (cont.)

```
      private void parseSkipFile(Path patternsFile) {
         try {
            BufferedReader fis = new BufferedReader(new
                 FileReader(patternsFile.toString()));
            String pattern = null;
            while ((pattern = fis.readLine()) != null)
               patternsToSkip.add(pattern);
            fis.close();
         } catch (IOException ioe) {
            System.err.println("Caught exception while parsing the cached file
                 '" + patternsFile + "' : " +
                 StringUtils.stringifyException(ioe));
         }
      }
   }

   public static class Reduce extends Reducer<Text, IntWritable, Text,
        IntWritable> {
      public void reduce(Text key, Iterable<IntWritable> values, Context
           context) throws IOException, InterruptedException {
         int sum = 0;
         for (IntWritable value: values)
            sum += value.get();
         context.write(key, new IntWritable(sum));
      }
   }
```

Word Count v2 (cont.)

```java
    public int run(String[] args) throws Exception {
        Job job = new Job();
        job.setJobName("WordCount-v3");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(Map.class);
        job.setCombinerClass(Reduce.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        FileInputFormat.setInputPaths(job, new Path(other_args.get(0)));
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileOutputFormat.setOutputPath(job, new Path(other_args.get(1)));

        List<String> other_args = new ArrayList<String>();
        for (int i = 0; i < args.length; ++i) {
            if ("-skip".equals(args[i])) {
                DistributedCache.addCacheFile(new Path(args[++i]).toUri(),
                        job.getConfiguration());
                job.getConfiguration().setBoolean("wordcount.skip.patterns", true);
            } else
                other_args.add(args[i]);
        }

        job.submit();
        return job.waitForCompletion(true) ? 0 : 1;
    }
```

Word Count v2 (cont.)

```
    public static void main(String[] args) throws Exception {
        System.exit(ToolRunner.run(new Configuration(), new WordCount(), args));
    }
}
```