MapReduce

CDP
Tutorial Outline

• MapReduce introduction and overview
• Real MapReduce implementation (Hadoop)
• Sort using MapReduce
• More examples
• Questions and solutions
Introduction

• MapReduce is a programming model and an associated implementation for processing and generating large data sets

• The programmer expresses the computation as two functions:
  – **Map**: process a *key/value* pair to generate a set of intermediate *key/value* pairs
  – **Reduce**: merge all intermediate *values* associated with the same intermediate *key*
The Map function

• Written by the user
• Takes an input pair and produces a set of intermediate key/value pairs
• The MapReduce library groups together all intermediate values associated with the same intermediate key and passes them to the reduce function
The Reduce function

- Written by the user
- Accepts an intermediate *key* and a set of *values* for that *key*
- Merges these *values* to form a possibly smaller set of *values*
  - Typically just zero or one output value is produced per reduce invocation (i.e. intermediate key)
Execution overview

Data → Splitter

Handle data source, provide input pairs to mappers

Map

Mapper → Combiner → Partition

Usually a reduce function

Shuffle/sort

Sort → Reducer

By default - hash

Reduce

Sort → Reducer

Provided, ascending

Output file 1 → Output file 2

Output file 1

Output file 2
Example: Word Count

• Count the number of occurrences of each word in a large collection of documents
• What should the map and reduce functions do?
Word Count: mapper

map(String *key*, String *value*) {
    // key: document name
    // value: document contents
    for each *word* in *value*:
        EmitIntermediate(*word*, 1);
}

• The map function emits each word with an associated count of occurrences (just “1” in this simple example)
Word Count: reducer

reduce(String \textit{key}, \textit{Iterator}\ values) \{
  // key: a word
  // values: a list of counts
  \textit{int result} = 0;
  \textit{for each} \textit{val in values}:
    \textit{result} += \textit{val}; // is \textit{v} always 1?
    \textit{Emit}(\textit{key}, \textit{result});
\}

• The reduce function sums together all counts emitted for a particular word
public class TokenizerMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);

  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    StringTokenizer st = new StringTokenizer(value.toString());
    while (st.hasMoreTokens()) {
      context.write(new Text(st.nextToken()), one);
    }
  }
}
public class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {

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

• Every reducer gets intermediate key/value pairs in *increasing* key order
  – Makes it easy to generate a sorted output file per partition

• Makes distributed sort a straightforward programming task
Distributed Sort

• Map:
  – Extract sorting key from the data (e.g. text line)
  – \textbf{Emit} (key, data) as intermediate output

• Reduce:
  – The Identity function
  – Simply \textbf{emit} intermediate \textbf{data} as final output
Distributed Sort (2)

• Partition function takes care of global ordering:
  – Each reducer gets a contiguous range of keys
    • Say *a-e, f-j, k-o, p-t, u-z*
  – Local sort takes care of local ordering
    • *Air* before *Arc*
  – Is load balance guaranteed?
Example: Word Order

• Given a text document, write a map-reduce program that orders the words by their number of occurrences in the text.

• For example, given the text:
  – “Gadolinium Gadolinium Gadolinium Gallium Garnet Garnet”

• The order should be:
  – Gallium
  – Garnet
  – Gadolinium
Example: Word Order (2)

• Solution will be in two map-reduce phases:
  1. Count the word occurrences in the document (i.e. Word Count from before)
  2. Order the words according to their number of occurrences that were calculated in the first phase
Example: Word Order (3)

• **Phase 1:** Count occurrences of each word in document:
  – *Map:*
    • For every word $w$: *Emit* $(w, 1)$ as key/value pair
  – *Reduce:*
    • Sum counts for each word $w$ and *Emit* $(w, \text{count}_w)$
Example: Word Order (4)

• **Phase 2:** order the words according to their number of occurrences:
  
  – *Map:*
    
    • for every pair \((w, \text{count}_w)\): *Emit* \((\text{count}_w, w)\) as an intermediate *key/value* pair (note change in key)
    
  – *Reduce:*
    
    • Basically the identity function
    
    • Only *emit* value as output

  – Anything special for the partition function?
public class SwapMapper extends Mapper<Text, Text, IntWritable, Text> {

    private Text word = new Text();

    public void map(Text key, Text value, Context context)
        throws IOException, InterruptedException {
        int count = Integer.parseInt(value.toString());
        context.write(new IntWritable(count), key);
    }
}
public class OutputReducer
    extends Reducer<IntWritable, Text, Text, NullWritable> {

    private final static NullWritable nothing = NullWritable.get();

    public void reduce(IntWritable key, Iterable<Text> values, Context context) 
        throws IOException, InterruptedException {
        for (Text val : values) {
            context.write(val, nothing);
        }
    }
}
Hadoop Input Format

• For each Hadoop job we define how to read the input from the files:
  – TextInputFormat (default): Splits the files to lines where the keys are the position in the file and the values are the lines
  – KeyValueTextInputFormat: Read each line from the files as text key/value separated by tab
  – Much more available implementations in Hadoop library
  – You can create your own input format (e.g. reading from the web)
public static void main(String[] args) throws Exception {
    // Used to pass parameters to the mappers and reducers
    Configuration conf = new Configuration();

    /* We chain the two Mapreduce phases using a temporary directory from which the first phase writes to, and the second reads from */
    Path TEMP_PATH = new Path("temp");

    ...
}
public static void main(String[] args) throws Exception {
    ...
    // Setup first MapReduce phase
    Job job1 = Job.getInstance(conf, "WordOrder-first");
    job1.setJarByClass(WordOrder.class);
    job1.setMapperClass(TokenizerMapper.class);
    job1.setReducerClass(IntSumReducer.class);
    job1.setMapOutputKeyClass(Text.class);
    job1.setMapOutputValueClass(IntWritable.class);
    job1.setOutputKeyClass(Text.class);
    job1.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job1, new Path(args[0]));
    FileOutputFormat.setOutputPath(job1, TEMP_PATH);
    ...
}
public static void main(String[] args) throws Exception {
    ...
    boolean status1 = job1.waitForCompletion(true);
    if (!status1) System.exit(1);

    // Setup second MapReduce phase
    Job job2 = Job.getInstance(conf, "WordOrder-second");
    job2.setJarByClass(WordOrder.class);
    job2.setMapperClass(SwapMappe.class);
    job2.setReducerClass(OutputReducer.class);
    ...
    job2.setInputFormatClass(KeyValueTextInputFormat.class);
    FileInputFormat.addInputPath(job2, TEMP_PATH);
    FileOutputStream.setOutputStreamPath(job2, new Path(args[1]));
    ...
}
public static void main(String[] args) throws Exception {

    ... 

    boolean status2 = job2.waitForCompletion(true);

    // Clean temporary files from the first MapReduce phase
    FileSystem fs = FileSystem.get(conf);
    fs.delete(TEMP_PATH, true);

    if (!status2) System.exit(1);

}
More examples

• Distributed Grep:
  – Map:
    • *Emit* a line if it matches the supplied pattern
  – Reduce:
    • The identity function

• Count URL Access Frequency:
  – Map:
    • Process logs of web page requests and *emit* (URL, 1)
  – Reduce:
    • Add together all values for the same URL and *emit* (URL, TotalCount) pair
More examples (2)

• **Reverse Web-Link Graph:**
  – *Map*:
    • *Emit* (target, source) pairs for each link in a given page
  – *Reduce*:
    • Merge all sources leading to the target URL, and *emit* (target, sources-list)

• **Inverted Index:**
  – *Map*:
    • Parses each document and *emit* (word, document-ID) pairs
  – *Reduce*:
    • For each word, sort the corresponding document-IDs and *emit* (word, document-IDs) pair
Question

• Input is a group of text files
• Write a map-reduce program that for every input document:
  – outputs the words that have the most appearances in it compared to the other documents
• For example:
  – doc1.txt : "w1 w1 w2 w3"
  – doc2.txt: "w1 w2 w3 w3"
• The output should be:
  – (doc1.txt, "w1, w2")
  – (doc2.txt, "w2, w3")
Question (2)

• Solution will be in 2 map-reduce phases:

  1. Count the number of occurrences of every word in each document
     • For example: \((w_1, \"(doc1.txt, 2), (doc2.txt, 1)\")\), ...

  2. Using the output from phase 1 as input, for every file output the words in which they appear the most
Question (3)

• Phase 1 - Count the number of occurrences of every word in each document:
  – Map:
    • Receive (DocumentName, DocumentContent) and emit $(w, DocumentName)$ for every word $w$
  – Reduce:
    • With every word $w$ get a list of document names (with duplications)
    • Merge and count document occurrences
    • Emit $(w1, "(doc1.txt, 2), (doc2.txt, 1)"), ...
Question (4)

• Phase 2: Using the output from phase 1 as input, for every file output the words in which they appear the most:
  – Map:
    • Get a list of documents per word:
      \((w, "(Doc1.txt, CountDoc1), (Doc2.txt, CountDoc2), \ldots")\)
    • **Emit** \((Doc1.txt, w)\) as intermediate *key/value* pair only for documents with highest counter
  – Reduce:
    • For each **document** (used as key), merge list of **words**
    • **Emit** \((document, \"w1, w2, \ldots\")\)