CS 555: Distributed Systems [MapReduce/Hadoop]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Types of tasks MapReduce is poor at
- Difference between Hadoop and Spark

Topics covered in this lecture

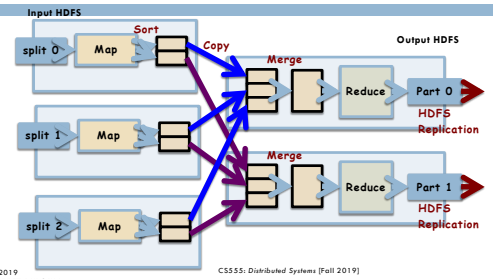
- Hadoop
  - Phases of Map
  - Phases of Reduce
  - Examples

Hadoop

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: MapReduce Dataflow



In Hadoop a Map task has 4 phases

- Record reader
- Mapper
- Combiner
- Partitioner

Map task phases: **Record Reader**

- **Translates** input splits into records
- Parse data into records, but does not parse the record itself
- Passes the data to the mapper in the form of a key/value pair
  - **key** in this context is *positional information*
  - **value** is the chunk of data that comprises a record

Map task phases: **Map**

- **User-provided code** is executed on each key/value pair from the record reader
- This user-code produces *zero or more* new key/value pairs, called the **intermediate pairs**
  - *key* is what the data will be grouped on and value is the information pertinent to the analysis in the **reducer**
  - Choice of key/value pairs is critical and not arbitrary

Map task phases: **Combiner**

- Can **group data** in the map phase
- Takes the intermediate keys from the mapper and applies a user-provided method to *aggregate values* in the small scope of that one mapper
- Significantly reduces the amount of data that has to move over the network.
  - Sending ("hello", 3) requires fewer bytes than sending ("hello", 1) three times over the network

Combiner function

- No guarantees on *how many times* Hadoop will call this on a map output record
  - The combiner should, however, result in the same output from the reducer
- Combiners must be **commutative** *and* **associative**
  - Sometimes they are also called **distributive**
  - Commutative: Order of operands (5+2) = 2+5
    - Division and subtraction are not commutative
  - Associative: Order of operators 5 x (5x3) = (5x5)x3
  - Non-associative and non-commutative: Vector cross products and matrix multiplication (AB≠BA) respectively

Map task phases: **Partitioner**

- Takes the intermediate key/value pairs from the mapper (or combiner) and splits them up into **shards**, one shard per reducer
- Default: key.hashCode() % (number of reducers)
  - Randomly distributes the keyspace *evenly* over the reducers
  - But still ensures that keys with the same value in different mappers end up at the same reducer

Map task phases: **Partitioner**

- Partitioner can be customized (e.g. for sorting)
  - Changing the partitioner is rarely necessary
- The partitioned data is written to the local file system for each map and waits to be **pulled** by its respective reducer

In Hadoop a Reduce task has 4 phases

- Shuffle
- Sort
- Reducer
- Output format

Reduce task phases: **Shuffle and sort**

- Shuffle
  - Takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running
- Sort
  - Individual data pieces are then **sorted by key** into one larger data list
  - Groups equivalent keys together so that their values can be iterated over easily in the reduce task

Reduce task phases: **Shuffle and sort**

- This phase is **not customizable** and the framework handles everything automatically
- The only control a developer has is how the keys are sorted and grouped by specifying a custom `Comparator` object

Reduce task phases: **Reducer**

- Takes the grouped data as input and runs a reduce function **once per key grouping**
- The function is passed the key and an iterator over all of the values associated with that key
  - A wide range of processing can happen in this function: data can be aggregated, filtered, and combined etc.
- Once the reduce function is done, it sends zero or more key/value pairs to the final step, the output format
- N.B.: map & reduce functions will change from job to job

Reduce task phases: **Output format**

- Translates the final key/value pair from the reduce function and writes it out to a file using a record writer
- By default:
  - Separate the key and value with a tab
  - Separates records with a newline character
- Can typically be customized to provide richer output formats
  - But in the end, the data is written out to HDFS, regardless of format

MapReduce Example

Word Count

- Word count over user-submitted comments on StackOverflow
- Content of the `Text` field will be retrieved and preprocessed
  - Then count how many times we see each word
- Example record from this data set is:
  - `< row Id =" 8189677" PostId =" 6881722" Text =" Have you looked at Hadoop?" CreationDate =" 2011-07-30T07: 29: 33.343" UserId =" 831878" />`
  - This record is the 8,189,677th comment on Stack Overflow, and is associated with post number 6,881,722, and is by user number 831,878.

Driver class for the example

```
public class CommentWordCount {
  …
  public static void main( String[] args) throws Exception {
    Configuration conf = new Configuration();
    …
    Job job = new Job( conf, "StackOverflow Comment Word Count");
    job.setJarByClass( CommentWordCount.class);
    job.setMapperClass( WordCountMapper.class);
    job.setCombinerClass( IntSumReducer.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]));
    System.exit( job.waitForCompletion( true) ? 0 : 1);
  }
}
```


Waiting for the job to complete

- The `waitForCompletion()` method on Job submits the job and waits for it to finish
  - The single parameter is a flag indicating whether verbose output is generated. When `true` the job writes information about its progress to the console
- The return value of the `waitForCompletion()` method is a Boolean indicating success (`true`) or failure (`false`), which we translate into the program's exit code of 0 or 1

The Mapper class

```
public static class WordCountMapper
    extends Mapper < Object, Text, Text, IntWritable > {
  private final static IntWritable one = new IntWritable( 1);
  private Text word = new Text();

  public void map( Object key, Text value, Context context)
      throws IOException, InterruptedException {
    Map <String,String> parsed
                    =MRDPUtils.transformXmlToMap(value.toString());
    String txt = parsed.get(" Text");

    StringTokenizer itr = new StringTokenizer( txt);
    while (itr.hasMoreTokens()) {
      word.set( itr.nextToken());
      context.write( word, one);
    }
  }
}
```


Some details about the Mapper class

- Notice the type of the parent class:
  `Mapper <Object, Text, Text, IntWritable>`
- Maps to the **types** of the input key, input value, output key, and output value, respectively.
  - The key of the input in this case is not useful, so we use `Object`
  - Data coming in is `Text` (Hadoop's special `String` type) because we are reading the data as a line-by-line text document
  - Our output key and value are `Text` and `IntWritable` because we will be using the word as the key and the count as the value

The word count Reducer

```
public class IntSumReducer extends
    Reducer < Text, IntWritable, Text, IntWritable > {

    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable < IntWritable > values,
                       Context context)
       throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable val : values) {
          sum + = val.get();
       }
       result.set( sum);
       context.write( key, result);
    }
}
```


Reducer class

- As in the mapper, we specify the input and output types via the **template** parent class
  - Types correspond to the same things: input key, input value, output key, and output value
- The input key and input value data types must **match** the output key/value types from the mapper
- The output key and output value data types must match the types that the job's `FileOutputFormat` is expecting

The reduce function has a different signature from map

- Gives you an `Iterator` over values instead of just a single value
  - We iterate over all values that have that key, instead of just one at a time
- ***key*** is **very important in the reducer** of pretty much every MapReduce job
  - Unlike the input key in the map.

More about reducer outputs

- Anything passed to `context.write` will get written out to a file
- Each reducer will create one file

ANOTHER EXAMPLE (AVERAGES)

Another example with the StackOverflow [1/2]

- Given a list of user's comment determine the average comment length per-hour
- To calculate average we need two things:
  - Sum values that we want to average
  - Number of values that went into the sum

Another example with the StackOverflow [2/2]

- Reducer can do this very easily by iterating through each value in the set and adding to a running sum while keeping count
- But if you do this you cannot use the reducer as your combiner!
  - Calculating an average is not an associative operation
    - You cannot change the order of the operators
    - mean(0, 20, 10, 25, 15) = 14 BUT ..
    - mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15

Approach to ensuring code reuse at the combiner

- Mapper will output two columns of data
  - Count and average
- Reducer will multiply "count" field by the "average" field to add to a running count *and* add "count" to the running count
  - Then divide the running sum with running count
    - Output the count with the calculated average

Mapper code

```
public static class AverageMapper extends
    Mapper < Object, Text, IntWritable, CountAverageTuple > {

  private CountAverageTuple outCountAverage = new CountAverageTuple();
  public void map( Object key, Text value, Context context)
     throws IOException, InterruptedException {
    Map < String, String > parsed =
            MRDPUtils.transformXmlToMap( value.toString());
    String strDate = parsed.get(" CreationDate");
    String text = parsed.get(" Text");
    // get the hour this comment was posted in
    Date creationDate = frmt.parse( strDate);
   outHour.set( creationDate.getHours());

   outCountAverage.setCount( 1);
   outCountAverage.setAverage( text.length());

    // write out the hour with the comment length
   context.write( outHour, outCountAverage);
  }
}
```


Reducer code

```
public class AverageReducer extends Reducer < IntWritable,
CountAverageTuple, IntWritable, CountAverageTuple > {
    private CountAverageTuple result = new CountAverageTuple();

    public void
    reduce(IntWritable key, Iterable < CountAverageTuple > values,
        Context context) throws IOException, InterruptedException {
       float sum = 0; float count = 0;

       // Iterate through all input values for this key
       for (CountAverageTuple val : values) {
          sum + = val.getCount() * val.getAverage();
          count + = val.getCount();
       }
       result.setCount( count);
       result.setAverage( sum / count);
       context.write( key, result);
    }
}
```


Data flow for the average example

	Input key	Input Value	
	Hour	Count	Average
Group 1	4	1	10
	4	1	8
	4	1	21
Group 2	3	1	1
	3	1	19
	9	1	7
	9	1	12

Setting:
Combiner executes over Groups 1 and 2
DOES NOT execute on the last two rows

Combiner Output/ Reducer Input

Output key	Output Value	
Hour	Count	Average
3	2	10
4	3	13
9	1	7
9	1	12

BACKUP TASKS

Stragglers

- Machine that takes an **unusually long time** to complete a map or reduce operation
- Can slow down entire computation

How stragglers arise

- Machine with a **bad disk**
  - Frequent, correctable errors
  - Read performance drops from 30 MB/s to 1 MB/s
- Over **scheduling**
  - Many tasks executing on the same machine
  - *Competition* for CPU, memory, disk or network cycles
- **Bug** in machine initialization code
  - Processor caches may be disabled

Alleviating the problem of stragglers

- When a MapReduce operation is close to completion
- Schedule **backup** executions of *remaining* in-progress tasks
- Task completed when
  - Primary or back finishes execution
- Significantly reduces time to complete large MapReduce operations

Skipping Bad Records

- 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

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

The contents of this slide-set are based on the following references

- Jeffrey Dean and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters.* OSDI 2004: 137-150
- *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems.* 1st Edition. Donald Miner and Adam Shook. O'Reilly Media ISBN: 978-1449327170. [Chapter 1-3]