Processing Distributed Data Using MapReduce

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

MapReduce

- A framework for computation on large data sets that are fragmented and replicated across a cluster of machines.
  - spreads the computation across the machines, letting them work in parallel
  - tries to minimize the amount of data that is transferred between machines
- The original version was Google's MapReduce system.
- An open-source version is part of the Hadoop project.
  - we'll use it as part of PS 4

Sample Problem: Totalling Customer Orders

- Acme Widgets is a company that sells only one type of product.
- *Data set:* a large collection of records about customer orders
  - fragmented and replicated across a cluster of machines
  - sample record:

```
('U123', 500, '03/22/17', 'active')
```

customer id → 'U123'; amount ordered → 500; date ordered → '03/22/17'; order status → 'active'

- *Desired computation:* For each customer, compute the total amount in that customer's active orders.
- Inefficient approach: Ship all of the data to one machine and compute the totals there.

Sample Problem: Totalling Customer Orders (cont.)

- MapReduce does better using "divide-and-conquer" approach.
  - splits the collection of records into subcollections that are processed in parallel
- For each subcollection, a *mapper task* maps the records to smaller key-value pairs – in this case, (cust_id, amount active).

```
('U123', 500, '03/22/17', 'active')   → ('U123', 500)
('U456',  50, '02/10/17', 'done')     → ('U456', 0)
('U123', 150, '03/23/17', 'active')   → ('U123', 150)
('U456',  75, '03/28/17', 'active')   → ('U456', 75)
```

- These smaller pairs are distributed by cust_id to other tasks that again work in parallel.
- These *reducer tasks* combine the pairs for a given cust_id to compute the per-customer totals:

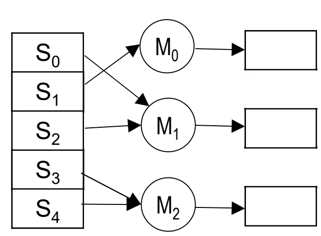
```
('U123', 500)                          ('U456', 0)
('U123', 150) → ('U123', 650)          ('U456', 75) → ('U456', 75)
```

Benefits of MapReduce

- Parallel processing reduces overall computation time.
- Less data is sent between machines.
  - the mappers often operate on local data
  - the key-value pairs sent to the reducers are smaller than the original records
  - an initial reduction can sometimes be done locally
    - example: compute local subtotals for each customer, then send those subtotals to the reducers
- It provides fault tolerance.
  - if a given task fails or is too slow, re-execute it
- The framework handles all of the hard/messy parts.
- The user can just focus on the problem being solved!

MapReduce In General: Mapping

- The system divides up the collection of input records, and assigns each subcollection S_i to a mapper task M_j.



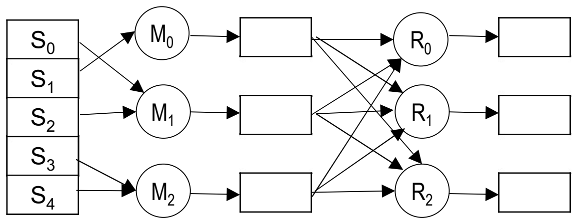
- The mappers apply a map function to each record:

```
map(k, v):    # treat record as a key-value pair
    emit 0 or more new key-value pairs (k', v')
```

  - the resulting keys and values (the *intermediate results)* can have different types than the original ones
  - the input and intermediate keys do *not* have to be unique

MapReduce In General: Reducing

- The system partitions the intermediate results by key, and assigns each range of keys to a reducer task R_k.



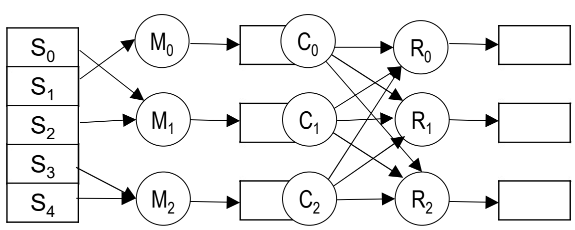
- Key-value pairs with the same key are grouped together:
  (k', v'_0), (k', v'_1), (k', v'_2) → (k', [v'_0, v'_1, v'_2, ...])
  - so that *all* values for a given key are processed together
- The reducers apply a reduce function to each (key, value-list):

```
reduce(k', [v'0, v'1, v'2, ...]):
    emit 0 or more key-value pairs (k", v")
```

  - the types of the (k", v") can be different from the (k', v')

MapReduce In General: Combining (Optional)

- In some cases, the intermediate results can be aggregated locally using *combiner* tasks C_n.



- Often, the combiners use the same reduce function as the reducers.
  - produces partial results that can then be combined
- This cuts down on the data transferred to the reducers.

Hadoop MapReduce Framework

- Implemented in Java
- It also includes other, non-Java options for writing MapReduce applications.
- In PS 4, you'll write simple MapReduce applications in Java.
- To do so, you need to become familiar with some key classes from the MapReduce API.
- We'll also review some relevant Java concepts.

Classes and Interfaces for Keys and Values

- Found in the `org.apache.hadoop.io` package
- Types used for values must implement the `Writable` interface.
  - includes methods for efficiently serializing/writing the value
- Types used for keys must implement `WritableComparable`.
  - in addition to the `Writeable` methods, must also have a `compareTo()` method that allows values to be compared
  - needed to sort the keys and create key subranges
- The following classes implement both interfaces:
  - `IntWritable` – for 4-byte integers
  - `LongWritable` – for long integers
  - `DoubleWritable` – for floating-point numbers
  - `Text` – for strings/text (encoded using UTF8)

Recall: Generic Classes

```
public class ArrayList<T> {
    private T[] items;
    …
    public boolean add(T item) {
        …
    }
    …
}
```

- The header of a generic class includes one or more *type variables*.
  - in the above example: the variable `T`
- The type variables serve as placeholders for actual data types.
- They can be used as the types of:
  - fields
  - method parameters
  - method return types

Recall: Generic Classes (cont.)

```
public class ArrayList<T> {
    private T[] items;
    …
    public boolean add(T item) {
        …
    }
    …
}
```

- When we create an instance of a generic class, we specify types for the type variables:

  ```
  ArrayList<Integer> vals = new ArrayList<Integer>();
  ```

  - `vals` will have an `items` field of type `Integer[]`
  - `vals` will have an add method that takes an `Integer`
- We can also do this when we create a subclass of a generic class:

  ```
  public class IntList extends ArrayList<Integer> {
      ...
  ```

Mapper Class

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
```

KEYIN, VALUEIN: type variables for the (key, value) pairs given to the mapper

KEYOUT, VALUEOUT: type variables for the (key, value) pairs produced by the mapper

- the principal method:

```
void map(KEYIN key, VALUEIN value, Context context)
```

- To implement a mapper:
  - extend this class with appropriate replacements for the type variables; for example:

```
class MyMapper
   extends Mapper<Object, Text, Text, IntWritable>
```

  - override `map()`

Reducer Class

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
```

KEYIN, VALUEIN: type variables for the (key, value) pairs given to the reducer

KEYOUT, VALUEOUT: type variables for the (key, value) pairs produced by the reducer

- the principal method:

```
void reduce(KEYIN key, Iterable<VALUEIN> values,
            Context context)
```

- To implement a reducer:
  - extend this class with appropriate replacements for the type variables
  - override `reduce()`

`Context` Objects

- Both `map()` and `reduce()` are passed a `Context` object:

```
void map(KEYIN key, VALUEIN value, Context context)
void reduce(KEYIN key, Iterable<VALUEIN> values,
            Context context)
```

- Allows `Mappers` and `Reducers` to communicate with the `MapReduce` framework.

- Includes a `write()` method used to output (key, value) pairs:

```
void write(KEYOUT key, VALUEOUT value)
```

Example

```
class MyMapper extends Mapper<Object, Text,
                       LongWriteable, IntWritable>
```

Which of these is the correct header for the `map` method?

A. `void map(LongWriteable key, IntWritable value, Context context)`

B. `void map(Text key, LongWriteable value, Context context)`

C. `void map(Object key, IntWriteable value, Context context)`

D. `void map(Object key, Text value, Context context)`

Example 1: Birth-Month Counter

- ***The data:*** text file(s) containing person records that look like this

  `id,name,dob,email`

  where dob is in the form `yyyy-mm-dd`

- ***The problem:*** Find the number of people born in each month.

Example 1: Birth-Month Counter (cont.)

- `map` should:
  - extract the month from the person's dob
  - emit a single key-value pair of the form `(`month string`, 1)`

```
111,Alan Turing,1912-06-23,al@aol.com          → ("06", 1)
234,Grace Hopper,1906-12-09,gmh@harvard.edu    → ("12", 1)
444,Ada Lovelace,1815-12-10,ada@1800s.org      → ("12", 1)
567,Howard Aiken,1900-03-08,aiken@harvard.edu  → ("03", 1)
777,Joan Clarke,1917-06-24,joan@bletchley.org  → ("06", 1)
999,J. von Neumann,1903-12-28,jvn@princeton.edu→ ("12", 1)
```

- The intermediate results are distributed by key to the reducers.

- `reduce` should:
  - add up the 1s for a given month
  - emit a single key-value pair of the form `(`month string, total`)`

```
("06", [1, 1])       → ("06", 2)
("12", [1, 1, 1])    → ("12", 3)
("03", [1])          → ("03", 1)
```

Mapper for Example 1

```
public class BirthMonthCounter {
    public static class MyMapper
        extends Mapper<Object, Text, Text, IntWritable>
```

- For data obtained from text files, the Mapper's inputs will be key-values pairs in which:
  - value = a single line from one of the files (a `Text` value)
  - key = the location of the line in the file (a `LongWritable`)
    - however, we use the `Object` type for the key because we ignore it, and thus we don't need any `LongWritable` methods
- The `map` method will output pairs in which:
  - key = a month string (use `Text` for it)
  - value = 1 (use `IntWritable`)

Mapper for Example 1 (cont.)

```
public class BirthMonthCounter {
    public static class MyMapper
        extends Mapper<Object, Text, Text, IntWritable>
    {
        public void map(Object key, Text value,
                        Context context)
        {
            String record = value.toString();
            // code to extract month string goes here
            context.write(new Text(month),
                          new IntWritable(1));
        }
    }
    ...
}
```

Splitting a String

- The `String` class includes a method named `split()`.
  - breaks a string into component strings
  - takes a parameter indicating what delimiter should be used when performing the split
  - returns a `String` array containing the components
- Example:

```
String sentence = "How now brown cow?";
String[] words = sentence.split(" ");
System.out.println(words[0]);
System.out.println(words[3]);
System.out.println(words.length);
```

would output:

Processing an Input Record in map

```
void map(Object key, Text value, Context context)
```

- Recall: `value` is a `Text` object representing one record.
  - for Example 1, it looks like:

    `111,Alan Turing,1912-06-23,al@aol.com`

- To extract the month string:
  - use the `toString()` method to convert `Text` to `String`:

    ```
    String line = value.toString();
    ```

  - split `line` on the commas to get the fields:

    ```
    String[] fields = line.split(",");
    ```

  - similarly, split the date field on the hyphens to get its components
  - could we just split `line` on the hyphens?

Reducer for Example 1

```
public static class MyMapper
    extends Mapper<Object, Text, Text, IntWritable>
{
    …
}

public static class MyReducer
    extends Reducer<Text, IntWritable,
                    Text, LongWritable>
{
   public void reduce(Text key,
        Iterable<IntWritable> values, Context context)
   {
       // code to add up the list of 1s goes here
       context.write(key, new LongWritable(total));
   }
   ...
```

- Use `LongWritable` to avoid overflow with large totals.

Processing the List of Values in `reduce`

```
void reduce(Text key, Iterable<IntWritable> values,
            Context context)
```

- Use a *for-each* loop. In this case:
  ```
  for (IntWritable val : values)
  ```
- More generally, if `values` is of type `Iterable<T>` :
  ```
  for (T val : values)
  ```
- To extract the underlying value from most `Writable` objects, use the `get()` method:
  ```
  int count = val.get();   // val is IntWritable
  ```
- However, `Text` doesn't have a get() method.
  - use `toString()` instead (see earlier notes)

Reducer for Birth-Month Counter

```
public class BirthMonthCounter {
 ...
 public static class MyReducer
     extends Reducer<Text, IntWritable,
                     Text, LongWritable>
 {
    public void reduce(Text key,
         Iterable<IntWritable> values, Context context)
    {
        long total = 0;
        for (IntWritable val : values) {
            total += val.get()
        }

        context.write(key, new LongWritable(total));
    }
    ...
```

- Use `long` and `LongWritable` to avoid overflow.

`Job` Objects

- We use a `Job` object to:
  - provide information about our MapReduce job, such as:
    - the name of the Mapper class
    - the name of the Reducer class
    - the types of values produced by the job
    - the format of the input to the job
  - execute the job
- We'll give you a template for the necessary method calls.

Configuring and Running the Job

```
public class BirthMonthCounter {
    public static class MyMapper extends... {
        ...
    public static class MyReducer extends... {
        ...
    public static void main(String args)
      throws Exception {
        // code to configure and run the job
    }
}
```

Configuring and Running the Job

```
public static void main(String[] args)
  throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "birth month");
    job.setJarByClass(BirthMonthCounter.class);

    job.setMapperClass(MyMapper.class);
    job.setReducerClass(MyReducer.class);

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

    // type for mapper's output value,
    // because its not the same as the reducer's
    job.setMapOutputValueClass(IntWritable.class);

    job.setInputFormatClass(TextInputFormat.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.waitForCompletion(true);
}
```

Example 2: Month with the Most Birthdays

- ***The data:*** same as Example 1. Records of the form

  `id,name,dob,email`

  where dob is in the form `yyyy-mm-dd`

- ***The problem:*** Find the month with the most birthdays.

Example 2: Month with the Most Birthdays (cont.)

- `map` should behave as before:

```
111,Alan Turing,1912-06-23,al@aol.com          → ("06", 1)
234,Grace Hopper,1906-12-09,gmh@harvard.edu    → ("12", 1)
444,Ada Lovelace,1815-12-10,ada@1800s.org      → ("12", 1)
```

- `reduce` needs to:
  - add up the 1s for a given month

    ```
    ("06", [1, 1])      → ("06", 2)
    ("12", [1, 1, 1])   → ("12", 3)
    ("03", [1])         → ("03", 1)
    ```
  - determine which month has the largest total
  - ***but...***
    - there can be multiple reducer tasks, each of which handles one subset of the months
    - each reducer can only determine the largest month in its subset
  - ***the solution:*** a *chain* of two MapReduce jobs

Example 2: Chaining Jobs

- First job = count birth months as we did in Example 1
  - map1: person record → (birth month, 1)
  - reduce1: (birth month, [1, 1, ...]) → (birth month, total)

- The second job processes the results of the first job!
  - map2: (birth month, total) → (**c**, **(birth month, total)**)
    - output **key c** = an arbitrary ***constant***, used for *all* k-v pairs
    - output **value** = a pairing of a birth month and its total

```
("06", 2) → ("month sum", "06,2")
("12", 3) → ("month sum", "12,3")
("03", 1) → ("month sum", "03,1")
```

    - because there is only one output key, there is only one reducer task!

  - reduce2: find the month with the most birthdays

```
("month sum", ["06,2", "12,3", "03,1"]) → ("12", 3)
```

Example 2: Chaining Jobs (cont.)

```
public class MostBirthdaysMonth {
    public static class MyMapper1 extends... {
        ...
    }
    public static class MyReducer1 extends... {
        ...
    }
    public static class MyMapper2 extends... {
        ...
    }
    public static class MyReducer2 extends... {
        ...
    }

    public static void main(String[] args) throws... {
        ...
}
```

Configuring and Running a Chain of Jobs

```
public static void main(String args)
  throws Exception {
    Configuration conf = new Configuration();
    Job job1 = Job.getInstance(conf, "birth month");
    job1.setJarByClass(MostBirthdaysMonth.class);
    job1.setMapperClass(MyMapper1.class);
    job1.setReducerClass(MyReducer1.class);
    ...
    FileInputFormat.addInputPath(job1, new Path(args[0]));
    FileOutputFormat.setOutputPath(job1, new Path(args[1]));
    job1.waitForCompletion(true);

    Job job2 = Job.getInstance(conf, "max month");
    job2.setJarByClass(MostBirthdaysMonth.class);
    job2.setMapperClass(MyMapper2.class);
    job2.setReducerClass(MyReducer2.class);
    ...
    FileInputFormat.addInputPath(job2, new Path(args[1]));
    FileOutputFormat.setOutputPath(job2, new Path(args[2]));
    job2.waitForCompletion(true);
}
```

Structure of the Java Files

- In theory, we could use multiple Java files for each problem:
  - one file for the program as a whole
  - one file for the Mapper class, one for the Reducer class, etc.
- Instead, we'll put all of the classes in the same file by using *static nested classes:*

```
public class MyProblem {
    public static class MyMapper extends ... {
         ...
    }
    public static class MyReducer extends ... {
         ...
    }
```

- Unlike an inner class (aka a *non*-static nested class), static nested classes do *not* depend on their outer class.
  - they are equivalent to an outer class from another file
  - allows the MapReduce system to instantiate them