Processing Distributed Data Using MapReduce, Part I

Computer Science E-66
Harvard University
David G. Sullivan, Ph.D.

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    amount ordered    date ordered    order status

- **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 (cust_id, amount active) pairs.
  - (`'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) ➾ (`'U123', 650)
  - (`'U456', 0) ➾ (`'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:
  
  \[
  \text{map}(k, v): \quad \# \text{treat record as a key-value pair} \\
  \text{emit 0 or more new key-value pairs } (k', v')
  \]

  • the resulting keys and values (the \textit{intermediate results}) can have different types than the original ones
  • the input and intermediate keys do \textit{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:
  
  $$(k', v'_0), (k', v'_1), (k', v'_2) \rightarrow (k', [v'_0, v'_1, v'_2, \ldots])$$

- so that all values for a given key are processed together

- The reducers apply a reduce function to each (key, value-list):
  
  $$\text{reduce}(k', [v'_0, v'_1, v'_2, \ldots])$$

  - 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.
Apache Hadoop

- An open-source framework for processing large data sets
- Can scale from single servers to large clusters of machines
- Includes a number of components:
  - HDFS: a distributed file system
  - a module for job scheduling and resource management across a cluster
    - coordinates the mappers and reducers
  - many others!

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 `Writable` 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

```java
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.)

```java
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:
  ```java
  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:
  ```java
  public class IntList extends ArrayList<Integer> {
    ...
  }
  ```

Mapper Class

```java
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    type variables for the (key, value) pairs given to the mapper
    type variables for the (key, value) pairs produced by the mapper
    type variables

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

    the principal method:

    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

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

    // type variables for the (key, value) pairs given to the reducer
    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:
  ```java
  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:
  ```java
  void write(KEYOUT key, VALUEOUT value)
  ```
Example

class MyMapper extends Mapper<Object, Text, LongWritable, 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

class MyMapper extends Mapper<Object, Text, LongWritable, 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)

• the Context object has:
  void write(LongWriteable key, IntWritable value)

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)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>111,Alan Turing,1912-06-23,<a href="mailto:al@aol.com">al@aol.com</a></td>
<td>(&quot;06&quot;, 1)</td>
</tr>
<tr>
<td>234,Grace Hopper,1906-12-09,<a href="mailto:ghm@harvard.edu">ghm@harvard.edu</a></td>
<td>(&quot;12&quot;, 1)</td>
</tr>
<tr>
<td>444,Ada Lovelace,1815-12-10,<a href="mailto:ada@1800s.org">ada@1800s.org</a></td>
<td>(&quot;12&quot;, 1)</td>
</tr>
<tr>
<td>567,Howard Aiken,1900-03-08,<a href="mailto:aiken@harvard.edu">aiken@harvard.edu</a></td>
<td>(&quot;03&quot;, 1)</td>
</tr>
<tr>
<td>777,Joan Clarke,1917-06-24,<a href="mailto:joan@bletchley.org">joan@bletchley.org</a></td>
<td>(&quot;06&quot;, 1)</td>
</tr>
<tr>
<td>999,J.von Neumann,1903-12-28,<a href="mailto:jvn@princeton.edu">jvn@princeton.edu</a></td>
<td>(&quot;12&quot;, 1)</td>
</tr>
</tbody>
</table>

- 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)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&quot;06&quot;, [1, 1])</td>
<td>(&quot;06&quot;, 2)</td>
</tr>
<tr>
<td>(&quot;12&quot;, [1, 1, 1])</td>
<td>(&quot;12&quot;, 3)</td>
</tr>
<tr>
<td>(&quot;03&quot;, [1])</td>
<td>(&quot;03&quot;, 1)</td>
</tr>
</tbody>
</table>

Mapper for Example 1

```java
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) {
            // code to extract month string goes here
            context.write(new Text(month),
                          new IntWritable(1));
        }
    }
    ...
}

We'll finish this example next time!