PART 1. LARGE SCALE DATA ANALYSIS USING MAPREDUCE

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

FAQs [1/4]
- 1/23 Add Without Override Deadline
- 3/20 Course Withdrawal Period Ends
- Accommodation request, honor student
  - Contact me by Jan 25 2017

FAQs [2/4]
- Term Project
  - Discussion board is ready on Canvas
  - Are you looking for teammate(s)? Post your message.
    - Your name
    - Strength
    - Some idea about your topic
  - Tips for selecting your term project
    - Consider the feasibility
    - Do you have a good data? (volume, area, and quality)
    - Do you have enough man power? Is your project for 2 members or 3 members?
    - Be specific
  - Computing resources
    - Amazon AWS: public datasets and computing facility
    - AWS education grant

FAQs [3/4]
- Programming Assignment 1
  - Due: Feb. 5
  - Help session: Jan. 27 Friday
    - Time and location: TBA
    - The link to the video clip will be posted on the class web
    - Please try to attend and ask questions!
    - This is not mandatory
- GTA Office hours (in CSB120)
  - Tuesday 8:00AM ~ 10:00AM
  - Thursday 10:00AM ~ noon

FAQs [4/4]
- Access to the article with copyrights
  - Use the CSU network
- Slides?
  - A draft of the lecture material is posted before the class
  - After the class, the finalized material is posted
- CS535 Big Data?
This material is developed based on,

  - Download this chapter from the CS435 schedule page


- MapReduce Design Patterns, Donald Miner and Adam Shook, O'Reilly, 2013

What is MapReduce?

MapReduce

- MapReduce is inspired by the concepts of map and reduce in Lisp.

  “Modern” MapReduce
  - Developed within Google as a mechanism for processing large amounts of raw data.
  - Crawled documents or web request logs
  - Distributes these data across thousands of machines
  - Same computations are performed on each CPU with different datasets

Mapper

- Mapper maps input key/value pairs to a set of intermediate key/value pairs
  - Maps are the individual tasks that transform input records into intermediate records
  - The transformed intermediate records do not need to be of the same type as the input records
  - A given input pair may map to zero or many output pairs
  - The Hadoop MapReduce framework spawns one map task for each InputSplit generated by the InputFormat for the job

Reducer

- Reducer reduces a set of intermediate values which share a key to a smaller set of values

- Reducer has 3 primary phases
  - Shuffle, sort and reduce
  
  Shuffle
  - Input to the reducer is the sorted output of the mappers
  - The framework fetches the relevant partition of the output of all the mappers via HTTP
  - Sort
  - The framework groups input to the reducer by keys
Example 1: WordCount

For text files stored under /usr/joe/wordcount/input, count the number of occurrences of each word.

How do files and directory look?

$ bin/hadoop dfs -ls /usr/joe/wordcount/input/

/ usr/joe/wordcount/input/file01
/usr/joe/wordcount/input/file02

$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file01

Hello World, Bye World!

$ bin/hadoop dfs -cat /usr/joe/wordcount/input/file02

Hello Hadoop, Goodbye to hadoop.

Example 1: WordCount

Mappers
1. Read a line
2. Tokenize the string
3. Pass the <key, value> output to the reducer

Reducers
1. Collect <key, value> pairs sharing same key
2. Aggregate total number of occurrences

What do you have to pass from the Mappers?

public static class Map extends 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);
    }
  }
}

Example 1: WordCount

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 val : values) {
      sum += val.get();
    }
    context.write(key, new IntWritable(sum));
  }
}
MapReduce Example 2

NCDC data example
- A national climate data center record
- Find the maximum temperature of a year (1900 ~ 1999)

```
0057
32130 USAF weather station identifier
99999 NAMD weather station identifier
19500101 observation date
0350 observation time
3127 latitude (degrees x 1000)
28783 longitude (degrees x 1000)
FM-12
0171 elevation (meters)
99999
V000
320 wind direction (degrees)
1 quality code
```

The first entries for 1990
```
% ls raw/ 1990 * | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

Analyzing the data with Unix Tools (1/2)
- A program for finding the maximum recorded temperature by year from NCDC weather records
```
#!/usr/bin/env bash
for year in all/*
do
echo -n " basename $ year .gz |" "\t"
gunzip $ year |
awk '{ temp = substr( $0, 88, 5) + 0;
q = substr( $0, 93, 1);
if (temp > max)
max = temp }'
END { print max }'
done
```

Analyzing the data with Unix Tools (2/2)
- The script loops through the compressed year files
- Printing the year
- Processing each file using awk
  - Extracts two fields
  - Air temperature and the quality code
- Check if it is greater than the maximum value seen so far
```
% ./ max_temperature.sh
1901 317
1902 244
1903 289
1904 256
1905 283
```

Results?
- The complete run for the century took 42 minutes
- To speed up the processing
  - We need to run parts of the program in parallel
  - Process different years in different processes
- What will be the problems?
Challenges

- Dividing the work into equal-size pieces
  - Data size per year?

- Combining the results from independent processes
  - Combining results and sorting by year?

- You are still limited by the processing capacity of a single machine (the worst one)!

Map and Reduce

- MapReduce works by breaking the processing into two phases
  - The map phase
  - The reduce phase

- Each phase has key-value pairs as input and output

- Programmers should specify
  - Types of input/output key-values
  - The map function
  - The reduce function

Visualizing the way the MapReduce works

Sample lines of input data

- These lines are presented to the map function as the key-value pairs
  - The keys are the line offsets within the file (optional)

- Reduce function iterates through the list and pick up the maximum reading
  - This is the final output

Comparison with other systems

- MPI vs. MapReduce
  - MapReduce tries to collocate the data with the compute node
  - Data access is fast
  - Data is local

- Volunteer computing vs. MapReduce
  - SETI@home
    - Using donated CPU time
MapReduce Example 3

Relational-Algebra style queries [1/5]
- Large relation can be stored as a file in a DFS (Distributed File System)
  - We will discuss DFS next class
- The data queries (e.g., SQL style) can be processed without storing dataset in a SQL database using MapReduce
  - Selection
  - Apply a condition \( C \) to each record in the relation and provide as output only those records that satisfy \( C \)
  - Selection
    
<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>220.4.5.20</td>
<td>160.33.1.3</td>
</tr>
<tr>
<td>220.4.5.20</td>
<td>100.33.1.5</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>79.45.66.9</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>100.33.1.5</td>
</tr>
<tr>
<td>100.33.1.5</td>
<td>79.45.66.9</td>
</tr>
</tbody>
</table>

Relational-Algebra style queries [2/5]
- Projection
  - Produces partial record from each record that contains only selected attributes
  - Union/Intersection and Difference

Data File 1 of day X

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>121.42.15.2</td>
<td>160.33.1.3</td>
</tr>
<tr>
<td>220.4.5.21</td>
<td>100.33.1.5</td>
</tr>
<tr>
<td>160.33.1.30</td>
<td>79.45.66.9</td>
</tr>
<tr>
<td>100.33.1.5</td>
<td>79.45.66.9</td>
</tr>
</tbody>
</table>

Data File 2 of day Y

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>220.4.5.20</td>
<td>160.33.1.3</td>
</tr>
<tr>
<td>220.4.5.20</td>
<td>100.33.1.5</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>79.45.66.9</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>100.33.1.5</td>
</tr>
</tbody>
</table>

Relational-Algebra style queries [3/5]
- Union/Intersection and Difference
- Set operations

Data File 1 of day X

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>121.42.15.2</td>
<td>160.33.1.3</td>
</tr>
<tr>
<td>220.4.5.21</td>
<td>100.33.1.5</td>
</tr>
<tr>
<td>160.33.1.30</td>
<td>79.45.66.9</td>
</tr>
<tr>
<td>100.33.1.5</td>
<td>79.45.66.9</td>
</tr>
</tbody>
</table>

Data File 2 of day Y

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>220.4.5.20</td>
<td>160.33.1.3</td>
</tr>
<tr>
<td>220.4.5.20</td>
<td>100.33.1.5</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>79.45.66.9</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>100.33.1.5</td>
</tr>
</tbody>
</table>

Relational-Algebra style queries [4/5]
- Natural Join
  - Given two datasets, compare each pair of records from each dataset
  - If the records agree on all the attributes that are common between schemas, then produce a new record that has all of the attributes in either schema

Data File 1: links

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>220.4.5.20</td>
<td>160.33.1.3</td>
</tr>
<tr>
<td>220.4.5.20</td>
<td>100.33.1.5</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>79.45.66.9</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>100.33.1.5</td>
</tr>
</tbody>
</table>

Data File 2: users

<table>
<thead>
<tr>
<th>Source IP</th>
<th>Destination IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>220.4.5.20</td>
<td>160.33.1.3</td>
</tr>
<tr>
<td>220.4.5.20</td>
<td>100.33.1.5</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>79.45.66.9</td>
</tr>
<tr>
<td>160.33.1.3</td>
<td>100.33.1.5</td>
</tr>
</tbody>
</table>
Relational-Algebra style queries [5/5]

- Grouping and Aggregation
  - Grouping
    - Given a dataset \( R \), partition its records according to their values in one set of attributes \( G \)
  - Aggregation
    - Then for each group, aggregate the values
    - SUM, COUNT, AVG, MIN and MAX

Computing Selections by MapReduce

- The Map function
  - For each record \( t \) in dataset \( A \), test if it satisfies \( C \). If so, produce the key-value pair \( (t, t) \)
  - Both of the key and values are \( t \)

- The Reduce function
  - It simply passes each key-value pair to the output

Computing Projections by MapReduce

- The Map function
  - For each record \( t \) in dataset \( A \), construct a record \( t' \) by eliminating all of the attributes those are not needed
  - Output the key-value pair \( (t', t') \)

- The Reduce function
  - For each key \( t' \) produced by any of the Map tasks, there will be one or more key-value pairs \( (t', t') \)
  - Eliminate duplicates
    - \( (t', t', t', t', \ldots) \rightarrow (t', t') \)

Computing Union by MapReduce

- The Map function
  - For each record \( t \) in dataset \( A \), produce the key-value pair \( (t, t) \)

- The Reduce function
  - For the shuffled pairs \( (t, t, t, \ldots) \)
  - Returns \( (t, t) \) as the result of eliminating duplications
Computing Union by MapReduce

- The Map function
  - Turn each input record \( t \) into a key-value pair \((t, t)\)

- The Reduce function
  - Associated with each key \( t \) there will be either one or two values

Computing Intersection by MapReduce

- The Map function
  - Turn each record \( t \) into a key-value pair \((t, t)\)

- The Reduce function
  - If key \( t \) has another value list \([t, t]\), then produce \((t, t)\)
  - Otherwise nothing

Computing Difference by MapReduce

- The Map function
  - For record \( t \) in dataset A, produce key-value pair \((t, A)\), and for a record \( t \) in dataset B, produce key-value pair \((t, B)\)

- The Reduce function
  - Merge the key-value pair
    - Concatenate values for each key
    - e.g. \((t_1, [A, B]) \rightarrow (t_1, [A, B])\)
    - For each key \( t \), if the associated value list is \( A \), then produce \((t, t)\)
    - Otherwise (the associated value list is \( AB \) or \( BA \) or \( B \), produce nothing)
Computing natural join by MapReduce (1/3)

- Joining dataset A and dataset B
  - Dataset A has attributes A1, A2, and A3
  - Dataset B has attributes A3, A4, and A5
- We must find records that agree on their A3 attributes
  - The third attribute from the dataset A and the first attribute of dataset B

Computing natural join by MapReduce (2/3)

Example-4: Implementation of Natural Join using MapReduce (3/2)

Map function
- For each record (A1, A2, A3) in dataset A, produce the key-value pair (A3, (“A”, (A1, A2)))
- For each record (A3, A4, A5) in dataset B, produce the key-value pair (A3, (“B”, (A4, A5)))

Reduce function
- For all of the pairs with same key:
  - Construct all pairs consisting of one with first component “A” and the other with first component “B”. e.g. (A3, (“A”, (A1, A2))) and (A3, (“B”, (A4, A5)))
  - Produce a record (A1, A2, A3, A4, A5)

Computing natural join by MapReduce (3/3)

- The Map function
  - For each record (A1, A2, A3) of Dataset A, produce the key-value pair (A3, (“A”, (A1, A2)))
  - For each record (A3, A4, A5) of Dataset B, produce the key-value pair (A3, (“B”, (A4, A5)))

- The Reduce function
  - Each key value A3 will be associated with a list of pairs that are either of the form (“A”, (A1, A2)) or (“B”, (A4, A5))
  - Construct all pairs consisting of one with first component dataset A and the other with first component dataset B: (A3, (“A”, (A1, A2))) and (A3, (“B”, (A4, A5)))
  - Merge the record and return the value
    - (A1, A2, A3, A4, A5)

Questions?