PART 1. LARGE SCALE DATA ANALYTICS

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

FAQs

• PA1 has been posted
  • Recitation: 9/13
  • Read the description and attend
  • Individual submission (No team submission)
  • If you have not been assigned the “port range”, please contact the GTA immediately

• AWS’ Elastic MapReduce (EMR) is not allowed for your PAs
• You can use it for the Term Project
Topics

- MapReduce Design Pattern II. Filtering Patterns
- MapReduce Design Pattern III. Data Organization Patterns

Part 1. Large Scale Data Analytics

Design Pattern 2: Filtering Patterns
MapReduce Design Patterns II: Filtering Patterns

2. Bloom Filter

Filtering Pattern 2. Bloom Filter

- Checking the membership of a set

- Known uses
  - Removing most of the non-membership values
  - Prefiltering a data set for an expensive set membership check
Building a Bloom filter

• \( m \)
  • The number of bits in the filter

• \( n \)
  • The number of members in the set

• \( p \)
  • The desired false positive rate

• \( k \)
  • The number of different hash functions used to map some element to one of the \( m \) bits with a uniform random distribution

Applying a Bloom filter

• Is 5 part of set \( T \)?
  • \( h1(5), h2(5), h3(5)'th \) bits are 1
    • 5 is probably a part of set \( T \)

<table>
<thead>
<tr>
<th>After encoding 5, 10 and 15</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check ( h1(5) = 7 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Check ( h2(5) = 5 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Check ( h3(5) = 5 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Applying a Bloom filter

- Is 9 part of set \( T \)?
- \( h_1(9), h_2(9), h_3(9) \)
- 9 is *not* a part of set \( T \)

<table>
<thead>
<tr>
<th>After encoding 5, 10 and 15</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check ( h_1(9) = 3 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Check ( h_2(9) = 5 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Check ( h_3(9) = 1 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Applying a Bloom filter

- Is 7 part of set \( T \)?
- \( h_1(7), h_2(7), h_3(7) \) th bits what are 1
  - 7 is *probably* a part of set \( T \)

<table>
<thead>
<tr>
<th>After encoding 5, 10 and 15</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check ( h_1(7) = 7 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Check ( h_2(7) = 1 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Check ( h_3(7) = 7 )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### BUILDING A BLOOM FILTER

(Repeat this step for 10 and 15)

- \( m = 8 \), \( n = 3 \) target set \( T = \{ 5, 10, 15 \} \)
- \( k = 3 \)
  - \( h_1(x) = 3x \mod 8 \)
  - \( h_2(x) = (2x + 3) \mod 8 \)
  - \( h_3(x) = x \mod 8 \)
  - \( h_1(5) = 7, \ h_2(5) = 5, \ h_3(5) = 5 \)

#### Initial Bloom Filter

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

After \( h_1(5) = 7 \):

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

After \( h_2(5) = 5 \):

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

After \( h_3(5) = 5 \):

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

### FALSE POSITIVE RATE

- A bloom filter with an optimal value for \( k \) and 1% error rate only needs **9.6 bits per key**.
- Add **4.8 bits/key** and the error rate decreases by **10 times**
- 10,000 words with 1% error rate and 7 hash functions
  - \(~12KB of memory\)
- 10,000 words with 0.1% error rate and 11 hash functions
  - \(~18KB of memory\)
How big should I make my Bloom Filter?

• Try various values of $k$ and $m$
  • To achieve target false–positive rate $(1 - e^{-kn/m})^k$

• Then, how many hash functions should I use?
  • The more hash functions you have
    • the slower your bloom filter
    • The quicker it fills up
  • If you have few hash functions
    • Too many false positives

• Given an $m$ and an $n$, the optimal value of $k$
  • $(m/n)\ln(2)$

Use cases

• Representing a very large dataset
• Reduce queries to external database
• Google BigTable
Downsides

• False positive rate

• **Hard to remove elements** from a Bloom filter set
  • Setting bits to zero
    • Often more than one element hashed to a particular bits
  • Use a Counting Bloom filter
    • Instead of bit, it stores count of occurrences
    • Requires more memory

Building Bloom Filter with MapReduce

[Diagram of MapReduce process for building Bloom Filter]
Running Bloom Filter with MapReduce

Bloom Filtering mapper (checking) [1/2]

```java
public static class BloomFilteringMapper extends Mapper < Object, Text, Text, NullWritable > {
    private BloomFilter filter = new BloomFilter();
    protected void setup(Context context) throws IOException, InterruptedException {
        // Get file from the DistributedCache
        URI[] files=DistributedCache.getCacheFiles(context.getConfiguration());
        System.out.println("Reading Bloom filter from: "+files[0].getPath());
        // Open local file for read.
        DataInputStream strm =
                new DataInputStream(new FileInputStream(files[0].getPath()));
        // Read into our Bloom filter.
        filter.readFields(strm);
        strm.close();
    }
```
Bloom Filtering mapper (Checking) [2/2]

```java
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    Map < String, String > parsed=transformXmlToMap(value.toString());
    // Get the value for the comment
    String comment = parsed.get("Text");
    StringTokenizer tokenizer = new StringTokenizer(comment);
    // For each word in the comment
    while (tokenizer.hasMoreTokens()) {
        // If the word is in the filter,
        // output the record and break
        String word = tokenizer.nextToken();
        if (filter.membershipTest(new Key(word.getBytes())))
        {
            context.write(value, NullWritable.get());
            break;
        }
    }
}
```
Filtering Pattern 3. **Top 10**

- Retrieves a relatively small number \((\text{top } K)\) of records, according to a ranking scheme in your dataset, no matter how large the data

- E.g. generate top 10 clients
- E.g. generate top 10 clients per zip code ..
- E.g. generate top 10 clients per day ..
- E.g. generate top 10 clients per age ..

- Known uses
  - Outlier analysis
  - Selecting interesting data
  - Data summarization
  - Catchy dashboards

The structure of Top 10 pattern with a single group
The structure of Top 10 pattern with multiple groups

```java
public static class TopTenMapper extends Mapper < Object, Text, NullWritable, Text > {
    // Stores a map of user reputation to the record
    private TreeMap < Integer, Text > repToRecordMap = new TreeMap < Integer, Text > ();
    public void map( Object key, Text value, Context context) throws IOException, InterruptedException {
        Map < String, String > parsed = transformXmlToMap( value.toString());
        String userId = parsed.get("Id");
        String reputation = parsed.get("Reputation");
        // Add this record to our map with the reputation as the key
        repToRecordMap.put(Integer.parseInt(reputation), new Text(value));
        // If we have more than ten records, remove the one with the lowest rep
        // As this tree map is sorted in descending order, the user with
        // the lowest reputation is the last key.
        if (repToRecordMap.size() > 10) {
            repToRecordMap.remove(repToRecordMap.firstKey());
        }
    }
}
```
Mapper [2/2]

```java
protected void cleanup(Context context)
    throws IOException, InterruptedException {
    // Output our ten records to the reducers with a null key
    for (Text t : repToRecordMap.values()) {
        context.write(NullWritable.get(), t);
    }
}
```

**Note:** `setup()` and `cleanup()` are called for each Mapper and Reducer. So, if there are 20 mappers running (10,000 inputs each), the setup/cleanup will be called 20 times.

Example:
```java
public void run(Context context) throws IOException, InterruptedException {
    setup(context);
    try { while (context.nextKey()) {
        reduce(context.getCurrentKey(), context.getValues(), context); }
    } finally { cleanup(context); }
}
```

Reducer

```java
public static class TopTenReducer extends Reducer<NullWritable, Text, NullWritable, Text> {
    // Stores a map of user reputation to the record
    // Overloads the comparator to order the reputations in descending order
    private TreeMap<Integer, Text> repToRecordMap = new TreeMap<Integer, Text>();
    public void reduce(NullWritable key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        for (Text value : values) {
            Map<String, String> parsed = transformXmlToMap(value.toString());
            repToRecordMap.put(Integer.parseInt(parsed.get("Reputation")), new Text(value));
            // If we have more than ten records, remove the one with the lowest rep
            // As this tree map is sorted in descending order, the user with
            // the lowest reputation is the last key.
            if (repToRecordMap.size() > 10) {
                repToRecordMap.remove(repToRecordMap.firstKey());
            }
        }
        for (Text t : repToRecordMap.descendingMap().values()) {
            // Output our ten records to the file system with a null key
            context.write(NullWritable.get(), t);
        }
    }
```
Filtering Pattern 4. **Distinct**

- You have data that contains *similar records* and you want to find a *unique set* of values
  - e.g. Generate a list of distinct user ids
### Mapper Code

```java
public static class DistinctUserMapper extends Mapper < Object, Text, Text, NullWritable > {
    private Text outUserId = new Text();
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        Map < String, String > parsed = transformXmlToMap(value.toString());
        // Get the value for the UserId attribute
        String userId = parsed.get("UserId");
        // Set our output key to the user's id
        outUserId.set(userId);
        // Write the user's id with a null value
        context.write(outUserId, NullWritable.get());
    }
}
```

### Reducer code

```java
public static class DistinctUserReducer extends Reducer < Text, NullWritable, Text, NullWritable > {
    public void reduce(Text key, Iterable < NullWritable > values, Context context) throws IOException, InterruptedException {
        // Write the user's id with a null value
        context.write(key, NullWritable.get());
    }
}
```
Design Pattern 3: Data Organization Patterns

Part 1. Large Scale Data Analytics

Data Organization Patterns

• Reorganizing data
  • Partitioning, Sharding and Sorting

1. “Structured” to “hierarchical” pattern
2. Partitioning and binning patterns
3. Total order sorting
MapReduce Design Patterns III: Data Organization Patterns

1. "Structured" to "hierarchical" pattern

Structured to Hierarchical

- Creates new records from data with a very different structure
  - e.g. transforms your row-based data to a hierarchical format such as JSON or XML

- Organizing StackOverflow data

```
<table>
<thead>
<tr>
<th>Table about Posts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table about Comments</td>
</tr>
</tbody>
</table>
```

```
RDBMS outputs
```

```
Hierarchical Format (JSON or XML)
```

```
Posts
Post
Comment
Comment
Comment
Post
Comment
Comment
Comment
```
Structure of the structured to hierarchical pattern

Driver Code

```java
public static void main( String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "PostCommentHierarchy");
    job.setJarByClass(PostCommentBuildingDriver.class);
    MultipleInputs.addInputPath(job, new Path(args[0]),
        TextInputFormat.class, PostMapper.class);
    MultipleInputs.addInputPath(job, new Path(args[1]),
        TextInputFormat.class, CommentMapper.class);
    job.setReducerClass(UserJoinReducer.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    TextOutputFormat.setOutputPath(job, new Path(args[2]));
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    System.exit(job.waitForCompletion(true) ? 0 : 2);
}
```
Mapper Code (for posts)

```java
public static class PostMapper extends Mapper<Object, Text, Text, Text> {
    private Text outkey = new Text();
    private Text outvalue = new Text();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
        // The foreign join key is the post ID
        outkey.set(parsed.get("Id"));
        // Flag this record for the reducer and then output
        outvalue.set("P" + value.toString());
        context.write(outkey, outvalue);
    }
}
```

Mapper Code (for comments)

```java
public static class CommentMapper extends Mapper<Object, Text, Text, Text> {
    private Text outkey = new Text();
    private Text outvalue = new Text();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
        // The foreign join key is the post ID
        outkey.set(parsed.get("PostId"));
        // Flag this record for the reducer and then output
        outvalue.set("C" + value.toString());
        context.write(outkey, outvalue);
    }
}
```
Reducer Code

```java
public static class PostCommentHierarchyReducer extends Reducer<Text, Text, Text, NullWritable> {
    private ArrayList<String> comments = new ArrayList<String>();
    private DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    private String post = null;
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        // Reset variables
        post = null;
        comments.clear();
        // For each input value
        for (Text t : values) {
            // If this is the post record, store it, minus the flag
            if (t.charAt(0) == 'P') {
                post = t.toString().substring(1, t.toString().length()).trim();
            } else {
                // Else, it is a comment record. Add it to the list, minus
                // the flag
                comments.add(t.toString().substring(1, t.toString().length()).trim());
            }
        }
        // If there are no comments, the comments list will simply be empty.
        // If post is not null, combine post with its comments.
        if (post != null) {
            // nest the comments underneath the post element
            String postWithCommentChildren = nestElements(post, comments);
            // write out the XML
            context.write(new Text(postWithCommentChildren), NullWritable.get());
        }
    }
} ...
MapReduce Design Patterns III: Data Organization Patterns

2. Partitioning pattern

Partitioning Pattern

- Moves the records into **categories**
  - But it doesn’t really care about the order of records
  - Shards, partitions, or bins

- e.g. Partitioning **by date**
  - Groups data based on date
  - Given a set of user information, partition the records based on the year of last access date, one partition per year
Structure of the partitioning pattern

Partitioner

- Partitions the key-value pairs of intermediate Map-outputs
- Uses a user-defined condition

- e.g. Process the input dataset to find the highest salaried employee by gender in different age groups

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Age</th>
<th>Salary</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>1201</td>
<td>James</td>
<td>45</td>
<td>50000</td>
<td>Male</td>
</tr>
<tr>
<td>1202</td>
<td>Matthew</td>
<td>40</td>
<td>51000</td>
<td>Male</td>
</tr>
<tr>
<td>1203</td>
<td>Kevin</td>
<td>34</td>
<td>45000</td>
<td>Male</td>
</tr>
<tr>
<td>1204</td>
<td>Maria</td>
<td>30</td>
<td>43000</td>
<td>Female</td>
</tr>
<tr>
<td>1205</td>
<td>Julia</td>
<td>20</td>
<td>23000</td>
<td>Female</td>
</tr>
<tr>
<td>1206</td>
<td>Lavanya</td>
<td>25</td>
<td>42000</td>
<td>Female</td>
</tr>
<tr>
<td>1207</td>
<td>Joseph</td>
<td>19</td>
<td>19000</td>
<td>Male</td>
</tr>
<tr>
<td>1208</td>
<td>Steve</td>
<td>22</td>
<td>32000</td>
<td>Male</td>
</tr>
<tr>
<td>1209</td>
<td>Max</td>
<td>24</td>
<td>30000</td>
<td>Male</td>
</tr>
<tr>
<td>1210</td>
<td>Jenifer</td>
<td>28</td>
<td>37000</td>
<td>Female</td>
</tr>
<tr>
<td>1211</td>
<td>Nick</td>
<td>18</td>
<td>19000</td>
<td>Male</td>
</tr>
<tr>
<td>1212</td>
<td>MaryAnn</td>
<td>33</td>
<td>56000</td>
<td>Female</td>
</tr>
<tr>
<td>1213</td>
<td>Kelvin</td>
<td>39</td>
<td>65000</td>
<td>Male</td>
</tr>
</tbody>
</table>
Partitioner

- **Map Tasks**
  - Input
    - Dummy key, data
    - (Dummy_key, “1201 \t James \t 45 \t male \t 50000”)
  - Method
    - Read the value and extract gender information
      ```java
      String[] str = value.toString().split(“\t”, -3);
      String gender = str[3];
      ```
  - Output
    - Gender data and value
      ```java
      context.write(new Text(gender), value);
      ```

---

Partitioner

- **Partitioner Task**
  - Dividing the data from the map task into segments
  - Input
    - A collection of key-value pairs from the map task
    - Key: gender, value: whole record data
  - Method
    - Read the age field and apply conditions
      ```java
      int age = Integer.parseInt(str[2]);
      if(age<=20) {
        return 0;
      } else if(age>20 && age<=30) {
        return 1 % numReduceTasks;
      } else {
        return 2 % numReduceTasks;
      }
      ```
  - Output
    - The data of key-value pairs are segmented into three collections of key-value pairs
    - The reducer works individually on each collection
Partitioner

• Reduce Task
  • The number of partitions (segments) is equal to the number of reduce tasks
  • The reducer will execute three times (in our example) with different collection of key-value pairs

---

Driver Code (last access date partitioner)

```java
... // Set custom partitioner and min last access date
job.setPartitionerClass(LastAccessDatePartitioner.class);
LastAccessDatePartitioner.setMinLastAccessDate(job, 2008);

// Last access dates span between 2008-2011, or 4 years
job.setNumReduceTasks(4);
...```
### Mapper Code (last access date partitioner)

```java
public static class LastAccessDateMapper extends Mapper<Object, Text, IntWritable, Text> {
    public static class LastAccessDateMapper extends Mapper<Object, Text, IntWritable, Text> {
        // This object will format the creation date string into a Date object
        private final static SimpleDateFormat fmt = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS");
        private IntWritable outkey = new IntWritable();
        protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
            // Grab the last access date
            String strDate = parsed.get("LastAccessDate");
            // Parse the string into a Calendar object
            Calendar cal = Calendar.getInstance();
            cal.setTime(fmt.parse(strDate));
            outkey.set(cal.get(Calendar.YEAR));
            // Write out the year with the input value
            context.write(outkey, value);
        }
    }
}
```

### Partitioner code (last access date partitioner)

```java
public static class LastAccessDatePartitioner extends Partitioner<IntWritable, Text> implements Configurable {
    private static final String MIN_LAST_ACCESS_DATE_YEAR = "min.last.access.date.year";
    private Configuration conf = null;
    private int minLastAccessDateYear = 0;
    public int getPartition(IntWritable key, Text value, int numPartitions) {
        return key.get() - minLastAccessDateYear;
    }
    public Configuration getConf() {
        return conf;
    }
    public void setConf(Configuration conf) {
        this.conf = conf;
        minLastAccessDateYear = conf.getInt(MIN_LAST_ACCESS_DATE_YEAR, 0);
    }
}```
Partitioner code (last access date partitioner)

```java
public static void setMinLastAccessDate(Job job, int minLastAccessDateYear) {
    job.getConfiguration()
        .setInt(MIN_LAST_ACCESS_DATE_YEAR, minLastAccessDateYear);
}
```

Reducer Code (last access date partitioner)

```java
public static class ValueReducer extends Reducer <IntWritable, Text, Text, NullWritable> {
    protected void reduce(IntWritable key, Iterable <Text> values, Context context) throws IOException, InterruptedException {
        for (Text t : values) {
            context.write(t, NullWritable.get());
        }
    }
}
```
Unevenly distributed partitions

- Observation
  - Recent years will have more users
- Provide finer grained segmentations to the recent years
  - e.g. Monthly partitions for recent 3 years

MapReduce Design Patterns II: Filtering Patterns

3. Total Order Sorting Pattern
Total Order Sorting Pattern

- Sorts your data
  - e.g. Sorting 1TB of numeric values
  - e.g. Sorting comments by userID and you have a million users

Structure of Total Order Sorting Pattern

- Two phases
  - **Analysis** phase
    - Determines the ranges
  - **Sorting** phase
    - Actually sorts the data
Structure of the Total Order Sorting Pattern
- Analysis phase

• Performs a **simple random sampling**
• Generates outputs with **the sort key as its output keys**
  • Data will show up as sorted at the reducer
• **Sampling rate**?
  • Assume that the number of records in the entire dataset is known (or can be estimated)
  • If you plan on running the order with a thousand reducers
    • Sampling about a hundred thousand records will be enough
• Only **one reducer** will be used
  • Collects the sort keys together into a sorted list
  • The list of **sorted keys will be sliced into the data range boundaries**

Structure of Total Order Sorting Pattern
- Sorting phase

• Mapper extracts the sort key
  • Stores the sort key to the “value”

• **Custom partitioner**
  • Use **TotalOrderPartitioner** (Hadoop API)
  • Takes the data ranges from the partition file and decides which reducer to send the data
  • Dynamic and load balanced

• **Reducer**
  • The number of reducers needs to be **equal to the number of partitions**
TeraSort Benchmark

- The most well-known Hadoop benchmark

- In 2008, Yahoo! Set a record by sorting 1 TB of data in 209 seconds
  - Hadoop cluster with 910 nodes
  - Owen O’Malley of the Yahoo!

- In 2009, Yahoo! Sorted 1PB of data in 16 hours
  - Hadoop cluster of 3800 nodes
  - For 1TB, it took 62 seconds
  - [http://sortbenchmark.org/YahooHadoop.pdf](http://sortbenchmark.org/YahooHadoop.pdf)

TeraSort Benchmark APIs

- TeraGen
  - MR to generate the data
- TeraSort
  - Samples the input data and uses MR to sort the data into a total order
- TeraValidate
  - MR that validates the output

- TeraSort is a standard MapReduce with a custom partitioner that uses a sorted list of N-1 sorted sampled keys that define the key range for each reduce
  - \( sample[i-j] \leq key < sample[i] \) are sent to reducer \( i \)
  - Total 1,000 lines of java code
Questions?