Topics

- MapReduce Design Pattern II. Filtering Patterns

This material is built based on,

- MapReduce Design Patterns
  - Building Effective Algorithms and Analytics for Hadoop and Other Systems
  - By Donald Miner, Adam Shook
  - November, 2012

FAQs

- PA0 has been posted
  - Feb. 6, 5:00PM via Canvas
  - Individual submission (No team submission)
  - If you are not assigned the "port range", contact GTA immediately

- Team for term project
  - 3-4 members
  - Jan. 30 (TOMORROW), 5:00PM via Canvas
  - Looking for team members? Post your advertisement on Piazza!

Filtering pattern

- Providing an abstract of existing data
- Many data filtering do NOT require the "reduce" part of MapReduce
  - It does not produce an aggregation
- Known uses
  - Tracking a thread of events
  - Distributed grep
  - Data cleaning
  - Closer view of data
  - Simple random sampling
  - Removing low scoring data
Filtering patterns covered in this class

1. Simple Random Sampling
2. Bloom filter
3. Top 10
4. Distinct

Filtering Pattern 1.
Simple Random Sampling

• Each record has an equal probability of being selected
• Useful for sizing down a data set
• For representative analysis

Writing a Simple Random Sampling filter

```java
public static class SRSMapper
extends Mapper<Object, Text, NullWritable, Text> {

private Random rands = new Random();
private Double percentage;

protected void setup(Context context) throws IOException, InterruptedException {
    // Retrieve the percentage that is passed in via the configuration
    // like this:
    // conf.set("filter_percentage", .5);
    // for .5%
    String strPercentage = context.getConfiguration().get("filter_percentage");
    percentage = Double.parseDouble(strPercentage) / 100.0;
}

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    if (rands.nextDouble() < percentage) {
        context.write(NullWritable.get(), value); // otherwise, drop it.
    }
}
```

Stratified Sampling with MapReduce

• Implementing the Stratified Sampling
• Proportion allocation
  • The size of portion in each stratum is taken in proportion to the size of the stratum
  • e.g. Sampling from Olympic athletes
  • With two strata, women and men participants
Stratified Sampling with MapReduce

- Implementing the Stratified Sampling
- Proportion allocation
  - The size of portion in each stratum is taken in proportion to the size of the stratum
- e.g. Sampling from Olympic athletes
  - With two strata, women and men participants
  1. Create the strata
  2. A stratum is made of elements belonging to the same class
  3. Apply Simple Random sampling to each stratum

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

What is a Bloom Filter?

- Burton Howard Bloom in 1970
- Probabilistic data structure used to test whether a member is an element of a set
- Strong space advantage

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

- \( m = 8 \), \( n = 3 \) target set \( T = \{ 5, 10, 15 \} \)
- \( k = 3 \)
  - \( h_1(x) = 3x \mod 8 \)
  - \( h_2(x) = (2x + 3) \mod 8 \)
  - \( h_3(x) = x \mod 8 \)
  - \( h_1(5) = 5, h_2(5) = 5, h_3(5) = 5 \)
  - \( h_1(10) = 6, h_2(10) = 7, h_3(10) = 2 \)

After encoding 5:
- \( h_1(5) = 5 \)
- \( h_2(5) = 5 \)
- \( h_3(5) = 5 \)

After encoding 10:
- \( h_1(10) = 6 \)
- \( h_2(10) = 7 \)
- \( h_3(10) = 2 \)

After encoding 15:
- \( h_1(15) = 7 \)
- \( h_2(15) = 1 \)
- \( h_3(15) = 0 \)

Applying a Bloom filter

- Is 8 part of set \( T \)?
  - \( h_1(8), h_2(8), h_3(8) \)
  - 8 is **NOT** a part of set \( T \)

After encoding 1, 10, and 15:
- \( h_1(1) = 0 \)
- \( h_2(1) = 3 \)
- \( h_3(1) = 0 \)
- \( h_1(10) = 0 \)
- \( h_2(10) = 5 \)
- \( h_3(10) = 0 \)
- \( h_1(15) = 7 \)
- \( h_2(15) = 7 \)
- \( h_3(15) = 7 \)
Applying a Bloom filter

- Is 7 part of set T?
  - \( h(7), h(2)(7), h(3)(7)/k \) bits are 1
  - 7 is probably a part of set T

<table>
<thead>
<tr>
<th>h(x)</th>
<th>y</th>
<th>z</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

After encoding 5, 10 and 15:

- \( h(5) = 111100100010 \)
- \( h(10) = 111100100010 \)
- \( h(15) = 111100100010 \)

Hash functions

- \( k \) Hash functions
  - Uniform random distribution in \([1..m)\)
- Cryptographic hash functions
  - MD5, SHA-1, SHA-256, Tiger, Whirlpool...
- Murmur Hashes (non-cryptographic), Jenkins, Java hashCode, Spooky, CityHash

Avalanche Effect

- Each bit should have 50% chances to change if you change 1 bit of the input
- Avalanche Diagram
  - A grid such that the \((x,y)\) cell’s color represents the probability that flipping \(x\)th bit of input will result of \(y\)th bit being flipped in the output

False positive rate (1/2)

\[
fp = (1 - \left(1 - \frac{1}{m}\right)^n)^k = (1 - e^{-\frac{k}{m}})^k
\]

\( m \) = number of bits in the filter
\( n \) = number of elements
\( k \) = number of hashing functions

- 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 $\left(1-e^{-kn/m}\right)$

- Then, how many hash functions should I use?
  - The more hash functions you have
    - the slower your bloom filter
  - If you have few hash functions
    - Too many false positives
  - Given an $m$ and an $n$, the optimal value of $k$
    - $\left(m/n\right)\ln(2)$

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

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();
    }
```
Filtering Pattern 3. **Top 10**

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

- Known uses
  - Ranking scheme in your application
  - Outlier analysis
  - Selecting interesting data
  - Data summarization
  - Catchy dashboards

The structure of Top 10 pattern

**MapReduce Design Patterns II: Filtering Patterns**

3. **Top 10 (Top K)**

**Mapper**

```java
protected void map(Context context) throws IOException, InterruptedException {
    try {
        while (context.nextKey() != null) {
            final String key = context.getCurrentKey();
            final Text value = context.getCurrentValue();

            // Process the key-value tuple
            // Example code:
            int valueInt = Integer.parseInt(value.toString());
            String keyStr = key.toString();

            // Perform filtering or aggregation
            if (valueInt > 10) {
                context.write(new IntWritable(valueInt), new Text(keyStr));
            }
        }
    } finally {
        cleanup(context);
    }
}
```

**Reducer**

```java
void reduce(Key key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
    String keyStr = key.toString();
    int valueSum = 0;
    for (Text value : values) {
        int valueInt = Integer.parseInt(value.toString());
        valueSum += valueInt;
    }
    context.write(new IntWritable(valueSum), new Text(keyStr));
}
```
Reducer

```java
public static class TopTenReducer extends Reducer <NullWritable, Text, NullWritable, Text> {
    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 (repToRecordMap.size() > 10) {
                repToRecordMap.remove (repToRecordMap.firstKey ());
            }
        }

        for (Text t : repToRecordMap.descendingMap ().values()) {
            context.write (NullWritable.get (), t);
        }
    }
}
```

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 {
        context.write (key, NullWritable.get ());
    }
}
```

Questions?