CS 378 – Big Data Programming

Lecture 15

Filtering Patterns

Review

- Assignment 7 – Filtering, Multiple Outputs

- Questions/issues:
  - Determining the session category
  - Using `AvroMultipleOutputs`
  - Output to submit

Basic Filtering - Review

- Some common basic filtering uses

- grep
- Random sample
- Score records on some criterion, apply a threshold
- Data cleansing

Distributed grep

- grep – Unix filtering utility

- Apply a regular expression to each input record
- Output records that match

Distributed grep

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

    private String mapRegex = null;

    public void setup(Context context) throws IOException,
        InterruptedException {

        mapRegex = context.getConfiguration().get("mapregex");
    }

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {

        if (value.toString().matches(mapRegex)) {
           context.write(NullWritable.get(), value);
        }
    }
}
```

Simple Random Sampling

- Each input record has equal probability of selection
- Does the selection predicate need to examine the record?
  - If we want the equal probability condition, then no.
  - If we want a biased sample, we can consider the record
- Like basic filtering, consider output file size

Simple Random Sampling

```
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);
    }
}
```

Bloom Filter

- Bloom filter like the basic filter
- But selection predicate is:
  - Does record contain a value from a predefined set?
- This set may be too large to fit in memory
- Bloom filter addresses this problem
  - Fixed size data structure

Bloom Filter

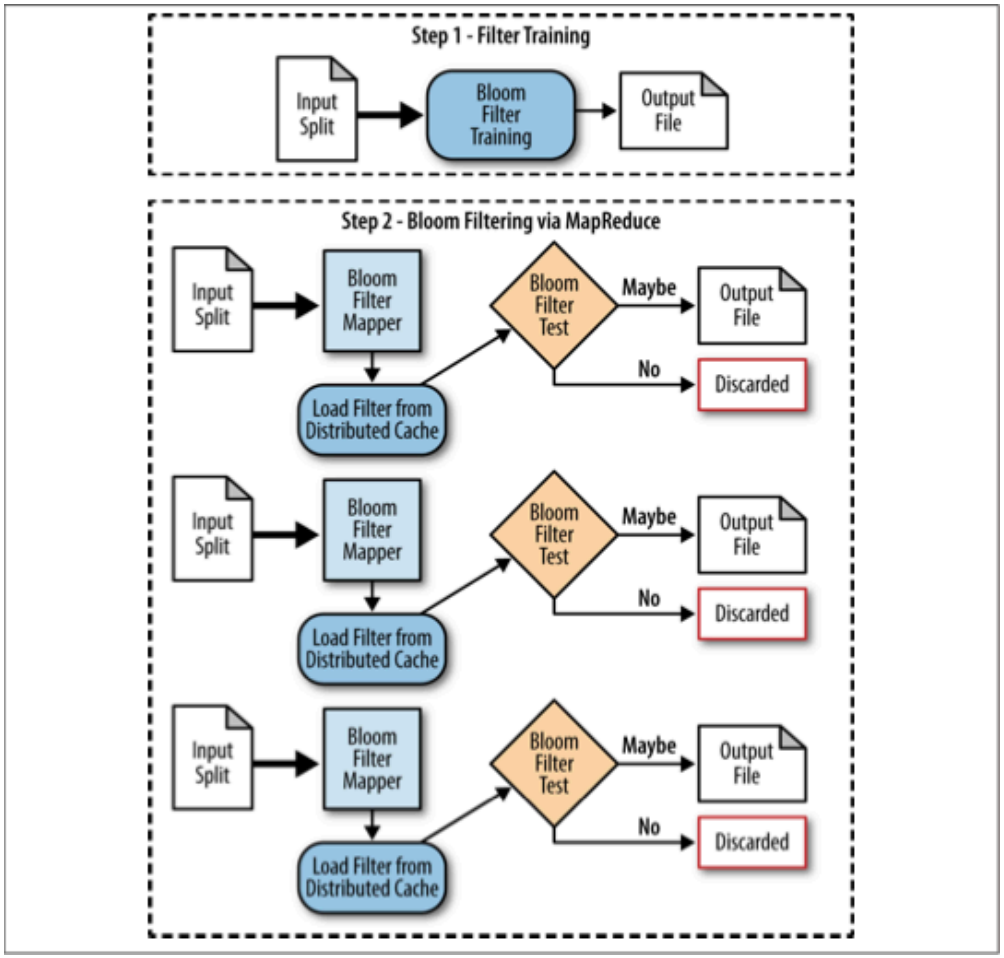
- Probabilistic data structure
  - Used to test whether something is in a predefined set
  - Can create “false positives”
    - Knows for sure that something is not a member of the set
    - Sometimes reports membership as true, when it is false
  - Never creates “false negatives”
    - Never reports “not a member” when it in fact it is a member
- Fixed size in memory
  - Train the filter using members of the set

Bloom Filter Training

- Data structure
  - Bit array
- Multiple hash functions
- Process all "keys' that will be in the set to populate the bloom filter data structure

Bloom Filter – Data Flow

Figure 3-2 from MapReduce Design Patterns



Bloom Filter

- Bloom filter commonly used as map-only
  - Output files will have some false positives
  - Code examples in the book (pp. 53 – 57)

- We can combine Bloom filter with reduce-side join
  - Bloom filter represented user IDs with submit events
  - Applied in the mapper
  - Reduced the data sent to reduce
  - Reduce eliminates false positives (non-submit sessions)

Reduce Side Join with Bloom Filter

- Train the filter
  - Read all log entries, identify userIds with submits
- Specify the trained data file in our driver app (run() method)
- Modify the mapper to load the trained Bloom filter
  - setup() method
- Reducer – what does it need to do?

Bloom Filter

- Can add members to the set (further training)
  - Can't remove members
  - There is a technique that allows removal
- Parameters of the filter
  - Number of bits in a bit array
  - Number of independent hash functions
- These can be tuned to get a certain false positive rate

Top Ten (or Top N)

- We know that we want a specific number of outputs
  - Based on some evaluation/ranking criterion

- An obvious approach is to sort first
- But total sort is expensive for large data
  - In Hadoop, or in a database
- Output should be significantly smaller than the input

- How might we accomplish this without sort?

Top Ten (or Top N)

- Start with a comparison method
  - Given two records, which one is larger

- Each mapper finds the top ten from its data
- Each mapper sends it top ten to reduce
  - When?
- Reduce finds the final top ten
  - How many reducers?
- What key and value are sent to reduce?

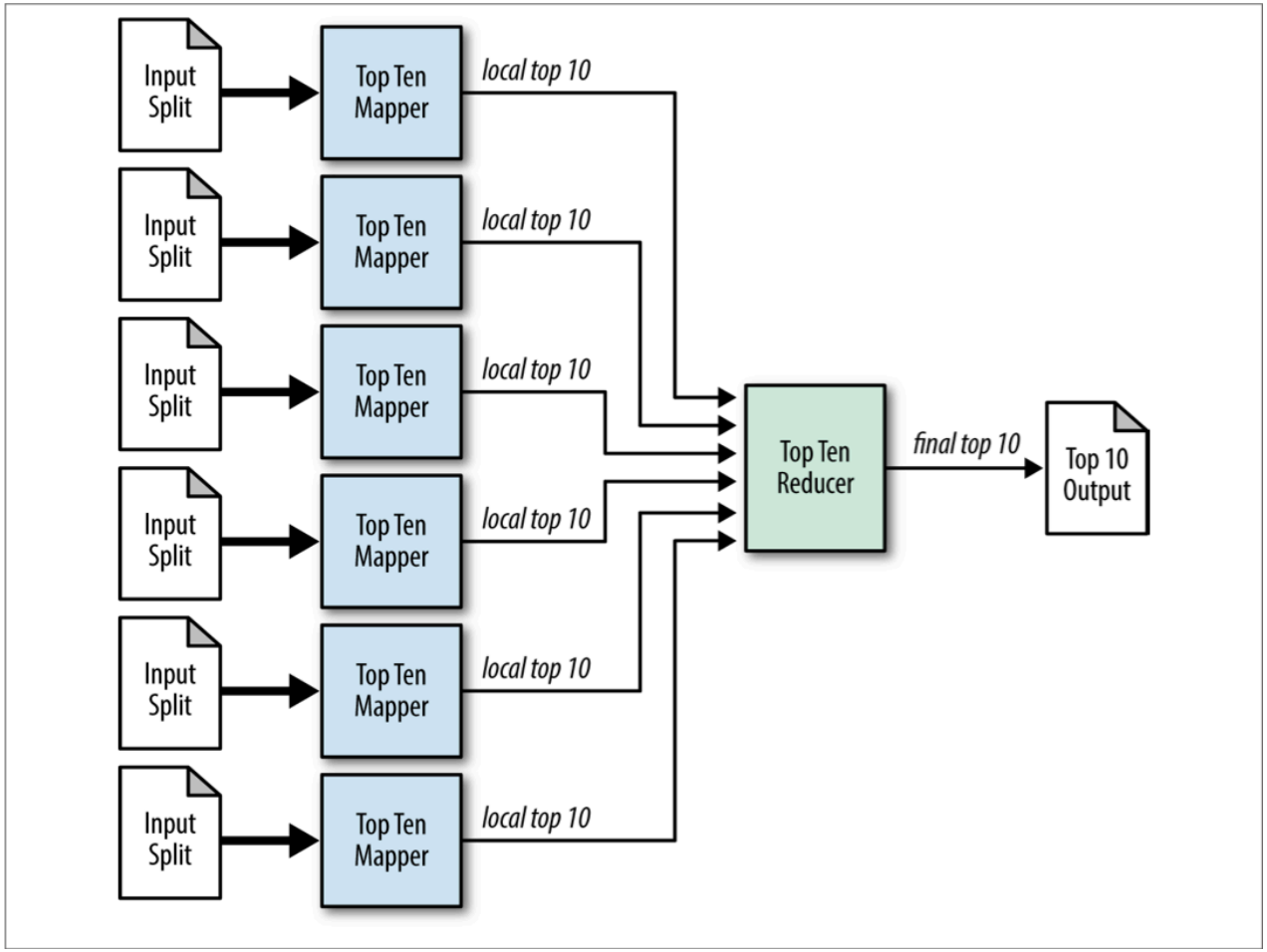
Top Ten (or Top N)

```
class mapper:
    setup():
        initialize top ten sorted list

    map(key, record):
        insert record into top ten sorted list
        if length of array is greater-than 10 then
            truncate list to a length of 10

    cleanup():
        for record in top sorted ten list:
            emit null,record
```

Top Ten (or Top N)

```
class reducer:
    setup():
        initialize top ten sorted list

    reduce(key, records):
        sort records
        truncate records to top 10
        for record in records:
            emit record
```

Top Ten (or Top N)



Top Ten (or Top N)

- Remember to copy records retained in `map()`
  - Why?
- Recall the key/value output by the mappers
  - Suppose we wanted the original key/value pair. How?
- Each mapper's Top N is output in `cleanup()`
  - How could we do Top N without using `cleanup()`?

Top Ten (or Top N)

- For top N, if N large, this pattern becomes inefficient

- Single reducer
- Data transferred to reduce
- Reduce input is sorted (expensive for large data)
- No parallel writes from reduce

Distinct

- Want only one record when duplicate records exist

- Map:
  - Extract the data of interest (if not the entire record)
  - Output this data as the key
  - Make the value output by `map()` NullWritable

- Reduce:
  - Simply write out each unique key (the data of interest)
  - Can use a large number of reducers

Distinct

- Can we use a combiner?

- If duplicates are rare, combiner doesn't help much
- If duplicates are common, or co-located, a combiner can greatly reduce the data transferred

- Suppose we want all the data in the record, and
  - The compare method is complex
  - Can we approach this problem differently?