MapReduce Filtering Patterns

- Filtering Patterns don't change the actual records.
- These patterns all find a subset of data:
  - it could be small
    - like a top-ten listing,
  - Or it could be large
    - like the results of a deduplication
- This differentiates filtering patterns from those in the previous lecture,
  - which was all about summarizing and grouping data by similar fields to get a top-level view of the data.
- Filtering is more about understanding
  - a smaller piece of your data
    - such as all records generated from a particular user
    - top ten most used verbs in a corpus of text.

Filtering can also be Considered a Form of Search

- If you are interested in finding all records that involve a particular piece of distinguishing information, you can filter out records that do not match the search criteria.

Sampling

- Sampling is one common application of filtering.
- It is about pulling out a sample of the data,
  - such as the highest values for a particular field or a few random records.
- Sampling can be used to get a smaller, yet representative, data set in which more analysis can be done without having to deal with the much larger data set.
- Many machine learning algorithms simply do not work efficiently over a large data set, so tools that build models need to be applied to a smaller subset.

Subsampling

- A subsample can also be useful for development purposes.
- Simply grabbing the first thousand records typically is not the best sample since the records are bound to be similar and do not give a good overall picture of the entire data set.
- A well-distributed sample will hopefully provide a better view of the data set and will allow your application and analytic development to be done against more realistic data, even if it is much smaller.

Patterns

- Four patterns are presented in this lecture:
  - Filtering
  - Bloom filtering
  - top ten
  - distinct.
- There are numerous ways to find a slice of your data.
- Each pattern has a slight nuance to distinguish it from the others, even if they all pretty much do the same thing.
- We will see a few clever uses of MapReduce in this lecture.
- Filtering, Bloom filtering, and simple random sampling allow us to use map-only jobs, which means we don't need a reducer.

Filtering - Pattern Description

- As the most basic pattern, filtering serves as an abstract pattern for some of the other patterns.
- Filtering simply evaluates each record separately and decides, based on some condition, whether it should stay or go.

Intent

- Filter out records that are not of interest and keep ones that are.
- Consider an evaluation *function f* that takes a record and returns a Boolean value of true or false.
- If this function returns true, keep the record; otherwise, toss it out.

Motivation

- Your data set is large and you want to take a subset of this data to focus in on it and perhaps do follow-on analysis.
- The subset might be a significant portion of the data set or just a needle in the haystack.
- Either way, you need to use the parallelism of MapReduce to wade through all of your data and find the keepers.
- For example, you might be interested only in records that have something to do with Hadoop:
  - Hadoop is either mentioned in the raw text or the event is tagged by a "Hadoop" tag.
  - Filtering can be used to keep records that meet the "something to do with Hadoop" criteria and keep them, while tossing out the rest of the records.

Motivation Cont'd

- Big data and processing systems like Hadoop, in general, are about bringing all of your organization's data to one location.
- Filtering is the way to pull subsets back out anddeliver them to analysis shops that are interested in just that subset.
- Filtering is also used to zoom in on a particular set of records that match your criteria that you are more curious about.
- The exploration of a subset of data may lead to more valuable and complex analytics that are based on the behavior that was observed in the small subset.

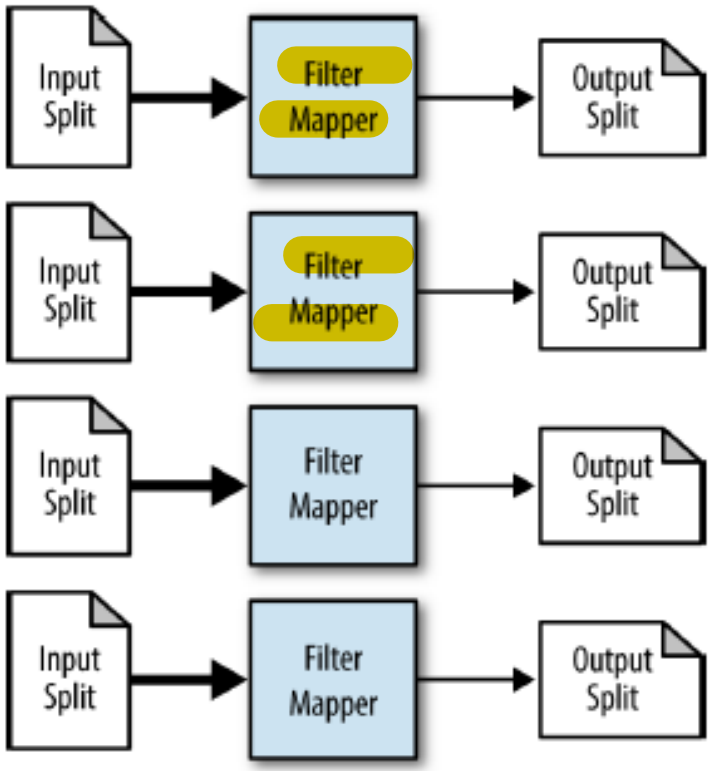
Applicability

- Filtering is very widely applicable.
- The only requirement is that the data can be parsed into “records” that can be categorized through some well-specified criterion determining whether they are to be kept.

Structure of the Filter Pattern

- The structure of the filter pattern is perhaps the simplest of all the patterns.

```
map(key, record):
        if we want to keep record then
            emit key,value
```



Filtering does not Require the "Reduce" Part of MapReduce

- This is because it doesn't produce an aggregation.
- Each record is looked at individually and the evaluation of whether or not to keep that record does not depend on anything else in the data set.
- The mapper applies the evaluation function to each record it receives.
- Typically, the mapper outputs the same key/value type as the types of the input, since the record is left unchanged.
- If the evaluation function returns true, the mapper simply output the key and value verbatim.

Outcomes

- The output of the job will be a subset of the records that pass the selection criteria.
- If the format was kept the same, any job that ran over the larger data set should be able to run over this filtered data set, as well.

Known Uses

1. Closer view of data
2. Tracking a thread of events
3. Distributed grep
4. Data cleansing
5. Simple random sampling
6. Removing low scoring data

1. Closer view of data

- Prepare a particular subset of data, where the records have something in common or something of interest, for more examination.

- For example, a local office in Maryland may only care about records originating in Maryland from your international dataset.

2. Tracking a thread of events

- Extract a thread of consecutive events as a case study from a larger data set.
- For example, you may be interested in how a particular user interacts with your website by analyzing Apache web server logs.
- The events for a particular user are interspersed with all the other events, so it's hard to figure out what happened.
- By filtering for that user's IP address, you are able to get a good view of that particular user's activities.

3. Distributed grep

- Grep, a very powerful tool that uses regular expressions for finding lines of text of interest, is easily parallelized by applying a regular expression match against each line of input and only outputting lines that match.

4. Data cleansing

- Data sometimes is dirty, whether it be malformed, incomplete, or in the wrong format.
- The data could have missing fields, a date could be not formatted as a date, or random bytes of binary data could be present.
- Filtering can be used to validate that each record is well-formed and remove any junk that does occur.

5. Simple random sampling

- If you want a simple random sampling of your data set, you can use filtering where the evaluation function randomly returns true or false.
- A simple random sample is a sample of the larger data set in which each item has the same probability of being selected.
- You can tweak the number of records that make it through by having the evaluation function return true a smaller percentage of the time.
- For example, if your data set contains one trillion records and you want a sample size of about one million, have the evaluation function return true once in a million (because there are a million millions in a trillion).

6. Removing low scoring data

- If you can score your data with some sort of scalar value, you can filter out records that don't meet a certain threshold.
- If you know ahead of time that certain types of records are not useful for analysis, you can assign those records a small score and they will get filtered out.
- This effectively has the same purpose as the top ten pattern discussed later, except that you do not know how many records you will get.

Resemblances

SQL

The filter pattern is synonymous to using the `WHERE` clause in a `SELECT *` statement. The records stay the same, but some are simply filtered out. For example:

```
SELECT * FROM table WHERE value < 3;
```

Pig

The `FILTER` keyword.

```
b = FILTER a BY value < 3;
```

Performance Analysis

- This pattern is basically as efficient as MapReduce can get because the job is map-only.

- There are a couple of reasons why map-only jobs are efficient.
  - Since no reducers are needed, data never has to be transmitted between the map and reduce phase.
  - Most of the map tasks pull data off of their locally attached disks and then write back out to that node.
  - Since there are no reducers, both the sort phase and the reduce phase are cut out. This usually doesn't take very long, but every little bit helps.

Performance Analysis Cont'd

- One thing to be aware of is the size and number of the output files.
- Since this job is running with mappers only, you will get one output file per mapper with the prefix part-m- (note the m instead of the r).
- You may find that these files will be tiny if you filter out a lot of data, which can cause problems with scalability limitations of the NameNode further down the road.
- If you are worried about the number of small files and do not mind if your job runs just a little bit longer, you can use an identity reducer to collect the results without doing anything with them.
- This will have the mapper send the reducer all of the data, but the reducer does nothing other than just output them to one file per reducer.
- The appropriate number of reducers depends on the amount of data that will be written to the file system and just how many small files you want to deal with.

Filtering Examples

- **Distributed grep**

- Grep is a popular text filtering utility that dates back to Unix and is available on most Unix-like systems.

- It scans through a file line-by-line and only outputs lines that match a specific pattern.

- We'd like to parallelize the regular expression search across a larger body of text.

- In this example, we'll show how to apply a regular expression to every line in MapReduce.

Mapper code. The mapper is pretty straightforward since we use the Java built-in libraries for regular expressions. If the text line matches the pattern, we'll output the line. Otherwise we do nothing and the line is effectively ignored. We use the setup function to retrieve the map regex from the job configuration.

```
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);
        }
    }
}
```

As this is a map-only job, there is no combiner or reducer. All output records will be written directly to the file system.

Example

- **Simple Random Sampling**

- In simple random sampling (SRS), we want to grab a subset of our larger data set in which each record has an equal probability of being selected.

- Typically this is useful for sizing down a data set to be able to do representative analysis on a more manageable set of data.

- Implementing SRS as a filter operation is not a direct application of the filtering pattern, but the structure is the same.

- Instead of some filter criteria function that bears some relationship to the content of the record, a random number generator will produce a value, and if the value is below a threshold, keep the record. Otherwise, toss it out.

Mapper Code. In the mapper code, the `setup` function is used to pull the `filter_percentage` configuration value so we can use it in the `map` function.

In the `map` function, a simple check against the next random number is done. The random number will be anywhere between 0 and 1, so by comparing against the specified threshold, we can keep or throw out the record.

```
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);
        }
    }
}
```

As this is a map-only job, there is no combiner or reducer. All output records will be written directly to the file system. When using a small percentage, you will find that the files will be tiny and plentiful. If this is the case, set the number of reducers to 1 without specifying a reducer class, which will tell the MapReduce framework to use a single identity reducer that simply collects the output into a single file. The other option would be to collect the files as a post-processing step using `hadoop fs -cat`.

Example

- **Bloom Filtering**
  - Pattern Description
    - Bloom filtering does the same thing as the previous pattern, but it has a unique evaluation function applied to each record.

- Intent
  - Filter such that we keep records that are member of some predefined set of values.
  - It is not a problem if the output is a bit inaccurate, because we plan to do further checking.
  - The predetermined list of values will be called the set of hot values.
  - For each record, extract a feature of that record.
    - If that feature is a member of a set of values represented by a Bloom filter, keep it; otherwise toss it out (or the reverse).

Motivation

- Bloom filtering is similar to generic filtering in that it is looking at each record and deciding whether to keep or remove it.

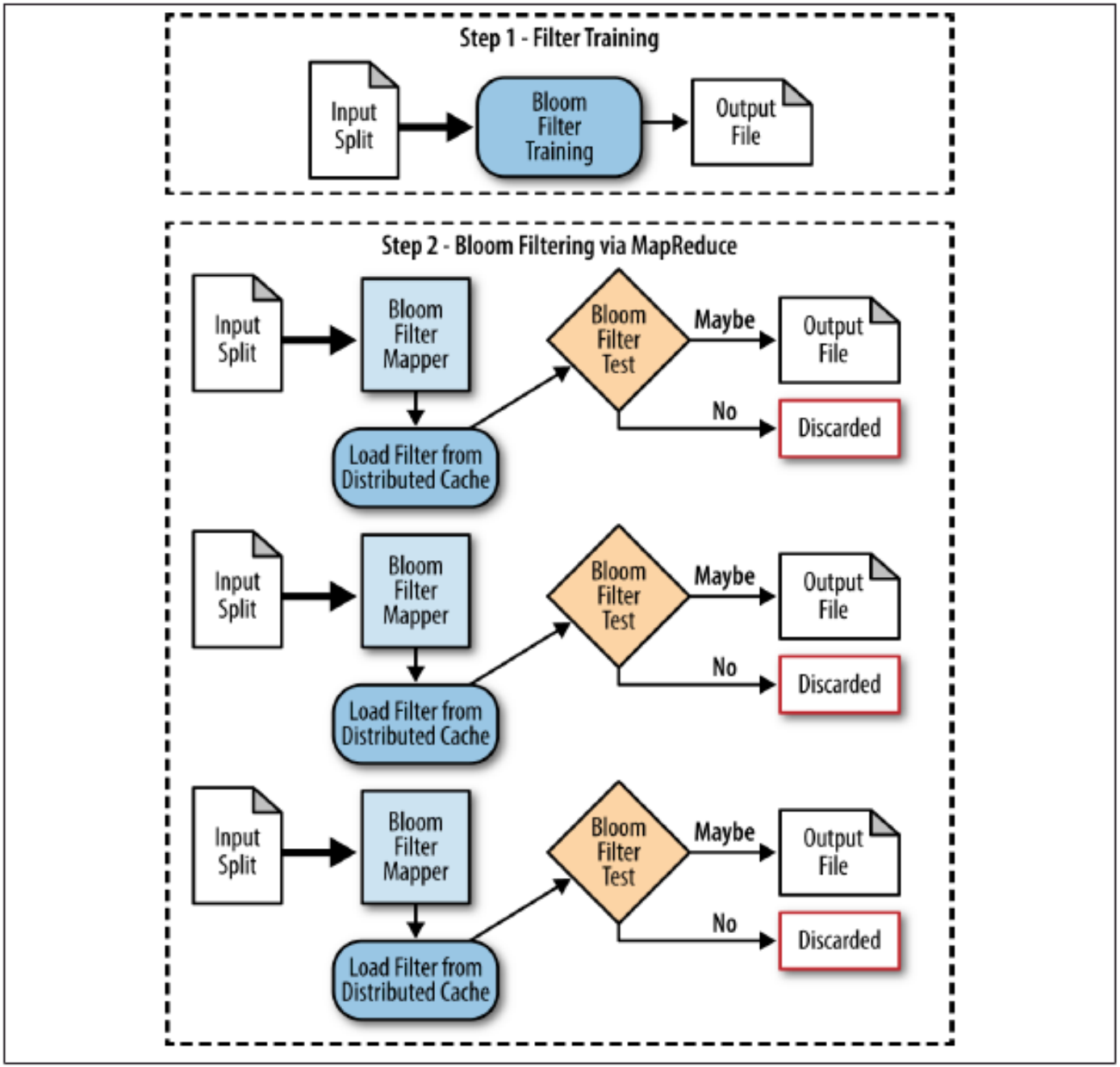
- However, there are two major differences that set it apart from generic filtering.
  - First, we want to filter the record based on some sort of set membership operation against the hot values.
    - For example: keep or throw away this record if the value in the user field is a member of a predetermined list of users.
  - Second, the set membership is going to be evaluated with a Bloom filter.
  - In one sense, Bloom filtering is a join operation in which we don't care about the data values of the right side of the join.

Applicability

- The following criteria are necessary for Bloom filtering to be relevant:
  - Data can be separated into records, as in filtering.
  - A feature can be extracted from each record that could be in a set of hot values.
  - There is a predetermined set of items for the hot values.
  - Some false positives are acceptable (i.e., some records will get through when they should not have).

Structure

- Figure in the next slide shows the structure of Bloom filtering and how it is split into two major components.
- First, the Bloom filter needs to be trained over the list of values.
- The resulting data object is stored in HDFS.
- Next is the filtering MapReduce job, which has the same structure as the previous filtering pattern, except it will make use of the distributed cache as well
- There are no reducers since the records are analyzed one-by-one and there is no aggregation done.



The structure of the Bloom filtering pattern

Steps in Bloom Filter

- The first step of this job is to train the Bloom filter from the list of values.
  - This is done by loading the data from where it is stored and adding each item to the Bloom filter.
  - The trained Bloom filter is stored in HDFS at a known location.

- The second step of this pattern is to do the actual filtering.
  - When the map task starts, it loads the Bloom filter from the distributed cache.
  - Then, in the map function, it iterates through the records and checks the Bloom filter for set membership in the hot values list.
  - Each record is either forwarded or not based on Bloom filter membership test.

- The Bloom filter needs to be re-trained only when the data changes.
- Therefore, updating the Bloom filter in a lazy fashion (i.e., only updating it when it needs to be updated) is typically appropriate.

Outcomes

- The output of the job will be a subset of the records in that passed the Bloom filter membership test.
- You should expect that some records in this set may not actually be in the set of hot values, because Bloom filters have a chance of false positives.

Known uses

Removing most of the nonwatched values

The most straightforward use case is cleaning out values that aren't hot. For example, you may be interested only in data that contains a word in a list of 10,000 words that deal with Hadoop, such as "map," "partitioning," etc. You take this list, train a Bloom filter on it, then check text as it is coming in to see whether you get a Bloom filter hit on any of the words. If you do, forward the record, and if not don't do anything. The fact that you'll get some false positives isn't that big of a deal, since you still got rid of most of the data.

Prefiltering a data set for an expensive set membership check

Sometimes, checking whether some value is a member of a set is going to be expensive. For example, you might have to hit a webservice or an external database to check whether that value is in the set. The situations in which this may be the case are far and few between, but they do crop up in larger organizations. Instead of dumping this list periodically to your cluster, you can instead have the originating system produce a Bloom filter and ship that instead. Once you have the Bloom filter in place and filter out most of the data, you can do a second pass on the records that make it through to double check against the authoritative source. If the Bloom filter is able to remove over 95% of the data, you'll see the external resource hit only 5% as much as before! With this approach, you'll eventually have 100% accuracy but didn't have to hammer the external resource with tons of queries.

Resemblances

- Bloom filters are relatively new in the field of data analysis, likely because the properties of big data particularly benefit from such a thing in a way previous methodologies have not.

- In both SQL and Pig, Bloom filters can be implemented as user-defined functions, but as of now, there is no native functionality out of the box.

Performance Analysis

- The performance for this pattern is going to be very similar to simple filtering from a performance perspective.
- Loading up the Bloom filter from the distributed cache is not that expensive since the file is relatively small.
- Checking a value against the Bloom filter is also a relatively cheap operation, as each test is executed in constant time.

Bloom Filtering Examples - Hot list

- One of the most basic applications of a Bloom filter is what it was designed for: representing a data set.
- For this example, a Bloom filter is trained with a hot list of keywords.
- We use this Bloom filter to test whether each word in a comment is in the hot list.
- If the test returns true, the entire record is output.
- Otherwise, it is ignored.
- Here, we are not concerned with the inevitable false positives that are output due to the Bloom filter.

Problem:

- Given a list of user's comments, filter out a majority of the comments that do not contain a particular keyword.

Bloom filter training

- To demonstrate how to use Hadoop Bloom filters, the following code segment generates a Bloom filter off a predetermined set of words.

- This is a generic application that takes in an input gzip file or directory of gzip files, the number of elements in the file, a desired false positive rate, and finally the output file name.

```java
public class BloomFilterDriver {
    public static void main(String[] args) throws Exception {
        // Parse command line arguments
        Path inputFile = new Path(args[0]);
        int numMembers = Integer.parseInt(args[1]);
        float falsePosRate = Float.parseFloat(args[2]);
        Path bfFile = new Path(args[3]);

        // Calculate our vector size and optimal K value based on approximations
        int vectorSize = getOptimalBloomFilterSize(numMembers, falsePosRate);
        int nbHash = getOptimalK(numMembers, vectorSize);

        // Create new Bloom filter
        BloomFilter filter = new BloomFilter(vectorSize, nbHash,
                Hash.MURMUR_HASH);

        System.out.println("Training Bloom filter of size " + vectorSize
                + " with " + nbHash + " hash functions, " + numMembers
                + " approximate number of records, and " + falsePosRate
                + " false positive rate");

        // Open file for read
        String line = null;
        int numElements = 0;
        FileSystem fs = FileSystem.get(new Configuration());

        for (FileStatus status : fs.listStatus(inputFile)) {
            BufferedReader rdr = new BufferedReader(new InputStreamReader(
                    new GZIPInputStream(fs.open(status.getPath()))));

            System.out.println("Reading " + status.getPath());
            while ((line = rdr.readLine()) != null) {
                filter.add(new Key(line.getBytes()));
                ++numElements;
            }

            rdr.close();
        }

        System.out.println("Trained Bloom filter with " + numElements
            + " entries.");

        System.out.println("Serializing Bloom filter to HDFS at " + bfFile);

        FSDataOutputStream strm = fs.create(bfFile);
        filter.write(strm);
        strm.flush();
        strm.close();

        System.exit(0);
    }
}
```

Mapper Code

- The setup method is called once for each mapper by the Hadoop framework prior to the many calls to map.
- Here, the Bloom filter is deserialized from the DistributedCache before being used in the map method.
- The DistributedCache is a Hadoop utility that ensures that a file in HDFS is present on the local file system of each task that requires that file.
- The Bloom filter was previously trained with a hot list of words.
- In the map method, the comment is extracted from each input record.
- The comment is tokenized into words, and each word is cleaned of any extraneous characters.
- The clean words are testing against the Bloom filter.
  - If the word is a member, the entire record is output to the file system.

```
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();
  }

  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;
      }
    }
  }
}
```

Example

- Bloom filters can assist expensive operations by eliminating unnecessary ones.
- For the following example, a Bloom filter was previously trained with IDs of all users that have a reputation of at least 1,500.
- By eliminating unnecessary queries, we can speed up processing time.
- Problem: Given a list of users' comments, filter out comments from users with a reputation of less than 1,500.

```
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 userid = parsed.get("UserId");

  // If this user ID is in the set
    if (filter.membershipTest(new Key(userid.getBytes()))) {
      // Get the reputation from the HBase table
      Result r = table.get(new Get(userid.getBytes()));
      int reputation = Integer.parseInt(new String(r.getValue(
          "attr".getBytes(), "Reputation".getBytes())));

      // If the reputation is at least 1500,
      // write the record to the file system
      if (reputation >= 1500) {
        context.write(value, NullWritable.get());
    }
  }
}
```

Top Ten

Pattern Description

The *top ten* pattern is a bit different than previous ones in that you know how many records you want to get in the end, no matter what the input size. In generic filtering, however, the amount of output depends on the data.

Intent

Retrieve a relatively small number of top *K* records, according to a ranking scheme in your data set, no matter how large the data.

Motivation

Finding outliers is an important part of data analysis because these records are typically the most interesting and unique pieces of data in the set. The point of this pattern is to find the best records for a specific criterion so that you can take a look at them and perhaps figure out what caused them to be so special. If you can define a ranking function or comparison function between two records that determines whether one is higher than the other, you can apply this pattern to use MapReduce to find the records with the highest value across your entire data set.

The reason why this pattern is particularly interesting springs from a comparison with how you might implement the top ten pattern outside of a MapReduce context. In SQL, you might be inclined to sort your data set by the ranking value, then take the top *K* records from that. In MapReduce, as we'll find out in the next chapter, total ordering is extremely involved and uses significant resources on your cluster. This pattern will instead go about finding the limited number of high-values records without having to sort the data.

Top 10 Examples

- What are the highest scoring posts on Stack Overflow?
- Who is the oldest member of your service?
- What is the largest single order made on your website?
- Which post has the word “mapreduce” the most number of times?

Applicability

- This pattern requires a comparator function ability between two records.
  - That is, we must be able to compare one record to another to determine which is “larger.”

- The number of output records should be significantly fewer than the number of input records because at a certain point it just makes more sense to do a total ordering of the data set.

Structure

This pattern utilizes both the mapper and the reducer. The mappers will find their local top *K*, then all of the individual top *K* sets will compete for the final top *K* in the reducer. Since the number of records coming out of the mappers is at most *K* and *K* is relatively small, we'll only need one reducer.

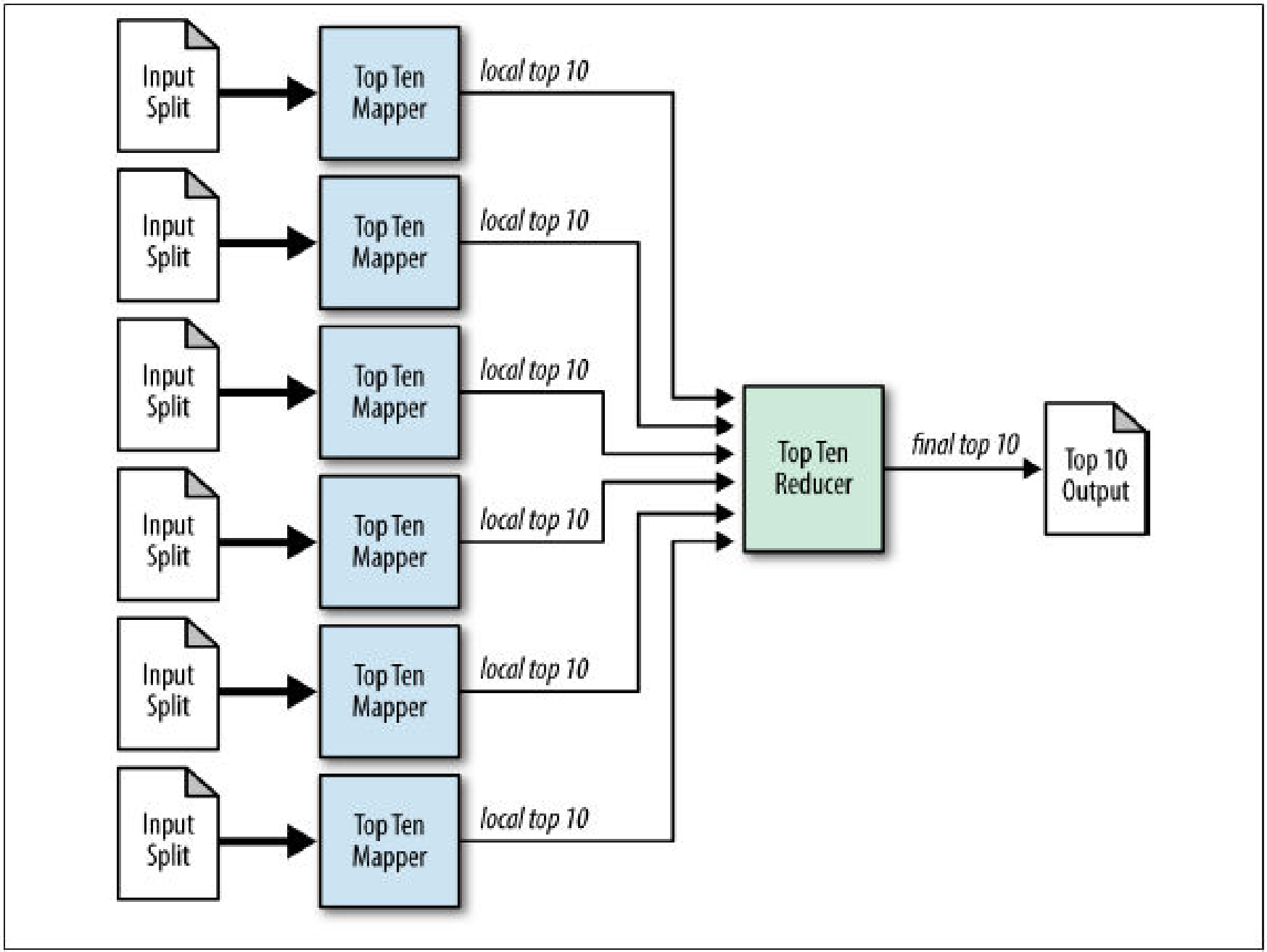
```
class mapper:
   setup():
      initialize top ten sorted list

   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

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

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



The structure of the top ten pattern

How it works

- The mapper reads each record and keeps an array object of size K that collects the largest K values.
- In the cleanup phase of the mapper (i.e., right before it exits), we'll finally emit the K records stored in the array as the value, with a null key.
- These are the lowest K for this particular map task.
- We should expect K * M records coming into the reducer under one key, null, where M is the number of map tasks.
- In the reduce function, we'll do what we did in the mapper: keep an array of K values and find the top K out of the values collected under the null key.
- The reason we had to select the top K from every mapper is because it is conceivable that all of the top records came from one file split and that corner case needs to be accounted for.

Known uses

Outlier analysis

Outliers are usually interesting. They may be the users that are having difficulty using your system, or power users of your website. Outliers, like filtering and grouping, may give you another perspective from your data set.

Select interesting data

If you are able to score your records by some sort of value score, you can pull the "most valuable" data. This is particularly useful if you plan to submit data to follow-on processing, such as in a business intelligence tool or a SQL database, that cannot handle the scale of your original data set. Value scoring can be as complex as you make it by applying advanced algorithms, such as scoring text based on how grammatical it is and how accurate the spelling is so that you remove most of the junk.

Catchy dashboards

This isn't a psychology book, so who knows why top ten lists are interesting to consumers, but they are. This pattern could be used to publish some interesting top ten stats about your website and your data that will encourage users to think more about your data or even to instill some competition.

Resemblances

SQL

In a traditional and small SQL database, ordering may not be a big deal. In this case, you would retrieve data ordered by the criterion for which you want the top ten, then take a limit. You could follow this same approach in MapReduce, but as you will find out in later patterns, sorting is an expensive operation.

```
SELECT * FROM table ORDER BY col4 DESC LIMIT 10;
```

Pig

Pig will have issues performing this query in any sort of optimal way. The most straightforward pattern is to mirror the SQL query, but the ordering is expensive just to find a few records. This is a situation in which you'll find major gains in using Java MapReduce instead of Pig.

```
B = ORDER A BY col4 DESC;
C = LIMIT B 10;
```

Performance Analysis

- The performance of the top ten pattern is typically very good, but there are a number of important limitations and concerns to consider.
- Most of these limitations spring from the use of a single reducer, regardless of the number of records it is handling.
- The number we need to pay attention to when using this pattern is how many records the reducer is getting.
- Each map task is going to output K records, and the job will consist of M map tasks, so the reducer is going to have to work through K * M records.
- This can be a lot.

A single reducer getting a lot of data is bad for a few reasons:

- The sort can become an expensive operation when it has too many records and has to do most of the sorting on local disk, instead of in memory.
- The host where the reducer is running will receive a lot of data over the network, which may create a network resource hot spot for that single host.
- Naturally, scanning through all the data in the reduce will take a long time if there are many records to look through.
- Any sort of memory growth in the reducer has the possibility of blowing through the Java virtual machine's memory. For example, if you are collecting all of the values into an ArrayList to perform a median, that ArrayList can get very big. This will not be a particular problem if you're really looking for the top ten items, but if you want to extract a very large number you may run into memory limits.
- Writes to the output file are not parallelized. Writing to the locally attached disk can be one of the more expensive operations in the reduce phase when we are dealing with a lot of data. Since there is only one reducer, we are not taking advantage of the parallelism involved in writing data to several hosts, or even several disks on the same host. Again, this is not an issue for the top ten, but becomes a factor when the data extracted is very large.

Performance Analysis Cont'd

- As K gets large, this pattern becomes less efficient.
- Consider the extreme case in which K is set at five million, when there are ten million records in the entire data set.
- Five million exceeds the number of records in any individual input split, so every mapper will send all of its records to the reducer.
- The single reducer will effectively have to handle all of the records in the entire dataset and the only thing that was parallelized was the data loading.
- An optimization you could take if you have a large K and a large number of input splits is to prefilter some of the data, because you know what the top ten was last time and it hasn't changed much.
- Imagine your data has a value that can only increase with time (e.g., hits on web pages) and you want to find the top hundred records.
- If, in your previous MapReduce job, the hundredth record had a value of 52,485, then you know you can filter out all records that have a value of less than 52,485. There is no way that a record with a value with less than 52,845 can compete with the previous top hundred that are still in the data set.

Should be used for small values for K

- For all these reasons, this pattern is intended only for pretty small values for K, in the tens or hundreds at most, though you can likely push it a bit further.
- There is a fuzzy line in which just doing a total ordering of the data set is likely more effective.

Top Ten Examples

- **Top ten users by reputation**
- Determining the top ten records of a data set is an interesting use of MapReduce.
- Each mapper determines the top ten records of its input split and outputs them to the reduce phase.
- The mappers are essentially filtering their input split to the top ten records, and the reducer is responsible for the final ten.
- Just remember to configure your job to only use one reducer!
- Multiple reducers would shard the data and would result in multiple “top ten” lists.
- Problem:
  - Given a list of user information, output the information of the top ten users based on reputation.

Mapper code

- The mapper processes all input records and stores them in a TreeMap.
- A TreeMap is a subclass of Map that sorts on key.
- The default ordering of Integers is ascending.
- Then, if there are more than ten records in our TreeMap, the first element (lowest value) can be removed.
- After all the records have been processed, the top ten records in the TreeMap are output to the reducers in the cleanup method.
- This method gets called once after all key/value pairs have been through map, just like how setup is called once before any calls to map.

```
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());
    }
  }

  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);
    }
  }
}
```

Reducer Code

- Overall, the reducer determines its top ten records in a way that's very similar to the mapper.
- Because we configured our job to have one reducer using
  job.setNumReduceTasks(1) and we used NullWritable as our key
  there will be one input group for this reducer that contains all the potential top ten records.
- The reducer iterates through all these records and stores them in a TreeMap.
- If the TreeMap's size is above ten, the first element (lowest value) is remove from the map.
- After all the values have been iterated over, the values contained in the TreeMap are flushed to the file system in descending order.
- This ordering is achieved by getting the descending map from the TreeMap prior to outputting the values.
- This can be done directly in the reduce method, because there will be only one input group, but doing it in the cleanup method would also work.

```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);
    }
  }
}
```

Combiner

- There is no need for a combiner in this job, although the reducer code could technically be used.
- The combiner would simply output the same ten records and thus cause unnecessary processing.
- Also, this job is hardcoded to find the top ten records, but could easily be configured to find the top K records using a variable captured in the setup method.
- Just be sure to keep in mind the limitations discussed in the Performance Analysis section as K increases.

Distinct - Pattern Description

- This pattern filters the whole set, but it's more challenging because you want to filter out records that look like another record in the data set.
- The final output of this filter application is a set of unique records.
- Intent:
  - You have data that contains similar records and you want to find a unique set of values.

Motivation

- Reducing a data set to a unique set of values has several uses.
  - One particular use case that can use this pattern is deduplication.
    - In some large data sets, duplicate or extremely similar records can become a nagging problem. The duplicate records can take up a significant amount of space or skew top-level analysis results.
    - For example, every time someone visits your website, you collect what web browser and device they are using for marketing analysis.
    - If that user visits your website more than once, you'll log that information more than once. If you do some analysis to calculate the percentage of your users that are using a specific web browser, the number of times users have used your website will skew the results. Therefore, you should first deduplicate the data so that you have only one instance of each logged event with that device.
- Records don't necessarily need to be exactly the same in the raw form.
- They just need to be able to be translated into a form in which they will be exactly the same.
- For example, if our web browser analysis done on HTTP server logs, extract only the user name, the device, and the browser that user is using. We don't care about the time stamp, the resource they were accessing, or what HTTP server it came from.

Applicability

- The only major requirement is that you have duplicates values in your data set.
- This is not a requirement, but it would be silly to use this pattern otherwise!

Structure

This pattern is pretty slick in how it uses MapReduce. It exploits MapReduce's ability to group keys together to remove duplicates. This pattern uses a mapper to transform the data and doesn't do much in the reducer. The combiner can always be utilized in this pattern and can help considerably if there are a large number of duplicates. Duplicate records are often located close to another in a data set, so a combiner will deduplicate them in the map phase.

```
map(key, record):
   emit record,null

reduce(key, records):
   emit key
```

The mapper takes each record and extracts the data fields for which we want unique values. In our HTTP logs example, this means extracting the user, the web browser, and the device values. The mapper outputs the record as the key, and null as the value.

The reducer groups the nulls together by key, so we'll have one null per key. We then simply output the key, since we don't care how many nulls we have. Because each key is grouped together, the output data set is guaranteed to be unique.

One nice feature of this pattern is that the number of reducers doesn't matter in terms of the calculation itself. Set the number of reducers relatively high, since the mappers will forward almost all their data to the reducers.

Outcomes

- The output data records are guaranteed to be unique, but any order has not been preserved due to the random partitioning of the records.

Known Uses

- **Deduplicate data**
  - If you have a system with a number of collection sources that could see the same event twice, you can remove duplicates with this pattern.
- **Getting distinct values**
  - This is useful when your raw records may not be duplicates, but the extracted information is duplicated across records.
- **Protecting from an inner join explosion**
  - If you are about to do an inner join between two data sets and your foreign keys are not unique, you risk retrieving a huge number of records.
    - For example, if you have 3,000 of the same key in one data set, and 2,000 of the same key in the other data set, you'll end up with 6,000,000 records, all sent to one reducer!
    - By running the distinct pattern, you can pair down your values to make sure they are unique and mitigate against this problem.

Resemblances

SQL

`SELECT DISTINCT` performs this operation for us in SQL.

```
SELECT DISTINCT * FROM table;
```

Pig

The `DISTINCT` operation.

```
b = DISTINCT a;
```

Performance Analysis

- The main consideration in determining how to set up the MapReduce job is the number of reducers you think you will need.
- The number of reducers is highly dependent on the total number of records and bytes coming out of the mappers, which is dependent on how much data the combiner is able to eliminate.
- Basically, if duplicates are very rare within an input split (and thus the combiner did almost nothing), pretty much all of the data is going to be sent to the reduce phase.
- You can find the number of output bytes and records by looking at the JobTracker status of the job on a sample run. Take the number of output bytes and divide by the number of reducers you are thinking about using.
- That is about how many bytes each reducer will get, not accounting for skew.

Performance Analysis Cont'd

- The number that a reducer can handle varies from deployment to deployment, but usually you shouldn't pass it more than a few hundred megabytes.
- You also don't want to pass too few records, because then your output files will be tiny and there will be unnecessary overhead in spinning up the reducers.
- Aim for each reducer to receive more than the block size of records (e.g., if your block size is 64MB, have at least 64MB sent to the reducer).
- Since most of the data in the data set is going to be sent to the reducers, you will use a relatively large number of reducers to run this job.
- Anywhere from one reducer per hundred mappers, to one reducer per two mappers, will get the job done here.
- Start with the theoretical estimate based on the output records, but do additional testing to find the sweet spot.
- In general, with this pattern, if you want your reducers to run in half the time, double the number of reducers... Just be careful of the files getting too small.

Distinct Examples - Distinct user IDs

- Finding a distinct set of values is a great example of MapReduce's power.
- Because each reducer is presented with a unique key and a set of values associated with that key, in order to produce a distinct value, we simply need to set our key to whatever we are trying to gather a distinct set of.
- **Problem:**
  - Given a list of user's comments, determine the distinct set of user IDs.

Mapper Code

- The Mapper will get the user ID from each input record.
- This user ID will be output as the key with a null value.

```
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

- The work of building a distinct set of user IDs is handled by the MapReduce framework
- Each reducer is given a unique key and a set of null values.
- These values are ignored and the input key is written to the file system with a null value

```
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());
  }
}
```

Combiner Optimization

- A combiner can and should be used in the distinct pattern.
- Duplicate keys will be removed from each local map's output, thus reducing the amount of network I/O required.
- The same code for the reducer can be used in the combiner.