CMPE 282 Cloud Services
MapReduce Design Patterns - Filtering

Instructor: Kong Li

Content

- What and Why
- MapReduce refresher
- Summarization Patterns
- Filtering Patterns
- Data Organization Patterns
- Join Patterns

Filtering Patterns

What: Extract interesting subsets

Why: I only want some of my data

- **Filtering**
- Bloom filtering
- **Top ten**
- **Distinct**

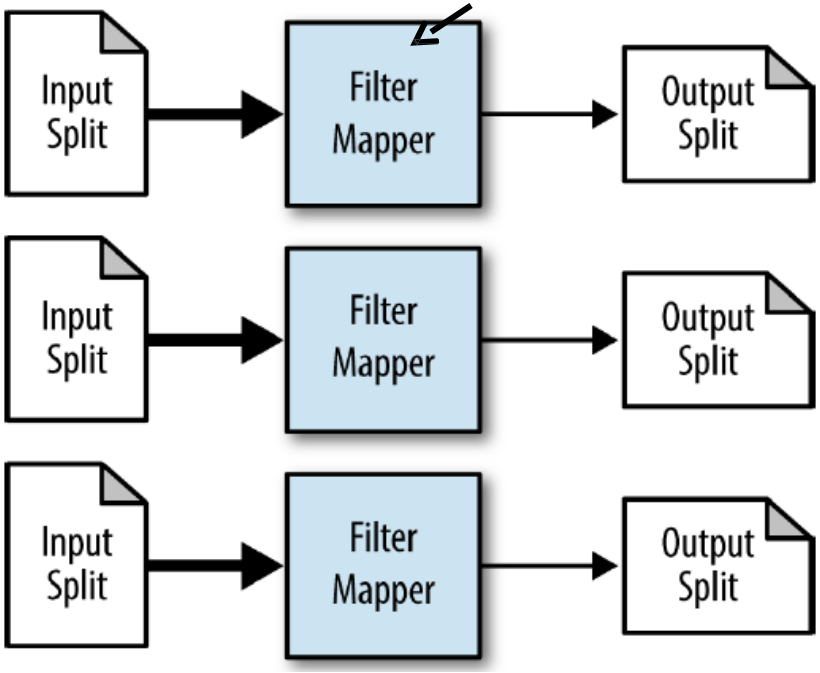
Filtering 1/4

- Intent
  - Filter out records that are not of interest
- 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
- Applicability
  - The data can be parsed into “records” that can be categorized through some well-specified criterion determining whether they are to be kept

Filtering 2/4

- Structure
  - No reducer

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



Filtering 3/4

- Known uses
  - Closer view of data
  - Tracking a thread of events
  - Distributed grep
  - Data cleansing
  - Simple random sampling
  - Removing low scoring data (if you can score your data)
- SQL

  SELECT * FROM table WHERE VALUE < 3
- Performance analysis
  - No reducers; no data transfer between the map and reduce phase
  - Most of the map tasks pull data off of locally attached disks and then write back out to that node
  - Both the sort phase and the reduce phase are cut out

Filtering 4/4

- Usage: DistributedGrep <regex> <in> <out>
  - DistributedGrep '(.*)MacBook(.*)' inDir outDir
- Driver: job.setNumReduceTasks(0);

```
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);
        }
    }
}
```

Top Ten 1/5

- Intent
  - Retrieve a relatively small number of top K records, according to a ranking scheme, no matter how large the data
- Motivation
  - To find the best records for a specific criterion
- Applicability
  - comparator function: compare two records to determine which is “larger”
  - # of output records should << # of input records, or else total ordering of the data set makes more sense

Top Ten 2/5

- Structure
  - Mapper: find local top K
  - (only one) Reducer: K*M records ➔ the final top K

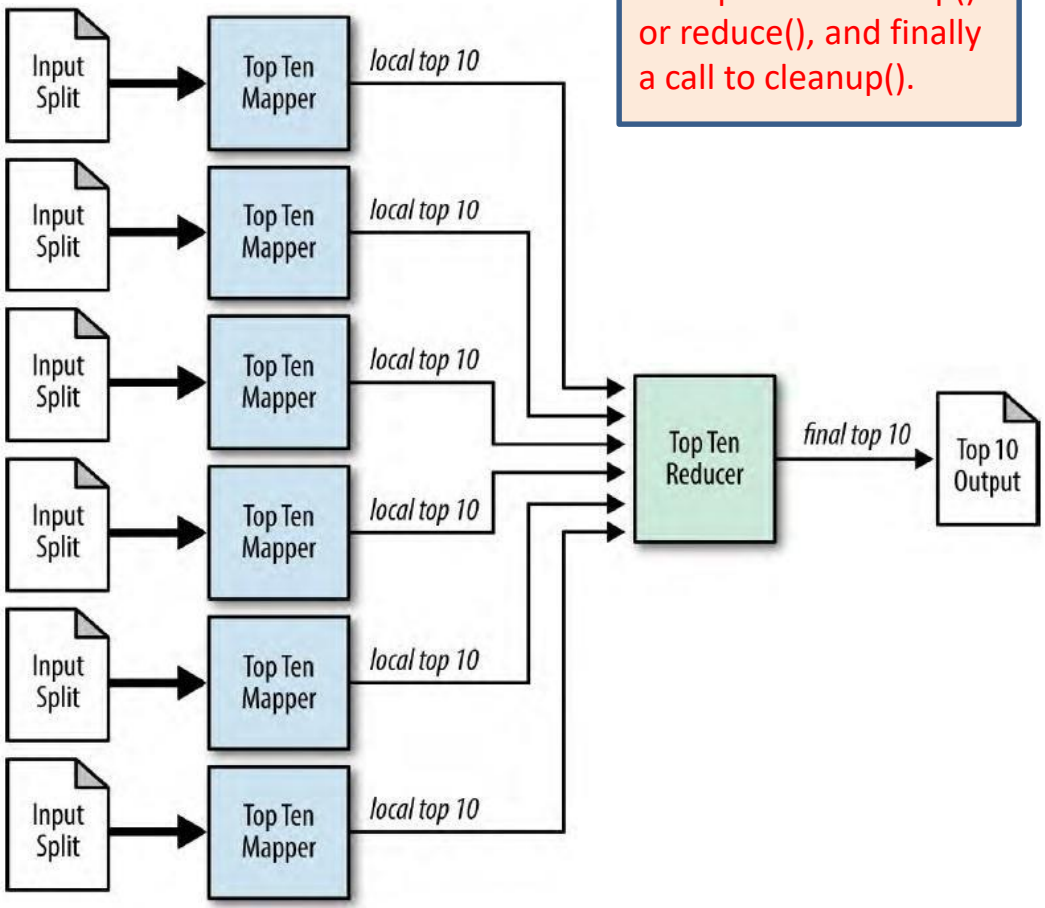
```
class mapper:
   setup():
      initialize top ten sorted list

   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
```



For **each** map or reduce **task**, setup() is called once, followed by multiple calls to map() or reduce(), and finally a call to cleanup().

Top Ten 3/5

- Known uses
  - Outlier analysis
  - Select interesting data (most valuable data)
  - Catchy dashboards
- SQL

  SELECT * FROM table WHERE col4 DESC LIMIT 10
- Performance analysis – one single Reducer gets K*M records
  - sort can be expensive: most of sorting done on local disk, instead of in memory
  - The reducer host receives a lot of data: network resource hot spot
  - Scanning through all map output in the reduce takes time
  - Any sort of memory growth in the reducer has the possibility of blowing through the Java virtual machine's memory
  - Writes to the output file are not parallelized
  - As K gets large, this pattern becomes less efficient
    - Optimization?

Top Ten 4/5

- TopTenDriver.java
- In: Users.xml

```
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);
    }
  }
}
```

Top Ten 5/5

```
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);
    }
  }
}
```

Distinct 1/4

- Intent
  - To find a unique set of values from similar records
- Motivation
  - Reduce a data set to a unique set of values
- Applicability
  - You have duplicates values in data set; it is silly to use this pattern otherwise

Distinct 2/4

- exploits MapReduce's ability to group keys together to remove duplicates
- Structure
  - Mapper: transforms the data and output (record, null)
  - Combiner: deduplicates duplicated records (that are usually located close to another)
  - Reducer: groups nulls together by key ➔ simply output (key, null)
    - any order is not preserved due to the random partitioning of the records

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

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

Distinct 3/4

- Known uses
  - Deduplicate data
  - Getting distinct values
  - Protecting from an inner join explosion
- SQL

  SELECT DISTINCT * FROM table;
- Performance analysis
  - # of reducers does not matter (why?)
    - Set # of reducers relatively high
  - If duplicates are rare within an input split, Mappers forward almost all their data to the reducers
  - Combiner optimization (how?)

Distinct 4/4

- DistinctUserDriver.java: get distinct user IDs
- In: Comments.xml

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

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

References

- Donald Miner and Adam Shook, *MapReduce Design Patterns*.
  - http://oreil.ly/mapreduce-design-patterns
  - https://github.com/adamjshook/mapreducepatterns