**Questions from Chapter 3 of textbook: Data-Intensive Text Processing**

**with MapReduce**

**Aparna Pavithran**

**Read Chapter 3 and answer following questions:**

# **Introduction**

**Answer:**

**a. Synchronization is the trickiest aspect of designing MR algorithms? Where do you get a chance to perform this? What techniques does a programmer have to control execution and manage the flow of data in MR?**

Synchronization is perhaps the trickiest aspect of designing MapReduce algorithms (or for that matter, parallel and distributed algorithms in general). Other than embarrassingly parallel problems, processes running on separate nodes in a cluster must, at some point in time, come together. For example, to distribute partial results from nodes that produced them to the nodes that will consume them. Within a single Map-Reduce job, there is only one opportunity for cluster-wide synchronization - during the shuffle and sort stage where intermediate key-value pairs are copied from the mappers to the reducers and grouped by key. Beyond that, mappers and reducers run in isolation without any mechanisms for direct communication.

The programmer does have a number of techniques for controlling execution and managing the flow of data in MapReduce.

1. The ability to construct complex data structures as keys and values to store and communicate partial results.

2. The ability to execute user-specified initialization code at the beginning of a map or reduce task, and the ability to execute user-specified termination code at the end of a map or reduce task.

3. The ability to preserve state in both mappers and reducers across multiple input or intermediate keys.

4. The ability to control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys.

5. The ability to control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer.

# **3.1 Local Aggregation**

**a. How does local aggregation lead to an increase in algorithmic efficiency?**

**Answer:**

In a cluster environment, with the exception of embarrassingly parallel problems, this necessarily involves transferring data over the network. Furthermore, in Hadoop, intermediate results are written to local disk before being sent over the network. Since network and disk latencies are relatively expensive compared to other operations, reductions in the amount of intermediate data translate into increases in algorithmic efficiency. In MapReduce, local aggregation of intermediate results is one of the keys to efficient algorithms. Through use of the combiner and by taking advantage of the ability to preserve state across multiple inputs, it is often possible to substantially reduce both the number and size of key-value pairs that need to be shuffled from the mappers to the reducers.

**b. What are two techniques for local aggregation?**

**Answer:**

The first technique for local aggregation is the combiner. Combiners provide a general mechanism within the MapReduce framework to reduce the amount of intermediate data generated by the mappers. They can be understood as “mini-reducers” that process the output of mappers. This results in a reduction in the number of intermediate key-value pairs that need to be shuffled across the network from the order of total number of terms in the collection to the order of the number of unique terms in the collection.

The second technique is in-mapper combining. With this technique, we are in essence incorporating combiner functionality directly inside the mapper. There is no need to run a separate combiner, since all opportunities for local aggregation are already exploited.

**c. What is meant by "in-mapper combining"? What are the two advantages of this design pattern? What are the two disadvantages?**

**Answer:**

The second technique is in-mapper combining. With this technique, we are in essence incorporating combiner functionality directly inside the mapper. There is no need to run a separate combiner, since all opportunities for local aggregation are already exploited.

There are two main advantages to using this design pattern:

First, it provides control over when local aggregation occurs and how it exactly takes place. In contrast, the semantics of the combiner is underspecified in MapReduce. For example, Hadoop makes no guarantees on how many times the combiner is applied, or that it is even applied at all. The combiner is provided as a semantics-preserving optimization to the execution framework, which has the option of using it, perhaps multiple times, or not at all (or even in the reduce phase). In some cases (although not in this particular example), such indeterminism is unacceptable, which is exactly why programmers often choose to perform their own local aggregation in the mappers.

Second, in-mapper combining will typically be more efficient than using actual combiners. One reason for this is the additional overhead associated with actually materializing the key-value pairs. Combiners reduce the amount of intermediate data that is shuffled across the network, but don't actually reduce the number of key-value pairs that are emitted by the mappers in the first place. With this intermediate key-value pairs are still generated on a per-document basis, only to be compacted" by the combiners. This process involves unnecessary object creation and destruction (garbage collection takes time), and furthermore, objects serialization and deserialization (when intermediate key-value pairs fill the in-memory buffer holding map outputs and need to be temporarily spilled to disk). In contrast, with in-mapper combining, the mappers will generate only those key-value pairs that need to be shuffled across the network to the reducers.

There are, however, drawbacks to the in-mapper combining pattern. First, it breaks the functional programming underpinnings of MapReduce, since state is being preserved across multiple input key-value pairs. Preserving state across multiple input instances means that algorithmic behavior may depend on the order in which input key-value pairs are encountered. This creates the potential for ordering-dependent bugs, which are difficult to debug on large datasets in the general case (although the correctness of in-mapper combining for word count is easy to demonstrate). Second, there is a fundamental scalability bottleneck associated with the in-mapper combining pattern. It critically depends on having sufficient memory to store intermediate results until the mapper has completely processed all key-value pairs in an input split.

**d. In order to achieve algorithmic correctness, the following should be true: (Fill in the blanks)**

**Answer:**

Reducer Input (K, V) = Mapper **Output (K, V)** = Combiner **Input (K, V)** = Combiner **Output (K, V)**

**e. In the algorithms 3.4 to 3.7, the textbook is illustrating how the input/output (K, V) of combiner and mapper must be modified to compute the mean of values associated with the same key. Since the mean function is not associative and commutative, you cannot use the reducer as the combiner. Identify which of the following operations are commutative and associative?**

**Answer:**

- finding max value - Yes

- finding min value - Yes

- finding product of values - Yes

- finding the median of the values - No

- finding 1st and 3rd quartile of values - No  
(If you don't know what they are, see here: <http://web.mnstate.edu/peil/MDEV102/U4/S36/S363.html> )

# **3.2 Pairs and Stripes**

**a. What are some applications of building a term co-occurrence matrix?**

**Answer:**

As a running example, we focus on the problem of building word co-occurrence matrices from large corpora, a common task in corpus linguistics and statistical natural language processing. Formally, the co-occurrence matrix of a corpus is a square n X n matrix where n is the number of unique words in the corpus (i.e., the vocabulary size). A cell mij contains the number of times word wi co-occurs with word wj within a specific context, a natural unit such as a sentence, paragraph, or a document, or a certain window of m words (where m is an application-dependent parameter). The upper and lower triangles of the matrix are identical since co-occurrence is a symmetric relation, though in the general case relations between words need not be symmetric. For example, a co-occurrence matrix M where mij is the count of how many times word i was immediately succeeded by word j would usually not be symmetric.

This task is quite common in text processing and provides the starting point to many other algorithms, e.g., for computing statistics such as point wise mutual information, for unsupervised sense clustering, and more generally, a large body of work in lexical semantics based on distributional profiles of words. The task also has applications in information retrieval, and other related fields such as text mining. More importantly, this problem represents a specific instance of the task of estimating distributions of discrete joint events from a large number of observations, a very common task in statistical natural language processing for which there are nice MapReduce solutions.

Beyond text processing, problems in many application domains share similar characteristics. For example, a large retailer might analyze point-of-sale transaction records to identify correlated product purchases (e.g., customers who buy this tend to also buy that), which would assist in inventory management and product placement on store shelves. Similarly, an intelligence analyst might wish to identify associations between re-occurring financial transactions that are otherwise unrelated, which might provide a clue in thwarting terrorist activity.

**b. Read the pairs and stripes approach carefully and understand their (K, V) pairs.**

**Algorithm 3.9 shows the details of the stripes approach. Notice that the associative array is defined within the map method. What could be the problem if it is defined within the initialize method (like shown in Algorithm 3.3 for in-mapper combining). Hint: Associative array for a term can grow pretty large and it will have to be stored in memory.**

**What is a possible solution?**

**Answer:**

For both algorithms, the in-mapper combining optimization can also be applied. However, there will be far fewer opportunities for partial aggregation in the pairs approach due to the sparsity of the intermediate key space. The sparsity of the key space also limits the effectiveness of in-memory combining, since the mapper may run out of memory to store partial counts before all documents are processed, necessitating some mechanism to periodically emit key-value pairs (which further limits opportunities to perform partial aggregation). Similarly, for the stripes approach, memory management will also be more complex than in the simple word count example. For common terms, the associative array may grow to be quite large, necessitating some mechanism to periodically flush in-memory structures.

It is important to consider potential scalability bottlenecks of either algorithm.

The stripes approach makes the assumption that, at any point in time, each associative array is small enough to fit into memory, otherwise memory paging will significantly impact performance. The size of the associative array is bounded by the vocabulary size, which is itself unbounded with respect to corpus size. Therefore, as the sizes of corpora increase, this will become an increasingly pressing issue, perhaps not for gigabyte-sized corpora, but certainly for terabyte-sized and petabyte-sized corpora that will be commonplace tomorrow. The pairs approach, on the other hand, does not suffer from this limitation, since it does not need to hold intermediate data in memory.

Below are previously published results on both the approach. Results demonstrate that the stripes approach is much faster than the pairs approach: 666 seconds (11 minutes) compared to 3758 seconds (62 minutes) for the entire corpus (improvement by a factor of 5.7). The mappers in the pairs approach generated 2.6 billion intermediate key-value pairs totaling 31.2 GB. After the combiners, this was reduced to 1.1 billion key-value pairs, which quantifies the amount of intermediate data transferred across the network. In the end, the reducers emitted a total of 142 million final key-value pairs. On the other hand, the mappers in the stripes approach generated 653 million intermediate key-value pairs totaling 48.1 GB. After the combiners, only 28.8 million key-value pairs remained. The reducers emitted a total of 1.69 million final key-value pairs (the number of rows in the co-occurrence matrix). As expected, the stripes approach provided more opportunities for combiners to aggregate intermediate results, thus greatly reducing network traffic in the shuffle and sort phase.

# **3.3 Computing Relative Frequencies**

**a. What is the drawback of absolute counts? What is meant by marginal of a word count?**

**Answer:**

The drawback of absolute counts is that it doesn't take into account the fact that some words appear more frequently than others. Word wi may co-occur frequently with wj simply because one of the words is very common. A simple remedy is to convert absolute counts into relative frequencies, f(wj, wi). That is, what proportion of the time does wj appear in the context of wi?

Marginal word count is the sum of the counts of the conditioning variable co-occurring with anything else.

**b. Between the pairs and stripes approach, which one makes relative frequency computation easier?**

**Answer:**

Computing relative frequencies with the stripes approach is straightforward. In the reducer, counts of all words that co-occur with the conditioning variable (wi in the above example) are available in the associative array. Therefore, it suffices to sum all those counts to arrive at the marginal (i.e., Sum of w N(wi;w)), and then divide all the joint counts by the marginal to arrive at the relative frequency for all words. Through appropriate structuring of keys and values, one can use the MapReduce execution framework to bring together all the pieces of data required to perform a computation.

In the pairs approach, the reducer receives (wi;wj) as the key and the count as the value. From this alone it is not possible to compute f(wj, wi) since we do not have the marginal. Fortunately, as in the mapper, the reducer can preserve state across multiple keys. Inside the reducer, we can buffer in memory all the words that co-occur with wi and their counts, in essence building the associative array in the stripes approach. To make this work, we must define the sort order of the pair so that keys are first sorted by the left word, and then by the right word. Given this ordering, we can easily detect if all pairs associated with the word we are conditioning on (wi) have been encountered. At that point we can go back through the in-memory buffer, compute the relative frequencies, and then emit those results in the final key-value pairs.

**c. If you want to compute relative frequency using the pairs approach, what are some modifications that need to be performed?**

**Answer:**

In the pairs approach, the reducer receives (wi, wj) as the key and the count as the value. From this alone it is not possible to compute f(wj, wi) since we do not have the marginal. Fortunately, as in the mapper, the reducer can preserve state across multiple keys. Inside the reducer, we can buffer in memory all the words that co-occur with wi and their counts, in essence building the associative array in the stripes approach. To make this work, we must define the sort order of the pair so that keys are first sorted by the left word, and then by the right word. Given this ordering, we can easily detect if all pairs associated with the word we are conditioning on (wi) have been encountered. At that point we can go back through the in-memory buffer, compute the relative frequencies, and then emit those results in the final key-value pairs.

**d. Using the pairs approach, how can you compute marginal of a word before the joint counts?**

**Answer:**

If it were possible to somehow compute (or otherwise obtain access to) the marginal in the reducer before processing the joint counts, the reducer could simply divide the joint counts by the marginal to compute the relative frequencies. The notion of “before" and “after" can be captured in the ordering of key-value pairs, which can be explicitly controlled by the programmer. That is, the programmer can define the sort order of keys so that data needed earlier is presented to the reducer before data that is needed later. However, we still need to compute the marginal counts. Recall that in the basic pairs algorithm, each mapper emits a key-value pair with the co-occurring word pair as the key. To compute relative frequencies, we modify the mapper so that it additionally emits a “special" key of the form (wi; \*), with a value of one that represents the contribution of the word pair to the marginal. Through use of combiners, these partial marginal counts will be aggregated before being sent to the reducers. Alternatively, the in-mapper combining pattern can be used to even more efficiently aggregate marginal counts.

In the reducer, we must make sure that the special key-value pairs representing the partial marginal contributions are processed before the normal key-value pairs representing the joint counts. This is accomplished by defining the sort order of the keys so that pairs with the special symbol of the form (wi; \*) are ordered before any other key-value pairs where the left word is wi. In addition, as with before we must also properly define the partitioner to pay attention to only the left word in each pair. With the data properly sequenced, the reducer can directly compute the relative frequencies.

**e. What is the order inversion design pattern? Outline the steps for using order inversion for relative frequency calculation?**

**Answer:**

This design pattern, which we call “order inversion", occurs surprisingly often and across applications in many domains. It is so named because through proper coordination, we can access the result of a computation in the reducer (for example, an aggregate statistic) before processing the data needed for that computation. The key insight is to convert the sequencing of computations into a sorting problem. In most cases, an algorithm requires data in some fixed order: by controlling how keys are sorted and how the key space is partitioned, we can present data to the reducer in the order necessary to perform the proper computations. This greatly cuts down on the amount of partial results that the reducer needs to hold in memory. To summarize, the specific application of the order inversion design pattern for computing relative frequencies requires the following:

* Emitting a special key-value pair for each co-occurring word pair in the mapper to capture its contribution to the marginal.
* Controlling the sort order of the intermediate key so that the reducer before any of the pairs representing the joint word co-occurrence counts processes the key-value pairs representing the marginal contributions.
* Defining a custom partitioner to ensure that all pairs with the same left word are shuffled to the same reducer.
* Preserving state across multiple keys in the reducer to first compute the marginal based on the special key-value pairs and then dividing the joint counts by the marginal to arrive at the relative frequencies.

# **3.4 Secondary Sorting**

**a. We know that one way of sorting by value is to use an in-memory data structure at the reducer.** **There is a memory bottleneck while doing this. Another approach is to let the MR framework take care of the sorting by a part of the value. This is known as secondary sorting or "value-to-key" design pattern. What modifications are needed to execute this design pattern?**

**Answer:**

The basic idea is to move part of the value into the intermediate key to form a composite key, and let the MapReduce execution framework handle the sorting. In the sensor data example, instead of emitting the sensor id as the key, we would emit the sensor id and the timestamp as a composite key:

(m1; t1) ! (r80521)

The sensor reading itself now occupies the value. We must define the intermediate key sort order to first sort by the sensor id (the left element in the pair) and then by the timestamp (the right element in the pair). We must also implement a custom partitioner so that all pairs associated with the same sensor are shuffled to the same reducer. Properly orchestrated, the key-value pairs will be presented to the reducer in the correct sorted order:

(m1; t1) ! [(r80521)]

(m1; t2) ! [(r21823)]

(m1; t3) ! [(r146925)]

. . .

. . .

However, note that sensor readings are now split across multiple keys. The reducer will need to preserve state and keep track of when readings associated with the current sensor end and the next sensor begin. With value-to-key conversion, sorting is offloaded to the MapReduce execution framework. This pattern results in many more keys for the framework to sort, but distributed sorting is a task that the MapReduce runtime excels at since it lies at the heart of the programming model.