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

Textbook can be downloaded from: https://lintool.github.io/MapReduceAlgorithms/

Read Chapter 3 and answer following questions:

Introduction

a. Synchronization is the most tricky 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 most tricky 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 MapReduce 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. Furthermore, the programmer has litter control over many aspects of execution, for example:

1. Where a mapper or reducer runs (i.e., on which node in the cluster).
2. When a mapper or reducer begins or finishes
3. Which input key-value pairs are processed by a specific mapper.
4. Which intermediate key-value pairs are processed by a specific reducer.

Nevertheless, the programmer does have a number of techniques for controlling execution and managing the flow of data in MapReduce. They are:

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?

The most important aspect of synchronization is the exchange of intermediate results, from the processes that produced them to the processes that will ultimately consume them. 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?

Combiners and In-mapper Combining

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

The in-mapper combiner aggregates data but do not write to local disk: They occur in-memory in the Mapper itself.

Advantages:
First, it provides control over then local aggregation occurs and how it exactly takes place. Second, in-mapper combining will typically be more efficient than using actual combiners.
Drawbacks:
First, it breaks the functional programming underpinnings of MapReduce, since state is being preserved across multiple input key-value pairs. 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)

Reducer Input (K, V) = Mapper ___output___ (K, V) = Combiner ___output ___ (K, V) = Combiner ___intput_____ (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?

- finding max value （commutative and associative）
- finding min value （commutative and associative）
- finding product of values （commutative and associative）
- 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

In corpus linguistics and statistical natural language processing, text processing.

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?

Combiners with the stripes approach have more opportunities to perform local aggregation because the key space is the vocabulary-asoociative arrays can be merged whenever a word is encountered multiple times by a mapper. In contrast, the key space in the pairs approach is the cross of the vocabulary with itself, which is far larger – counts can be aggregated only when the same co-occurring word pair is observed multiple times by an individual mapper (which is less likely than observing multiple occurrences of a words, as in the stripes cases)/

So the in-mapper combining optimization cann be applied.

3.3 Computing Relative Frequencies

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

It doesn’t take into account the fact that some words appear more frequently than others. It means the computation divides all the joint counts by the marginal to arrive at the relative frequency for all words.

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

Stripes

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

How might one compute relative frequencies with the pairs approach? 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?

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), and then divide all the joint counts by the marginal to arrive at the relative frequency for all words

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

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 key-value pairs representing the marginal contributions are processed by the reducer before any of the pairs representing the joint word co-occurrence counts. • 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 marginals 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.

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.