Lesson 3

MapReduce Algorithm Design

Pros and Cons

In addition to preparing the input data, the programmer needs only to implement the mapper, the reducer, and optionally, the combiner and the partitioner.

However, this also means that any conceivable algorithm that a programmer wishes to develop must be expressed in terms of a small number of rigidly-defined components that must fit together in very specific ways.

LOCAL AGGREGATION

The first technique for local aggregation is the combiner.

In this example, the combiners aggregate term counts across the documents processed by each map task.

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. Why?

LOCAL AGGREGATION

An improvement on the basic algorithm is shown next. Only mapper is modified. An associative array (i.e., Map in Java) is introduced inside the mapper to

tally up term counts within a single document: instead of emitting a key-value pair for each term in the document, this version emits a key-value pair for each unique term in the document. Given that some words appear frequently within a document, this can yield substantial savings in the number of intermediate key-value pairs emitted, especially for long documents.

LOCAL AGGREGATION

```
class Mapper
    method Map(docid a; doc d)
        H =  new AssociativeArray
        for all term t in doc d do
            H{t} =  H{t} + 1 . //Tally counts for entire doc
        for all term t in H do
            Emit(term t; count H{t})
```

LOCAL AGGREGATION

This basic idea can be taken one step further, as shown next (once again, only the mapper is modified). Recall, a (Java) mapper object is created for each map task, which is responsible for processing a block of input key-value pairs. Prior to processing any input key-value pairs, the mapper's Initialize method is called, which is an API hook for user-specified code. In this case, we initialize an associative array for holding term counts.

LOCAL AGGREGATION

Since it is possible to preserve state across multiple calls of the Map method (for each input key-value pair), we can continue to accumulate partial term counts in the associative array across multiple documents, and emit key-value pairs only when the mapper has processed all documents. That is, emission of intermediate data is deferred until the Close method in the pseudo-code. Recall that this API hook provides an opportunity to execute user-specified code after the Map method has been applied to all input key-value pairs of the input data split to which the map task was assigned.

LOCAL AGGREGATION

```
class Mapper
    method Initialize
        H = new AssociativeArray
    method Map(docid a; doc d)
        for all term t in doc d do
            H{t} = H{t} + 1 .        //Tally counts across docs
    method Close
        for all term t in H do
            Emit(term t; count H{t})
```

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. This is a sufficiently common design pattern in MapReduce called, "in-mapper combining". There are two main advantages to using this design pattern:

Advantages of the in-mapper combining pattern

First, it provides control over when local aggregation occurs and how it exactly takes place.

In contrast, the semantics of the combiner is under speciffied 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.

Advantages of the in-mapper combining pattern

Second, in-mapper combining will typically be more efficient than using actual combiners.

One reason 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.

Advantages of the in-mapper combining pattern

This process involves unnecessary object creation and destruction (garbage collection takes time), and furthermore, object 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.

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).

Drawbacks to the in-mapper combining pattern

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.

Drawbacks to the in-mapper combining pattern

In the word count example, the memory footprint is bound by the vocabulary size, since it is theoretically possible that a mapper encounters every term in the collection. Heap's Law, a well-known result in information retrieval, accurately models the growth of vocabulary size as a function of the collection size - the somewhat surprising fact is that the vocabulary size never stops growing. Therefore, the algorithm in Figure 3.3 will scale only up to a point, beyond which the associative array holding the partial term counts will no longer fit in memory

Drawbacks to the in-mapper combining pattern

One common solution to limiting memory usage when using the in-mapper combining technique is to “block" input key-value pairs and “flush" in-memory data structures periodically. Instead of emitting intermediate data only after every key-value pair has been processed, emit partial results after processing every n key-value pairs. This is straightforwardly implemented with a counter variable that keeps track of the number of input key-value pairs that have been processed.

Drawbacks to the in-mapper combining pattern

As an alternative, the mapper could keep track of its own memory footprint and flush intermediate key-value pairs once memory usage has crossed a certain threshold. In both approaches, either the block size or the memory usage threshold needs to be determined empirically: with too large a value, the mapper may run out of memory, but with too small a value, opportunities for local aggregation may be lost.

Drawbacks to the in-mapper combining pattern

In Hadoop physical memory is split between multiple tasks that may be running on a node concurrently; these tasks are all competing for finite resources, but since the tasks are not aware of each other, it is difficult to coordinate resource consumption effectively. In practice, however, one often encounters diminishing returns in performance gains with increasing buffer sizes, such that it is not worth the effort to search for an optimal buffer size.

Drawbacks to the in-mapper combining pattern

In MapReduce algorithms, the extent to which efficiency can be increased through local aggregation depends on the size of the intermediate key space, the distribution of keys themselves, and the number of key-value pairs that are emitted by each individual map task. Opportunities for aggregation come from having multiple values associated with the same key (whether one uses combiners or employs the in-mapper combining pattern). In the word count example, local aggregation is effective because many words are encountered multiple times within a map task.

Drawbacks to the in-mapper combining pattern

Local aggregation is also an effective technique for dealing with reduce stragglers (see Section 2.3) that result from a highly-skewed (e.g., Zipfian) distribution of values associated with intermediate keys. In our word count example, we do not filter frequently-occurring words: therefore, without local aggregation, the reducer that's responsible for computing the count of `the' will have a lot more work to do than the typical reducer, and therefore will likely be a straggler. With local aggregation we substantially reduce the number of values associated with frequently-occurring terms, which alleviates the reduce straggler problem.

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

Although use of combiners can yield dramatic reductions in algorithm running time, care must be taken in applying them. Since combiners in Hadoop are viewed as optional optimizations, the correctness of the algorithm cannot depend on computations performed by the combiner or depend on them even being run at all. In any MapReduce program, the reducer input key-value type must match the mapper output key-value type: this implies that the combiner input and output key-value types must match the mapper output key-value type (which is the same as the reducer input key-value type).

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

In cases where the reduce computation is both commutative and associative, the reducer can also be used (unmodified) as the combiner (as is the case with the word count example). In the general case, however, combiners and reducers are not interchangeable.

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

Consider a simple example: we have a large dataset where input keys are strings and input values are integers, and we wish to compute the mean of all integers associated with the same key (rounded to the nearest integer). A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session - the task would correspond to computing the mean session length on a per-user basis, which would be useful for understanding user demographics.

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

Figure 3.4 shows the pseudo-code of a simple algorithm for accomplishing this task that does not involve combiners. We use an identity mapper, which simply passes all input key-value pairs to the reducers (appropriately grouped and sorted). The reducer keeps track of the running sum and the number of integers encountered. This information is used to compute the mean once all values are processed. The mean is then emitted as the output value in the reducer (with the input string as the key).

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

```
class Mapper
    method Map(string t; integer r)
        Emit(string t; integer r)
class Reducer
    method Reduce(string t; integers [r1; r2; …])
        sum = 0;  cnt = 0
        for all integer r in integers [r1; r2; ...] do
            sum = sum + r;   cnt = cnt + 1
        ravg = sum / cnt
        Emit(string t; integer ravg)
```

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

This algorithm will indeed work, but suffers from the same drawbacks as the basic word count algorithm in Figure 3.1. it requires shuffling all key-value pairs from mappers to reducers across the network, which is highly inefficient. Unlike in the word count example, the reducer cannot be used as a combiner in this case. Consider what would happen if we did: the combiner would compute the mean of an arbitrary subset of values associated with the same key, and the reducer would compute the mean of those values.

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

As a concrete example, we know that:

Mean(1; 2; 3; 4; 5) ≠ Mean(Mean(1; 2);Mean(3; 4; 5))

In general, the mean of means of arbitrary subsets of a set of numbers is not the same as the mean of the set of numbers. Therefore, this approach would not produce the correct result.

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

So how to take advantage of combiners? An attempt is shown in Figure 3.5. The mapper remains the same. Added a combiner that partially aggregates results by computing the numeric components necessary to arrive at the mean. The combiner receives each string and the associated list of integer values, from which it computes the sum and count. The sum and count are packaged into a pair, and emitted as the output of the combiner, with the same string as the key. In the reducer, pairs of partial sums and counts can be aggregated to arrive at the mean. Till now, all keys and values in our algorithms have been primitives.

ALGORITHMIC CORRECTNESS WITH LOCAL AGGREGATION

However, there are no prohibitions in MapReduce for more complex types, and, in fact, this represents a key technique in MapReduce algorithm design that we introduced at the beginning of this chapter. We will frequently encounter complex keys and values throughput the rest of this book.

Figure 3.5

```
class Mapper
    method Map(string t; integer r)
        Emit(string t; integer r)
class Combiner
    method Combine(string t; integers [r1; r2; …])
        sum = 0, cnt = 0
        for all integer r in integers [r1; r2; ...] do
            sum = sum + r, cnt = cnt + 1
        Emit(string t; pair (sum; cnt)) .
```

Figure 3.5 (cont'd)

```
class Reducer
    method Reduce(string t; pairs [(s1; c1); (s2; c2) : : :])
        sum = 0
        cnt = 0
        for all pair (s; c) in pairs [(s1; c1); (s2; c2) : : :] do
            sum = sum + s
            cnt  = cnt + c
        ravg = sum/cnt
        Emit(string t; integer ravg)
```

Figure 3.5 comments

Unfortunately, this algorithm will not work. Recall that combiners must have the same input and output key-value type, which also must be the same as the mapper output type and the reducer input type. This is clearly not the case. To understand why this restriction is necessary in the programming model, remember that combiners are optimizations that cannot change the correctness of the algorithm.

Figure 3.5 comments

So let us remove the combiner and see what happens: the output value type of the mapper is integer, so the reducer expects to receive a list of integers as values. But the reducer actually expects a list of pairs! The correctness of the algorithm is contingent on the combiner running on the output of the mappers, and more specifically, that the combiner is run exactly once. Recall from our previous discussion that Hadoop makes no guarantees on how many times combiners are called; it could be zero, one, or multiple times.

The corrected version

Figure 3.6. In the mapper we emit as the value a pair consisting of the integer and one - this corresponds to a partial count over one instance. The combiner separately aggregates the partial sums and the partial counts (as before), and emits pairs with updated sums and counts. The reducer is similar to the combiner, except that the mean is computed at the end. In essence, this algorithm transforms a non-associative operation (mean of numbers) into an associative operation (element-wise sum of a pair of numbers, with an additional division at the very end).

Figure 3.6

```
class Mapper
    method Map(string t; integer r)
        Emit(string t; pair (r; 1))
class Combiner
    method Combine(string t; pairs [(s1; c1); (s2; c2); …])
        sum = 0, cnt = 0
        for all pair (s; c) 2 pairs [(s1; c1); (s2; c2); …] do
            sum =  sum + s, cnt =  cnt + c
        Emit(string t; pair (sum; cnt))
```

Figure 3.6 (cont'd)

```
class Reducer
    method Reduce(string t; pairs [(s1; c1); (s2; c2); …])
        sum = 0, cnt = 0
        for all pair (s; c) in pairs [(s1; c1); (s2; c2); …] do
            sum = sum + s, cnt = cnt + c
        ravg = sum/cnt
        Emit(string t; integer ravg)
```

Figure 3.6 comments

What would happen if no combiners were run? With no combiners, the mappers would send pairs (as values) directly to the reducers. There would be as many intermediate pairs as there were input key-value pairs, and each of those would consist of an integer and one. The reducer would still arrive at the correct sum and count, and hence the mean would be correct. Now add in the combiners: the algorithm would remain correct, no matter how many times they run, since the combiners merely aggregate partial sums and counts to pass along to the reducers.

Figure 3.6 comments

Note that although the output key-value type of the combiner must be the same as the input key-value type of the reducer, the reducer can emit final key-value pairs of a different type.

In Figure 3.7, we present an even more efficient algorithm that exploits the in-mapper combining pattern. Inside the mapper, the partial sums and counts associated with each string are held in memory across input key-value pairs.

Figure 3.6 comments

Intermediate key-value pairs are emitted only after the entire input split has been processed; similar to before, the value is a pair consisting of the sum and count. The reducer is exactly the same as in Figure 3.6. Moving partial aggregation from the combiner directly into the mapper is subjected to all the tradeoffs and caveats discussed earlier this section, but in this case the memory footprint of the data structures for holding intermediate data is likely to be modest, making this variant algorithm an attractive option.