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.

PAIRS AND STRIPES

One common approach for synchronization in MapReduce is to construct complex keys and values in such a way that data necessary for a computation are naturally brought together by the execution framework. We first touched on this technique in the previous section, in the context of “packaging" partial sums and counts in a complex value (i.e., pair) that is passed from mapper to combiner to reducer.

This section introduces two common design patterns we have dubbed “pairs" and “stripes" that exemplify this strategy.

Example - Building word co-occurrence

Problem of building word co-occurrence matrices from large corpora, is a common task in corpus linguistics and statistical natural language processing. Formally, the co-occurrence matrix of a corpus is a square n × n

matrix where n is the number of unique words in the corpus (i.e., the vocabulary size). A cell m_{ij} contains the number of times the i-th word co-occurs with j-th word 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).

Example - Building word co-occurrence

Note that 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 m_{ij} is the count of how many times i-th word was immediately succeeded by j-th word would usually not be symmetric.

Example - Building word co-occurrence

This task is quite common in text processing and provides the starting point to many other algorithms. 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. Indeed, concepts presented here are also used in Chapter 6 when we discuss expectation-maximization algorithms.

Example - Building word co-occurrence

Beyond text processing, problems in many application domains share similar characteristics. 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. 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.

Example - Building word co-occurrence

It is obvious that the space requirement for the word co-occurrence problem is $O(n^2)$, where n is the size of the vocabulary, which for English corpora can be hundreds of thousands of words, or even billions of words in web-scale collections. The computation of the word co-occurrence matrix is quite simple if the entire matrix fits into memory. Although compression techniques can increase the size of corpora for which word co-occurrence matrices can be constructed on a single machine, it is clear that there are inherent scalability limitations.

Pairs approach solution

As usual, document ids and the corresponding contents make up the input key-value pairs. The mapper processes each input document and emits intermediate key-value pairs with each co-occurring word pair as the key and the integer one (i.e., the count) as the value. This is straightforwardly accomplished by two nested loops: the outer loop iterates over all words (the left element in the pair), and the inner loop iterates over all neighbors of the first word (the right element in the pair).

Pairs approach solution

The neighbors of a word can either be defined in terms of a sliding window or some other contextual unit such as a sentence. The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Thus, in this case the reducer simply sums up all the values associated with the same co-occurring word pair to arrive at the absolute count of the joint event in the corpus,

which is then emitted as the final key-value pair. Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.

Pairs approach – Figure 3.8

```
class Mapper
    method Map(docid a; doc d)
        for all term w in doc d do
            for all term u in Neighbors(w) do
                Emit(pair (w; u); count 1) .
class Reducer
    method Reduce(pair p; counts [c1; c2; …])
        s = 0
        for all count c in counts [c1; c2; …] do
            s = s + c .
        Emit(pair p; count s)
```

Stripes approach

Like the pairs approach, co-occurring word pairs are generated by two nested loops. However, the major difference is that instead of emitting intermediate key-value pairs for each co-occurring word pair, co-occurrence information is first stored in an associative array, denoted H. The mapper emits key-value pairs with words as keys and corresponding associative arrays as values, where each associative array encodes the co-occurrence counts of the neighbors of a particular word (i.e., its context).

Stripes approach

The MapReduce execution framework guarantees that all associative arrays with the same key will be brought together in the reduce phase of processing. The reducer performs an element-wise sum of all associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. The final associative array is emitted with the same word as the key. In contrast to the pairs approach, each final key-value pair encodes a row in the co-occurrence matrix.

Stripes approach – Figure 3.9

```
class Mapper
    method Map(docid a; doc d)
        for all term w in doc d do
            H = new AssociativeArray
            for all term u in Neighbors(w) do
                H{u} = H{u} + 1 .        //Tally words co-occurring with w
            Emit(Term w; Stripe H)
class Reducer
    method Reduce(term w; stripes [H1;H2;H3; : : :])
        Hf  = new AssociativeArray
        for all stripe H in stripes [H1;H2;H3; …] do
            Sum(Hf; H) .                  //Element-wise sum
        Emit(term w; stripe Hf )
```

Pairs or Stripes

It is immediately obvious that the pairs algorithm generates an immense number of key-value pairs compared to the stripes approach. The stripes representation is much more compact, since with pairs the left element is repeated for every co-occurring word pair. The stripes approach also generates fewer and shorter intermediate keys, and therefore the execution framework has less sorting to perform. However, values in the stripes approach are more complex, and come with more serialization and deserialization overhead than with the pairs approach.

Pairs or Stripes

Both algorithms can benefit from the use of combiners, since the respective operations in their reducers (addition and element-wise sum of associative arrays) are both commutative and associative. However, combiners with the stripes approach have more opportunities to perform local aggregation because the key space is the vocabulary - associative arrays can be merged whenever a word is encountered multiple times by a mapper.

Pairs or Stripes

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 word, as in the stripes case)

Pairs or Stripes

For both algorithms, the in-mapper combining optimization discussed in the previous section can also be applied; the modification is sufficiently straightforward that we leave the implementation as an exercise for the reader. However, the above caveats remain: there will be far fewer opportunities for partial aggregation in the pairs approach due to the sparsity of the intermediate key space.

Pairs or Stripes

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.

Pairs or Stripes

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 (recall the previous discussion of Heap's Law).

Pairs or Stripes

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.

Pairs or Stripes

Given this discussion, which approach is faster? Here, we present previously published results that empirically answered this question.

They have implemented both algorithms in Hadoop and applied them to a corpus of 2.27 million documents from the Associated Press Worldstream (APW) totaling 5.7 GB. Prior to working with Hadoop, the corpus was first preprocessed as follows: All XML markup was removed, followed by tokenization and stop word removal using standard tools from the Lucene search engine.

Pairs or Stripes

All tokens were then replaced with unique integers for a more efficient encoding. Figure 3.10 compares the running time of the pairs and stripes approach on different fractions of the corpus, with a co-occurrence window size of two. These experiments were performed on a Hadoop cluster with 19 slave nodes, each with two single-core processors and two disks.

Pairs or Stripes

Results demonstrate that the stripes approach is much faster than the pairs approach:

- 11 minutes compared to 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 (the number of non-zero cells in the co-occurrence matrix).

Pairs or Stripes

- 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. Figure 3.10 also shows that both algorithms exhibit highly desirable scaling characteristics - linear (input size)

Pairs or Stripes

Viewed abstractly, the pairs and stripes algorithms represent two different approaches to counting co-occurring events from a large number of observations. This general description captures the gist of many algorithms in fields as diverse as text processing, data mining, and bioinformatics. For this reason, these two design patterns are broadly useful and frequently observed in a variety of applications.

Pairs or Stripes

To conclude, it is worth noting that the pairs and stripes approaches represent endpoints along a continuum of possibilities. The pairs approach individually records

each co-occurring event, while the stripes approach records all co-occurring events with respect a conditioning event. A middle ground might be to record a subset of the co-occurring events with respect to a conditioning event.

Pairs or Stripes

We might divide up the entire vocabulary into b buckets (e.g., via hashing), so that words co-occurring with wi would be divided into b smaller “sub-stripes", associated with b separate keys, (wi; 1); (wi; 2) : : : (wi; b). This would be a reasonable solution to the memory limitations of the stripes approach, since each of the sub-stripes would be smaller. In the case of b = |V|, where |V| is the vocabulary size, this is equivalent to the pairs approach. In the case of b = 1, this is equivalent to the standard stripes approach.

COMPUTING RELATIVE FREQUENCIES

Let us build on the pairs and stripes algorithms presented in the previous section and continue with our running example of constructing the word co-occurrence matrix M for a large corpus. Recall that in this large square n x n matrix, where n = |V| (the vocabulary size), cell mij contains the number of times word wi co-occurs with word wj within a specific context. 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.

COMPUTING RELATIVE FREQUENCIES

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? This can be computed using the following equation:

$$f(w_j \mid w_i) = N(w_i, w_j) / \sum_{w'} N(w_i, w')$$

Here, N(.,.) indicates the number of times a particular co-occurring word pair is observed in the corpus. We need the count of the joint event (word co-occurrence), divided by what is known as the marginal (the sum of the counts of the conditioning variable co-occurring with anything else).

COMPUTING RELATIVE FREQUENCIES

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., $\sum_{w'} N(wi, w')$)

and then divide all the joint counts by the marginal to arrive at the relative frequency for all words. This implementation requires minimal modification to the original stripes algorithm in Figure 3.9, and

COMPUTING RELATIVE FREQUENCIES

illustrates the use of complex data structures to coordinate distributed computations in MapReduce. 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. Note that, as with before, this algorithm also assumes that each associative array fits into memory.

COMPUTING RELATIVE FREQUENCIES

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.

COMPUTING RELATIVE FREQUENCIES

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.

COMPUTING RELATIVE FREQUENCIES

There is one more modification necessary to make this algorithm work. We must ensure that all pairs with the same left word are sent to the same reducer. This, unfortunately, does not happen automatically: recall that the default partitioner is based on the hash value of the intermediate key, modulo the number of reducers.

COMPUTING RELATIVE FREQUENCIES

For a complex key, the raw byte representation is used to compute the hash value. As a result, there is no guarantee that, for example, (dog, aardvark) and (dog, zebra) are assigned to the same reducer. To produce the desired behavior, we must define a custom partitioner that only pays attention to the left word. That is, the partitioner should partition based on the hash of the left word only.

COMPUTING RELATIVE FREQUENCIES

This algorithm will indeed work, but it suffers from the same drawback as the stripes approach: as the size of the corpus grows, so does that vocabulary size, and at some point there will not be sufficient memory to store all co-occurring words and their counts for the word we are conditioning on. For computing the co-occurrence matrix, the advantage of the pairs approach is that it doesn't suffer from any memory bottlenecks. Is there a way to modify the basic pairs approach so that this advantage is retained?

COMPUTING RELATIVE FREQUENCIES

Such an algorithm exists, although it requires the coordination of several mechanisms in MapReduce. The insight lies in properly sequencing data presented to the reducer. If it were possible to somehow compute (or 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.

COMPUTING RELATIVE FREQUENCIES

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

COMPUTING RELATIVE FREQUENCIES

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.

COMPUTING RELATIVE FREQUENCIES

A concrete example is shown in Figure 3.12, which lists the sequence of key-value pairs that a reducer might encounter. First, the reducer is presented with the special key (dog; *) and a number of values, each of which represents a partial marginal contribution from the map phase (assume here either combiners or in-mapper combining, so the values represent partially aggregated counts). The reducer accumulates these counts to arrive at the marginal, $\sum_{w'} N(dog, w')$ The reducer holds on to this value as it processes subsequent keys.

COMPUTING RELATIVE FREQUENCIES

(dog, *)	[10,30]	$\sum_{w'} N(dog, w') = 40$
(dog, cat)	[2,1]	f(cat | dog) = 3/40 = 0.075
(dog, rat)	[4]	f(rat | dog) = 4/40 = 0.1
…		
(dog, zebra)	[1,2,2,1]	f(zebra | dog) = 6/40 = 0.15
(doge, *)	[20, 10, 40]	$\sum w' N(doge, w') = 70$

COMPUTING RELATIVE FREQUENCIES

After (dog, *), the reducer will encounter a series of keys representing joint counts; let's say the first of these is the key (dog; aardvark). Associated with this key will be a list of values representing partial joint counts from the map phase (two separate values in this case). Summing these counts will yield the final joint count, i.e., the number of times dog and aardvark co-occur in the entire collection. At this point, since the reducer already knows the marginal, simple arithmetic suffices to compute the relative frequency. All subsequent joint counts are processed in exactly the same manner.

COMPUTING RELATIVE FREQUENCIES

When the reducer encounters the next special key-value pair (doge; *), the reducer resets its internal state and starts to accumulate the marginal all over again.

Observe that the memory requirement for this algorithm is minimal, since only the

marginal (an integer) needs to be stored. No buffering of individual co-occurring word counts is necessary, and therefore we have eliminated the scalability bottleneck of the previous algorithm.

Order Inversion

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.

Order Inversion

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:

Order Inversion

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.

Order Inversion

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.

Secondary Sorting

MapReduce sorts intermediate key-value pairs by the keys during the shuffle and sort phase, which is very convenient if computations inside the reducer rely on sort order (e.g., the order inversion design pattern described in the previous section). However, what if in addition to sorting by key, we also need to sort by value? Google's MapReduce implementation provides built-in functionality for (optional) secondary sorting, which guarantees that values arrive in sorted order. Hadoop, unfortunately, does not have this capability built in.

Secondary Sorting

Consider the example of sensor data from a scientific experiment: there are m sensors each taking readings on continuous basis, where m is potentially a large number. A dump of the sensor data might look something like the following, where rx after each

timestamp represents the actual sensor readings (unimportant for this discussion, but may be a series of values, one or more complex records, or even raw bytes of images).

Secondary Sorting

(t1;m1; r80521)

(t1;m2; r14209)

(t1;m3; r76042)

…

(t2;m1; r21823)

(t2;m2; r66508)

(t2;m3; r98347)

Suppose we wish to reconstruct the activity at each individual sensor over time.

Secondary Sorting

A MapReduce program to accomplish this might map over the raw data and emit the sensor id as the intermediate key, with the rest of each record as the value:

m1 → (t1, r80521)

This would bring all readings from the same sensor together in the reducer.

Secondary Sorting

However, since MapReduce makes no guarantees about the ordering of values associated with the same key, the sensor readings will not likely be in temporal order. The most obvious solution is to buffer all the readings in memory and then sort by timestamp before additional processing. However, it should be apparent by now that any in-memory buffering of data introduces a potential scalability bottleneck. What if we are working with a high frequency sensor or sensor readings over a long period of time? This approach may not scale and reducer would run out of memory trying to buffer all values associated with the same key.

Value-to-Key Conversion

This is a common problem, since in many applications we wish to first group together data one way (e.g., by sensor id), and then sort within the groupings by another value (e.g., by time). There is a general purpose solution, which we call the “value-to-key conversion" 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. In the above 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)

Value-to-Key Conversion

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)]

…

Value-to-Key Conversion

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. The basic trade off between the two approaches discussed above (buffer and in-memory sort vs. value-to-key conversion) is where sorting is performed. One can explicitly implement secondary sorting in the reducer, which is likely to be faster but suffers from a scalability bottleneck.

Value-to-Key Conversion

With value-to-key conversion, sorting is offloaded to the MapReduce execution framework. Note that this approach can be arbitrarily extended to tertiary, quaternary, etc. sorting. 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.

SUMMARY

“In-mapper combining", where the functionality of the combiner is moved into the mapper. Instead of emitting intermediate output for every input key-value pair,

the mapper aggregates partial results across multiple input records and only emits intermediate key-value pairs after some amount of local aggregation is performed.

SUMMARY

The related patterns “pairs" and “stripes" for keeping track of joint events from a large number of observations. In the pairs approach, we keep track of each joint event separately, whereas in the stripes approach we keep track of all events that co-occur with the same event. Although the stripes approach is signicantly more efficient, it requires memory on the order of the size of the event space, which presents a scalability bottleneck.

SUMMARY

“Order inversion", where the main idea is to convert the sequencing of computations into a sorting problem. Through careful orchestration, we can send the reducer the result of a computation (e.g., an aggregate statistic) before it encounters the data necessary to produce that computation.

“Value-to-key conversion", which provides a scalable solution for secondary sorting. By moving part of the value into the key, we can exploit the MapReduce execution framework itself for sorting.

SUMMARY

Ultimately, controlling synchronization in the MapReduce programming model boils down to effective use of the following techniques:

1. Constructing complex keys and values that bring together data necessary for a computation. This is used in all of the above design patterns.
2. Executing user-specified initialization and termination code in either the mapper or reducer. For example, in-mapper combining depends on emission of intermediate key-value pairs in the map task termination code

SUMMARY

3. Preserving state across multiple inputs in the mapper and reducer. This is used in in-mapper combining, order inversion, and value-to-key conversion.
4. Controlling the sort order of intermediate keys. This is used in order inversion and value-to-key conversion.
5. Controlling the partitioning of the intermediate key space. This is used in order inversion and value-to-key conversion.

(Optional)

"Hadoop-The definitive guide by Tom White"

Each map task has a circular memory buffer that it writes the output to. The buffer is 100 MB by default (the size can be tuned by changing the mapreduce.task.io.sort.mb property). When the contents of the buffer reach a certain threshold size (mapreduce.map.sort.spill.percent, which has the default value 0.80, or 80%), a background thread will start to spill the contents to disk.

Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete. Spills are written in round-robin fashion to the directories specified by the mapreduce.cluster.local.dir property, in a job-specific subdirectory. Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort.

Running the combiner function makes for a more compact map output, so there is less data to write to local disk and to transfer to the reducer.“

So if this sorted result is given as input to the combiner then the combiner output must be sorted. For example in the W1D4 solution , cell (**Combiner 1 – Input Split 2)**

<rat, 1>, <cat, 2>, <bat, 1>

must read

<bat, 1>, <cat, 2>, <rat, 1>

and so on.