blue [3, 8, 4, 19]

Class Mapper
method map (string t, integer r)
EMIT (string t, pair (r, 1))

---

class Combiner (string t, pairs [(s1, c1), (s2, c2), (s3, ...)])

sum ← 0
count ← 0
for all pair (s, c) in pairs
sum ← sum + s
count ← count + c
EMIT (string t, pair (sum, count))

---

class Reducer (string t, pairs [(s1, c1), (s2, c2)..

sum ← 0
count ← 0
for all pair (s, c) in pairs
sum ← sum + s
count ← count + c
Average ← sum / count
EMIT (String t, Integer Average)

CSE4/587 Exam 1 Data-intensive Computing Spring 2017

Name: _____

CSE487 CSE587 (circle one)

1. (20 points) Given the following text data made up of the following sentences, process it using the classical "wordcount" MapReduce program. Provide the output <k,v> pairs

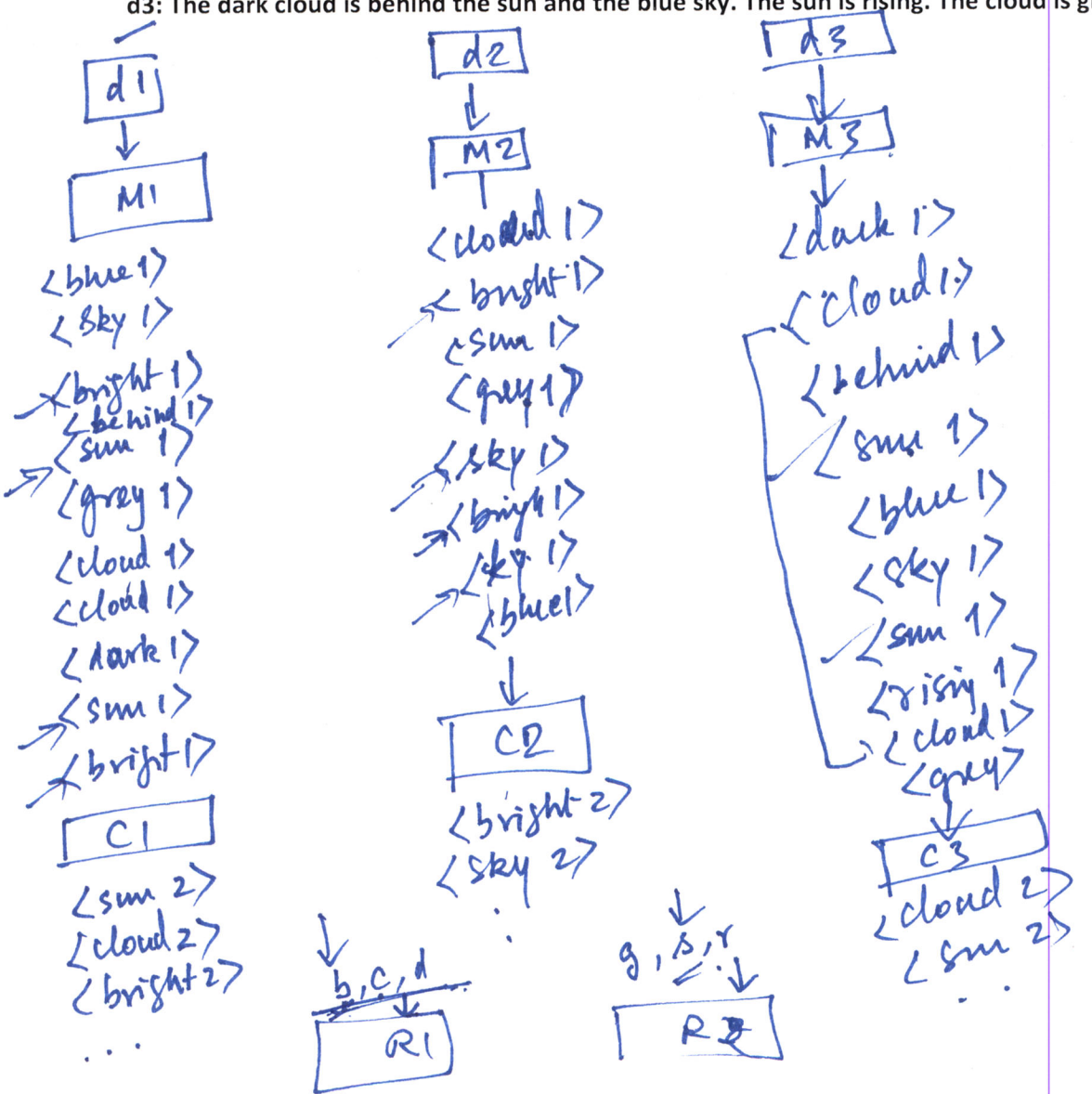
(i) at the output of Mappers,
(ii) at the output of Combiners,
(iii) at the input of the Reducers and
(iv) at the output of Reducers.

Assume 3 Mappers and 2 Reducers. Assume stop words {the, The, a, is, are, and}

d1: The blue sky and bright sun are behind the grey cloud. The cloud is dark and the sun is bright.

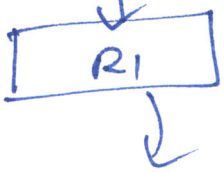
d2: The cloud is bright and the sun is grey. The sky is bright. The Sky is blue.

d3: The dark cloud is behind the sun and the blue sky. The sun is rising. The cloud is grey.



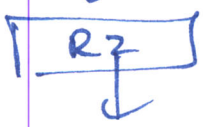
c, d

<bright [2,2]>
<blue [1, 1, 1]>
<behind [1, 1]>



<behind 2>
<blue 3>
<bright 4>

<sky [1, 2 1]>
<sun [2, 1, 2]>



<sky 4>
<sun 5>

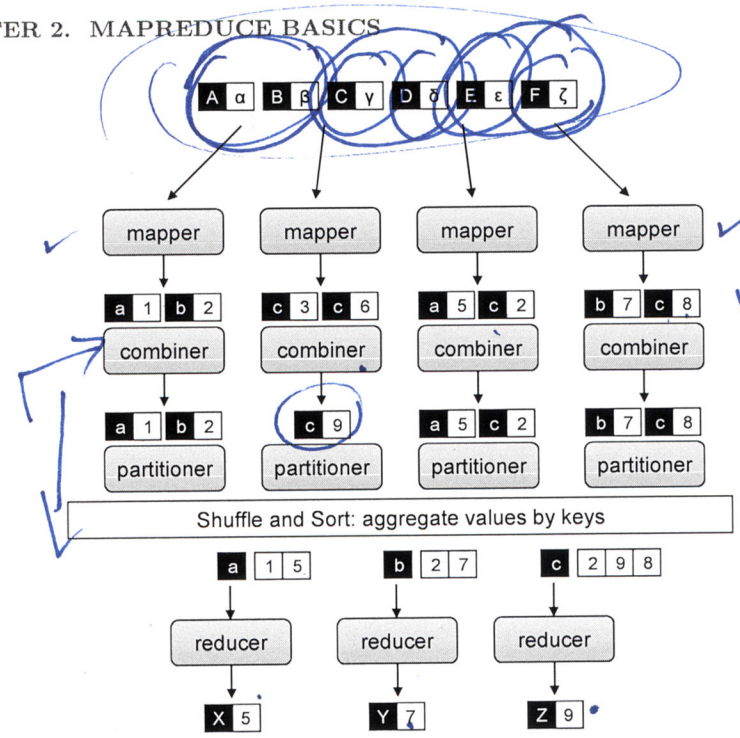


Figure 2.4: Complete view of MapReduce, illustrating combiners and partitioners in addition to mappers and reducers. Combiners can be viewed as "mini-reducers" in the map phase. Partitioners determine which reducer is responsible for a particular key.

a combiner can significantly reduce the amount of data that needs to be copied over the network, resulting in much faster algorithms.

The complete MapReduce model is shown in Figure 2.4. Output of the mappers are processed by the combiners, which perform local aggregation to cut down on the number of intermediate key-value pairs. The partitioner determines which reducer will be responsible for processing a particular key, and the execution framework uses this information to copy the data to the right location during the shuffle and sort phase.[13] Therefore, a complete MapReduce job consists of code for the mapper, reducer, combiner, and partitioner, along with job configuration parameters. The execution framework handles everything else.

[13] In Hadoop, partitioners are actually executed before combiners, so while Figure 2.4 is conceptually accurate, it doesn't precisely describe the Hadoop implementation.

- Section 3.3 shows how co-occurrence counts can be converted into relative frequencies using a pattern known as "order inversion". The sequencing of computations in the reducer can be recast as a sorting problem, where pieces of intermediate data are sorted into exactly the order that is required to carry out a series of computations. Often, a reducer needs to compute an aggregate statistic on a set of elements before individual elements can be processed. Normally, this would require two passes over the data, but with the "order inversion" design pattern, the aggregate statistic can be computed in the reducer before the individual elements are encountered. This may seem counter-intuitive: how can we compute an aggregate statistic on a set of elements before encountering elements of that set? As it turns out, clever sorting of special key-value pairs enables exactly this.

- Section 3.4 provides a general solution to secondary sorting, which is the problem of sorting values associated with a key in the reduce phase. We call this technique "value-to-key conversion".

- Section 3.5 covers the topic of performing joins on relational datasets and presents three different approaches: *reduce-side*, *map-side*, and *memory-backed* joins.

3.1 LOCAL AGGREGATION

In the context of data-intensive distributed processing, the single 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.

3.1.1 COMBINERS AND IN-MAPPER COMBINING

We illustrate various techniques for local aggregation using the simple word count example presented in Section 2.2. For convenience, Figure 3.1 repeats the pseudo-code of the basic algorithm, which is quite simple: the mapper emits an intermediate key-value pair for each term observed, with the term itself as the key and a value of one; reducers sum up the partial counts to arrive at the final count.

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Figure 3.1: Pseudo-code for the basic word count algorithm in MapReduce (repeated from Figure 2.3).

The first technique for local aggregation is the combiner, already discussed in Section 2.4. Combiners provide a general mechanism within the MapReduce framework to reduce the amount of intermediate data generated by the mappers—recall that they can be understood as "mini-reducers" that process the output of mappers. 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.[1]

An improvement on the basic algorithm is shown in Figure 3.2 (the mapper is modified but the reducer remains the same as in Figure 3.1 and therefore is not repeated). 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 (for example, a document about dogs is likely to have many occurrences of the word "dog"), this can yield substantial savings in the number of intermediate key-value pairs emitted, especially for long documents.

[1] More precisely, if the combiners take advantage of all opportunities for local aggregation, the algorithm would generate at most $m \times V$ intermediate key-value pairs, where m is the number of mappers and V is the vocabulary size (number of unique terms in the collection), since every term could have been observed in every mapper. However, there are two additional factors to consider. Due to the Zipfian nature of term distributions, most terms will not be observed by most mappers (for example, terms that occur only once will by definition only be observed by one mapper). On the other hand, combiners in Hadoop are treated as *optional* optimizations, so there is no guarantee that the execution framework will take advantage of all opportunities for partial aggregation.

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

Figure 3.2: Pseudo-code for the improved MapReduce word count algorithm that uses an associative array to aggregate term counts on a per-document basis. Reducer is the same as in Figure 3.1.

This basic idea can be taken one step further, as illustrated in the variant of the word count algorithm in Figure 3.3 (once again, only the mapper is modified). The workings of this algorithm critically depends on the details of how map and reduce tasks in Hadoop are executed, discussed in Section 2.6. 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. 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.

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.[2] This is a sufficiently common design pattern in MapReduce that it's worth giving it a name, "in-mapper combining", so that we can refer to the pattern more conveniently throughout the book. We'll see later on how this pattern can be applied to a variety of problems. 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.

[2]Leaving aside the minor complication that in Hadoop, combiners can be run in the reduce phase also (when merging intermediate key-value pairs from different map tasks). However, in practice it makes almost no difference either way.

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Figure 3.3: Pseudo-code for the improved MapReduce word count algorithm that demonstrates the "in-mapper combining" design pattern. Reducer is the same as in Figure 3.1.

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 the algorithm in Figure 3.2, 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, 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.

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. Ultimately, this isn't a big deal, since pragmatic concerns for efficiency often trump theoretical "purity", but there are practical consequences as well. 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

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)
1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Figure 3.4: Pseudo-code for the basic MapReduce algorithm that computes the mean of values associated with the same key.

of values associated with the same key, and the reducer would compute the mean of those values. As a concrete example, we know that:

$$\text{Mean}(1, 2, 3, 4, 5) \neq \text{Mean}(\text{Mean}(1, 2), \text{Mean}(3, 4, 5))$$

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.[5]

So how might we properly take advantage of combiners? An attempt is shown in Figure 3.5. The mapper remains the same, but we have 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 of those values and the number of integers encountered (i.e., the 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. Up until now, all keys and values in our algorithms have been primitives (string, integers, etc.). However, there are no prohibitions in MapReduce for more complex types,[6] and, in fact, this represents a key technique in MapReduce algorithm design that we introduced at the beginning of this

[5]There is, however, one special case in which using reducers as combiners *would* produce the correct result: if each combiner computed the mean of equal-size subsets of the values. However, since such fine-grained control over the combiners is impossible in MapReduce, such a scenario is highly unlikely.

[6]In Hadoop, either custom types or types defined using a library such as Protocol Buffers, Thrift, or Avro.

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Combiner
2:     method Combine(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))                ▷ Separate sum and count

1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Figure 3.5: Pseudo-code for an incorrect first attempt at introducing combiners to compute the mean of values associated with each key. The mismatch between combiner input and output key-value types violates the MapReduce programming model.

chapter. We will frequently encounter complex keys and values throughput the rest of this book.

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. 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

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))

1: class Combiner
2:     method Combine(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))

1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Figure 3.6: Pseudo-code for a MapReduce algorithm that computes the mean of values associated with each key. This algorithm correctly takes advantage of combiners.

how many times combiners are called; it could be zero, one, or multiple times. This violates the MapReduce programming model.

Another stab at the algorithm is shown in Figure 3.6, and this time, the algorithm is correct. 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).

Let us verify the correctness of this algorithm by repeating the previous exercise: 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

```
1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S{t} ← S{t} + r
7:         C{t} ← C{t} + 1
8:     method Close
9:         for all term t ∈ S do
10:            Emit(term t, pair (S{t}, C{t}))
```

Figure 3.7: Pseudo-code for a MapReduce algorithm that computes the mean of values associated with each key, illustrating the in-mapper combining design pattern. Only the mapper is shown here; the reducer is the same as in Figure 3.6

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. 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.

Finally, 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. 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.

3.2 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. Building on previously

published work [54, 94], this section introduces two common design patterns we have dubbed "pairs" and "stripes" that exemplify this strategy.

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 $n \times n$ 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 word w_i co-occurs with word w_j 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). 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 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 pointwise mutual information [38], for unsupervised sense clustering [136], and more generally, a large body of work in lexical semantics based on distributional profiles of words, dating back to Firth [55] and Harris [69] in the 1950s and 1960s. The task also has applications in information retrieval (e.g., automatic thesaurus construction [137] and stemming [157]), 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. Indeed, concepts presented here are also used in Chapter 6 when we discuss expectation-maximization algorithms.

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. The algorithms discussed in this section could be adapted to tackle these related problems.

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 real-world English corpora can be hundreds of thousands of words, or even billions of words in web-scale collections.[7] The computation of the word co-occurrence matrix is quite simple if the entire matrix

[7]The size of the vocabulary depends on the definition of a "word" and techniques (if any) for corpus preprocessing. One common strategy is to replace all rare words (below a certain frequency) with a "special" token such as <UNK> (which stands for "unknown") to model out-of-vocabulary words. Another technique involves replacing numeric digits with #, such that 1.32 and 1.19 both map to the same token (#.##).

fits into memory—however, in the case where the matrix is too big to fit in memory, a naïve implementation on a single machine can be very slow as memory is paged to disk. 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. We describe two MapReduce algorithms for this task that can scale to large corpora.

Pseudo-code for the first algorithm, dubbed the "pairs" approach, is shown in Figure 3.8. 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). 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.

An alternative approach, dubbed the "stripes" approach, is presented in Figure 3.9. 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). 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.

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

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence

1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                              ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

Figure 3.8: Pseudo-code for the "pairs" approach for computing word co-occurrence matrices from large corpora.

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                    ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H1, H2, H3, ...])
3:         Hf ← new AssociativeArray
4:         for all stripe H ∈ stripes [H1, H2, H3, ...] do
5:             Sum(Hf, H)                             ▷ Element-wise sum
6:         Emit(term w, stripe Hf)
```

Figure 3.9: Pseudo-code for the "stripes" approach for computing word co-occurrence matrices from large corpora.

stripes approach are more complex, and come with more serialization and deserialization overhead than with the pairs approach.

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. 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).

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

Given this discussion, which approach is faster? Here, we present previously-published results [94] that empirically answered this question. We 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.[8] Prior to working

[8]This was a subset of the English Gigaword corpus (version 3) distributed by the Linguistic Data Consortium (LDC catalog number LDC2007T07).