Lesson 3

MapReduce Algorithm Design

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 $n \times n$ 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 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.