Lesson 3

MapReduce Algorithm Design

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, $f(wj \mid wi)$. That is, what proportion of the time does wj appear in the context of wi? This can be computed using the following equation:

$$f(wj \mid wi) = N(wi, wj)/\sum_{w'} N(wi, w')$$

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.