http://sonerbalkir.blogspot.com/2010/01/simulating-secondary-sort-on-values.html

Tuesday, January 26, 2010

Simulating Secondary Sort on Values with Hadoop

Sorting and grouping the records come free with Hadoop, it's how map reduce works. In some applications however, you may need to get the values at the reduce step in sorted order. This is not a default behavior and it may be a bit tricky at first glance, since the records coming from the mappers do not follow a deterministic order. I'll describe a sample problem and show how to achieve sorting the values using a trick that is already present in the API.

Consider the following problem: We have a corpus with many documents, and we would like to compute the ***document weight*** of each word for every document. We can formulate the document weight of a word ***w*** in document ***d*** as follows:

D(w,d) = (frequency of w in d) / (frequency of w in the corpus)

For simplicity, let's assume each document has a distinct positive document id, specified as ***doc_id***.

How would you compute the document weight of every word for each document in the corpus by a single map reduce job? Would it be enough to just emit pairs of the form **(word, doc_id)** and group these records inside the reducers? No, observe that there are two things missing in this approach.

First, for a particulcar word ***w***, we need to compute the denominator*(frequency of w in the corpus)* to perform any of the computations, and we don't know it yet.
Second, we need the keep track of these corpus frequencies - probably in an in memory data structure like an array or a hash table. What if we have billions of documents?

We obviously need a better solution that's scalable.
Let's get a bit more creative and emit two things inside the mappers.

```
map {
    for each word w observed in document doc_id {
        emit <word#0, 1>
        emit  <word#doc_id, 1>
    }
}
```

Notice that for each word, we emit two different pairs where each pair has a key created by concatenating two strings with a # character used for simplicity as a delimiter, ***word # number***

The first record always contains a 0 in its number field, we will use it to compute the denominator. The second field contains the corresponding document's id, which is guaranteed to be a positive number according to the input specifications.

It is now a good time to give an example to visualize things and better understand how the intermediate records look like.

Suppose we have two documents with id's 1 and 2, and let's assume the word '**olive**' occurs twice in document 1 and once in document 2. Here is how these documents look like:

.... **olive** ... .
.. .. **olive** ....

----- document 1-------

and

....

.... **olive** ...

----- document 2-------

The intermediate records from the mappers might then look like:

olive#0, 1
olive#1, 1
olive#0, 1
olive#1, 1
olive#0, 1
olive#2, 1

and since Hadoop would automatically sort them based on their keys - which in this case are strings - before the reduce step, they would look like:

olive#0, 1
olive#0, 1
olive#0, 1
olive#1, 1
olive#1, 1
olive#2, 1

Observe that this automatic sorting facility gave us an ordered collection of records based on their document id's for free! The only problem we have here right now is the keys olive#0, olive#1, and olive#2 being three different strings and hence they will go to different reducers. We want to guarantee that all the records of the form ***olive#X*** will go to the same reducer, namely the *olive reducer*.

In order to accomplish this, we'll change the default partitioner and overwrite it. The default partitioner works by simply hashing the whole key of each record.

```
int partition(record ) {
  return hash(key) % N ;
  // where N is the number of reducers
}
```

We can override it to make sure that any record of the form ***olive#X*** will go to the same reducer.

```
int partition(record ) {
  string real_key = key.substring(0,last index of '#');
  return hash(real_key) % N;
}
```

*You can use **org.apache.hadoop.mapreduce.Job's** **setPartitionerClass*** method to use a custom partitioner with a map reduce job.

Before we go any further, I should probably mention the comparators used by the Hadoop framework for a map/reduce job. There are two comparators used during different stages of the computation:

Sort Comparator Class: Used to control how the keys are sorted before the reduce step.
Grouping Comparator Class: Used to control which keys are a single call to reduce.

The default behavior is to use the same comparator class for both, but this can be customized via ***Job***'s ***setSortComparatorClass*** and ***setGroupingComparatorClass*** methods respectively.

Now back to our problem...

Overriding the partitioner ensures that all of the ***olive#X*** records will go to the same reducer, but it doesn't specify the order by which the groups are created. For example, we know ***olive#0***, ***olive#1*** and ***olive#2*** will go to the same reducer, but they are still treated as part of different groups, because the default grouping comparator - ***org.apache.hadoop.io.TextComparator*** - is used to decide which keys are a single call to reduce.

In order to resolve this tiny glitch, all we have to do is to write our own grouping comparator class which extends ***TextComparator*** and overrides the compare() method so that it only takes the first part of the key (up to #) into consideration. I'll skip the code for this step since it is more or less the same thing we did for the partitioner.

By using a custom partitioner and overriding the default sort comparator class, we guarantee that all of the ***olive#X*** records will go to the same reducer group, and they will be ordered by their document id's. There will be one group for each word ***w***, and all of the document id's that contain ***w*** will be processed by the same reducer starting from doc_id = 0 (which will be used to compute the denominator).

Observe how we take advantage of the two comparators in the framework. We use the **sort comparator class** to do a total ordering first, and then implement a custom **grouping comparator class** to simulate a secondary sort on values by changing the way the groups are created. I'm saying "simulate" because we don't actually perform a second sort on values, they come already sorted!

Finally, our reducer class should look like something similar to this:

```
public class Reducer {

  long denominator;

        public void reduce(key, values) {

        string word = key.substring(0, last index of '#');
        string doc_id = key.substring(last index of '#' + 1, key.length);
        if (doc_id == 0 ) {
```

```
            // compute the denominator for this word
            denominator = 0;
          for each v in values
              denominator += v;
          } else {

          // we already know the denominator for this word, use it
          long sum = 0;
       for each v in values
          sum += v;
      emit(key, sum / denominator);
          }
}
```

Please keep in mind that I concatenated two strings for the purpose of clarity, but in a real application this may result in performance problems. The best way to emit tuples (multiple attributes) as a key or value is to write custom WritableComparable objects containing multiple fields and implement the required methods including write, read, compareTo and equals. Also, using a combiner will obviously reduce the network traffic and boost the performance.