MapReduce examples

CSE 344 — section 8 worksheet

May 19, 2011

In today's section, we will be covering some more examples of using MapReduce to implement relational queries. Recall how MapReduce works from the programmer's perspective:

1. The input is a set of (key, value) pairs.
2. The map function is run on each (key, value) pair, producing a bag of intermediate (key, value) pairs:

   ```
   map (inkey, invalue):
     // do some processing on (inkey, invalue)
     emit_intermediate(hkey1, hvalue1)
     emit_intermediate(hkey2, hvalue2)
     // ...
   ```

3. The MapReduce implementation groups the intermediate (key, value) pairs by the intermediate key. Despite the name, this grouping is very different from the grouping operator of the relational algebra, or the `GROUP BY` clause of SQL. Instead of producing only the grouping key and the aggregate values, if any, MapReduce grouping also outputs a bag containing all the values associated with each value of the grouping key. In addition, grouping is separated from aggregation computation, which goes in the reduce function.
4. The reduce function is run on each distinct intermediate key, along with a bag of all the values associated with that key. It produces a bag of final values:

   ```
   reduce(hkey, hvalues[]):
     // do some processing on hkey, each element of hvalues[]
     emit(fvalue1)
     emit(fvalue2)
     // ...
   ```

Suppose you have two relations: $R(a,b)$ and $S(b,c)$, whose contents are stored as files on disk. Before you can process these relations using MapReduce, you need to parse each tuple in each relation as a (key, value) *pair*.

This task is relatively simple for our two relations, which only have two attributes to begin with. Let's treat $R.a$ as the "key" for tuples in R (note that this does not have to be a true key in the relational sense), and similarly, let's treat $S.b$ as the key for S.

For the "values" of the (key, value) pairs, we won't use the single non-key attribute. Instead, we'll use a composite value consisting of both attributes, plus a tag indicating where the value came from. So, for relation R, the values will have three components, value.tag, which is always the string "R", and value.a and value.b for the two attributes of R. Similarly, relation S's tuples will have MapReduce values containing value.tag, which is always "S", and value.b and value.c for S's attributes. This convention allows us to use the same `map` function for tuples from both relations; we'll see the use of that soon.

Given this notation for the (K,V) pairs of R and S, let's try to write pseudocode algorithms for the following relational operations:

1. Selecting tuples from R: $\sigma_{a<10}R$

   Solution:
   In this simple example, all the work is done in the map function, where we copy the input to the intermediate data, but only for tuples that meet the selection condition:

   ```
   map(inkey, invalue):
     if inkey < 10:
       emit_intermediate(inkey, invalue)
   ```

   Reduce then simply outputs all the values it is given:

   ```
   reduce(hkey, hvalues[]):
     for each t in hvalues:
       emit(t)
   ```

2. Eliminate duplicates from R: $\delta(R)$

   Solution:
   For this problem we will use a simple trick, described in your textbook — we'll use the fact that duplicate elimination in the bag relational algebra is equivalent to grouping on all attributes of the relation. MapReduce does grouping for us, so all we need is to make the entire tuple the intermediate key to group on.

```
map(inkey, invalue):
  // We won't use the intermediate value,
  // so we just put in a dummy value
  emit_intermediate(invalue, 'abc')
```

   Once we do that we just output the intermediate key as the final value:

```
reduce(hkey, hvalues[]):
  emit(hkey)
```

3. Natural join of R and S: $R \bowtie_{R.b=S.b} S$

   Solution:
   The map function outputs the same value as its input, but changes the key to always be the join attribute b:

```
map(inkey, invalue):
  emit_intermediate(invalue.B, invalue)
```

   Then, after the MapReduce system groups together the intermediate data by the intermediate key, i.e. the b values, we use the reduce function to do a nested loop join over each group. Because all the values from each group have the same join attribute, we don't check the join attribute in the nested loop. We do need to check which relation each tuple comes from, so that (for example) we don't join a tuple from R with itself, or with another R tuple.

```
reduce(hkey, hvalues[]):
  for each r in hvalues:
    for each s in hvalues:
      if r.tag = 'R' and s.tag = 'S':
        emit(r.a, r.b, s.c)
```

   Note that the analogy with the nested loop join breaks down in another way: instead of reading the relations directly from disk, we have used MapReduce to build the entire Cartesian product of R and S, and then grouped the Cartesian product by the join attribute b. So this is actually a very inefficient way to compute a join.

4. 3-way natural join: $R \bowtie_{R.b=S.b} S \bowtie_{S.c=T.c} T$, where we introduce a new relation $T(c, d)$.

   Solution:
   One way to do this join might be to split the join into two MapReduce jobs. The first job does one of the joins, using the previous algorithm for a single natural join. The second job then joins the result of the first job with the relation left over.

   This approach actually results in two different possible implementations, depending on which pair of relations we want to join first. One way to judge which method is "better" is to look at the number of tuples in each relation, and compare the relative sizes of the first job's output under each implementation. The smaller size for the first join is then considered better, because in our naïve view, smaller is faster — smaller data sets mean less network traffic, which is the real bottleneck of a distributed system like MapReduce (although the gap is narrowing rapidly).

5. Grouped and aggregated join: $\gamma_{(a,\text{sum}(c)\to s)}\left(R \bowtie_{R.b=S.b} S\right)$

   Solution:
   As with the three-way join, we'll probably want to use more than 1 MapReduce job for this problem. One approach uses 2 jobs: the first job does the 2-way join as we did in the problem above, then the second job does all the grouping and aggregation. Grouping is done automatically by MapReduce; all we have to do is to output the grouping attribute(s) as the intermediate key in map:

```
map (inkey, invalue):
  emit_intermediate(invalue.a, invalue)
```

   Then we can compute the sum aggregate in reduce:

```
reduce (hkey, hvalues[]):
  val sum := 0
  for each t in hvalues:
    sum += t.c
  emit(hkey, sum)
```

Another solution approach is based on the observation that you don't need to carry around all the tuples in the Cartesian product to compute the aggregate value sum(C). You do need all three distinct attributes, because the same tuple from S may appear many times in the join, due to several tuples in R that have the same b attribute). But you don't need every possible combination of values of all three attributes.

Instead, you can do the grouping and aggregate computation on S alone, grouping on $S.b$ instead of $R.a$. Then join the grouped results with R. Finally, repeat the grouping and aggregation (to take care of those S repeats), but with $R.a$ as the (correct) grouping attribute.

This sounds more complicated, and it is, using 3 MapReduce jobs instead of two. Can you think of a scenario where this 3-job algorithm might actually be faster than the 2-job algorithm?