Finding Top K using MapReduce

Overview

The Top K problem is defined as follows:

> Given a list of objects find K objects that have the highest value of a specific property all objects possess.

Examples. There are plenty of examples showing the need for solving the Top K problem. Some questions leading to this are below.

- Find 10 tallest basketball players in NBA.
- Find 20 products with the highest total sales volume across all branches of the store chain.
- Find the person with the highest salary in the organization.
- Find 30 students closest to graduation (based on their units counts).

Properties. The Top K problem has the following properties that make it challenging to implement it in MapReduce framework.

- **Globality.** This is a global problem in a sense that without observing all data it is impossible to produce the proper output.
- **Bad reduce-side algorithm.** One way to solve this problem is on the reduce side. This however (see below) is highly inefficient.

Solution Overview. The map-side solution to the Top K problem leverages the fact that the MapReduce mapper API has two more functions in addition to map(): setup() and cleanup().

Storing Top K records

Whether solve reduce-side or map-side, we need a data structure to *keep the top K records* while processing the data. The following can be used:

1. **Priority Queue.** The queue will store the Top K currently observed objects. We take advantage of size(), getMin(), removeMin() and insert() functions.
2. **Sorted List.** insert() must keep the list sorted. removeMin() pops the top element.
3. **Array.** If K is small, linear scans of the array to find the smallest element are going to be cheap.
4. **Hash Table.** Assuming we have the ability to linearly scan it.

Reduce-side solution

A straighforward solution that finds the Top K record is for map() to emit all records under the same key, and for reduce() to discover the top K records.

In pseudocode below, we assume that each input record stored in `record` has a field `v` which is used to produce the Top K records.

```
constant int K := .. ; // assume K is given.

function map(key, value) {

   emit(1, value);
}


function reduce(key, values) {

PriorityQueue queue := new PriorityQueue();  // initialize an empty priority queue

for record in values do

  if queue.size() <= K then   // at the beginning keep adding records to the queue
     queue.insert(record);

  else                       // once queue is full
     current := record.v;
     min := queue.getMin();

     if current > min then   // replace worst record if current record is better
        queue.removeMin();
        queue.insert(record);
     end if
  end if
end for

for r in queue do  // output result
  emit(r, null)
end for
}
```

Problem with reduce-side solution. Reduce() is the bottleneck here, because all objects are emitted with the same key.

Effectively, there is no distribution of work.

Mappers

For map-side solution, we need to expand our concept of the mapper.

Mapper API. We consider a MapReduce mapper to implement the following three functions:

1. map(key, value): accepts a key-value pair and emits (if necessary) new key-value pairs.
2. setup(): this function is run **once** for each input split *before* map() is run on each record of the split.
3. cleanup(): this function is run **once** for each input split *after* the last map() function is run on the final record of the split.

We also take the Object-Oriented view of the MapReduce mappers, and allow for instance variables representing necessary for data processing data structures to be present in the mapper, and *to be manipulated by* setup(), map() and cleanup().

Map-side Top K

Our map-side solution of the Top K problem proceeds as follows:

1. Split input data into multiple splits.
2. For each split, use a mapper to find and emit the Top K objects in the split.
3. Use reduce() to merge the Top K lists from each mapper instance and produce the overal Top K list.

Normally, K is much smaller than the size of the incoming dataset (N). If m is the number of splits, we can safely assume that $K \cdot m << N$, and therefore, only a fraction of data will be emitted from mappers to the reducer.

The pseudocode implementation of this is presented below.

```
class Mapper {

constant int K = ..;
PriorityQueue queue;

function setup() {

  queue:= new PriorityQueue();   // setup() simply initializes the priority queue
}

function map(key, value) {

   if queue.size < K then
       queue.insert(value);
   else
      current := value.v;
      min := queue.getMin();
      if current > min then
         queue.removeMin();
         queue.insert(value);
      end if
   end if
}

function cleanup() {
```

```
  for each r in queue do
     emit(1, r);
  end for
}
}  //Mapper

// Reducer class only requires use of reduce() function


function reduce(key, values) {

 PriorityQueue queue := new PriorityQueue();  // initialize an empty priority queue

for record in values do

  if queue.size() <= K then   // at the beginning keep adding records to the queue
     queue.insert(record);

  else                       // once queue is full
     current := record.v;
     min := queue.getMin();

     if current > min then   // replace worst record if current record is better
        queue.removeMin();
        queue.insert(record);
     end if
  end if
end for

for r in queue do  // output result
  emit(r, null)
end for
}
```

Note: The reduce() function for the map-side Top K is the same as for reduce-side Top K.

The key difference is how many records are passed out of the mapper and into the reducer.