CS6240 Parallel Data Processing Sec 01 Spring 2017

Harsha Jakkappanavar

Design Discussion:

The solution design for this following homework follows the "Overall Workflow Summary" mentioned in the HW3 pdf. There are three jobs that are identified and they all run from the same run method, the second job (refining the page ranks) runs 10 times. Please find below the pseudocode of the MapReduce program for each of these jobs.

Pre-processing:

```
// The map function takes in a line from the bz2 file, parses it
// using the WikiParser that is provided, strips off any path
// information from the beginning as well as the .html suffix
// from the end, leaving only the page name, also discards any
// pages and links containing a tilde character. the and creates
// a PageRankEntity object containing the adjacency list of
// outlinks.

// PageRankEntity{pageName, pageRank, outlinkSize, outlinks}

map(offset B, Line l){

     PageRankEntity p = parseLine(l);

     // emit the PageRankEntity object containing the graph
     // structure.

     emit({p.pageName, p.outlinkSize}, p);

     // emit all the pages in the outlink of the PageRankEntity
     // object, this takes care of the pages that appear in a
     // link, but the page itself does not exist in the data. We
     // treat it as a dangling node.

     for each page in p.outlinks

          emit({page, 0}, new PageRankEntity(page));

}
```

```
// Applying Secondary sort and defining a partitioner to make it
// sure that all the intermediate keys having the same page name
// arrives at the same reducer.

// Defining a key comaparator to sort the intermediate keys in
// the descending order of the outlinks’ size and a
// group comparator to group the keys by page name, ignoring the
// outlinks’ size.

// This helps in making sure that a PageRankObject with actual
// content is always in the beginning.(this object will have
// outlinkSize as 0 for the dangling node.)


// Intermediate values (p1, p2,…) is an Iterable of
// PageRankEntity objects.

// The reduce function picks the first object from the
// intermediate value iterator and emits it, the output format is
// as follows:

<page name>~<page rank(0.0 initially)>~<outlinks’ size>~<outlink 1>~<outlink 2>~…

reduce({pageName, outlinks’ size}, [p1, p2,…]){

     p = getFirstObjectFromIterator(intermediateValue);

     emit(p.toString());

}
```

Page Rank:

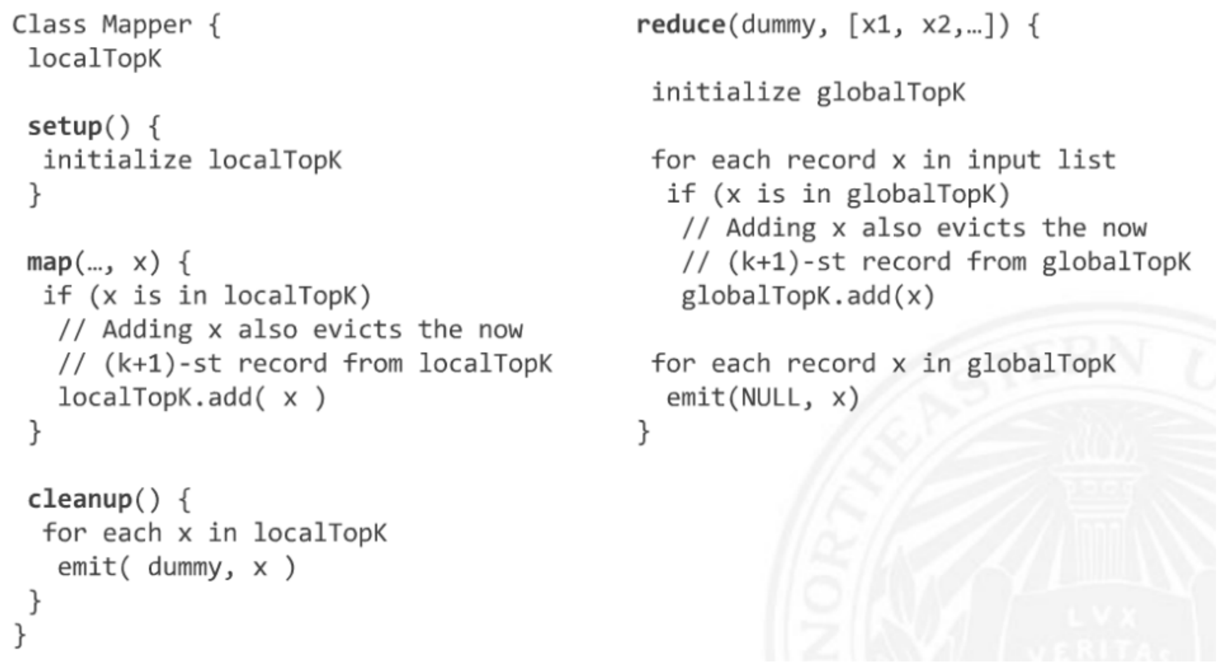
The algorithm for page rank calculation is similar to the one shared in the learning module, except that the page rank calculation at the reducer includes dangling factor and uses the formula mentioned in the learning module for Dangling nodes. The Here's the brief pseudocode:

```
map(offset B, Line l){
      PageRankEntity p = parseLine(l);
      // following the pseudocode from the learning module.
      emit(p.pageName, p);
      if outlinks(p) is 0
            // if this is a dangling node contribute the page rank
            // towards danglingFactor calculation
            emit("dummyDanglingPage", p.pageRank);
      else{
            // distribute the page rank among the outlinks.
            pr = p.pageRank / p.outlinkSize;
            for each outlink in p.outlinks
                  emit(outlink, pr);
      }
}
```

```
// Intermediate values (p1, p2,…) is an Iterable of
// (PageRankEntity objects | page rank values).

reduce(pageName(|“dummyDanglingPage”), [p1, p2,…]){

     if pageName is not “dummyDanglingPage”

          summation = 0;

          P = NULL;

          for all p in [p1, p2,…]

               if isPageRankEntity(p)

                    // recovering the graph structure

                    P = p;

               else

                    summation += p;

          // page rank is updated to P using the following
          // formula.

          $P(n) = \alpha \frac{1}{|V|} + (1-\alpha)\left(\frac{\delta}{|V|} + \sum_{m \in L(n)} \frac{P(m)}{C(m)}\right)$

          emit(P);

     else

          runningDanglingFactor = 0;

          for each page rank pr value in [p1, p2,…]

               runningDanglingFactor += pr;

          save the runningDanglingFactor onto counter for the
          next iteration

}
```

Top-k:

The top-k program follows the algorithm and code design from the learning module. Here's the pseudocode.



```
Class Mapper {
 localTopK

 setup() {
  initialize localTopK
 }

 map(…, x) {
  if (x is in localTopK)
   // Adding x also evicts the now
   // (k+1)-st record from localTopK
   localTopK.add( x )
 }

 cleanup() {
  for each x in localTopK
   emit( dummy, x )
 }
}
```

```
reduce(dummy, [x1, x2,…]) {

 initialize globalTopK

 for each record x in input list
  if (x is in globalTopK)
   // Adding x also evicts the now
   // (k+1)-st record from globalTopK
   globalTopK.add(x)

 for each record x in globalTopK
  emit(NULL, x)
}
```

Data transfer:

On the Machine 1 (1 master and 5 workers):

Iteration	Data transfer from Mappers to Reducers (bytes)	Data transfer from Reducer to HDFS (S3) (bytes)
1	1299454847	1130026513
2	1489970037	1132504918
3	1488018806	1131351461
4	1489418911	1132838158
5	1488293820	1132954597
6	1488837284	1132839306
7	1488407632	1131852031
8	1488608788	1131790272
9	1488391721	1132860033
10	1488651577	1132837142

On the Machine 2 (1 master and 10 workers):

Iteration	Data transfer from Mappers to Reducers (bytes)	Data transfer from Reducer to HDFS (S3) (bytes)
1	1339937128	1130027261
2	1527594955	1132805314
3	1525444356	1131356480
4	1526881419	1131356480
5	1525704116	1132951723
6	1526347759	1132833936
7	1525907769	1132894161
8	1526073595	1132830088
9	1525989258	1132856862
10	1526063243	1132830531

The amount of data transferred from Mappers to Reducers and Reducer to HDFS(S3 in our case) is pretty same throughout the 10 iterations on both the machines, mainly because the number of pages that are transferred remain constant throughout the 10 iterations and the |V| (total number of pages) is maintained. We are not losing out on pages across iteration.

Performance Comparison:

Job run times:

On Machine 1 (1 master and 5 workers)

Pre-processing time	2250 (secs)
Ten Iterations of PageRank	1267 (secs)
Top-100 pages	58 (secs)

On Machine 2 (1 master and 10 workers)

Pre-processing time	896 (secs)
Ten Iterations of PageRank	799 (secs)
Top-100 pages	39 (secs)

The number of worker machines is increased from Machine 1(1 master and 5 workers) to Machine 2(1 master and 10 workers), hence the speedup is expected to increase. This is evidently seen in the observed records above. The pre-processing step showed a maximum speed up among the 3 jobs, the reason being increased number of worker nodes. This results in the increased number of Mappers and Reducers, hence the processing time reduces considerably.

Top100 Pages from local and EMR:

The top100 pages from local and EMR machines of two configurations can be found in the attachment. Please find the link to the same in README.

The results of the pages seem fairly reasonable, some of the pages that made it to the top 100 are about countries, books, science and war which is reasonable. Also some of the wiki-ad pages like Wikimedia_commons and Wikimedia_foundation made it to top 100 because links to these pages appeared on many pages.