PARALLEL DATA PROCESSING IN MAPREDUCE – CS 6240
PROF. MIREK RIEDEWALD
VIJET BADIGANNAVAR

Pseudocode for Preprocessing job

```
// Setup function that initializes a set that stores all the links associated with this map function
// required to keep the count of total number of links
setup () {
    Initialize a set s that stores all the links
}

// map function that formats the input in the required format. If the link is a dangling node
then it has an adjacency list with value NA
// Format: linkId|outlink-1~outlink-2~~pagerankValue
//Ex: 1024|1024~Holy_Roman_Empire_f314~Pope~~0.0
//Ex: 1124|NA~~0.0 (Dangling Node)

map (offset B, Line l) {
    // l is a record that contains linkName:<htmlpage>

    Split l by ":" to obtain <linkName> and <html text>

    // a linkName is invalid if it contains "~"
    if (linkName is not valid) {
        return;
    }

    parse the html text and obtain all the hrefs that are buried under the div with id= "body
    Content" into the list links

    Iterate over the links and emit the record in the format
    linkId|outlink-1~outlink-2~~0.0  or
    linkId|NA~~0.0 as L
    emit (null, L);
    add the LinkName to set s
}

// this function just updates the count of all the links parsed so far to keep count to total no of
// nodes
cleanup () {
    increment the counter("TOTAL_LINKS") by size of s
}

//driver function
main () {
    // run this job
    // get total documents count from the counter("TOTAL_LINKS")
}
```

Explanation:
This job contains only mapper as we are just filtering and reshaping the record.

Pseudocode for Initializing all the links with default PageRank values job

Mapper:

```
        // Setup function that computes the default Page rank of all the values
        setup () {
            Obtain the TOTAL_LINKS from the configuration and compute 1/ TOTAL_LINKS
            as DEFUALT_PAGE_RANK_VALUE;
        }

        // map that reads each input and update the page rank value of each record by
        //DEFUALT_PAGE_RANK_VALUE
        map (offset B, Line l) {
            replace the page rank value (0.0) from l by DEFUALT_PAGE_RANK_VALUE
            emit (null, l)
        }
```

Explanation:
This job contains only mapper as we are just reshaping the record.

Pseudocode that computes the Page rank values of all the links iteratively

Mapper:

```
        // Setup function that computes the default Page rank share that should be equally distributed
        // across all the links
        setup () {
            Obtain the TOTAL_DOCS & TOTAL_DANGLING_PR_VALUE from the config
            and compute the PAGE_RANK_SHARE
        }

        // Map function that sends page rank share to each of the outlinks with outlinks as “”. It also
        // emits the outlinks with prValue INFINTIY to send the outlinks to the reducer.
        map (offset B, Line l) {
            // PageRankLinkRecord is {outlinks, prScore}

            split l and obtain the LinkName, ListOfOutlinks, Pagerank
            pageRankValue = (Pagerank+ PAGE_RANK_SHARE)/NumberOfOutlinks;

            // send PageLinkRecord with outlinks as “” and PRvalue as PAGE_RANK_SHARE
            if (linkName is danglingNode) {
                emit (nodeId, {“”, PAGE_RANK_SHARE});
                return;
            }

            // send PageLinkRecord with outlinks as “” and pageRankValue for each of outlinks
            for each outlink in ListOfOutlinks
                emit (nodeId, {“”, pageRankValue})

            // send the outlinks for this node
            emit (nodeId, {ListOfOutlinks, INFINITY})
        }
```

Reducer:

```
// Setup function that computes the default Page rank share for all of all the values
setup () {
        Obtain the TOTAL_RECORDS from the config
}

reduce (Text nodeId, [pr1,pr2,pr3…]) {
        // [pr1,pr2…pn is inputlist of PageRankRecord]
        // PageRankRecord p is {outlinks,prValue}

        outlinkslist = null;
        newprScore = 0.0;
        // accumulate the pagerank scores for nodeId
        foreach (p in inputlist){
                if(outlinkslist ==null && p.prScore is INFINITY){
                        outlinkslist = p.outlinks;
                        continue
                }
                newprScore+=p.prScore;
        }
        // compute the new Pagerank score
        newPrScore = 1.0/ TOTAL_RECORDS *0.15 + (0.85) * newPrScore;

        // accumulate the pagerank of the dangling nodes
        if (outlinksList == null){
                update the counter("DANGLING_NODES") by newPrScore;
                outLinksList = "NA"
        }

        emit(null,"node|outlinksList~~newPrScore");
}
```

Explanation:

The Mapper is going to receive the TOTAL_LINKS and TOTAL_PR_DANGLING_NODES through the configuration. It computes the share that needs to be divided among all the links and adds that value to the pagerank of each link. This new pagerank value is divided equally among all the outlinks and it will sent to the reducer with each outlink node as key and equal share as value. The outlinks associated with the node n is also sent to the reducer with INFINITY as the PR value.

The reduce function is going to accumulate the pagerank(PR) value for each of the nodes and is going to is going to compute the new page rank (according to formula). If the node is a dangling node then that page rank value is updated to the counter. The new record with the outlinks obtained is written back to HDFS.

This is given as an input to the next mapper job and the same process continues.

Top-50 records

Mapper:

```
// Map function that sends page rank share to each of the outlinks with outlinks as "". It also
// emits the outlinks with prValue INFINTIY to send the outlinks to the reducer.
map (offset B, Line l) {
    // MapperKey is {key, value}
    pr = obtain pagerank value from l
    emit ({"dummy", pr}, l);
}

// Key comparator that sorts the links by descending order of their pageranks
KeyComparator (MapperKey k1, MapperKey k2) {
    // MapperKey is {key, value} (key="dummy", value = pagerank)
    Sort by MapperKeys by value field in descending order
}

// Grouping comparator that groups the keys by key name so that all the records belonging to
// the same key are retrieved in a single reduce call
GroupComparator (MapperKey k1, MapperKey k2) {
    // MapperKey is {key, value} (key=" dummy", value = pagerank)
    Sort by k1.key;
}

// Reduce function that accumulates the pagerank values emitted with the key "dummy" and is
// grouped using the same by descending order of pagerank values. It just iterates over list and
// outputs the top 100 records
Reduce (MapperKey k1, [l1, l2, l3…]) {
    // MapperKey is {key, value} (key= "dummy", value = pagerank)
    // [l1, l2…ln is inputlist of Text] representing Link
    emit top 100 records from the list l
}
```

Amount of Data transferred between during each iteration:

Iterations	Mapper to reducer (in bytes)	Reducer to HDFS (in bytes)
1	1806157778	1402543305
2	2147917470	1400369350
3	2143315246	1407543836
4	2148459961	1404657132
5	2149464593	1406064045
6	2143494784	1408235571
7	2147393732	1387947450
8	2148943202	1408266743
9	2148125255	1408212232
10	2148150253	1407511232

The data changes during the first iteration as the number of links will be increased but after 2nd iteration it will remains constant. There will be minor changes due to the fact of page rank values changing during the iteration. After the first iteration all the links that are original links as well as the outlinks (including dangling nodes) will be written to the file and it will be passed subsequently over the iterations. Hence the results.

Performance Comparison:

6 m4. large machines (1 master and 5 workers)

	Start Time	End time	Total Time
Preprocessing Time	17:30:01,419	17:45:28,346	~ 15 minutes
Time to run 10 iterations	17:45:29,214	18:09:20,134	~ 24 minutes
Time to find top 100 records	18:09:20,650	18:10:00,846	~ 40 seconds

11 m4. large machines (1 master and 10 workers)

	Start Time	End time	Total Time
Preprocessing Time	18:45:54,264	18:54:21,367	~ 9 minutes
Time to run 10 iterations	18:54:22,280	19:08:57,201	~ 14 minutes
Time to find top 100 records	19:08:57,883	19:09:36,061	~ 40 seconds

As the number of nodes increases, the speedup is expected to increase and it can be found from the results. All the computation phases of all the jobs shows a good speedup. The time taken by the corresponding job (6 machines vs 11 machines) is less in 11 machines when compared to that of 6 machines during all the computation phases.

Outputs of local and EMR:
The files are attached with the submission!

Some of the records seems to be reasonable. There are some records like "Wikimedia commons" etc that are present in all the pages thus these ads come to the top. But most of them are reasonably relevant information.