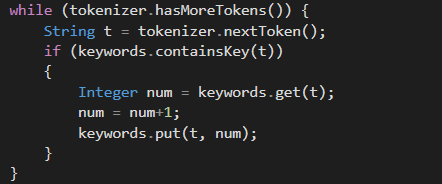
**Program 3 Report**

**Documentation of your inserted-indexing program**

1. Map Phase

I use a HashTable to track the counts of each keyword. Detailed steps are as the following:

* 1. Put keywords in HashTable. All counts are set to be 0.
  2. Read files and do counting of keywords. Update the counts of keywords in HashTable.



1. After reading all files splits, use a string to hold filename and counts. then pass a Text object to output collect.

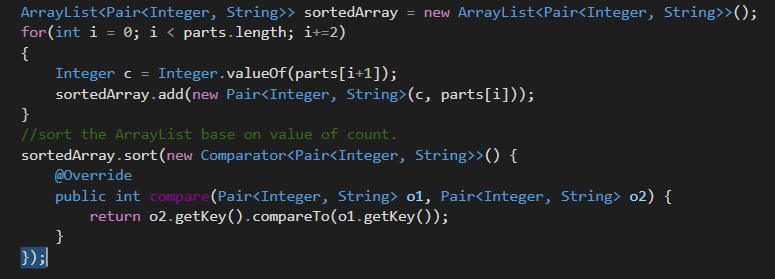
String sValue = filename + "\_" + num.toString();

output.collect(new Text(key), new Text(sValue));

1. Reduce Phase

I prepare a HashMap to main all file counts similar to what I did in Map().

**A new feature:** After counts are updated, then it’s time to sort all these file counts. I use an ArrayList with Pair objects to do the sorting task.

After sorting is done, I then output the results as Text objects. All the file counts are presented in a decreasing order.

**Execution Output**

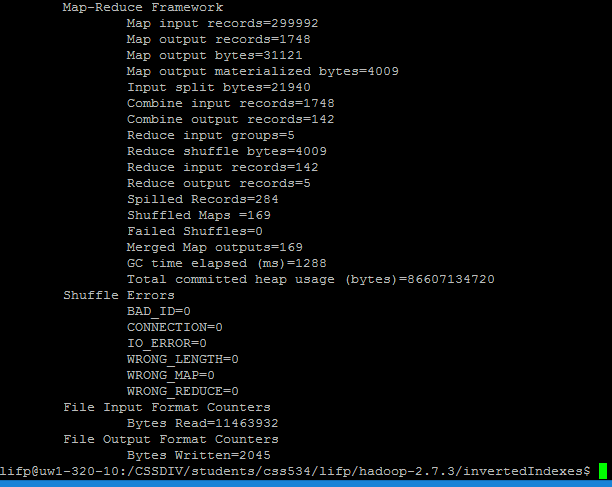
Hadoop-2.7.3 with 4 CPU-core machines :

1 computing node: 1288

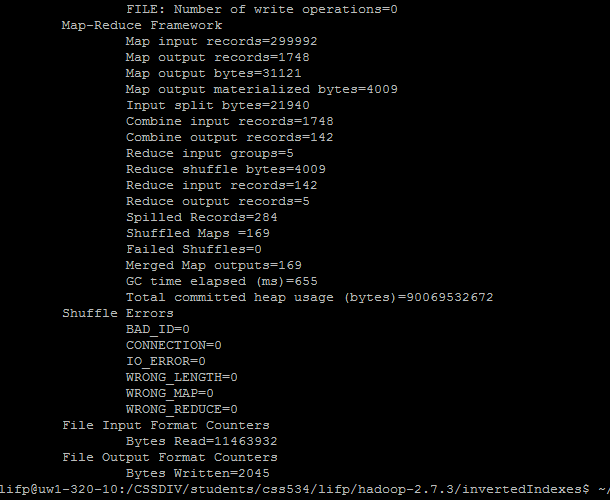
4 computing nodes: 655 ms

Performance improvement is above the criteria 1.1

1 computing node execution output: (the log is too long to copy, so I only post the final parts of the execution output. )



4 computing nodes execution output:



Content of part-00000 file:

HDLC rfc1662.txt 4 rfc2863.txt 3 rfc891.txt 2 rfc907.txt 2 rfc2865.txt 1 rfc1122.txt 1

LAN rfc2067.txt 17 rfc950.txt 12 rfc2427.txt 11 rfc1122.txt 10 rfc1694.txt 7 rfc1748.txt 6 rfc1213.txt 5 rfc1658.txt 5 rfc1659.txt 5 rfc1660.txt 5 rfc1212.txt 5 rfc2895.txt 4 rfc1724.txt 3 rfc1629.txt 3 rfc1559.txt 3 rfc1155.txt 2 rfc5321.txt 2 rfc2115.txt 2 rfc1661.txt 1 rfc1044.txt 1 rfc2348.txt 1 rfc3461.txt 1 rfc4862.txt 1 rfc1123.txt 1 rfc2613.txt 1

PP rfc2067.txt 1

TCP rfc793.txt 278 rfc1122.txt 221 rfc4271.txt 126 rfc1001.txt 123 rfc5681.txt 83 rfc5036.txt 58 rfc5734.txt 42 rfc1123.txt 41 rfc1002.txt 38 rfc1213.txt 33 rfc1191.txt 25 rfc1006.txt 24 rfc1981.txt 18 rfc2460.txt 12 rfc791.txt 12 rfc1035.txt 12 rfc2132.txt 11 rfc5321.txt 10 rfc2865.txt 10 rfc854.txt 10 rfc4861.txt 9 rfc2920.txt 9 rfc2741.txt 8 rfc1939.txt 8 rfc4862.txt 7 rfc3551.txt 6 rfc1034.txt 5 rfc864.txt 5 rfc1772.txt 5 rfc959.txt 5 rfc866.txt 3 rfc863.txt 3 rfc862.txt 3 rfc867.txt 3 rfc1055.txt 3 rfc5531.txt 3 rfc2895.txt 3 rfc1042.txt 3 rfc865.txt 3 rfc792.txt 3 rfc1356.txt 3 rfc3912.txt 3 rfc3801.txt 3 rfc1132.txt 2 rfc1188.txt 2 rfc895.txt 2 rfc894.txt 2 rfc1288.txt 2 rfc1044.txt 2 rfc5065.txt 2 rfc1201.txt 2 rfc1184.txt 2 rfc3986.txt 2 rfc2355.txt 2 rfc1390.txt 2 rfc2067.txt 1 rfc3550.txt 1 rfc2289.txt 1 rfc4941.txt 1 rfc868.txt 1 rfc5322.txt 1 rfc6152.txt 1 rfc1155.txt 1 rfc1870.txt 1 rfc5730.txt 1 rfc1658.txt 1 rfc919.txt 1 rfc907.txt 1 rfc922.txt 1 rfc4456.txt 1

UDP rfc1122.txt 65 rfc1002.txt 50 rfc1001.txt 33 rfc1123.txt 25 rfc2865.txt 24 rfc1542.txt 21 rfc1213.txt 19 rfc3550.txt 15 rfc1035.txt 13 rfc3417.txt 12 rfc951.txt 11 rfc5036.txt 10 rfc2460.txt 8 rfc768.txt 6 rfc2131.txt 5 rfc4502.txt 5 rfc864.txt 4 rfc3551.txt 4 rfc863.txt 3 rfc1350.txt 3 rfc862.txt 3 rfc2895.txt 3 rfc792.txt 3 rfc1191.txt 3 rfc1981.txt 3 rfc867.txt 3 rfc866.txt 3 rfc865.txt 3 rfc5531.txt 2 rfc791.txt 2 rfc2453.txt 2 rfc3411.txt 2 rfc4862.txt 2 rfc1034.txt 2 rfc1055.txt 1 rfc1629.txt 1 rfc868.txt 1 rfc2348.txt 1 rfc2132.txt 1 rfc950.txt 1

**Discussions**

Programmability: What if you wrote the same program using MPI? Compare MapReduce and MPI?

Compare to using MapReduce, it’s much more complicate to write the same program using MPI.

1. MPI requires developers to assign the tasks for each computing nodes, either give them same or different amount tasks.
2. Data need to be sent/received between the nodes in MPI for parallel computing. Moreover, some operations are required to be done in critical section, otherwise deadlocks may occur.
3. MPI is not fault-tolerant, it’s developers’ responsibility to take care of that. While MapReduce has automated fault-tolerance.

If using MapReduce, there is no need to worry about above scenarios. The programmability of MapReduce is better than MPI.

However, for operations that need communications between nodes and code flexibility, MPI is a better choice.

Usability: Summarize the MapReduce pros and cons

|  |  |
| --- | --- |
| Pros | Cons |
| * Easy to code * Scalability * Great performance can be achieved for data has no dependencies between tasks * Good data locality * Fault tolerance: allows for nodes to fail without interrupting the processing. | * For jobs on small data sets, there is an execution overhead that means even simple MapReduce jobs may take a minimum of 10 seconds. * Not for real time processing. Can’t provide fast responses within seconds. * Not for intermediate processes need to talk to each other |

Possible applications: Consider what applications can be written better with MapReduce

MapReduce is best used when data enters the range of hundreds of Gigabytes or more. MapReduce is designed to be embarrassingly parallel. To get the most performance means that the data is huge in size and there is no communication between worker nodes, so no data is really shared between them. For example, processing geographical data to locating roads connected to a given intersection and so on.

Any performance consideration

To get good performance developers should avoid using MapReduce for those operations:

1. Real-time processing.
2. Intermediate processes need to talk to each other.
3. Processing requires a lot of data to be shuffled over the network.
4. The operations need to handle streaming data.
5. Iterations. When you need to process data again and again.
6. When map phase generates too many keys, then sorting takes forever.