CS 425 MP 4 Report

Group 55: Yuchen Liang (yliang35) and Xilun Jin (xjin12)

Design

In this MP, we implemented Crane, a distributed data stream processing system, using Python that has functionalities similar to those of Spark Streaming.

The system uses tuples, spouts, bolts, sinks that are similar to Spark. The data are transmitted between processing nodes as tuple streams. The spout starts the tuple streams and sequentially sends data tuples to bolts. The bolts receive the tuples, process them, and send to the next bolts.

There are four types of bolts, all accepting a user-defined function. The filter bolts only forward tuples satisfying the function condition. The transform bolts transform each tuple using the function. The join bolts join the incoming tuple with a static database (including a file) with conditions specified by the function. The reduce bolts aggregate the final tuples into a cache for aggregation by a function. In our infrastructure, the final bolt is denoted as sink. The only difference with an ordinary bolt is that it saves the result to local instead of sending it further. At the end of job, the sink will send the result key (i.e. file name in SDFS) back to the master.

Jobs are submitted in RPC from client to master or from master to itself. The function of each bolt is executed locally at each machine.

Failure is considered in two ways. The failure of master will be detected by a backup master. Before task begins, the master send topology to the backup, so the latter can pick up the remaining job quickly. In such occasions, the backup master will send a message to the sink, telling it to forward the final result key to itself. The failure of workers is detected by the master. When failure occurs, the master will restart the entire job to ensure the validity of the final result.

All previous MPs are useful in our design. MP1 helps us retrieve and compare the intermediate result at different bolts. MP2 helps Crane detect the failure of master and workers. MP3 helps us store the final result in a reliable and accessible way.

Applications

In the testing phase, we design the following 3 tasks.

1. Word Count

This application counts the total number of occurrences of each word in the file. The input is any (meaningful) text document. The user should define the functions as followed. The spout sends a single line to the next "split" bolt, which splits the received line into individual words. Then, each single word is sent to "pair" bolts for grouping (i.e. $w \rightarrow < w, 1 >$). Finally, the tuple is aggregated w.r.t. the second column at the reduce bolt at the end.

2. URL Count

This application counts the total number of occurrences of a URL. The input could be any text document containing meaningful URLs. Compared to the previous application, only words representing specific URLs are counted. The topology is designed similar to the previous task, plus an additional filter bolt between "split" and "pair" bolts.

3. Distance-two Nodes

This application finds all distance-2 nodes from the start node. The distance-2 nodes for a node x in a directed graph are those that x can reach with exactly 2 hops. The input to this application is a text file where each line indicates a directed edge. The spout takes a single line and sends to transform bolts to decode. Then, the tuples are joined with the original database in the join bolt. The final result contains a distance-2 node list for each starting node.

Experiment and Plots

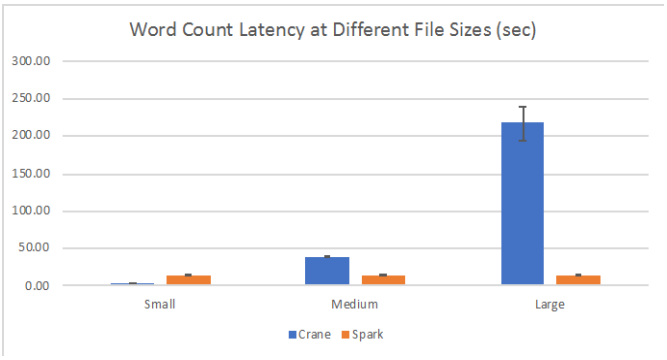
In both experiments, we measured the latency to complete each single task. The latencies were measured over different input file sizes and different number of workers.

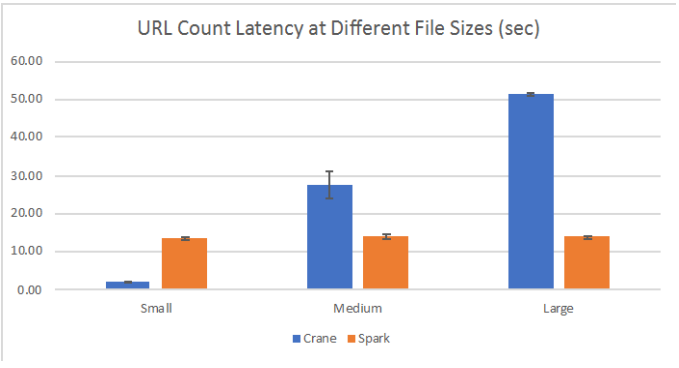
1. File size (using 5 workers)

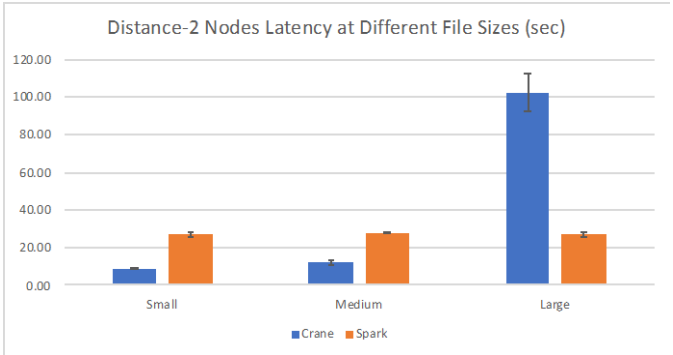
Job 1	*Crane*			*Spark*		
File sizes	Small (sec)	Medium (sec)	Large (sec)	Small (sec)	Medium (sec)	Large (sec)
1	3.34	40.92	198.23	14.67	15.31	14.19
2	2.65	37.68	211.13	13.6	14.9	13.09
3	3.55	38.81	242.75	14.37	14.75	14.86
Avg	3.18	39.14	217.37	14.21	14.99	14.05
Std	0.47	1.64	22.91	0.55	0.29	0.89

Job 2	*Crane*			*Spark*		
File sizes	Small (sec)	Medium (sec)	Large (sec)	Small (sec)	Medium (sec)	Large (sec)
1	2.42	29.09	50.8	14.01	13.51	13.53
2	1.98	23.43	51.27	13.32	14.74	14.23
3	1.96	30.18	51.54	13.26	14.17	13.74
Avg	2.12	27.57	51.20	13.53	14.14	13.83
Std	0.26	3.62	0.37	0.42	0.62	0.36

Job 3	*Crane*			*Spark*		
File sizes	Small (sec)	Medium (sec)	Large (sec)	Small (sec)	Medium (sec)	Large (sec)
1	9.24	11.51	112.65	26.169	28.079	26.754
2	8.28	13.38	100.32	27.989	28.101	26.428
3	9.30	12.23	93.12	27.238	28.299	28.566
Avg	8.94	12.37	102.03	27.13	28.16	27.25
Std	0.57	0.94	9.88	0.91	0.12	1.15





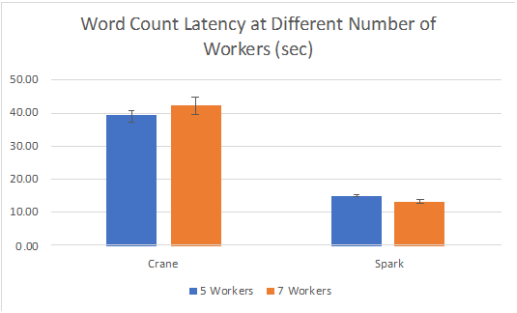


From the plots, we find that in all three jobs, on average, Crane is faster than Spark at small file sizes, but much slower at large file sizes. This is what is expected, because Spark initiates complex parallelism and topology. Hence, if the input size is small, Spark's process will produce larger overhead time than Crane, but Spark will gain greatly if the input size is large.

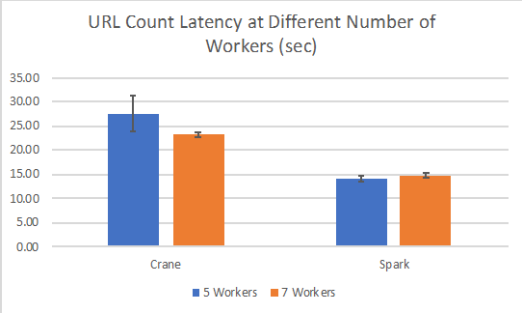
2. Number of workers (using medium file size)

**Note: 7 is the largest worker number after Master, Backup master, and Client are excluded.*

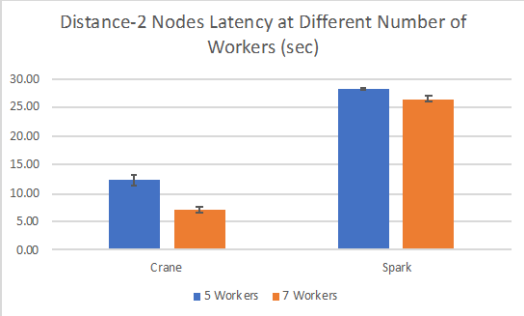
Job 1	*Crane*		*Spark*	
Number of Workers	5 (sec)	7* (sec)	5 (sec)	7 (sec)
1	40.92	45.17	15.31	13.3
2	37.68	40.13	14.9	13.02
3	38.81	41.26	14.75	14.28
Avg	39.14	42.19	14.99	13.53
Std	1.64	2.64	0.29	0.66



Job 2	*Crane*		*Spark*	
Number of Workers	5 (sec)	7 (sec)	5 (sec)	7 (sec)
1	29.09	23.54	13.51	15.14
2	23.43	23.5	14.74	14.27
3	30.18	22.66	14.17	14.92
Avg	27.57	23.23	14.14	14.78
Std	3.62	0.50	0.62	0.45



Job 3	*Crane*		*Spark*	
Number of Workers	5 (sec)	7 (sec)	5 (sec)	7 (sec)
1	11.51	7.5	28.079	26.345
2	13.38	6.43	28.101	26.247
3	12.23	7.23	28.299	27.068
Avg	12.37	7.05	28.16	26.55
Std	0.94	0.56	0.12	0.45



From the plots, we can see that increasing number of workers from 5 to 7 does not have much effect on latency. This is because the increase is too tiny to discover the difference. For Crane, the latency sometimes increases because more workers could result in more overhead network transmission time. Note that for job 3, Spark uses tremendous overhead time for join operation with a static database.