MP4:Crane

Group 38: Rahul (rahulsk2) & Venkat (vn7)

1) Design:

For MP4, we reuse the master node from MP3 as the master for MP4, to make use of the existing infrastructure. Bolts are randomly mapped onto worker machines, and the individual nodes take responsibility for setting up their network connections for the purpose of streaming data. In our scheme, a bolt connects to all of its input bolts to establish the topology. To achieve this, each bolt informs its output bolts of the port number to collect to it on.

Our design is a pure streaming solution, with no micro batching. To terminate the computation, we use a sentinel string "TOFFEE" throughout our code to indicate to a data processor when no further data can be expected from that particular input source.

We describe jobs to be run with 2 files - a topology file, which describes the topology of the DAG using JSON, and a harness Python 3 script called “Methods.py” that examines its arguments to run the appropriate function. We chose this approach because it allowed us to support arbitrary bolts(including stateful ones) without having to reimplement an entire programming framework.

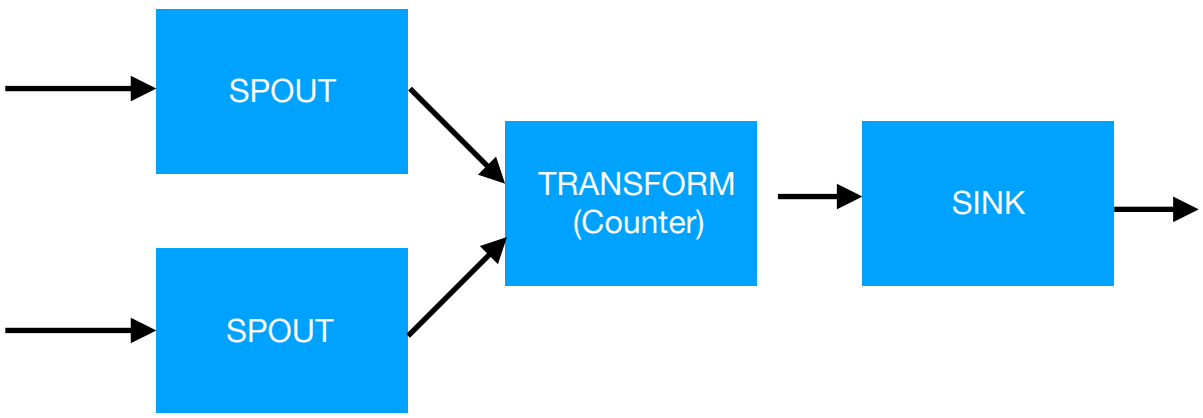
When a failure occurs, the master restarts the job after a short delay(to allow the information about the failure to propagate through the entire network). Individual bolts are designed to automatically terminate when one of the bolts they contact crash, allowing the entire system to reach a quiescent state before the master restarts the job. The delay by the master allows multiple failures close to each other to not cause unnecessary job restarts. This would map onto the real life case of something like a rack failure.

2) Chosen Applications

We have chosen simple applications that could successfully demonstrate the flexibility in node topologies that our design supports and also all the filter join and transform type operations. The three applications are as under:

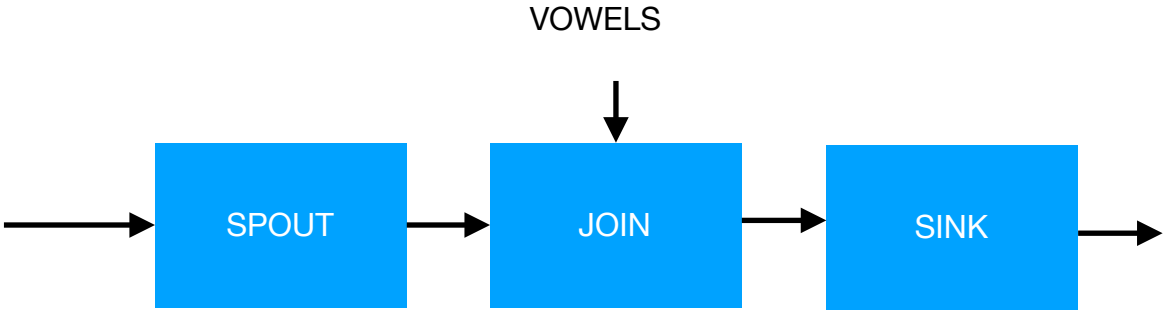
a) WordCounter:

This is a simple word counter, but we demonstrate the flexibility of our topology definitions by using two different input files - One with Fruits and another with Colours
The transform aggregates the data into a single counter and the sink finally outputs this.



b) AlphabetJoin:

This is the demonstration of join functionality, The spout reads a file containing alphabets A-Z, The spout joins it with a database file of Vowels AEIOU. Finally the Sink outputs these tuples.



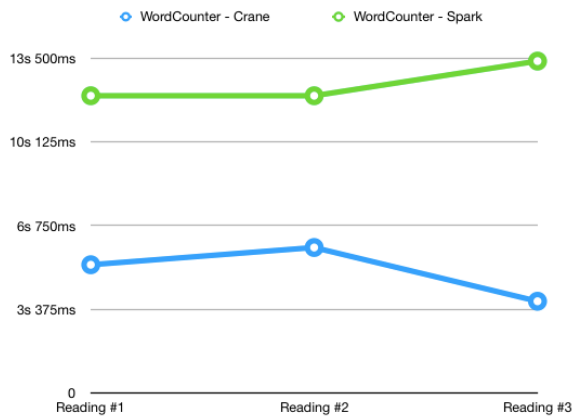
c) AlphabetConsonants:

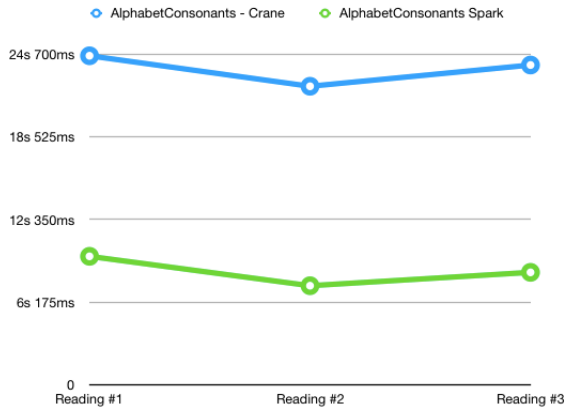
This is the demonstration of Filter functionality, The spout reads a file containing alphabets A-Z, The filter merely drops the input strings that are vowels. The Sink finally outputs just the consonant strings.

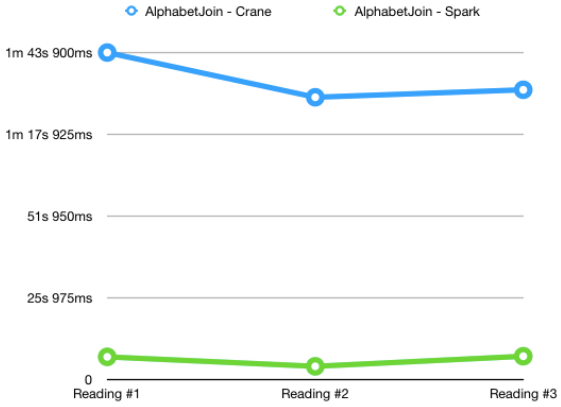
3) Performance Measurements:

Run Crane	Reading #1	Reading #2	Reading #3	STDDEV
WordCounter - Crane	5s 171ms	5s 867ms	3s 695ms	1s 109ms
AlphabetJoin - Crane	1m 43s 810ms	1m 29s 633ms	1m 31s 999ms	7s 595ms
AlphabetConsonants - Crane	24s 613ms	22s 330ms	23s 921ms	1s 171ms
Run Spark				
WordCounter - Spark	12s	12s	13s 400ms	808ms
AlphabetJoin - Spark	7s 300ms	4s 300ms	7s 500ms	1s 793ms
AlphabetConsonants Spark	9s 600ms	7s 400ms	8s 400ms	1s 102ms

4) Interpreting the Results:







We could note that for the Word Counter task, Crane performs better than Spark, but for the other ones, Spark wins.

When dealing with smaller sized Inputs, the latency of file Operations appears to outweigh the latency of the network, and that must be why Crane wins.

Crane does not have to deal with the scheduling delays that we get in spark.

With the increase in size of data to be processed though, Crane seems to perform worse than Spark. This might be due to the fact that we do not do any batching in Crane and process the input tuple by tuple.