CS425 MP4 Group 44 Crane Streaming System

Shichu Zhu(szhu28), Fan Shi(fanshi2)

Crane System Design:

Our Crane system is a distributed streaming processing with generic client developing interface. The Crane system use the idea of Apache Spark project and build on our previous work. The system is fault-tolerant and can reach a higher speed compared to Spark system.

Our crane build on our previous work of failure detector and SDFS. Crane has three component in its system: Master, Standby master and workers. The master is used to start and supervise each job and standby master is the replica of master which store the whole state of current running system. Workers are the nodes that are processing streaming data. Based on the existing framework, each node in the system will store a whole membership list so that master will choose appropriate number of nodes as workers for specific job. Workers can be separated as Spout, Bolt and Sink based on their functionality. Workers are chosen based on the task-topology and the storage of input files which can improve the performance of the crane system.

Our Crane is faulty-tolerant to multiple failures. Based on the existing design, each node can sense the failure of member in the system. Once the member of a task is failed, the crane system will stop the current job and re-config the task topology based on original design. The system will recover from the failure within 4 seconds and provides exactly the same result as it has never encounter system failure.

The Crane system provides a generic programming framework for user to design their specific streaming processing job. They need to provide client source code as will as a topology configuration file. All the client code will be packed and push to SDFS. The task master will parse the topology configuration and construct the tsak topology based on the current VM membership list. Then each worker will pull the zip folder and start its own work. The ending of job will be sent to master once the Sink worker finishing written the last byte. Job master (or standby Master if master failed) will acknowledge the client that task is done.

We utilize the plugin method provided by goland which allow us to implement the generic programming interface. We will run the system and the node can “plugin” user’s client code to do specific work.

Client Programming Interface:

At first, user has to provide a topology configuration file of the task. User need to specify each worker’s role, mark them as “spout”, “bolt” and “sink”. For each worker, user also need to denote the predecessor of it simply by adding the predecessor worker number into current worker’s list. We only support one spout and sink per task.

The used need to define the client code for each task worker. We will use byte stream transferring data across VMs and then decode the byte streaming before provide to client layer. User can directly work on the user-friendly client data and doesn’t have to parse them. Spout and Sink’s functionality is different from normal bolt, User need to specify the method to get the data and write result to file.

Our current design also support multi-task run on the cluster. A node can take several worker job from different task. All the task are independent.

System measurement Crane and Spark

We will run multiple job in our system as well as py-spark to test the performance of our Crane system. For each task we will run for multiple times. We will include both test result as well as figure for each job.

In the system measurement we will firstly apply two real data set, then we write a socket simulator to simulate the dynamic input string of communication network.

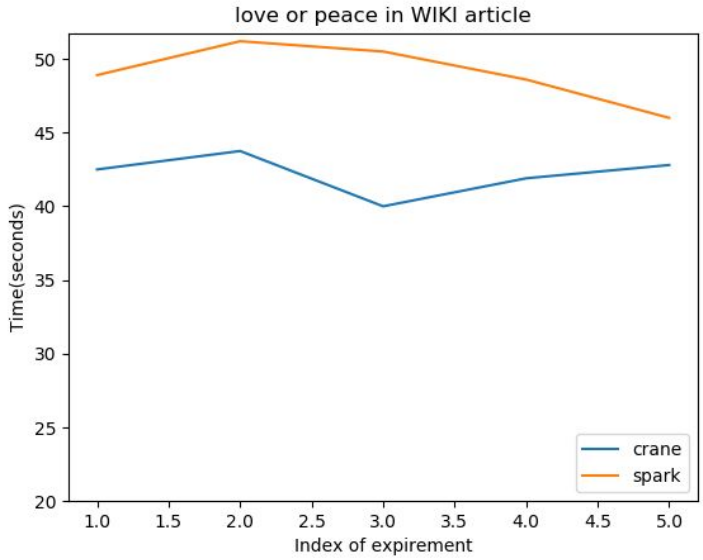
Measurement 1. Separate "Love and Peace" message from plain wiki text

In this test, we use the wiki dataset from "http://snap.stanford.edu/data/wikispeedia.html" which contains all the wiki articles. We will first read in the data set, separate the sentence with more that 10 character (valuable sentence), then we will filter all the sentence include "love" or "peace" and finally write all the sentence to a txt file.

Finally we see that this wiki dataset contains 4705 sentences with "love" or "peace" and output file size is 330 KB.

Operation time of Crane and py-Spark is shown below.

Crane	42.5s	43.75s	40.0s	41.9s	42.8s
Spark	48.9s	51.2s	50.5s	48.6s	46s

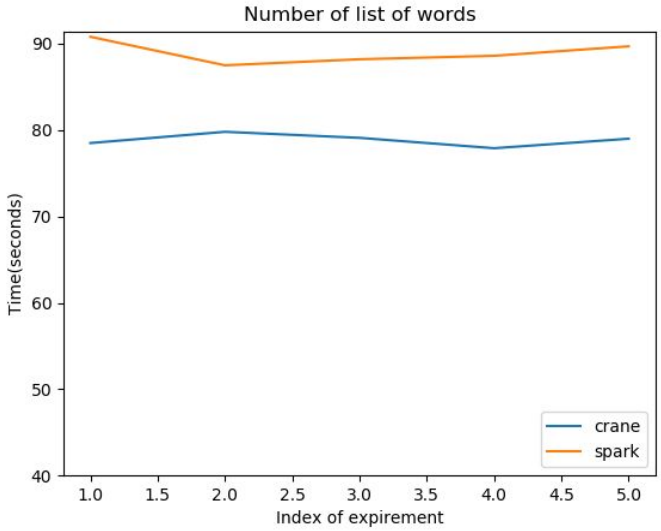


Mean: 42.9 (Crane)
Standard Deviation: 1.394 (Crane)
In this test, we check the basic functionality of our system. It read in file stream and filter the result with specific string we want.
The system is a little faster than py-Spark. As we expected, go language provide better performance and our system have shorter system load time.

Measurement 2. Count specific word in WIKI plain article

In this measurement we create a specific subset of words that we would like to count numbers. We reuse the dataset of wiki Plain text. This time the operation include two main operation. First we transform all the words into lower case, then we will apply map reduce count the total numbers of each word.

Crane	78.5s	79.8s	79.1s	77.9s	79s
Spark	90.8s	87.5s	88.2s	88.6s	89.7s

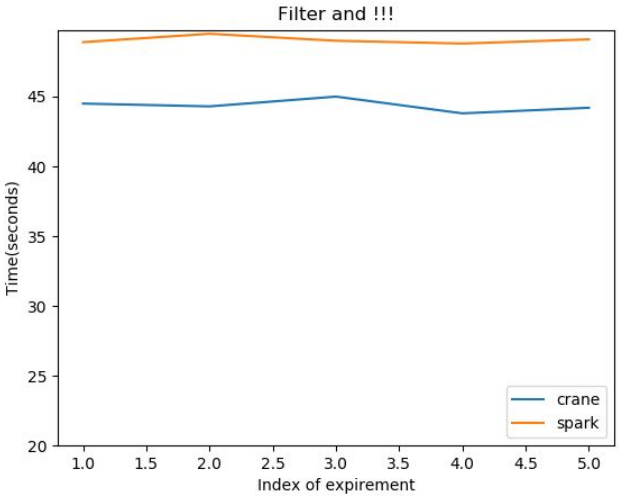


Mean: 78.86(Crane)
Standard Deviation:0.7 (Crane)

In this test, we can see that our system can performance much better than py-Spark. But the two task may have different execution pattern since we are more familiar with application on our own system.

Measurement 3. Filter, uppercase and !!!

In this measurement, we want to test the operation to modify the original content of input streaming. Both systems will read in file stream and filter the sentence with length larger than 15. Then change all the character to uppercase. Then add "!!!" to each sentence.



In this test we can see that the processing time is very close and with stable output. The reason is that the work is not hard, and only parameter for processing time is simply the length of dataset.

The difference of execution time comes from the loading time of two systems. Crane system require less time to load job.