1. Design

Crane is a stream processing system similar to Apache Storm. To make it simple, Crane consists several classes corresponding the terminologies of <Master>, <Slave>, <Spout>, <Bolt> and <Tuple>. While using Crane, each application needs to be defined as a specific topology, which determines the flow of data tuples, the number/functionality of its own bolts and where the spout is, (web source, file source or database), based on the application itself. A topology set spout which starts the data streaming and a chain of bolts. Each topology can be seen as Directed acyclic graph. Inside which, each bolt processes incoming tuples and sends the outgoing tuples to the next bolt, or sends the acknowledgement to the master node if the current tuple has been fully processed.

1.1 Main Classes:

CraneMaster():

1. Class for master node in crane cluster. Two threads, crane_monitor and crane_aggregator, running to collect acknowledgement, as well as partial results, from slave nodes and check if all tuples have been fully processed, after which the master node will fetch the final results for the current topology.
2. Functions: *start_top(), terminate(), crane_monitor(), crane_aggregator(), _unicast(), udp_unicast(), emit() and print_result()*

CraneSlave():

1. Class for slave/worker nodes. Once master node open the spout(s), slave nodes will expect to receive tuple stream(s) from either the master node or other slave nodes, which should holds a higher position than itself in the workflow of the current topology.
2. Functions: *run(), terminate(), slave_receiver() and exec_msg()*

Collector():

1. Collector is a helper class inside CraneSlave(). The idea is to separate the message handling and main working process of CraneSlave(). It is also a common practice in Apache Store. Two kinds of message are sent out by collector, messages with type ‘emit’ would be transmitted to other slave nodes. Meanwhile, messages with type ‘ack’, usually include partial results from a number of fully-processed tuples, would be transmitted back to master node.
2. Functions: *_unicast(), udp_unicast(), set_master(), emit() and ack()*

Topology():

1. Topology is a core class for job construction. Our three demo topologies are built via this class. Upon initialization, one need to set the spout and bolt(s) for the topology one want to build.
2. Functions: *set_spout() and set_bolt()*

Blot():

1. Bolt class, which is the helper class/interface of the ‘Topology' class to set bolt(s). All topologies need to implement its own bolt() class, like SplitBolt() and CountBolt() for word count, to accommodate its own functionality and purposes.
2. One abstract function available: *execute()*

Spout():

1. Spout class. Helper class of the ‘Topology' class to set spout based on file source, in other words, static data source/database.
2. Two functions available: *next_tup() and close()*.

Tuple():

1. Wrapper class and conceptual body of terminology <Tuple> in Apache Storm. Each tuple, upon construction, will be assigned to an **uuid** and its’ own content body, which could be of any type and provides flexibility for incoming data type.

TupleBatch():

1. The idea of this 'TupleBatch' is a bit like Apache Spark's Resilient Distributed Dataset to ensure **fault tolerance**. TupleBatch's size could be altered at 'Util.py' and customizable for different workloads and applications.

1.2 Fault Tolerance

As TCP is chosen for the internal communication between Crane nodes. Only failures on certain Crane nodes and processes would bring fault to our systems. Hence, we borrowed the idea of <Fully processed Tuple> from Apache Storm to ensure the fault tolerance.

In a nutshell, each tuple would be assigned to a **universal unique id** (uuid) and <Master> node would remember this id, as well as the timestamp created when this tuple is sent out from it, associated with a copy of this tuple itself. If and only if this tuple has been fully processed, which depends on the ongoing topology, the uuid of this tuple can be sent back to <Master> and the uuid on <Master> of that tuple would be 'xor'ed/set to 0. The crane_aggregator would constantly check if all sent-out tuples has been fully process by checking if the uuid of those tuple has been reset. If not, it would compare the timestamp on that tuple to the current time. If the difference is larger to a certain threshold, which is 10 seconds in Crane and 30 seconds in Apache Storm, it would assume some node failed and re-run the tuple based on the local copy.

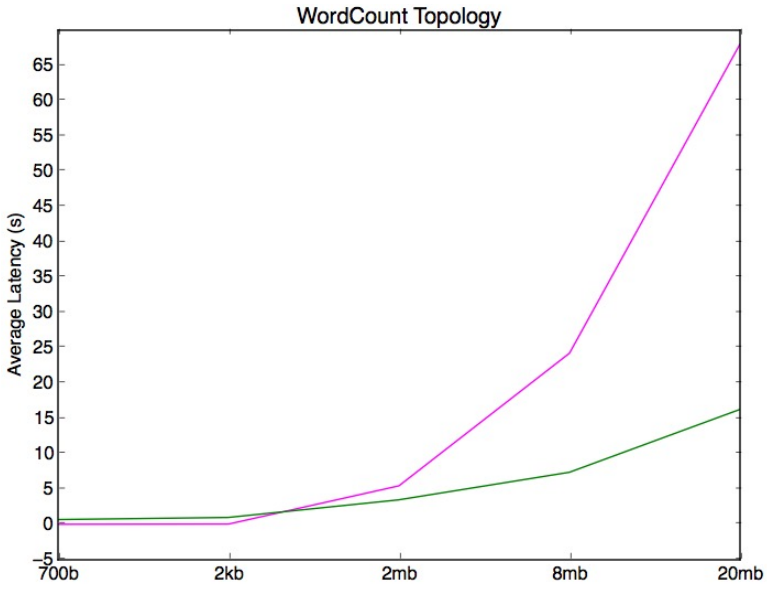
2. Discussion

We implement three topologies in Crane and Spark: wordcount, pagerank, and twitter user filter. We will compare and discussion the performance of Crane and Spark with respect to the average latency. For all figures below, the magenta line is Crane and the green line is Spark. Crane is faster than Spark on smaller datasets, and Spark performs better on larger datasets.

2.1 Word Count

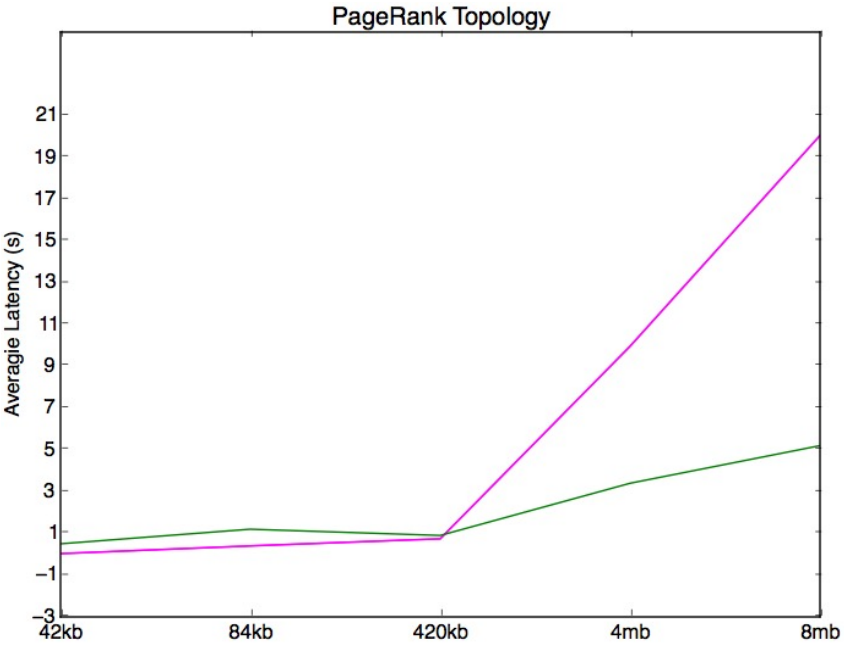
We examine Crane and Spark on five datasets. On an 2Mb dataset, Spark start running faster than Crane. It might seem like Crane running time grows exponentially as dataset size grow up. However, please note that dataset size growth is not linear, the Crane running time is actually linear as dataset size grows, as we expect with the Streaming design.

When a worker fails in Crane, the running tuples in the failed worker get re-runed, which increase the running time by around 30%.



Dataset Size	Normal	One Worker Fail
8MB	24.26s	30.23s
20MB	68.02s	92.94s

2.2 Page Rank

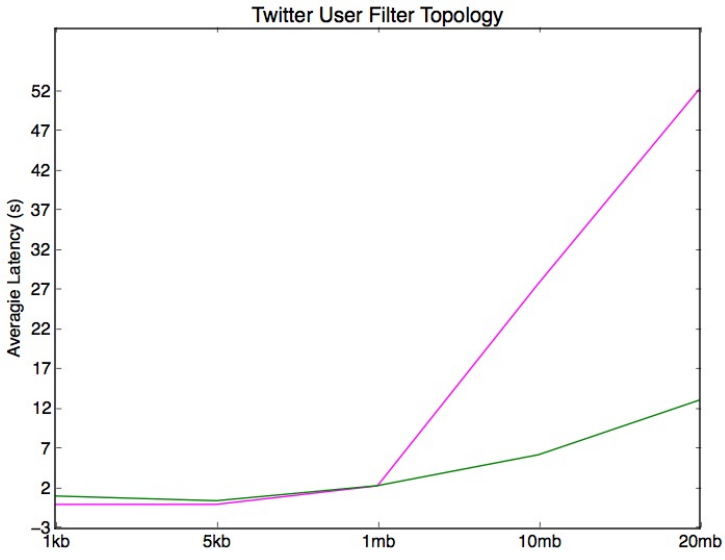


Size	Normal	One Worker Fail
4MB	9.96s	27.34s
8MB	20.05s	32.91s

PageRank is a naive but useful ranking algorithm, which takes url tuples as inputs. Each URL has a list of neighbors which can be empty. Every URL add 1/ size_of_neighbors on its neighbors. The PageRank topology is consists of the Parse Neighbor Bolt and the Contribution Computation Bolt. The running time comparison on five datasets between Crane and Spark is shown here. As shown, Crane is faster than Spark on smaller datasets, while Spark is better with larger ones.
Crane running time roughly follows a linear growth which has a slightly growing slope. Crane suffers from message exchange junk with larger datasets, which is the reason why the slope grows as dataset size grows.

We also examine Crane's robustness on fault tolerance. The time increment with one node failure does not have a distinct pattern. The reason for this is that time increment highly depends on the stage of the failed node, which is hard to control in experiments.

2.3 Twitter User Filter



Size	Normal	One Worker Fail
10MB	27.90s	28.70s
20MB	52.43s	77.42s

The Twitter User Filter application compute the number of Twitter users who have age larger or equal than 50. The topology contains a filter bolt and a count bolt. As we expected, Crane runs faster than Spark on the two smaller datasets and runs slightly faster on the 1MB one. The results are as shown. The re-running time upon node failure can be really short or about 30% of the original running time, depends on the stage a node at when it fails.