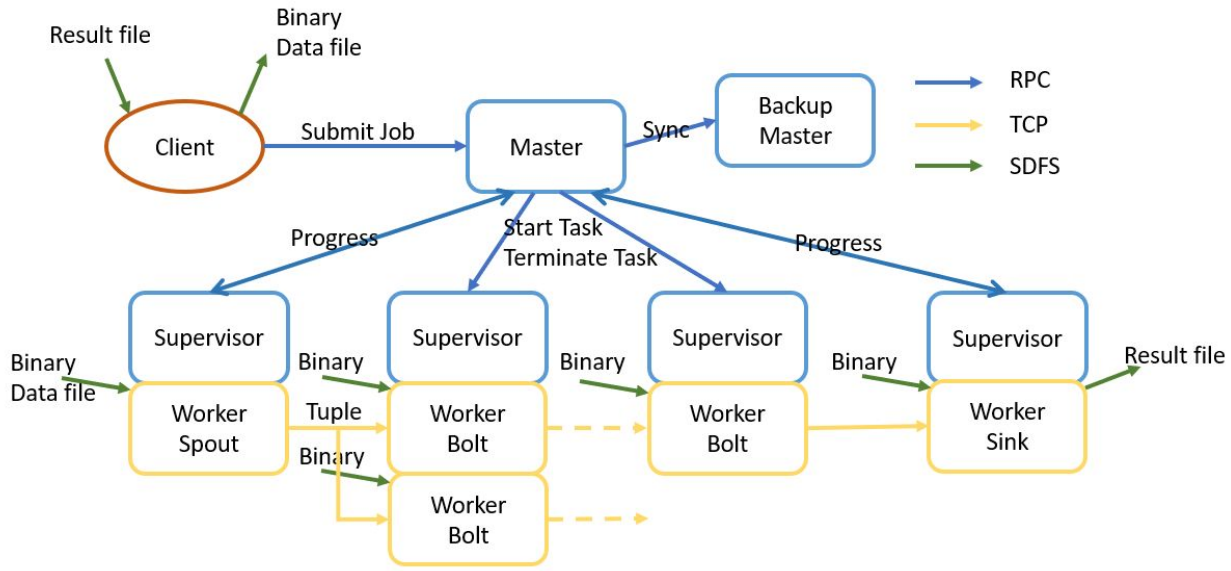
CS 425 MP 4 Report

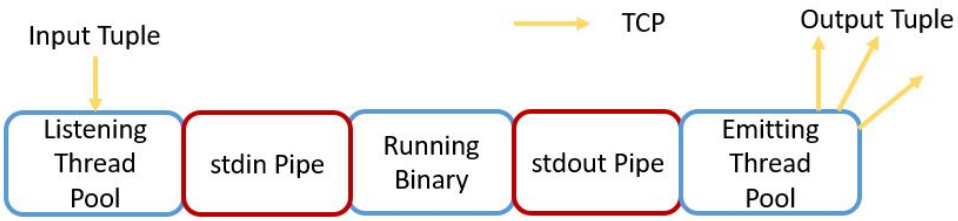
Ziang Wan (ziangw2), Yan Xu (yanxu3) - Group 58

System Design

Our Crane system is implemented in Golang. The overall design can visualized as:



1. Our Crane system has two components: the ***master*** node that controls task execution and job results; the ***supervisor*** node that spawns and terminates workers. We use TCP to send data tuples. We use RPC for job control communications. We use MP3 SDFS for file transmissions.
2. The entire application is called a ***job***. Each job has a ***topology*** that describes the job structure like what storm does. Each topology element is a ***task***, which can be either a ***spout***, ***bolt*** or ***sink***.
3. Master node executes our ***topology scheduling algorithm*** that evenly distributes work to the entire cluster and support ***user-defined parallelism for bolts***. We use ***random-pick*** as our grouping strategy. Upon failures, the master node will terminate all existing tasks, return the scheduling algorithm and start the entire job all over again.
4. The supervisor node spawns a ***worker*** upon requested by the master node. The worker communicate with the executable binary through ***stdin and stdout pipes***. It also has a go routine ***thread pool*** that handles network communications. To visualize the general mechanism of the workers:



System Performance Analysis Metrics

1. ***Throughput***: it is measured as the maximum number of tuples our system can process within a second. The throughput of our system is limited by network communication between workers. We implement a thread pooling mechanism that allows 512 network threads for each VM. The implementation leads to a ***maximum throughput of ~1200 tuples/sec***.

2. ***Latency***: it is measured as the time it takes for a tuple to start at the spout and end at the furthest sink. According to our tests, the network -> stdinPipe -> stdoutPipe -> network process will take ***~40 milliseconds per node*** for each incoming tuple.

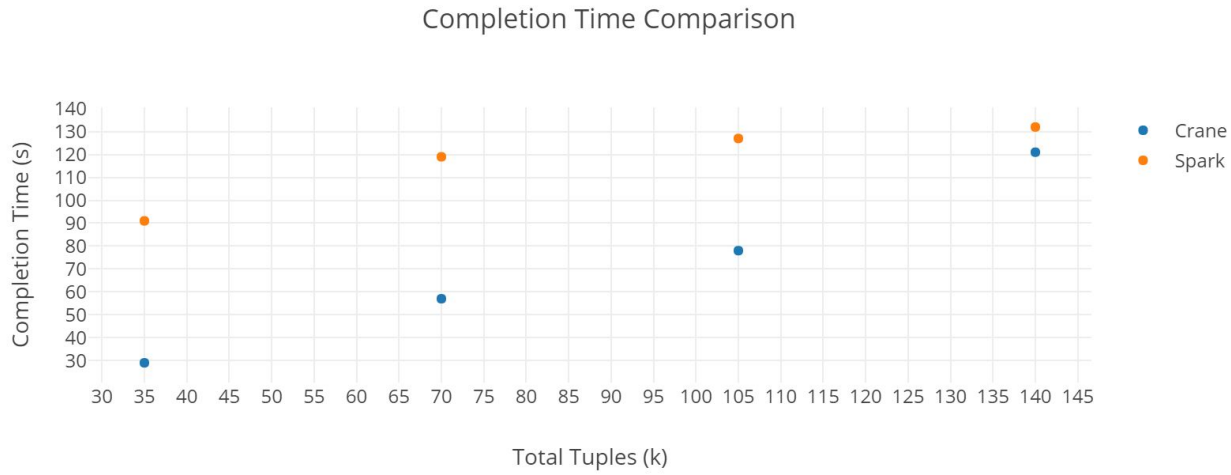
Sample Applications & Comparison with Spark Streaming

We choose to use throughput as our comparison metric. We run both crane and spark on a cluster of 5 VMs. We make the source stream sends out data at full speed and compare the completion times given the same size of data.

1. Reddit comment daytime distribution: we count the distribution of reddit comments over the time of a day.

Crane Topology: Spout → Bolt (3 VM Parallel) → Sink

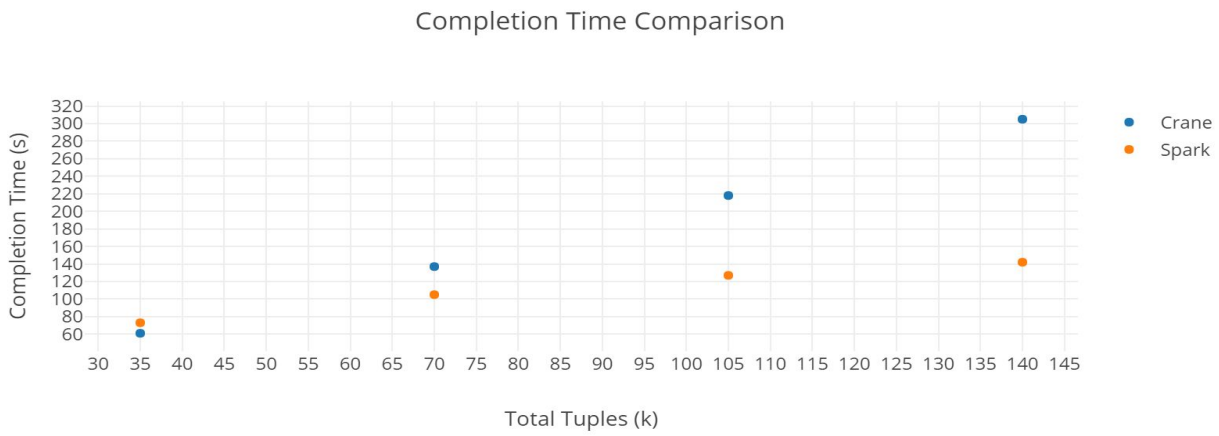
Spark Topology: Map → Filter → Map → Map → Reduce



2. Reddit comment word count: We count the occurrence of English words used in reddit comments.

Crane Topology: Spout → Bolt (2 VM Parallel) → Bolt (2 VM Parallel) → Sink

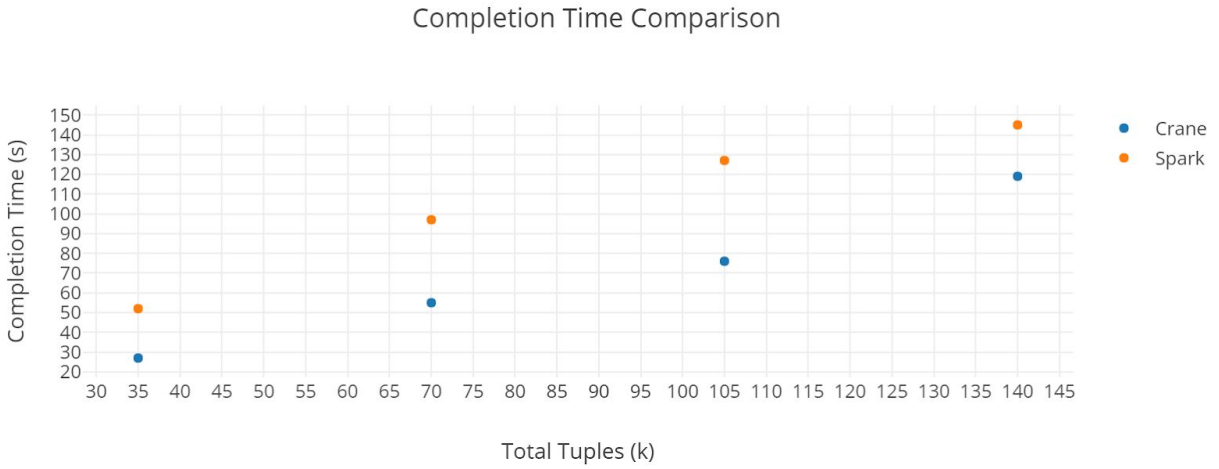
Spark Topology: Filter → Map → Map → FlatMap → Filter → Reduce



3. Top Ten Active Reddit User: we count the top ten users that make the most reddit comment so far.

Crane Topology: Spout → Sink

Spark Topology: Map → Map → Reduce → Transform



Trend Analysis

1. While both the completion time for Crane and Spark seem to be a line, the Spark completion time has a much larger y-intercept. In other words, Spark Streaming has a much higher setup cost. Our Crane system is more light-weighted and produces immediate results faster than Spark Streaming.
2. The throughput for our crane system drops when the topology gets more complicated. The reason might be our thread pooling implementation. It is observed that the we constantly reach the pooling limit and thus waste time waiting for previous goroutines to complete. It is unclear to me what the spark topology is, or how it differs across the three applications.
3. For simple topologies, our system has a higher throughput than Spark Streaming. When our system encounters more complicated topologies, its throughput is lower than Spark Streaming.