CS 425 -- MP4 Report Group 3 Ankit Gohel (ankitng2), Mohammad M. Ahmad (mmahmad3)

Architecture

Our architecture follows that of storm: We have a master node (called the Nimbus) which gets a job from the Crane client, and assigns jobs to different nodes in the cluster. The Nimbus also handles node failures and reassigns jobs on the failed machine when failure information is conveyed to it by the failure detector protocol from MP2 running in the background. Each node runs a supervisor, which listens for assigned jobs from the Nimbus.

Data Flow

To submit a job, the client takes in a file-path to a YAML file. The YAML file defines the topology and functionality of the components in Crane. The user must specify the components used in the topology, such as spouts and bolts and the attributes associated with them. The details are passed to the Nimbus, which is responsible for resource allocation and scheduling, assigning tasks to nodes. It is also responsible for keeping the system fault-tolerant using the failure detector component (described in MP2). In case of any failures detected, the Nimbus restarts the running applications, re-assigning tasks to nodes that are alive.

Each worker node (VM) has a supervisor running. When the Nimbus contacts the supervisor to assign task(s) on the server, the supervisor starts a worker thread for each task. Workers can be either spouts or bolts. Spout is a worker that is responsible for reading in a stream of data and forwarding to its bolt child(ren). Processing is managed by bolts which can perform one of four actions (per bolt):

- Filter: bolt removes tuples that do not satisfy the condition specified by the user, forwarding the remaining tuples to its child(ren), if any.
- Transform: bolt performs modifications on received tuples, transforming them into new tuples, and then forwards them to its child(ren), if any.
- Aggregator: bolt allows state-like behavior to perform aggregation functions, such as “count”. The state-like functionality allows storing a previous aggregate result which other bolts cannot do.
- Join: bolt performs a SQL-like join based on a user-specified attribute to combine tuples with corresponding tuples in the filestream, when possible. The resulting tuples are forwarded to the child(ren), if any.

A bolt that does not have any child is called a “sink”. A sink is responsible for writing the final tuples to an output file. The file is stored in SDFS (described in MP3) and sent back to the client using SDFS APIs.

Programming framework

Jobs are written in the form of YAML files, which the user then submits via the **start** command on the Crane client. They are written in the following structure:

<worker_id>:
type: spout/bolt
input: This field needs to be set for the spout. The input can be a file/database.
sink: True/False. If this is set to true, this worker can have no more children, and has to specify a way to collect the output.
children: List of worker ids that are the children of this worker - this is how the topology is specified

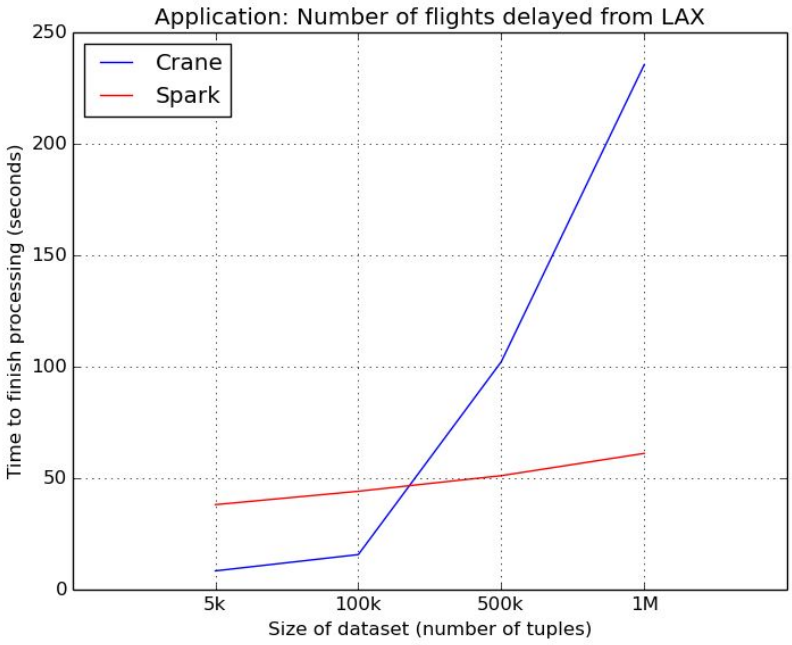
function_type: transform/filter/join/aggregate. If join is specified, then additional fields join_on (which columns to join on), and join_input (file/database to use for join)
function: The function to be executed by the user, as a string. This is later converted to a function object by Crane using python's eval method.
output: Specifies the file name, where the output of the job will be written. The file is written to SDFS to ensure fault tolerance

Multiple such workers can be written to specify an entire job, and any DAG based topology is supported. The user can configure everything apart from the IPs and ports of the workers, which are dynamically assigned by the Nimbus based on the list of alive machines in the cluster.

Applications

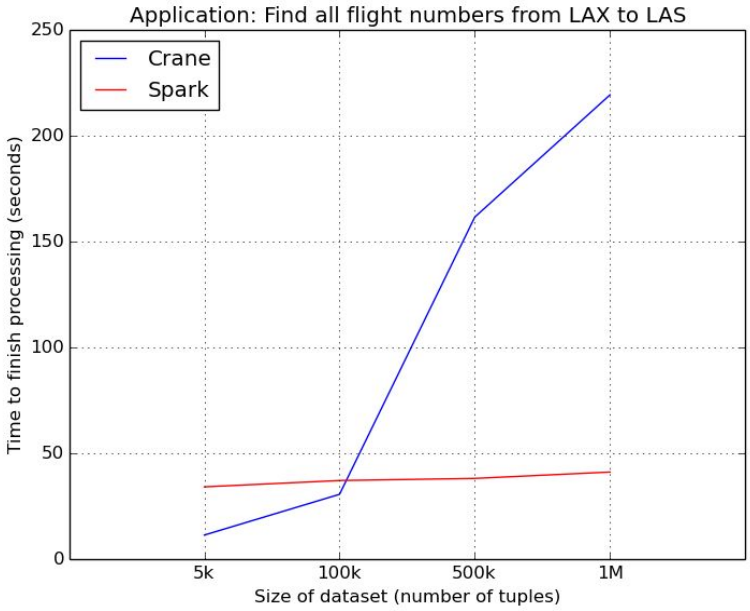
1. Counting the number of flights delayed from LAX airport

For this application, we used the airline dataset from http://stat-computing.org/dataexpo/2009/the-data.html. Each tuple has multiple fields: arrival and departure airports, airline name, flight number, tail number, time delayed at arrival, time delayed at departure etc. We used a topology consisting of one filter to filter tuples where the departure airport is LAX and departure time is greater than zero, one transform and one aggregator to count all such flights. The graph shows the performance of Spark on a cluster containing 1 master and 5 slaves, and Crane on this application for different sizes of the dataset.



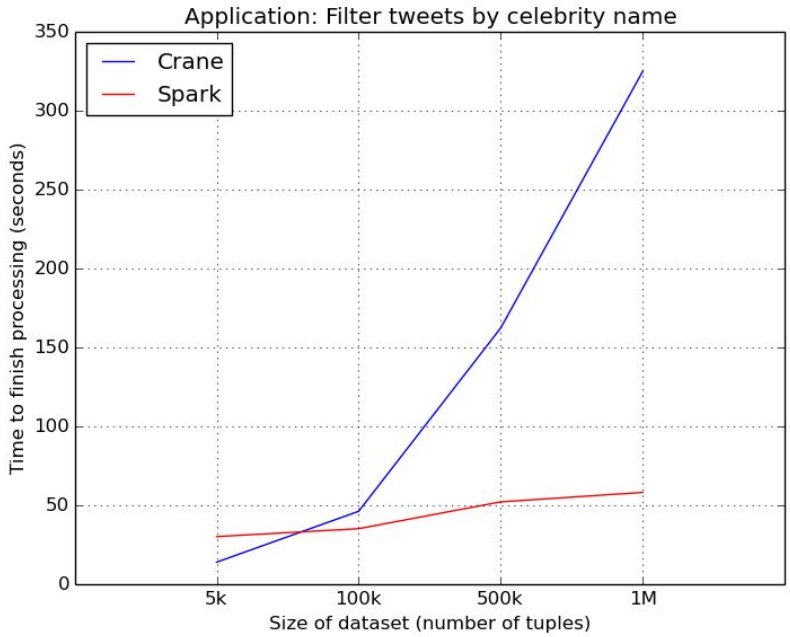
2. Finding all flight numbers for flights between LAX and LAS airports

We used the same dataset as in [1] for this application. The topology for this consists of a filter to filter for departure and arrival airports, and a transform to concatenate the airline name and the flight number. The graph for performance looks similar.



3. Counting tweets

For this application, we used a dataset of tweets from https://www.kaggle.com/kazanova/sentiment140. Each tuple consists of a tweet ID, the date and timestamp, user ID and the text of the tweet. We use a tree topology for this application, with two filters in parallel - one which counts the tweets made on a Monday, and the other counts the tweets which mention '@mileycyrus'. These two filters lead to a transform and an aggregator to count all tweets satisfying either of the two filters.



Looking at the plots for the three applications, we can say that Crane performs better than Spark Streaming on smaller datasets (upto 100,000 tuples). This is because Spark Streaming takes time initially to setup and start executors. However, once the setup is complete, Spark streaming easily beats the processing time of Crane. Thus you should choose Crane if you have smaller jobs, and Spark otherwise.