CSCI-591: Pipelines Testing and Release Strategy

Eric DiGiovine and Nate Woods
edigiovine75@gmail.com and Nathan.J.K.Woods@gmail.com
Department of Computer Science – Montana State University
April 29, 2016

I. Abstract

The operating cost of modern data-centers is the primary concern when considering "cloud" based algorithms. These data-centers contain numerous machines that run distributed, graph based algorithms in order to fully take advantage of the available resources. We attempt to lower the operating cost of such systems by leveraging the graph structure during the testing phases of of a software release cycle. Applying this technique involves modifying typical system hypervisors and the ability to support a multi-staged environment. We use the term multi-staged to concisely describe any composite environment consisting of a development environment, a QA environment, a staging environment and the production environment. We verified this result by running a sample application on both architectures and measuring operating cost of the underlying systems. Our tests confirm, for this case, that operating costs can be reduced significantly by this modification of data-center hypervisors.

II. Introduction

In modern Software Engineering, graph decomposition can be used to construct systems capable of scaling across datacenters efficiently and reliably. This technique allows for modular software design and is extremely powerful. Unfortunately, the testing and releasing of such systems has not taken advantage of this modularity and has fallen back on existing methods resulting in high operating costs. We propose a testing strategy that takes advantage of a decomposed structure with the intent of decreasing operating cost. Others should care about this research as it may reduce datacenter operating costs. We formally give both our hypotheses for this research:

Null-Hypothesis: The operating cost of DAG systems remains constant, regardless of stateful nodes or existing practices.

Hypothesis: By using stateful nodes when routing traffic, we can reduce the operating cost of DAG systems.

III. Background

With the advent of "Big-Data", many techniques[6][7] have been proposed to process data too large for a single machine. Using ideas borrowed from archaic unix pipes, modern software engineers have found a way to distribute these large data processing tasks across many machines by breaking the algorithms into a graph structure. This new family of algorithms is called "Stream Processing", which allows small machines to work together to process data much larger that could be handled by each machine independently. Over the remaining portion of this section we will discuss how the concept of UNIX pipes has returned in the realm of modern stream processing.

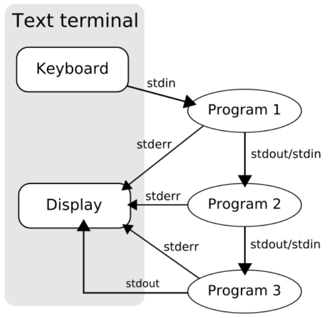

Fig. 1. Sample UNIX Pipeline

a) Unix Pipes: were developed to allow programs to communicate over a standardized communication model and were denoted as a vertical bar "—" when chaining commands. This communication model allowed for data to pass from program to program and allow notifications to be broadcast to the user without interrupting the regular flow of data between commands. This approach allowed small specialized programs to be used as building blocks to construct a larger program to perform specific data processing tasks. With over 50 of the standard building blocks in each unix/linux system available today, this design has withstood the winds of change and persevered through many technical revolutions.

b) Big Data: presented a multitude of problems for pipes as they were designed for dealing with data on a single machine. Instead of leveraging the existing technology, new technologies such as databases and large software monoliths emerged to help deal with the large amount of data that needed to be processed. As databases and monoliths became larger and larger, programs started breaking into small, distributed, reusable pieces that could be fanned out on dedicated machines. While this design mirrored existing traits of problems solved by Unix Pipes, instead of upcycling existing technology, new technology emerges to support this form of distributed processing. Soon the advent of a methodology came about that was near to the original intentions of Unix Pipes.

c) Stream Processing: is the family of algorithms designed to process data in the form of streams[8]. This concept treats large amounts of data as data streams and plumbs the data between nodes that run small forms of specialized code. This logic is truly the resurrection of Unix Pipes to process “big-data”. Googles MillWheel project seeks to support this concept of connecting programs with an interface. In addition, MillWheel provides persistent state managed by the hypervisor that allows the floating of processes from machine to machine in the event of a hardware overload/failure. Graph based systems have been treated like existing software and require three full clusters to stably operate (staging, production, rollback). This technique works well for legacy designs, but does not take advantage of the decomposed nature of a graph. Leaving the potential for wasted cycles and increased cost. For example, if a graph consisted of one million nodes, and updates were required for two nodes, the existing release strategy requires a full clone of the graph to be instantiated, just to test the two nodes requiring updates.

IV. Approach

As graphs grow in popularity, their testing and release methods should also evolve. Instead of maintaining separate graphs, we propose a method that maintains a subgraph of nodes within a single graph cluster. This subgraph represents a version of the system. By running multiple versions of a single node in different sub-graphs, we can use a single graph with more intelligent routing to reduce overall operating cost of graph-based systems. In the following paragraphs, we describe the important hardware and software attributes of our cluster setup that we implemented to test our hypothesis.

a) Observables: We are measuring the overall power consumption of the data-center which will allow us to directly analyze the overall operating cost of our system. Along with power consumption we are looking into exploring bandwidth and overall processing power required to produce a similar result. With these measurements in mind, we will be looking into both hardware and software solutions to the cluster, to see which proves to be more efficient in minimizing the overall cost of operation.

b) Architecture: The design of our system is based around the MillWheel architecture: a graph with nodes that can be distributed based on an identifier within each data stream. Our implementation uses a primary server (hypervisor), a multitude of agents and a modular API to fulfill this architecture. The primary server’s job is that of process orchestration, which communicates with agents, spins up nodes, and manages overall stream flows. Agents run on each physical machine and manage nodes of the overall graph. The modular API allows for standardized communication between each agent and server.

c) Implementation: On the software side, we use existing open source software to build a stable foundation for our system. At the lowest level we are using Docker[2] based modules that run on Debian[1] systems. We use Docker to push workspaces across the cluster, without the overhead of having to install the workspaces on each machine. We only installed Debian on each machine, which minimizes initialization time over the cluster. This setup allows us to use existing Docker cluster monitoring software to monitor the overall performance of the system. These systems will communicate via the NATS[3] protocol which guarantees 5-million messages per second per NATS server. Using Golang[5] as our programming language allows the performance of C with simplified concurrent processing and garbage collection. Finally we use Protocol Buffers[4] as our standard method of communication between systems since it allows versioned and backwards compatible messages that are compressed by design, reducing the overall network overhead.

A. Test Subject

In order to observe the overall performance of each operating approach, we built a distributed web-crawling algorithm. This algorithm uses a three node graph (Index, Crawl, and Store) to accomplish the task of crawling large scale websites. Additionally, in an attempt to reduce the number of variables that arise when using the Internet we have constructed a subject website that provides consistently sized pages and removes the probability of external network delays skewing our results.

a) Graph Structure: Our sample web-crawler splits the process of crawling pages into 3 nodes: Indexing, Crawling and Storing. Indexing is a process that maintains a list of previously seen web pages. This index store is considered a single source of data and is unable to be fanned out to multiple machines. Alternatively, the Crawler is designed to be stateless which allows it to be fanned out across a multitude of machines. The Crawler receives pages from the indexer that needs to be crawled, and emits more links as found on the initial page along with the page content to the Storage process. Finally, the storage process can be configured in either a stateful or stateless configuration and is used to persist the content from a particular subject site.

b) Subject Site: To reduce the amount of external dependent variables, such as network lag and server delays, that are involved when building a web crawler, we constructed a website generator to mitigate these problems. The site generator constructs a doubly linked ring of web pages. This ring-structure ensures that the random link generation process does not construct any isolated sub-portions of the web-site. For the purpose of our testing we constructed a site with 10 million node website with 30 links per node (including the two links for previous and next pages in the ring). By writing this in GO, we were able to create an effective random site generator in 100 lines of code (including comments) that can generate consistently sized pages that respond in 100s.

c) Hardware: We had limited resources available when building the cluster. The cluster contains nine Dell Optiplex 960s and three Dell Optiplex 745s. The Dell Optiplex 960s have 8GB of ram with 2.83GHz Intel Core 2 Quad processors. The only exception here is that one of the 960 machines only had 4GB of ram available. The Dell Optiplex 745s have 2GB of ram with 2.13GHz Intel Core 2 Duo processors. The computers were connected through a 10/100 switch, which was the only switch available. Finally, we used a virtualized switch that ran pfSense on a 2006 Mac Pro. While none of the networking technology was up to todays modern standards of performance, the communication between nodes was designed to be smaller in payload size, allowing the reduced networking performance.

V. Methods

To test our hypotheses, we constructed the web-crawler with hypothetical updates to each node. Using these updates with various system configurations allows us to compare operating costs. Our Control configuration will consist of several graphs running side-by-side in a modular (pod) configuration while our Candidate configuration uses a single pod to operate sub-graphs. By measuring the observables mentioned earlier, we will be able to compare the configurations operating costs.

A. Tests

To verify our algorithms work correctly, we mirror three versions of the graph algorithm. Each of these versions operates a slightly different version of the graph, with updates to a single node as the difference between graphs. By measuring the duration and power consumption of this configuration, we will have a baseline to compare our test results with in order to show overall improvement. Once a baseline tests are executed, we run the same tests in a single graph with multiple versions of nodes and a more intelligent manager that can route requests through different paths. This will allow us to hopefully reduce the number of workers in the graph and potentially reduce the overall operating cost of the entire system.

VI. Experiments

Using various pod configurations and multiplicities we were able to compare our proposed combined graph operating costs with that of existing mirrored-graph based operating costs. Our first experiment limited us by the amount of memory available to the primary servers due to a poor choice in hardware configurations. Our second experiment removed this impediment by using a machine with more memory as the primary server, which allowed us to measure the operating costs of our web-crawler in an existing infrastructure. Finally, our third experiment combines all versions of the nodes into a hyper-graph and allows the measurement of our proposed design.

We ran three different experiments on the cluster. Each experiment tracked the max store for three different rounds per experiment, increasing its size by a factor of ten each round. We observed the time taken to complete the web crawl, which is listed as the duration. We also took in an observed KWH measurement and compared it to a calculated version given the known watts used and the time taken to complete the web crawl. If the web crawl had to be terminated, we kept track of both a crawled and a stored set of pages at the time the algorithm was terminated.

A. Experiment One

For our first experiment, our cluster setup consisted of 3 pods, each consisting of three 960s configured as workers and a single 745 configured as the master. Each of the three pods operate a slightly different version of the graph (one node in each runs different code than the rest). The results for the first experiment can be viewed in Table I.

Rounds	1	2	3
Max Store	1k	10k	100k
Duration	42.16s	28m 53.62s	2h
Observed KWH	0.12	0.47	1.6
Calculated KWH	0.010	0.421	1.896
Crawled	n/a	n/a	65,513
Stored	n/a	n/a	22,377

TABLE I
Data From Experiment One

We first notice that in this experiment, the 745 machines are configured as pod masters. We discovered that pod masters need as much memory as can be given, so as to better route data within the pod. Operating all three pods, the system consumed 875 Watts and pulled 6.22 Amps. Based on the results obtained from experiment one, as the max store increases, the duration and the KWHs also increase. Once the experiment reached round 3, which consisted of a max store of 100k, the duration was estimated around 2.1 hours and was killed early. We ended up stopping test-run three early because the machine was spending so much time swapping data into and out of disk-backed swap, the algorithm was sitting in deadlock waiting for the memory swap to complete. This could have been improved upon by implementing disk-backed, paged queuing, but due to time constraints was considered as a future improvement.

B. Experiment Two

The results for the second experiment can be viewed in Table II.

Rounds	1	2	3
Max Store	1k	10k	100k
Duration	40.67s	4m 34.70s	30m
Observed KWH	0.02	0.08	0.34
Calculated KWH	0.010	0.067	0.438
Crawled	n/a	n/a	more than 100k
Stored	n/a	n/a	23,482
Crawl Duration	15s	36s	12m 1s

TABLE II
Data From Experiment Two

In the second experiment, we rectified the mistake of having a weak machine as the head of the pod. One of the 960 machines was made pod master, and from the experimental results we see slight improvements across the board. This experiment also used all three pods, which means it also used 875 Watts and pulled 6.22 Amps. Another addition to this experiment is the reporting of the "Crawl Duration" value, which is the time that the system had crawled the "Max Store" amount of pages, but was waiting for the "Storage" queue to complete the "Store" requests. Based on the results in this experiment, and remembering that a 960 machine is now the pod master, we see an overall decrease in duration across the board, and we also see an increase in the number of pages crawled and stored, with the increase in crawled pages being more substantial than the increase in stored pages. From the results we also know that the last completed crawl finished at 12 minutes and one second, which implies that the rest of the overall duration was used for storing the pages.

C. *Experiment Three*

The results for the third experiment can be viewed in Table III.

Rounds	1	2	3
Max Store	1k	10k	100k
Duration	31.54s	4m 48.74s	45m
Observed KWH	0.01	0.02	0.18
Calculated KWH	0.002	0.019	0.179
Crawled	n/a	n/a	94,127
Stored	n/a	n/a	34,216
Crawl Duration	7s	56s	n/a

TABLE III
DATA FROM EXPERIMENT THREE

The third experiment is testing our proposed method. For the configuration, we only use one pod, with a 960 machine as the pod head. This a huge reduction in hardware usage, going from using three pods to using just one pod for this experiment. The graph run by this pod consists of all the versions of nodes from the previous tests, with different paths through the overall graph for each request. The singular pod used 239 Watts and pulled 2.15 Amps. These new number correctly reflect the amount of machines in operation, since these values are about a third of the values obtained from experiments one and two. Based on the results of this experiment, we do see a decrease in energy consumption, but we also see an increase in duration. What we also notice, looking at the third round of the experiment, is that experiment three searched more space than experiment one and slightly less than experiment two, but it stored more values than both experiment one and experiment two.

D. *Discussion*

The results in Table IV are summarized from Table II and Table III.

For discussion here, is it important to note the KWH multiplied by the actual runtime of the cluster to get the "Calculated" energy consumption. This was done because of the inaccuracy of measuring devices in the ability to measure KWH consumption. One can compare our calculated values with the "Observed" values to see how much difference was noted. This time difference came from the time between meter

Experiment/Rounds	2.1	2.2	3.1	3.2
Max Store	1k	10k	1k	10k
Duration	40.67s	4m 34.70s	31.54s	4m 48.74s
Observed KWH	0.02	0.08	0.01	0.02
Calculated KWH	0.010	0.067	0.002	0.019
Crawl Duration	15s	36s	7s	56s

TABLE IV
DATA COMPARISON BETWEEN EXPERIMENTS TWO AND THREE

reset to starting the experiment and experiment completion to reading meter delays. As there was no computer readable interface to the meter, these measurements are subject to human error and delays.

Next, we can observe with relatively small increases in operation time, we can drastically reduce the overall operating costs of using such a system by combining the graphs into a super-graph. Thus confirming our hypothesis.

E. *Threats to Validity*

In the following paragraphs, we discuss some of our challenges with this project, and possible roots for threats to validity in our project.

a) Construct Validity: Our project presents a few construct validities that may affect the overall results. To start, we only ran our experiments on pods of size four. Based on the results obtained from our experiments, one could argue that by increasing the number of machines in a pod, one can achieve better computational power and exhibit drastic savings on energy costs and hardware costs. Another construct validity that needs to be discussed is the use of different machines in the pods. If all the machines in the cluster were the same machine with the same specifications, then one can speculate the results from our experiments would have improved slightly. Another construct validity that we identify is the lack of paged queuing within the pods of the cluster. As given in the experimental results, the runs at 100k max store size mostly halted due to constant memory swap, meaning the process could not finish. If paged queuing for the tasks had been implemented, we would have seen more fruitful results from higher max store values.

b) Internal Validity: In terms of internal validities, we made a solid attempt to reduce the number of causal arguments made about the experiments run on the cluster. We also will need to run more tests on different cluster sizes and pay more attention to hardware details (such as removing more hardware from independent machines to test their efficiency). Based on that fact, we refuse to make any causal claims about our method improving the efficiency of the cluster and reducing the amount of hardware used. What we can say is that there is a strong correlation between the amount of hardware used in a cluster and the efficiency of the cluster, both in terms of energy and in time.

c) External Validity: In terms of external validities, we did our best to not make any overarching generalizations about the results from our experiments. We simply note that we observed promising results from our experiments, and that only in the context of our project were we able to show the advantages of our proposed method.

F. Contributions

The main contribution of this project is the using of a combined graph structure to reduce the overall costs of development, QA, and staging environments. Seeing positive experimental results gives hope to the reduction of resources used to produce code in a multi-staged environment.

VII. Conclusion

In summary, we investigated the amount of resources needed to successfully operate a multi-staged environment, while at the same time confronting the main issue with multi-staged environments, which as was stated earlier is the amount of resources used and required to run tests. We also provided a possible solution to the amount of resources used by consolidating the standard multi-staged environment into a combined graph structure, where we preserve all the testing paths used in the standard method yet reduce the amount of resources needed to test the new structure. We tested this idea on a cluster running a web crawler and found that while the time taken to complete increases, the amount of resources, both hardware and energy, is drastically reduced. What we also take into account implicitly is the time required to run each environment independently vs. everyone using the same environment. Overall, we have shown that by combining the environments used for development, QA and staging, with everyone sharing the environment, there is great potential for increased efficiency in testing and reduced overall cost of testing.

References

[1] Debian the universal operating system. https://www.debian.org/. Accessed: 2016-04-29.

[2] Docker build, ship, and run any app, anywhere. https://www.docker.com/. Accessed: 2016-04-29.

[3] NATS cloud native, open source, high performance messaging. http://nats.io/. Accessed: 2016-02-20.

[4] Protocol Buffers google developers. https://developers.google.com/protocol-buffers/. Accessed: 2016-04-29.

[5] The Go Programming Language go is an open source programming language that makes it easy to build simple, reliable, and efficient software. https://golang.org/. Accessed: 2016-04-29.

[6] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] Michael Stonebraker, Uur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.