Nicholas Marcopoli and Austin Sura

December 7, 2018

Douglas Thain

University of Notre Dame

Distributed Video Hosting with DigitalCinema

ABSTRACT

Video hosting sites like Google's YouTube and Vimeo need to ensure that their videos are reliably served to their extremely large user base. The most pragmatic way to achieve such reliability is the use of the distributed storage systems and load balancing models that can sustain heavy traffic over long periods of time. The Hadoop File System (HDFS) architecture is one way of achieving a reliable system that can sustain itself when network traffic becomes too great for any single server to handle. In this paper, we will evaluate the effectiveness of using HDFS as a distributed storage solution for a streaming video service by comparing its total and client throughput to those of a single server solution under high traffic conditions. We find that the HDFS solution performs approximately two times better on this metric.

INTRODUCTION AND BACKGROUND

Popular video streaming websites, like YouTube, store video data in large data centers in order to constantly serve video to their large user base. Their video data is likely replicated across many servers to ensure that upon server failure, users are still able to access videos, as if the server never failed. The Hadoop File System (HDFS) operates in a similar manner. In a Hadoop cluster, data is spread across *N* data nodes with a replication factor of *M* (these numbers default to 3 and 3, respectively). The name node directs uploads to any of the data nodes and downloads to any data node where the desired data has been replicated. The theoretical advantage such a setup is twofold: the cluster can divide the load of users accessing the website among each datanode, while also providing protection against a server failure on an individual data node. HDFS can be accessed both from the command line and from an internet browser using WebHDFS.

Amazon Web Services (AWS) provides dedicated virtual machines for users to run applications on with their EC2 web service. The VMs are divided by compute capacity and are priced accordingly - for this project, we utilize the t2-micro and t2-xlarge instances. EC2 instances are usually charged by the compute capacity per hour or per second, and are priced in four different groups: On-Demand, Spot, Reserved, and Dedicated Hosts, where On-Demand is the most flexible and expensive of the four. We use On-Demand instances for this project due to the simplicity in running software at any point with minimal interruptions. AWS also offers the Elastic Beanstalk service for deploying scalable web applications, which allows a user to upload a zip file of their PHP, Python, Node, etc. web application. We used this service to host our PHP application in order to abstract away the scalability of the web application specifically, and focus more on the scalability of our video storage model with Hadoop.

OVERVIEW AND SETUP

Our initial plan for our service, DigitalCinema, was to create a small Hadoop cluster using four t2-micro EC2 instances, where one instance would serve as the name node and the other three would serve as data nodes. We would follow an online tutorial[1] to ensure setup could go smoothly and we could begin uploading videos. Unfortunately, our initial setup was riddled with stubborn challenges. First, the tutorial's link to the Hadoop software was broken, so we instead tried following the tutorial using an old source version of Hadoop. Many of the tutorial steps did not align with our version of Hadoop. Eventually, we located the correct version of the software and were able to complete the tutorial.

After correctly setting up Hadoop, we uploaded a simple text file to the root of HDFS and attempted to access it from HDFS's web client, WebHDFS. When we tried to download the file, we were met with our second problem: WebHDFS directed us to an internal AWS URL used in communications between the name node and data nodes. This URL is inaccessible from any outside source. We tried a number of different potential solutions to fix this issue. First, we tried editing the /etc/hosts and /etc/hostname file on each of the server to match the public IP addresses of each server in an attempt to allow the name node and data nodes to connect through publicly accessible IP's. This was unsuccessful and resulted in the disconnect of each node in the cluster. Our next attempt was to simply use a Python script to replace the internal IP with the node's corresponding public IP. This was successful, and we proceeded using this strategy: whenever we called the name node to upload or download a file, we would replace the internal IP with the public IP and upload or download accordingly with our script.

Our next step was to upload video files to the HDFS cluster. We were easily able to do this using Hadoop's command line interface on the name node, but wanted to do so without interacting with the name node's shell at all. After looking through the official Hadoop documentation[2], we discovered that WebHDFS supports downloading to HDFS through a simple

GET request, and uploading with a simple PUT request. The curl command to upload is shown below:

```
curl -i -X PUT "http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=CREATE
                    [&overwrite=<true|false>][&blocksize=<LONG>][&replication=<SHORT>]
                    [&permission=<OCTAL>][&buffersize=<INT>]"
```

We used the above curl commands to upload several MP4 video files. These files could easily be downloaded by accessing their respective WebHDFS URLs in a browser as well as through a curl command. We wanted to stream these videos through our website, however, instead of forcing users to download the files and play them on their local machines. We tested the streaming capabilities of the files by entering the files' WebHDFS URLs into VLC Media Player's "streaming" option. The videos streamed successfully, and we believed that this would translate to streaming success in our web application. We wrote a simple HTML website that used the `<video>` tag to stream data from our Hadoop server, but none of our uploaded videos would stream. After hours of struggle, we finally realized that VLC transcoded the MP4 files to a streamable format, OGG. When we manually converted the uploaded MP4 videos to OGG, they streamed successfully. The transcoding process is very expensive, so rather than transcode every uploaded MP4 video ourselves, we decided that it would be appropriate if we offloaded this step to our users. Our web application now requires that only the OGG video format can be uploaded to our service for streaming; any user who wishes to upload a video must first convert it to OGG. We then created a Python script that would automatically upload these OGG files to our HDFS cluster.

We added the finishing touches to our website - a search feature for each of the uploaded videos, a clean, aesthetic interface with Bootstrap, and an upload confirmation screen shown after the user successfully uploads a file. We zipped our web application and uploaded the file to Elastic Beanstalk to host our web application, but immediately found that our application could not interact with our Python scripts for translating the internal curl URLs to publicly accessible URLs and downloading and uploading to the cluster. This resulted in no files being listed as on the cluster in our "Home" video selection screen, and the user could not upload any files. To solve this, we removed our Python scripts and ported the code to PHP, which could be used in Elastic Beanstalk. The videos then showed up on our selection screen, the user could view them, and the user could upload new videos to HDFS. We also ensured that our unscaled website was structured properly, in that it did not make use of our HDFS cluster and simply used its own local file system for video storage.

At this final point in our structure set up, our systems could be accurately described using the following diagrams:

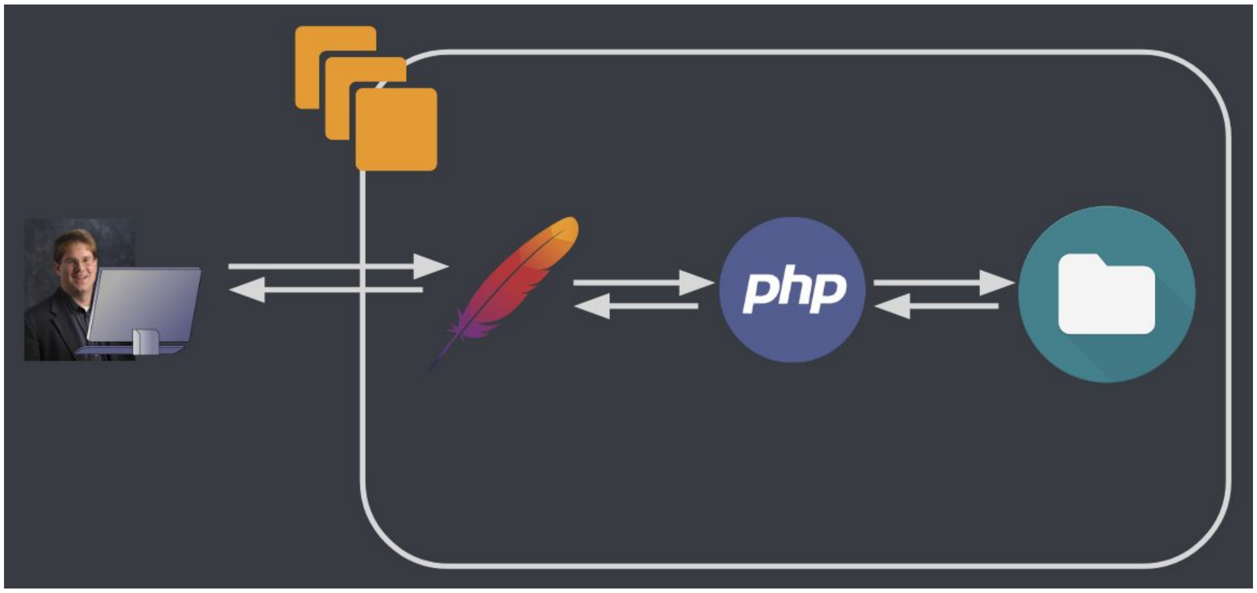


Figure 1: Unscaled Architecture

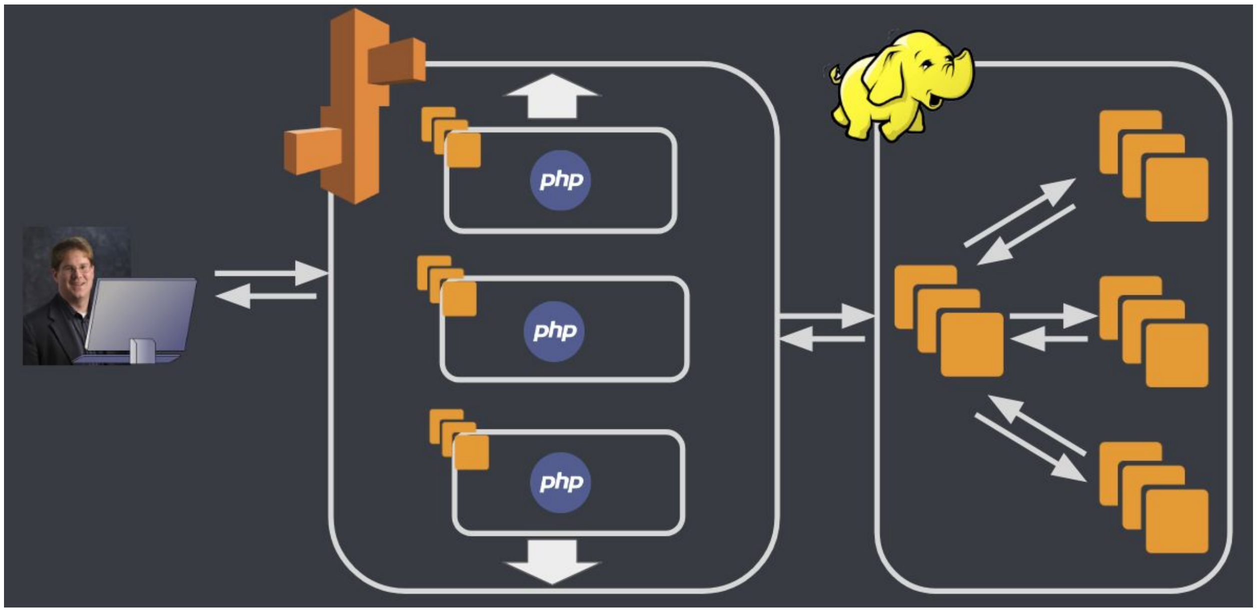


Figure 2: Scaled Architecture

Both our scaled (HDFS) unscaled (local on EC2) versions of the website were properly set up, and we proceeded to test both systems.

TESTING METHODOLOGY

We wanted to simulate many users watching videos on DigitalCinema as an evaluation of our HDFS video hosting solution. We considered the HTCondor system as a viable option for this evaluation. HTCondor allows a user to use many machines to run many jobs at once, often in parallel. We decided to make use of the Notre Dame HTCondor pool to simultaneously upload and download videos to and from the both scaled HDFS host and the unscaled host to provide a comparison of the technologies used. This method is inherently flawed, due to the fact that HTCondor jobs rely on machines not running existing jobs to run new jobs. Therefore, if the HTCondor pool is filled, the jobs will not run in parallel, and they will not accurately simulate a large user load using DigitalCinema. We decided to run our jobs during times of light HTCondor usage - namely, in the late evenings - so that most of our jobs would run in parallel.

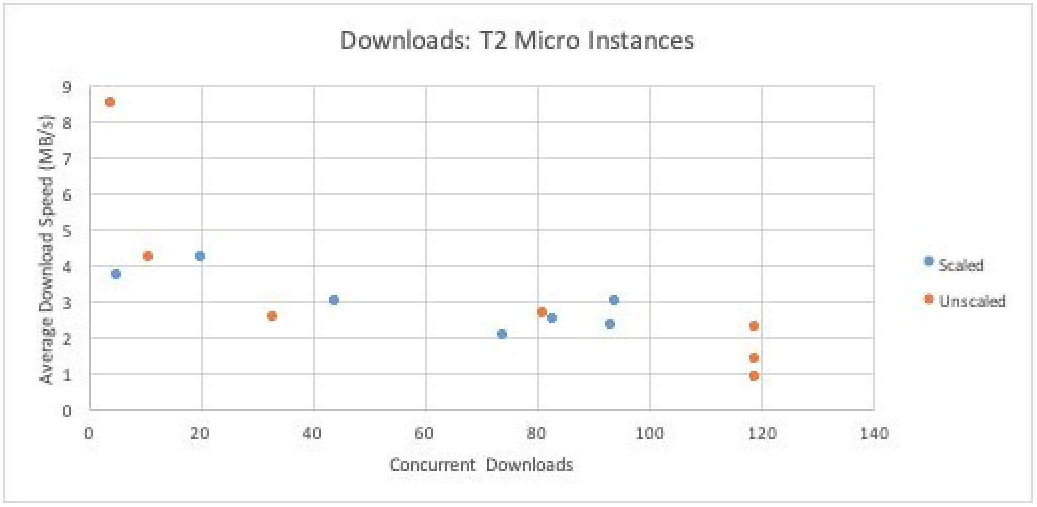
Another problem we had when running our tests was that many of the Condor machines are firewalled in such a way that they cannot access our web application. This causes fewer valid results than we expected. We also encountered a cap on the amount of users that could access the website at once. After sending 100, 250, 500, and (in the case of downloading) 1000 users, our results would max out at approximately 80 concurrent users for our t2-micro instances. In addition, we were only able to run three concurrent uploads at once on our scaled HDFS host. We ran a second set of tests on t2-xlarge instances with exactly the same configurations in an attempt to resolve these seemingly resource-related issues.

We ran four different tests using Condor, which evaluated the following scenarios: unscaled downloading, unscaled uploading, scaled downloading, and scaled uploading. Each script created a condor submit file with the desired number of jobs and submitted to the pool. In the unscaled downloading script, each condor job executed a python script that simply executed the curl command to download a video off of the unscaled server's local filespace. In the unscaled uploading script, the jobs executed a python script which used the scp command to upload a video to the unscaled server's local filespace. The testing for the scaled implementations was more involved. For the scaled downloading, our helper python script first executed a WebHDFS GET command on the datanode. This curl command produces an error message with the invalid internal link. Using string manipulation, our script executes a second GET on the correct link to download the desired video from the HDFS cluster.

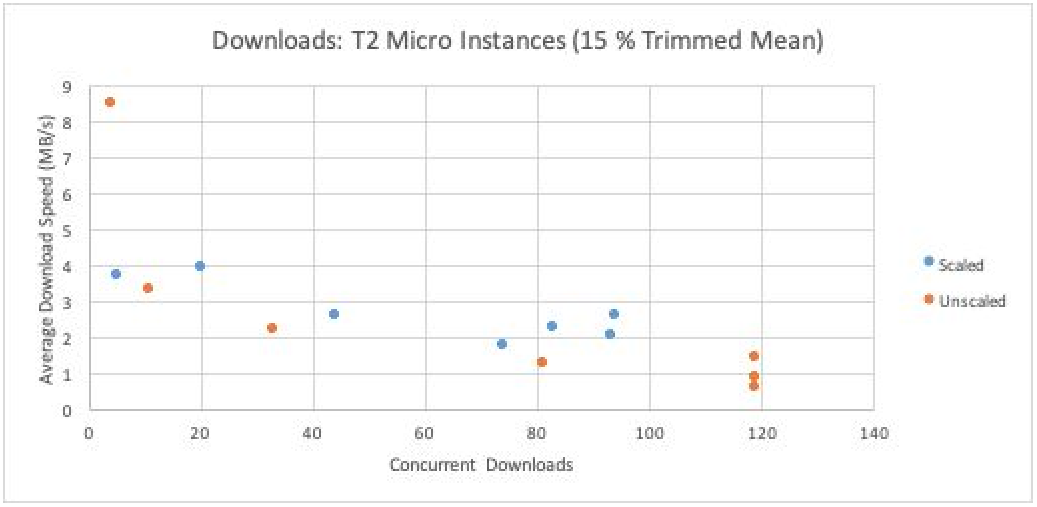
The scaled uploading testing script functioned in a similar way. First, a WebHDFS PUT command is executed onto the name node, and again a faulty link is returned. Using this link, a correct URL is created and a second PUT command is executed to send the video to the HDFS cluster. For the two different downloading scripts, both scaled and unscaled, we executed trials with 10, 20, 50, 100, 250, 500, and 1000 jobs. For the uploading scripts we ran trials with nearly the same number of jobs, but excluded the 1000 job trial due to space limitations on our EC2 instances. Each job that was executed would produce an output file that contained the data we desired, which we then concatenated and performed parsing to aggregate this data for later study.

TESTING RESULTS

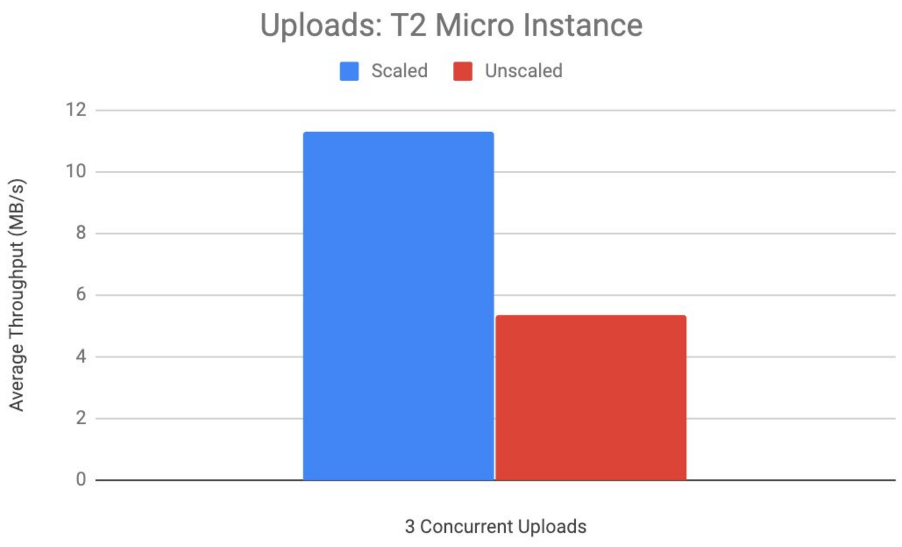
Below is a graph of the downloading test results for the micro instance:



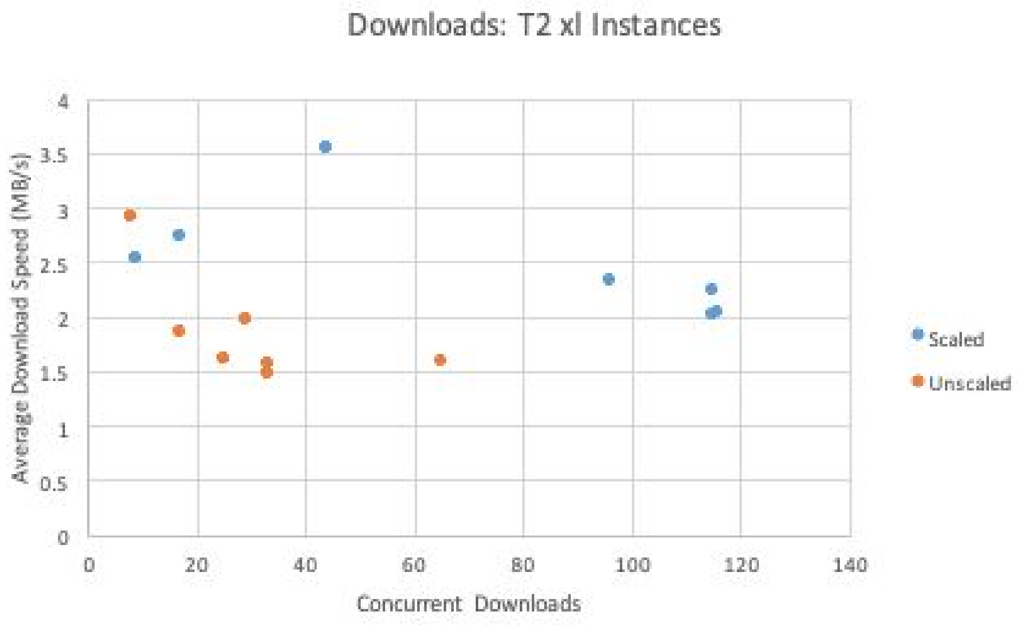
The X-axis displays the concurrent downloads that were achieved in testing and the Y-Axis displays the average Download speed per download. As you can see, unscaled implementation was faster for a small number of downloads, and were roughly equal as the downloads increased. These results surprised us as we expected a speedup using the scaled implementation. Upon further analysis of the data we noticed a large standard deviation and a high number of outliers in the unscaled data. We decided that calculating a 15 percent trimmed mean could provide a more accurate representation of the typical downloading speed, and this is shown in the graph below:



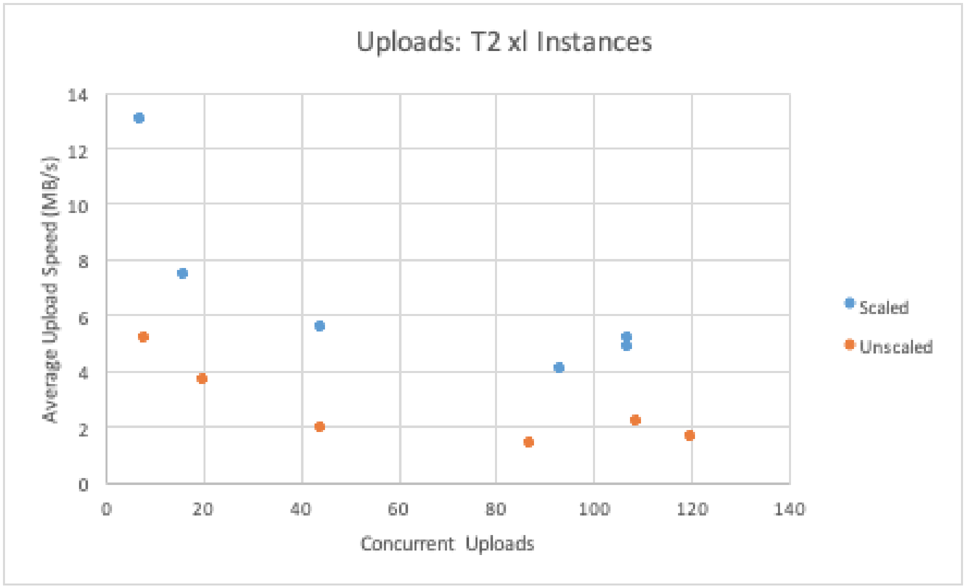
As you can see, taking the trimmed mean improved the relative performance of the scaled vs. the unscaled, but the difference is marginal. Next, a graph of the upload speed comparison for the micro instance is shown below:



Due to limitations of our Hadoop cluster on the micro instances, we were only able perform three concurrent uploads. Nonetheless, the scaled implementation performed much better than the unscaled with this small sample size. As you will see with the t2-xlarge instances, the performance benefit continues to be great with a larger number of concurrent downloads. Now, we will transition into the data collected from the EC2 t2-xlarge instances, starting with downloads:

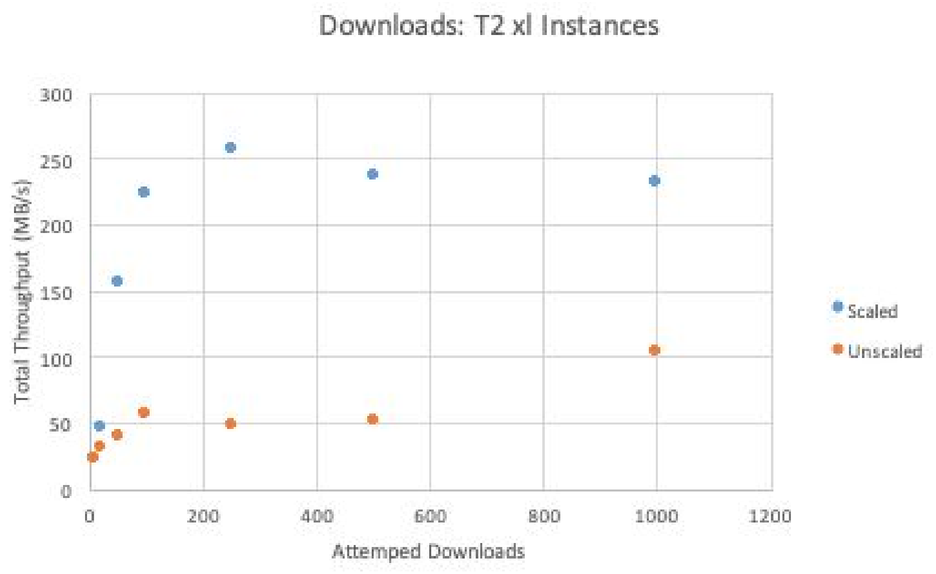


Unlike the micro instances, the improvement found with the scaled implementation is apparent. During our testing, the scaled version was able to sustain more concurrent downloads, while also generally having a faster average download speed. We did not feel that a trimmed mean was necessary for this data, since the scaled implementation was already clearly superior. There is benefit to using the scaled version when it comes to uploads as well, as seen in the chart below:

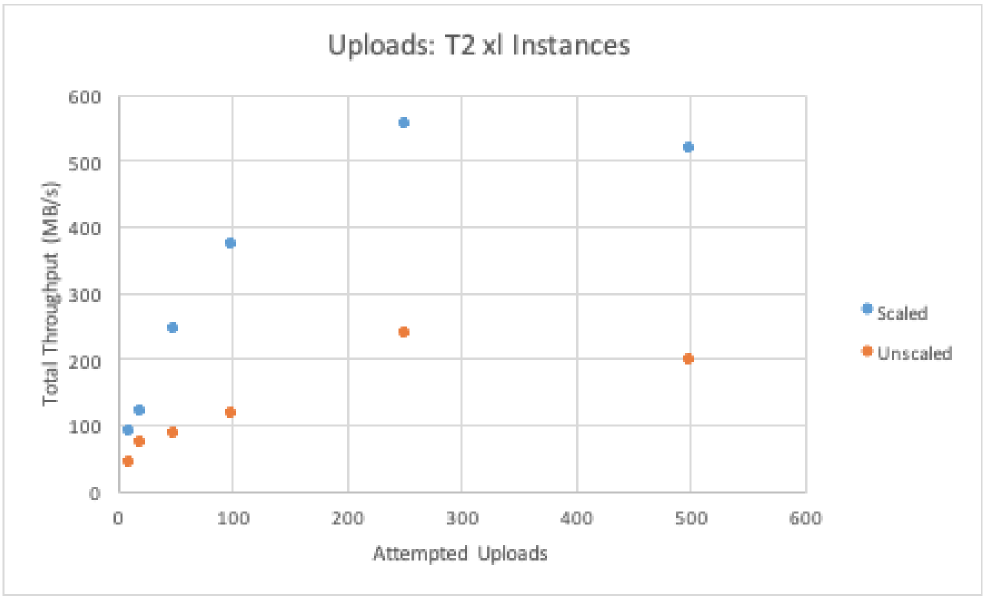


As demonstrated in the chart above, the scaled implementation was significantly higher than the unscaled implementation in terms of average upload speed. Both superior upload and download performance is likely due to the fact that the users upload directly to one of the three data nodes, thereby distributing the load to improve performance.

We also created graphs displaying the total throughput achieved for each number of attempted downloads. Attempted downloads aligns with the condor methodology mentioned above, 1000 attempted downloads corresponds to 1000 condor jobs. The throughput for downloading is displayed below:



This form of data truly displays the merits of our scaled system; the scaled implementation achieved throughput upwards of two to three times the unscaled version. This chart also demonstrates the diminishing returns experienced as the amount of concurrent downloads become very large. Similar results were found with uploading, shown below:



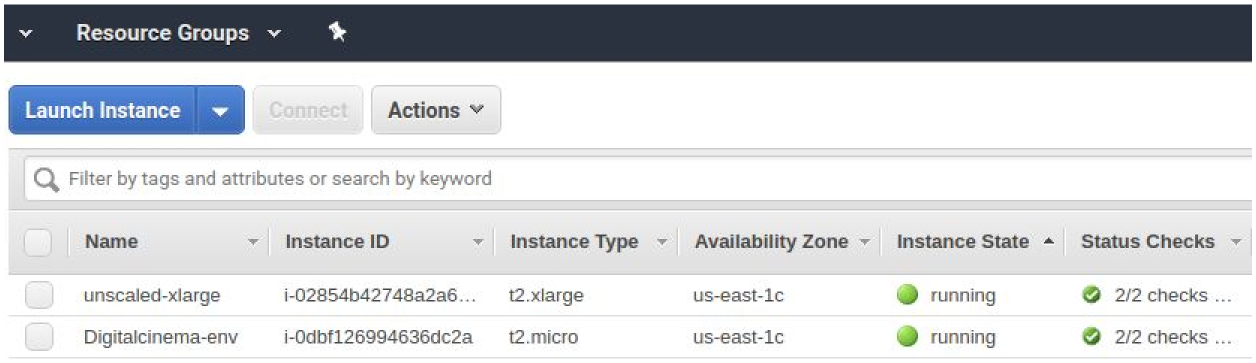
As expected, we again see a significant throughput increase with our scaled implementation.

CONCLUSION

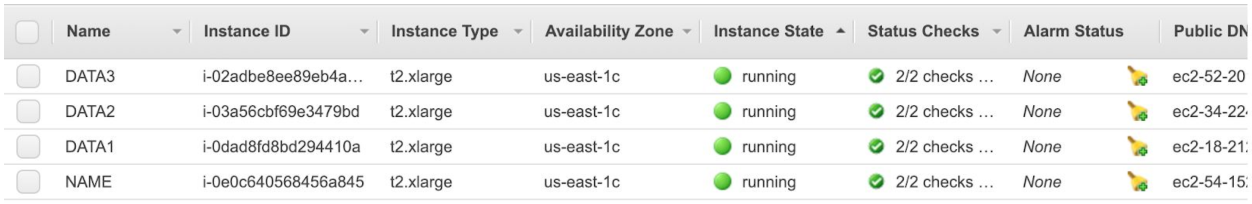
What was unclear in the t2-micro instances became quite clear in the t2-xlarge instances: the HDFS-backed web application performed significantly better than the web application served by its local storage when the user load scaled up to hundreds of times of the initial single user case. In each of our tests, the HDFS web application performed better, but in the t2-xlarge instances, the HDFS application had a nearly two to three times better total throughput, and was capable of serving almost double the amount of concurrent downloads compared to the local storage backed application.

We believe that this model would scale quite well. Given more compute resources, particularly more t2-xlarge VMs to run HDFS on, this model would likely perform well under even higher user loads. It remains for future work to determine the accuracy of this hypothesis, the improvement in throughput due to an increased number of data nodes and replication factor, and the cost-effectiveness of such a large model.

SUBMITTED CODE AND OTHER EVIDENCE



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks
unscaled-xlarge	i-02854b42748a2a6…	t2.xlarge	us-east-1c	● running	✓ 2/2 checks …
Digitalcinema-env	i-0dbf126994636dc2a	t2.micro	us-east-1c	● running	✓ 2/2 checks …



	Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status		Public DN
	DATA3	i-02adbe8ee89eb4a…	t2.xlarge	us-east-1c	● running	✓ 2/2 checks …	*None*		ec2-52-20
	DATA2	i-03a56cbf69e3479bd	t2.xlarge	us-east-1c	● running	✓ 2/2 checks …	*None*		ec2-34-22
	DATA1	i-0dad8fd8bd294410a	t2.xlarge	us-east-1c	● running	✓ 2/2 checks …	*None*		ec2-18-21
	NAME	i-0e0c640568456a845	t2.xlarge	us-east-1c	● running	✓ 2/2 checks …	*None*		ec2-54-15

Our code was submitted to our dropbox; above are screenshots of running instances to prove that we did indeed run a Hadoop cluster, an Elastic Beanstalk web application and a single unscaled EC2 instance.

REFERENCES

1. Sridhar, Jay. "How to Setup an Apache Hadoop Cluster on AWS EC2." Novixys Software Dev Blog, Novixys Software Inc., 30 Mar. 2017, www.novixys.com/blog/setup-apache-hadoop-cluster-aws-ec2/.

2. "Apache Hadoop Documentation." Apache Hadoop, The Apache Software Foundation, 7 Sept. 2018, hadoop.apache.org/docs/r1.0.4/.