A Scalable Media Job Framework for an Open Source Search Engine

Pooja Mishra and Chris Pollett

Abstract—This paper explores efficient ways to implement various media-updating features like news aggregation, video conversion, and bulk email handling. All of these jobs share the property that they are periodic in nature, and they all benefit from being handled in a distributed fashion. The data for these jobs also often comes from a social or collaborative source. We isolate the class of periodic, one round map reduce jobs as a useful setting to describe and handle media updating tasks. As such tasks are simpler than general map reduce jobs, programming them in a general map reduce platform could easily become tedious. This paper presents a MediaUpdater module of the Yioop Open Source Search Engine Web Portal designed to handle such jobs via an extension of a PHP class. We describe how to implement various media-updating tasks in our system as well as experiments carried out using these implementations on an Amazon Web Services cluster.

I. Introduction

An important aspect of creating a modern search engine is the ability to display various media sources in an appropriate way. These media sources include news feeds, videos, images, group discussions, blogs, etc. Often this data coming from users of these systems sharing videos or collaborating on ideas on thse sites. Many search engines such as Google, Yahoo, and Bing have these features incorporated and customized according to the needs of their users. These tasks are often too slow to be done online within a web application, but using traditional big data algorithms such as Map Reduce might be overkill. In this paper, we describe an extension to the Open Source Search Engine Yioop that allows efficient handling of these tasks. We then look at three tasks: news aggregation, video recoding, and bulk email and describe how they can be handled by our system. We provide benchmarks comparing our approach to earlier implementations of these tasks in Yioop.

As an introductory example to the issues we are trying to address with our system, consider news aggregation. A news aggregator is a site where a user can go to see news headlines from a variety of news sources on the web. If, when a user came to a news aggregator site, an aggregator had to go out and download each news source's web page, group them by article kind, and display the result, the experience would be intolerably slow. On the other hand, traditional web crawls can take weeks or months, and this would be followed by non-trivial indexing and data processing. News is supposed to be timely, and so traditional web crawling is not immediately suited to this task. Further, the importance of various news sites, the categories of news they relate, and so on are relatively unchanging, so complicated re-ranking tasks, which may require heavy duty map-reduce jobs such as page rank, might be unnecessary. One could imagine trying to solve the problem by just having a cron job that periodically downloads web pages from a fixed list of web sites and does simple indexing on them. To some degree this solves the problem if the number of news sites is on the order of thousands or tens of thousands, but if we try aggregating news from millions of sites, feeds, etc. this quickly becomes unwieldy. What is needed is a more distributed approach.

From the above, we can identity three properties of a robust solution to the news aggregation task:

1) It runs periodically with a period measured in seconds, minutes, or hours.
2) Data processing on given news sources should be light-weight.
3) It should scale to handle as many news sites, or feeds as desired. To do this, it probably needs to be distributed.

By distributed in the above we mean the same solution or job should be run by multiple machines but using different input news sites. This entails we need some mechanism for combining the results to get the single experience the end user sees. Framed in terms of the Map Reduce model, we have a Mapper that maps different news sites to different machines for download, followed by a Reducer, which can in this case even be at query time, which combines the results. Traditional Map Reduce allows for multiple rounds of a map followed by a reduce. In our case, we only need one round. So we can solve the problem of news aggregation, with a periodically run, single-round map reduce job.

It turns out this news aggregation is not the only task that can be solved by such jobs. In this paper, we consider two other such tasks: video recoding and bulk emailing. For video recoding, we imagine that videos are being uploaded to a web site in a variety of video file formats (.asf , .flv, .ogv, .mpg, .webm, etc) and we want to convert them to a common format (.mp4) for the purpose of streaming them. Recoding can be a computationally intensive task, so one would like to spread the task among several machines to allow greater recoding throughput. Additionally, uploaded files can be of a variety of lengths, it can make sense to split longer files into files of a common length to do better load balancing. So the video recoding tasks becomes a periodic job which checks for new videos, then for each video splits the video and distributes its part to recoding machines, and finally, receives and assembles the results into whole recoded videos.

Bulk emailing, for this paper, is the task of sending out

Pooja Mishra is with Cisco Systems. 2534 Lagoon Way San Jose CA 95132 email:pomishra@cisco.com

Chris Pollett is professor in the Department of Computer Science, San Jose State University, 1 Washington Square, San Jose CA 95192 e-mail: chris@pollett.org.

notification emails (for instance, saying there is a new post), to all members subscribed to a group or following a thread on a discussion board. If the size of the group is large, then it is not practical to send out these emails during a single web request. On the other hand, queueing the emails and sending them from a single mail server has scaling limits. So we can imagine a periodic job where every so many minute we split the mails across several machines and have each machine send out their allotment of email.

Several periodic, task-oriented, variants of Map Reduce have been considered previously. Oozie [8] is a workflow engine that allows one to schedule Hadoop Map/Reduces jobs. Workflows are specified as XML documents and allow one to run a sequence of jobs contingent on earlier jobs successes and at given times. This does allow for the scheduling of periodic jobs but programming for Hadoop is non-trivial represents a barrier to using the such jobs for system maintenance task like those described above. To some degree job task implementation can be simplified by using map reduce streaming jobs, but then one ends up with a hodgepodge of some things being from the Hadoop framework, some things not. For simple map reduce jobs involving Unix system commands, one can use the system bash-reduce [2]. This tool allows a sequence of shell commands to be mapped out to several machines, or cores on the same machine, and the results of these commands can then be fed to a reducer script. bash-reduce is light-weight, and shell scripting does encourage rapid development for tasks such as those described above. Combined with a cron jobs to allow tasks to be carried out in a periodic fashion this could solve the problems we are trying to address. The drawback of course is that shell programming is somewhat limited compared to a full-fledged programming language, and if we start coding some of our tasks to be executed in a general purpose language such as C, then the system could quickly become unmaintainable. Our media updater for the Yioop Open Source search engine, we feel provides the ease of development that a system like bash reduce offers, but with the benefits of being part of single framework like an Oozie/Hadoop set-up.

Real-time, stream-oriented data processing and complex event processing systems are also closely related to our system. An example of such a system might be Yahoo's S4 system described in Neumeyer, et al [6]. Such systems are designed to consume a stream of real time data, compute intermediate values on it, and possibly emit new streams. Neumeyer, et al.'s paper described using such a system on stream of incoming words to find the top k most frequent words. We could imagine these words coming from feeds. In complex event processing, one might imagine having feeds of financial information, and the processing on the feed stream as performing tasks like buy and sell orders. Data mining of streams with map reduce has been considered previously. For example, Walmart's Muppet system [5] can perform MapUpdate tasks on incoming streams, which is very similar to the kind of one-round map reduce, we are considering above. The Muppet system paper [5] describes using the system to monitor check-ins by retailers, to detect hot Twitter topics, and to keep Twitter reputation score up to date. These systems though rely on having a relatively heavy-weight architecture already deployed in-order for their frameworks to run. We hope our system is in some sense simpler for a small to medium scale enterprise to deploy, yet garners to such an enterprise, the same kinds of abilities as these more heavy-weight systems.

As we have stated, our system is coded as a component of the Yioop search engine [10]. This is an open-source search engine created by the second author, Chris Pollett. The engine was designed to allow it do web-scale crawls with minimal dependencies on other projects, the main dependency being just PHP 5.4 or higher. It is written in a scripting language which tends to make extending and tinkering with it easier, further the source code is thoroughly documented. Yioop has been used in billion page web crawls, and has been used in numerous master's student projects at San Jose State University. Prior to the work described in this paper, the Yioop engine did have a news aggregator feature, did allow for video uploads, and did support emails in response to group posts. However, each of these features was restricted to the scale of what could be handled by a single machine, and these tasks were not abstracted out as jobs to be handled by a general media handler.

We now discuss the organization of the rest of this paper. In the next section, we give some background on Yioop software and describe our media updater framework in Yioop. This is followed by one section each for the example jobs identified above. These sections include some performance experiments. The last section then summarizes our results and draws conclusions.

II. Background

In this section, we provide details on Yioop software needed to understand its media updater system.

When deployed in a distributed setting, identical copies of the Yioop software are installed on multiple machines or virtual machines. Each machine is configured with the address of a name server machine – the machine responsible for coordinating the activities of the other machines. On a given machine, five different kinds of Yioop processes might be run: A web app used to handle web requests to the search engine, a queue server for maintaining a by-document partition of search indexes and queues of what to crawl next, a mirror process which might be used to mirror an index held by a different machine, a fetcher process for downloading and performing initial processing of crawl documents, and a media updater process for handling the kinds of jobs described in the introduction. We list all the processes for completeness, but the jobs we will write only involve the web app on the name server and the media updater.

The name server itself runs the same software as all the other machines, however, it stores in its database what machines make up the cluster and what activities are currently being run by the cluster. The web interface of the name server can be used by an administrator to configure the machine list and the active activities.

Each of the five processes except the web app mentioned above has a basic event loop. As part of the event loop, a given

process uses an HTTP request to the name server to find out what activities it should be performing. For example, a fetcher process might contact the name server to find out what is the current web crawl being performed, and what are the urls of the queue servers involved in this crawl. Given this information the fetcher might contact the web app on the machine with a running queue server to get a list of urls to download next. A queue server process might use information from the name server to know what url hash ranges it is responsible. A mirror process similar gets information from the name server to know what machine it is mirroring.

Prior to the work of the present paper, the media updater process was run only on the name server and was only used to periodically download feed urls specified in the name server's database. Our first enhancement was to allow this media updater process to run on all machines in a Yioop cluster. As part of its event loop, the media updater contacts the name server for a list of jobs that should be run. In what follows, we assume we have a media updater running on the server together with media updaters running on client machines.

Media updater jobs are specified as a subclass of a PHP MediaJob class. This class has eight main methods which may be overridden:

init() is run after the class' constructor and is intended to be what the user uses as a constructor.

checkPrerequisites() returns a boolean about whether the job should be run. It allows the job coder to check things like the system time to determine when a job should run.

nondistributedTasks() is run only on the name server when the Yioop administrator has specified that the media updater should run in non-distributed mode.

prepareTasks() is run on the name server's media updater only. It gets the data needed by the job ready before it is mapped to a client machine.

getTasks() is run by the name server web app when a client makes an HTTP request for data for the MediaJob. It is supposed to take data output by `prepareTasks()` and send the client its portion of this data.

doTasks() is run after the client media updater has received the `getTasks()` data. It then does processing on this data.

putTasks() is run by the name server web app when a client makes an HTTP request to send processed information back to the web server.

finishTasks() is run on the name server's media updater only. It applies a reduce operation, or final computations, after the data has been sent back to name server.

The media updater's event loop, after finding out a list of MediaJob's to runs, invokes each found job's `init()` method, then periodically cycles through the job list calling each job's `run()` method. This method of the base MediaJob class calls `checkPrerequisites()`, and if this returns true, calls the other methods listed above, depending on if the job is being run in a distributed or non-distributed context, and depending on whether it is being run on the name server or a client. Given this background, we now discuss our three example jobs built using this framework.

III. News Update Job

Let's consider what properties might be expected of a news handler in a search engine, taking our inspiration from two of the most-known news aggregators, Yahoo News and Google News. As early as the mid-nineties, Yahoo had a headlines section obtaining news from Reuters [12]. Over time additional news feed sources have been added, and Yahoo also began creating its own news content. In addition, different mechanisms to rank the popularity of news items, for example, via link clicks or frequency emailed, have been deployed [7]. User personalizations, such as the ability to follow selected news streams, have been provided [11]. Google News was created more recently than Yahoo. It was first released in beta in 2002 and officially in 2006 [3]. It uses automated story selection, but where humans could add sources. Both Yahoo and Google integrate news as part of search results and also allow users to search within news. Since the same or related news story may come from multiple sources, both systems also support grouping and deduplicating news stories. From examining these systems, one can come up with a list of features we would like of a web-based news aggregating system:

1) Integrates external news feeds and internally generated content.
2) Refreshes frequently so that news stays current.
3) Is searchable and allows content to appear within general search engine search results.
4) Uses a ranking mechanism that can be meshed with the ranking mechanism of the standard search results.

Even before the work of the current paper, Yioop had facilities to accomplish the above. We outline how these facilities work in order to understand what the distributed news updater job needs to do. In the administrative web panels of Yioop, administrative users can add and delete RSS, Atom, JSON, or HTML scrape pages. Url's and relevant XPath's entered for these pages are then stored in the Yioop database. This allows an administrator to manage external news feeds. The Yioop search engine comes with the ability for user's to create groups with varying levels of editing privileges. Each group has associated with it a feed and this feed can be output in RSS if desired. Using this mechanism, popular feeds can be added to the list of news feed sources. User groups also have a mechanism for voting up or down particular feed items. Prior to this project, the media updater on the name server would once an hour download the search sources that the administrator had specified. It would then compare feed items both based on hashes of content as well as GUIDs to determine which items had not been previously stored and then add them to a table in the Yioop database. Items greater than a week in age would be deleted. Finally, an inverted index of the rows of the feed item table would be created to facilitate search.

Since early in the development of the Yioop project, Yioop has supported crawling and storing web crawl indexes across several machines. Indexes are partitioned across machines using document partitioning: A given machine in the Yioop cluster will store the inverted index for all documents in the web crawl whose URLs were in a particular hash range. When a query is processed from the web interface, the same query is then run on each partition making up the index, and the results are combined. Originally, news results were stored only on the name server. When a query was processed, the query was processed against the name server news results and combined with any index results on the name server, this in turn was combined with query results from other machines in the cluster. A special "meta word", `media:news`, is added to all news documents, so that if a user wants to search just news results this meta word can be added to the query to perform the desired restriction. Since each Yioop machine in a cluster has the same code base, and does a check for news to combine with existing query results, to make the original news updater distributed, one could imagine by hand evenly allocating the feed source urls between the machines in the cluster, and then running news media updaters on each machine using only its allocated sources. The distributed query mechanism from before could then be used to serve news in search results.

To implement this in an automated fashion using the new Yioop media job framework, we make a subclass NewUpdateJob of MediaJob. The administrator is responsible for adding feeds only to the name server and we assume that this has been done prior to running this job. Then when a MediaUpdater runs its NewUpdateJob, its `run()` method detects if it is being run on the name server or client. In the former case, its behavior would look like:

NewUpdateJob Behavior on a Name Server Web App.

```
getTasks($machine_id) :
    Uses client $machine_id to get those url'
        s of feeds whose hash maps to $machine
        _id.
    return information for these feeds to
        requesting client
```

Notice NewUpdateJob does not override `prepareTasks()`, `putTasks()`, or `finishTasks()`, and these would be inherited as empty methods from the base class. On a client, NewUpdateJob's behavior would like:

NewUpdateJob Behavior on a Client.

```
checkPrerequisite():
    Checks if it has been more than an hour
        since news was updated. This could be
        made less if desired.

doTasks($tasks):
    $tasks is an array of feed information
        gotten by an HTTP request to the name
        server earlier in the run() method. On
         the name server, getTasks() would
        have been invoked to fulfill this
        request.
    foreach ($tasks as $feed):
        Download the page for $feed's url.
        For each feed item in downloaded page
            , check if it's new and not a
            duplicate. If so, add it to the
            local database table for feed
            items.
    Delete expired feed items from feed table
    Rebuild index shard for feed items.
```

If the Yioop cluster administrator decides to configure their site so as not to use a distributed media updater, and only runs the media updater on the name server, then the `nondistributedTasks()` method of NewsUpdateJob is used. The `run()` method in this case, first calls `checkPrerequisite()` to determine if it is time to time download feeds again, and if so, calls `nondistributedTasks()` . This in turn obtains all the feed urls from the name server's database and calls `doTask($tasks)` with this information.

As we indicated in the introduction of this paper, the two listings above can be viewed as carrying out the "mapping portion" of a map reduce algorithm. The pre-existing query mechanism which runs the same query on each machine then merges the results could be viewed as playing something-like the role of a query time "reduce operation".

A. News Update Job Performance Testing

Performance experiments for our news update job conducted on a cluster on AWS machines, each with the following specifications: 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, running Ubuntu Linux. A Yioop instance was installed on one of these machines and then cloned using the API option in AWS. After distributing the code, news sources were added to the name server. Timing measurements were then performed using different numbers of machines and different numbers of news sources. The results of these measurement are plotted in Fig. 1. As one can see, as the number of news sources increases, the time required to build the index shard steadily increases. However, the time can be reduced by adding additional machines to the cluster. Two machines in the cluster reduce the time almost exactly by 50 percent, which demonstrates that hashing sources' urls to machines is evenly distributing the work needed to index all the feeds.

IV. Video Convert Job

The NewsUpdateJob example did not as part of the job make use of a reduce operation. We next consider the job of converting videos to a particular format to illustrate a MediaJob that does this.

Many search engines have the capability to upload videos and view them later. One popular example is Google subsidiary, YouTube. Wikipedia also allows one to associate public domain videos with wiki pages. Calculation of relevance of a video to a search query can be done by looking at graphs such as those based off co-viewed videos of users and may involve a map reduce algorithm not unlike page rank [1]. In the Yioop search engine, videos can be uploaded to group wiki

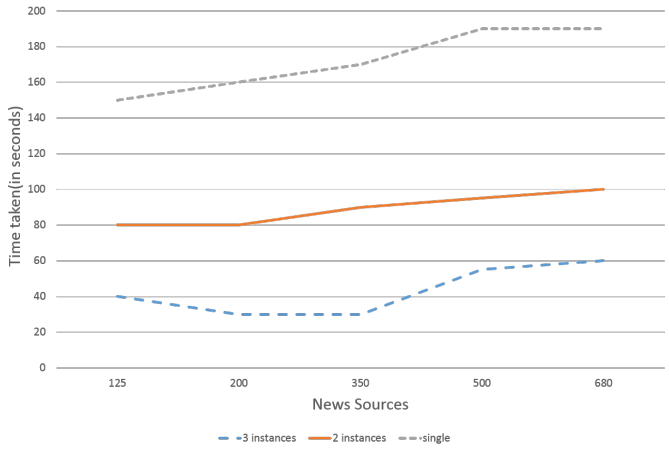

Fig. 1: NewsUpdateJob Performance

pages and to posts in a group's feed. There is also built-in public group wiki which can be used to configure the overall look and feel of a Yioop installation. The upload feature can be used to display videos off arbitrary Yioop pages provided the uploader has permissions on that page. Further wiki pages in Yioop can be of different types one of which is a gallery type suitable for displaying photos or videos. Videos in Yioop are streamed using HTTP pseudo-streaming [9]. To ensure that videos are streamable across a wide spectrum of modern browsers, the media job we are going to describe in this section was created to convert uploaded videos to mp4.

Prior to the work of this paper, Yioop could be configured to convert videos on the name server. After conversion it would ensure that converted videos got moved to the correct folder for the wiki page in question. This name server only conversion was computationally intensive for the name server, and tended to slow down other important processes such as query processing. So on the main live test site, yioop.com, it was turned off. To scale and make feasible video conversion, we wrote a subclass VideoConvertJob of MediaJob. Our job makes use of the free software project known as `FFmpeg` [4] to handle video manipulation.

When a file is uploaded to a wiki page as page resource, its media type is checked to see if it is a video file which should be converted. If it is, and we are running is a distributed media updater mode, then after it is moved to that wiki page's resource folder, a subfolder of the convert directory is created, and a text file with file information and another indicating that a video needs to be split for conversion are written. All of this processing is relatively fast, and to this point, processing is done by the web app on a file upload. The remainder of the processing is done by the VideoConvertJob. This job as written has a `checkPrerequisites()` which only runs the job when Yioop is in a distributed media mode setting, and for this reason, the `nondistributedTasks()` is the default do nothing method of the base MediaJob class. For this job, there are two groups of methods on the name server: Those that run in the media updater and those that run in the web app. The media updater name server methods are as follows:

Name Server MediaUpdater VideoUpdateJob Methods.

```
prepareTasks():
    for each video convert subfolder:
        if a split file exists:
            Read the file info file to get
                information on the video to be
                converted
            Split the video into 5 minute
                segments using FFmpeg.
            Output segments to video convert
                folder.
            Remove split file.
            Write a convert count file with
                the number of files to convert
finishTasks():
    for each video convert subfolder:
        if the number in convert count file
            equals the number of mp4 segments
            uploaded to the video converted
            subfolder:
            Move file info file and count
                file to video converted
                folder.
            Delete video convert subfolder.
    for each video converted subfolder:
        if a convert count file exists but
            an assemble file does not:
            Write an assemble file to
                instruct FFmpeg how to
                concatenate video segments to
                build converted file
    for each video converted subfolder
        if an assemble file exists:
            Use assemble file with ffmpeg to
                concatenate converted video
                segments to wiki resource
                folder
            Delete video converted subfolder
                assemble
            Create a thumbnail for converted
                video with FFmpeg
```

After `prepareTasks()` has run on a video convert folder, its segments will be ready to be sent to the client for conversion. The process of segmenting a file using ffmpeg is much faster and lightweight than the process of converting from one video format to another, which will be seen in our experimental results in the next section. The method finishTasks() above handles videos after converted segments have been sent back from clients to the name server. It can be viewed as the "reduce step" of the Map Reduce paradigm. It checks if all the segments of the video file have been uploaded, and if so, concatenates them to make a final converted video file back in the wiki page's resource folder. The VideoConvertJob methods that run in the name server's web app are responsible for getting video segments to give to client's for conversion, and for receiving converted segments and moving them to the correct folder. They are:

Name Server Web App VideoUpdateJob Methods.

```
getTasks($machine_id, $data = null):
    for each convert subfolder:
        for each video segment to convert as
            $file_path:
```

```
                if not exists timestamp file for
                    segment or timestamp is
                   expired:
                    break out of both for each
                        loops
        Write a new timestamp file for $file_path
        return associative array with the file
            name of $file_path, its subfolder name
            , and file's contents.

putTasks($machine_id, $data):
    Here $data contains an associative array
        with the converted video segment file
        name, the convert subfolder, and
        converted segment data
    Compute converted subfolder name from
        convert subfolder name
    Create converted subfolder if it doesn't
        exists
    if the segment name does not exist in the
         converted subfolder:
        Make file in converted subfolder with
             segment file name and converted
            segment data
        Delete the original, unconverted
            segment from the convert subfolder
```

Notice, unlike the NewsUpdateJob, neither `getTasks` nor `putTasks` makes use of its $machine_id argument. The name server methods, as we have seen, do all the bookkeeping, which, although somewhat more intricate, are less computationally expensive than the actual video conversion which occurs on the client. This VideoUpdateJob methods on the client to do the conversion are as follows:

Client MediaUpdater VideoUpdateJob Methods.

```
checkPrerequisite():
     if in distributed mode:
          return true
     return false
doTasks($tasks):
     Here $tasks contains an associative
         array with the file name, convert
         subfolder name, and data
     for a video segment to convert. doTasks
         () is called from the base classes
         MediaJob's run()
     method after it has made a request to
         the Name Server to execute getTasks()
         .
     Remove any previously existing convert
         subfolder with the same name
     Make a new convert subfolder
     Write $task's data to a file with $task's
          file name in the folder just created
         .
     Convert just written video segment file
         to mp4 using FFmpeg.
     Create an associative array with the
         converted file name, folder name, and
          converted data
     return array, so run() method can send
         it to the name server
```

1) Video Updater Performance Testing: Fig. 2 shows the results of our performance tests for the VideoUpdateJob. To test the VideoUpdateJob, again an AWS cluster was configured and the time it took to convert the videos was measured, varying the number of machines and the length of the video to convert. The one machine case measures the original non-distributed code. As one would expect, on any number of machines, a longer video takes longer to convert. We also see a similar improvement in speed going from one to two to three machines, that we did in the news update job situation. For a fifty minute video, on a single machine it takes about 300 seconds to convert a video, 170 seconds in the two machine case, and 120 seconds in the three machine case. So both the two and three machine cases and about the same amount, 20 seconds, above the ideal speed-up of a factor or 2 in the first case, or 3 in the second. This factor can be attributed to the bookkeeping, segmenting times, and network communication times. These would be roughly the same for the two and three machine case. Although it may look from the graph that there is little advantage of our distributed set-up for shorter videos in going from two to three machines, one has to remember the above graph is for the conversion of a single file, averaged several times. In a typical scenario, one would have several outstanding files to be converted, and here having additional machines would help.

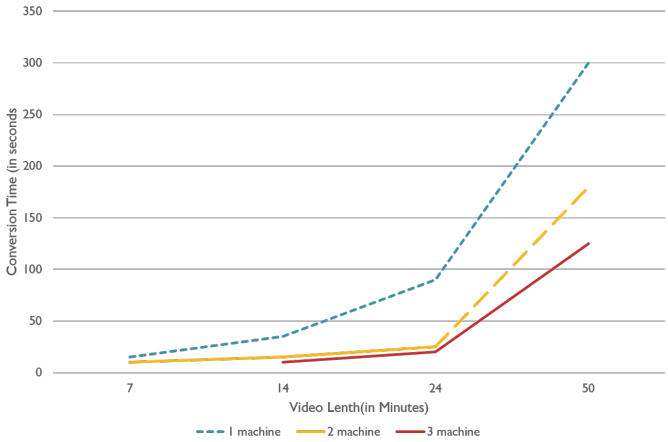

Fig. 2: VideoUpdateJob Performance

V. Bulk Email Job

Our last example is perhaps the simplest of our three examples. Prior to our work there were several situations in which Yioop might need to send out emails: new user registration, password recovery, notifications of new posts in discussion groups, and notification of membership requests to groups. To send a single email from the web app in response to a user's request would in general not be very time consuming, but to notify all members of a large discussion group of a new post might an impractically long time. Yioop sends emails using a MailServer class which either directly uses PHP's built-in mail() function or by using its own simple implementation of SMTP.

The first step to improving the web app only approach to sending email was to add a Server Settings activity that allows a Yioop site administrator to choose between web app-based

emails, or media updater based emails. In the first case, the prior email system of Yioop is used. To handle the second situation, the MailServer class was modified so that it could write emails into text files in a mail directory. A given text file in this folder has its creation timestamp as its name and is appended to for a five minute interval with all newly received emails, after which a new text file is started. We then created the BulkEmailJob, a subclass of MediaJob, to handle sending the emails from the media updater. If the media mode is non-distributed, then when this job is run, periodically, the nondistributedTasks() method is called. It looks in the mail directory for a file older than five minutes, read it, sends out the emails it contains, and deletes it. In the distributed setting, on the name server we have only the following method overridden:

Name Server Web App BulkEmailJob Methods.

```
getTasks($machine_id) :
    $sendable_file = false
    for each $email_file in mail directory:
        if older than 5 minutes:
            $sendable_file = $email_file
            break
    if $sendable_file is false:
        return false
    Create an associative array with name $
       sendable_file 's file name and with
       data its contents
    return array
```

The $sendable_file's file name is used only for log messages that the media updater outputs. To handle actually sendings emails contained in an email file on the client, the following two overridden methods come into play:

Client BulkEmailJob Methods

```
checkPrerequisite():
     if in distributed mode or set to use
         mail server in media updater:
         return true
     else
         return false
doTasks($tasks):
    $tasks is an associative array with the
       name of the to-process email file and
       its data contents
    Split data contents into an array of
       emails to send
    foreach email to send:
         send email
    return false to indicate no putTasks
```

VI. Conclusion

We have presented the media updater framework for the Yioop open-source search engine. A design goal of this framework was to make it easier for people to code periodic search engine, wiki, or web-site related jobs developed using Yioop, with the intention that the execution of these jobs will scale to larger deployment settings. Jobs in our framework run periodically. The NewsUpdaterJob, VideoUpdateJob, and BulkEmailJob example jobs we have explained, show that it is relatively easy to write lightweight, distributed jobs in our framework. The experiments we conducted with these jobs illustrate the advantage of running these kind of periodic jobs in a distributed setting. As the Yioop project itself has minimal dependencies on other projects and is written in the popular scripting language PHP, it has been relatively easy for many students at San Jose State to get up to development speed on this project. It seems promising that the media job framework will facilitate future improvements to Yioop such as the ability to periodically process movie and weather feeds, perform traffic analytics, and to perform supplementary crawls for the main crawl used to serve search results.

References

[1] S.Baluja, R. Seth, D. Sivakumar, Y. Jing, J.Yagnik, S. Kumar, D. Ravichandran, and M. Aly. *Video Suggestion and Discovery for YouTube: Taking Random Walks Through the View Graph*. Proceeding of WWW 2008.

[2] Bash Reduce GitHub Page. Retrieved on Sep. 11, 2015 from https://github.com/erikfrey/bashreduce.

[3] Krishna Bharat. *And now, News*. The Official Google Blog. Jan. 23, 2006.

[4] FFmpeg. Retrieved Dec 4., 2015 from http://ffmpeg.org/.

[5] W.Lam, L.Liu, S.Prasad, A.Rajaraman, Z.Vacheri, and A.Doan. *Muppet: Mapreduce-style processing of fast data*. Proceedings of the VLDB Endowment (PVLDB), 5:18141825, 2012.

[6] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. *S4: Distributed Stream Computing Platform.* In Data Mining Workshops, International Conference. IEEE Computer Society. pp 170–177. 2010.

[7] P. O'Connell. *New Economy; Yahoo charts the spread of the news by e-mail, and what it finds out is itself becoming news*. New York Times. Jan. 29, 2001. http://www.nytimes.com/2001/01/29/business/\qquadnew-economy-yahoo-charts-spread-e-mail-what-it-finds-\qquaditself-becoming.html

[8] Oozie 4.2.0 Documentation. Retrieved on Sep. 11, 2015, from, http://oozie.apache.org/docs/4.2.0.

[9] Wikipedia Progressive Download page. Retrieved Oct. 8, 2015 from https://en.wikipedia.org/wiki/Progressive_download.

[10] Yioop Documentation from Seekquarry. Retrieved on Sep. 11, 2015 from http://www.seekquarry.com/p/Documentation.

[11] A. Silberstein , J. Terrace , B. F. Cooper , R. Ramakrishnan. Feeding Frenzy: Selectively Materializing Users Event Feeds . In SIGMOD 2010.

[12] Yahoo! Headline. Nov. 28, 1996. Internet Archive. https://web.archive.org/web/19961128074525/http://www8.yahoo.com/headlines/