Distributed Archive Crawls in Yioop!
CS280 Final Report

Shawn Tice

May 16, 2012

The project goal was to make managing Yioop archive crawls[1] easier by relaxing the setup requirements while still utilizing Yioop's distributed architecture to perform page processing and indexing. While a normal distributed web crawl might often make use of several machines to fetch many pages from many servers simultaneously, a distributed archive crawl requires fetching a collection of pages from one or more (but rarely more than a hundred or so) large files, all initially hosted in one location. Previously Yioop accomplished archive crawls by requiring the user to manually place archive bundle files on each of the available fetcher machines in a specific directory structure, at which point the crawl could proceed much like a web re-crawl. My modifications allow the user to place an archive bundle on the name server in a specific location, then initiate a new archive crawl which will take care of splitting up the archive bundle a few hundred pages at a time and distributing the processing of those pages across several fetchers and queue servers.

In this brief report I will outline the changes I made to Yioop's architecture and give an example of running an archive crawl under the new system, discuss the performance issues I ran into, provide some rationale for choosing values for the tuning parameters, and conclude with recommendations for future work. The new system is currently functional, but has only been lightly tested on sample archives, and will probably require more genuine use and small modifications in order to get it running smoothly in general.

The New Archive Crawl Architecture

Where there used to be just two types of crawls—web crawls and archive crawls—there is now a distinction between a web archive "re-crawl" and a genuine archive crawl. The handling of the former is the same as it always was, while the handling of the latter is completely new. An archive crawl now begins with a single directory nested in the Yioop working directory of the name server.

[1] Except where stated otherwise, by "archive crawl" I mean indexing an archive bundle *not* in Yioop's native web archive format; I will refer to indexing a web archive as a "web re-crawl".

There is one directory per archive bundle to be crawled, and each directory contains a user-created configuration file and a collection of compressed files in a format specific to the archive type[2]. For example, the directory structure for a MediaWiki archive bundle might look like the following:

```
Yioop/workdir/cache/archives/
      mediawiki-en/
            arc_description.ini
            enwiki.xml-p001p009.bz2
            enwiki.xml-p010p019.bz2
            ⋮
      another_archive_bundle/
      another_archive_bundle/
```

The name of each archive bundle directory is unimportant; the configuration file specifies a name and type for the bundle, and the type is used to decide which archive bundle iterator will be used to extract pages from compressed archive files. Presently the only valid types are:

- `ArcArchiveBundle`
- `MediaWikiArchiveBundle`
- `OdpRdfArchiveBundle`

I modified the crawl model to search through the `archives` directory to build a list of archive bundles available for crawling, and to set the directory and archive type for each bundle in the initial crawl parameters. When the fetcher sees that there is an archive directory and type associated with a new crawl time it knows that it is dealing with a new-style archive crawl rather than a web re-crawl, and so instead of going to the scheduler to get the crawl parameters and a fetch schedule it requests this information from the name server. Each time a fetcher requests a "schedule" from the name server the name server responds with the current crawl parameters and a batch of some number of pages extracted from the appropriate archive bundle. The fetcher process these pages just as though it had downloaded them from remote servers, extracting links, tokens, and other information, then passes this information on to the queue servers in the usual round-robin fashion. After the page data gets from the name server to a fetcher the crawl process looks much like a normal web crawl. Exactly how many pages the name server should send at once may depend on a number of considerations, and we will talk about these in the section on performance.

There is a difficulty in getting the page data from the name server to the fetcher. The controller which doles out page data to fetcher process only lasts long enough to handle a single fetcher request, and is then effectively purged

[2]The three archive types are MediaWiki, ODP RDF, and ARC.

from memory. Consequently, the controller must instantiate a new archive iterator for each request, and that iterator must pick up where it left off on the previous request. Furthermore, if two separate fetchers make a request for page data at the same time (which is not unlikely at all, since it takes a while to extract page data from archive bundles), two or more separate controller processes must coordinate so that no two fetchers receive the same batch of pages. The archive iterators were not originally created to deal with these complications, and so I ended up spending a lot of my implementation time patching the iterators (and especially the MediaWiki iterator) so that they could efficiently seek into zipped archive files and pick up where they had left off.

To deal with coordinating concurrent requests for page data, I implemented an exclusive file lock with blocking and a timeout. When a fetcher makes a request for page data the controller that handles the request first tries to acquire an exclusive lock in order to access the bundle archive; if another controller is already extracting pages then the second controller will block until the lock is released, and in this manner controllers queue up waiting for their respective turns. If a controller waits too long to acquire a lock then the corresponding fetcher process will give up on the request and try again in five seconds; when the controller eventually gets the lock it will realize that too much time has passed and immediately release it. As with the number of pages to send in each batch, the decision as to how much time is "too much" will depend on a number of considerations which we will discuss in the performance section.

An Example

Suppose we want to index a collection of Wikipedia pages which we have downloaded from the WikiMedia "dumps" website as several `*.bz2` files. We will use the directory structure provided previously, and the `arc_description.ini` file will look something like:

```
arc_type = 'MediaWikiArchiveBundle';
description = 'Wikipedia Dump 12 May 2012';
```

Once we have created the directory and added the configuration file and bzip files, we open a web browser and navigate to the "Options" page of the "Manage Crawls" task in the Yioop web interface hosted on the same machine where we placed the archive bundle (the name server). We select "ARCFILE::Wikipedia Dump 12 May 2012" from the drop-down, click on "Save Options", then return to the main "Manage Crawls" task, where we give the crawl a name and click "Start New Crawl" to kick things off.

On their next crawl time checks the fetchers should register a new archive crawl and start requesting page data from the name server. Once all of the archive pages have been exhausted the fetchers should just get "No Data" messages from the name server, and after the crawl has been manually stopped by clicking the "Stop Crawl" button they will return to periodically checking with the queue servers for schedule updates.

Performance Considerations

The two main tuning parameters introduced by the new archive crawling system are the number of pages to fetch from an archive bundle in one go (we will call this the batch size), and how long (or whether) a fetch controller should wait to acquire a lock for the archive bundle before giving up and failing the request (we will call this the lock timeout). These parameters are set by the constants `ARCHIVE_BATCH_SIZE` and `ARCHIVE_LOCK_TIMEOUT` respectively. We will consider the batch size first, then consider the lock timeout, which depends on the batch size.

There are two primary factors which determine the appropriate batch size: the number of fetchers and the overhead required to seek into an archive file and parse out page data. Extracting page data is relatively slow, and even once an archive iterator has been tuned for performance it might take around one second to extract a hundred pages, plus some extra overhead that increases linearly (slowly) with the offset into the archive file at which the pages reside. Thus even for a small number of fetchers it makes no sense to extract a huge number of pages at once because the requests for page data will start timing out frequently while the fetch controllers wait for their turns.

On the other hand, if the batch sizes are made too small then the overhead due to setting up and tearing down connections, instantiating archive iterators, and seeking into archive files will outweigh the benefit of having extra fetchers. Furthermore, because the amount of work a fetcher has to do per batch increases superlinearly with the batch size while the work the name server and queue servers must do increases linearly, small batch sizes tend to overwhelm the more scarce system resources and underutilize the fetchers.

So we want to strike a balance that minimizes overhead costs by making batch sizes as large as possible without starving the fetchers by causing their requests to time out. I did not have a lot of time to test various settings, but I found that a batch size of a hundred was much too small for even two fetchers, and resulted in the name server and queue server being swamped with work such that the fetchers spent more time waiting for data to process than they did processing it. Five hundred turned out to be a much more reasonable batch size; it kept the fetchers busy processing for around thirty seconds, while it only took the name server and queue server a few seconds to do their per-request work. Note that there is a relatively low upper bound on the number of pages that can be extracted from an archive per second, and so even for an ideal batch size there are a limited number of fetchers and queue servers that a given archive crawl can effectively utilize.

The lock timeout depends on the batch size because it must be long enough to provide a chance for a request to succeed even if there are requests in front of it, but not so long that it is possible to acquire a lock before a fetcher gives up on the request, then have the request time out after the iterator has begun extracting pages. For example, if the batch size were so large that it took nearly thirty seconds to extract a full batch of pages and the request timeout were set at thirty seconds, then if the lock timeout were a second or more it would be

possible for the fetcher to give up on the request after the controller had acquired the lock and started extracting pages, which would result in lost page data. On the other hand, if the lock timeout were a tiny fraction of a second then it would be common for requests to fail due to some other fetcher getting there first, and fetchers would frequently waste time sleeping five seconds between each failed request. Thus the lock timeout should be as long as possible while still leaving sufficient time for the worst-case batch of pages to be extracted without the request timing out.

Conclusion and Future Work

Distributed archive crawls should now be working for each of the three main kinds of archive bundles (ARC, MediaWiki, and ODP RDF). Furthermore, web re-crawls and live web crawls should continue to work in the same manner that they always have. This functionality comprises the majority of my work for this semester project.

A large part of my efforsts ended up being devoted to figuring out how to efficiently seek into bzip and gzip files in order to make it feasible to instantiate a new archive iterator to handle each request for a batch of pages. I was able to significantly improve the performance of the archive iterators, but consequently there are a few important aspects of the system that I did not get to. The two most critical are probably flushing processed page data to disk between dumps from fetchers to the appropriate queue servers and keeping a copy of raw page data with the fetchers until it has been processed so that it would be possible to gracefully recover from an error without losing data.

Additionally, the batch size and lock timeout parameters may be set so as to roughly accommodate each of the different archive types, but will have different optimum settings depending on the specific type. Each archive iterator has its own unique quirks which will determine how long it takes to fetch N pages, and this can have a large impact on the optimum parameters. It would be ideal to have rough defaults, but to be able to specify per-archive settings in the `arc_description.ini` file, which would then be propagated to the name server.