Managing duplicates across sequential crawls

Kristinn Sigurðsson

National and University Library of Iceland
Arngrímsgötu 3
107 Reykjavík
Iceland
kristsi@bok.hi.is

Abstract. Dealing with documents that remain unchanged between harvesting rounds is an important topic for many organizations archiving the World Wide Web. This paper discusses some of the key problems in dealing with this and then outlines a simple, yet effective way of managing at least a part of it. This is done in form of an add-on module for the popular web crawler Heritrix. The paper contains the results of crawls using this new software. Finally, there is a discussion on the limitations and some of the future work needed to improve duplicate handling.

1 Background

Archival quality harvesting of, at least parts of, the World Wide Web is quite possible today. The scope of the Web is immense but as long as we either have sufficient resources or are willing to reduce our scope to fit within our resources, it is possible to harvest it quite thoroughly. It has been known, however, for a long time that this is not enough. The Web changes every day, indeed every minute. It is this transient nature that has spurred the efforts to save it. We know that what is there today may be gone tomorrow. But what if it isn't?

Since the Web changes, we must crawl it repeatedly. This poses a new challenge, what do we do about the documents that remained unchanged? Does it even matter? Can't we just keep the duplicates?

The answer will vary from one organization to the next. For those doing deep and thorough crawls of reasonably large scopes at frequent intervals, the amount of duplicate data can be immense.

Typically, people will cite one or more of the following four reasons as the main impetus behind any attempt or desire to reduce duplicates when crawling:

1) Reduce load on web servers. Improve politeness.
2) Reduce bandwidth usage.
3) Reduce storage costs.
4) Improve the quality of the collection or its presentation. Implying that a series of duplicate 'entries' make it harder to analyze actual changes.

Previously, I've looked at the possibility of doing *incremental* crawling [1]. Incremental crawling relies on only saving changed and new documents. It requires the crawler to be aware of when changes occur in online resources. In theory this approach seems very attractive. It even allows the software to adjust its crawling to match observed change behavior. Sadly, in practice things are less attractive.

One of the primary problems in dealing with duplicates is *detecting* them!

Of course, it is simple to do a bit by bit comparison, but that is frequently irrelevant. HTML pages in particular will frequently have sections that change either randomly or much more frequently than the actual *content* on the page. Detecting a change in the actual content has proven difficult. It is possible to address each instance, of course, but that approach does not scale as it requires too much human intervention. There is no general solution that is guaranteed to never miss a valid change.

Without the advantage of adaptive scheduling, incremental crawl strategies offer little above the more traditional *snapshot* strategies where each crawl has no memory of the past. Indeed, maintaining the state of an incremental crawl becomes more complex and ultimately limits its scalability and performance. The performance, in particular, being an aspect that adaptive scheduling was supposed to improve on.

We are therefore left with our original problem for any but the most modest crawls. How can we manage duplicates using a snapshot crawling strategy? Or, more to the point, what can we do to improve Heritrix's [2] snapshot crawls in this regard without compromising the crawlers performance?

2 The trouble with HTTP headers

Since various meta-data is provided by the HTTP headers when using the protocol it is very attractive to take advantage of it to eliminate duplicates before you even download the body of the document. In particular the datestamp (*last-modified*) field, which is supposed to contain the time when the document was last changed, and the Etag which is supposed to contain an entity tag which "*may* be used for comparison with other entities from the same resource" [3] are considered.

In fact it has been shown that using them is quite reliable as far as correctly indicating changes [4]. This header information is however less useful in accurately indicating that a documents has not changed leading to unnecessary downloads. Despite this, it seems prudent to examine the use header information to avoid what downloads we can.

Unfortunately there is a problem with this, even assuming that the values are always correct. If we use it to avoid duplicates, we run the risk of missing a lot of changed content because an interim page (between the seeds and the changed material) has not changed. This is common where there are 'index' pages and also where redirection occurs. For example http://mbl.is would be a reasonable seed for this large news web. However, the page it refers to never changes, instead it will forward the user to http://mbl.is/mm/frettir/ for reasons of its own. While this particular example can be overcome by using another seed, it nevertheless illustrates the overall problem. Checking each seed may be impossible, and certainly checking each URL is out of the question.

This wouldn't be a problem in an incremental crawl where the crawler already knows all the links that the duplicate pages had contained. However, in a snapshot crawl we are not as lucky. This does not mean, though, that HTTP headers are of no use. As we'll see later, they can be used effectively, but only if you target the right documents.

3 Being pragmatic – picking low hanging fruit

Not all documents are created equal. It has been pointed out that some types of documents are more likely to remain unchanged than others [5]. This is fairly logical. There are a multitude of content management systems that create dynamic HTML pages, but very few have dynamic pictures and it is extremely rare for other documents to be generated on demand. That is not to say that they will never change, only that they are less likely to do so.

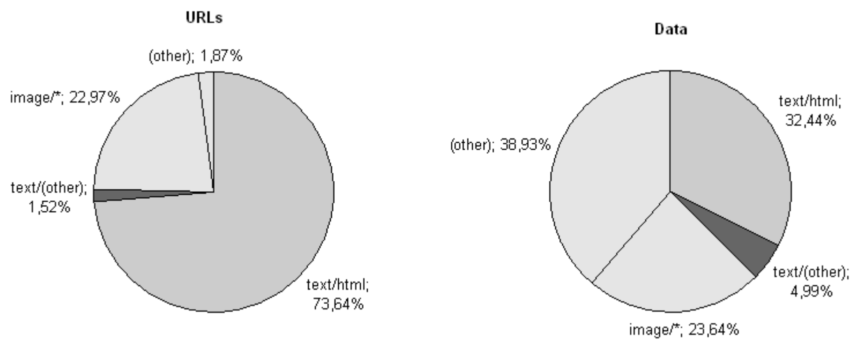


Fig. 1. A comparison of the amount of different mime-types collected during a snapshot crawl of the .is top level domain. On the left the number of URLs is counted while the amount of data is used on the right.

As figure 1 illustrates, HTML documents account for up to 74% of the documents crawled in a large harvest cycle. However, when we consider the amount of data (uncompressed) HTML files only account for a little over 32%. Meanwhile, documents other than the various pure text and image formats represent almost 39% of the data while being almost insignificant in the total number of documents.

It seems clear that by filtering out unchanged 'non-text' documents we can, at least potentially, achieve a considerable reduction in the amount of redundant data we must store. In fact, as we'll get into in greater detail later, we found that it is possible to reduce the amount of non-text data anywhere from 50% to 90% depending on how far apart the crawls where performed.

Additionally, if downloaded content is stored in compressed files, it is worth noting that while HTML and other plain text files compress extremely well, other document formats (such as images, video, audio etc.) are typically already compressed as a result of the file format itself. Consequentially they do not yield to further compression making them an even larger percentage of the stored data.

Also of interest is the fact that these non-text documents do not (usually) contain extractable links. That means that if we can determine an acceptable way to filter them out based on the HTTP headers, we can safely discard them without the penalty discussed in the previous chapter.

Finally, it is very rare for non-text based documents to have sections that change independently of the documents *content*. Even automatically generated images, such as graphs, only change when the content changes. This is in stark contrast with the behavior of HTML documents in particular. It means that strict comparison, for example, of a secure hash of the document such as SHA-1 [6], can be trusted to accurately reflect whether or not a change has occurred.

4 The DeDuplicator

With the basic strategy now in hand the problem becomes how to look up the past values of individual URLs. Initial ideas were influenced by the incremental strategies and would have entailed some form of data store that was maintained at crawl time, updated as new URLs were crawled and re-used for subsequent crawls.

This concept had the potential to seriously impact the crawler's performance. Currently, one of the main difficulties in Heritrix is maintaining its database of crawls discovered, processed etc. Compounding this issue by adding another data store or augmenting the existing database did not seem like a good idea. Fortunately there was another solution, unique to snapshot crawls.

Since snapshot crawls have clearly defined cycles between revisits, it became obvious that the best way to minimize the impact on the crawler's performance would be to create an index in between snapshots.

With this in mind work began on the DeDuplicator add-on to Heritrix that would implement this strategy.

The new software consists of three primary components.

1) A command line tool to build indexes, called DigestIndexer.
2) A Heritrix module that discards documents that have not changed, called the DeDuplicator processor.
3) A Heritrix module that aborts download of documents if the HTTP header indicates that it has not changed, called the DeDupFetchHTTP processor..

We will cover the main aspects of each, but first we need to consider what data the index must contain.

4.1 The index

The DeDuplicator uses the Lucene information retrieval software [7] to handle the actual index. There were many reasons for choosing Lucene. It is open source and Java based allowing easy integration and native access to the index. Berkley DB [8], already used extensively by Heritrix, would also satisfy those criteria. However, where Lucene won out was the sheer *flexibility* of its indexes. Each entry can contain any arbitrary number of fields and it is extremely easy to change what fields are indexed and which are not. As we'll see this flexibility allows us to fine tune the index to contain exactly the elements we need, improving performance.

In Lucene, each 'document' (a set of related data) consists of a number of named fields. Each field can be either indexed or not and not all documents need to contain the same fields. Any *indexed* fields can be searched and the hits returned from searches will allow access to the unindexed fields.

Following is a list of data the index can contain. Most are optional and it depends on how the index is to be used which fields must be included and which should be indexed.

- **URL** – Mandatory. Clearly we need to be able to associate downloaded documents with indexed ones based on the URL. Typically this field would be indexed, but it is also possible to index by *digest* when filtering documents *after* download.
- **Digest** – Mandatory. A unique hash of the documents contents. Heritrix automatically calculates such a hash using a SHA-1 algorithm [6]. It seemed best to utilize this already present feature. Heritrix refers to this hash as a document's 'content digest' and we'll do the same from here on. When filtering duplicates after download the digest is used along with the URL to check if it is a duplicate. This field can be indexed.
- **Timestamp** – Optional. The time when the document was last harvested. This value may be a little malleable based on where the data comes from.
- **Etag** – Optional. Etag support is somewhat limited at the moment.
- **Normalized URL** – Optional. Often URLs will vary from one crawl to the next in insignificant ways. Such as different parameters, 'www.' prefixes etc. When comparing content digests it is therefore possible to compare normalized URLs (with such variances removed) instead of the original URLs. Will be indexed if URL is indexed.
- **Origin** – Optional. A bit of meta-data on where the original capture of a document can be found. The exact use of this field is at the operator's discretion. We will return to this field later when we talk about the WARC file format [9].

4.2 Building an index – DigestIndexer

The index could be built by iterating over the ARC files, extracting the necessary information from the meta-data associated with each entry in the ARC. While perfectly feasible, an easier way of doing this is to process the crawl.log file that Heritrix generates during a crawl.

The crawl.log file contains all the information we need. The URL, a timestamp from when it was crawled and the content digest. The normalized URL needs to be calculated regardless of the source. The crawl.log also contains information about duplicates encountered which can thus be carried forward to newer indexes.

The file does not, however, contain the Etag so it would be necessary to build the index from the ARC files if we wished to use that or modify Heritrix to include the

Etag in the crawl.log file. However, the file's much more compact nature means that indexing is significantly quicker.

The software presently allows the operator considerable flexibility in how the index is constructed. It is possible to specify exactly what data is included and what form of indexing is performed. That is, is the URL indexed or the content digest (or both)?

The software is essentially two components. The index writer and a crawl data iterator that reads the crawl.log, ARC files or other source of information used to build the index. The crawl data iterator can be easily replaced enabling you to build indexes based on any data set. As noted above the iterator provided works with Heritrix crawl.log files and, consequently, does not support Etags.

The index time is a factor of the amount of data and how it should be indexed. Typically it takes only a few minutes to index one million URLs. Tens of millions of URLs may take hours. It is easy to adjust exactly what URLs are included based on their mimetypes.

4.3 Filtering after download – DeDuplicator

When crawling, or processing, a given URL Heritrix applies a series of *processors* [10]. This series is entirely at the operator's discretion although for any meaningful crawl a certain sequence of processor is needed.

It is thus very easy to introduce additional processors.

The *DeDuplicator* processor is designed to run after the, so called, FetchHTTP processor that downloads HTTP documents. It has two distinct matching modes, by URL or by content digest.

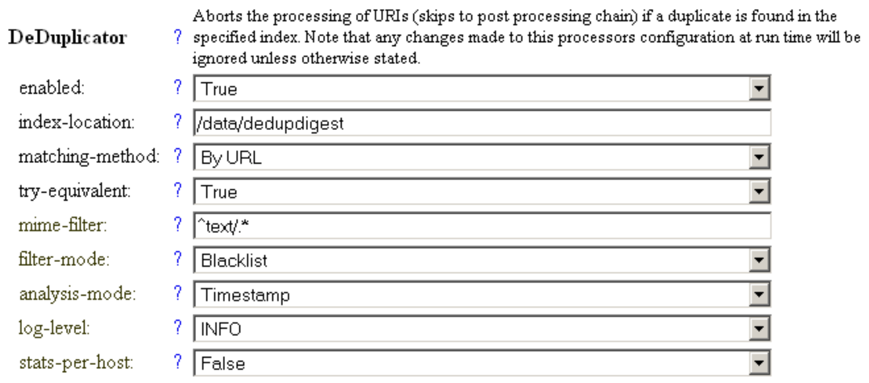
Matching by URL requires a search in the index for the current URL. If no hit is found the index can be optionally searched again, but this time for an equivalent URL using the normalized URL field. If no match is found there either then the URL is considered to be new. If a hit is found the content digests are compared. If identical, further processing of the URL is skipped.

Typically the processor should be right after the fetch processors, so the link extractors and writing processors are skipped. However, it could also be placed after the link extractor to work with documents that contain links we need.

Matching by content digest works in a similar fashion. If there is no match then the document is new. If there is a match then the URLs are compared and then the normalized URLs. If a match is found then further processing may be skipped.

It is however interesting to note that any match by content digest means that the document has been stored before. While we, currently, insist on an URL match as well, that is simply because the ARC files do not offer the ability to write a 'reference' link stating that a document identical to this other document at a different URL was found when this URL was visited. This facility is planned in the revised ARC format WARC.

The DeDuplicator also, optionally, analyses the accuracy of the 'last-modified' value. This is however only done when an *exact* URL match is made. This facility can be used even when matching by digest but requires that the URL has been indexed.



DeDuplicator ? Aborts the processing of URIs (skips to post processing chain) if a duplicate is found in the specified index. Note that any changes made to this processors configuration at run time will be ignored unless otherwise stated.

enabled: ?	True
index-location: ?	/data/dedupdigest
matching-method: ?	By URL
try-equivalent: ?	True
mime-filter: ?	^text/.*
filter-mode: ?	Blacklist
analysis-mode: ?	Timestamp
log-level: ?	INFO
stats-per-host: ?	False

Fig. 2. The DeDuplicator processor's configurations.

The analysis simply compares the prediction made by comparing the value to the one stored in the index against the result of the the processor's normal matching. The number of accurately predicted changes and non-changes are tallied along with the wrongly predicted changes and non-changes. This facility is primarily a statistics gathering tool for those wishing to evaluate how reliable it would be to use the header information to eliminate duplicates.

Finally, the processor offers the facility to gather statistics per host. This allows a test crawl to discover how accurate, for example, the HTTP headers of individual hosts are.

4.4 Filtering based on header – DeDupFetchHTTP

The DeDuplicator also has a replacement processor for the FetchHTTP processor that examines the HTTP header before proceeding with the download of the content body. It compares the information in the header with information stored in the index if the exact URL can be found in it. It is called *DeDupFetchHTTP*.

Using normalized URLs is not allowed here. This is because the URL normalization is overly aggressive, two different documents may easily resolve to the same normalized URL. This isn't a problem when you also require the content digest to match as the probability of two different documents having the same content digest are infinitesimal. For filtering based on the HTTP headers we do not have that security and so we only allow exact URL matches. Perhaps a more strict URL normalization could be added that would be safe for this usage but given that only a very small part of the duplicates have been discovered via this mechanism it isn't a priority.

The processor offers four matching schemes, using datestamp, Etag, 'datestamp *and* Etag' and 'datestamp *or* Etag' [4]. That is to say, using either of the two indicators discussed previously or both, where they must either agree on change, or where only one needs to indicate change.

It is worth noting that when using the 'last-modified' we evaluate the actual time and compare it against the date in the index. This is necessary since the date in the index may not be the document's 'last-modified' value. In the current implementation it is the time when the document was last crawled as it is based on the timestamp in the crawl.log. The crawl.log does not contain the last-modified field. We note this since many previous experiments with HTTP header information have regarded any change to the 'last-modified' value to indicate change.

This module has not been tested to the same extent as the regular DeDuplicator processor and, while it fully supports Etag, support for it in the index writer is missing since Etag is not included in the crawl.log.

4.5 Installation and integration with Heritrix

The software has been made available under the Lesser General Public License [11] and can be downloaded from its own website[13]. To install a binary release, users will need to have an existing Heritrix installation. Once the tarball containing the DeDuplicator module is untarred into Heritrix's home directory, Heritrix will recognize it and the processors discussed above will show up as options in the configuration of crawls.

Since even the indexing part of the software leverages Heritrix components, it goes without saying that it is entirely non-functional without a compatible Heritrix installation. At the moment it has been confirmed to be compatible with Heritrix 1.6.0 and onward, although it should work with even older versions.

5 Crawl results

During the first half of 2006, several crawls were made at the National and University Library of Iceland with the DeDuplicator. The crawls can be divided into two categories.

One, are the large snapshots of the entire .is top level domain. One such crawl was conducted in early March and another in late May. Each crawl was started with roughly 15 thousand seeds covering all the second tier domains under .is. The crawls covered about 24 million URLs and spanned a little over two weeks each.

Second, in relation to the 2006 local election held in late May, weekly crawls were made of selected sites of particular interest. The number of sites grew during the projects duration to around 180. Each crawl lasted 24 hours and covered about 1.3 million documents.

In both cases, the crawls were quite deep, with the goal of making as thorough captures as possible. Typically, only a handful of very deep sites were still being processed when the crawls were stopped.

The following analysis relies on data from these crawls.

5.1 Data

Since our concern was primarily to reduce storage costs, we emphasized the post-download analysis of documents. It is safe to say that the results were highly encouraging.

The March snapshot of the .is domain was performed using an index based on a crawl made four months earlier. Aside from adding the DeDuplicator, the configuration of the crawl closely mimicked its predecessor[1]. The total amount of data downloaded was similar (1260 GB) however the amount of data stored on disk was reduced from 898 GB to 522 GB (this includes ARC files, logs, configuration files and anything else associated with the crawl). A saving of nearly 42%.

Looking at the numbers more closely we found that 60.66% of the documents considered by the DeDuplicator processor (i.e. non-text documents) were duplicates. All told, of the 24 million URLs crawled, 7.1 million were non-text documents and of those 4.3 million were duplicates.

```
Processor: is.hi.bok.digest.DeDuplicator
  Function:          Abort processing of duplicate
                                                records
                     - Lookup by url in use
  Total handled:      7101240
  Duplicates found:   4307660 60.66%
  Bytes total:        941110791887 (876 GB)
  Bytes discarded:    520523548534 (484 GB) 55.30%
  New (no hits):      2793586
  Exact hits:         4148297
  Equivalent hits:    159357
  Timestamp predicts: (where exact URL hits could be
                                                 found)
  Change correctly:   36058
  Change falsely:     203906
  Non-change correct: 3783013
  Non-change falsely: 3281
  Missing timpestamp: 162195
```

Fig. 3. The DeDuplicator's processor report for the March .is crawl.

It is interesting to note that 4,148,297 documents were ruled duplicates based on exact URL match (another 159,357 based on normalized URL match) while there were only a total of 4,188,453 URLs that matched an URL in the index (regardless of whether or not the content digest also matched.) This means that only 40,156 of the documents crawled (about 0.5%) had actually *changed*. The remainder are new documents. This supports our original assumption that these documents don't usually change.

[1] Post crawl analysis had identified some new crawler traps and handling for them was added to the March crawl.

The non-text documents accounted for 876 GB of which 484 GB were deemed duplicate data. A saving of about 55%. These numbers were encouraging enough, however we also realized during the post crawl analysis that the crawler had discovered several new sites providing HTTP access to FTP sites hosting Linux distributions that had not been crawled before. This explains why the amount of duplicate data was a smaller proportion of the total than the number of documents. Typically, we have found the reverse to be true, presumably because larger files remain more constant.

The second large scale crawl was conducted at the end of May, about 2 months after the completion of the March crawl. The May crawl's index was based on the March crawl.log.

The results from the May crawl largely bear out the results of the March crawl. Since it was made with a fresher index, the numbers are somewhat higher. About 67% of the documents have been deemed duplicates.

Looking at the weekly election crawls, we see an even more exciting picture. As much as 95% of the data was being discarded as duplicates. The fact that new sites were often being added makes the numbers a little less accurate than we would like but the overall picture was consistent.

Based on our experiences we can say that there are several factors that will influence how much duplicate data will be encountered. The most significant, obviously, is the amount of time that passes between snapshots and mapping out that curve might be an interesting experiment for the future. However, we must also consider the depth of the crawl and the nature of the sites being crawled. Obviously, deeper crawls will encounter more 'archive' data that remains relatively immutable. Also, the nature of the sites will greatly impact this. Some sites change much more than others. Especially in our weekly crawls (where sites were chosen in part because they are very active) we would expect the yields to lower.

There is clearly a lot of work that could be done analyzing trends here, but we leave that for the future.

We've also experimented with applying the DeDuplicator to text document once link extraction has been performed. During one of our weekly crawls 32% of processed documents and 24% of the data were deemed duplicates. This result is of course much worse then the ones we've seen for non-text documents and as we'll see a little later, not worth the performance hit in larger crawls.

5.2 HTTP headers

As noted the DeDuplicator processor can gather statistics about the usefulness[2] and reliability[3] of the HTTP headers. For all of our crawls we have had this option enabled and the results have been fairly consistent.

[2] "Usefulness is measured in the percentage of unchanged pages we avoid downloading" [4]

[3] "Reliability is measured in the percentage of changed pages we decide to download." [4]

The analysis is performed for every document whose URL is found in the index. It will then fall into one of the following categories:

- **Change correctly** – The header information correctly indicates change.
- **Change falsely** – The header information incorrectly indicates change. While this reduces the usefulness of the information it will not cause us to miss any content. It only leads to some needless downloads.
- **Non-change correct** – The header information correctly indicated that the document had not changed.
- **Non-change falsely** – The header information *incorrectly* indicated that the document had not changed. This is, of course, the worst case scenario. Any document that is incorrectly assumed to be a duplicate is missed.
- **Missing timestamp** – The header did not contain the 'last-modified' value. This reduces the usefulness of the header but otherwise isn't an issue.

The results of our March .is crawl can be seen in figure 3. Table 1 contains the values from the May crawl which may be more accurate since a minor bug in the analysis code was discovered during the time between the two crawls.

Table 1. HTTP Header analysis from the May .is crawl.

	Count	**Percentage**
Change correctly:	24,081	0,54%
Change falsely:	139,213	3,14%
Non-change correct:	4,095,377	92,43%
Non-change falsely:	1,247	0,03%
Missing timestamp:	170,777	3,85%
Total URLs analyzed	4,430,695	100%

The total number of URLs analyzed (i.e. URL was found in the index) was about 64% of the total number of non-text documents. As table 1 clearly shows only 0.03% of the documents are incorrectly assumed to have not changed which seems very good. However this is 4.92% of the documents that had actually changed leading to a reliability of only 95.08%. Other crawls produce similar numbers.

This is much worse than the 99.7% reliability discovered by [4]. Note however that in [4] the percentage of changed documents in total is much higher. That study examined *all* file types. They also crawled much more frequently and as a consequence don't have the large number of 'new' documents that have no past information. If we count all of those new document as 'changed' (having gone from non-existance into the current version – the distinction is open to argument) the accuracy is actually 99.95%.

Ultimately, all we can say is that it is a very small number of the overall volume that is incorrectly discarded in this manner. The decision on whether or not to employ this type of filtering will most likely depend on the nature of the crawl. You could argue that for very large scale crawls the minute number of files that you miss is outweighed by the number of additional files you can crawl because of the reduced load on your bandwidth. That is of course assuming bandwidth is an issue.

That leads us to the usefulness of the datestamp where we note that 92.43% of the headers correctly indicate that the document hasn't changed. Considering only the documents that haven't changed this is a usefulness of 96.71%, notably higher than the 63.7% observed by [4].

In both cases it is possible that the results are because of differences between the .dk and .is top level domains, but it is more likely that the nature of documents plays a role here, as well as the depth of the crawls. In particular, we note that since non-text documents are typically not generated automatically the web servers are more likely to be able to provide 'last-modified' data as these are files in their directories. That is the most plausible explanation, but it would take further study to confirm it.

We'll simply say that if you are willing to miss a few documents here and there, considerable savings can be achieved using this method

To test this out we performed an extra iteration of one of our weekly crawls using the DeDupFetchHTTP processor to test filtering based on header information. A total of 65% of the documents it handled (i.e. non-text documents) were discarded based on the 'last-modified' information.

The DeDuplicator processor was then applied to what was downloaded and 11% of that was deemed duplicate. In total 76% of the documents where thus deemed duplicates. This was slightly higher than the 75% deemed duplicate by a crawl run one day later that was identical except that it didn't use the DeDupFetchHTTP processor. One extra day of new material easily accounts for the 1% difference.

Since most of the additional duplicates (80%) found by the DeDuplicator processor where in fact detected using normalized URLs, these numbers are fairly consistent with what could be expected. The DeDupFetchHTTP processor will catch well over 90% of the changed content. We estimate that we saved around 30 GB in network traffic (of a total of 47 GB without it).

The abnormally low number of duplicates (75%) compared with the large crawls is due to the nature of the sites being crawled weekly. They were chosen, in part, because they change frequently. It is worth noting however that the amount of data discarded was typically around 95% of the non-text data.

5.3 Performance

It is difficult to accurately gauge the impact that the DeDuplicator has on Heritrix's performance. The weekly crawls were simply too small for there to be any impact, since they are limited by politeness rules and not available resources. Comparing the large scale crawls is also somewhat difficult. They were made months apart and with somewhat different configurations and software versions.

We have however observed certain things. When trying to use a full index (i.e. filter out text documents as well as non-text) the impact on the crawler is significant and unacceptable for the large crawls. It is however negligible for the weekly crawls. The impact of just filtering out non-text documents on large crawls seems small initially, in fact we observed improved document per second ratios compared to earlier crawls for the first few hours, but over the course of the crawl we estimated a 10-15% reduction in crawl speed.

This behavior may be related to how Heritrix handles its internal state which is largely stored in Berkley Databases (BDB) [8]. The BDBs are written to disk but also have effective in-memory caching meaning that initially there is little need to read from disk. However, as the crawl progresses the BDB must increasingly access content on disk. This then begins to contend with Lucene's request for disk access. More effective in-memory structures may help alleviate this.

6 Issues and future work

There are a number of issues we could raise. Improved change detection and more reliable HTTP headers are prime candidates. However, they are so large in scope that any effective solution is not only beyond the focus of this paper, but such a solution might well make the DeDuplicator unnecessary as incremental crawling would become a feasible choice.

There is however one particular issue with the DeDuplicator that we must address. What happens to the duplicates?

So far we have been focusing on detecting and eliminating duplicates, but this leads to ‘holes’ in our archives. That may not be a serious issue if the outcomes of all the crawls relying on each other are stored/indexed together, but what if that is not the case? Indexing a collection with tens of millions of documents is difficult; with hundreds of millions it becomes almost impossible with current tools.

The DeDuplicator already has a partial answer. Each URL can be associated with an *origin*. Origin is simply a text string and is operator configurable at the moment. Typically we expect it to be a reference to a particular crawl.

The only place this information will be stored, at present, is in the crawl.log that maintains a list of all processed URLs and various details of the processing. Since the crawl.log is used to build indexes it is possible to carry the origin of duplicates on through any number of repeated crawls.

This is however not a complete solution. We would like to store this information in the web archive itself, however the ARC file format used to store crawl data does not offer any facility for that. Fortunately, the new Web ARChive File Format (WARC) [9] does.

Basically, the WARC file format is a substantial revision of the previous ARC format. Most notable for the discussion at hand is the inclusion of the *revisit* record type. Effectively this means that we can record in the WARC that a particular URL was visited and found to be unchanged. The revisit record requires a parameter named 'Related-Record-ID' that points to the original record. Clearly our ‘origin’ would be the data for this field.

The WARC file format is in the process of being submitted as a standard to ISO [12]. Once the format has been finalized and support for it has been added to Heritrix, it will be possible to much more accurately record the results of crawls using the DeDuplicator.

7 Conclusions

The DeDuplicator software has proven itself to be an extremely useful addition to Heritrix. While there are difficulties in presenting collections that have been filtered with it, the introduction of the WARC file format should greatly alleviate that.

Let's recall the four main arguments for doing this type of filtering, presented at the beginning of the paper:

1) Reduce load on web servers. Improve politeness.
2) Reduce bandwidth usage.
3) Reduce storage costs.
4) Improve the quality of the collection or its presentation.

The first two both focus on reducing network traffic. As we've already discussed, in the absence of adaptive incremental strategies (which are not feasibly on a large scale) the only option to achieve this is to use the HTTP headers. We've seen how the DeDuplicator can achieve this and at what cost.

Reducing storage cost is where the DeDuplicator shines. Considering the terabytes of data many organizations are harvesting each year, a significant saving can be achieved, especially by those organizations doing frequent and/or deep crawling.

The last issue revolves more on presentation; we do not want multiple unchanged documents to be presented. The assumption is that people are most interested in the changes. The DeDuplicator may improve this slightly but ultimately does not tackle in any comprehensive manner since accurately detecting change in text documents is very difficult.

It is worth noting that the last issue can potentially be handled by access tools that can use various 'close enough' comparison algorithms etc. without fear of losing data. At the moment we recommend that strategy be pursued to improve presentation.

References

1. Kristinn Sigurðsson: *Incremental Crawling with Heritrix*. Accessed May 2006. http://www.iwaw.net/05/papers/iwaw05-sigurdsson.pdf
2. Gordon Mohr et al.: *Introduction to Heritrix*. Accessed May 2005. http://www.iwaw.net/04/proceedings.php?f=Mohr
3. R. Fielding et al: *Hypertext Transfer Protocol -- HTTP/1.1*. Accessed May 2006. http://www.w3.org/Protocols/rfc2616/rfc2616.html
4. Lars R. Clausen: *Concerning Etags and Datestamps*. Accessed May 2005. http://www.iwaw.net/04/proceedings.php?f=Clausen
5. Andy Boyko: *Characterizing Change in Web Archiving*. Internal IIPC document, unpublished.
6. *FIPS 180-1 – Secure Hash Standard*. Accessed June 2006. http://www.itl.nist.gov/fipspubs/fip180-1.htm
7. *Lucene*. Accessed June 2006. http://lucene.apache.org/
8. *Berkeley DB Java Edition*. Accessed May 2005. http://www.sleepycat.com/products/je.shtml
9. John A. Kunze et al.: *The WARC File Format (Version 0.9)*. Accessed June 2006.

http://archive-access.cvs.sourceforge.net/*checkout*/archive-access/archive-access/src/docs/warc/warc_file_format.html?revision=1.10
10. John Erik Halse et al.: *Heritrix developer documentation*. Accessed May 2005. http://crawler.archive.org/articles/developer_manual.html
11. *GNU Lesser General Public License*. Accessed June 2006. http://www.gnu.org/licenses/lgpl.html
12. ISO – International Organization for Standardization. Accessed June 2006. http://iso.org/
13. *DeDuplicator*. Accessed August 2006. http://vefsofnun.bok.hi.is/deduplicator/