Incremental crawling with Heritrix

Kristinn Sigurðsson

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

Abstract. The Heritrix web crawler aims to be the world's first open source, extensible, web-scale, archival-quality web crawler. It has however been limited in its crawling strategies to *snapshot* crawling. This paper reports on work to add the ability to conduct *incremental* crawls to its capabilities. We first discuss the concept of incremental crawling as opposed to snapshot crawling and then the possible ways to design an effective incremental strategy. An overview is given of the implementation that we did, its limits and strengths are discussed. We then report on the results of initial experimentation with the new software which have gone well. Finally, we discuss issues that remain unresolved and possible future improvements.

1 Introduction

With an increasing number of parties interested in crawling the World Wide Web, for a variety of reasons, a number of different crawl types have emerged. The development team at Internet Archive [12] responsible for the Heritrix web crawler, have highlighted four distinct variations [1], *broad*, *focused, continuous* and *experimental* crawling.

Broad and focused crawls are in many ways similar, the primary difference being that broad crawls emphasize capturing a large scope[1], whereas focused crawling calls for a more complete coverage of a smaller scope. Both approaches use a *snapshot strategy*, which involves crawling the scope once and once only. Of course, crawls are repeatable but only by starting again from the seeds. No information from past crawls is used in new ones, except possibly some changes to the configuration made by the operator, to avoid crawler traps etc.

The snapshot strategy (sometimes referred to as periodic crawling [10]) is useful for large-scale crawls in that it minimizes the amount of state information that needs to be stored at any one time. Once a resource has been collected, the crawler need

[1] In the context of web crawling, *scope*, refers to the set of URIs that are, by some virtue, a desired part of the crawl. That is if encountered they should be crawled. This is also sometimes refered to as a segment.

only store a fingerprint of the URIs[2] and data structures are also simplified since *first-in-first-out* queues can be used to hold discovered resources until they can be processed.

This makes it possible to crawl a fairly large scope using a snapshot approach and initially this is what Heritrix did. As of version 1.0.0, it offered decent capabilities for focused crawls and work has continued on expanding its capabilities to cope with larger crawls.

However, this does not do a good job of capturing changes in resources. Large crawls take time, meaning that there is a significant gap between revisits. Even if crawled within a reasonable amount of time, a snapshot crawl will fail to detect unchanged documents, leading to needless duplicates. Snapshot crawling is therefore primarily of use for large scale crawling, that is either crawling a large number of websites or trying to crawl each website completely (leading to very 'deep' crawls) or both.

Continuous crawling requires the crawler to revisit the same resources at certain intervals. This means that the crawler must retain detailed state information and, via some intelligent control, reschedule resources that have already been processed. Here, an *incremental strategy* is called for rather than the snapshot strategy used in broad and focused crawls.

An incremental strategy maintains a record of each resource's history, this is used in turn to determine its ordering in a priority queue of resources waiting to be fetched.

Using adaptive revisiting techniques, it is possible to capture the changes in online resources within its scope far more accurately, in turn allowing the crawler to revisit each page more often. In addition, revisits are not bound by crawling cycles (as with the snapshot approach), since any page can come up for revisiting at any time, rather than only once during a run of the crawler. However, because of additional overhead, and because of the need to revisit resources, an incremental strategy is unable to cope with as broad a scope as a snapshot crawl might.

To sum things up, the choice between an incremental or snapshot strategy can be described as choosing between space and time completeness. Naturally, we would wish to capture both well and with no hardware limitations we might do so, but in light of limited bandwidth and storage, difficult decisions must to be made.

As noted above, Heritrix already implements a snapshot strategy for its focused crawls. While work continues to expand its capabilities in order to scale up to truly broad crawls, the need for an incremental strategy also needs to be addressed. This led us to implementing a generic incremental strategy for Heritrix. One that would both serve as a proof of concept, that the Heritrix framework could in fact support an incremental strategy and would provide the basic building blocks for tailoring such crawls.

Several approaches to creating an incremental crawler have been either implemented or proposed [10], [13], [5], none of them takes the Heritrix framework into account. As a result we have chosen to look at the basic requirements of continuous crawling and what knowledge we have of the way documents change on the World Wide Web, to develop an appropriate strategy for Heritrix.

---

[2] URIs (Uniform Resource Identifiers) are a superset of the more familiar URLs. Heritrix, while supporting only URLs, uses the more general term URI for future proofing.[7]

What we wanted to create is an addition to Heritrix that allows for incremental crawling, while both maintaining Heritrix's inherent modularity and not compromising the efficiency of its snapshot crawling. Making it possible to swap out appropriate portions of the logic by substituting certain modules. Basically, provide the building blocks for an incremental strategy with Heritrix and one solid implementation of it.

We have focused on 'adaptive' techniques as a way to address the extreme variety found on the Internet. While mostly heuristic in nature, we believe that an adaptive approach can make much better use of available resources.

In the following sections, we'll start by looking at what we know of the Web that can help shape possible strategies for incremental crawling. We then continue to describe how we implemented one variation of those for Heritrix and discuss briefly our experiences with the resulting experimental features. Finally, we'll outline some of the main challenges that still lie ahead.

2 Incremental strategy

The basic goal of any incremental strategy is to capture every change in all the resources within its scope. Clearly, unless the scope is extremely limited, this cannot be achieved, since it would require a constant flurry of requests to the server to check for changes and even if you did have the resources to do that, it would place an excessive strain on the server.

One obvious strategy here is to crawl the entire scope, then start again, and replay the crawl. Resources that disappear between iterations are discarded, while newly discovered resources are inserted into the sequence. While this would work, it fails to take advantage of a critical fact; the resources have widely varying probabilities of change within the same timeframe.

We need a way to predict how often a resource is likely to change in a given time span, or rather, how long we can wait before revisiting it and still be reasonably sure of not missing a version. This calls for a set of heuristics that estimate the probability of change and, using that estimate, attempts to schedule resources for revisiting at the appropriate time.

To get an idea what these heuristics might use, let us review what we know about resources on the internet and documents on the World Wide Web in particular. It is worth noting quickly that the following lists for the most part general trends that, while common enough to serve for heuristic evaluation, do have many exceptions.

Firstly, resources that change often are likely to continue to do so. If we have observed that a document changes frequently, it is probably safe to visit it frequently and vica versa. Of course, this offers no guarantee. Websites may be updated sporadically. A typical example would be the website of a politician. During most of his elected tenure the website might only be updated once every few months. However, when elections are approaching, the website is updated much more frequently, possibly many times a day.

Despite this drawback, most websites and resources are believed to have reasonably regular change rates. After all, a web site that wishes to attract regular

readers needs to be consistent in its presentation of new material. As noted before, this is not an absolute and further research is needed to establish just how much an issue erratic change rates in web pages are.

Secondly, the file type of resources, or documents, significantly affects the probable change rates. [2] While changing HTML or other simple text files is both easy and straight forward (not to mention expected by the user in many cases and frequently handled by content management systems), changing images is more complex. In fact browsers and proxies will generally cache images, while HTML pages are requested for each page view. Images and other multimedia files are usually not generated on demand, unlike many HTML files, and when a new or changed graphic is needed, it is customary to *add* rather than replace an image.

This allows us to predict different initial wait times between revisits before we gather any further information that enables us to adapt to observed change rates. HTML and other pure text documents can be assigned a relatively short initial wait, while a longer initial wait time is assigned to image files. Higher bandwidth media such as audio and video files are even less likely to change, due to their complex nature and can thus be assigned even higher initial wait times. Formatted text files such as Word and PDF documents are also less likely to change than pure text files.

There is a certain amount of connection between the size of a resource and its change rate. As the size increases, its *likely* change rate goes down. This assumes that image, audio, video and formatted files are generally larger than pure text documents. While not always true, this generally holds.

We are therefore able to use this information not only to reduce the number of visits we must make to a certain part of our scope, but to one of the most bandwidth intensive part of our scope.

Another factor that might be taken into consideration is a document's location in a web site's hierarchy. Generally, the front page of a web site changes more often than any given page in the archive (where changes may never occur). Assuming that the initial seeds are the front pages of each site, it would be possible to use the depth from seed as a factor in deciding the initial wait time.

Finally, one could conceivably do some form of contextual analysis on the resource's content or metadata to evaluate the probable change rate. For example [4] observes that documents without the *last-modified* information are about twice as likely to change between two visits, as those that have it. There are undoubtedly many other possible indicators that could be used, but covering them all is well outside the scope of this article.

The primary goal in using incremental crawling is to to utilize bandwidth more effectively and thus limit the stress our harvesting places on web servers. It is worth noting however that while doing this we also achieve a secondary benefit in reduced storage costs, since duplicate data need not be stored. That however brings us to another problem; detecting change.

One of the key aspects discussed above, relies on being able to detect if a change has occurred between the times when a resource is visited. Doing a simple, strict hash, comparison does achieve this to a degree. It certainly detects any change and reduces the odds that we discard a changed document to nearly zero. Unfortunately, it relies on a bit by bit comparison and takes no notice of the actual content. Thus *any* change

will be detected, even something as utterly trivial as a clock feature on a web site changing every second.

The problem lies in the dynamic content generation of web servers. While normally, any change would be of interest, automatic changes that do not relate to the content, but rather time or some other factor, are not necessarily of interest. We are primarily interested in changes to the content.

A good deal of work has been done on the subject of detecting documents that are "roughly the same," [6] for example outlines a "shingling" method for evaluation, saying, basically, that if enough shingles are the same for two documents, they are the same.

While such approximations may be valid most of the time, it is questionable to employ them without first examining how they handle the documents within the intended scope of any given crawl.

An alternative idea is to provide the operator with the ability to selectively weaken the strict hash. This could be done by specifying regular expressions and omitting the sections of documents that match them when creating a hash. While costly in terms of processor power, it allows the operator to tackle known issues.

While one might argue that just tackling known issues is not enough when crawling the entire web, it is worth remembering that continuous crawls are more likely to be focused on a carefully selected, and in comparison to broad crawls, small number of web sites. This method also has the advantage of always erring on the side of caution.

3 Implementing a strategy

When developing an addition to the Heritrix crawler, two primary aspects need to be addressed. The first is the so-called *Frontier*. A frontier is a pluggable class that maintains the state of the crawl. The default frontier implements a snapshot strategy, where it is assumed that URIs are to be crawled once and once only. Clearly, this won't work.

The other important aspect is the chain of processors. When the Frontier issues a URI for crawling, it passes through a series of processors. Each processor is a pluggable class, and there is no limit to the total number of processors, so new ones can be added. Typically, each processor does a discrete unit of work, like fetching the document over the internet, performing link extraction or writing the contents to disk. This provides a modular system, allowing us to make full use of existing processors, while adding a few extra that are needed to manage a continuous crawl.

We created two key processors, the *ChangeEvaluator* and the *WaitEvaluator*.

The ChangeEvaluator runs right after a document has been fetched. It decides if the document has changed or not based on a hash comparison (more on that later). Changed documents are marked appropriately and move on down the processor chain, whereas the processing of unchanged documents is preempted and they are sent to the post processing portion for completion. It is still regarded as a 'success' but we skip over link extraction that is of no value if a document has not changed.

This preemption also bypasses the writing phase of the processor chain. While this is acceptable with current ARC formats, since they do not allow records stating that a duplicate was found, in the future that may change.

To facilitate this, a *content state*, could be added to the CrawlURI object, and thus to the Heritrix framework API. The CrawlURI objects do allow any arbitrary data to be stored and retrieved in a hash map. This is sufficient for most data, however, it might be argued that the content state was of a more general nature. In the future, processors should respect the 'content state' information of the URIs being passed through the processing chain, passing over unchanged documents or otherwise treating them in an appropriate manner. Ultimately, this was not done because of a desire to minimize changes to the Heritrix framework. Instead the content state is stored in the CrawlURI's hash map. Hopefully this will be revisited in the future.

The *WaitEvaluator* runs in the post processing section of the URI processing chain. A wait interval is assigned to each URI between fetches. When a URI reaches the WaitEvaluator it examines this value and calculates a new one based on whether or not the document had changed between visits. If the wait interval is missing, a default initial value is assigned. That should only happen when a URI is processed for the first time. URIs that cannot be evaluated for change, however, are assigned a fixed and permanent wait interval. Currently this applies only to DNS lookups.

A simple strategy for estimating a new wait interval is used. If a document has changed, its wait interval is divided by a constant. Similarly, if it is unchanged it is multiplied by a constant. By default, only the previous wait interval is considered. However, odds are that URIs will be somewhat overdue, possibly very overdue and thus there is an optional feature to use the actual time that has elapsed between visits in determining the new wait time.

The factors, as well as the initial wait time and maximum and minimum wait times are all configurable elements. Furthermore, Heritrix's overrides and refinements can be used to specify these values based on a variety of criteria.

Using the overrides[8] in Heritrix, it is possible to override the settings of the WaitEvaluator on a 'per host basis.' Essentially, each domain or host can have different initial wait time, factors etc. This allows the operator to pre-adapt the crawl to known conditions.

In order to differentiate the settings based on the content type of documents, several specialized WaitEvaluators are provided. The specialized WaitEvaluators have the same settings, but also have a configurable regular expression that must match documents' content types. The different WaitEvaluators can thus be 'stacked up' in the processing chain. Once a WaitEvaluator has handled a URI subsequent evaluators will pass on it. At the bottom of the stack the generic WaitEvaluator then captures all file types not explicitly handled. This is very similar to how the fetch and link extraction processors operate.

We then created a new Frontier, dubbed the *AdaptiveRevisitFrontier* (or ARFrontier for short). While it relies, like other frontiers, on a series of per-host based queues, it implements them as priority queues rather than simple *FIFO* queues. Additionally, the queues have richer state control, allowing them to be automatically suspended (or, in Heritrix jargon, snoozed) if no URI is ready for crawling. This is accomplished by assigning each URI a 'time of next processing' when they are scheduled or return from being processed. This time is based on the URIs assigned

wait interval, which in turn is based on the WaitEvaluator's prediction of when it will have changed again. While our implementation of the queues is more cumbersome in terms of memory than the leaner snapshot frontier, it does offer a simpler approach to maintaining the crawl's state and by extension makes the frontier's design less error prone[3].

It was immediately clear that to achieve this, it would be necessary to use a much more flexible approach to storing data on disk than the custom written solution that the previous *HostQueuesFrontier* used. We quickly narrowed the candidates down to the Berkley DB Java Edition[9]. It, being an embedded, pure Java, object oriented database, seemed like the perfect candidate. In fact, it has proven itself to be quite apt to the task.

Interestingly enough, parallel to the development of the ARFrontier, the Internet Archive began developing a new Frontier to overcome many of the shortcomings of the HostQueuesFrontier that had shipped with version 1.0.0. This new Frontier (The BdbFrontier) was also based on the Berkley DB and support for it is now an integral part of the Heritrix framework.

The ChangeEvaluator relies on strict hashing (actually performed automatically by the FetchHTTP processor when a document is fetched). While generally acceptable, there are known instances of irrelevant data that changes with each visit. To address this, we created another processor, HTTPContentDigest. It is inserted into the processing chain between the fetch processor and the ChangeEvaluator. An operator enters a regular expression matching sections in documents that should be omitted. The processor then recalculates the hash, ignoring any sections that match the regular expression for documents that it runs on.

While costly, it is only intended to be used against specific known issues. By using overrides and refinements, an operator can create a crawl order that only applies this processor on URIs that are known to be troublesome. It is thus possible to apply different regular expression to each issue and *only* to the documents that contain the problematic section or are very likely to do so.

On another front, we wanted to take advantage of HTTP header information. We decided to create a filter that the HTTP fetcher processor could run 'mid fetch'[4] to evaluate if the download should be aborted because the document had not changed since the last visit.

The rather cumbersomely named *HTTPMidFetchUnhangedFilter* checks both the *last-modified* timestamp and the *etag* value and compares it to the results of the most previous crawl of the URI. If neither is present, the filter can of course not do anything. If only one is present, the filter will determine that the document is unchanged if that value has not changed. If both indicators are found, then they must agree that the document has not changed, otherwise the filter assumes that it has changed. This provides high reliability at the cost of some usefulness. When both indicators are present, you could somewhat increase their usefulness by only

[3] The juggling of Frontier queues and maintaining their status has been one of the most problematic parts of the previous Heritrix frontiers.

[4] The FetchHTTP runs custom filters 'mid fetch' after the HTTP headers have been downloaded and prior to the download of the content body. Downloads of documents that do not match this filter are aborted.

downloading documents when both indicators predict change, however this slightly reduces their reliability[3] and so we opted for the more reliable approach.

4 Results

Test crawls run with the software have gone quite well. Depending on the initial and minimum wait settings, as well as the politeness restrictions, it is necessary to limit the number of URIs crawled on each host. As expected, if there are too many, the software is unable to visit them in a timely fashion and their processing is constantly overdue. While this will slowly correct itself on sites where most of the documents remain unchanged, sites with a large number of frequently changing documents will not be as accommodating. If the scope cannot be reduced, then either the minimum time between visits must be increased (this can be done selectively so that 'key' documents are visited more often) or the politeness restrictions loosened.

The ARFrontier supports host valence[5] higher than one, allowing for multiple connections, the minimum wait time between connections can also be reduced to crawl more aggressively. An experimental feature for valence higher than 1 was provided in the snapshot frontier that shipped with Heritrix 1.0.0. This has not been included in the newer BdbFrontier due to the added complexity. The ARFrontier's state rich host queue system however allowed us to add this feature without undue complications.

While the number of URIs per host is somewhat limited by these factors, the overall number of URIs is not as limited. A certain amount of memory overhead is incurred for each new host that is added, but it is still feasible to crawl up to thousands of hosts (although this has not been fully tested yet), and potentially as many as millions of URIs at once. However, as the number of hosts increase, they start blocking each other. Depending on the hardware, Heritrix may be able to crawl as many as 50-60 documents a second. Once you have enough hosts to keep it fully occupied, despite politeness restrictions, a backlog can occur, whereas, if the number of hosts (or URIs) is too small to fully occupy the available resources, URIs will only become overdue because their host is overworked.

Following these successful initial test, the software will be used by the National and University Library of Iceland to augment the regular snapshot crawls of the .is domain. A selection of thirty to forty web sites containing news, political discussions and other similarly valuable content will be crawled continuously. The exact parameters, as far as minimum wait times etc. are still being developed.

We are continuing to experiment with the software, evaluating what are good wait intervals, how well the content prediction works etc. This work will likely continue for some time. A considerable amount of works does remains, both further testing the software and also in experimenting with the configurations to improve the incremental behavior. Not to mention testing possible alternative wait estimation algorithms.

---

[5] The *valence* of a host dictates how many simultaneous request Heritrix can make to that host. Generally, a valence of 1 is recommended to avoid hammering a site.

Finally, the AR 'module' became a part of the standard Heritrix release as of version 1.4.0. The AR work has led to several, mostly minor, changes in the Heritrix framework to better accommodate incremental crawling. This was done without adversely affecting snapshot strategies and serves to strengthen Heritrix's overall flexibility.

Primarily, these changes focus on the *CrawlURI* object that wraps the URI with other essential data. It is passed from the Frontier to the processors and back and represents effectively both the URI and its state. For snapshot crawling, this object could be discarded at the end of the processing cycle[6]. An incremental crawler does not have this luxury, but it was not just a simple matter of keeping it around. Some of the data, by its very nature, must be reset before each processing round. In fact, since the CrawlURI offers a generic keyed hash table where frontiers and processors can store any arbitrary information, it was possible that a processor might store a non-serializable object there, effectively crashing the crawl.

With the AR module now officially released as a part of Heritrix, we hope that others will use it and provide feedback, both in the form of comments and suggestions and also by contributing code.

This was solved by redesigning the CrawlURI to separate the data into transient and persistent data, leaving the responsibility of accurately declaring the data to the processors and other modules. Data that is common to all CrawlURIs was similarly divided and frontiers need only call a post processing cleanup method once they are ready to insert them back into the queues.

While this new addition to Heritrix does not solve every issue with incremental crawling, it does provide the framework for conducting such crawls. Having taken full advantage of Heritrix modular nature, each aspect of the crawling strategy (be it content comparison, change prediction etc.) can be customized, while still making full use of the other components.

5 Unresolved and future issues

Volatile change rates remain an issue. Some papers[10] have suggested that (in general) document changes follow a *Poisson process* and that this could be used to create a better change prediction algorithm. This remains to be seen, but it is clear that there are web sites out there whose change rates vary significantly over the course of some time.

The strategy discussed here has tried to (potentially) capture *all* changes, allowing the revisit intervals to drop down to as little as a second. While that seems excessive, many sites do change often each hour. However, the change rate may be heavily dependant on the time of day [4]. Might it perhaps be best to use all visits from the last 24 hours to predict the next likely change? One could take that further and suggest that *all* previous visits be taken into account.

The possibilities for analyzing such data and even cross-referencing it with other documents within the same sites (that may have a longer history for example) or even

[6] While the existing snapshot frontiers do crawl prerequisite information, such as robots.txt and DNS lookups again at fixed intervals, this is done by recreating the CrawlURIs.

from other sites (with similar characteristics) are almost endless. Such analysis would be costly in terms of performance and currently Heritrix's architecture does not allow a processor handling one URI, to access meta-data stored for another URI. These are, however, implementation issues and if an algorithm were devised that made use of such information to more accurately predict a document's change rate, they would probably be worth overcoming.

Change detection also remains to be tackled. We provided a simple strict comparison, coupled with the ability to selectively 'overlook' section of documents. While this works most of the time it may not be suitable for large incremental crawls and can be a bit unwieldy as well.

This is actually a very large topic and a comprehensive overview is beyond the scope of this article. We'll simply state that the better we are able to detect changes in *content* and separate those changes from changes in *layout* the more effective any adaptive revisiting policy will become. But that still won't address such issues as, say, an 'archive' document containing a section with 'current front page story.' The current front page story is still content of sorts, it just isn't of any worth in the current context.

'Close enough' comparison algorithms [6] may overcome this to some extent, but may cause us to overlook minor changes in the actual content. This minor – in terms of the amount of bytes affected – change might however be considered a significant change in relation to the content. For example, a company or other party might amend a press release on their website to 'tone down' the headline. This might be as small a change as omitting a single word. Yet the fact that this word was deleted might be of considerable significance.

Unfortunately, using HTTP headers does not do much to address this. Dynamically generated content is both the most likely to have irrelevant, constantly changing, data and be missing useful header info. Studies indicate that while HTTP headers are usually reliable in predicting change, their usefulness in accurately predicting non-change is much lower [3]. Barring a remarkable improvements in the use of HTTP headers on the part of web servers, it is doubtful that they will be of greater use than in the current implementation.

For this project, we considered keeping track of the prediction made by the HTTP headers and compare them with the hash comparison. The idea was to use this to build up a 'trust level.' Once a certain threshold is reached we would start using the header information to avoid downloads, but still do occasional samples, reducing the number of samples with rising trust levels. Ultimately this was not done since it seems quite rare that they predict no change, when a change has in fact occurred, but adding this functionality would almost eliminate the chance of us failing to download a changed document because of faulty HTTP header information.

We have primarily focused on assessing change rates by looking at the documents history of change (as observed by us). However, a significant amount of other factors could also be taken into account. As discussed earlier, the documents 'position' in relation to the 'front page' or seed can be considered. Alternatively (or additionally), a ranking system of some sort could be used to evaluate a documents relative worth.

Looking even further, when crawling sites offering RSS feeds[11] or other similar services, we could use them as triggers for revisiting certain sections of the webs, as well as the (presumably) new stories being advertised via the feeds. Obviously not all

sites offer such feeds, nor are all changes reported by them, but they would allow us to capture all major changes accurately, assuming that the feed is well handled.

On a more immediate and practical note, Heritrix's statistics are still rather snapshot oriented. Especially those provided by the web user interface. We took advantage of a feature of the crawl log that allows custom modules to append arbitrary data to each URI's line. This means that it contains such information as the wait time, how overdue it is and the number of visits to a URI. In addition, the frontier report that is accessible via the web GUI contains some highly useful state information.

The progress statistics are however less accommodating and do not differentiate between multiple visits to the same URI and visits to different URIs properly. This is especially notable in the 'console' progress bar in the web GUI that will never reach 100% because a continuous frontier always has more stuff to crawl.

Some discussions have taken place about allowing pluggable web pages and controls in Heritrix but until something along those lines is provided, accurate statistics will be slightly problematic. This is most noticeable in monitoring a crawl, but we do not feel that this causes any problem in post crawl analysis where the crawl log can be analyzed.

6 Parting words

Heritrix aimed to provide a good general platform for web crawling, and therefore we have aimed to provide a good general platform for continuous web crawling within that framework. We are aware that it will not solve every issue, but we are happy to say that it meets our initial criteria and we can only hope that as others try it out and add their improvements and alternative implementations that the AR addition will further strengthen Heritrix.

Finally a word of thanks to Gordon Mohr and Michael Stack of the Internet Archive for their assistance in the development effort. Þorsteinn Hallgrímsson of the National and University Library of Iceland also deserves a big thank you, as without him this project would not have been possible. Lastly, a thank you to the Internal Internet Preservation Consortium for their generous support.

References

1. Gordon Mohr et al.: *Introduction to Heritrix*. Accessed May 2005. http://www.iwaw.net/04/proceedings.php?f=Mohr
2. Andy Boyko: *Characterizing Change in Web Archiving*. Internal IIPC document, unpublished.
3. Lars R. Clausen: *Concerning Etags and Datestamps*. Accessed May 2005. http://www.iwaw.net/04/proceedings.php?f=Clausen
4. Brian E. Brewington and George Cybenko: *How dynamic is the Web?* WWW9 / Computer Networks, 33(16): 257-276, 2000
5. Marc Najork and Allan Heydon: *High-Performance Web Crawling*. Accessed May 2005. ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-173.pdf

6. Andrei Z. Broder: *On the Resemblance and Containment of Documents*. IEEE SEQUENCES '97, pages 21-29, 1998.
7. John Erik Halse et al.: *Heritrix developer documentation.* Accessed May 2005. http://crawler.archive.org/articles/developer_manual.html
8. Kristinn Sigurðsson et al.: *Heritrix user manual*. Accessed May 2005. http://crawler.archive.org/articles/user_manual.html
9. *Berkeley DB Java Edition*. Accessed May 2005. http://www.sleepycat.com/products/je.shtml
10. J. Cho and H. Garcia-Molina: *The evolution of the web and implications for an incremental crawler*. In Proc. of 26th Int. Conf. on Very Large Data Bases, pages 117-128, 2000.
11. Mark Pilgrim: *What is RSS*? Accessed May 2005. http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html
12. *Internet Archive*. Accessed May 2005. http://www.archive.org
13. *Nedlib Factsheet*. Accessed February 2005. http://www.kb.nl/nedlib/