University of Iceland
Faculty of Engineering
Department of Computer Science

Adaptive Revisiting with Heritrix

Master Thesis (30 credits/60 ECTS)

by
Kristinn Sigurðsson
May 2005

Supervisors:
Helgi Þorbergsson, PhD
Þorsteinn Hallgrímsson

Útdráttur á íslensku

Veraldarvefurinn geymir sívaxandi hluta af þekkingu og menningararfi heimsins. Þar sem Vefurinn er einnig sífellt að breytast þá er nú unnið ötullega að því að varðveita innihald hans á hverjum tíma. Þessi vinna er framlenging á skylduskila lögum sem hafa í síðustu aldir stuðlað að því að varðveita prentað efni.

Fyrstu þrír kaflarnir lýsa grundvallar erfiðleikum við það að safna Vefnum og kynnir hugbúnaðinn Heritrix, sem var smíðaður til að vinna það verk. Fyrsti kaflinn einbeitir sér að ástæðunum og bakgrunni þessarar vinnu en kaflar tvö og þrjú beina kastljósinu að tæknilegri þáttum.

Markmið verkefnisins var að þróa nýja tækni til að safna ákveðnum hluta af Vefnum sem er álitinn breytast ört og vera í eðli sínu áhugaverður. Seinni kaflar fjalla um skilgreininu á slíkri aðferðafræði og hvernig hún var útfærð í Heritrix. Hluti þessarar umfjöllunar beinist að því hvernig greina má breytingar í skjölum.

Að lokum er fjallað um fyrstu reynslu af nýja hugbúnaðinum og sjónum er beint að þeim þáttum sem þarfnast frekari vinnu eða athygli. Þar sem markmiðið með verkefninu var að leggja grunnlínur fyrir svona aðferðafræði og útbúa einfalda og stöðuga útfærsla þá inniheldur þessi hluti margar hugmyndir um hvað mætti gera betur.

Keywords

Web crawling, web archiving, Heritrix, Internet, World Wide Web, legal deposit, electronic legal deposit.

Abstract

The World Wide Web contains an increasingly significant amount of the world's knowledge and heritage. Since the Web is also in a constant state of change significant efforts are now underway to capture and preserve its contents. These efforts extend the traditional legal deposit laws that have been aimed at preserving printed material over the last centuries.

The first three chapters outline the fundamental challenges for collecting the Web and present the software, Heritrix, which has been designed to perform this task. The first chapter focuses on the reasons and history behind this endeavour, with chapters two and three focusing on more technical aspects.

The goal of this project was to develop a new way of collecting parts of the Web that are believed to change very rapidly and are considered of significant interest. The later chapters focus on defining such an incremental strategy, which we call an 'adaptive revisting strategy' and how it was implemented as a part of Heritrix. A part of this discussion is how to detect change in documents.

Finally we discuss initial impressions of the new software and highlight areas that require further work or attention. As the goal of the project was primarily to establish the foundation for such incremental crawling and provide a simple and sturdy implementation, this section contains many thoughts on issues that could be improved on in the future.

Table of contents

Tables

Figures

1. Background

Since the World Wide Web's inception in the early '90s it has grown at a phenomenal rate. The amount and diversity of content has rapidly increased and almost from the very start, the only way to locate anything you didn't already have a link to was to use a search engine.

It is fair to say that search engines have been critical in the development of the World Wide Web. They continually explore the web and index the pages they discover. Exploring the web or crawling it as it's more commonly known, is one of the keys to creating an effective search engine. However, if you are crawling the web, you can do a lot more than just *index* the contents. Soon people began to realize that the web isn't just growing very fast, it is also constantly *changing*. In order to preserve what is on the web now, you need to archive it.

1.1 Web archiving

The first serious attempts at archiving the World Wide Web began in 1996 when Brewster Kahle founded the non-profit organization Internet Archive (IA) in San Francisco with the goal of creating a permanent collection of the web, freely accessible to anyone [15]. In order to achieve this, IA negotiated with a company called Alexa Internet. Alexa conducted large crawls for data mining purposes and IA then received the stored content after a certain amount of time had elapsed.

The content is stored on hard disks and made accessible via a tool known as the *Wayback Machine* [21]. The Wayback Machine allows users to browse the archive using URLs, much as you would browse the real web. They can furthermore choose amongst the many versions of each URL stored in the collection. Full text searches, like those offered by web search engines, are not possible.

Around the same time as IA began its collections, Sweden's national library began a project, Kulturarw[3], to collect all Swedish webpages, essentially creating a snapshot of the Swedish part of the web. The goal being to preserve them for future generations [22]. At the same time in

Australia, work began on more selective archiving of hand picked web sites [23].

Soon, many other national libraries were also either experimenting with web archiving or actively engaged in it. The web was becoming of ever increasing importance in daily life and increasingly held information that was not available elsewhere.

1.2 Legal deposit

Traditional legal deposit laws require that for all books published in a country, some number of copies be handed over to the national library, or its equivalent, for long term preservation. This tradition traces it origins back to the great library of Alexandria. In Iceland the earliest legal deposit laws date back to 1662 when a royal decree was issued that two copies of every book printed in the country’s only press should be handed over to the king of Denmark [15].

Legal deposit laws continued to develop over the course of the next few centuries. In 1977 they were extended to cover not only printed material, but also pre-recorded audio. This precedence of widening the scope of legal deposit laws was continued with the most recent legislation from 2003. They now cover not only printed material and analog and digital recordings (compact disks) released, but also digital material on the web [15].

Most people don't realize just how extensive the legal deposit laws are. Almost anything printed in volume at a printing press should be handed in. This doesn't just include books and magazines, but also pamphlets and various material intended for mass advertising via mail.

1.3 Electronic legal deposit laws

Legal deposit of digital media is commonly referred to as *electronic legal deposit* to separate it from traditional legal deposits. While solid media, such as CDs and DVDs can be collected with traditional methods, collection of content made available on the web can not. It is simply infeasible to require everyone who puts information on the web to also hand over a copy to the national library.

The basic reason for this is simple; there are a lot of people creating a great deal of content. With traditional material, the responsibility of complying with legal deposit laws can be placed on the (relatively) few replication companies. Whether they print books or press CDs it is clear who is responsible. With the web, the replication has essentially been taken out of the equation.

People put content on their websites and replication occurs as other people access the website. One might argue that the Internet Service Provider or hosting company should be responsible for delivering the content for legal deposit, but that would place a considerable and unfair burden on them.

The point is that the only way to collect web based material is to go and get it. This is why many national libraries have been working on web crawling and archiving.

It varies from country to country exactly what parts of the web should be archived. In Iceland the law extends to "*all web pages and other data – that are published or made available to the public on the Icelandic part of the World Wide Web, i.e. the national domain .is, as well as material published on other top level domains in Icelandic or by Icelandic parties*."[19] The basic assumption here is that we can not know what material may be of value in the future, so we try to capture all of it or at least as much as possible. Others choose to selectively crawl parts of the web, based on some criteria or topic.

1.4 Cooperation

In the summer of 1997, the national libraries of the five Nordic countries began to cooperate informally in the collection and future preservation of each country's web. This cooperation was fomalized in 1998 and in 2000 they launched the Nordic Web Archive [24] project. Its goal was to create an access tool for the web collections. This project continued until the end of 2003 when the NWA software was released under an open source license.

In July of 2003, 11 national libraries, along with the Internet Archive, founded the International Internet Preservation Consortium (IIPC) [25].

Its mission is to "*acquire, preserve and make accessible knowledge and information from the Internet for future generations everywhere, promoting global exchange and international relations*" [26].

The IIPC began work on several fronts to clarify the issues and put forth requirements for the tools needed to achieve its goal. One of the requirements put forward, were those for a web crawler, suitable for harvesting and archiving the World Wide Web in a consistent manner.

The IIPC was however not able to jointly undertake the task of developing such a tool, despite some interest. Therefore, the Internet Archive chose to push ahead with its own web crawler project, Heritrix. Once this became clear, the national libraries of the Nordic countries (NNL) wanted to put their support behind the Heritrix project. NNL ultimately decided to send two developers to work on-site with the developers at the Internet Archive in San Francisco. The developers they sent were John Erik Halse of the National Library of Norway and Kristinn Sigurðsson of the National and University Library of Iceland.

The advantages of providing this form of support were many. For one, it would bring important technical expertise back home once it was completed. Having men on-site, rather than working remotely, would also avoid the many troubles inherent in distributed projects, reducing the load in project management.

The requirements document created by the IIPC for a web crawler was used as a basis in the Memorandum of Understanding signed by the IA and the NNL. It was clear from the start that not all of the requirements could be met within the timeframe given and so they were prioritized based on what the NNL felt were the most critical needs and what could realistically be achieved in six months.

Ultimately, this cooperation went extremely well. The only significant feature not addressed, was the ability to crawl continuously in an incremental manner. That is the subject we intend to tackle here.

2. Crawling strategies

It is important to clarify what exactly a web crawler is. In its most basic form, it is a piece of software that starts out with a (usually small) set of URIs[1], possibly just one. It then fetches the document that that URI refers to and extracts links to other URIs from it. Those newly discovered URIs are then fetched and links extracted from them and the cycle continues.

Of course it's not quite that simple in practice. First off, link extraction can be very difficult. HTML pages can usually be processed easily enough. Just access the *href* attributes on the *a* tags and the *src* attributes on some other tags like *img*. This in fact works quite well, but the problem is that not all links are encoded in these fairly simple HTML tags and attributes. For example, JavaScript may construct links on the fly. Links may also be embedded in other file types, such as PDFs, Word files, Macromedia Flash etc.

Another problem is politeness. Hammering a web server with requests is unacceptable behavior for a web crawler and would most likely cause it to be blocked by the servers in question. Some control over the number of requests made is therefore needed. Enforcing a minimum wait time between requests is a good way to do that, but unless you break it down to individual web servers, it would slow your crawl to a, well, crawl. Therefore, you need to maintain separate queues of URIs for each web server. That in turn can be tricky, because the same host name might be served by several web servers or one web server might manage several host names! Typically politeness is either enforced based on host names or IP numbers. Neither approach is perfect, but they generally limit the stress placed on web servers sufficiently.

There are many other considerations, such as obeying the robots.txt rules where they are found and correctly identifying character sets, especially in HTML documents that use multibyte encodings. Not to mention challenges related to automatically generated content, such as calendars that effectively lead to infinite amounts of content.

[1] URIs are a superset of the more familiar URL. Our use of the term URI is discussed in chapter 2.1.

More pertinent to the discussion at hand is the ever changing nature of the web. Crawling 'the web', is almost impossible. Any reasonable attempt will have to compromise on the scope of the crawl, either by skimming the top of websites, or selectively crawling only those sites that meet a certain criteria.

The web is growing very fast. Trying to keep up with it is almost impossible. Worse, it is also *changing* very rapidly. This means that content quickly goes out of date. It also means that if there is a long wait between the extraction of a link from a document and the time when that link is visited, the contents may well have changed from those that were being linked to. In other words, our archive will show a link between documents A and B even if document A had originally meant to link to content that was very different from what we eventually got when B was visited. This means that web resources must be visited within a reasonable amount of time from their discovery. During huge crawls, however, this can be extremely difficult, as the discovery 'frontier' keeps expanding.

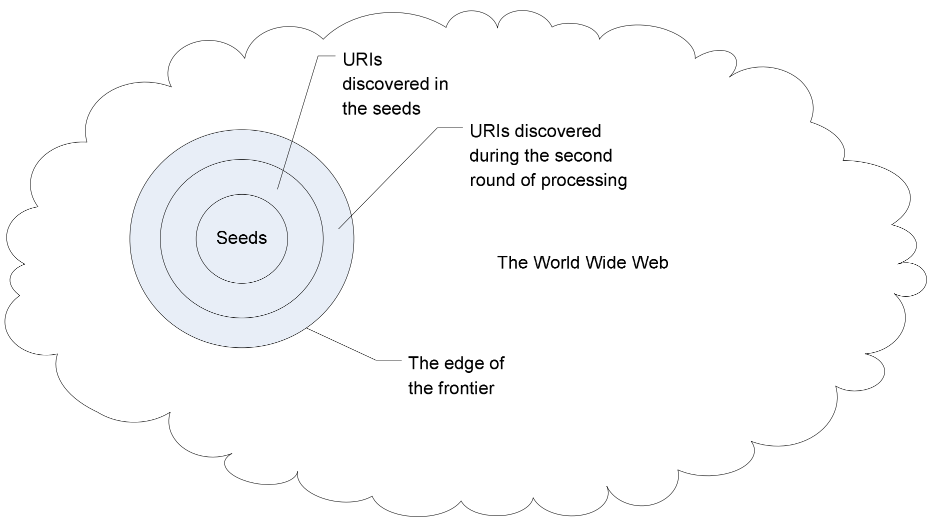

URIs discovered in the seeds
URIs discovered during the second round of processing
Seeds
The World Wide Web
The edge of the frontier

Figure 1. A conceptual illustration of the frontier concept in crawling.

Figure 1 illustrates the basic concept of a *frontier*. Initially the only URIs within it are the seeds, i.e. the initial set of URIs. During each round of processing more URIs are discovered and added to the frontier, pushing it out. Obviously the graphic exaggerates the sizes and simplifies the process. We know that the frontier will expand at varying rates depending on the number of URIs found in downloaded documents and based on the speed and politeness restrictions for different servers. Thus the frontier may rapidly expand on one website, while making less progress on another. It does however illustrate well how the crawling focus moves slowly away from the seeds, moving outwards with the frontier.

If there is no limit placed on the frontier it will continue to expand indefinitely or until hardware resources are exhausted. As we noted above, the frontier's scope will most likely be reduced, either by limiting its depth into each website, or limiting the number of websites. Possibly both.

The reasons for this are simple. We need to end the crawls at some point in order to repeat them. The Web's ever changing nature requires that crawls be repeated within a reasonable timeframe (typically within months, possibly much less). We generally do not have the luxury of crawling the same scope indefinitely. Limiting the number of documents downloaded from a single website is a common trade-off to allow more sites to be covered.

We are now at the topic of this chapter; crawling strategies. Depending on the purpose behind each crawl, there are a number of different crawling strategies. Each strategy has its own pros and cons and the choice of which to use must take into account the purpose of the crawl. What is the goal? To crawl the maximum number of sites within two months? To crawl a fixed part of the web (like a country domain) as thoroughly as possible within as short a time as possible? To crawl a small number of websites as often as possible with the aim of capturing *every* detail?

The ultimate goal can vary considerably. The ones mentioned are common types of crawls, but within each scenario there are many possible permutations. Restrictions on the type of documents to download, time of day when crawling can be performed, etc.

The Heritrix development team highlighted four types of crawls that they believed Heritrix needed to be capable of [1].

- *Broad crawling*
  Large, high-bandwidth crawls. The number of sites and pages collected are as important as the completeness of coverage for each site. An extreme broad crawl would be one that tried to capture the entire web.

Typically broad crawls will favor limiting their coverage of individual sites in order to be able to crawl a greater number of sites. Trading off website completeness in favor of getting better coverage of a larger section of the World Wide Web.

- *Focused crawling*
  Small to medium sized crawls using a selection criterion to limit the scope of the crawl. I.e. to restrict the frontier to a predetermined part of the web. This can be done in a variety of ways. For example, by using a list of allowed domains or search downloaded documents for keywords and stop performing link extraction once they are not found. The keywords would presumably be related to some topic and be rare enough to actually limit the scope. Ideally the crawls should achieve high completeness within their scopes.
- *Continuous crawling*
  These crawls are different from the above in that URIs are not simply 'completed.' Instead, the crawler continuously revisits all URIs within its frontier. This severely limits the speed with which the frontier can grow as the crawler must split its time between revisiting URIs and processing new ones. If we wish the revisits to always happen in a timely fashion, this will eventually place a hard limit on the size of the frontier where additional URIs can no longer be accommodated without falling behind on revisits. Continuous crawls therefore need to be carefully scoped.
- *Experimental crawling*
  This category encapsulates the myriad of atypical crawls that may be carried out to experiment with new techniques, protocols etc.

The above definitions do not exactly describe crawling strategies but rather crawling purposes. When we examine the strategies for broad and focused crawling we quickly note that a very similar approach can be used. The main difference lies in how the crawl is scoped. But they both use a basic *snapshot strategy*.

A snapshot strategy is what we have already described. Start with the seeds and work outwards. Once processed a URI will not come up again. If rediscovered it is rejected as a duplicate. Thus the crawl spreads out from the seeds and at any given moment the only activity is at the edge of its frontier.

Revisiting using a snapshot strategy relies on repeating a crawl. That is, once a crawl is complete, and this may take days, weeks and even months, the crawl is started again from the seeds. The web will, of course, have changed so the new crawl will not proceed in an identical manner, but should roughly cover the same content as the previous one.

Each iteration of a crawl captures a single snapshot of the web at that time, although in reality ‘that time’ varies from one document to the next. Stacked up these snapshots can provide a coarse picture of the changes that occur on the web over time. But unless the scope is quite limited allowing for frequent recrawling, a lot of intermittent changes will be missed. On the other hand if the scope *is* limited enough to allow, say, daily crawls, this will lead to a huge number of duplicates, since a lot of the documents within the scope will not have changed.

A snapshot strategy is therefore good at broad and focused crawls and can potentially capture a very large scope. However, it is not very good at capturing changes over time, or, in other words, performing a continuous crawl.

For continuous crawling we need an *incremental strategy*. The goal of continuous crawls is not (as such) to capture the largest possible scope or to crawl websites as thoroughly as possible. The objective is to capture the changes that occur on the website accurately. This means that within a single 'crawl' each URI will be visited multiple times. This is in stark contrast to snapshot strategies which assume that each URI will only be visited once during each crawl and between crawls all information is lost and new ones begin from the seeds.

Crawls using an incremental strategy will typically last longer, in fact ideally there is just one 'crawl' that runs as long as we remain interested in collecting the websites within its scope. It begins like a snapshot crawl and the frontier slowly expands as new URIs are discovered. Rather than discarding URIs that have already been crawled they are requeued. This means that the queue of waiting URIs can never decrease in size and will in fact, if the crawl's scope is large enough, continue to grow indefinitely.

This brings us back to the need for clearly defined scopes for incremental crawls. Assuming that the scope is sufficiently limited, the crawl eventually reaches a balance where it is cycling through the pool of URIs.

It retains information about past visits to the URIs allowing for duplicate detection and elimination.

However, other than that we haven't really achieved very much beyond what repeating a snapshot crawl would give us. In order to better utilize our resources we want to differentiate between URIs and visit them only as often as they change. We'll discuss the difficulties inherent in this later, but this kind of adaptation to the changes on the web allows an incremental crawl to make maximum use of its resources.

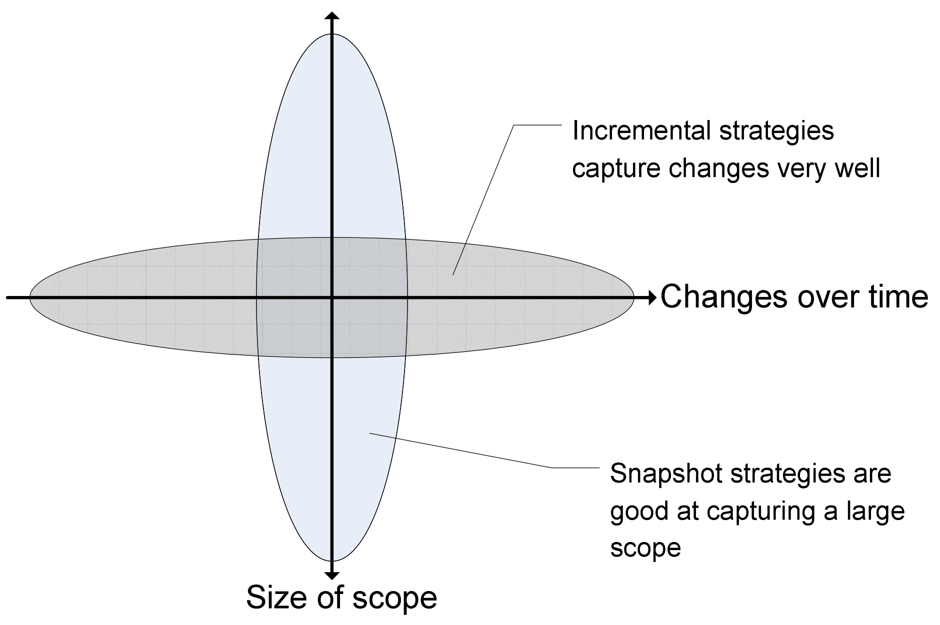

Incremental strategies capture changes very well
Changes over time
Snapshot strategies are good at capturing a large scope
Size of scope

Figure 2. The different emphasis of incremental and snapshot strategies

Figure 2 illustrates how the two crawling strategies differ in their emphasis. Of course you can do multiple snapshot crawls to also cover changes over time, as you could have many incremental crawls running to cover a larger section of the web.

In practice the two strategies complement each other. For example, when archiving a country domain, a snapshot could be taken of the entire domain at regular intervals. Perhaps 2-4 times a year. This does a good job of capturing the majority of the web without too many duplicates. The myriad of websites, such as news journals etc. that change daily could then also be archived using an incremental crawl that was limited to just those sites, and possibly just the 'top' of those sites. That is, ignoring their largely static archives and leaving those for the periodic snapshot crawls.

With sufficient resources it is possible to create an incremental crawler that can capture the entire web, thus removing any need for snapshot crawls. This is however, beyond the expected abilities of IIPC members. It is probable that large commercial outfits that crawl the web as a key part of their business will try to do this. However little information is available on the details of their crawls, and even if there were, it is doubtful how it would apply to archival crawling using more limited resources.

2.1 Terminology

Crawl. It is worth restating exactly what meaning we apply to this key concept. A crawl captures the processing of a set of seed URIs, link extraction, the processing of discovered URIs etc. until the scope has been exhausted. Each crawl starts without any information about prior crawls. Of course, experiences with prior crawl may have led to changed *settings*, but the crawler has no memory of those crawls as such.

A web crawler is then a software system capable of performing a crawl.

We have specified two crawl strategies, snapshot and incremental. Snapshot crawling is also sometimes known as periodic crawling [10]. The two crawl *types*, broad and focused, that Heritrix [1] offers are both based on a snapshot strategy. Frequently snapshot crawling and broad crawling is used interchangeably. Similarly incremental crawling and continuous crawling is often used interchangeable and even sometimes referred to as iterative crawling, although the last is to be discouraged.

Documents containing URIs are said to *link* to those documents that the URIs represent. Sometimes resources on the web are not complete 'documents' in the traditional sense, they require the loading of additional resources to fully render and these resources are specified by URIs. Typically this involves loading images in HTML pages. If the URI's placement within the document where it was discovered indicates that it is a resource required to render the originating document, that link is considered an *embed* and we say that the document embeds the resource the URI refers to. The distinction between links and embeds is important since we will often want to treat embeds differently, in order to ensure that we get all the parts of the logical document, e.g. an HTML page, represents.

This text also frequently refers to URIs. Uniform Resource Identifiers are a superset of the more familiar URLs or Universal Resource Locators and both refer to online resources, typically web pages and other web content. Heritrix, while only supporting URLs at the moment uses the more generic URI both in documentation and in implementation for future proofing.

The terms process, fetch, download and crawl a URI are largely synonymous. Processing and crawling refer to the entire process of downloading a URI from the web, performing link extraction, consider preconditions etc. and are slightly large in scope. Crawling is slightly more abstract then processing, which tends to consider the actual software processing. Fetching and downloading generally refers to just the physical act of downloading the resources over the internet and are used interchangeably.

Similarly, the terms URI, document, file and resources usually refer to the same thing. It is an internet resource identified by a URI, typically a URL referring to a webpage or other similar resource. The URI is the address or handle for the object and that term is generally used to abstract the type of resource in question, as they type quite frequently irrelevant. Document, file and resource are used interchangeably, we use those terms typically, when referring to some action on or property of the actual resource.

For most technical terms we prefer to use the same conventions as Heritrix. For example, a scope is a section of the World Wide Web that is to be crawled. This is also sometimes referred to as a segment. In the next few chapters we'll encounter many more such terms. While the meaning of most of them will either be clear from the context or explicitly explained, since we strive to adhere to Heritrix's conventions, the Heritrix user manual[8] and developer's documentation[7] are ideal sources for further clarification if needed.

3. Heritrix

Heritrix is the Internet Archive's (IA) open-source, extensible, web-scale, archival quality web crawler project [1]. The project was aided by two developers sent to work with IA in San Francisco by the National Libraries of the Nordic countries. This cooperation lasted for six months starting in late October 2003. The IIPC [25] had come up with a document detailing the desired behavior in a web crawler suitable for the kind of archival quality crawls that its members were interested in and this was used as a basis for determining what features to focus on. This included, among other things advanced configuration options allowing for great flexibility in the software.

It is generally understood that the demands placed on a web crawler change rapidly, just like the World Wide Web. Thus the emphasis on making this an open-source project to allow third parties to customize and/or contribute code and ensuring that the basic Heritrix framework is extensible so that such additional work could be integrated easily.

In order to achieve this, the Heritrix software can be divided into two parts; framework and pluggable modules. The framework provides generic control over a crawl. This includes providing a user interface, managing the running processes and an elaborate settings framework to simplify how crawls are configured. For the implementation of a crawl, a series of pluggable modules can be used to dictate each step. They are divided into four groups:

- Frontiers
- Processors
- Scopes
- Filters

Heritrix provides enough solid implementations of these modules to run a reasonably large focused crawl. Figure 3 illustrates how these components fit into Heritrix's overall architecture.

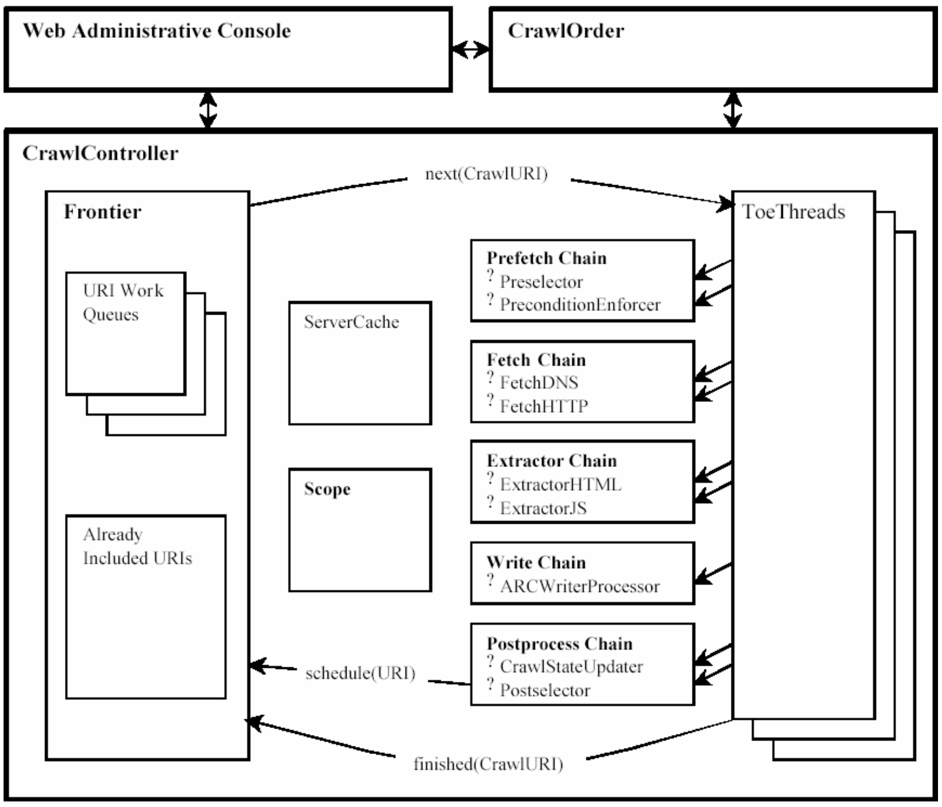


Figure 3. Heritrix's basic architecture [1]

3.1 CrawlController

The CrawlController object is at the core of the Heritrix framework. It essentially manages each crawl. When a crawl is being created (via the UI or command-line options) a CrawlController object is created. It in turn reads the settings provided to it and instantiates all the modules needed for the crawl. It also manages the ToeThreads (see 3.2), creating them and setting them to work on the Frontier.

All of the primary logs that Heritrix creates are also created here and other components access them via the CrawlController. In fact, almost every component has access to the CrawlController, either directly or via the settings handler which is discussed later.

Once the CrawlController has set everything up it waits for instructions to start the crawl. Once started it can be instructed to effect a pause, resumption or a termination of the crawl. This is generally only used by the UI. To enforce these orders it issues instructions to the relevant components.

When writing modules for Heritrix interactions with the CrawlController are rarely of any great significance. As it implements generic services, any new modules are likely to utilize them in a similar manner to existing code, i.e. accessing the logs and firing off events. It is just important that they (especially frontiers) correctly implement the methods that the CrawlController uses to control the crawl.

3.2 ToeThreads

Heritrix is multithreaded and uses a pool of worker threads to process the discovered URIs. These worker threads are called ToeThreads. The Heritrix FAQ [17] explains the logic behind this unusual name. "*While the mascots of web crawlers have usually been spider-related, I'd rather think of Heritrix as a centipede or millipede: fast and many-segmented. Anything that 'crawls' over many things at once would presumably have a lot of feet and toes. Heritrix will often use many hundreds of worker threads to "crawl", but 'WorkerThread' or 'CrawlThread' seem mundane. So instead, we have 'ToeThreads'.*"

ToeThreads request URIs from the Frontier. Assuming one is available, the Frontier will issue a URI. The ToeThread then applies a series of processors to the URI. Once all configured processors have been applied, the URI is returned to the Frontier and the ToeThread requests another URI to process.

It is the Frontier’s responsibility to pass the ToeThreads signals to pause, resume and terminate crawls. Typically, this is done when a ToeThread requests a new URI. The frontier can then either throw an EndedException or make the ToeThread wait until the crawl can be resumed.

3.3 The Settings Framework

Heritrix boasts a very comprehensive and flexible settings framework. It allows any module to publish a variety of configurable parameters. Furthermore, the values of these parameters can vary based on the current host and several other criteria.

Any module that needs to publish configurable parameters extends a basic configuration class, ModuleType. The ModuleType implements a *javax.management.DynamicMBean* [18] interface [7]. This means that the objects can be easily queried as to what settings they use.

There are three basic 'Types' settings in Heritrix: ComlexTypes, SimpleTypes and ListTypes.

The ComplexType may contain an array of other settings, including, possibly, other ComplexTypes. Variations on this include the MapType and the ModuleType which is the basic class for all pluggable classes in Heritrix.

SimpleTypes are settings that do not contain other settings. In other words strings, integers and such. It is capable of handling any Java type, but in practice it is used primarily for strings, numbers and booleans. It can also be assigned an array of legal values. The settings framework then enforces those restrictions.

The ListType behaves somewhat like a SimpleType, in that it does not contain other settings, but does contain a variable number of items. Implementations for lists containing Strings, Integers and Longs are provided.

By subclassing ModuleType, components can easily add configurable settings by constructing the settings types they need and adding them to their 'definition' in their constructor. A simple code example from the Heritrix Developers Documentation [7] follows:

```
public final static String ATTR_MAX_OVERALL_BANDWIDTH_USAGE =
        "total-bandwidth-usage-KB-sec";
private final static Integer DEFAULT_MAX_OVERALL_BANDWIDTH_USAGE =
        new Integer(0);
...

Type t;
```

```
t = addElementToDefinition(
    new SimpleType(ATTR_MAX_OVERALL_BANDWIDTH_USAGE,
    "The maximum average bandwidth the crawler is allowed to use.\n" +
    "The actual readspeed is not affected by this setting, it only " +
    "holds back new URIs from being processed when the bandwidth " +
    "usage has been to high.\n0 means no bandwidth limitation.",
    DEFAULT_MAX_OVERALL_BANDWIDTH_USAGE));
t.setOverrideable(false);
```

The code example illustrates the adding of an Integer setting, named "total-bandwidth-usage-KB-sec". Two constants are created for the settings name and default value. Declaring these as constants is considered good practices in the Heritrix project, rather than using the values directly in the constructor.

The constructor then invokes *addElementDefinition()* which is inherited from the settings framework's ComplexType class. The method accepts any Type object. Once added, the setting that the Type object stands for is a part of the component and the settings framework handles reading and writing the information to disk. The user interfaces also accesses these settings through the framework.

Several operations on the types allow for a good deal of further customization. It is possible to specify if the attribute is overrideable (more on that in Context based settings below) and if it is an expert setting (UI hides them by default). It is also possible to set constraints on the input, for example by using regular expressions that the input must match.

This means that any module added to Heritrix can easily expose any of its settings. Reading and writing it to disk is managed for it, and the user interface automatically adds it to its configuration pages. All the modules need to do is to extend the ModuleType object, or another object that does so. This greatly simplifies writing pluggable modules for Heritrix.

3.3.1 Context based settings

When a module reads from a setting, it can either do so in a 'global' context or based on a URI. Consider the following code example:

```
try {
   int value = ((Integer) getAttribute("setting", curi)).intValue();
} catch (AttributeNotFoundException e) {
   // Handle AttributeNotFoundException
}
```

In the above, the value of attribute *setting* is retrieved. The *getAttribute()* method accepts on one hand the name of the attribute and on the other hand a *context* object (curi). If the context is null then this relates directly to the base settings. However, if a URI (or rather a CrawlURI as we'll see later) is provided then the value of the setting may vary depending on the URI.

This is because the Heritrix settings framework uses an elaborate system of overrides based on host names. Basically, when looking up the settings for the URI "bok.hi.is", the settings framework first checks if it has been specified at the narrowest level, i.e. "bok.hi.is", if nothing is found there we recurse down to "hi.is," then "is" and finally check the global settings.

All settings have a global value, if none is found when reading the setting from disk the default value specified for it is used. This recursive lookup is implemented in such a way that no additional overhead is incurred.

Thus any setting or attribute that a module has, can have different values based on the URI it is handling at the time. In fact, the settings framework also has ‘refinements’ that work very much like the overrides. They base their decision on whether to return a different value from the global one, based on configurable criteria. These criteria may, or may not depend on the URI. Available criteria include one that uses time of day, another that uses the port on the host and a third that uses a regular expression match against the URI.

Refinements can be made on either the global level or any level of overrides. They are not inherited to overrides, so that if a refinement is specified on a global level and an override is created, the override takes precedence. Another refinement can then be made to the override.

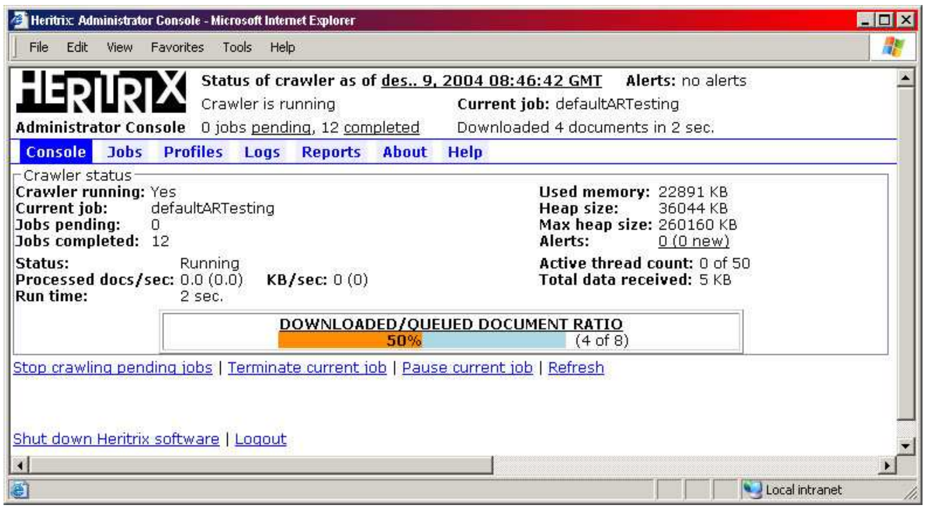
The refinements use special *criteria* to decide when the refinement values should be used instead of the default values. These criteria are fixed and new ones can not be added in a pluggable manner. New criteria require a change in the Heritrix framework. This effectively places them off-limits as far as customization goes, unless the objective is to get the new criteria accepted as a part of the Heritrix framework.

In order to take advantage of these features, modules must supply a URI to the *getAttribute()* method. If the setting in question should not be overrideable, it is possible to specify that when the Type object is created. The user interface will take note of this and not provide the option to override or refine the setting.

Furthermore, these settings can be changed at run time (although it is generally wise to pause the crawl when doing that). The user interface immediately updates the settings objects and they are written to disk. If modules wish to capture changes to the settings, and that is generally preferred, they need to ensure that they always access them via the *getAttribute()* method since using a stored class variable means that the change would go undetected. The nature of some settings makes it illogical for them to change at run time, such as the location of settings files. Modules can simply ignore changes to them if this is the case.

To summarize, the settings framework allows the operator to fine tune any exposed settings. As we see later, this is a very important property.

3.4 The Web User Interface



Heritrix: Administrator Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
HERITRIX
Console Jobs Profiles Logs Reports About Help
Local intranet

Status of crawler as of des.. 9, 2004 08:46:42 GMT **Alerts:** no alerts
Crawler is running **Current job:** defaultARTesting
Administrator Console 0 jobs pending, 12 completed Downloaded 4 documents in 2 sec.

Crawler status
Crawler running: Yes
Current job: defaultARTesting
Jobs pending: 0
Jobs completed: 12
Status: Running
Processed docs/sec: 0.0 (0.0) **KB/sec:** 0 (0)
Run time: 2 sec.
Used memory: 22891 KB
Heap size: 36044 KB
Max heap size: 260160 KB
Alerts: 0 (0 new)
Active thread count: 0 of 50
Total data received: 5 KB

DOWNLOADED/QUEUED DOCUMENT RATIO
50% (4 of 8)

Stop crawling pending jobs | Terminate current job | Pause current job | Refresh

Shut down Heritrix software | Logout

Figure 4. Heritrix's web user interface. Console – main screen.

While it is possible to launch crawls via Heritrix's command line, using existing XML configurations files, the software also provides a comprehensive user interface that is accessible via a web browser.

Heritrix uses an embedded Jetty server [27] to provide this functionality and thus the UI is closely integrated with the underlying engine.

The UI allows the operator to monitor crawls in progress, create new crawl jobs and profiles, review logs of ongoing and past crawls and provides access to comprehensive reports on ongoing crawls.

More importantly, the UI allows the operator to affect an ongoing crawl. This is done by manipulating the CrawlController and/or the settings. In addition to pausing, resuming and terminating ongoing crawls, the UI allows the operator to change the settings used for the ongoing crawl. Not all changes are allowed, for example, it is not possible to change what components are used. However, the configurable parameters that are used to set their behavior can usually be changed on the fly. For some changes it is recommended that the crawl be paused while they are made.

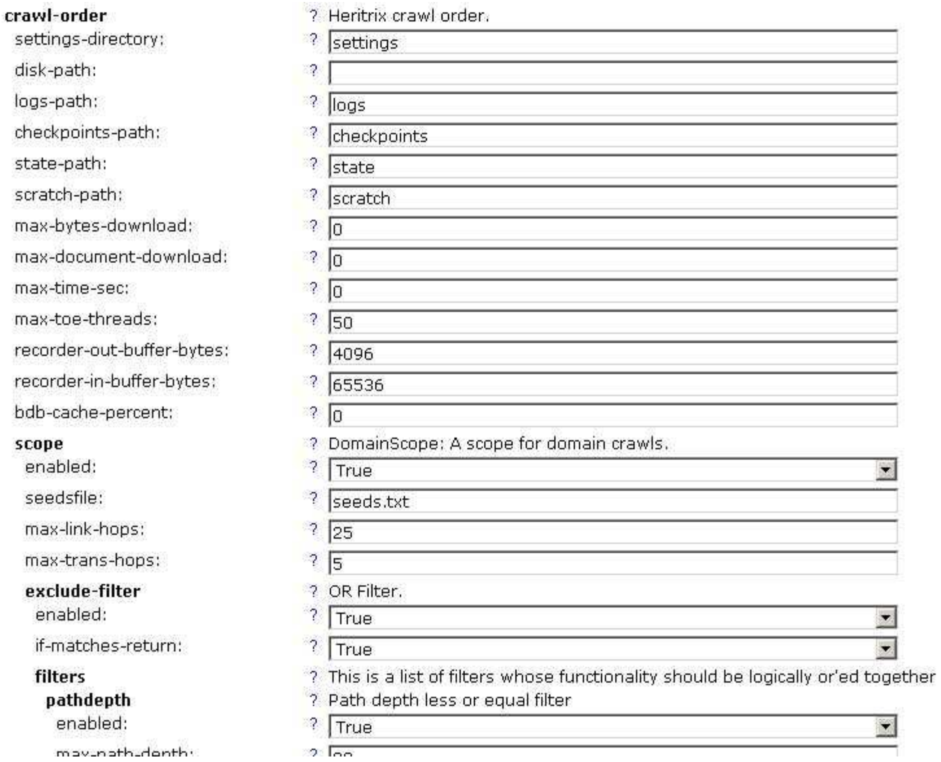
3.4.1 Jobs and profiles

The concept of 'jobs' in Heritrix is tied tightly to notion of 'crawls.' A job encapsulates a single crawl. That is to say it contains all the settings needed to properly configure a crawl in Heritrix, plus useful state information about the crawl. Profiles are essentially the same as jobs, in that they contain all the settings needed to configure a crawl, but are considered to be templates for jobs.

Typically, when creating a new crawl a profile that describes it is created. Using that profile numerous jobs can be launched, with varying settings and seeds if needed. Thus each crawling strategy being utilized can have its own profile, saving considerable time when launching new crawls.

All jobs and profiles are created based on existing profiles or jobs (Heritrix provides a simple default profile). Thus the user interface clones the settings of the parent and allows the user to edit it. The actual settings are stored on disk in an XML file. While it is possible to edit them directly the user need never bother with it if using the UI. This is because the UI enables the user to edit any part of it and also does a good deal of error prevention.

The most important part of editing a job or profile is done on the 'Settings' page (Figure 5). The actual settings framework arranges the modules into a tree-like structure with an object called CrawlOrder at the root. This allows the UI to recursively make its way down the settings tree, printing out the configurable attributes of each module. The UI creates combo boxes where legal values have been defined or Boolean values are expected and also handles simple lists, allowing modules to specify almost any type of setting. When editing profiles the UI will warn if any restrictions have been placed on the input and are not being met. It will not allow such input in jobs.



crawl-order	? Heritrix crawl order.
settings-directory:	? settings
disk-path:	?
logs-path:	? logs
checkpoints-path:	? checkpoints
state-path:	? state
scratch-path:	? scratch
max-bytes-download:	? 0
max-document-download:	? 0
max-time-sec:	? 0
max-toe-threads:	? 50
recorder-out-buffer-bytes:	? 4096
recorder-in-buffer-bytes:	? 65536
bdb-cache-percent:	? 0
scope	? DomainScope: A scope for domain crawls.
enabled:	? True
seedsfile:	? seeds.txt
max-link-hops:	? 25
max-trans-hops:	? 5
exclude-filter	? OR Filter.
enabled:	? True
if-matches-return:	? True
filters	? This is a list of filters whose functionality should be logically or'ed together
pathdepth	? Path depth less or equal filter
enabled:	? True
max-path-depth:	? [illegible]

Figure 5. Heritrix’s settings

The UI then offers two pages to edit which modules are being used. The first, "Modules" allows the user to set which frontier implementation to use, which scope to use and what processors should make up the processing chain and in what order. New modules can easily be added by putting JAR files in Heritrix's class path and ensuring the *.options* files are included in them [7].

The "Submodules" page allows the user to add filters to any module that accepts filters. All processors do this, and filters can be placed on them to make URIs skip over them when the filter criteria are met. Scopes generally also allow filters and any module can opt to allow specialty filters. It also enables the user to set URL canonicalization rules. These are applied to discovered URIs to, for example, strip session IDs, etc.

Finally, the UI allows the user to specify overrides and refinements for the settings. If an override or refinement is created, the UI offers the option of entering new values for any overridable setting. It also allows the addition of further filters; other modules can not be added and no modules can be removed in an override.

It is worth pointing out that each instance of Heritrix can (and usually does) manage several jobs at once, however only one of them can be active, or crawling. That is only one crawl can be actively performed at once, all other jobs wait in a simple queue. Jobs are also retained after completion to allow access to their information and so that new jobs can be created based on them.

3.4.2 Logs and reports

The UI also gives access to most of Heritrix's logs i.e. all the logs that are related to one crawl. There is also a general log to capture various outputs that are not directly related to a single crawl job. This log also captures some output from third party modules used by Heritrix.

The most interesting of these logs are the crawl.log and progress-statistics.log.

The crawl.log contains a single line for each URI that is processed, unless the processing leads to a retriable error, such as missing preconditions (DNS lookup or robots.txt information). The log contains a timestamp, the processing result code[2], size of the downloaded document, the document URI, the URI of the document that contained the link to this URI and a few other items. At the end of this line is an annotation field. This section allows modules to write arbitrary data about this URIs processing to the log.

The progress-statistics.log is compiled by a StatisticsTracker object. Supposedly it should be pluggable, but in practice this has proven difficult since the UI relies on this object's presence. It receives

[2] The result code is the HTTP status code if one was received, otherwise an appropriate code determined by Heritrix. See [8] for an overview of defined codes.

notifications about the completion of each URI processed and at fixed intervals prints progress statistics that include how many URIs have been discovered, crawled, the number of document processed per second etc.

Other logs are primarily focused on capturing various errors. The UI can also flag alerts that have been raised, so most serious errors during a crawl should be clearly noticeable.

Using the StatisticsTracker, the UI is also able to provide a variety of other information. Most of the information on the front, ‘Console,’ page is from it. Each job also has a ‘Crawl report’ that contains major progress data and details on the number of documents per HTTP status code, host and file type.

Additionally, the UI allows access to several module specific reports that are only accessible while a crawl is in progress. The most important is the Frontier report, which details the status of the frontier. The exact nature of the report varies based on the implementation of the frontier. Other such reports include a Threads report and Processors report, the latter interrogates all the processors for their individual reports and displays the combined results.

Together this means that the UI provides excellent means of monitoring the progress of any crawl. Once a crawl has ended the logs remain available, but most other info is dumped to disk as text reports and are not accessible directly from the UI.

3.5 Frontiers

The Frontier being used is the most essential part of each Crawl. While the CrawlController manages generic details about the crawl, such as access to logs, creating ToeThreads etc., the Frontier manages the *state of the crawl*.

In our earlier discussion of crawling strategies we discussed the abstract concept of a URI frontier, i.e. any URI behind it has already been discovered, the ones at the edge of it still need to be processed and we have yet to encounter the ones outside it. Frontiers in Heritrix are so named because they effectively codify this concept.

Each Frontier knows what URIs it has encountered. As they are discovered (or loaded as seeds) they are scheduled with the Frontier. It then maintains a list of what URIs remain to be crawled and in what order. The ToeThreads request URIs from the Frontier, process them, schedule discovered links and return the processed URI back. This means that the Frontier object entirely controls the progress of the crawl by how it issues, or does not issue, URIs.

The Heritrix framework does not provide one definitive Frontier. Rather, it specifies an API for Frontiers via an interface. It then provides some useful implementations of these, but allows for additional Frontiers being created to replace them. Frontiers must implement this interface, but they should also subclass the ModuleType class to gain access to Heritrix's settings framework, discussed earlier.

Let's quickly review the most essential methods that the Frontier interface specifies, with an eye towards understanding the requirements placed on Frontiers.

The two key methods are *next()* which returns a URI for crawling, and *schedule()* which accepts a URI and schedules it for later crawling. Additionally the *finished()* method is used to signal that the processing of a URI that the Frontier issued has been completed.

The entire responsibility of ordering the scheduled URIs is placed on the Frontier. It must, for example, implement a politeness policy to prevent Heritrix from crawling any web server too aggressively. This typically means maintaining numerous separate queues for each politeness unit, usually a host name or IP number so that URIs from it can be withheld for a while after each time the web server is contacted. If separate queues were not maintained the crawl would be unable to proceed on other web servers during these politeness waits.

The exact implementation of how a Frontier issues scheduled URIs dictates the crawling strategy. Any Frontier can order its URIs based on any criteria that suits the purposes of the particular crawl in progress. Use of some relevancy indicator to sort URIs for processing could, for example, be used on topical crawls[3]. Also it is common to quickly issue URIs discovered embedded in documents, that is pictures and similar files

[3] Topical crawls are crawls focused on collection pages related to a specific topic or event.

displayed within the originating document, rather than links to separate files.

The Frontier also maintains a number of running statistics, accessible via several methods specified by the Frontier interface.

- *Discovered URIs*
  Total number of unique URIs that have been scheduled.
- *Queued URIs*
  The number of URIs waiting to be issued.
- *Finished URIs*
  Total number of URIs that have been issued and finished processing. URIs that are returned with retriable errors don't count as they are reinserted into the queue (and thus the total number of queued URIs)
- *Successfully processed URIs*
  The total number of URIs that where processed successfully. That is, a response was received from the web server. The response may have been a 404 or other HTTP error codes.
- *Failed to process URIs*
  Total number of URIs that could not be processed for some reason. Typically the web server could not be contacted, but there are many other possible reasons.
- *Disregarded URIs*
  The number of URIs that have been disregarded for some reason. Usually this is because a robots.txt rule precludes a fetch attempt.
- *Total bytes written*
  The sum total of the byte size of all downloaded documents

A lot of this data is also maintained by the StatisticsTracker and the UI uses it to access these values, but during the crawl, the Frontier provides the most up-to-date information on these values.

While most of these statistics are fairly straightforward, some, the number of queued URIs in particular, assume a snapshot based crawl strategy. After all, with an incremental strategy, there are always (in theory) an infinite number of URIs waiting to be processed (if we count not only the different URIs, but also the many revisits). In practice the best approach is to simply equate the number of queued URIs with the number of discovered URIs. We'll discuss this in more detail later. It is brought up here simple to point out the fact that the Frontier interface was developed at a time when only a snapshot based Frontier implementation existed and it bears some evidence of that.

The Frontier interface specifies several methods to allow the CrawlController to notify the Frontier of a change in the crawl state, that is to pause, resume and terminate the crawl or *pause()*, *unpause()* and *terminate()* respectively. The Frontier is then responsible for obeying the new state by withholding URIs, resume issuing URIs or throwing an EndedException whenever a ToeThread requests a URI.

The Frontier is responsible for firing off CrawlURIDispostionEvents whenever a URI completes processing. This is done via a method on the CrawlController. This allows any interested module to monitor the progress of the crawl. Mostly used by the StatisticsTracker.

There are also a number of methods that deal with monitoring the Frontier and manipulating it at run-time. Of the former, the *report()* method is the most significant as it should return a human-readable report that is displayed in the UI. The *oneLineReport()* method is a much abbreviated version of this. There is also a complicated system for iterating over all the URIs within the Frontier.

For manipulating the Frontier there is a way to delete URIs, either by name or regular expression. There is also a method for loading recovery files if the Frontier writes a transaction log that can be replayed.

While it is not enforced by the Frontier interface, the Frontier should apply URI canonicalization rules. URIs often contain meaningless data, such as session IDs that need to be removed. The canonicalization system that Heritrix has applies user specified fixes to them. However, while the canonicalizers themselves are handled by the framework, the Frontiers must *apply* them to the URIs. This is typically the first thing done after the *schedule()* method is invoked and before the URIs is evaluated for existing duplicates.

This is not an exhaustive overview of all the things a Frontier handles. Essentially, a Frontier directs the progress of the crawl and there is a lot of things it *can* do. The above is most of what it has to do.

3.5.1 HostQueuesFrontier

The HostQueuesFrontier was the first Frontier created for Heritrix. It has since been deprecated, but its design greatly influences the ones that have followed it.

As the name implies it is based around a number of host queues. That is each host has its own named queue. As discussed earlier this is a good way of enforcing politeness.

The HostQueuesFrontier queues were simple *first-in-first-out* queues that were custom written for Heritrix, including its disk backing ability. That is, writing parts of the queues to disk.

Each host queue was assigned a state, initially the states were:

- *Ready*
  URIs from it can be issued.
- *Snoozed*
  URIs from it can not be issued until the 'wake up' time is reached.
- *Empty*
  Does not contain any URIs.
- *Busy*
  A URI from this queue is currently being processed.

But as the crawling strategy became more complex the following were slowly added:

- *Inactive*
  An inactive queue, while having URIs that are ready for crawling, is being held back. This is to improve broad crawl behavior. I.e. focus on a limited number of queues at a time, just enough to fully utilize the machine resources. This helps ensure that URIs from the same host are crawled within a more reasonable timeframe than round-robining through all the host queues.
- *Frozen*
  In practice this isn't used. But the ability was added to mark queues as frozen to signify that they have been put aside. That is, the crawler will not continue with them until the operator intervenes. This is to stop processing of hosts that exhibit some form of bad behavior.

The HostQueuesFrontier is a simple snapshot Frontier. Once a URI has been processed, only a hash of it is stored and new URIs are compared to the list of existing hashes. If no conflict is found, they are scheduled. If there is a conflict, the new URI is considered to be a duplicate and is discarded. The only exception from this is with robots.txt and DNS lookups, they are repeated at fixed intervals. So URIs that are marked as prerequisites are allowed to be scheduled again. This doesn't constitute a real incremental strategy since the prerequisites are only recrawled if the host they refer to is still being crawled. Then, once the existing information expires on robots.txt or DNS information, this triggers the *rediscovery* of the preconditions. A true incremental strategy would revisit them as a matter of policy, not in response to this form of rediscovery.

Despite the name, the 'host queues' need not be based on hosts. Queues are keyed, or named, either by the host name of the URI, or its IP number. To do this there are two *QueueAssignmentPolicies*. The *HostnameQueueAssignmentPolicy* is the default one but the user can configure the Frontier to use the *IPQueueAssignmentPolicy* instead. Any newly discovered CandidateURI is fed to the configured policy and it returns a queue key. This is stored in the CandidateURI and is accessible via the get and set methods for *class key*. An interface, URIWorkQueue defines these queues, and it accepts the class key.

One of the primary problems with this Frontier was the overhead incurred by each queue. Each was represented by an object that required memory and could not be backed out on disk without completely redesigning them. This made truly broad crawls all but impossible since they would exhaust the available memory at some point, as the number of encountered hosts grew. This led to the development of the BdbFrontier.

3.5.2 BdbFrontier

The BdbFrontier relies on the Berkley DB Java Edition [9], for object serialization, rather than the custom written queues of the HostQueuesFrontier. The significant advantage inherent in this was that the queue itself could be stored in the database, thus allowing for many more queues. The limiting factor became disk space, which is much more abundant than memory.

By using the Berkley DB, the concerns of writing to and reading from disk, including caching and other related issues, were effectively removed from the Frontier and relegated to a third party tool that was written expressly for the purpose of managing a large amount of data, further improving performance. We will discuss the Berkley DB in more detail later, in chapter 6.2.

Aside from the much improved handling of queues, the BdbFrontier is very much like the HostQueuesFrontier and implements essentially the same snapshot strategy. It does improve on several points, including the ability to add a 'budget' for each queue. The cost of each URI is evaluated based on the selected policy and once a queue’s budget is exhausted no further URIs are crawled. Also, if using the 'hold queues' option, that is focusing on a small number of queues at a time, the queues remain active for a fixed amount of 'cost,' after which time they become inactive until it is their turn again.

The cost/budget addition to the Frontier enhances the possibilities in configuring broad crawls. As was discussed earlier, broad crawls typically trade-off on the depth with which they crawl each site, in favor of crawling more sites.

3.5.3 AbstractFrontier

The AbstractFrontier was developed alongside the BdbFrontier and is meant to be a partial implementation of a generic Frontier. That is, it is meant to implement those parts of the Frontier that are largely independent of the crawling strategy being implemented. This includes, to various degrees, management of numerous general purpose settings, statistics, maintaining a recovery log and more. Included are numerous useful methods to evaluate URIs etc. It also handles URI canonicalization.

The BdbFrontier subclasses the AbstractFrontier. The AbstractFrontier is provided to simplify the creation of new Frontiers by doing all the routine work here, allowing new Frontiers to focus on their crawling strategies, rather than having to tackle all the mundane aspects of a Frontier. Furthermore, this can simplify code maintenance in the future, both if a change to the Frontier API is introduced, and also if new or improved functionality is developed that should be applied to most or all Frontiers.

3.5.4 Making other Frontiers

The BdbFrontier is the main Frontier that is provided with Heritrix. The HostQueuesFrontier and a specialized version of it called DomainSensitiveFrontier are considered deprecated.

Creating a Frontier need not be very complicated. In fact the Heritrix Developers Guide [7] shows how a very simple, yet fully functional, Frontier can look. At the most basic level, the Frontier is just a simple *FIFO* queue. The example enforces global politeness, that is a pause after each URI. This works, of course, but it isn't especially efficient. And that is where the complication comes in; we need to create efficient Frontiers that perform a crawl that conforms to our needs.

The BdbFrontier has focused on improving the performance of snapshot crawls, whether they are focused or broad. In fact, it could also (and perhaps better) be described as a snapshot Frontier. While it is certainly possible that one might create another Frontier simply to improve on the BdbFrontier's performance in snapshot crawling, that would, if successful, typically make the existing Frontier obsolete.

However, we already know that snapshot crawling is not the only type of crawling. This flexibility in Heritrix architecture that allows us to create additional Frontiers means that we can implement new ones that are optimized based on the demands of a different crawling strategy.

It is of course conceivable to create a Frontier that encompasses all (known) crawling strategies. In fact the BdbFrontier already handles a number of variations on the snapshot strategy. Doing that would however risk trading-off performance in one type of crawl for the ability to perform another type of crawl. In the best case scenario, while no performance penalty is incurred, the code's complexity would increase markedly, making the software far more difficult to maintain.

3.6 URIs, UURIs, CandidateURIs and CrawlURIs

Currently, Heritrix only supports URLs however it always uses URIs for future proofing [7]. A URI class is provided in the Java Foundation Classes [28] that can represents a URI in programs. This however has some bugs and complies very strictly with RFC2396 [29] making it unusable in practice.

Instead, Heritrix uses a UURI, or "Usable URI" that subclasses the URI class from the Apache Commons project (*org.apache.commons.httpclient.URI*). The UURI class adds some processing of the URI so that any UURI that is created successfully can be assumed to be "useable." The URI must be legal and certain superficial similarities are removed [7] (to improve duplicate detection).

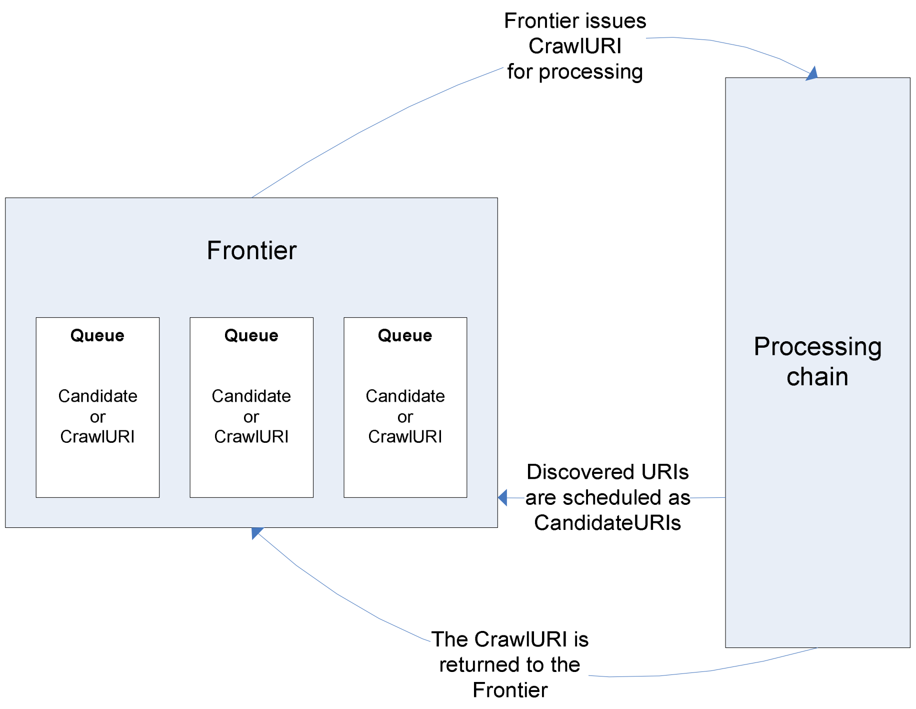
This is only half the story. Heritrix frequently needs to attach a large amount of meta-data to the URI. For this purpose two additional URI classes have been introduced, they do not subclass URI or UURI, but rather contain such an object.

The CandidateURI class is subclassed by the CrawlURI class. Roughly, a CandidateURI object holds the data needed from the time a URI is discovered and until it should be processed, the CrawlURI then adds the data needed in order to complete the processing. Since many URIs are rejected as duplicates or deemed out of scope, the use of an interim CandidateURI is seen as a way to save memory. In practice this has been limited. The CrawlURI originally contained a hash map that could store any arbitrary data. This map has since been moved to the CandidateURI and access to it limited to accessors on the object. The CandidateURI however remains and URIs that are scheduled with the Frontier are always wrapped in a CandidateURI.

An interface, *CoreAttributeConstants*, declares a number of constants that represent keys for various data typically stored in the CandidateURI and CrawlURI. This provides a central source for the keys and any class using them should always refer to those constants.

Figure 6 illustrates different uses of CandidateURIs and CrawlURIs. URIs being processed are wrapped in a CrawlURI. A UURI is created for newly discovered URIs and that is then wrapped in a CandidateURI to preserve such information as parent URI, path from seed, etc.

The CandidateURI is scheduled with the Frontier and it depends on the Frontier when (if ever) the CandidateURI becomes a CrawlURI. The Frontier may do this immediately, after determining that it is not a duplicate or, at the latest, just as the URI is being issued for processing.



Frontier issues CrawlURI for processing
Frontier
Queue
Candidate or CrawlURI
Queue
Candidate or CrawlURI
Queue
Candidate or CrawlURI
Processing chain
Discovered URIs are scheduled as CandidateURIs
The CrawlURI is returned to the Frontier

Figure 6. CandidateURI and CrawlURI lifecycles

CandidateURIs and CrawlURIs have a scheduling directive associated with them. This directive indicates that a URI must be crawled before any other URI with a lower scheduling directive and is primarily used to ensure that preconditions are fetched before the URI that needs them is tried again. It is also used to give preferential treatment to embedded URIs. Frontiers need to obey this directive or risk having the crawl dead end since prerequisite URIs are not being fetched prior to the URIs that require them.

The scheduling directive hierarchy is as follows, starting with the lowest:

- NORMAL
  Default scheduling directive, no preferential treatment. In practice most URIs will have this set.
- MEDIUM
  Generally used for preferencing embeds.
- HIGH
- HIGHEST

The last two are rarely used, although the algorithm for setting the scheduling directive on prerequisites simply sets it one level higher than the URI that requires the prerequisite. This means that in practice the HIGHEST level should always be reserved for prerequisites. In fact, there is little, if any, need for more than two levels for any other scheduling. If more complex ordering is desired in the queuing of URIs, additional fields should be added to support that.

3.7 The processing chain

As discussed earlier, the actual processing of URIs is handled by the ToeThreads. They request a CrawlURI from the Frontier and then subject it to a series of *Processors*.

The processors are pluggable classes in Heritrix, and technically you could create a crawl with no processors, although it wouldn't do much except run quickly through the seeds, returning them each with a failure code.

The processing chain is split up into five sections based on the different tasks that are going to be carried out:

1. *Pre-fetch*
   This should contain any processor that needs to run before the software attempts to contact a web server and download the relevant document. Typically, this includes rechecking the scope of the URI (in case it has been modified since the URIs discovery) and ensuring that all preconditions are met.
2. *Fetch*
   In this part, processors that fetch the document are stacked. Different processors are configured for each protocol that is supported. They should each pass on any URI whose protocol they do not understand. Effectively meaning that only one fetch processor is applied to each CrawlURI. Currently only DNS and HTTP fetchers exist, but plans exist for an FTP fetcher as well.
3. *Extractor*
   Here the numerous processors, that extract links from downloaded documents, are applied. Each processor typically handles a particular type of document. Some, more generic extractors may attempt to extract links from multiple file types, but can avoid double work by

taking note of a 'link extraction completed' flag that an earlier link extractor raised.
This section of the processing chain is also used for other analysis processors that should be applied after the document is downloaded and before we store it on disk

4. *Write/index*
   Here processors are placed that write the downloaded documents to permanent storage.
5. *Post-processing*
   The final section holds processors that do "clean-up" and ready the URI for its return to the Frontier. Under certain conditions, earlier processors, especially in the pre-fetch section, may abort the processing of a URI if some condition isn't met. In those cases the processing chain jumps to this section. This section also holds a processor that turns the discovered links into CandidateURIs and feeds them to the Frontier.

Each section can contain any number of processors. It is also relatively easy to create additional processors. By subclassing the *Processor* class, the only thing that needs to be done is create suitable Constructors to comply with the settings framework and override the *innerProcess()* method that accepts a CrawlURI. Within the method any arbitrary code can be executed and it will have full access to the CrawlURI being processed.

The only communication between processors and other modules are via the CrawlURIs. Processors modify them and can also attach any arbitrary data to them, keyed by a string value. This enables custom written modules to pass information to each other via the CrawlURI without requiring any customization of the CrawlURI.

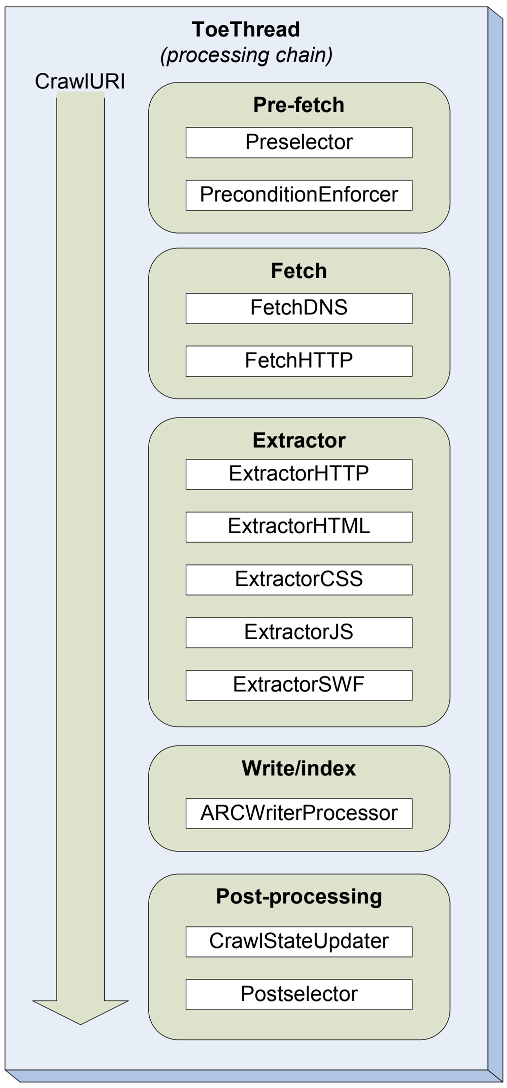


Figure 7. A look at a typical processing chain. Each section contains the default processors that Heritrix typically uses. These include fetch processors for DNS and HTTP protocols, extractors for HTTP headers and HTML, CSS, JavaScript and Macromedia Flash documents. The writer processor outputs ARC [7] files, commonly used for archiving web resources. There are several other processors provided with Heritrix that are not included in the default configuration. The CrawlURI is passed through the processors, one by one in order unless one of them preempts the processing, in which case the CrawlURI passes directly to the Post-processing chain.

3.8 Scopes

The scope of a crawl is defined by a scope object in Heritrix. This is a pluggable module and different implementations focus on different ways of limiting the scope. From unlimited scopes for broad crawling, to scopes limited to certain domains, hosts or other criteria.

The task of limiting a crawl's scope is essentially unrelated to the crawl strategy. It is true that different crawl strategies should, generally, be scoped differently, but this does not follow those lines exactly. For example, broad scopes typically only limit the *depth* of the crawl, that is how many link hops from the seed the Frontier is allowed to progress. On the other hand focused crawls usually limit *what domains or hosts should be included*. An incremental crawl might however want to do *both*.

The scoping model in Heritrix is not perfect, and work continues to improve it. It does however offer a sufficiently wide range of options to create an appropriate scope for any crawl strategy that we have discussed in the previous chapter.

The following types of scoping is currently available in Heritrix:

- *BroadScope*
  As the name implies it is intended for broad crawls. Has configurable limits on the depth of the crawl.
- *DomainScope*
  Limits the crawl to the domains found in the seed list. That is any domain that a seed refers to is judged to be inside the scope, others are not.
- *HostScope*
  Similar to DomainScope but limits the crawl further, to just the hosts that the seeds refer to.
- *PathScope*
  A further refinement on the HostScope, this limits the crawl to only a specific subsection of a host, denoted by a path segment. E.g. the seed `http://my.com/thisonly/`, would judge any URL beginning with `my.com/thisonly/` as being within the scope, but not other parts of the host.

- *FilterScope*
  This is a specialty scope that is by default the same as BroadScope, but any number of filters can be added to it.
- *SurtPrefixScope*
  SURT (Sort-friendly URI Reordering Transform) [7] rearranges the typical "backwards" hostnames like *crawler.archive.org* into *org,archive,crawler*. This scope allows the operator to specify a hostname surt-prefix and only those URIs who match it will be judged in scope. For example the prefix *org* would accept the earlier example of *crawler.archive.org*, but would not accept *organization.com*.

Additionally, all scopes allow for transitive includes, i.e. URIs that are outside the scope, can be ruled inside if there are special circumstances. For example, a DomainScope might want to capture embedded offsite images. This requires a transitive include that rules URIs that are embedded within a document, but normally out of scope, to be included.

Scopes also, optionally, use filters to fine tune their behavior and avoid certain traps.

3.9 Filters

Like scopes, filters are still being worked on in Heritrix. Their basic function is to accept a URI and render a *true* / *false* verdict as to whether it matches the filter. They can be applied to scopes and processors and all modules can also accept filters for specialized purposes if they so choose. For example, the HttpFetcher processor uses a filter to determine if a download should be terminated after the HTTP headers have been downloaded. Like all processors it also has a general purpose filter that decides if the processor should be applied to the URI at all.

The main problem with filters has been their ambiguity. The effect of a filter rendering a *true* verdict depends on where the filter has been placed. Sometimes it will mean that a URI is accepted, sometimes that it is rejected.

Heritrix offers several general purpose filters as well as some highly specialized ones, mostly used in the scopes. Among the more general ones are filters that use a regular expression and compare the URI string and the content type to it. More specialized filters include ones that check the path depth, that is count the number of slashes in the path and one that checks for repeating path elements. Both of those are added to scopes to prevent certain crawler traps that cause an infinite series of links.

Creating custom filters is relatively straight forward, as with creating processors. Simply subclass *Filter*, add an appropriate constructor and override the *innerAccepts()* method. The method accepts an object, rather than a CrawlURI. In fact the object is almost always a CrawlURI or a CandidateURI and the filters need to typecast the objects before evaluating them.

4. The objective

Heritrix already has an effective snapshot strategy. While work continues to extend its capabilities, it can never offer anything more than simple repeats of crawls for incremental crawling. While this is of some use, many IIPC members, many of the Nordic Countries in particular, wanted the ability to crawl continuously while discarding duplicate records.

This desire was at least partially because of familiarity with an older web crawler, Nedlib [13] that operated along those principles. Active development has long since stopped on Nedlib and continuing to use it was not an option. Heritrix was clearly going to be the new standard for web crawling, in most of the countries that had been using Nedlib, certainly in Iceland.

The obvious solution, therefore, was to create an addition to the Heritrix framework that would enable continuous crawling. The idea was discussed during a Heritrix workshop held in Copenhagen, Denmark at the Royal Library on June 9th-11th 2004. In attendance were representatives from the Internet Archive and several National Libraries (the author of this thesis represented the National Library of Iceland at the proceedings). Together they identified the following four key reasons for an iterative crawler:

1. Save storage costs
2. Crawl faster
3. Reduce user-visible redundancy
4. Politeness

On closer examination, points 1 and 3 are essentially the same. By detecting duplicates and discarding them, significant savings can be made in storage costs. Of course, the exact amount varies depending on the frequency of visits compared to actual change rates. Furthermore, if duplicates are discarded immediately the access interface will not show numerous 'versions' that are all identical.

Point 3 could actually be achieved by implementing document comparison in the access tool. Using it to filter out duplicates. This could however increase response times, requiring more processor power for the access software or allowing the service to be degraded. Similarly, point 1 could by achieved via post crawl analysis that discarded duplicates, but that would require additional computational resources to go through the collection as well as temporary storage of extraneous material.

The most efficient method seems to be to immediately detect duplicates during the crawl. The documents are already being processed to a significant degree and, more importantly, duplicate detection at crawl time allows us to tackle points 2 and 4.

Points 2 and 4, while stated as separate reasons, are actually functions of the same constraint. For each host we can only crawl so many documents per second. The exact values depend on variable politeness settings, but generally no more than 10 (and often much less) documents can be fetched from a host each second. If we wish to continuously crawl a host as frequently as possible with the aim of capturing *every change*, we would need to crawl as fast as the politeness restrictions allow us. The total number of documents on the host would then determine how frequently each document was visited. On very large sites, this could be quite rarely. While most documents do not change that often, a certain subset may change very frequently.

The objective therefore is to focus on those documents that change often, while visiting the others less frequently. By doing this we can both crawl faster, essentially by making each iteration smaller and at the same time limit the stress our crawling places on the web servers.

The obvious way of performing this distinction between rapidly changing documents and those that do not change very often is to monitor if a change has occurred between visits. If it has, we are not crawling fast enough. If no change has occurred, we can crawl slower.

Our objective was, therefore, to create the necessary add-on modules for Heritrix to achieve a crawl that adapted its rate of visit to particular documents based on observed change rates. In doing so we also hope to clarify the concept of continuous crawling versus snapshot crawling.

4.1 Limiting the project

The goal was to create an add-on to Heritrix that could handle long running crawls. One of the most difficult aspects of Heritrix has been scaling up the scope of the crawls.

The problem is effectively twofold. As the number of documents increases, additional resources are required to manage them. This is especially true as the number of hosts (or whatever base unit is used for politeness) grows since each forms an independent queue. On the other hand, with increasing number of documents, the wait time from discovery to crawl increases since bandwidth and processing power is limited.

For broad crawls, the latter is largely a non-issue. Various tactics can be employed to ensure that closely related documents are fetched within a reasonable time of each other, for example by preferencing documents embedded in other documents. Therefore, this has largely been an issue of improving performance, reducing and optimizing disk access and CPU cycles. Ideally, bandwidth would be the limiting factor. In fact, most of these issues can be overcome by segmenting the crawl and running Heritrix on several machines in parallel.

For this project, it was decided that the total number of hosts would be limited to under a hundred. That is to say, that we would not require the software to be able to handle more. In addition, a limit of URIs was decided in the low millions. This corresponds to a full days crawling using Heritrix's snapshot abilities running on decent hardware. This would free us from tackling the difficult task of optimizing the queue behavior in Heritrix and allow full focus on the adaptive revisiting strategy.

While work has continued on improving the queuing in Heritrix's snapshot oriented Frontier, it cannot be directly applied to an incremental Frontier. This is because an incremental Frontier needs to maintain more state information about queues and queue items. This includes the time when URIs can next be issued, and where they should be placed on return to the queue after each round of processing.

As with the snapshot crawls, incremental crawls can be segmented and run in parallel.

The project needed to tackle a variety of issues, such as how to use information of prior changes to adjust revisiting times and how to detect changes in documents. These are complicated and extensive fields of research that we could not hope to cover exhaustively. Accordingly, we limit ourselves to providing at least a basic solution to them and suggesting possible approaches to improving on them. These solutions entail creating the necessary Heritrix components to use them, making it easier to 'plug in' future enhancements in their place.

Ultimately it is our objective to create a framework for continuous crawling using Heritrix.

5 Defining an adaptive revisiting strategy

We now define an incremental strategy that uses observed change rates to adapt its revisit scheduling.

The basic goal of any incremental crawl is to capture any change in all the documents within its scope. In reality, the only way to be sure that you are capturing *every* change is to visit, or crawl, the document at very short intervals. In practice, this is of course impossible, where we may have hundreds of thousands of documents that need to be monitored. We therefore need to manage our resources carefully and try to revisit documents only as often as is *likely* needed. As was discussed earlier, this does not only conserve bandwidth and possibly storage space, but also allows us to crawl more documents on the same host.

Because of politeness restrictions the total number of requests sent to a host over a period of time is fixed. Since we must wait a fixed minimum amount of time between requesting documents from the same host we may fully utilize this capacity by visiting 10 documents every second. In other words, if we need to revisit each of them every second we can only crawl those 10 whereas we could crawl 100 documents every 10 seconds etc. In most cases 10 documents per second would represent an excessive load on the web servers, only large sites running on powerful or multiple servers would be unaffected by such an aggressive crawl. This, of course, assumes that our hardware can do this, which is unlikely if we are trying to crawl many sites in parallel with such frequency.

Even choosing a longer interval of one hour would not eliminate the problem. As the wait interval becomes longer, we do gain the ability to crawl more, but we also risk missing changes. Since we know that documents change with varying frequency it is essential that we utilize this knowledge to optimize our crawls. That is, make the wait interval vary between documents and have it approach the actual interval between changes of the document as much as possible.

In order to achieve this we must visit the same document more than once. If the document has changed between visits, we can assume that its probable interval between changes is less than the time that elapsed between our visits and reduce the wait time before our next visit. Similarly we would increase the wait time if the document had not changed.

This approach does make the assumption that document change rates remain relatively constant. This may however not be the case. For example, a politician's website may see little or no change over the course of his elected tenure, only to become extremely active a month or two prior to elections. After the elections the website may return to its more placid state. If we wish to crawl very aggressively, variations over the course of the day may come into play. A news website is far more likely to be updated during business hours than during the night. Even though updates may come at night the majority of updates take place when people are at work.

Despite these drawbacks, most websites and documents are believed to have reasonably regular change rates over long periods. After all, a website that wishes to attract regular readers needs to be consistent in the presentation of new material. We will always risk loosing interim versions, especially if documents are changing several times a day, but it is extremely difficult to prevent this without visiting the documents excessively. Also, we should consider that in most cases, the changes introduced are *additions* and material is moved down or into archives, not lost. Therefore, we can hope to capture all the content, even if we do not capture incremental change.

Ultimately this strategy relies on a heuristic approach. Based on whether a document has changed or not, we increase or decrease the amount of time we wait between visits. However, we can try to add more data for the heuristics to consider.

One such consideration would be the document's content type, that is the nature of its content. Is it text, graphical images, audio or video files etc. We know that the content type significantly affects the probable change rates[2]. This is obvious to infer from a logical look at some of the different types. HTML and other text files are quite easy to change and in fact it is extremely common that web servers compile HTML pages on demand based on information stored in databases. Such pages are extremely easy to change and they are expected to change very often. In contrast, images rarely change. There are some examples of software that creates images on-demand, such as graphs and similar material, however this is much less common.

Content such as audio and video is quite complex to change. It is of course possible that the document a URI refers to may be replaced by another one, but this is unlikely in a practical sense. If it is new content, then it would make sense to create a new link, why reuse the URI? We can not entirely rule it out, but it is unlikely. Similarly, many other content types represent material that seldom changes and logically if new content should replace older content, it should do so in an obvious way, using a different URI. For example, an updated version of a software package might replace an older one, but normally you would want the URI to reflect this.

We can not entirely rule out any change in any document type, but we can assign them different initial wait times to reflect the inherently different probable change rates. This immediately saves a great deal of resources since it would take several iterations of the adaptive strategy outlined above to 'discover' that an image is changing ten times less frequently than the HTML document that links to it.

We may however wish to handle embedded files a little differently. If a HTML page has changed, one might argue that all the images on it (and other embedded documents) must be refreshed in order to ensure that we have accurately captured the page. Any *new* embedded documents would of course be fetched as quickly as possible, but if one of them has changed, our collection would fail to highlight that, since the embedded document had not been expected to change. We might choose to accept this, or we may want to set a rule that each time a document has changed, any document embedded in it must also be revisited, regardless of its probable change rate. We simply move it to the head of the queue to ensure that we get a consistent view of the HTML page in our collection.

It is worth noting here that there is some correlation between document sizes and document types and therefore between document sizes and probable change rates. Document types that are typically larger are less likely to change. Text files, such as HTML are usually quite small, a few kilobytes in size. Images range from a few kilobytes up to megabytes. Audio and video files are typically in the megabytes etc. We note this not because we intend to use size as a factor in determining the probable change rate (we might, but we'll leave that for future work), but rather to emphasize the usefulness of immediately discriminating based on file types. Using such an approach we do not only save on visits to resources that are very unlikely to have changed, we save on visits to the *largest* resources, the bandwidth savings are likely to significantly exceed the proportional savings in number of visits. For example, during a crawl of the Icelandic national domain, *.is*, in 2004, audio and video files accounted for less than 0.3% of the total number of files downloaded, they however accounted for 11% of the total amount of data collected [16].

The heuristics might also look at any number of other indicators. For example, [4] observes that documents without the *last-modified* HTTP header are about twice as likely to change as those with the header. This type of information could be utilized to further improve on our initial wait time. Some other possible indicators might be the length of the path, distance from seed and, as noted earlier, the size of the document. As we discussed before, our objective is to create a basic incremental strategy and without greater investigation using those indicators would be nothing more than a guessing game. There are also, undoubtedly, many other possible factors, but covering them all is well outside the scope of this project.

To summarize, our incremental strategy, which we are calling an *Adaptive Revisiting* strategy, operates by crawling URIs normally. Each URI, after initial crawling, is assigned a wait time, based on its content type. Once the wait time has elapsed, they are recrawled and based on whether or not they have changed we increase or decrease the wait time before the next round.

To properly configure the crawl, a default initial wait time needs to be assigned for every content type that is specified. It should be possible to define any grouping of content types, allowing the operator to decide how finely the content based discrimination operates. A minimum and maximum wait times for each content types will also be needed as well as the factors to divide or multiply with in the case of changed and unchanged documents respectively.

5.1 Detecting change

One of the key requirements for using an adaptive revisiting strategy and of considerable importance to any incremental crawl is the ability to accurately detect when a document has changed. In our definition of an adaptive revisiting strategy we assume that this is possible since otherwise it is impossible to implement an adaptive strategy. Even an incremental crawler that did simple repeats, crawling the same URIs in the same order on each iteration would benefit from the ability to detect if a document has changed. Using such information it would be possible to discard duplicates and only store changed files, saving on storage costs.

Unfortunately, detecting changes in documents is not easy. Especially when we are both concerned with accurately detecting change and non-change. It is clear that the adaptive algorithm will be thrown off just as much by a failure to detect a non-change, as a failure to detect a change in the document. However, because we want to err on the side of caution (this meaning we'd rather download extraneous content than risk missing something) we generally prefer to fail to detect a non-change to the alternative. If we fail to detect a non-change, we will needlessly assume the document has changed and visit it more often than need be. On the other hand a failure to detect a change will cause the algorithm to visit a page less frequently, even though the content is already changing faster than we visit the URI.

The simplest form of change detecting is a straight bitwise comparison. This is straightforward enough and well understood. It can be simplified by creating strong hashes of the content and comparing the hashes. The probability of failing to detect a change is extremely small if a good hash function is used and can safely be ignored. Likewise, one could argue that the probability of failing to detect a non-change is equally small. Unfortunately, things aren't that simple.

We now come to *defining* what constitutes a change in a document. If we define change as any change to the sequence of bits, we are fine and the hash comparison is sufficient. This is however an unsatisfactory definition. It fails to take into account the fact that most documents on the web (HTML files in particular) are loaded with a myriad of data that is not directly related to the content. This includes the layout, but more importantly, HTML pages in particular often contain dynamically generated information that changes with each visit, but should *not* be considered a part of the content. The classic example would be a web page, possibly the front page of a news service that prints the date and time on each visit. This value will change with every visit, but does not constitute a change in the content. Documents like this could cause the adaptive strategy to severely over-visit them, wasting bandwidth and storage as well as placing an unnecessary strain on the web servers in question. It would also cause additional effort later when trying to analyze the contents of the archive, since it contained duplicates that could not be automatically detected and would need to be manually sorted.

Unfortunately, while it is quite simple and straightforward to compare two documents on a bit by bit level, trying to compare the 'content' is much more complicated. It requires some understanding of what constitutes content and what can be ignored. One option, in dealing with HTML files, is to ignore the markup code since, typically, it only contains layout information. Yet even this falls far short of our needs, since we have already pointed out an example where irrelevant information would be a part of an HTML's 'content'. Therefore the distinction drawn between content and presentation in HTML is of no use to us.

Some alternatives do present themselves. We know that, primarily, the problem relates to web servers inserting dynamically generated sections unrelated to the main text of the documents. Therefore, the sections are likely to be a relatively small part of the document. In such an instance, a 'roughly the same' or ‘close enough’ comparison might suffice. That is to say, instead of comparing two documents and declaring them either identical or different, we use a gradient for the difference between them. Documents that are sufficiently similar are treated as being unchanged. This means minor changes will be ignored, whereas changes to the general body of a document should be detected.

A good deal of work has been done on the subject, including such methods as *shingling*[6] that provide pretty good results.

Unfortunately not all minor changes are unimportant. Consider for example the press release archive of a company, government agency or other similar entity. Let's suppose that several months after issuing a press release it has become clear that the policy set forth is no longer entirely valid, might in fact show the issuing party in a bad light. The temptation to 'tone down' the headline to lessen the impact might well be too much for some. The actual change could be very small, the inclusion, change or deletion of one word to change an absolute statement to a conditional or similar. There is no denying that such a change is minor in the number of bits affected and any 'roughly the same' algorithm would deem the document unchanged. However, this change might well be of considerable importance in an historical context. Since the primary users of this software are National Libraries and other parties dedicated to *preserving* an accurate account of the World Wide Web, being able to capture all important changes is critical.

Logically we must then conclude that while 'close enough' algorithms could be used for change detection, it should not be the de facto choice. We should always err on the side of caution. Therefore we will emphasize strict comparisons despite the fact that they give us a lot of false positives when it comes to change detection. It is implemented by using SHA-1 [14] hashing.

Since we know that there are numerous sites of interest that do contain troublesome pages in terms of change detection, we have opted to include a simple, operator configurable, addition to strip sections of documents before the content hashes are calculated. The idea being that operators could designate troublesome sections of documents with regular expressions. These sections would then be overlooked or stripped out when the hash is calculated.

While this does not in any way provide a generic solution, it does allow operators to tackle almost any previously encountered issue. This in turn enables them to continuously crawl any website without the adaptation algorithm being fed bad information, assuming that the operator is willing to invest the time needed to set up the stripping where needed.

As an alternative to examining content in order to determine if it has changed we might want to use the HTTP header values *last-modified* and *etag*. If they are implemented correctly a change to either of these indicators between requests equates to a change in the document. Since the headers can be downloaded before the document body, we can evaluate them and then decide if the document should be downloaded, thus avoiding downloads of unchanged documents. This goes even further in saving bandwidth and reducing server loads than with content hashing for change detection, where we always had to download a document for each revisit.

Unfortunately, this approach is also flawed. Numerous websites simply fail to include this information or give false information. A Danish study [3] of the reliability and usefulness of Etags and datestamps found that while the reliability was quite high, the usefulness was less than 65%. That is to say, reliability is the percentage of checks where a change is accurately predicted. Thus, a low reliability would indicate that we are missing changed documents, something we wish to avoid at all cost. Usefulness is then the percentage of checks where a non-change is accurately predicted, a low usefulness leads to unnecessary downloads.

Four different combinations of using datestamps and Etags were evaluated. In all cases, if the indicators are missing, they assume that a change has occurred and download the document.

1. Download when datestamp has changed
2. Download when Etag has changed
3. Download when both indicators have changed (or just one if the other is missing)
4. Download when either indicator has changed

Table 1. Reliability and usefulness of the four schemes when missing indicators are taken into account. This statistic is based on 16 harvests of the front pages of Danish web domains. 346,526 web servers were contacted in each harvest [3].

Scheme	**Reliability**	**Usefulness**
1. Datestamp only	99,70%	63,7%
2. Etag only	99,91%	52,5%
3. Datestamp and Etag	99,66%	64,1%
4. Datestamp or Etag	99,72%	53,5%

Scheme 2 achieves the highest reliability but has the lowest usefulness, most likely indicating that Etags, while generally correct, are not that common.

In fact none of the schemes are useful more than two thirds of the time and while their reliability is quite high, none of them is 100%, meaning that we run the risk of losing some changes if we use them.

In other words, change indicators in HTTP headers will never suffice as the sole content change detectors for an adaptive revisiting strategy. While we might accept the reliability as being good enough, their usefulness is so low that we would have a significant number of documents (about one third if the results of [3] apply universally) that are observed to change *every* time we visit them. Those documents would be visited excessively and run the risk of monopolizing the crawlers time while other, better-behaved, documents wait.

However, since they *are* quite reliably, we could use these indicators to abort the download of documents, but make no assumptions about the ones we do download. That is to say, if the datestamp and/or Etag predict no change, we accept that and abort the download of the document, saving on bandwidth. If a change is predicted, we proceed to download it, but instead of relying on this prediction, we pass the downloaded document into the hash comparison that we discussed earlier. This allows us to make use of the datestamp/Etag reliability wherever possible, but using hash comparisons elsewhere preventing the low usefulness rate from being a problem.

This gives us the bandwidth saving and reduced server stress of the HTTP headers approach and the improved accuracy of content hashing. It *is* more likely to falsely predict a non-change since the reliability of the HTTP headers is not 100% (unlike standard hashing), but if this is acceptable it can be used to improve the crawlers resource utilization. Especially since the politeness wait that follows a download of a document from a web server until we can contact that server again, can be reduced since fetching the HTTP headers is a much less costly operation than downloading the entire document body.

6. Integration with Heritrix

Let us consider what we would need to implement a strategy, like the one described in the previous chapter, in Heritrix.

The most obvious difference is the need to have a Frontier that repeats URIs rather than queuing, issuing and, essentially, forgetting them. We need a Frontier that queues, issues and requeues the URIs. The requeueing is needed so that they will be tried again at an appropriate time. That is, reissuing them when the adaptive algorithm estimates that they will have changed. We need a priority queue, using a time of next probable change for sorting.

So, why don't we just augment the existing BdbFrontier to handle priority queues and (optional) requeueing of URIs? The short answer is: Simplicity.

The long answer goes something like the following. We discussed earlier the limits we placed on this project. One of the key aspects in the current efforts on the BdbFrontier is improving its performance, and that means making it capable of handling more URIs and more hosts within a single crawl. Reduce contention and ensure that the Frontier is not a bottleneck, while at the same time managing numerous, large queues.

Conversely, we have accepted the fact that for our incremental strategy the number of hosts and URIs will be somewhat limited. This in turn allows us to design a Frontier with a more rigid structure. In order to accommodate the large number of potential queues, the BdbFrontier must ensure that each queue has very limited overhead (preferably none). As will become clear when we go into the details of the AdaptiveRevisitingFrontier, this would make a robust implementation of it impossible and require a much more delicate state control.

Of course it is by no means impossible to improve the BdbFrontier to allow for incremental crawling. It is simply a trade-off. Imposing additional demands on it to maintain priority queues, in place of simple *first-in-first-out* queues as well as handling the added constraint that each URI contains a timestamp for when they can next be fetched is simply counterproductive to the more essential mission of improving its snapshot performance.

While having two Frontiers can impose an added burden in maintaining both, it allows each to be optimized for its type of crawling. Essentially, we prefer to create specialized Frontiers rather than a jack of all trades.

There is also the fact that the BdbFrontier, as well as Heritrix in general, has been designed primarily with snapshot crawling in mind. Before, in our discussion of Heritrix's architecture, we included a high-level schematic (Figure 3) describing it. This figure was taken from an article by the Heritrix development team [1] and the decision to include it, rather than our own illustration of Heritrix's architecture, was to highlight the inherent structure in its design. Note, for example, that the Frontier contains two objects. A stack of queues and something labeled 'already included URIs.' For snapshot crawling, this is the most practical solution and reflects the workings of the original HostQueuesFrontier and the BdbFrontier. Newly discovered URIs are queued and a hash of them is stored. New URIs are only compared against the existing hashes. Once a URI has been issued, the only record of it will be the hash.

This is of course extremely efficient for snapshot crawling, but entirely unacceptable for incremental crawling. In fact, given that we need to implement priority queues for incremental crawling, we will be needing random access to the queues and the very nature of an incremental crawl requires that no URIs are ever *removed*, meaning that we can just as effectively implement the duplicate detection in the queues, saving on the additional complication of maintaining the hashes. It could be argued that the hashes provided faster, more efficient, duplicate detection, but as we noted earlier, our goal for this initial work on an incremental Frontier is stability.

Fortunately, this issue is easily overcome by creating a new Frontier. There are a number of other factors in Heritrix's design that were not so easily circumvented and we'll highlight them where they arise, as we discuss the modules created to implement the adaptive revisiting crawl strategy.

6.1 Changes to the CrawlURI

Very few changes needed to be made to the Heritrix framework to implement the adaptive revisiting strategy. However, because our incremental strategy reuses the CrawlURIs, unlike snapshot strategies, some modifications were necessary. Primarily, we needed to ensure that data that should be reset for each round of processing, was in fact reset, and that data that needed to persist did just that.

There are actually two facets to this since the CrawlURI is composed of a set of predetermined class variables and a number of data items stored in it and accessible via a string key.

What we eventually wound up doing was to review all the member variables and make those that should not be carried around between processing attempts *transient*. This alone was not sufficient since many variables need to be reset to default values and so the preexisting method *processingCleanup()* was augmented to capture all of this. The Frontiers are then responsible for invoking it if they intend to reserialize the CrawlURI after it is returned to them.

As for the keyed items, the *processingCleanup()* goes through all of them and removes any items whose key is not on a list of persistent items. Prior to this the entire set was serialized and that could cause run time errors if a non-serializable object had been placed in it. Methods were then added to the CrawlURI to add and remove keys from the list of persistent data. The responsibility for adding keys to this list then falls on the modules that require that the data, represented by the key, be persistent. This list is static, so modules can add the keys they need in their constructors.

It was not feasible to simply drop all the keyed items, since some vital data could only be stored there, such as the ‘time of next processing’ for CrawlURIs. It was necessary to be able to store both persistent and transient data and our implementation makes transient the default, trusting that modules will register the keys they need. Since not all Frontiers reserialize the CrawlURI, if persistence was the default some modules storing unsafe data (non-serializable) might fail to make them transient, causing problems when running under other Frontiers than the one it was originally designed for. For example, the DNS fetcher does this and would have had to declare its data transient if it were not the default.

6.2 The AdaptiveRevisitingFrontier

The AdaptiveRevisitingFrontier (ARFrontier) is, conceptually, based on the HostQueuesFrontier. It was developed at the same time as the BdbFrontier. When work on it began, it was obvious that the custom written queues that the HostQueuesFrontier used would not be adequate. For one thing, they did not allow random access reads, let alone writes. This made it impossible to modify them to be priority queues.

Priority queues are fairly simple in theory, and can be easily implemented, with linked lists for example. However, due to the amount of data involved, it is essential that the queues exist primarily on disk. This is critical for incremental crawling which must retain the CrawlURIs, whereas snapshot based Frontiers can usually just store the CandidateURIs and only create CrawlURIs once the URI is ready for processing. Of course, snapshot Frontiers are also likely to have many times the number of URIs at any one time.

The first thing that has to be solved is the issue of handling the serialization and deserialization of a significant amount of data. Rather than implementing a custom solution, which would probably have been more complex than the rest of the project, we choose to find suitable third party software to handle the task.

One idea was to utilize a generic database. Thus we would use JDBC [30] to connect to a properly set up database that could store the data. This would delegate the tasks of writing the data to disk, indexing it, and setting up an in-memory cache to the database software. There was some precedence for this, as Nedlib[13], which implemented an incremental strategy, used a MySQL database to store its state.

The fact that Nedlib was ultimately abandoned might not be seen as the most ringing endorsement for this strategy, but we believe that the concept is, in general terms, sound. There are however some problems with this approach which we can not overlook.

First, there is the issue of operational *simplicity*. Adding a dependency to a third party product was not considered a good move. Even if we had chosen a suitable open source database, such as MySQL for example, and included it with the Heritrix distribution, it would still mean that two separate programs needed to be installed and started for this to work.

While it would, potentially, be quite powerful to allow the user to set up and use any database application that they are already using, the fact that Heritrix would be writing and reading a tremendous amount of information means that in practical terms the database would have to be located on the same machine (or accessed via a high-speed network with little other traffic at the very least) and do nothing else. Obviously, if the database was also handling other applications, the load from Heritrix could seriously degrade their performance.

Then there is the issue of performance. Additional to the above is the fact that communications would have to be via a JDBC connection, rather than native access. This would impose a performance penalty.

There is also an issue with extensibility. The fact is that CrawlURIs include a lot of data. This data would has to be converted for storage in a table. Since the CrawlURIs contain a hash map of data that may contain any number of entries, this gets complicated and would probably require that a serialized version of an object be stored in one field for restoration, with other fields being used only for indexing. Since object serialization in Java is bulky (a lot of information about the class is repeated for each object), this is far from ideal. In fact, one of the main problems with the custom written queues used by the HostQueuesFrontier was the extremely bulky nature of the queues which stored serialized versions of the CandidateURIs and CrawlURIs.

Finally, relational databases simply contain a great number of features entirely superfluous to the task at hand. A more streamlined piece of software would be preferable.

For all the above reasons, traditional object oriented databases were excluded as solutions to the disk storage problem.

The desired features for a data storage solution were:

- Open source with a license what would allow us to distribute it with Heritrix.
- Implemented in Java, like Heritrix.
- Capable of being embedded in Heritrix, rather than running as a separate program.

- Capable of storing Java objects in an efficient manner.
- High performance, capable of managing large amounts of data in an expeditious manner.

One particular solution was quickly arrived at: Berkley DB, Java Edition [9] by Sleepycat Software. The Berkley DB, Java Edition is, as the name would imply, a Java implementation of the highly successful Berkley DB product. It was designed from the ground up in Java and takes full advantage of the Java Environment. It enables the storage of objects in a very efficient manner and, to quote Sleepycat Software's product page, "*...supports high performance and concurrency for both read-intensive and write-intensive workloads*"

Later the product page goes on to say, "*Berkeley DB JE is different from all other Java databases available today. Berkeley DB JE is not a relational engine built in Java. It is a Berkeley DB-style embedded store, with an interface designed for programmers, not DBAs. Berkeley DB JE's architecture employs a log-based, no-overwrite storage system, enabling high concurrency and speed while providing ACID transactions and record-level locking. Berkeley DB JE efficiently caches most commonly used data in memory, without exceeding application-specified limits. In this way Berkeley DB JE works with an application to use available JVM resources while providing access to very large data sets.*" [9]

Or, in other words, exactly what we need. The Berkley DB can be easily embedded in Heritrix, and since the access interface is on a programming level we have native access to it, allowing for the best possible speed. Its data management is also very much in sync with our needs, allowing fast writes and reads and offering a decent caching mechanism. It is also able to handle very large data sets, a vital requirement, since we want to manage millions of URIs at once.

Shortly after initial testing of the Berkley DB had gone well, parallel work began on the BdbFrontier which also utilizes it. While both the ARFrontier and the BdbFrontier use the Berkley DB for data storage they do so in drastically different manners. Since the BdbFrontier is primarily interested in limiting the per-queue overhead, it tries to ensure that the queues themselves are largely stored in the database. The ARFrontier, with different priorities, goes another route and still has one queue object in memory for each actual queue.

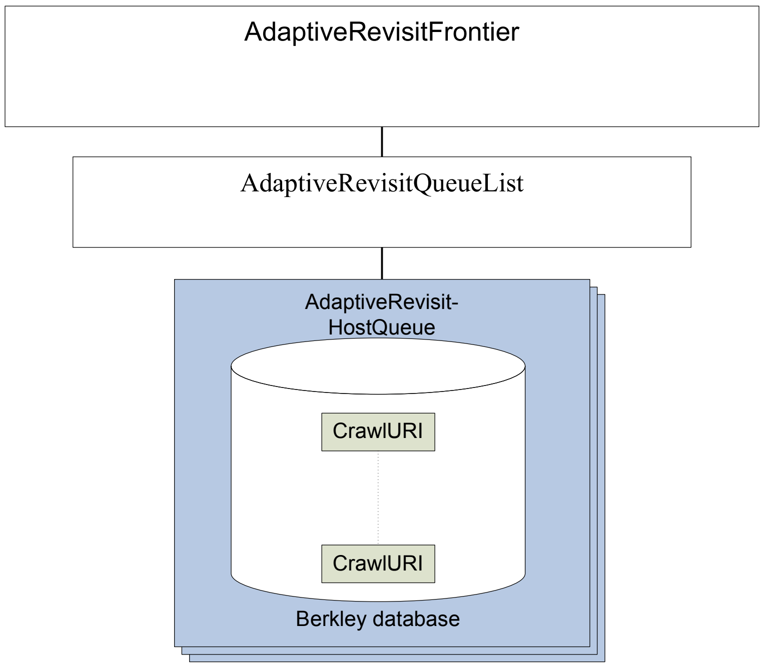


Figure 8. A high-level view of the AdaptiveRevisitFrontier architecture

Figure 8 gives a very rough view of the AdaptiveRevisitingFrontier's architecture. A series of queues, named AdaptiveRevisitHostQueues are used and managed by the AdaptiveRevisitQueueList. Both the queues and the list are covered in more detail later. Together they handle all the data management and storage and are largely responsible for maintaining the state of the crawl.

The ARFrontier publishes several configurable parameters, using Heritrix's settings framework. The settings are:

- *delay-factor*
  Related to politeness rules. Indicates how long to wait before contacting the same host again. The wait time will be the time it took to process the last URI from the host, multiplied by this factor. This adjusts the politeness to take into account heavily stressed servers that would typically be slower to complete a transaction. It also means increased politeness following the download of a larger file. Default value is 5. This is the same default value used in other Heritrix Frontiers.

- *min-delay-ms*
  A lower bound for the politeness wait between processing two URIs from the same server. Default value is 2000, or two seconds. This is the same default value used in other Hertrix Frontiers and is more than adequte for ensuring good politness.
- *max-delay-ms*
  An upper bound for the politeness wait between processing two URIs from the same server. Default value is 5000, or five seconds. This is the same default value used in other Hertrix Frontiers.
- *max-retries*
  Some errors, that prevent the successful processing of URIs, are considered *retriable*. That is, they are generally caused by transient issues, such as network connectivity, remote hosts being down etc. They are therefore tried repeatedly until the URI is crawled successfully or the maximum number of retries, dictated by this parameter, is exceeded. Default value is 30, same as other Heritrix Frontiers.
- *retry-delay-seconds*
  The amount of time to wait before retrying a URI that had a retriable error. Default value is 900 or 15 minutes. Again, this is a typical value for Frontiers in Heritrix.
- *host-valence*
  Host valence dictates the number of *simultaneous* connections to the same host. Normally this value should be 1, as multiple simultaneous connections are considered to be impolite at best. However, when crawling websites that are known to be able to handle a significant load and/or we have the permission of the websites' owners/operators, we may wish to crawl them more aggressively. The default value is 1.
- *preference-embed-hops*
  Documents, HTML pages in particular, frequently embed other documents in addition to linking to them. When a document is embedded in another document it is needed in order for the originating document to be displayed properly. Typically, these embedded documents are images. This value dictates the number of sequential embed 'hops,' or links to treat in a preferential manner. That is, schedule them so that they are fetched before anything else in that queue. Default value is 0, which differs from the more typical value of 1 in the BdbFrontier. The reason for this is that preferencing embeds in our adaptive revisiting strategy has the effect that the embedded

documents are always fetched right after the originating documents, regardless of their own change rate.

The naming of the settings, i.e. all lower case with dashes to separate words and ending with the scale of the setting (min, sec etc.), follows standard practices in Heritrix. The reasons for this are partly technical, but mostly a legacy issue. When creating settings, modules can associate a description, or help text, that is accessible in the user interface by clicking a question mark next to the name of each setting.

Most of these settings are fairly standard for Frontiers. In fact, only the *host-valence* is not used by the BdbFrontier. This feature was introduced in the HostQueuesFrontier but was generally considered buggy for values larger than 1. The feature was omitted from the BdbFrontier for those reasons. Its implementation in the ARFrontier is due to the fact that such focused crawling of very select sites is far more likely to require it. Also, as we'll see later, the design of the host queues made its implementation much easier than would be possible in other Frontiers.

The other settings relate to some essential Frontier responsibilities. Ensuring that the crawl behaves in a polite manner, retrying URIs that have retriable errors, and preferencing embedded documents. All other behavior of a crawl, such as wait times between revisiting a URI in an incremental crawl, are configured outside the Frontier. Some parameters are configured as a part of the CrawlOrder, but most are associated with a particular processor, scope, or filter. These settings are used by their respective modules and, if needed, any directions to the Frontier based on them are passed via the CrawlURI.

Figure 9 shows, roughly, how a URI is passed around. Starting with a request by a ToeThread for the 'next' URI, how a URI is selected from the pool of pending URIs, processed and returned. The figure is complicated somewhat by the fact that preconditions may not have been met. This requires that processing be preempted and the required information be gathered first. Currently DNS information and robots.txt is needed for each host before any URIs from it can be crawled. Fortunately, most of this is handled by the processors. The only thing the ARFrontier has to keep track of, is to reschedule URIs that fail because of missing preconditions *after* the preconditions have been met.

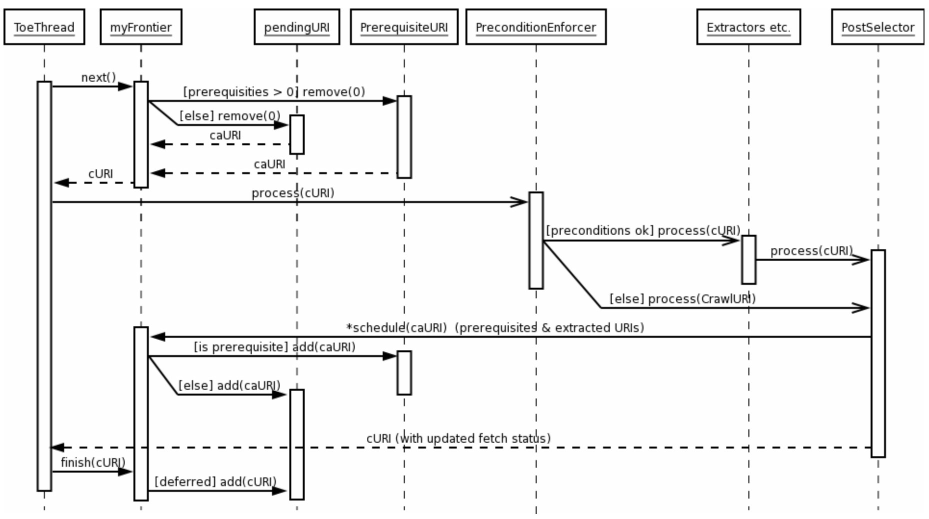


Figure 9. Frontier data flow[7]. This illustrates how data flows between the Frontiers, the ToeThreads and the processors. It only focuses on the most essential data contained in the CrawlURI, that is the fetch status, which may include failure to meet preconditions, failure to crawl, crawled successfully etc. The section marked 'pendingURI' represent the Frontier's URI queues.

Since we need to store various additional information in the CrawlURIs' keyed list, the CoreAttributeConstants interface was extended with the AdaptiveRevisitAttributeConstants interface which adds all the data keys needed by the adaptive revisiting strategy.

Let us now go through the life cycle of a URI in the ARFrontier.

CandidateURIs are scheduled, either by the Postselector working its way through discovered links, or loaded from seeds at the start of the crawl. CandidateURIs that are scheduled via the *schedule()* method are queued into a ThreadLocal queue to avoid the synchronization overhead that would otherwise be incurred from scheduling numerous URIs sequentially. This queue is processed when the ToeThread returns the URI being processed, via the *finished()* method. As the initial loading of seeds is done internally in the Frontier, this batching is avoided in their handling.

The scheduling of CandidateURIs requires that they first be converted into CrawlURIs. This simplifies the issuing of URIs, where we can now assume that all issuing URIs are CrawlURIs, rather than having a mix of CandidateURIs and CrawlURIs. This is in stark contrast with the snapshot

Frontiers, which generally assume that anything in the queues is a CandidateURI, since the CrawlURIs should only exist for the processing part.

Once a CrawlURI has been created, it is assigned a 'time of next processing,' which is set to the current time. Then the Frontier determines if it should receive preferential treatment because it is an embed. If so, the CrawlURI's scheduling directive is raised. More on the effect of this when we discuss the ordering of the priority queues later.

Now the ARFrontier is ready to insert the CrawlURI into its proper host queue. The ARFrontier looks this up via the AdaptiveRevisitQueueList. The name of the host is created using Heritrix's *HostnameQueue-AssignmentPolicy*. This name act as a key for the lookup of the queues.

In theory the key can be anything and is not restricted to hostnames. In fact the BdbFrontier offers an alternative assignment policy based on IP numbers, as well as allowing the user to specify the queue names, using the override capabilities of the settings framework to specify different queue names for different hosts. Effectively, allowing an entire domain to be one queue, for example, rather than several queues, one for each host under the domain.

In practice however, assignment by IP numbers is difficult since IP numbers are not known until after the DNS lookup. Until then, the IP assignment policy returns host names. In an incremental crawl, this means that 3 URIs (seed, robots.txt and DNS) will remain in the host named queue, while all subsequent URIs from that host end up in the IP named queue. This is acceptable in snapshot crawling, where the host named queue disappears once the seed has been crawled, but unacceptable in an incremental crawl where it remains.

If no queue exists for the given host name, the Frontier directs the queue list to create a new one for that key. The CrawlURI can now be added to the queue.

The host queues take responsibility for duplicate elimination.

When the ARFrontier needs to issue a URI, it consults the queue list for the current 'top' queue (more on that later). If this queue is not ready, the thread is ordered to wait until it becomes ready. Whenever the processing of a URI is completed, all waiting threads are notified so they can check if the top queue is now ready. Eventually a queue should become available.

The ARFrontier then requests the next URI from the queue. The amount of time that the URI is overdue is then calculated and stored in the CrawlURI. This value is logged later and is also optionally used by the WaitEvaluator.

Once a URI is issued, the ARFrontier only keeps track of it by the state of its queue, but does not otherwise concern itself with it. If a URI is never returned back to the Frontier, via the *finished()* method, then the queue in question, assuming a host valence of 1, will become permanently snoozed, effectively blocking it from further progress. While this should never happen, bugs in the processors or ToeThread can have this effect. If this occurs, the operator can intervene and 'kill' the ToeThread, causing the URI to be returned to the Frontier with an error code.

Eventually, the URI should be returned to the Frontier. It is here that the ARFrontier departs from other Frontiers in its handling, although much remains similar.

First the Frontier determines how the URI fared in the processing. There are four possible outcomes:

1. *URI was successfully processed*
   The web server was contacted, we may have received a HTTP error, such as '404 URL not found.' We still treat those as successes and will revisit them.
2. *URI encountered a retriable error*
   There are two flavors of this. Errors requiring immediate retrying, such as missing preconditions (the preconditions are scheduled with higher scheduling directive and should be processed ahead of any repeat attempt) and those that should be retried after a delay. This assumes that transient network errors caused the error and retrying the URI at a later time may yield a different result.
3. *URI was disregarded*
   Typically means that that robots.txt rules precluded a fetch attempt. This state means that the URI should be regarded as being out of scope. We will not be revisiting it.

4. *URI failed to crawl*
   This might be caused by a bug in one of the processors, but otherwise means that the URI is not crawlable. May be an unsupported protocol or a retriable error has exceeded its maximum number of retries. These URIs will not be revisited.

Successfully crawled URIs have a new 'time of next processing' calculated, based on the current time and the wait time determined by the WaitEvaluator. As we'll discuss later, the queues ensure that URIs that are not yet ready for processing are not issued.

Each of these conditions are handled separately, but in a similar manner. Unless the URI is to be retried, an entry in the crawl.log is made at this time. The ARFrontier adds several notes to the *annotation* section of the log. That is, it adds the current wait time for the URI, how often the document has been visited, how many versions have been encountered of it and finally how overdue the URI was for this round of processing. This is written in a compressed form like the following:

```
... wt:45s0ms,3vis,1ver,ov:3ms
```

These additions to the log allow the operator to track the URI as it is visited again and again and track how the adaptive strategy is responding to it. Whether or not the document has changed is determined elsewhere and those modules are responsible for writing that data to the log as we'll see later.

The disregarded and failed URIs are stamped with a 'time of next processing' that is as far into the future as possible. We use the maximum value for a long datatype and follow the practice in Java of representing time as the number of milliseconds since midnight, January 1, 1970 UTC. Setting this value means that the URI will never (in practice) be crawled again. It remains in the queue for duplicate detection and possible reporting purposes. The queues very rarely delete URIs.

The 'time of next processing' for URIs that are to be retried remains unchanged. Retry delays are enforced in the same way as politeness delays.

Once the proper 'time of next processing' has been set, the CrawlURI is returned its queue.

6.2.1 AdaptiveRevisitHostQueue

As is probably clear by now, the AdaptiveRevisitHostQueue (or ARHostQueue) class implements the majority of the adaptive revisiting strategy. It is similar, conceptually, to the URIWorkQueue used by the HostQueuesFrontier, but has improved its state control significantly and uses an entirely different data structure.

Having the ARHostQueue extend the URIWorkQueue interface is wholly unacceptable. This is primarily because the URIWorkQueue's state is controlled externally. It provides methods for snoozing (suspending), waking up (resuming) etc. rather than entering and exiting those states naturally. This creates additional complexity in the Frontier, which must maintain the state of every queue. Instead, the ARFrontier lets the queues themselves handle all this and never affects their state directly. The ARHostQueue's state is only affected by the Frontier as a *result* of its use of the queue. These actions include getting URIs, which can potentially make the queue *busy*, 'returning the URI', which can potentially make the queue snooze etc.

This is important, not only because it greatly simplifies the implementation of the Frontier, but also because the state is now much more complex since the queue may need to be snoozed, not just for politeness and error retry waits, but also because the 'next' URI is not yet ready.

In addition the ARHostQueues do not require several of the methods and facilities provided in the URIWorkQueue interface. For example, only the following states are needed for the ARHostQueues:

- *Empty*
  Initial state. Once the first URI has been added, this state will likely never reoccur since URIs are almost never deleted. Since queues are only created on demand, this state is extremely transient, and, aside from the *add()* method, nothing can be done with it while in this state.
- *Ready*
  The 'next' URI's time of next processing is in the past and any wait for politeness or error retry is over.
- *Busy*
  The queue has issued as many URIs as its *valence* allows without any

of them being returned. It will remain busy until one of the issued URIs is returned.

- *Snoozed*
  Either the queue is respecting politeness or error retry wait, or the next URI's time of next processing is in the future. In either case, the exact time when it will wake up can be accessed. As the state is calculated on demand, the queue will automatically move to a *ready* state once this time is reached. There is no need to explicitly 'wake up' the queue.

As we discussed in our overview of Heritrix, these are the essential states for host queues. The more advanced states, such as *inactive*, are not used since the progress of an incremental crawl should be primarily dictated by the time of next processing. Focusing on a fixed subset of hosts, as the active/inactive state is meant to achieve, is counter productive. Incremental crawls must be scoped so that the entire scope can be collected within a reasonable timeframe. Otherwise the URIs will be issued chronically late.

Despite the name, the ARHostQueues do not provide traditional queuing methods, enqueue and dequeue, since the URIs are never actually dequeued. When the queue issues them, the URIs are marked as being processed, but remain in the ARHostQueue, held aside, and the queue is simply waiting for the Frontier to update them.

Therefore the primary methods of the ARHostQueue are *add()*, *next()* and *update()*. The *next()* method operates much like a dequeue, in that it returns the top URI in the queue, or rather, the next URI to be processed from the queue. The naming reflects this quality and corresponds to the Frontier's *next()* method invoked by the ToeThreads to get the next URI for processing.

The *add()* and *update()* methods create the distinction between initial enqueuing of a URI to the queue and requeueing it after a round of processing. The terms adding and updating are much clearer than enqueue and requeue. The distinction is necessary, because when adding a URI, it may already exist, and we do not wish to modify the existing one or replace it. When updating, the URI certainly exists, and is being held aside. The act of updating it will return it into the queue and will also, possibly, affect the state of the queue, depending on whether it was *busy* or not.

Other methods that the ARHostQueue provides to the ARFrontier are *peek()*, which returns the current top URI, regardless of whether it is safe to start processing it or not. As the name indicates, this method is only used to examine the queue.

Finally, there are methods for getting the state and the time when the queue will next be ready. The latter only returns useful data if the current state is *snoozed*. If the current state is *ready*, it should return some time in the past and if the state is *busy*, it should return some time in the far future (maximum value of long).

Using these methods, it is relatively straightforward for the ARFrontier to manage the crawl, leaving the details of the queuing strategy to the ARHostQueue. Deciding which queue to use each time is handled by the AdaptiveRevisitQueueList.

Let's now examine how the ARHostQueue is implemented.

The essential part is the embedded Berkley DB that stores the actual queue. The Berkley database stores objects, keyed by other objects. That is, each database entry is composed of two elements, a key element and a data element, as noted, both of which can be any Java objects. The database is then indexed by the key element.

Berkley DB offers significant support for object serialization. This includes storing the class 'template,' that is all the serialization data that relates to the actual class, rather than the instance specific data, only once in a separate database, called a class catalog. This drastically reduces the amount of space each serialized object requires.

Since the BdbFrontier was introduced, support for Berkley DBs has been integrated into Heritrix's framework and this includes a common class catalog that the Frontier can access through the CrawlController. This is then passed to the ARHostQueues at construction time. The same applies to the Bdb environment that specifies the location on disk where the database should be stored etc. The queues therefore only need to implement their own databases.

We need to have the CrawlURIs not only sorted by the time when they should be processed next, but also by their URI (as a string). This means that we need to have two indexes for the database. While each Berkley database can only have one key/value pair and is indexed by the key, it is possible to create, what is called, a secondary database that is linked to a primary one. This secondary database simply indexes the data in the primary database with another key. Entries are made into the secondary database automatically when they are entered into the primary, once the association between the databases has been made. The database can then be accessed normally, either by looking up a value by its key or iterating through the keys.

We decided to use the URI as the key in the primary database and create a secondary database to act as the de facto priority queue. Because the scheduling directive also influences this the secondary key needs to be a composite of it and the time of next processing.

Fortunately, the Berkley DB API makes this relatively easy. The secondary database is fed a custom written key creator. This key creator receives the data being stored (the CrawlURI) and uses it to construct any key it wishes, using information obtained from it. By basing the key creator on the *TupleSerialKeyCreator* provided with Bdb, it is relatively straight forward to access the scheduling directive and the time of next processing and add each to the serial (or composite) key.

In addition to these primary and secondary databases, the ARHostQueue also has a database to store URIs that are currently being processed. If host valence higher than 1 were not allowed, this would not be needed, since the queue would become busy and the issued URI can simply remain at the front of the queue until the processing is completed, at which time it would be assigned a new time of next processing. However, with the possibility that multiple URIs are issued at the same time, this becomes much more complicated. Moving the URIs temporarily to a separate database was considered the simplest solution to this. Alternatively the URIs could have been marked as being processed and remained at the front of the queue, but this would have complicated the issuing of URIs and the calculation of the time when the queue will next be ready (which is based partly on the time when the 'next' URI will be ready for processing. The database of in URIs being processed is indexed by URI strings.

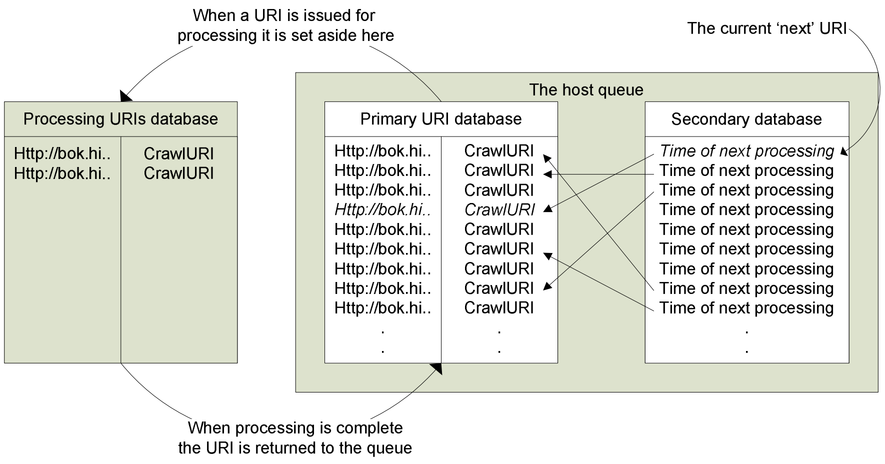


Figure 10. AdaptiveRevisitHostQueue databases. The secondary database is slightly simplified, in actuality, the scheduling directive is also taken into account, not just the time of next processing.

When adding to a queue, the Frontier specifies whether the time set on it should overwrite the time on any possible duplicates. The earliest time will however always be used. That means that a URIs time will never be 'moved back.' Also, if the scheduling directive is higher, both it and the time of next processing will be automatically updated.

Adding a URI can potentially change the state of a queue, but only if it was either empty, or snoozed because the next URI was not yet ready and the newly added URI can be processed immediately.

The queue becomes busy when the number of URIs being processed concurrently reaches the host valence setting. Once a URI is returned, the queue's state may change to ready or snoozed, depending on whether or not the queue should wait before issuing its next URI, either for politeness reasons or error retry wait.

The wait time is enforced separately for each concurrent connection allowed. That is, if host valence is 2, and one URI is returned, the queue will snooze for the time specified for it, 1 second for example. If the second URI is returned half way through this wait, the queue's state remains unchanged, as does the time when it will next be ready to issue a URI. Once the second has elapsed, a URI can be issued and once it has been the queue will go back to being snoozed until the wait time for the second URI has elapsed.

These wait times are implemented using a simple long array, with each number representing the time when that 'processing slot' can next be used. A time in the past means that the slot is free. A value of -1 is entered to signify that a URI is being processed.

This means that the queue's state changes automatically from snoozed to ready as time passes.

6.2.2 AdaptiveRevisitQueueList

The AdaptiveRevisitQueueList class manages the ARHostQueues. The list not only allows the Frontier to look them up by host name, but also orders them so that the 'next' queue is always the one that has been ready to issue a URI for the longest period of time. The list is ordered by the queues' 'next ready time' which in turn reflects the time when the front URI will be ready for crawling or when an enforced wait time has passed.

This ordering of queues ensures that no queue is starved, while also favoring those that are 'most overdue.'

The ARQueueList is fairly simple. It has its own database that contains the names of all its queues. As we'll discuss later, this is done purely for recovery purposes. It then contains a HashMap of the queues, keyed by their names and a TreeSet of queues sorted by the queues' next ready time.

When creating queues, the list will register itself as their owner, allowing the queues to notify it when they need to be reordered. It then adds them to the above data structures. Access to them is then relatively straight forward.

The list also provides some utility methods, such as compiling a report of all the queue's states, getting the number of queues and total number of URIs in all the queues.

6.2.3 Synchronous Access

All access to the ARFrontier is synchronized except the *schedule()* method which, as discussed above, queues up scheduled items in a TreadLocal queue, only to be flushed when the *finished()* method, access to which is synchronized, is invoked.

The reason for this high level synchronization is one of simplicity. While it might be perfectly feasible to handle synchronization at a lower level, possibly reducing contention on the Frontier somewhat, it would require considerable work to analyze and implement. Furthermore, this has typically not been done in other Frontiers, and the potential gain is unknown. Access to the ARQueueList for example would make it impossible for threads working on different queues to avoid overlapping.

6.2.4 Recovery

The BdbFrontier (and other existing Heritrix Frontiers) utilize an inefficient and somewhat complicated recovery journal or log for crash recovery. The way this works is rather simple in theory, but a bit more complicated in execution. Basically each 'transaction' in the Frontier is recorded to a compressed log (it is compressed because otherwise its size quickly becomes an issue). The transactions include scheduling of URIs, their issue, and the completion of their processing.

When a crawl has crashed, for whatever reason, the log can be 'replayed,' bringing the Frontier back to the state it was in when the crash occurred. In practice, when dealing with larger, multi million URI crawls, this takes an excessive amount of time.

Improved recovery abilities remain an untackled subject in Heritrix, partly because snapshot crawls can quite frequently get away with not having such an option. The reason for this lies in the fundamental nature of the snapshot crawl. It is started, it works its way through the webspace, and then it terminates. A crash in a very large crawl can have serious implications and may require the crawl to be restarted, but since the crawls always take a finite amount of time (usually on the scale of months at most) it is possible to execute it in one continuous run.

No proper *checkpointing* facility currently exists, whereby an accurate representation of the crawl state is stored on disk for later retrieval. Improvements on this are planned for the 1.6.0 release of Heritrix.

No such luck for incremental crawls that by their very nature have to run for an indeterminate amount of time. In theory, possibly forever. So it is absolutely vital that an incremental crawl can be stopped and started with an acceptable overhead.

Using the recovery log method is clearly impossible since the amount of time required is a factor of the number of URIs that have passed through the Frontier. In a snapshot crawl this is roughly equal to the number of discovered URIs, but in an incremental crawl, where each URI is processed repeatedly, this value can be many times the number of URIs. After running for a year, for example, a crawl with 100,000 URIs that are on average visited once a day, will have processed over 36 million URIs. Replaying the recovery log is not feasible.

Therefore, we needed an alternative way to recover from crashes and also to allow the operator to suspend and resume crawls if, for example, the computer needs to be upgraded.

Fortunately, the Berkley DB has good crash recovery abilities that we can take advantage off. Basically it can be assumed that anything that was written to a database is safe. Thus we only need to ensure that enough data is stored in the databases to allow the crawl to be reconstructed from them.

The databases are each named. When the ARFrontier is created, it gets a path to the 'state' directory, i.e. the directory where the crawls state should be written, from the configurations. The state directory is a standard configuration parameter that all Frontiers should use. The ARFrontier then, via the ARQueueList, tries to open a specified database, named 'hostNames' in that directory. If unsuccessful, a new one is created and we can then safely assume that this is a new crawl. If successful, the state directory refers to an existent crawl and the database containing the names of all the host queues has just been opened.

The ARQueueList now runs through all the entries in this list and creates a host queue for it. As each host queue's database is named using the hostname, the existing databases for each one should be opened. The only problem with this is that the valence for each queue will be the default value and no overrides are respected. This is due to the fact that a URI of some sort is required when accessing the settings for the overrides to be compared to. The valence could probably be stored in the hostNames database but implementing that is a subject for the future.

The ARHostQueue constructor will realize that existing databases are being opened and will move any URIs in the processing database over to the queue. It will also count the number of URIs in the queue. This is somewhat inefficient as it requires a loop through all the elements in the database, but is quite manageable for 10-100 thousand items. In the future the count should probably be stored in some database.

6.2.4 Frontier features not implemented

The Frontier interface specifies some methods for iterating through all the URIs in the Frontier and for deleting items, either by name or regular expression. Neither is currently supported in the ARFrontier. This is primarily because we placed greater emphasis on the core functionality and were unable to tackle this as well.

Adding the ability to iterate through the URIs in the ARFrontier, requires a good deal of additional work, primarily in designing a 'marker' for the current location.

Deletion of items is also not implemented. Currently, the only time a URI is deleted is when the Frontier decides to 'forget' a URI. I.e. a URI is deemed out of scope late (when being processed rather than when it is discovered, usually the result of the operator changing the scope) but we know that it may be discovered via some other path where it should be included. This is because crawls are frequently limited to certain domains or hosts but are still allowed to fetch 'offsite' documents if they are embedded, rather than linked to. There is also the possibility that the number of link hops was exceeded but the document might be rediscovered at a lower 'depth.'

For all these reasons it is necessary to be able to forget a URI and the update method has a parameter to indicate this. If set, the URI is dropped from the queue it is as if it was never encountered. This is, however, the only instance of a URI being deleted and is done for purely practical reasons. Typically we do not wish to ever delete any URI, that might have failed to crawl, for example, because we may very well encounter them again and we do not wish to requeue them if that happens.

For these reasons, the host queues offer no delete functionality, but this could be added. It remains to be seen, however, how useful that would be. One might speculate that the deletion should be implemented by setting the time of next processing to the extreme future (maximum value of a long) as is done for URIs that have failed to crawl or should be disregarded. This effectively deletes them from the crawl but keeps them in the queues and prevents them from begin rediscovered. This is also more in keeping with what the other Frontiers do, since they implement the delete by removing the URI from the queues only, but the hash of the URI used for duplicate detection remains unaffected.

The ARFrontier also does not apply the URI canonicalization. This feature was added after development of the ARFrontier began and since the Frontier interface does not enforce its implementation it was overlooked until after the ARFrontier was integrated into Heritrix's primary code base. This oversight will likely be addressed in the near future.

6.2.5 AbstractFrontier

When work began on the ARFrontier, the AbstractFrontier was fairly unstable as work on it had only recently begun. Since the AbstractFrontier, especially at that time, was geared towards the BdbFrontier it was decided not to base the ARFrontier on it. The reason for this was that while the concept was sound, it would be extremely difficult to develop a new type of Frontier while having to constantly compromise in terms of what the AbstractFrontier was doing and would allow.

Once the ARFrontier was ready for integration into Heritrix's primary code base, this subject was revisited. While the AbstractFrontier had by then improved considerably, several issues were quickly discovered which made it difficult to rewrite the ARFrontier to subclass the AbstractFrontier.

First there is the fact that the AbstractFrontier supports either hostname or IP based politeness. As discussed above, IP based politeness poses additional complications for incremental based Frontiers. While this could probably be overcome by renaming queues (for example) it would require considerable effort to implement.

The AbstractFrontier also implements bandwidth limiters, both for total bandwidth usage and per politeness unit (hostname or IP). This feature is not in the ARFrontier and it is uncertain how it would affect it. Likely, it would require the addition in the ARFrontier of calculating wait times for when bandwidth usage has exceeded the allowed amount. Overall bandwidth limitations are implemented in the AbstractFrontier.

While the ARFrontier allows the preferencing of embeds, it defaults this value to 0, unlike the AbstractFrontier which defaults it to 1. The reasons for this were discussed earlier. As there is no way, currently, for subclasses to override the default settings of their parents, subclassing AbstractFrontier would change the default value of this for the ARFrontier. Some of the feedback we have received from others testing the software indicates that the Frontier's behavior when preferencing embeds can be confusing, so we would prefer to continue defaulting it to zero.

The AbstractFrontier implements the recovery journal. This is an unwanted and unnecessary feature for the ARFrontier. As discussed earlier, the ARFrontier implements a much more robust recovery scheme that is completely incompatible with the recovery journal system.

There are also some issues with how the ARFrontier and AbstractFrontier keep track of some statistics. For example, the total number of queued URIs is calculated in the ARFrontier by adding up the size of the queues whereas the AbstractFrontier retains a central value for this. The difference in maintaining statistics could probably be overcome by either overriding or ignoring the AbstractFrontier's data. However, it adds another level of complexity to any attempt at basing the ARFrontier on the AbstractFrontier.

Finally, there is some (fairly minimal) assumption in the AbstractFrontier that BdbWorkQueues are being used. Notably in the method *noteAboutToEmit()* which is primarily used for the recovery journal implementation. This would need to be addressed in the AbstractFrontier itself before the ARFrontier could be modified to subclass it. Clearly any truly *abstract* Frontier must be free of such implementation specific objects.

The above summation may not be an exhaustive summary of the difficulties in using the AbstractFrontier, it merely relates the most obvious difficulties that were discovered by reviewing the code of the AbstractFrontier. Other, more serious design incompatibilities may be discovered if and when an attempt is made to base the ARFrontier on the AbstractFrontier.

In the long run, the ARFrontier will almost certainly be modified to subclass the AbstractFrontier. The benefits in terms of simplified code maintenance are considerable and ultimately well worth the initial investment. However, it was clear that this will not only require changes to the ARFrontier, but also significant streamlining of the AbstractFrontier to make it more agnostic towards different crawl strategies. Doing this, however, falls outside the scope of this project.

6.3 New Processors

While the ARFrontier implements most of the incremental strategy, it needs be told how long to wait before URIs can be revisited. For this purpose three processors were created, HTTPContentDigest, ChangeEvaluator, and WaitEvaluator. In order to do a crawl with the ARFrontier these need to be inserted into the processing chain (see Figure 11) at the appropriate locations.

That is not to say that these processors *in particular* are needed in order to use the ARFrontier, but rather, some processors that perform the same functions are needed. This corresponds to the general use of processors in Heritrix, the ones provided are solid implementations of common functionality. They can however easily be replaced by custom written modules if we wish to modify their behavior.

There was no need to modify or replace any of the existing processors for the adaptive revisiting strategy, since the processing of URIs remain largely the same. We only needed to add some additional processing.

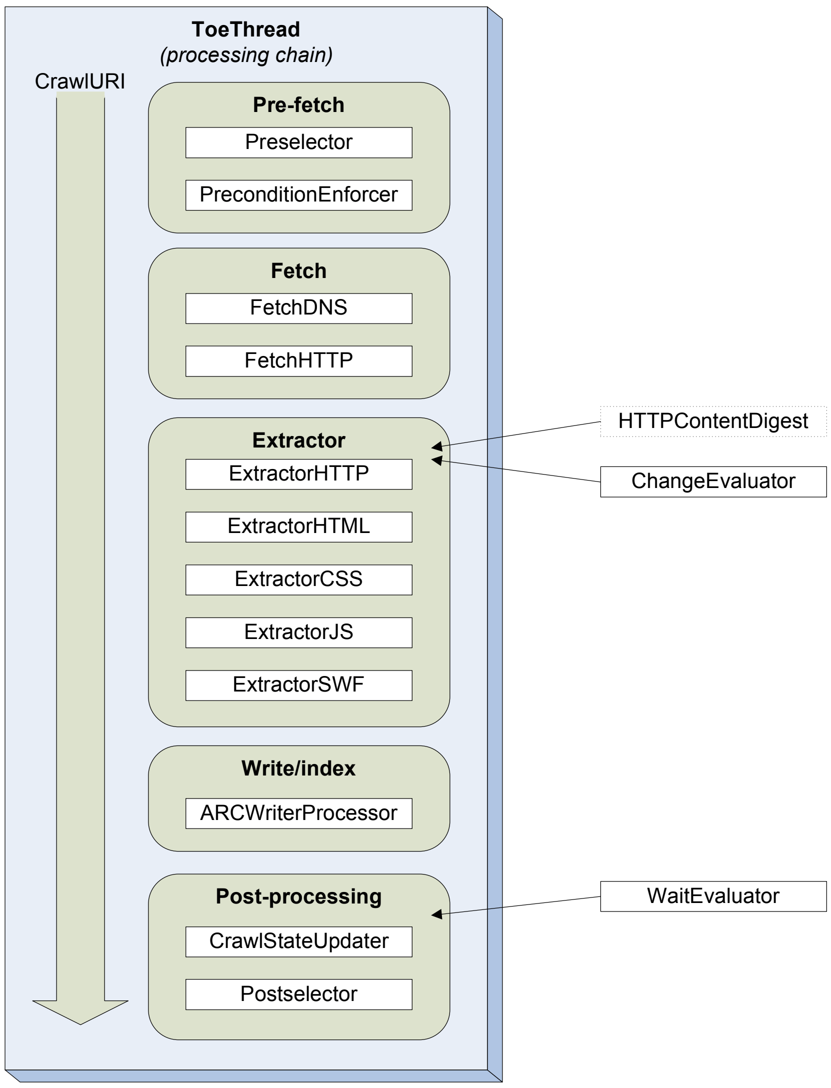


Figure 11. Illustrates how the processors for the adaptive revisiting strategy fit into a typical processing chain. The HTTPContentDigest processor is optional, but the ChangeEvaluator and the WaitEvaluator are required.

6.3.1 ChangeEvaluator

The ChangeEvaluator compares a downloaded documents hash to the hash from a previous visit to determine if the document has changed. It assumes that this hash has been calculated and stored in the CrawlURI (accessed via the *CrawlURI.getContentDigest()* method.) This is generally the case since the FetchHTTP processor automatically calculates the hash when it downloads documents.

The ChangeEvaluator does not care about the type of hash, since it just compares two instances of it and rules that they are either identical or not, but typically these are SHA-1 [14] hashes, since that is what the FetchHTTP processor uses. However, changing the hashes will have no impact on the ChangeEvaluator so long as they a very low probability of a collision where two dissimilar documents compute as having the same hash.

The ChangeEvaluator stores the content state in the CrawlURI's keyed list. Initially the idea was to add content state to the CrawlURI object itself, defaulting it to *unknown*. However, this was not accepted when merging the AR code into Heritrix's main code base, mostly because it was unclear if any other modules might ever be interested in it. This may be revisited again at a later date. Instead, the content state is recorded in the keyed list, which is perfectly satisfactory. The only potential down side is that other modules, processors in particular, are less likely to take advantage of it.

If no content hash is provided the ChangeEvaluator does not make any ruling, leaving the content state as unknown. Currently this applies only to DNS lookups.

Currently, the ChangeEvaluator will preempt the processing chain if the document is discovered to be unchanged. This moves the URI straight to the post processing section, bypassing the link extractors and write/index processors. This is quite satisfactory at the moment, but in the future there may be write/index processors that are interested in recording that an unchanged document was visited at a given point in time. To achieve this, the CrawlURI's processing should not be interfered with, but all processors running after the fetchers should check the content state and decide based on it whether they should run or not. Some would always run, such as the Postselector, but extractors would only run on changed documents and those whose state is unknown.

In fact, the ChangeEvaluator already behaves in this manner. If the content state is already set on a CrawlURI, it will only update the counters of how many visits have been made to the URI and how many versions encountered in those visits. It will not reevaluate if a change has occurred.

In practice, this happens when the download of a document is skipped because HTTP header information indicates that no change has occurred.

If the ChangeEvaluator decides to preempt the processing chain, having discovered an unchanged document, this is noted in the crawl.log by writing *unchanged* to its annotations field.

6.3.2 WaitEvaluators

The WaitEvaluator implements the actual adaptive logic. Basically the WaitEvaluator checks if the document has changed or not, and based on that it multiplies or divides the wait interval by a certain factor. For documents whose content state is unknown a default, unchanging, wait interval is set. Essentially, if you wanted a non-adaptive incremental crawl you could simply omit the ChangeEvaluator and then the crawl would repeat at intervals dictated by this default value.

The initial wait interval, default wait interval, and the factors for changed and unchanged documents are all configurable settings. Additionally, the WaitEvaluator allows the operator to specify maximum and minimum wait intervals. This can be used, for example, to prevent constantly changing pages from 'clogging' the system. It also ensures that all documents are revisited at some point. Without a maximum wait interval the wait interval of a document that was never observed to have changed would grow exponentially over time, eventually we'd stop visiting it. While this fits in with the adaptive concept, in practice we usually want to at least get one complete copy every year.

Finally, the WaitEvaluator has a toggle that allows the overdue time to be used when a new wait interval is calculated. If this option is selected (off by default) then the amount of time the URI was overdue for processing will be added to its wait interval before the factor is applied to it. This feature was added because in practice a lot of URIs, especially on larger crawls, will be overdue since there are too many URIs that need processing. This problem is especially noticable during the early part of a crawl when the adaptive algorithm is still learning what documents rarely change and is slowly increasing their wait time. This feature can significantly speed that up. However, the feature can have the paradoxical effect of increasing the wait time of changed documents if they were significantly overdue. On the other hand, it could also be said that the feature allows the crawl to adapt to the resources that are available.

All this affords the operator full control over the adaptive behavior. Furthermore, the settings can all be overridden for any host using Heritrix's overriding schemes. Thus the adaptive behavior can easily be tailored for each domain and host.

The above is enough to implement all the aspects of the adaptive algorithm discussed in the last chapter, except one. It does not allow the operator to discriminate in the settings based on the content type.

The initial idea to tackle that subject was to use Heritrix's refinements framework. A new criteria could be added that compared a regular expression to the documents content type and if it matched the values in the refinment would override the default.

Only, this approach had two problems.

One, the refinement criteria were only being passed UURIs and not CandidateURIs or CrawlURIs. This meant that the content type was not available to them. A solution for this was found, but it required some changes to numerous classes, both in the settings part of Heritrix and in the canonicalization. While this approach was initially pursued, the extent of the changes needed did not become apparent until it was time to integrate it into the main codebase. Therefore, despite the fact that this had been successfully implemented, this approach was abandoned. Largely because of fears that introducing such a large change to the code could be problematic, especially when a release was on the horizon, but also because of the other drawback of this approach.

While the refinement approach is an exceptionally economical solution, taking advantage of existing capabilities, it can make it difficult to determine exactly what rules apply to what URIs at any given time. This is largely a result of the obfuscated nature of the interaction between overrides and refinements. Basically, refinements are not inherited into overrides. Therefore setting up a refinement to specify different wait times for images would be lost when we created an override to specify different wait times for a specific host. This is how the system was intended to work, but it can be confusing with a large number of overrides and refinements.

A better approach was therefore needed. Using a similar approach as the extractors, where a number of processors exist to do the same thing, only on different files, we created some new, specialized, WaitEvaluators.

First the generic WaitEvaluator was subclassed to create the ContentBasedWaitEvaluator. It adds a regular expression, like the refinement criteria would have, that is compared against the documents content type and only if it matches is the evaluation carried through.

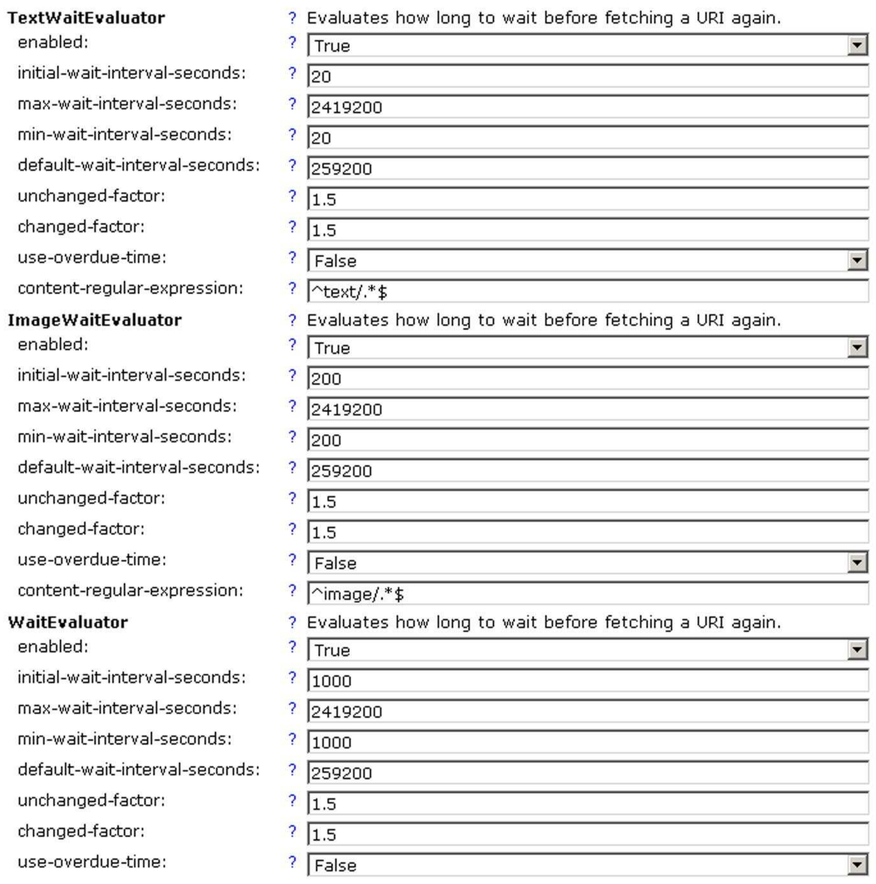
Based on this, TextWaitEvaluator and ImageWaitEvaluator were created with the regular expression configured to match text and image files respectively. The regular expressions though can still be changed if the operator wishes. The default values for the initial wait have also been modified for these.

The WaitEvaluator was then modified to set a flag in the CrawlURI once it had been run and to pass on objects that had this flag. This ensures that once one WaitEvaluator has been run, others will ignore the object.

With all of this in place, it is simply a matter of 'stacking up' the WaitEvaluators, starting with the specialized ones and ending with the generic WaitEvaluator which will process anything not handled by the others.

While only two specialized evaluators are included, they can be modified to fit other criteria, and the ContentBasedWaitEvaluator can also be used. If more than three specialized evaluators are needed, it is a simple matter to create additional ones by subclassing ContentBasedWaitEvaluator. Doing that will also allow for new default values, more applicable to the task at hand.

Figure 12 illustrates how these evaluators might be configured.



TextWaitEvaluator	?	Evaluates how long to wait before fetching a URI again.
enabled:	?	True
initial-wait-interval-seconds:	?	20
max-wait-interval-seconds:	?	2419200
min-wait-interval-seconds:	?	20
default-wait-interval-seconds:	?	259200
unchanged-factor:	?	1.5
changed-factor:	?	1.5
use-overdue-time:	?	False
content-regular-expression:	?	^text/.*$
ImageWaitEvaluator	?	Evaluates how long to wait before fetching a URI again.
enabled:	?	True
initial-wait-interval-seconds:	?	200
max-wait-interval-seconds:	?	2419200
min-wait-interval-seconds:	?	200
default-wait-interval-seconds:	?	259200
unchanged-factor:	?	1.5
changed-factor:	?	1.5
use-overdue-time:	?	False
content-regular-expression:	?	^image/.*$
WaitEvaluator	?	Evaluates how long to wait before fetching a URI again.
enabled:	?	True
initial-wait-interval-seconds:	?	1000
max-wait-interval-seconds:	?	2419200
min-wait-interval-seconds:	?	1000
default-wait-interval-seconds:	?	259200
unchanged-factor:	?	1.5
changed-factor:	?	1.5
use-overdue-time:	?	False

Figure 12. The UI settings for three WaitEvaluators. Two specialized ones, text and images, and one general purpose evaluator for everything else. The values for initial wait times and minimum wait time are extremely small and would likely be much higher in a crawl of any magnitude.

The WaitEvaluators need to be in the post processing section of the chain, since they should be run on both changed and unchanged documents.

6.3.3 HTTPContentDigest

As discussed in the previous chapter, when we are trying to detect if a change has occurred, we may wish to ignore certain sections of the document. The HTTPContentDigest processor adds that ability to Heritrix.

The FetchHTTP processor automatically creates a hash based on the contents of the document. Normally, this is all we need, but if there is a known section in certain documents that contains problematic content, the HTTPContentDigest can be set (via overrides and/or refinements) to recalculate the hash for those documents. Prior to doing the recalculation, the processor will apply a regular expression to the document and replace any section that matches the regular expression with a single blank character. This replacement is actually done on a copy of the document and the downloaded version remains intact for other processors. Effectively, the hash is thus calculated based on a document with the problematic sections erased.

Using this processor is expensive, both in terms of processor power and memory so obviously it should be configured so that it is only applied to exactly those documents that may require it. Careful use of overrides, refinements and filters is essential here. Both to ensure that it is only applied to documents requiring it and also in order to be able to vary the regular expression since different sections will need to be dealt with in different documents. For example, using it on non-text documents will likely be meaningless. As with all processors, filters can be placed on it to limit what URIs it handles and thus the *ContentTypeRegExpFilter* could be used to ensure that the processor is only applied to text documents.

The processor has a setting that allows the operator to specify a maximum size for the documents that it processes. This reduces the change that abnormally large documents cause the crawl to slow down when this processor suddenly needs to handle them. This setting defaults to one megabyte.

The HTTPContentDigest processor must be applied after a document is downloaded and prior to the ChangeEvaluator.

6.4 Using HTTP headers

As discussed briefly earlier, the FetchHTTP allows the operator to specify filters that are applied after the HTTP headers have been downloaded but before the content body. This makes it possible to write filters that reject downloading a document based on its HTTP header.

This is exactly what was suggested in the last chapter. By using the *Etag* and *datestamp* information, a prediction can be made as to whether or not a document has changed.

That is what the cumbersomely named HTTPMidFetchUnchangedFilter does. Basically, it implements scheme 4 [3], where if both values are present they must agree on predicting no change, otherwise a change is predicted. This scheme was chosen since it had the highest reliability of any approach that considered both values. In the future the option of choosing any of the four proposed schemes should be added.

The filter stores the values of the headers in the CrawlURI's keyed list so that old values can be compared to newer ones. Any disparities in the stored and current values are interpreted as indicating a change. This means that even if the datestamp becomes 'older,' we'll regard that as a change.

By returning false, which it does if no change is predicted based on the HTTP headers, the filter will cause the FetchHTTP processor to abort the downloading of the document. A note to this effect *midFetchAbort* will be made in the log.

The filter will also set the content state to *unchanged* if this occurs, but it does not change the state if it predicts a change since the rate of false positives on that is extremely high. The ChangeEvaluator continues to handle those.

7. Results

Initial test crawls have gone very well, with the software exhibiting good stability running over the course of several days. Depending on the initial and minimum wait settings, as well as politeness restrictions, it is necessary to limit the size of the scope for each host [20] otherwise, as expected, the adaptive algorithm is unable to work effectively since there are always more URIs that need to be crawled, then time available for crawling. Also, if the number of hosts is excessive, local machine resources may become a limiting factor.

Unfortunately, no suitable metric exists to measure the performance of such crawls. We can say that it behaved exactly as expected when running against a website that we controlled, so the adaptive algorithm is sound. The progress rate of the test crawls was always limited by politeness factors, rather than machine resources, so we also know that the ARFrontier can crawl with a modest number of hosts at a respectable pace.

As discussed earlier, this was to be expected and is a direct consequence of politeness restrictions and limited local resources. Effectively, there is a maximum number of documents that can be downloaded from any one host in a given amount of time. Therefore, there are only two ways to increase the size of the scope on each host. Either you reduce politeness restrictions or increase the wait times between revisits.

It helps to configure the crawl as accurately as possible in terms of expected wait times, declaring images and such with higher wait times. However, there is clearly a link between the size of the scope and a requirement for a minimum wait time. As the scope grows, so also must the minimum wait time, or rather the average wait time of all documents belonging to the host.

Future work will undoubtedly focus on trying to establish good values to balance expected wait times and the size of scopes. Our initial crawls, while supporting our assumptions on the overall crawl behavior are far from extensive enough to be able to provide significant insight into further optimization.

The fact that the ARFrontier supports a host valence higher than 1 allows for very aggressive crawling. While this is generally to be avoided, there may be scenarios where we have permission to crawl more aggressively. This feature allows us to double or triple (or more) the number of documents belonging to the host that can be visited.

The maximum number of hosts depends greatly on the hardware available. Since politeness restrictions do not reach across hosts, they only affect each others scheduling when too many hosts need to process URIs at once. Due to politeness restrictions, each host is unlikely to crawl more than 2-3 URIs per second (although this can vary). Heritrix has shown an ability to crawl upwards of 50-60 documents per second on decent hardware. This would seem to limit the number of hosts to about 20-30. However, in practice most hosts are unlikely to require the fetching of 2-3 URIs per second, especially not over longer periods. The initial discovery phase will probably generate a significant backlog as URIs are discovered much faster than they can be processed, much like in snapshot crawl. However, assuming that the initial wait times are moderate, the backlog should be mostly completed before the first round of repeat visits.

The total number of hosts then depends on the kind of load each host is likely to place on the system. Their politeness settings and the number of URIs for each host dictates the maximum number of documents that they may need to crawl per second. The wait times are likely to further decrease this unless the site is changing very often and minimum wait times are low.

While not tested extensively, it seems perfectly plausible to have at least a hundred hosts running in parallel. The size of each host depends on how often the documents need to be revisited. If the wait time is generous in terms of the size of the host then this will free up additional resources to crawl other hosts.

Finding an appropriate balance for these factors is still a work in progress. No practical crawling has been performed yet but it is planned as we discuss below. These will likely evolve over time as experience is gathered. It is also quite likely that the adaptive heuristics will be improved to look at more than just the last version.

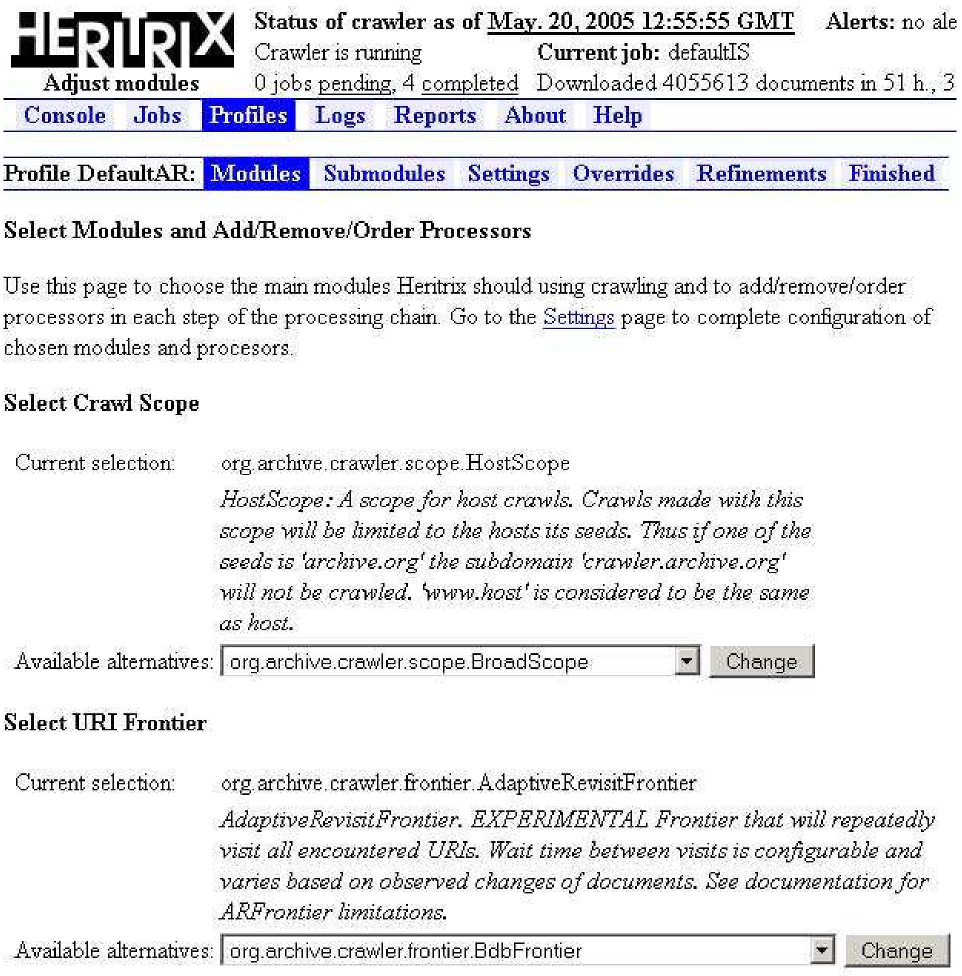
During test crawls, some interesting behavior has been observed. As expected, pages with constantly changing sections have been discovered. For example, a news site with a stock 'ticker' that changed constantly during trading hours. There was also a page that only displayed the ticker. One might argue that their content is changing with each visit, but it's doubtful if we want to record it that closely. The stock ticker probably isn't an issue if the minimum wait time is measured in hours, since the news site itself changes quite rapidly. However, if the minimum wait time is very low, a minute for example, then this becomes an issue.

On another site pages were observed with content that changed randomly with each visit. The site offered a form of 'yellow pages' index service of businesses. The pages in question were category front pages which displayed several randomly chosen entries from their categories. A crawl of this website would have to be configured either to disregard the entire body of the document when creating comparison hashes or, more likely, to visit these pages only at fixed, infrequent intervals.

Test crawling was conducted with very limited scopes for each host. Basically, it only allowed two to three link hops (varied between crawls) into each host and disallowed crawling of any offsite embedded documents. Even with this shallow scope, many sites turned up thousands of documents, indicating that the websites have a fairly broad structure. This is in line with what we had expected.

The decision to disallow offsite documents was largely a matter of practicality. An offsite document that has been crawled once would continue to be revisited again and again, long after the reference to it has been removed from one of the sites we are targeting. Unfortunately, Heritrix does not provide information about how URIs were judged to be in scope, so we can not differentiate between these and regular URIs.

This project was developed in a separate branch of the Heritrix project and was initially made available to the public in December 2004. At the time the software contained many bugs, but nevertheless drew some attention from third parties interested in testing. By March 2005 the software was considered essentially stable and towards the end of that month work began to integrate it into Heritrix's main code base. This was done in preparation for the release of version 1.4.0 of Heritrix in late April. The ARFrontier is currently labeled as being experimental, much as the BdbFrontier was in 1.2.0, since it has not yet received widespread testing and use.


Status of crawler as of May. 20, 2005 12:55:55 GMT **Alerts:** no ale

Crawler is running **Current job:** defaultIS

0 jobs pending, 4 completed Downloaded 4055613 documents in 51 h., 3

Adjust modules

Profile DefaultAR:

Select Modules and Add/Remove/Order Processors

Use this page to choose the main modules Heritrix should using crawling and to add/remove/order processors in each step of the processing chain. Go to the Settings page to complete configuration of chosen modules and procesors.

Select Crawl Scope

Current selection: org.archive.crawler.scope.HostScope

HostScope: A scope for host crawls. Crawls made with this scope will be limited to the hosts its seeds. Thus if one of the seeds is 'archive.org' the subdomain 'crawler.archive.org' will not be crawled. 'www.host' is considered to be the same as host.

Available alternatives: org.archive.crawler.scope.BroadScope Change

Select URI Frontier

Current selection: org.archive.crawler.frontier.AdaptiveRevisitFrontier

AdaptiveRevisitFrontier. EXPERIMENTAL Frontier that will repeatedly visit all encountered URIs. Wait time between visits is configurable and varies based on observed changes of documents. See documentation for ARFrontier limitations.

Available alternatives: org.archive.crawler.frontier.BdbFrontier Change

Figure 13. The UI page for configuring crawl modules. The AdaptiveRevisitFrontier has been selected as the Frontier.

With its integration into Heritrix completed we believe that it will form the basis for future incremental crawling amongst those parties that are currently using Heritrix. In particular, the National and University Library of Iceland will be utilizing it to augment its regular crawls of the .is country domain [16]. These crawls will consist of approximately 30 hand picked sites. The selection criteria is still being considered, but the sites chosen for this are likely to be ones containing news, political discussions and other similar content of obvious interest.

8. Unresolved and future issues

A considerable amount of work remains to determine what settings yield the best result. Volatile change rates, in particular, are likely to make this hard. 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 time [20].

The subject of improving the adaptive heuristics is in its infancy. The strategy developed here has tried to (potentially) capture *all* changes, with the minimum wait times possibly dropping as low as one second. This is, of course, left to operator discretion, but considering the frequent changes of some sites, may be needed.

However, as the wait time drops, volatile change rates over the course of a single day become a factor. The change rate may be heavily dependant on the time of day [4]. It might therefore be best to use information from (at least) the past 24 hours when estimating wait times. In fact, it is probable that some advantage could be gained from viewing an even larger sample. Knowledge of time of day, weekdays versus weekends, public holidays etc. could all be incorporated, in theory, to improve the wait time heuristics.

The possibilities for analyzing such data and even cross-referencing it with data from other, similar, documents are almost endless. Such cross-referencing could both be useful to take into account site specific trends, by comparison to other documents from the same site, and trends related to the document type, by comparing to documents of similar nature on other sites.

This is likely to be quite expensive in terms of processing power, not to mention requiring a significant amount of work to determine appropriate linkages to utilize. Furthermore, Heritrix's current architecture does not allow processors to access meta-data on URIs, other than the one they are currently working on. Incorporating such advanced features would require some augmentation on its part.

Change detection also remains a significant problem. We have provided a simple strict hash approach, with the option of selectively ignoring sections in documents known to be troublesome. This works quite well most of the time, but has definate drawbacks, especially when crawling a relatively large number of hosts and URIs where the operator is less able to manually notice and address all problems.

During a meeting held by the National and University Library of Iceland with several large content providers, they expressed an interest in facilitating change detection by declaring the sections in their web pages with tags that remain to be decided. The idea was that these tags would denote any troublesome sections and our incremental crawls would be configured to have the hashes overlook them. The content providers' benefits would be reduced load on their servers without compromising the quality of their content in the library's archives. This approach, while workable on a small scale, would clearly be of little practical use for larger crawls, spanning hundreds, if not thousands of sites.

Improved change detection is a very large and complex issue. A comprehensive overview is well beyond the scope of this project. We can however 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 revisit algorithm will be. However, that still won't address the problem where some content in a document is of trivial importance. For example, when archived stories at a news site contain a section that tells the reader about the current 'front page' story. The current front page story changes with great frequency, but it is captured when we crawl the actual front page and its inclusion in archive pages is essentially trivial.

As previously discussed, it has been suggested that 'close enough' comparison algorithms [6] may overcome this to some extent. They operate by overlooking minor changes in the documents, for example, by splitting them up into a certain number of overlapping sections, or shingles, and checking if there is sufficient number of identical sections. If there are then the document is considered to be 'close enough' and is regarded as unchanged.

While the 'close enough' approach seems attractive, it does have a notable drawback; it assumes that small changes are never of consequence. In practice, however, it is easy to think of significant issues where a small change is of considerable interest.

Using ‘close enough’ detection may however be a suitable compromise for larger crawls. The alternative is to overcrawl, possibly do so quite heavily. Also, if a ‘close enough’ hashing is added to Heritrix, it is fairly simple to configure different sites to use it or the strict hash. This way sites known to be well behaved could continue to use the strict hash, while other sites, possibly of lesser interest, would use ‘close enough’ comparisons.

Unfortunately, HTTP header information does little to improve this situation. In practice, dynamically generated content is the most likely one to be missing good header information on content change for practical reasons. Dynamically generated content is also more easily modified and thus more likely to change. 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 dramatic improvement in the use of HTTP headers by websites, it is unlikely that they will be of greater use than what is provided 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 predicting change rates based on a document’s history of change. However, there may be other factors that significantly affect this. As mentioned earlier, a documents location within a site can be significant. “Front pages” usually change much more often than archives. This probably varies somewhat from site to site, but adding some discrimination, perhaps based on the distance from seed or the length of the URI string, might improve the initial wait time.

Alternatively (or additionally) a ranking system could be devised that takes diverse factors such as number of links to and from a document, presence of keywords, etc. into account. Developing such a ranking system would require considerable effort and implementing it is also likely to require additional changes to Heritrix. Again, this may vary from one website to another, and also on the purpose of the crawl. The ranking could be biased to favor pages that we deem important on some basis, such as the presence of keywords.

Looking even further ahead, 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 practical note, Heritrix's statistics remain heavily snapshot oriented. Especially those provided by the web user interface. We took advantage of options provided by the crawl.log to record important additional material and this works quite well. We also implemented a Frontier report that is accessible via the user interface that contains various useful data on the state of the crawl.

Heritrix however provides a considerable amount of 'progress' data. Much of this fits only marginally with incremental crawling since the data doesn't really account for multiple visits of the same URI. For example, the main *Console* page offers a progress bar that shows the percentage of completed URIs against the number already completed and those that remain queued. Clearly an incremental crawl will never be completed so this progress bar is of little actual value.

None of the above issues are acute problems, but it means that monitoring an incremental crawl requires some additional interpretation of the data being provided. In the future, Heritrix may offer the possibility for modules to provide customized web pages for control and display of data. This is not on the immediate work schedule, but once implemented, statistics reporting on incremental crawls can be significantly improved and customized to fit its needs.

None of these problems should affect later day analysis of a crawl. The progress data only complicates the monitoring of a crawl. Analysis work is done based on the logs, and, as noted earlier, they reflect the actual progress quite well.

9. Acknowledgements

I would first like to thank my supervisors, Helgi Þorbergsson for his guidance and support and Þorsteinn Hallgrímsson for his tireless support of this and related projects and making it possible for me to undertake this task. Also I'd like to give special thanks to Gordon Mohr and Michael Stack of the Internet Archive for their assistance during the development phase. Finally I'd like to thank the IIPC 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/
14. *FIPS 180-1 – Secure Hash Standard*. Accessed May 2005. http://www.itl.nist.gov/fipspubs/fip180-1.htm
15. Þorsteinn Hallgrímsson: *Varðveisla íslenskra vefsíðna*. Bókasafnið, 26. árgangur 2004, p. 16-22
16. Kristinn Sigurðsson: *Söfnun vefsíðna og Heritrix*. Unpublished
17. *Heritrix FAQ.* Accessed April 2005. http://crawler.archive.org/faq.html
18. *Java Management Extensions*. Accessed April 2005 http://java.sun.com/products/JavaManagement/

19. *Reglugerð 982/2003. Reglugerð um skylduskil til safna*. Accessed April 2005. www.reglugerd.is
20. Kristinn Sigurðsson, Incremental crawling with Heritrix. Unpublished
21. *Internet Archive: Wayback Machine*. Accessed May 2005 http://www.archive.org/web/web.php
22. *Kulturarw*^{3}. Accessed May 2005. http://www.kb.se/kw3/Default.aspx
23. *Pandora Archive*. Accessed May 2005. http://pandora.nla.gov.au/
24. *Nordic Web Archive*. Accessed May 2005. http://nwa.nb.no/
25. *International Internet Preservation Consortium*. Accessed May 2005. http://netpreserve.org
26. *International Internet Preservation Consortium: Mission*. Accessed May 2005. http://netpreserve.org/about/mission.php
27. *Jetty Java HTTP Servlet Server*. Accessed May 2005. http://jetty.mortbay.org/jetty/
28. *Java Foundation Classes*. Accessed May 2005. http://java.sun.com/products/jfc/index.jsp
29. T. Berners-Lee et al: *Uniform Resource Identifiers (URI): Generic Syntax*. Accessed May 2005. http://www.ietf.org/rfc/rfc2396.txt
30. *JDBC Technology*. Accessed May 2005. http://java.sun.com/products/jdbc/