Incremental crawling with Heritrix

Kristinn Sigurðsson
National and University Library of Iceland
kristsi@bok.hi.is

IWAW 2005

Introduction

Crawl variations

(Heritrix dev team)

- Broad crawling
- Focused crawling
- Continuous crawling
- Experimental crawling

Crawling strategies

- Broad and focused crawling share many things
  - Differ primarily in scope *focus*
  - Both utilize a *snapshot crawling* strategy
- Snapshot crawling
  - Typically large scope
    - Broad or deep
  - Repeatable, but without using knowledge of previous crawls
    - Typically large intervals between repeats
  - Each URI visited once only

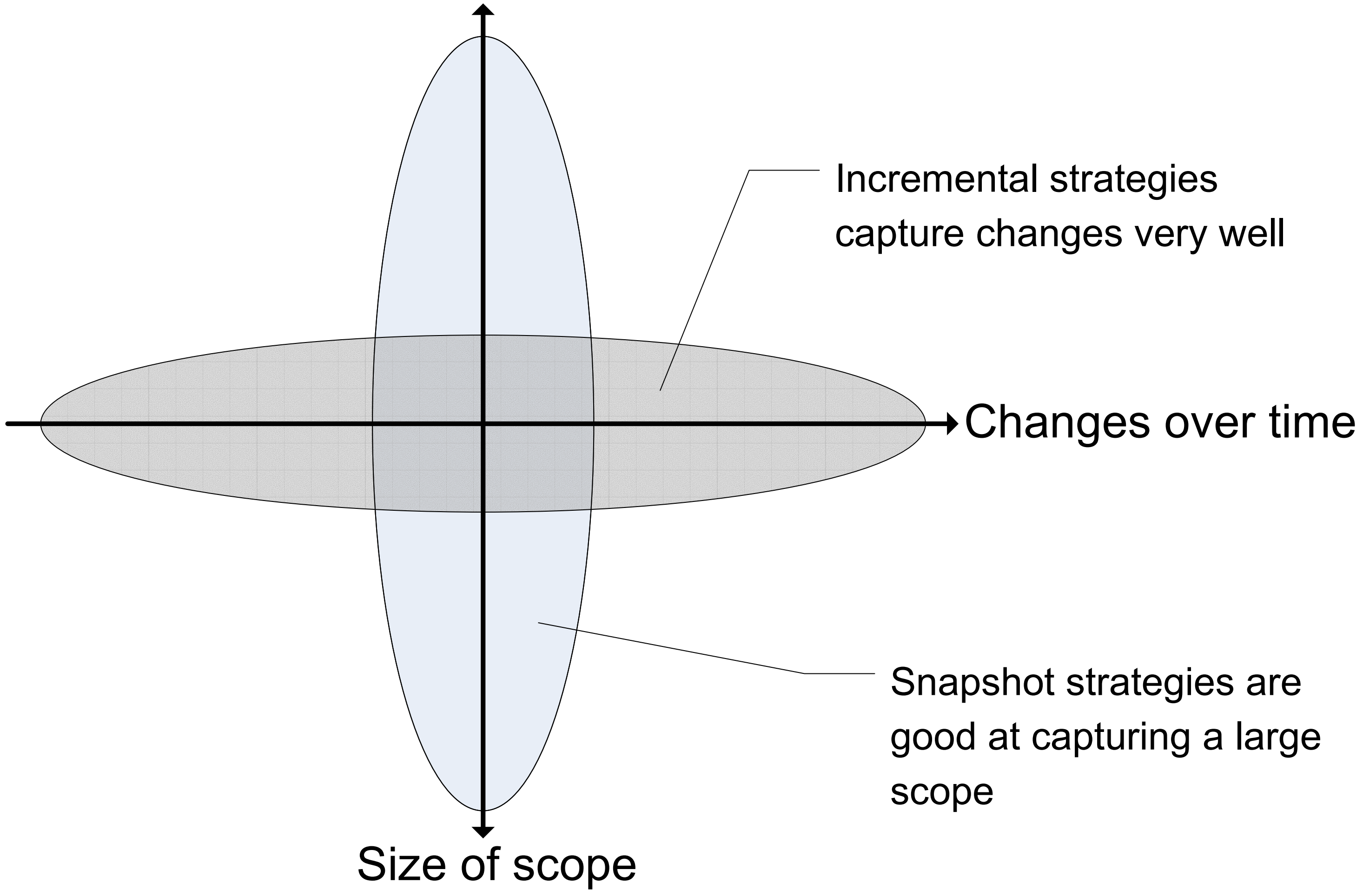
Crawling strategies cont.

- Continuous crawling
  - Requires visiting resources repeatedly within a single “crawl”
    - I.e. one run of the crawl software
  - Requires an *incremental crawling* strategy

An incremental strategy

- Each URI visited repeatedly
- Can do a good job of capturing changes in URIs
- Doesn't handle large collections as well
  - Needs to revisit URIs within a reasonable timeframe

Snapshot v/ Incremental


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

Heritrix

- Has decent snapshot capabilities
  - BdbFrontier
- Lacked all ability to crawl incrementally
  - Our purpose was to address this without compromising Heritrix's
    - Snapshot capabilities
    - Inherent modularity

The goal therefor

- Create an ‘add-on’ module for Heritrix that implements an incremental crawl strategy
  - Key issues:
    - How will this fit in with Heritrix’s architecture?
    - Defining a strategy

Defining a strategy

- Goal: Capture all changes
  - This is infeasible
- Periodic revisiting
- **Adaptive revisiting**
  - Adapting to observed change frequencies
  - Heuristic driven

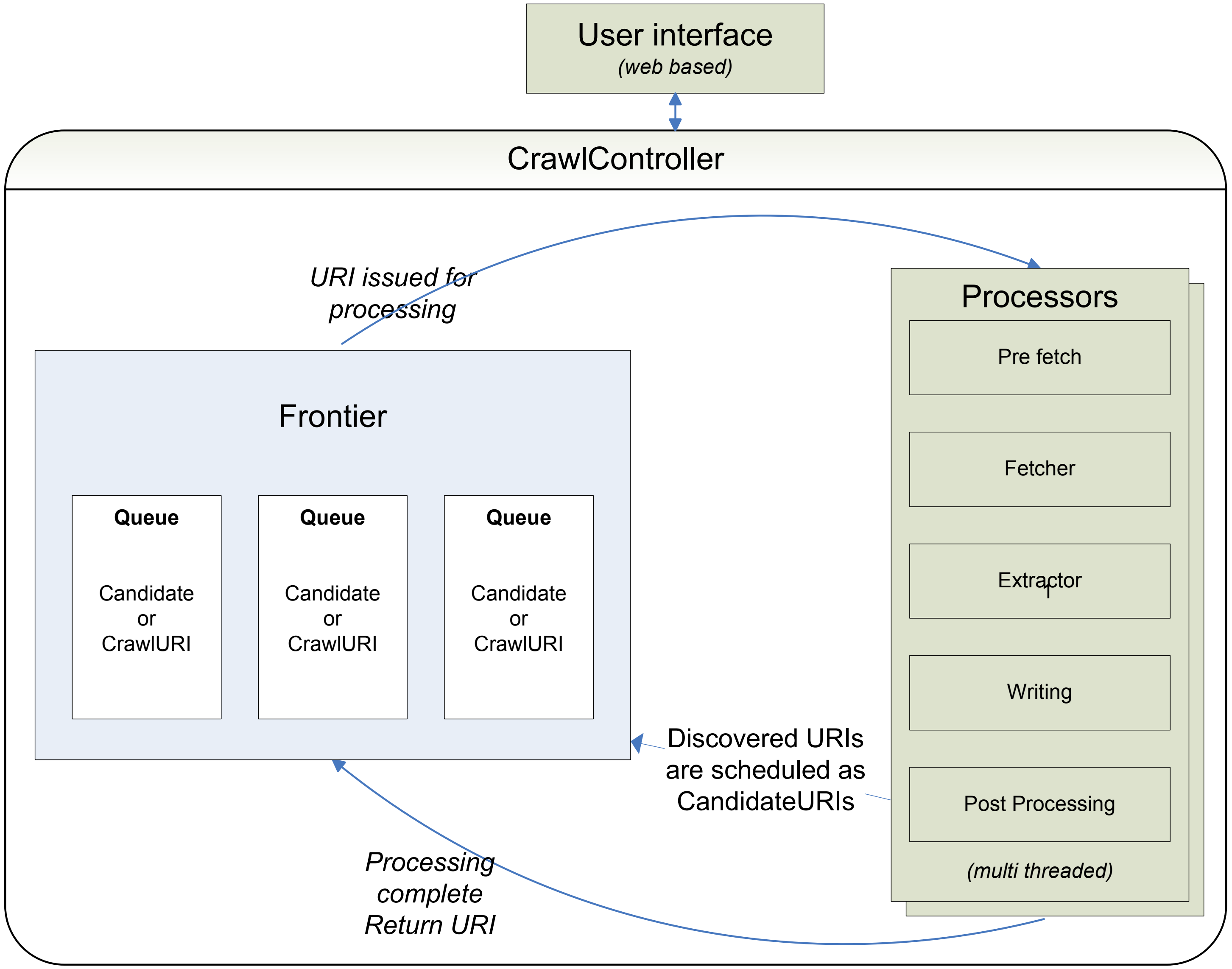
Heuristics

- Resources that change often are likely to continue to do so
- The file type of resources significantly affects the probable change rates
- Other
  - Document hierarchy
  - Presence or abscense of meta-data
    - Last-modified – resources that are missing this are about twice as likely to change as those with it
  - And many others

A strategy to implement

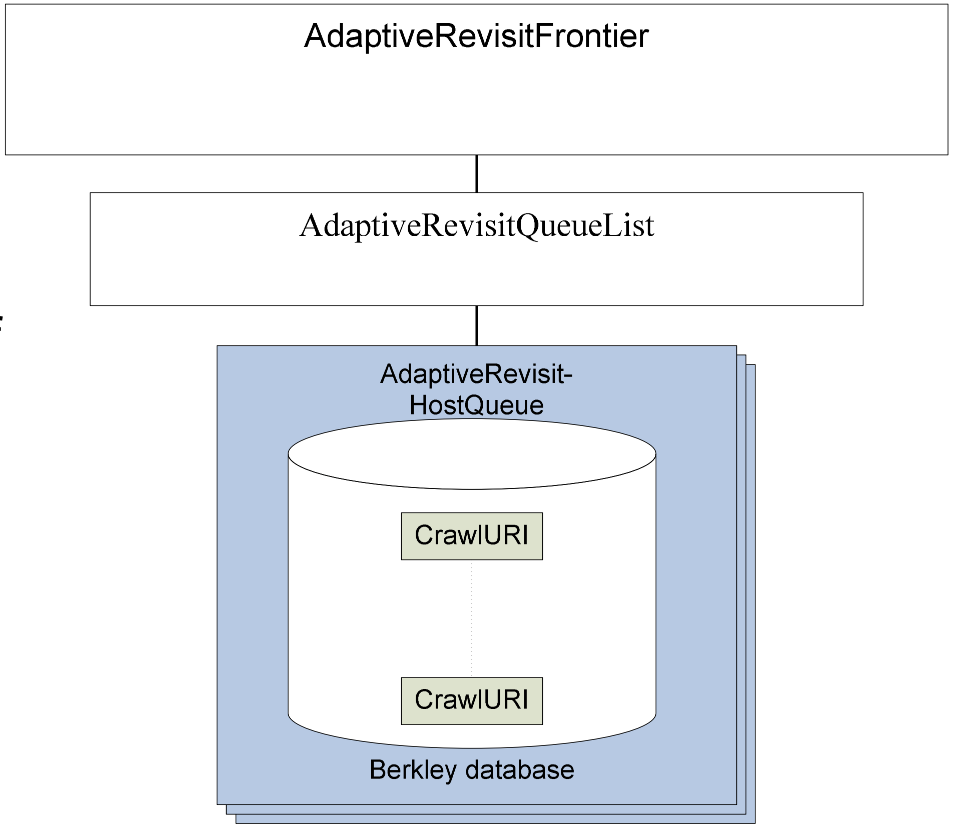
1. Scope is crawled (discovery)
2. Each crawled URI is assigned a revisit time
   - Initial wait interval depends on file type
3. After revisiting the wait interval is
   - Increased by a factor if change is detected
   - Decreased by a factor if no change is detected

Heritrix architecture (briefly)

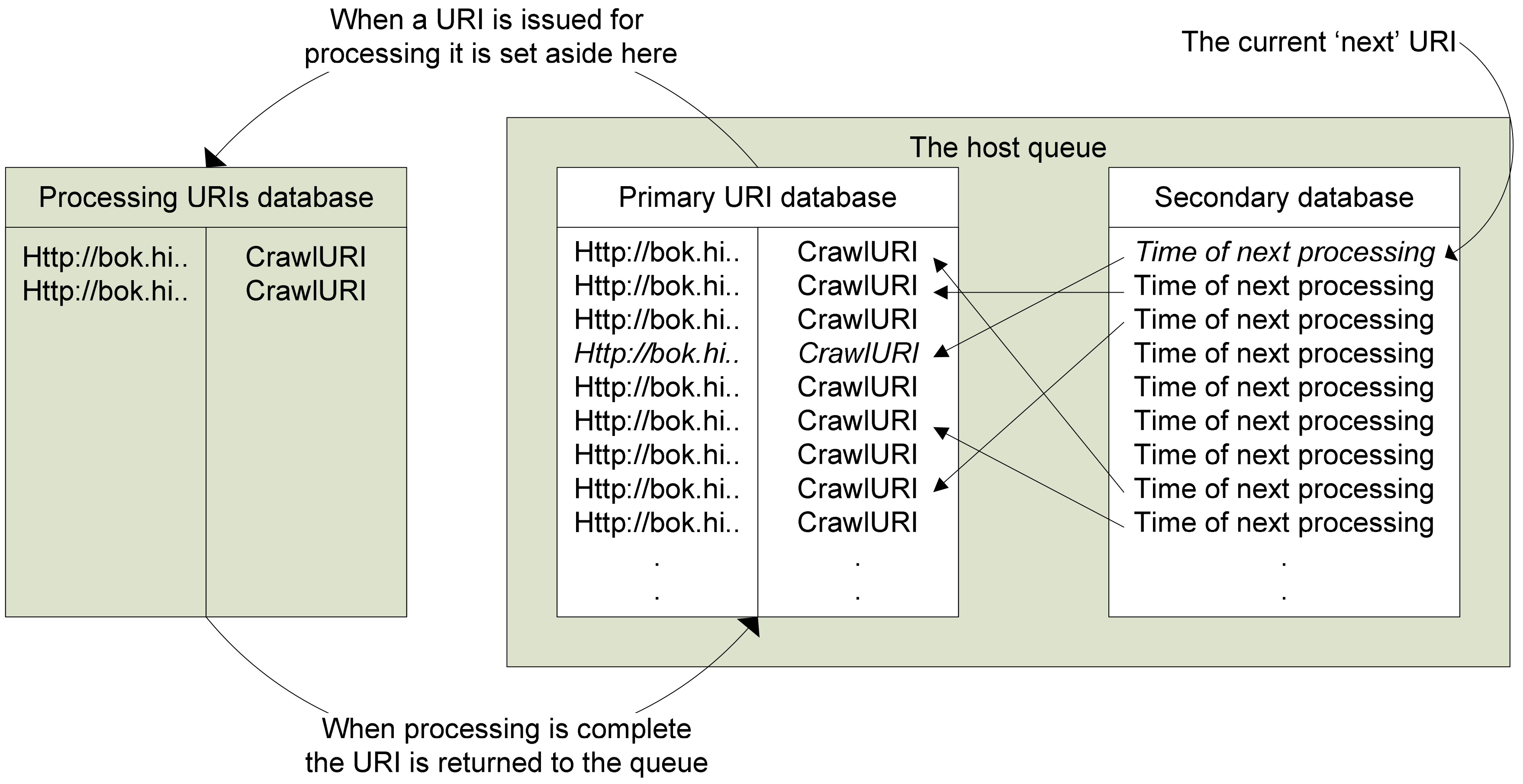


AdaptiveRevisitFrontier

- Relies on a series of host specific queues
  - Priority queues rather then FIFO
    - Priority based on ‘time of next processing’ and scheduling directive
  - Implemented using Berkley DB
  - Host ‘valence’ > 1 supported



HostQueues


When a URI is issued for processing it is set aside here
The current 'next' URI
The host queue
Processing URIs database
Http://bok.hi..
CrawlURI
Primary URI database
Http://bok.hi..
CrawlURI
Secondary database
Time of next processing
When processing is complete the URI is returned to the queue

ChangeEvaluator

- Compares the hash of the current document with a hash of the previous fetch, stored in the CrawlURI
  - Hash (SHA-1) is created by the FetchHTTP
    - Only works with HTTP protocol
  - The type of the HASH is unimportant

Change detection

- The ChangeEvaluator assumes that the hash provides a good indicator for change
- We know this may not be so
- The advantage of using a strict hash is that the probability of falsely assuming no change (i.e. missing a version) is virtually nill.
- However, we know that many (often automatically generated) changes do not represent changes in the actual *content*.

HTTPContentDigest

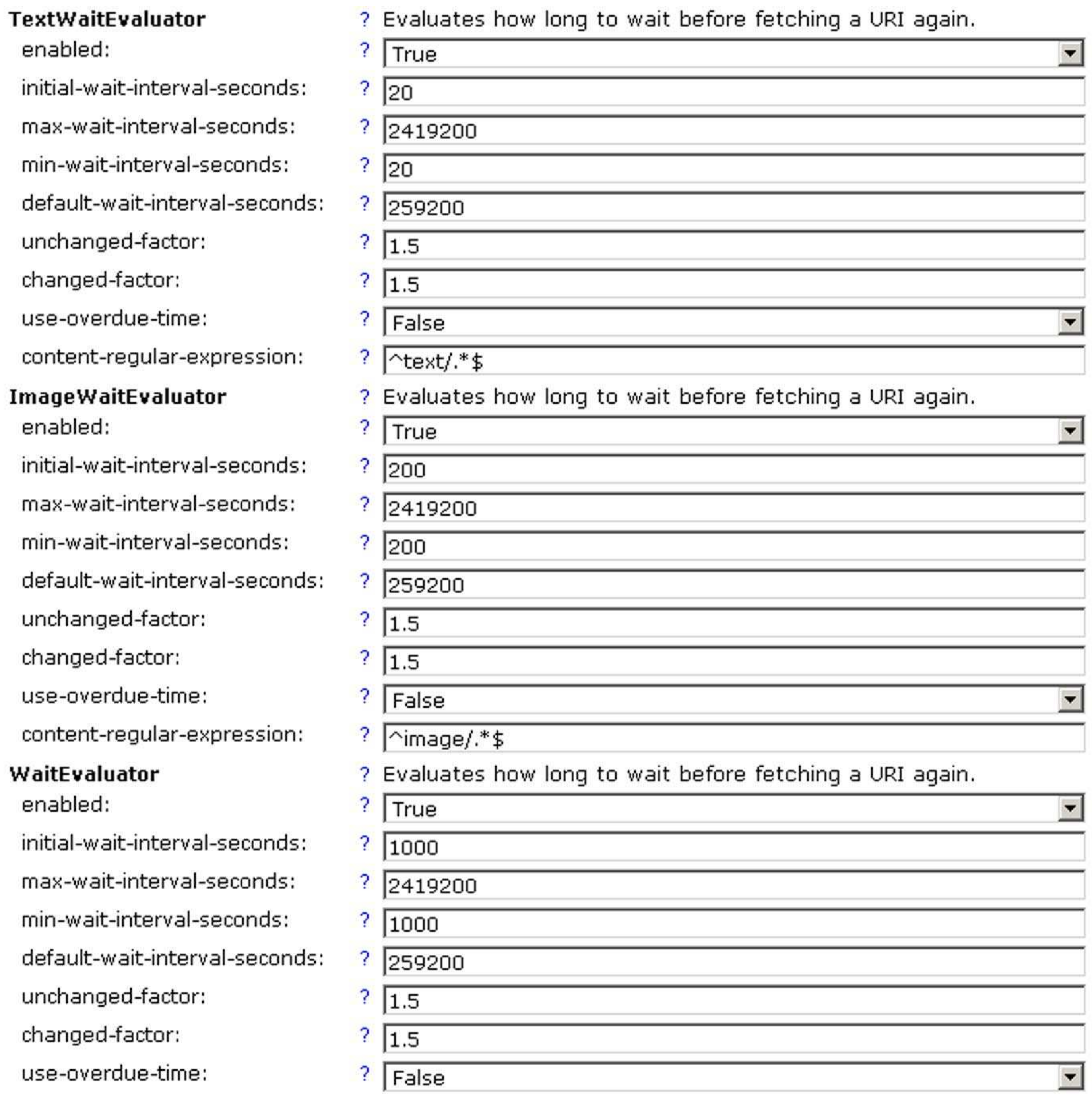
- Selectively weakens the content hash
  - User inputs a regular expression matching known problematic areas of documents
  - Downloaded document is processed and areas matching the reg.expr. are removed
  - Hash is calculated on the duplicate document thus created

HTTPMidFetchUnchangedFilter

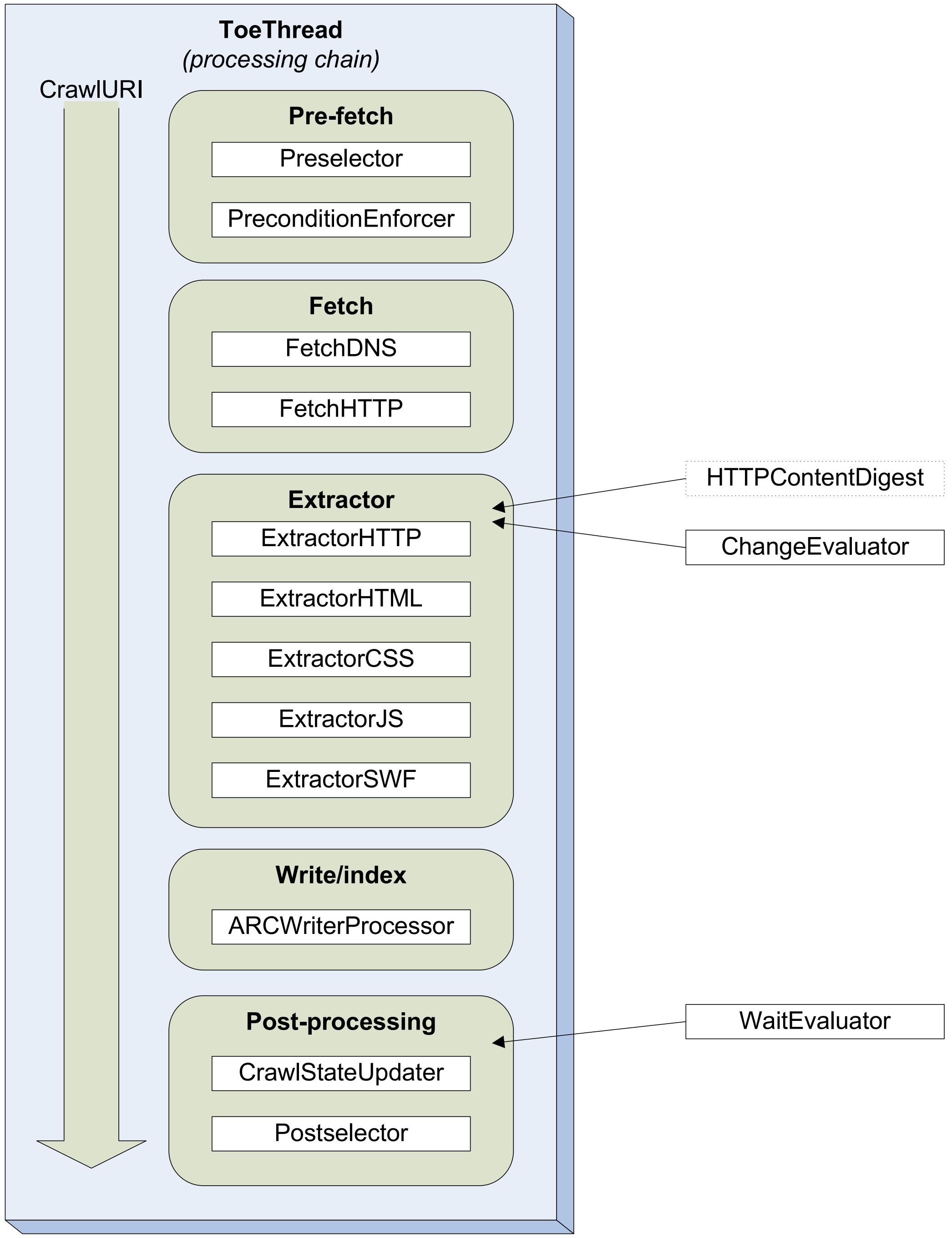
- Applied to the FetchHTTP processor
- Checks HTTP header
  - **last-modified**
  - **etag**
- If only one is present, the filter will determine that the document is unchanged if it is unchanged
- If both are present, they must agree that the document is unchanged
- If the filter decides that a document has not changed, it aborts the download of the HTTP document body

WaitEvaluators

- Implements the adaptive strategy
  - Determines the wait intervals for URIs
- Multiple WaitEvaluators are used
  - One for each document type
  - Document types are specified by reg.expr. matching the relevant mime types.



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


ToeThread
(processing chain)
CrawlURI
Pre-fetch
Preselector
PreconditionEnforcer
Fetch
FetchDNS
FetchHTTP
Extractor
ExtractorHTTP
ExtractorHTML
ExtractorCSS
ExtractorJS
ExtractorSWF
Write/index
ARCWriterProcessor
Post-processing
CrawlStateUpdater
Postselector
HTTPContentDigest
ChangeEvaluator
WaitEvaluator

Summary of implementation

- Highly modular
  - Easy to costumize any given aspect of an incremental crawl
- Using Heritrix's settings system of overrides and refinements a crawl can be very finely tuned

Results

- Initial crawl went very well
  - Frontier is stable
  - Crawls can be suspended and resumed easily
  - Performance (i.e. size of crawl) could be better
    - Crawling several thousand URIs per host over dozens of hosts is currently about as much as it can handle
- Included in Heritrix 1.4.0
  - Marked as ‘experimental’
- Work continues
  - Will be used for continuous crawling in Iceland

Specific issues

- Improve performance
  - Use of ‘fingerprint’ list of already included URIs
- Canonicalization support is limited at present

Future work

- Irregular change frequencies of the same URLs
  - Some websites are updated sporadically
    - Example: A politicians website
      - Updated often before elections, but rarely in between
  - The crawler will be slow to detect and adjust to these
  - Possible solution: Allow operators to ‘wipe clean’ or reset the wait interval for selected URLs or domains

Future work cont.

- Change detection
  - Very difficult topic
  - Explore ‘close enough’ comparisons
- Further testing and experimentation is needed
  - Fine tune the available parameters
    - What are good values
  - Explore using other/additional factors for evaluating wait times