Harvester design

Contents

Database

Description of the individual tables and their columns can be now found in the Derby SQL create script createfullhddb.sql.

Harvesting roundtrip

This section describes what goes on during a harvest, from the templates are uploaded and harvests created till the ARC files are uploaded to the archive and historical information about the harvest is stored.

Initial steps

To create a harvest in the first place, we need to have a template to base it on. Additionally, we need a schedule for a selective harvest or some domains to harvest for a snapshot harvest.

Uploading templates

Templates (Heritrix order.xml files) are uploaded using the **Definitions-upload-harvest-template.jsp** page. Templates are internally represented with the HeritrixTemplate object, which as part of its constructor verifies that certain elements - later modified programmatically - exist in the template. The template is then stored in the **templates** table in the database.

Creating domains

Domains can either be auto-created through selective harvest definitions or mass upload, or they can be created manually through the interface. Domains are represented with a Domain object, which in turn contains a list of SeedList objects and a list of DomainConfiguration objects. These are stored in the **domains**, **seedlists**, and **configurations** tables respectively. New domains are created with a single configuration, a single seedlist containing **http://www.<domain>** as its only seed, and a limit on number of bytes to download.

Creating schedules

NetarchiveSuite comes with four schedules, repeating respectively hourly, daily, weekly and monthly. More schedules can be created using the web interface. Schedules are represented with the Schedule object and stored in the **schedules** table in the database.

Creating a selective harvest

Harvests are created using the web-based user interface, as described in the User Manual.

Job creation and and dispatching

This functionality is handled by the classes in the harvest scheduler package.

The HarvestJobGenerator selects all harvests that are ready to harvest, i.e. harvests that are activated and where the next harvest date is in the past (or, for snapshot harvest, they haven't run yet). For each harvest, a new thread is started to perform the operations required to send off the jobs – this keeps snapshot harvests from blocking scheduling of selective harvests. Since the threads may run for a while, we keep a set of harvests currently undergoing scheduling and uses it to avoid that the same harvest gets scheduled several times concurrently.

Splitting

Splitting a harvest into jobs and sending those jobs off to the harvesters happens, as mentioned, in a separate thread. The first part of splitting is to reduce the harvests into chunks that can be handled by the HarvestJobGenerator itself – since the data structures for domain configurations and their associated domains contain a fair amount of information, we cannot keep all of them in memory at the same time. For that reason alone, we split harvests into arbitrary chunks with no more domain configurations per chunk than the **configChunkSize** setting allows. We use an iterator to avoid keeping all the domains and their configurations in memory for this operation, which iterates over all configurations in sorted order. The configurations are sorted by order.xml template, then by maximum number of bytes to harvest, and finally by the expected number of objects harvested. This makes sure that configurations that should be in the same job are sorted next to each other. A FilterIterator weeds out configurations in a snapshot harvest whose domains either are marked as aliases or were completely harvested in the snapshot harvest that the current snapshot harvest is based on.

For each chunk, we iteratively create new jobs by taking one domain configuration at a time and checking if it can be added to the job we're building. If it cannot, we store the job and start making a new one. Note that the jobs created are not submitted to the harvesters yet, that happens asynchronously as part of the dispatching check.

The check for whether a configuration can be added to a job is the most complex part of the scheduling system. It is based on the need to partition the domains into chunks such that all domains in a job take approximately the same amount of time to harvest and doesn't exceed Heritrix memory limits. The estimation of the size of a domain is complicated by the facts that previously unharvested domains have an unknown size, and that domains can easily increase in size by several orders of magnitude by adding forums, image galleries or crawler traps. Furthermore, each Heritrix instance can only use one order.xml file.

Whether a domain configuration can be added to a job is a multi-stage check with the following stages:

1. The configuration must not already have been added to the job.
2. The job must not end up with more than configChunkSize configurations.
3. The configuration must use the same crawl template as the other configurations in the job.
4. If the byte limit for this job is determined by the harvest definition, the configuration must not have a smaller byte limit than the definition specifies. If the byte limit for the job is determined by the other configurations in the job, this configuration must have the same byte limit as the other configurations.
5. The expected number of objects harvested by all configurations in the job, based on previous harvests of the configurations, must not exceed the* maxTotalSize* setting.
6. The **relative** difference between the largest and smallest expected numbers of objects harvested by configurations in the job must be no more than the *maxRelativeSizeDifference *setting. Note that the default setting for this is 100, so expectations within a job differ by a factor 100, not just 100%. This prevents jobs from finishing many small configurations quickly and take a long time to finish a few, large configurations.

- However, if the **absolute** difference between the largest and smallest expected numbers of objects harvested by configurations in the job is less than the .minAbsoluteSizeDifference setting, the relative difference is ignored. This allows the very smallest configurations to be lumped together in fewer jobs.

Note: Check on overrides.

The expected number of objects is found based on previous harvests of a given configuration and a few assumptions about the development of web sites. If a configuration hasn't been harvested before, defaults from the settings file are used. Expectations for previously harvested domains are calculated as follows:

1. The "best" previous harvest to estimate from is found by picking the most recent complete harvest using the configuration, or the harvest that harvested the most objects if the configuration never completed.
2. The expected size per object is found based on the average size in the "best" previous harvest, if that harvest got enough objects to be considered (at least 50), but at least as many as the **expectedAverageBytesPerObject** setting.
3. A maximum number of objects is found based on the current limits of the configuration and the harvest and the expected size per object. If neither configuration nor harvest imposes any limits, an artificial limit for estimation purposes is taken from the* maxDomainSize* setting.
4. A minimum number of objects is the number of objects found in the the "best" previous harvest, or is 0 if no previous harvest was found.
5. If the configuration had previously been completed, the estimated number of objects is the difference of minimum and maximum divided by the **errorFactorPrevResult** setting plus the minimum.

- Otherwise, the estimated number of objects is the difference of minimum and maximum divided by the **errorFactorBestGuess** setting plus the minimum.

1. The expected number of objects is capped by the maximum based on the limits.

The **errorFactorBestGuess** setting should generally be smaller than the the **errorFactorPrevResult** setting, since there is more uncertainty about the actual number of objects when the harvest has never been completed. These two settings are best understood as the largest possible factor of error between our estimate and reality. If we use an error-factor of 10, we accept that while configurations could end up growing by as much as the hard limits allow, we split as if they only grow by one-tenth that amount. In most cases, growth will be limited, but it is likely that if a new archive, forum or such is added to a site, the site can grow significantly between harvests. These settings determine the trade-off between the likelihood that some sites have grown a lot and the desire to keep similar-sized configurations in the same job.

Once the job does not get any more domain configurations added to it, it is added to the database with status 'New', and cannot change further except for status updates.

When all domain configurations for a harvest have been placed in jobs, the time for the next execution of the harvest is set. Note that the execution time is updated regardless of whether the jobs are actually successfull, or even have been run. Additionally, the counter of number of runs is updated.

If there are any errors in the scheduling process, including the creation of jobs, the harvest is deactivated to prevent the system from being overloaded with broken scheduling attempts.

Talking to the harvesters

New Jobs are sent to any available harvesters as a DoOneCrawlMessage . This message contains not only the Job object, but also some metadata entries that are associated with the job. Currently, the metadata consists of a listing of the aliases that were used in the job creation and of a listing of the job IDs that should be used to get the deduplication index. The harvests report they are available via. JMS when they are ready to process harvest jobs (see HarvestDispatcher).

The DoOneCrawlMessages are placed on a JMS queue, either `ANY_HIGHPRIORITY_HACO` for selective/event harvests or `ANY_LOWPRIORITY_HACO` for snapshot harvests. At the same time, the job is set to status 'Submitted', indicating that it's in queue for being picked up by a harvester. The names of these queues is a historical artifact and does not indicate that "high priority" jobs can "get ahead" of "low priority" jobs, and there could potentially be just one or more than two queues. Notice that since the JMS messages are expected to be cleaned from the queues at system restart, we assume that any messages about jobs in state "Submitted" are lost after a restart, and they are therefore automatically resubmitted at system startup.

Each HarvestControllerServer application listens to just one of the two queues. When it receives a message (remember that JMS guarantees exactly-once delivery for queues), it immediately sends a message back that tells the HarvestJobManager that the job has been picked up and can be put in state 'Started'.

At this point, the HarvestControllerServer has accepted that it will attempt to run the job and can start to set up the files necessary for running Heritrix.

Harvest setup

The directory used in a crawl is created by the HarvestControllerServer, using the job id and timestamp in the directory name. Details on what Heritrix reads and writes can be found in the Heritrix "outside the GUI" page.

Running Heritrix

The HeritrixLauncher class sets the correct file paths in the Heritrix order.xml file and keeps an eye on the progress of the harvest. If Heritrix does not download any data for a period of time defined by the **noResponseTimeout** setting, HeritrixLauncher will stop the crawl. This is to avoid a single very slow web server from extending the crawl for very little gain. Also, if no crawler threads are active in Heritrix for a period of time defined by the **inactivityTimeout** setting, HeritrixLauncher will stop the crawl. This is a workaround for a subtle bug in Heritrix.

Heritrix is run by the harvester system as a standalone process. This allows access to Heritrix' web interface. The interfacing to the Heritrix process is controlled by JMXHeritrixController, an implementation of the HeritrixController interface. General documentation on JMX can be found as part of the Java documentation, on the Sun JMX Technologies pages, in the JMX Accelerated Howto, and via the JMX Wikipedia page. Heritrix' documentation of its JMX interface is partially described in the JMX feedback page, but can also be investigated in more depth via the Heritrix JMX command-line client, and in the source files Heritrix.java and CrawlJob.java (links for Heritrix version 1.12.1).

JMXHeritrixController starts a new process as part of its constructor, putting the jar files in **lib/heritrix/lib** and the NetarchiveSuite jar files in the classpath. The process is started in the directory created by the HarvestControllerServer, and all files created as part of the crawl are put into that directory. Stdout and stderr from Heritrix, along with a dump of the startup environment, are put in the **heritrix.out** file. The full command line used for running Heritrix is put in the log file.

Before the process is started, a shutdown-hook is added to attempt proper cleanup in case the harvest controller is shut down prematurely. Notice that this hook is removed if the process finishes normally.

After constructing the JMXHeritrixController object, HeritrixLauncher calls the initialize() method on the JMXHeritrixController, which first checks that we're talking to the correct Heritrix instance (in case one was left over from earlier), then uses the addJob JMX command to create a job for the crawl. Before returning from initialize, we call getJobName() to extract from the job a unique name we can use to locate it by later. getJobName() also has the task to wait (using exponential back-off) until the job has actually been created, since the addJob command can return before the job actually exists.

After initialize() is done, the requestCallStart() method executes the JMX command requestCrawlStart to start the job, and we then enter a loop for the duration of the crawl. Inside the loop, we check for the two timeouts as well as for orderly termination of the job and log status reports every 20 seconds. These logs can be seen by the user in the System Overview web page.

Access to the Heritrix user interface can be had by connecting to the port specified by the **heritrixAdminGui** setting, using the admin name and password specified by the **heritrixAdminName** and **heritrixAdminPassword** settings, respectively.

The cleanup of the JMXHeritrixController involves issuing the shutdown JMX command to Heritrix, then waiting for a while (duration defined by the processTimeout setting) for Heritrix to end crawls and write its reports. If Heritrix doesn't stop within the timeout period, we forcibly kill it. After that, we collect the exit code and wait for the stdout/stderr collector processes to finish.

Creating metadata

After the heritrix has finished with the harvesting, the harvest is documented, and the result of this documentation is stored in a separate arcfile prefixed with the job id, and ending with "-metadata-1.arc". This metadata file contains all heritrix logs and reports associated with this harvest(crawl.log, local-errors.log, progress-statistics.log, runtime-errors.log, uri-errors.log, heritrix.out, crawl-report.txt, frontier-report.txt, hosts-report.txt, mimetype-report.txt, processors-report.txt, responsecode-report.txt, seeds-report.txt), some metadata about the job itself, and CDX'es of the contents of the arcfiles created by Heritrix. A CDX line points to where an object is located in an arcfile, its length and mimetype. This metadata arcfile is uploaded along with the rest of the arcfiles.

Uploading

When Heritrix is finished, and the metadata arcfile created, all arcfiles are uploaded to the archive using a ArcrepositoryClient.

Storing harvest statistics

When uploading is done, a status message is sent back to the scheduler, containing error reports and harvest statistics. Errors are split into harvest errors and upload errors, since upload is attempted even if the harvest fails. For each, a short error description and a longer, detailed description are sent. The statistics sent are the following for each domain harvested:

- Number of objects harvested
- Number of bytes harvested
- Reason the harvest stopped, one of completed (no more objects found), object-limit (hit maximum allowed number of objects), size-limit (hit maximum allowed number of bytes, as specified by the harvest), config-size-limit (hi maximum allowed number of bytes, as specified by the configuration), and unfinished (the harvest was interrupted before any of the other stop reasons applied).

...need to specify what gets counted within a domain...

...need to clarify the states of a harvest...

When the status message is received, the statistics from it is stored per domain in the database, along with the job number, the harvest number, the domain name, the configuration name, and a timestamp for receipt of the information.

After the harvest statistics have been sent to the database, the HarvestController application checks, if there is free space on the machine for a new harvestjob. If this is the case, it starts to listen on the job queue. If not, it goes into a dormant mode.

Old jobs

When a harvester application starts up, it checks whether any jobs are left from previous runs, in case the harvest or the upload was aborted. If there is, the last three steps described above are taken for the old jobs before the harvest application starts listening for new jobs.

Deduplication in NetarchiveSuite

deduplication is performed by using the DeDuplicator module developed by Kristinn Sigurdsson as the first of Heritrix write-processors:

```
<newObject name="DeDuplicator" class="is.hi.bok.deduplicator.DeDuplicator">
        <boolean name="enabled">true</boolean>
        <map name="filters"/>
        <string
name="index-location">/home/test/JOLF/cache/DEDUP_CRAWL_LOG/empty-cache</s
tring>
        <string name="matching-method">By URL</string>
        <boolean name="try-equivalent">true</boolean>
        <boolean name="change-content-size">false</boolean>
        <string name="mime-filter">^text/.*</string>
        <string name="filter-mode">Blacklist</string>
        <string name="analysis-mode">Timestamp</string>
        <string name="log-level">SEVERE</string>
        <string name="origin"/>
        <string name="origin-handling">Use index information</string>
        <boolean name="stats-per-host">true</boolean>
      </newObject>
```

This uses a Lucene (v. 2.0.0) index with information about previously harvested objects. This index may be empty.

In NetarchiveSuite, the index contains entries for objects fetched in an earlier harvest job, which this harvestjob is likely to revisit, i.e. this new harvestjob revisits some of same domains that the previously harvest jobs did.

The following arc record part of the metadata for every harvestjob mentions the NetarchiveSuite harvest jobs, that this job can be seen to continue. In the following record, only job with ID 2 is mentioned:

```
metadata://netarkivet.dk/crawl/setup/duplicatereductionjobs?majorversion=1
&minorversion=0&harvestid=3&harvestnum=0&jobid=4 130.226.228.7
20081215100759 text/plain 1
2
```

Each entry in the index contains the URL (unNormalized), content-digest, mimetype , Etag (if available), and the origin of the entry <arcfile>,<offset>.

Only non-text fetched URLs are indexed, as only URIs with non-text mimetypes are looked up by the DeDuplicator.

During the actual crawl-time, all non-text URIs are looked up in the index, and if a match (A URI is matched, if the URI is found, and it has the same digest as the current URI) is found in the index, the URI is ignored by the ARCWriter, but an entry is written to the crawl-log that contains the reference to the original stored URI. This reference is written to the Annotations part of the crawl-line (12th part of the crawl-line): deduplicate:arcfile,offset

Generation of deduplication indices is made by merging information in the crawl-log with information in the CDX-files generated for each job at the conclusion of the harvest.

The CDX'es contain information about the contents of the arc-files generated by Heritrix, but they lack information of the deduplicated objects. CDX-files containing references also to deduplicated objects can be generated from the crawl-logs by the tools provided in the wayback module. These tools are described in the Additional Tools manual.

The merging of this information in NetarchiveSuite was necessitated by the way we do Quality Assurance of the harvestJobs, which is done on a job by job basis, so we needed a way to refer to the deduplicated objects.