WIRE Documentation

Carlos C. Castillo

Center for Web Research

<ccastill@dcc.uchile.cl>

Abstract

This document describes the WIRE application, including instruction on how to run WIRE, and the API for programming extensions to WIRE.

Table of Contents

Chapter 1. Overview

The WIRE project is an effort started by the Center for Web Research for creating an application for information retrieval, designed to be used on the Web.

Currently, it includes:

- A simple format for storing a collection of web documents.
- A web crawler.
- Tools for extracting statistics from the collection.
- Tools for generating reports about the collection.

The main characteristics of the WIRE software are:

- Scalability: designed to work with large volumes of documents, tested with several million documents.
- Performance: written in C/C++ for high performance.
- Configurable: all the parameters for crawling and indexing can be configured via an XML file.
- Analysis: includes several tools for analyzing, extracting statistics, and generating reports on sub-sets of the web, e.g.: the web of a country or a large intranet.
- Open-source: code is freely available.

Chapter 2. Installation

Table of Contents

Downloading

The home page of WIRE is http://www.cwr.cl/projects/WIRE/

The last version can be downloaded from http://www.cwr.cl/projects/WIRE/releases/

If you use WIRE, it is advisable to join the wire-crawler@groups.yahoo.com mailing list to receive announcements of new releases.

Requirements

The following packages are required to install WIRE:

- adns - Asynchronous DNS resolver (note that as of May 2006 the resolver still does not handle DNS chains, a non-standar behavior implemented by some Websites like microsoft.com and others from akamai; a patched version of adns is available).
- xml2 - XML library, including xml and XPath parsers, VERSION 2.6 or newer
- swish-e - The search engine uses swish-e.

The following packages are suggested for best results:

- Also, LaTeX (with fullpage.sty, included in tetex-extras in some distros) and gnuplot are required to generate the reports.

  djbdns - Useful for setting a local DNS cache

  docbook-xsl - Required for generating locally the documentation

  Important

  Before running the application, you must set the environment variable `WIRE_CONF`.

To get optimal results, i.e., to be able to use the crawler with a high degree of parallelization, see kernel tunning.

Installing

To install, unpack the distribution, cd to the created directory, and execute:

```
% ./configure
% make
% make install
```

If you have access to the CVS

If you checkout the distribution from the CVS, you need to do the following in the WIRE directory:

```
%aclocal
%autoheader
%automake -a
%autoconf
```

Before executing **./configure**; this is to be able to use the distribution with different versions of autoconf.

Kernel tunning

The kernel has to be tuned for optimal performance. This can be easily done using the /proc filesystem in Linux (tested in Kernel 2.4).

Log in as root and issue the following commands:

```
% echo 32768 > /proc/sys/fs/file-max
% echo 131072 > /proc/sys/fs/inode-max
```

This sets the maximum number of open files and inodes.

Sometimes you must also set these limits as per-user, in Linux, edit the `/etc/security/limits.conf` file, adding these lines:

```
*       soft    nofile  32000
*       hard    nofile  32000
```

You must make sure that the `/etc/pam.d/login` file includes the limits file:

```
session required        pam_limits.so
```

Note that changing user limits requires the user to logout/login to apply changes.

For compile-time directives regarding limits, see the section on large file support.

Chapter 3. Configuration

Table of Contents

Overview

The configuration file is an XML file. The environment variable `WIRE_CONF` must point to this configuration file. We provide a sample, commented configuration file that includes several of the defaults we use. However, there are some parameters you must set before starting the collection.

Sample configuration

The exact structure varies between versions. See the file sample.conf for an up-to-date example. On this documentation, *`config/x/y`* refers to a variable in the configuration file.

Chapter 4. Data Format and API for the Repository

Table of Contents

Overview

The data format of the collection or repository is designed to scale to several million documents. It is composed of several directories. Each directory is pointed to by a configuration variable.

Text and HTML storage (text/)

The storage subsystem is a large file and some data structures designed to store variable-length records containing the downloaded pages and to detect duplicates.

It is implemented using a free-space list with first-fit allocation, and a hash table for duplicate detections. It never stores an exact duplicate of something you have stored before.

There is an internal limit of `MAX_DOC_LEN` bytes of storage for each document stored, this is set in the source in the file `lib/const.h`

> ### Warning
>
> The system must provide large file support, to be able to create files over 4GB long, if the collection is large, enabling large-file support is explained in the next section. Most modern filesystems support these types of files.

Large file support

All of your source files that needs to be linked with this library must use `config.h`, and those who need `O_LARGEFILE` must include `features.h`.

Otherwise, the compiler will complain because off_t is usually long, but it should be long long with large file support; the error reported by the linker is usually very cryptic (related to something missing in the STL, for instance).

To check if everything was compiled OK, use:

```
% nm --demangle libwire.a | grep storage_read
```

There should be at least one argument of type long long, this indicates that off_t is correct.

API

Opening

To open a storage, use:

```
storage_t *storage = storage_open( char *dirname );
```

`dirname` must be a valid directory, writable by the effective uid. The program will create the required files if necessary. To close, use:

```
storage_close( storage_t *storage );
```

This should be called in the cleanup handler of your application.

Storing documents

To save a document in the storage, you need content and a docid for that document:

```
char *buf = ".....";  /* Document contents */
docid_t docid = xxxx; /* Usually obtained via urlidx */
off_t len = (off_t)strlen( buf );
```

Now you can store the document, checking for the results of the function.

```
doc_hash_t doc_hash;
docid_t duplicate_of;

storage_status_t rc;
rc = storage_write( storage, docid, buf, len, &(doc_hash), &(duplicate_of) );
if( rc == STORAGE_UNCHANGED ) {

/* Document unchanged, already stored with the same content and docid */

} else if( rc == STORAGE_DUPLICATE ) {

/* Document is duplicate, not stored
variable duplicate_of contains the docid of the older
document with the same content */

} else if( rc == STORAGE_OK ) {

/* Document stored */
```

```
}
```

The return values of storage_write are the following:

`STORAGE_UNCHANGED`: The document already exists with the same docid and content, so it has not changed.

`STORAGE_DUPLICATE`: There is another document (its docid will be returned in duplicate_of) with the same content.

`STORAGE_OK`: The document was stored. Maybe it already existed with the same docid, but the content has changed, or it is a new document.

In all cases, the variable doc_hash will contain a hash function of the document content.

Reading documents

To read a stored document, you need first the docid of the document:

```
docid_t docid = xxxx;
```

Now, you must allocate memory and retrieve the document.

```
char *buf = (char *)malloc(sizeof(char)*MAX_DOC_LEN);
off_t len;
storage_status_t rc;

rc = storage_read( storage, docid, buf, &(len) );
if( rc == STORAGE_NOT_FOUND ) {
/* Document inexistent */
} else if( rc == STORAGE_OK ) {
/* Document OK */
}
```

The return values of storage_read are the following:

`STORAGE_NOT_FOUND`: The requested docid is not stored.

`STORAGE_DUPLICATE`: The requested docid was retrieved to buf.The content length is in length

To check if a document exists without loading it to memory, use:

```
docid_t docid = xxxx;
bool rc = storage_exists( storage, docid );
```

Deleting a document

To remove a document from the storage, use:

```
docid_t docid = xxxx;
storage_delete( storage, docid );
```

Notice that the storage subsystem does not save a list of all the duplicates of a document, so if you delete the main document, the duplicates pointer of the others will point to an inexistent document.

Metadata index (metaidx/)

A metadata index is a composed of two files of fixed-size records, one for the metadata about documents, and one for the metadata about sites.

Both data types: doc_t and site_t are defined in `metaidx.h`

The first record in each of these files is special. The second record is for the document with docid=1, the third record for docid=2, etc. This is useful because docid=0 is forbidden, and information about document k is located at offset sizeof(doc_t) * k.

The URL of a document, and the hostname of a site, are not stored in the metadata index, because they are variable-sized records. They are stored in the url index.

API

Opening and closing

A metadata index is opened with the command:

```
metaidx_open( char *directory_name, bool readonly );
```

The directory_name must already exist as a directory, and must be writable by the effective uid. The library checks for the files, and creates them if necessary. If readonly is true, you cannot write data to the metaidx.

To close the metadata index, use

```
metaidx_close( metaidx );
```

Notice that this function should be called from the cleanup handler of your application.

Retrieving information

To retrieve from the metadata file, the programmer must supply the required memory for the object, allocating a doc_t variable and must set the field `doc.docid`.

```
doc_t doc;
doc.docid = 42;
metaidx_doc_retrieve( metaidx, &(doc) );
```

A status code will be returned; usually all errors generated by this library are fatal.

Storing information

To store in the metadata file, the programmer must set all the data.

```
doc_t doc;
doc.docid = 42;
doc.xxxx = yyyy;
metaidx_doc_store( metaidx, &(doc) );
```

And to store a site:

```
site_t site;
site.siteid = 38;
site.xxxx = yyyy;
metaidx_site_store( metaidx, &(site) );
```

Counting

To check how many documents are stored, use

```
metaidx_t metaidx = metaidx_open( dirname );
docid_t ndocs = metaidx->count_doc;
siteid_t nsites = metaidx->count_site;
```

URL index (urlidx/)

This module provides access to an index of site-names (such as: www.yahoo.com), and paths (such as: customers/help/index.html). The module is optimized to provide fast responses for these queries:

Site name to/from siteid: Converts a string, representing a sitename, into a siteid and viceversa.

Site-id + Path to/from docid: Converts a string, representing a path into a site identified by siteid, into a docid and viceversa.

The functions are implemented using on-disk hash tables, and in the case of site names, this hash table is loaded into memory for faster access.

API

Opening and closing

To open a url index, provide the name of a directory to store the hash tables:

```
urlidx_t *urlidx = urlidx_open( char *dirname, bool readonly );
```

The directory must exist, and be writable by the current uid. The urlidx will create the required files.To close the url index, use

```
urlidx_close( urlidx );
```

This function should be in the cleanup handler of your application. Note that many operations are done in memory, so if you do not close the urlidx, some changes might be lost.

Querying for site names

To get the siteid for a site name, use:

```
siteid_t siteid;
urlidx_status_t rc;
rc = urlidx_resolve_site( urlidx, "www.yahoo.com", &(siteid) );
if( rc == URLIDX_EXISTENT ) {

/* The siteid existed and was returned */

} else if( rc == URLIDX_CREATED_SITE ) {

/* A new siteid was allocated and returned */

}
```

If the site already exists, this will return the corresponding siteid, otherwise, a new siteid will be allocated and returned in the variable siteid.

To make the reverse query, i.e.: asking for the site name of a siteid, use:

```
siteid_t siteid = xxxx;
char sitename[MAX_STR_LEN];
urlidx_site_by_siteid( urlidx, siteid, sitename );
```

Notice that the protocol part of the url (http://) is not saved here. The protocol is considered as a metadata in this framework. Don't add the protocol to the sitename.

Querying for paths

To query for a path or file inside a website, you need to get first the siteid of the site in which the file or directory is located.

> **Important**
>
> To save space, path names never contain the leading slash. "/sports/tennis.html" is wrong, but "sports/tennis.html" is right.

Suppose you have the following url "http://www.yahoo.com/news/science.cgi". The first step is to query for the siteid:

```
siteid_t siteid;
urlidx_resolve_site( urlidx, "www.yahoo.com", &(siteid) );
```

Now, we can query for the path:

```
urlidx_status_t rc;
docid_t docid;
rc = urlidx_resolve_path( urlidx, siteid, "news/science.cgi", &(docid) );
if( rc == URLIDX_EXISTENT ) {

/* The document existed */

} else {

/* A new docid was allocated and returned */

}
```

If the path didn't exists, a new docid will be allocated. To make the reverse query (asking for the path of a docid), you don't need to know the siteid:

```
char path[MAX_STR_LEN];
docid_t docid = xxxx;
urlidx_path_by_docid( urlidx, docid, path );
```

Link index (link/)

Links found but not added to the collection (e.g. to multimedia files, etc.) are saved as text files.

Adjacency lists for the graph

Harvest index (harvest/)

One directory per harvest round.

Fixed-size records for metadata

Lists of strings for site names and paths

Chapter 5. Web crawler

Table of Contents

The web crawler is composed of several programs.

Normal operation of the web crawler involves repeatedly running the cycle "seeder-manager-harvester-gatherer-seeder" ... many times.

wire-bot-reset

Clears the repository, creates empty data structures and prepares everything for a new crawling. As some of the data structures require the disk space to be allocated from the beginning, this will take some time depending on your settings for `maxdoc` and `maxsite` in the configuration file.

wire-bot-seeder

Receives URLs from the gatherer (or from the initial URLs) and adds documents for them to the repository. This is used both to give the crawler the initial set of URLs and to parse the URLs that are extracted by the gatherer program from the downloaded pages.

wire-bot-manager

Creates batches of documents for the harvester to download.

This programs sorts the documents from the collection by their "scores" and creates a batch of documents for the harvester. The scores are given by a combination of factors described in the configuration file.

Score function

The score of each page is calculated as defined in the configuration file; this function currently includes pagerank, depth, dynamic/static pages.

The manager tries to determine for each document which is the probability of that document being outdated. If this probability is, say, 0.7, then its current score is defined to be 0.7 x score. Its future score will be 1 x score.

The set of pages that have the higher difference between future score and current score will be selected for the next harvester round.

wire-bot-harvester

This program downloads the documents from the Web. The program works in its own directory with its own data structures, and can be stopped at any time.

> **Important**
>
> If for any reason the harvester fails, you must cancel the current batch using **wire-bot-manager --cancel**, and then re-generate the current batch using **wire-bot-manager**.

wire-bot-gatherer

Parses downloaded documents and extract URLs. This takes the pages downloaded by the harvester from its directory, and merges those pages into the main collection.

wire-bot-run

Run several crawler cycles of the form "seeder-manager-harvester-gatherer"

Step-by-step instructions for running the crawler

The following assumes that you have downloaded and installed the WIRE program files.

1. Create the directory for the collection

Decide on which directory is the collection going to be, in this example, we will assume the */opt/wiredata* directory is used, and then create that directory:

```
% mkdir /opt/wiredata
```

2. Copy the sample configuration file into the directory

The purpose of this is to be able to use different configuration files in different collections.

```
% cp /usr/local/share/WIRE/sample.conf /opt/wiredata/wire.conf
```

3. Set the environment variable `WIRE_CONF`

Under tcsh:

```
% setenv WIRE_CONF /opt/wiredata/wire.conf
```

Under bash/sh:

```
% WIRE_CONF=/opt/wiredata/wire.conf; export WIRE_CONF
```

4. Edit the configuration file, mandatory parameters

Edit the file `/opt/wiredata/wire.conf`. This is an XML file containing the configuration.

Set the base directory of the collection:

`config/collection/base`=*/opt/wiredata*

And set the top level domain:

`config/seeder/accept/domain-suffixes`=Set to the top level domain(s) you want to crawl

Now set the limits for maximum number of documents and sites:

`config/collection/maxdoc`: Set higher than the estimated maximum number of documents in the collection

`config/collection/maxsite`: Set higher than the estimated maximum number of sites in the collection

The later are usually set based on the number of domains, i.e.: for a domain with 50,000 domains, we usually set maxsite to 150,000 and maxdoc to 15,000,000.

The last mandatory parameter is the IP address of the DNS resolvers. See the file `/etc/resolv.conf` in your machine to set this configuration. Note that it is better to have several DNS resolvers, otherwise there will be many DNS lookup errors as a single DNS server will not be able to handle all the requests by WIRE. However, WIRE retries several times for each site name, so eventually even with a single DNS server many site names will be found.

`config/harvester/resolvconf`: Add one "nameserver" line per each name server

The other values are set to rather conservative defaults; see the sample.conf file for more details.

5. Create a blank collection

This command will cleanup an existing collection, and prepare the directories for a new collection.

```
% wire-bot-reset
```

It may take a while, depending on your settings for maxdoc and maxsite.

6. Add starting URLS

You need a file with a set of starting URLs. This file contains absolute URLs, one URL per line; it is customary to leave this file inside the collection directory. Copy it, for example, to `/opt/wiredata/start_urls.txt`.

Example:

```
http://www.site1.zzz/
http://www.site2.zzz/
http://www.site2.zzz/inner/urls/
http://www.site3.zzz/
```

Note that the 'http' must be included in the start url file. If you have IP addresses for some of the Web sites, include them like this:

```
http://www.site1.zzz/ IP=145.123.12.9
http://www.site2.zzz/ IP=68.19.145.4
...
```

To load it into the repository, use:

```
% wire-bot-seeder --start /opt/wiredata/start_urls.txt
```

7. Run TWO test crawler cycles

Run a test crawler cycle to see how is everything going. The first crawler cycle is special as it will only download robots.txt files, and will resolve the IP addresses of the seed URLS. Use the following command line to run two test cycles.

```
% wire-bot-manager
% wire-bot-harvester
% wire-bot-gatherer
% wire-bot-seeder
% wire-bot-manager
% wire-bot-harvester
% wire-bot-gatherer
% wire-bot-seeder
```

If the four programs succeed in the two rounds, then it is most likely than the configuration is correct. Pay special attention to the output of the second **wire-bot-harvester**, to see if it is able to contact the given websites. If the harvester fails, you must run **wire-bot-manager --cancel** to cancel the current cycle before continuing.

8. Run several crawler cycles

Use the included program wire-bot-run to run several cycles of the crawler; consider that in each cycle at most `WIRE_CONF:config/manager/batch/size` documents will be downloaded; so if the batch size is 100,000 documents, 50 cycles will download at most 5 million pages; normally it will be about half of that value because of page errors, saturated servers, etc.

```
% nohup wire-bot-run 50 >& /opt/wiredata/run.log &
```

If you have sysadmin privileges, you can run the crawler at maximum priority using "nice". The following line locates the process-id of wire-bot-run and changes it to the highest priority (you must run this command as root):

```
% nice -19 `ps -o "%p" --no-headers -C wire-bot-run`
```

Stopping the crawler

To stop the crawler, you have to first verify in which stage the crawler is running. To do so, do **ps -efa | grep wire**

If the current process is the **wire-bot-harvester** you can safely stop it using:

```
% killall wire-bot-run
% killall wire-bot-harvester
% wire-bot-manager --cancel
```

The last command is absolutely necessary, otherwise, a partial harvest round will be left.

If the current process is not the harvester, you have to wait until it is, and then stop the crawler. It is not a good idea to interrupt neither the gatherer nor the seeder.

If the machine hangs, or power fails, before running **wire-bot-run** you have to run **wire-bot-manager --cancel**.

Chapter 6. Statistics and reports

Table of Contents

These tools are designed to extract data from the repository to text files and to generate reports.

Interactive querying

wire-info-shell is an interactive program for querying the collection. Type **help** at the prompt to view the list of commands, or **quit** to exit the session.

All the commands write their output to the standard output. This is the list of commands:

Main commands

help: Prints a help message with all the commands.

quit: Exists the program

summary: Shows summary information about the crawler

Information about a document

doc*docid*: Prints all the metadata of the specified document-id. Prints the url of the specified document-id.

read*docid*: Prints the content of a document.

links*docid*: Prints all the links of a document.

Information about a site

site*siteid*: Prints all the metadata of the specified site-id. Prints the hostname of the specified site-id.

site*sitename*: Searches for the site-id of the specified hostname.

Information about a harvester

harvester*harvestid*: Shows data about a harvester batch.

Information about the collection

urlidx: Check the status of url index

metaidx: Check the status of the metadata index. This is useful to get the number of known pages.

linkidx: Check the status of the link index

storage: Check the status of the storage

Extract raw data

wire-info-extract is a tool for extracting readable copies of the contents of the repository, to make analysis with external programs. The data format for the output of **wire-info-extract** is comma-separated-values, i.e.: one record per line, and multiple fields separated by commas. The first record on each file contain the name of the fields.

See:

```
% wire-info-extract -h
```

For a list of available options.

.

A very useful command is:

```
% wire-info-extract --seeds
```

As it can be used to generate the list of starting URLs for the next crawl, including all pages with at least one page downloaded OK, and including their IP addresses. Note that you must run **wire-info-analysis --site-statistics** first, to count the number of downloaded pages per site.

Analyzing data to generate statistics

wire-info-analysis is a program for generating statistics, all the statistics will be generated into the `analysis/` directory.

For help on the different statistics available, use:

```
% wire-info-analysis -h
```

All of the analysis phases are listed in order. This program works in several phases or steps. On each step, a command line switch must be entered. The full sequence must be entered in this order:

```
% wire-info-analysis --pagerank
% wire-info-analysis --hits
% wire-info-analysis --sitelink-analysis
% wire-info-analysis --doc-statistics
% wire-info-analysis --site-statistics
% wire-info-analysis --extension-statistics
% wire-info-analysis --harvest-statistics
% wire-info-analysis --lang-statistics
```

Note that the link analysis phases must me done first, as they generate data that is used by the other phases.

> ### Warning
>
> Some of the statistics may take a long time, specially those related with link analysis. DO NOT USE --link-analysis unless you have enough memory for the task: it is better to run each link graph analysis in a separate process (see below).

Link analysis involves an iterative computation until a target average error is reached. This process can be controlled using `WIRE_CONF:config/manager/score/*/max-error`.

Each analysis phase has a command line switch. The command line switches are explained below:

Link analysis in the graph of pages

--link-analysis: This is equivalent to **--pagerank --hits --wlrank**.DO NOT USE this form unless you have enough memory for the complete task, it is better to run the following tasks in separate process.

--pagerank: Generates the Pagerank of the pages. The calculation requires about 25Mb of memory for each million documents.

`doc.pagerank` is written to the metadata

--**wlrank**: Generates the weighted link rank of pages. This is pagerank weighted by the tag and position of the link in the page. The calculation requires about 25Mb of memory for each million documents. `doc.wlrank` is written to the metadata

--**hits**: Generates the static Hubs and Authorities score of pages. This uses Bharat's heuristic of discarding internal links, using only links between pages at different websites. The calculation requires about 50Mb of memory for each million documents. `doc.hub` and `doc.authrank` are written to the metadata

Link analysis in the graph of websites

--**sitelink-analysis**: This is equivalent to --sitelink-generate --sitelink-component --sitelink-siterank

--**sitelink-generate**: Generates a graph with links, in which all the links to and from pages on the same site are collapsed to a single link. This only generates the graph. Files with links are written to: `sitelink/`

--**sitelink-components**: Site components are generated using Broder's graphs components based on the biggest strongly-connected-component. The sites considered are only those with at least one page downloaded. Component and link-statistics of each site are written to metadata. Statistics are written to `analysis/sitelink/`

--**sitelink-siterank**: An equivalent of pagerank, considering the links between sites, is calculated for each website. `site.siterank` is written to the metadata

Generating statistics

--**doc-statistics**: Generates summary statistics and data tables about all the metadata. It will filter the documents by status (all documents, gathered or downloaded documents, and new documents that have not been visited), and by static/dynamic URLs (all documents, static documents, and dynamic-ally generated pages). This generates all the combinations. For general statistics, it is recommended to use the directory doc_gathered_all, as it includes all the pages that have been downloaded, including static and dynamic pages. Files with data are written to `analysis/doc_X_X`

--**site-statistics**: Generates summary statistics and data tables about websites. Statistics are written to `analysis/site`

--**extension-statistics**: Generates summary statistics about links to domains outside the selected ones, or links to images or multimedia files. Statistics are written to `analysis/extension`

--**harvest-statistics**: Generates statistics about harvest rounds. Statistics are written to `analysis/harvest`

--**lang-statistics**: Generates statistics about languages. Statistics are written to `analysis/lang`

Generating reports

The analysis package includes programs to generate graphs and data tables. This depends on the following packages:

- perl5
- XML::LibXML perl module
- latex (with longtables and fullpage support)
- gnuplot

You must have those packages installed to generate the reports.

Configuring the reports

The exact content of the reports in terms of which data tables and graphs will be generated cannot be configured at this time.

However, it is possible to configure the range for some graphs, as well as the fitting parameters for them. See `WIRE_CONF:config/analysis` in sample.conf. Ranges are given in the configuration in the format

```
[xmin xmax] [ymin ymax]
```

. The default range is

```
[] []
```

which includes all points, but this default might be in insatisfactory for some graphs. The recommendation is to generate the reports, then view the result, then change the range parameters and re-generate the reports. Note that only the report generation is necessary if you change the range parameters, it is not necessary to repeat the analysis.

Procedure for generating reports

The wire-info-analysis program generates multiple .csv (comma-separated values) files in a directory.

The files are analyzed by a wire-report-* program:

1. For each data table, a `xxx_tab.tex` file is generated containing a latex table.
2. For each graph, a `xxx.gnuplot` file is generated, containing a script which is later used by gnuplot to generate `xxx.eps` an encapsulated postscript file. Additionally, a `xxx_fig.tex` file is generated containing the commands for inserting the postscript file into the Latex output.
3. A `report.tex` file is generated. LaTeX is then called to generate a .PDF file which is moved to the root of the analysis directory.

All these files are generated in a subdirectory of `analysis/` under the collection, and they can be modified manually. For instance, the gnuplot scripts can be modified to plot a graph in a different range.

Report about documents

```
% wire-info-analysis --link-analysis
% wire-info-analysis --doc-statistics
% wire-report-doc
```

This report includes statistics about all the documents that were attempted to be downloaded by the crawler. This includes all documents whose status is `STATUS_DOC_GATHERED`. The report contains:

- Page depth. Depth=1 is the front page of each site.
- HTTP code. This is the result code. HTTP codes below 100 are internally used for error conditions in which a successful connection with the HTTP server was not completed.
- Mime type. Normally, images or multimedia files are not downloaded by the crawler. However, web servers can respond with any mime-type, so this is a list of mime-types of successfully downloaded pages.
- Document age in months and years. This is the date of last modification minus the date of last visit. To account for differences in the clock of servers, any timestamp in the future, but less than 1 day, is considered as if it were now.
- Several link score measures, including pagerank, weighted link rank, hub score, authority score, in-degree and out-degree.
- Raw content length. The size of unparsed data is the number of bytes actually transfered. There is a limit on the number of bytes downloaded, see `WIRE_CONF:config/harvester/maxfilesize` for the limit.
- Content length, the number of bytes kept after removing most of the formatting tags.

Additionally, scatter plots will be generated using a sample of the documents. Use `WIRE_CONF:config/analysis/doc/sample-every` to control the size of the sample. Note that if the sample size is changed, it is necessary to re-run the **wire-info-analysis** program.

The parameters for the data tables and graphs in this report can be found under `WIRE_CONF:config/analysis/doc`

Report about sites

```
% wire-info-analysis --sitelink-analysis
% wire-info-analysis --site-statistics
% wire-report-site
```

This report includes statistics about websites.

Note that it is likely that the crawler does not download the entire websites, so some statistics must be considered as bounds on the data. E.g. the content length of the website downloaded by the crawler is a lower bound on the size of the website itself.

The age of the oldest page of a website is a lower bound on the age of the website itself. e.g.: if the oldest page in a website is 6 months old, then the website exists at least since 6 months ago. The age of the newest page of a website is an upper bound on the update frequency of the website. e.g.: if the newest page in a website is 6 months old, then the website was updated at most 6 months ago.

The contents of the report include:

- Top websites by number of pages, size, etc.
- Raw content length. Number of bytes downloaded.
- Sum of several link analysis statistics.
- In-degree and out-degree. Number of different sites pointing to/from this site.

- Siterank. This is like pagerank, but applied to sites.
- Cumulative distribution and histograms of these statistics.

Several scatter plot are also generated relating different statistics. This is done using a sample of one every `WIRE_CONF:config/analysis/site/scatter-plot-every` sites. Changing this number requires to execute the wire-info-analysis program again.

Report about the graph of links between sites

```
% wire-info-analysis --sitelink-analysis
% wire-report-sitelink
```

This report includes information about the graph of links between sites.

The contents include:

- Size of components, related to the main strongly connected components.
- Histogram of strongly-connected component sizes.

Report about the harvest batches

```
% wire-info-analysis --harvest-analysis
% wire-report-harvest
```

This report includes information about the cumulative score acquired during each harvest round. In all the graphs, time is plotted against some score. This is done to analyze how quickly is the crawler finding important pages in terms of e.g.: link score.

The contents of this report include:

- All the relevant data about each harvest round.
- Number of sites included in each round.
- Number of pages successfully downloaded.

Report about document languages

```
% wire-info-analysis --lang-statistics
% wire-report-lang
```

This report contains statistics about the languages in which documents are written. The heuristic used involves several files containing frequent words on different languages. See `WIRE_CONF:config/analysis/lang` for details.

This report also contains the number of pages in each language found at each page depth.

Report about the link extensions

```
% wire-info-analysis --extension-statistics
% wire-report-extension
```

The links to external domain include images (mostly banners, pointing to .net or .com domains). That's why there is a list that includes only pages. Links to external domains are affected by:

1. How big are external domains, i.e.: .COM is very large and hence very linked
2. There are copies of Open Directory Project data everywhere, so you will always find links to very small countries
3. Popular country-code top level domains that are not used by their meaning as countries; here are some examples:

Country Code	Country	Comercial usage
.WS	Samoa	WebSite
.TV	Tuvalu	TeleVision
.TO	Tonga	Used in "go.to/...", "crawl.to/", etc.
.TK	Tokelau	It sounds cool ?
.NU	Niue	Sounds like "new"

This report contains the following information:

- Most linked top level domains.
- Most linked top level domains vs size of domains. This is the ratio between the number of links found and the actual size of a domain. If links were uniformly distributed, this should be 1 for each domain, but they are not, and this list attempts to reflect which domains are more popular.
- Most linked file type extensions, by file type.

The file type extensions can be configured using `WIRE_CONF:config/analysis/extension`

Chapter 7. Indexing and searching

Table of Contents

WIRE includes a program for creating a swish-e index. This program depends on SWISH-e to work.

Creating the index

To create the index, use:

```
% wire-search-indexer --format swish-e --config --index
```

This command first creates a configuration file for SWISH-E (option --config), and then executes SWISH-E (option --index). If you want more control over the indexation parameters, first execute with --config, then edit the configuration file manyall, then execute with --index.

SWISH-E extracts the data according to the configuration file, and will call **wire-search-feeder**, which generates an output that is read by **swish-e** and then indexed. The index is generated in the `index` subdirectory of the collection. Please refer to the swish-e documentation for instructions on how to search using this index.

If you want to create a partial index, starting with document number `FIRST_DOCID` up to document number `LAST_DOCID`, use:

```
% wire-search-indexer --format swish-e --from FIRST_DOCID --to LAST_DOCID
```

Using the index: normal search

To execute a query, first create the index using the procedure described above.

Verify that the index file was created. See the file `index/index.swish-e` under your collection's directory.

To execute a query, use the following (replace "BASE" by the directory of your collection):

```
% swish-e -w query -f BASE/index/index.swish-e
```

Certain properties are saved to the index (e.g.: `swishdescription` contains the first bytes of the text), to see the list of properties, use:

```
% swish-e -T INDEX_METANAMES -f BASE/index/index.swish-e
```

Chapter 8. Version history

Table of Contents

Version 0.22

- Fixed an issue in a 64bit architecture when the metaidx is more than 4GB; and a rare issue with the gatherer when the parser returns an empty document -- thanks Attila Zséder

Version 0.21

- Fixed compilation issues in a 64bits architecture, warnings in universalchardet, and a bug in urlidx.cc with respect to relative paths starting in a dynamic URL -- thanks Rhodrigo Meza.

Version 0.20

- Fixed compilation warnings and added a --texts option to the export.

Version 0.19

- Fixed a bug in wire-bot-harvester that affected a tiny portion of web servers (old versions of Netscape Enterprise Server and others). Now all the lines in the HTTP request end in \r\n instead of \n -- thanks Rhodrigo Meza for reporting and help debugging this.

Version 0.18

- Fixed bug in wire-bot-seeder, the outlinks of the last document were not being processed -- thanks Dmitry Ruzanov, Sergei Shebanin and Ivan Sigaev for reporting this.

Version 0.17

- Fixed bug in wire-info-statistics --lang-analysis (perfhash was too small).

Version 0.16

- Fixed nasty bug in parsing of ROBOTS.TXT files (was saving siteid instead of docid). Thanks to Peter Halacsy and Daniel Varga for reporting the error.
- Several fixes to prevent buffer overflow errors.

Version 0.15

- Implemented Locality-Sensitive Hashing Sketches using the irudiko library. This was done by Angelo Romano; to use it you can set the parameter `WIRE_CONF:gatherer/use-sketches` in the configuration file. This will compute a sketch for each document, which later you can extract using wire-info-extract.

Version 0.14

- Fixed bug in checking site length.

Version 0.13

- Fixed bug that increased CPU usage when there were just a few sites available.
- Documented IP=x.x.x.x extension.
- Included link to patched ADNS version in documentation -- thanks to Hugo Salgado and Rhodrigo Meza.

Version 0.12

- Exports links to text shows only the links among OK pages and/or redirects.

- Redirects are considered in PageRank and HITS computation.
- Fixed error in parsing of robots.txt for cases with multiple user-agent matches.
- Fixed bug in number format of report library (thanks Bartek!).

Version 0.11

- Added an option to reduce disk space by keeping only stats on the links to multimedia files. See the sample configuration file: `config/seeder/extensions/`. If you are upgrading, we recommend you to copy this part of the sample configuration file to your own.
- Fixed warnings during the compilation
- Fixed a nasty bug in the parser, that affected the caption of links under some circumstances
- Fixed a bug that appeared when the domain suffix was .none, the external links among sites were not saved

Version 0.10

- Improved performance, reduced memory usage of the seeder and manager program
- Fixed bug of URL parsing that can cause an unusually high (more than 40%) number of broken links if certain special characters are present in URLs. The error was detected during a crawl of the Polish web, previous crawls should not be affected.

Version 0.9b

- Fixed a bug with the gatherer and collections with more than 50 million documents

Version 0.9

- Added support for MD5
- Added configuration settings for UTF-8 conversion
- Added default configuration settings

Version 0.8

- Introduced the UTF-8 converter

Chapter 9. Known bugs

- "Chunked" content-encoding is not supported properly, meaning that some hex numbers (indicating the size of each chunk) will appear in the middle of some pages.
- Dictionary-based language detection does not work to well, it would be much better to use a bayesian detector. To work around this, you can extract a part of the collection and then run a language detector externally

Chapter 10. Acknowledgements

This project is funded by the Center for Web Research.

The Center for Web Research (CWR) is possible thanks to the Millennium Program.

Design:

- Ricardo Baeza-Yates
- Carlos Castillo

Programming:

- Carlos Castillo - Web crawler and repository
- Emilio Davis - Swish-e integration and weighted link score
- Felipe Lalanne - Charset detection and several bugfixes