WebDB

MichaelCafarella February 7, 2004

The WebDB maintains all of Nutch's information about the web. It keeps information on all known pages and the links that connect them. Anytime a Nutch tool needs information about the entire web (such as during fetchlist generation, or web graph link analysis), that tool needs to use the WebDB.

The WebDB is the only Nutch data structure that is designed to be frequently updated and maintained, rather than rebuilt from scratch with every new index cycle. Clearly, the WebDB must also scale efficiently with the size of the document set. These two facts have made the WebDB more difficult to understand than the typical Nutch component.

This document describes the WebDB API, its implementation, and how the WebDB has been extended for distributed operation.

The IWebDBReader interface is used by any tool that needs only to read from the db. Here are the methods of IWebDBReader, with some commentary.

---

public Page getPage(String url) throws IOException;

The getPage() function searches by URL for a web page object in the WebDB. All web pages are guaranteed to be unique by URL, so this method can return at most one object. This method is useful for a number of tools.

---

public Page[] getPages(MD5Hash md5) throws IOException;

This getPages() function can return an arbitrary number of web Page objects. This method searches by an MD5 hash, computed over the contents of the page's HTML. Although web pages are always unique by URL, they may not be unique by MD5. Many web pages are simply copies; that is, the same content can be found at many different URLs.

---

public boolean pageExists(MD5Hash md5) throws IOException;

This method simply checks whether the WebDB contains a page with the given content checksum. If the WebDB contains 1 or more such pages, it will return true.

---

public Enumeration pages() throws IOException;

The function pages() will return an Enumeration, allowing the caller to efficiently iterate over every Page in the database. The results are sorted alphanumerically by URL.

---

public Enumeration pagesByMD5() throws IOException;

The function pagesByMD5() does exactly the same thing as pages(), except the results are sorted by the MD5 checksum. That means that all pages with identical content will appear consecutively in the Enumeration.

---

public Link[] getLinks(UTF8 url) throws IOException;

Return all the WebDB's hyperlink objects that point to the given URL. Some pages have a very large number of incoming URLs, so it might not be smart for us to return an array of these objects. Probably want to switch this to an Enumeration some time in the future, so all such Links don't need to be in memory at once.

---

public Link[] getLinks(MD5Hash md5) throws IOException;

This getLinks() function will return all the links that originate from the content identified by the given MD5 checksum.

Nutch can place a limit on the maximum number of outlinks a given webpage can have, so it's fine for this method to return an allocated array of Links.

Note that Link objects are considered to originate in some MD5-identified content, and point at a certain URL. That means that it is not quite right to say that a Nutch Link "links two web pages together." The source content might be duplicated under many different URLs. Thus, the WebDB maintains a Link's "source page" only indirectly: a Link has source content identified by MD5, and we can find all the Pages that contain that content.

Link objects in the WebDB are unique for any URL-MD5 combination. Clearly, many Links can point to a single URL. Also, a single piece of MD5-identified content can be the source of many Links.

---

public Enumeration links() throws IOException;

The links() method returns every Link in the WebDB, sorted by target page URL.

---

public long numPages();

Just get how many Page objects are in the WebDB.

---

public long numLinks();

Returns just the number of links in the WebDB.

---

public void close() throws IOException;

Shut down the db.

The WebDB thus needs to provide fast read access to both Pages and Links, each sorted by both URL and by MD5. We assume that the WebDB will always contain more items than we can fit in memory at any one time. So, the best solution is to build a small sorted index of all db items; using this in-memory index, we can compute the disk offset for the item we actually require.

If we expected completely random access to all entries in the db, the best solution would be to build two indices for each data set, and search for an entry with the appropriate index. However, we know that most operations performed with the WebDB are batch-oriented, and that WebDB clients can usually sort read requests to match one of the two index orderings. This is a huge advantage. If the WebDB has written objects to disk in the same order as the key being queried, then it can save a lot of random disk seeks; in fact, the disk head will only move forward, never backward, during a batch of reads.

But there can only be one disk layout of objects for a given key - the data will physically appear sorted either by MD5 or by URL. Which one should we pick? Saving random seek time is so valuable that it is a shame to pick one key over the other. Instead, the WebDB picks both. The WebDB actually keeps two copies of both the Page set and the Link set. It is worth the extra storage space in order to avoid the time-consuming disk seeks.

The WebDB thus contains four sorted and indexed disk files in total:

– Pages, sorted by URL – Links, sorted by URL – Pages, sorted by MD5 – Links, sorted by MD5

We can keep all the indices in memory at once, and use the sorted file that matches the function call's desired key.

It might seem like writing to the WebDB has just become much more burdensome - every write needs to be reflected in two different locations. This does make for some extra work, but the work takes only a constant multiple of the time needed to construct a single indexed file. We will see that most of the WebDB's complexity comes from trying to efficiently construct just one such file.

Nutch tools use the IWebDBWriter interface to add, edit, or delete objects from the WebDB. Its methods are listed below.

---

public void addPage(Page page) throws IOException;

addPage() will add the given Page object to the WebDB, overwriting any Page that might have previously existed with the same URL name. However, if there is an already-present Page, then the link analysis score will be maintained.

This method is useful when updating the WebDB with recently-downloaded Pages. Sometimes the Page might have been previously-seen, and other times the Page will be completely new. If the Page was previously-seen, it is likely that the link analysis score is still somewhat useful.

---

public void addPageWithScore(Page page) throws IOException;

addPageWithScore() works exactly as addPage(), except that link analysis scores are not preserved when there is already a Page in the WebDB with the same URL.

This method is useful for programs actually performing link analysis, which want to replace the Page's previous analysis score.

---

public void addPageIfNotPresent(Page page) throws IOException;

Works just as addPage(), except if there is already a Page with the given URL, the add operation will fail. The method call will have no effect in that case.

---

public void addPageIfNotPresent(Page page, Link link) throws IOException;

This addPageIfNotPresent() function works just as the previous addPageIfNotPresent(). The only difference is that the given Link is also added. If the Page is successfully added, then the Link will be added as well. If the Page is not added, then neither will the Link be added.

---

public void deletePage(String url) throws IOException;

The deletePage() function simply removes the Page in the WebDB (if any) with the given URL. If there is no Page with that URL, the method silently fails.

Note that if this method succeeds, and if the deleted Page is the only one in the WebDB with its MD5 content hash, it means that any Links that originate from that content are no longer valid. In that case, any such Links will be automatically deleted from the WebDB. In fact, deletePage() is the only way to remove a Link.

---

public void addLink(Link link) throws IOException;

The method addLink() will add the given Link to the WebDB. If there is a Link in the WebDB with the same URL-MD5 keys, then it will be replaced by the new Link.

Note that the Link's source MD5 key must be present for a Page object already in the WebDB. Every Link in the WebDB always has source content, identified by MD5. Links can't be added until that source MD5 exists, and when the source is deleted, the outgoing Links are deleted, too.

---

public void close() throws IOException;

Close down the WebDBWriter, and flush all edits. The close() method can actually take a long time to complete, as the WebDBWriter defers most of its work to the last minute.

Note the following:

- We cannot expect edits, via calls to the WebDBWriter interface, to be nicely ordered.
- Some methods need to know the WebDB state to complete successfully. e.g., the WebDBWriter must know whether the given Page has been inserted before.
- Most methods act simultaneously on several of the above-mentioned four tables. Clearly, adding an element will result in writing to the table sorted by URL and to the table sorted by MD5. Others may edit all four tables, for example when adding both a Page and a Link.

Again, we cannot expect to fit the entire set in memory at any one time.

Further, disk seeks are still just as damaging to performance as they would have been during the read stage. Not only do we need to produce a database that can be read with a minimum of seeks, the act of production can't involve many seeks either.

Here is how the WebDBWriter handles all of these problems:

1. All method calls simply log the caller's desire to perform an edit. No edits are actually performed until the caller invokes close().
   2. Upon close(), the WebDB sorts by URL all edits that need to be applied to the first table: Pages-by-URL. (We use the block-sort algorithm, described below*, to efficiently sort a set of elements that are too large to fit in memory.)
   3. We apply the sorted edit-stream to the existing table of Pages-by-URL. Do this by opening both sorted files, and reading the first element of each. Compare the two elements, and do the following:
      - If the smallest item comes from the existing WebDB, then there must be no edit that concerns that item. Append it to a "new Pages-By-URL" file. Read the next element from the stream of existing WebDB elements.
      - If the two items are equal, then the edit is trying to do something to an object that already exists in the WebDB. Depending on the type of edit (e.g., addPage() vs addPageIfNotPresent()) we could:
        - Ignore the edit and simply append the existing WebDB object to the "new Pages-By-URL" file.
        - Modify the existing WebDB object according to the edit, and append the resulting object to the "new Pages-By-URL" file.
        - Throw away the existing WebDB object, and possibly append something to the "new Pages-By-URL" file according to the edit. If the edit came from a call to deletePage(), then we will append nothing at all.

Whatever happens, we read new items from both the stream of edits, and the stream of existing WebDB elements.

If the smallest item comes from the set of sorted edits, we check whether it needs to be matched with an object from the WebDB. For example, if the edit was elicited by a call to deletePage(), then we know there is no appropriate object in the database to be deleted. So the delete operation fails silently. But if the edit came from a call to addPage(), then we can execute it successfully by appending a new Page object to the "new Pages-By-URL" file.

After executing the edit, we read the next edit in the stream.

We repeat the above procedure until both the old WebDB stream and the stream of edits are exhausted. By the end of this process, we will have a brand-new and accurate Pages-By-URL file. Delete the old one, and delete the set of sorted edits.

It should be clear that we have executed a minimum of disk seeks. We only read files from front-to-back, never seeking to arbitrary locations. We only append to the end of the output file.

The only problem with our method as described is that it only works for Pages-By-URL!

Remember we still have three more tables that need to be modified. Those tables will be edited differently than the table above. (Maybe we will be adding Links instead of deleting Pages.) That's fine - while writing down the edits for every table, we just write down four different sets. We blocksort each set, and apply each set individually.

But even worse, a single edit that needs to be applied to two different tables might need to be applied in a certain order. Imagine that we are processing a call to addPageIfNotPresent(). The addPageIfNotPresent call will need to insert to both the "Pages-By-URL" and the "Pages-By-MD5" table. But since Pages are unique by URL, not MD5, we will only know when a Page is present when we are iterating by URL. We handle this case roughly as follows:

1. The addPageIfNotPresent() function writes down an edit for only the "Pages-By-URL" table.
   2. The close() function is called, and we start processing edits. We process edits to the "Pages-By-URL" table as described above. However, processing an edit now consists of more than just choosing whether to append to the "Pages-By-URL" table. We must also choose whether to emit an edit for the "Pages-By-MD5" table.

If we find during the Pages-By-URL stage that the addPageIfNotPresent() call succeeds, we append the new item and we also emit a new addPage() edit for future "Pages-By-MD5" processing. But if the addPageIfNotPresent() call fails, we do not emit an edit for the later "Pages-By-MD5" processing stage.

So creating a new WebDB consists of 4 WebDB-edit merge stages, each slightly different. Further, the sequence of steps is very important. We process the tables in this order:

1. Pages-By-URL 2. Pages-By-MD5 3. Links-By-MD5 4. Links-By-URL

Stage 1 comes before stage 2 mainly because of the addPageIfNotPresent problem. All Page processing needs to come before Link processing because of garbage collection; a deleted Page may result in many Link deletes. Stage 3 comes before stage 4 because Links are always deleted by source MD5, not by the target URL.

Most of the file WebDBWriter? consists of four large loops, each processing a a different table. Each loop contains a large if-then statement that performs exactly the right operation.

*How Blocksorting Works

Any set that is too large to fit in memory can be sorted with this technique.

Say your disk file is k times larger than what you can fit in memory. Read the first 1/k of the file into memory, and use some traditional method to sort the elements. Write out the contents to a new file. Do this for the remaining chunks of your file. You now have k sorted files.

The only remaining task is to merge these k files as efficiently as possible. Open each of the k sorted files and read the first item. Append the smallest of the k items to the new output file. Advance the file pointer for the file of the item you just wrote to disk. Most of the time, the disk head will not be seeking to a random new location. You now have k items again, one of them brand new. Repeat until you have consumed all input files.

If you have a large cache, and possibly a striped disk array, you should be able to avoid waiting for many disk seeks. If k is very large, you can construct a tree of merged subfiles. First merge files 0 .. k/2 into file "A". Then merge files k/2+1 .. k into file "B". Finally, merge files A and B. You should choose the details of your sort according to your quantity of memory, the likely size of your file to be sorted, and the character of your disks.

This sounds fairly efficient ... but is it really faster than "merge sort", which was designed for slow secondary media ?

http://c2.com/cgi/wiki?SortingAlgorithms

http://www.guides.sk/CRCDict/HTML/externalsort.html

http://en.wikipedia.org/wiki/Merge_sort

The WebDB as described here performs very well, with a very good average operation time even for very large databases. But it isn't perfect.

One drawback is that our stream-merge technique for editing the WebDB uses a lot of disk space. The process of applying edits can consume up to twice the space of the finished product.

A much larger drawback is that works only on a single machine. No matter how efficiently we can apply edits, there is a limit to how much a single box can do. In practice, a single box can write about 200M pages in a 30-day period. Since every other Nutch tool is easy to distribute across multiple machines, our elaborate WebDB is still a bottleneck.

However, we've recently created a version of the WebDB that can be read and written to by multiple machines simultaneously. That should allow the WebDB to process arbitrary numbers of elements, and enable the entire Nutch system to maintain a fresh index of size much larger than 200M.

As of early February 2004, the DistributedWebDBWriter and DistributedWebDBReader are not as well-tested as the standard

WebDB. But they do use much of the same code and should eventually replace the single-processor-only versions.

The distributed version of WebDB is moderately more complicated, but uses the same approach as described here.

More on the distributed WebDB coming soon.