Full-text Search Over WebDAV Repository

Vladica M. Ognjanović[1] and Milan Lj. Gocić[2]

***Abstract* – This paper introduces an advanced architecture of full text search over multiple WebDAV repositories. Proposed approach achieve integrated full text search including two important functionalities, namely the possibility to return results which only partially match the query and to rank results accordingly.**

In essence, our work is related with high performance indexing, i.e. processing the original data into highly efficient cross-reference lookup tables in order to facilitate rapid searching.

Experimental results from various tests show that execution time is within the limits of a few seconds making this solution applicable for most common users.

***Keywords* – WebDAV, full-text search, inverted index, metadata search, context search.**

I. INTRODUCTION

The amount of necessary information in a business world is growing at a prodigious rate. The users are becoming more dependent on search engines for locating relevant information.

In most cases, an issued query will result in hundreds of matching documents. In order to avoid flooding the users with a huge amount of results, the search engines present the results in batches of 10 to 20 relevant documents. The user then looks through the first batch of results and if the answer does not exist, he can potentially request to view the next batch. This process is repeated until the user either finds what he is looking for or gives up trying.

Architecture that we propose in this paper allows user to efficiently search over multiple WebDAV (Web-based Distributed Authoring and Versioning) [1] repositories. Currently, existing search support over WebDAV is limited to one repository at the time. It is also dependant on support of a server operating system indexing service (in our case windows indexing service). There are two types of search: search of a documents metadata and a context search. The metadata search includes retrieving some of the document properties like url, creation date, last modified date, length and others. The context search is used for fetching the documents based on their context and user request. This architecture introduces a multi-user system capable of handling the context search of multiple repositories with fast response, reviewing of results, partially matching the query and ranking results.

[1]Vladica M. Ognjanović is with Accordia Group, Kneginje Ljubice 1/1, 18000 Niš, Serbia, E-mail: vladicaognjanovic@gmail.com

[2]Milan Gocić is with Accordia Group, Kneginje Ljubice 1/1, 18000 Niš, Serbia, E-mail: mgocic@yahoo.com

II. ARCHITECTURE

The goal of this architecture is to simplify the search of multiple repositories. Users must register to the system and specify web folders of their interest. The system uses WebDAV as a front-end to collect documents and a file system as a back-end to store essential data of documents. Architecture of the solution is presented in Fig. 1.

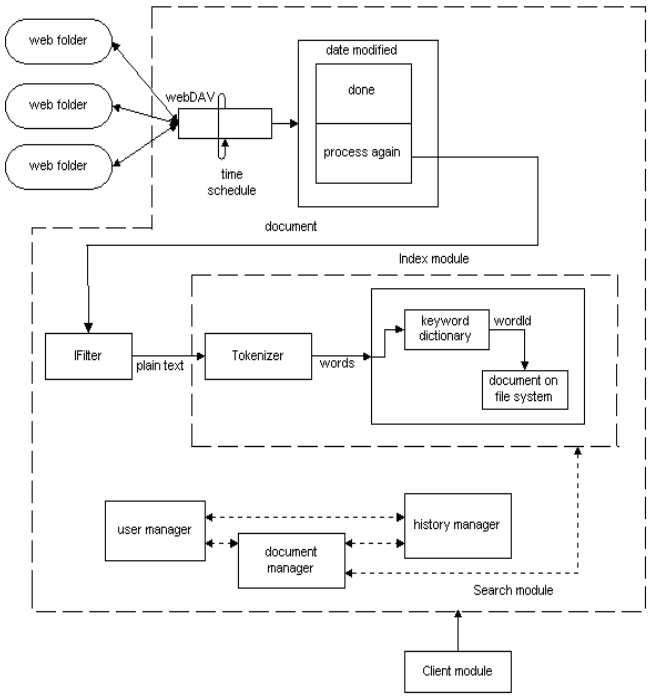

Fig. 1. Search service architecture

The presented search service architecture consists of five main modules: WebDAV module, IFilter module, indexer module, search module and client module.

A. WebDAV module

WebDAV [2-3] is a set of extensions (new headers and new methods) added to HyperText Transfer Protocol 1.1 to support collaborative authoring on the Web. While HTTP is a reading protocol, WebDAV is a writing protocol created by a working group of the Internet Engineering Task Force (IETF), which has defined extensions for six capabilities: overwrite protection, properties, name-space management, version management, advanced collections and access control.

The WebDAV extensions support the use of HTTP for interoperable publishing of a variety of content, providing a

common interface to many types of repositories and making the Web analogous to a large-grain, network-accessible file system. To this time, WebDAV is incorporated into most current operating systems and applications where it performs seamlessly. Since DAV is a single wire protocol like HTTP, it offers a more secure and faster method of file transfer than the one provided by the dual-channel File Transfer Protocol (FTP). Searching using metadata was one of the priorities in the development of DAV. Searching a context of remote web items over WebDAV is supported by operating system on the server in most cases is time consuming.

Collecting properties of web items is a primary task of the WebDAV module. A special accent is put on href, getlastmodified and iscollection properties. Metadata search process is done in three steps:

- Querying server for items metadata is done with a query like:

```
<?xml version="1.0"?>
<a:searchrequest xmlns:a="DAV:">
<a:sql>
        SELECT * FROM SCOPE('DEEP
        TRAVERSAL OF "<web folder
        ulr>'")
</a:sql>
</a:searchrequest>
```

  In this example all properties are requested from server. Explicit request of some properties can be done by modifing SELECT clause.

- Processing of a server reply is based on parsing XML. Server response contains many properties, but we will consider only the listed ones.

```
<?xml version = "1.0"?>
<a:multistatus xmlns:b = "… " xmlns:a = "DAV:">
<a:response>
  <a:href>
        <item url>
  </a:href>
    <a:propstat>
      <a:status>
          HTTP/1.1 200 OK
      </a:status>
      <a:prop>
                  …
        <a:getlastmodified
                  b:dt = "dateTime.rfc1123">
              Sat, 20 Jan 2007 17:14:02 GMT
        </a:getlastmodified>
        <a:iscollection b:dt = "boolean">
              0
        </a:iscollection>
                  …
      </a:prop>
    </a:propstat>
</a:response>
</a:multistatus>
```

  Documents and collections are identified by ***<itemurl>*** and the difference between is made by iscollection property.

- Based on all collected information on some item this module will either send document for a further processing or not. As shown in Fig. 1. this decision is based on the getlastmodfied field.

B. IFilter module

IFilter [4] is a COM component that uses installed IFilter providers to extract the text and to allow the indexer to read different file formats. The providers for the various formats are available from most vendors.

The IFilter interface scans documents for text and properties and extracts chunks of text from these documents, filtering out embedded formatting and retaining information about the position of the text, providing the foundation for building higher-level applications such as document indexers and application-independent viewers.

This interface is implemented to provide a filter for extracting information from a proprietary file format so that the text and properties can be included in the index.

IFilter module searches for a dll and ClassID of COM object responsible for filtering a specific file extension. Then it loads that dll, creates an appropriate COM object and returns a pointer to the IFilter instance which is used to read a plain text from a document.

C. Indexer module

The goal of storing an index is to optimize the speed and performance of finding relevant documents for a search query. Fast query evaluation makes use of an index: a data structure that maps terms to the documents that contain them. With an index, query processing can be restricted to documents that contain at least one of the query terms.

Search engine architectures vary in how indexing is performed and in index storage to meet the various design factors. Types of indices include: suffix trees, trees, inverted indices, citation indices, Ngram indices, term, document matrices.

Architecture presented in this paper is based on inverted indices. An inverted index [5-7] is an optimized structure that is built primarily for retrieval, with update being only a secondary consideration. The basic structure inverts the text so that instead of the view obtained from scanning documents where a document is found and then its terms are seen, an index is built that maps terms to. Instead of listing each document once, an inverted index lists each term in the collection only once and then shows a list of all the documents that contain the given term. Each document identifier is repeated for each term that is found in the document.

Index module architecture is given in Fig. 2. The core of index module is a keyword dictionary. Keyword dictionary is actualy a dictionary where key is a word and the value is a structure containing two integers and one boolean. Integers are wordId and rank, and boolean is reserved for all illegal

words and characters (like stop words, commas, etc.). The rank represents a frequency of word in user queries, and it is primarily used for search suggestion.

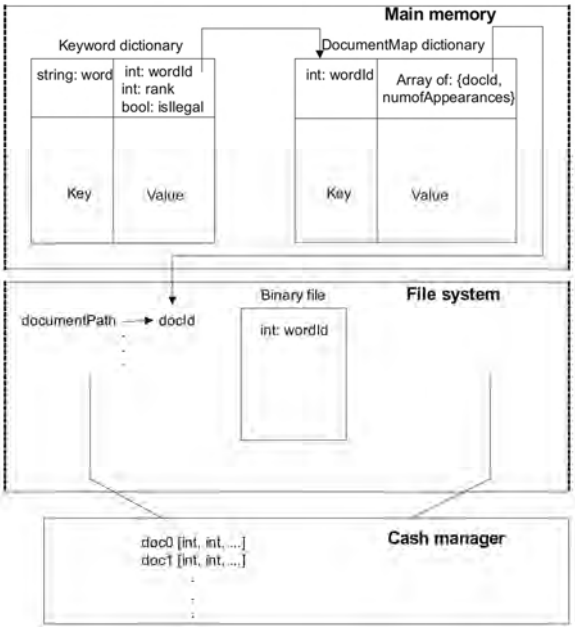

Fig. 2. Index module architecture

In close relation to keyword dictionary is a document map dictionary which is also represented as dictionary, with wordId as the key and array of structure of two integers as the value, the first integer is document id and the second is number of appearances. These two dictionaries reside in main memory. It is important to point out that every keyword and document have unique identifier.

When we get a word from a document, the first we check if a dictionary already contains that word, and if that is true we get the corresponding value in the documentMap dictionary and increase the number of appearances for the current document or add a new entry. Otherwise we create new word id and check if a word or a character is illegal. If illegal we just add it to dictionary, and if not we create a new entry in the documentMap dictionary.

Building index for some document is very simple. If a document is already processed we simply delete all previous information (this includes updating keyword and document map dictionary, and moving the appropriate file from file system to history). After extracting words from documents, we update keyword and document map dictionary as described. Every wordId is stored in the file system and these wordIds array represents documents in this architecture. The maintenance of documents on the file system is the main reason why all the words (legal and illegal) are kept in an one place.

Cash manager is utilized for quick accessing to frequently used documents and to managing the main memory occupied by cash.

Some of major factors in designing a search engine's architecture include:

- merge factors – merging is done by rebuilding index on document,
- storage techniques - file system is used as a back-end,
- index size - considering word encoding index size can be estimated up to 200 times smaller than original documents size,
- lookup speed – lookup time is within the limits of a few seconds,
- maintenance - rebuilding index makes a maintenance simplified, and
- fault tolerance - rebuilding index and periodical autosave of objects in main memory makes this approach reliable.

D. Search module

The primary task of search module is to manage all other modules, as shown in Fig. 1. User, document and history manager are additional parts of the search module. Document manager encapsulates document information, document owners and last index build time. History manager is used for versioning documents. For storage reasons it can be limited to last 10 versions of documents and the compression could be applied. User manager keeps track of user in system, his documents, authentication and user notification (notification of document updates by email).

Search module periodically iterate throw list of registered repositories and triggers WebDAV module. WebDAV module collects documents metadata, filters documents by iscollection and getlastmodified property and sends filtered documents to IFilter. IFilter extracts plain text from documents. The plain text is forwarded to tokenizer that is used for deriving words from text. Index module sends existing document index to history and rebuilds index on a current document.

Document searching involves:

- extracting tokens from user query,
- inquiring keyword dictionary for a list of document ids for each token,
- intersection of lists from keyword dictionary,
- ranking results based on tokens appearance in a document, and number of tokes found in document,
- returning rank results to user,
- results represent documents urls.

User can review a part of the document that contains tokens from search query. Namely, since all words from documents are encoded with corresponding ids, sliding window with the size of 50 words is used to parse document. Every window that contains ids of search tokens is decoded and returned to user.

E. Client module

Client module represents user interface, providing user registration and search service.

III. EXPERIMENTAL RESULTS

The prototype based on the proposed architecture is used for obtaining the experimental results; all tests were performed on four representative documents of different

types. The first and third documents are word document; the second is power point document; and the fourth is a PDF a document.

Index size in term of word count is given in Table 1.

TABLE I
WORD COUNT AND INDEX SIZE

Document number	Number of words per document (approximately)	Index size on disk [kB]
1	3500	14
2	2000	9
3	10000	39
4	34300	135

Original document size and index size ratio are represented in Table 2. Having in mind that each word is represented with four bytes the result can be estimated like in the following example: the expected index size of the third document with approximately 10000 words would be 10000 * 4B = 40 KB

TABLE II
ORIGINAL DOCUMENT SIZE AND INDEX SIZE RATIO

Document number	Original document size[kB]	Original document size / index size
1	3900	278
2	172	19
3	6300	161
4	145	1.07

If all words from document are present in keyword dictionary and if a document is already processed by tokenizer, the build index times would be as shown in Table 3.

TABLE III
DOCUMENT INDEXIG TIME

Document number	Build index time [s]
1	2.3
2	2.2
3	2.7
4	3.6

IV. CONCLUSION

In this paper we have proposed an architecture for full-text search over WebDAV repository.

This architecture improves efficiency of retrieval and enables efficient update of the inverted lists, which is especially important for swift acknowledgement of the document updates.

We plan to implement a complete system based on the proposed architecture that will validate the experimental results.

REFERENCES

[1] M. E. O'Shields, P. J. Lunsford II, "WebDAV: A Web-Writing Protocol and More", Journal of Industrial Technology, Volume 20, Number 2, 2004.

[2] E. J. Whitehead, M. Wiggins, "WebDAV: IETF Standard for Collaborative Authoring on the Web", IEEE Internet Computing, September/October 1998, pages 34- 40.

[3] WebDAV, Retrieved February 1, 2007, www.webdav.org

[4] IFilter, Retrieved April 10, 2007, http://msdn2.microsoft.com/en-us/library/ms691105.aspx

[5] A. Trotman, "Compressing inverted files", Kluwer Int. J. Inform., 2003., 5–19.

[6] E. M. Voorhees, D. K. Harman, "TREC: Experiment and evaluation in information retrieval", MIT Press, Cambridge, MA., 2005.

[7] J. Zobel, A. Moffat, "Inverted files for text search engines", ACM Computing Surveys (CSUR), Volume 38 , Issue 2 Article No. 6, ACM Press New York, NY, USA, 2006.