

# Managing tokenizers in XML search

**Jacek Ambroziak**

Sun Microsystems, Burlington, USA

Jacek.Ambroziak@east.sun.com

<http://www.sun.com/xml>

## **Abstract:**

*We describe the indexing component of our full text search engine for XML. Our main goals are to be able to index documents of different types and to handle their structural and lexical conventions with precision. Indexing of each document type is controlled by an XSLT indexing stylesheet. One of the roles of the stylesheet is to select tokenizers to segment text components of documents into indexable tokens. Both the indexing stylesheets and tokenizers, which are represented as Java objects, can be downloaded by the indexer over the net.*

## **Introduction**

Tokenization forms an integral part of full text indexing. In the process, character representations of source documents are segmented into tokens, which are the atomic units of information typically corresponding to natural language word forms. In addition to word forms tokenization can identify other types of meaning carrying symbols such as numbers, e-mail addresses, URLs, and, depending on the document's domain of discourse: telephone numbers, prices, measures, filenames and so on.

No single set of tokenization rules can handle all languages and/or domains. Even a single XML document may use several lexical conventions in different elements, e.g. in addresses, quoted code, tables, and so on. Instead of providing one very smart tokenizer attempting to cover all needs we designed a flexible architecture of pluggable tokenizers. Represented as Java objects they can be downloaded on demand and under the control of an indexing XSLT stylesheet applied to process text of a document in a context dependent fashion. The stylesheet's templates assign tokenizers to elements based on document structure and/or element attributes.

## **Tokenizers**

Tokenizers are program modules responsible for segmentation of text into broadly conceived of 'words.' Identified during indexing tokens are then often normalized to lowercase and put into an inverted file that forms the core of the text index. Lowercasing loses some potentially useful information but we won't concern ourselves with this aspect of text processing here. The inverted file index of normalized tokens, where token types are associated with documents and document positions of their occurrence, becomes the fundamental data structure used by a query engine to process user queries. The importance of tokenization stems from the fact that the output from this process determines what will be searchable: how the indexer/query engine will 'see' document collections.

Segmentation of text into tokens depends on application of tokenization rules which dictate where token boundaries should be. The simplest form of the process identifies the so called "black and white tokens"—contiguous regions of whitespace characters separating tokens formed exclusively with non-whitespace characters. This approximation of 'real' tokenization works quite well for limited purposes but it fails to acknowledge the special role of punctuation characters, which in addition to whitespace are often token delimiters and thus do not belong to tokens proper. In the next approximation we treat punctuation as

whitespace and continue to apply the black and white mechanism. This variation represents one sweet point of the trade-off between mechanism simplicity and usefulness.

## **The central problem**

As we happen to live in the age of information flood and thus depend more and more on the quality of search engines to find information we seek with minimal effort, we need further sophistication of tokenizers (program modules responsible for tokenization).

For one thing, the simple process described so far is useless for some Asian languages that don't use whitespace for text segmentation. Token boundaries need to be identified within stretches of 'black' characters. Furthermore, when we want tokens, the atomic carriers of information in text, to represent addresses, measures, dates, prices, filenames and so on, we need genuine tokenization rules to reflect conventions employed in forming these units from ordinary and punctuation characters. For instance "4/9/2000" can be a US date. The same date in Central Europe will be represented as "9.4.2000". In both cases punctuation characters (slashes, dots) act as linking characters and not as delimiters. As we can see tokenization is a form of pattern recognition in text and can be used to classify tokens and perhaps normalize their meanings, for instance recognize dates and compute their standard representation. It will then be possible to find passages in text containing dates belonging to given intervals, sort text fragments by date, and so on.

The fundamental problem that we are addressing in this paper is the realization that no single set of tokenization rules can cover the demands of all varieties of indexable text. Individual natural languages and domains of discourse (chemistry, electronics, mathematics, music) will bring their own sets of conventions. Special document sections (keywords, bibliography, etc.) can likewise impose segmentation rules of their own.

The way we address the problem depends on a modular architecture: tokenizers sharing a common (and simple) interface can be plugged in during indexing and applied to text processing in a context dependent way.

The second main point to note is that tokenizer modules can be downloaded over the net together with the documents they are intended to process. In this way the publisher of the information can have complete control over how their documents will be indexed. The tokenizers they supply will be reused for all documents of the same type with regard to lexical conventions. When the conventions change the tokenizers can follow the evolution. Different tokenizer versions can coexist easily since tokenizers are assigned to text sections dynamically on a per section basis. One can also imagine network accessible repositories of standard tokenizers to address common needs (news, product catalogs, and so on). Perhaps tokenizers can accompany standard document schemas in schema repositories.

The indexing system we are describing here is written completely in Java. The language makes it natural, easy and safe to download tokenizer objects (toklets?) over the net and run their code within the indexing environment.

## **XSLT driven indexing**

How are the tokenizers assigned to text sections? The indexing of XML documents in our system is performed under the control of XSLT "indexing stylesheets" . In a typical application stylesheets bind displayable qualities to structures within XML documents. Stylesheet templates assign font properties to

runs of text, introduce purely graphical elements, copy and rearrange text and so on. Documents of differing types are transformed for a common graphical output convention.

By analogy, an indexing stylesheet transforms documents for full text indexing. The transformation involves selecting elements and attributes to be indexed, ignoring others, adding text not present in the original document to act like an annotation and to help find information through querying. One can think about the result of an indexing transformation as a document of an 'indexable' document type. Both text oriented and data oriented documents can be transformed to this type. In reality no output representation is built: the transformation engine directly controls the index building.

Of crucial importance to the present paper is that indexing stylesheets are responsible for binding tokenizers to document fragments. Both the indexing stylesheets and tokenizer objects can be downloaded over the net.

## Application and examples

The described methodology is used to index all types of documents from Shakespeare's plays (for demo purposes) to UNIX manual pages and Solaris documentation. One interesting application involves using the javadoc utility (see Java2 documentation) together with an XmlDoclet to generate XML documentation directly from Java code. javadoc extracts information on program structure (packages, classes, constructors, methods, etc.) from Java sources and associates programmer comments with program elements. All this information is available through the doclet API to the XmlDoclet which outputs valid XML documentation of the source programs. The documentation collections are then indexed and can be searched.

The search engine accepts queries expressed as sequences of query terms. It then finds text passages that contain as many as possible of the query terms (and/or their morphological variants) in close proximity, preferably in the same order and using the same morphological form. The passages are then ranked according to how close do they match the original query expression .

The search engine additionally accepts context descriptions in the form of XPath expressions. These denote sets of nodes—the range of the query. Specific structural elements can be searched e.g. titles, abstracts, titles of major sections). The search range can also be specified by element attributes e.g. [`@level='novice'`]).

Search engines, one per collection/index, can then become network services thanks to Jini technology. Search engines are deployed as Jini services with meta information describing contents/topic of the indexed collection. The search services register with lookup services and place the meta information in lookup attributes. Based on the attributes clients can select and query collections they need. In this **Federated Search Service** individual services can run an arbitrary nodes of the network. Jini takes care of the decentralized administration of the federated service. Services can be plugged in or out independently. The services are available to multiple users querying them at the same time. Each user can query several collections at once: the query results are then merged.

An auxiliary service is responsible for displaying query hits. A query hit is a Java object (transmitted over RMI from remote search services) containing information on the relevant passage location. The information contains the URL of the original XML document, (possibly forked) path (XPath) location of the passage in the document structure, and token numbers of the matching terms to be highlighted. Whenever a user wants to see the matching passage together with its surrounding context she invokes the remote rendering Jini service to get the relevant document fragment. The service gets and parses (if not already cached) the requested document, locates an appropriate transformation, and runs it over the needed document fragment. In the process, the original tokenizer used to index the elements in scope of the query hit is run again to find

the 'words' to highlight. The document fragment with highlighted terms is then shipped back to the user over RMI together with the document's table of contents (if not requested and shipped before). We currently use HTML as the medium of document visualization. It has the benefit that both conventional browsers and existing Java2 Swing components can be used to visualize documents.

Let's take a look at some examples from the indexing stylesheet for the javadoc/XmlDoclet application.

```
<xsl:template match="/|package|classDesc|interfaceDesc">
  <xsl:apply-templates/>
</xsl:template>
```

This template simply selects document branches to be preprocessed.

```
<xsl:template match="class|interface">
  <index:attribute index:nodeID="{generate-id(current())}"
    index:attributeName="inPackage">
    <xsl:apply-templates/>
  </index:attribute>
</xsl:template>
```

The above template matches class and interface elements and adds their 'inPackage' attribute to enable searches using this attribute to narrow down context. Please note that the `index:attribute` element follows `xsl:apply-templates` thus making the attribute range extend over all elements attributes and text recursively indexed by embedded template applications.

```
<xsl:template match="lead|detail|parameter|throws|returns|author">
  <index:element index:tokenizer="com.sun.javasearch.util.SimpleTokenizer">
    <xsl:apply-templates/>
  </index:element>
</xsl:template>
```

Here `index:element` element conveys the information to the index builder that the matching elements need to be indexed, i.e. their descendant text nodes need to be tokenized and the positions of the tokens stored in the index. An attribute specifies which tokenizer to use. The tokenizer class must be available to the current class loader if not loaded already. The indexer creates and caches an instance of the named tokenizer class and applies the tokenizer object to process descendant text nodes. When another tokenizer is specified for a subelement, the current tokenizer is pushed onto a tokenizer stack and reactivated when indexing pops to the previous scope. The index builder notes which tokenizer is used for which text node and stores the tokenizer object with the index to support subsequent term highlighting during querying.

```
<xsl:template match="text(">
  <index:text index:nodeID="{generate-id(current())}">
    <xsl:value-of select="."/>
  </index:text>
</xsl:template>
```

The `index:text` element causes the actual tokenization to happen using the tokenizer currently on the top of the tokenizer stack. Numbers of lowercased tokens not on the stoplist are stored in the index and linked to the document structure. The tokenizer objects need to implement a very simple interface with `public void setText(String text)` and `public Token nextToken()` methods, where `Token` is a class that comes with the system and has a `public Token(String string, int start, int end)` constructor.

## Implementation details

The indexing/query engine system described here builds on our previous experiences of developing a 100% Java search engine for JavaHelp . The original search engine code has been extended to handle XML documents and store their structure in a compressed format. The indexing stylesheets are interpreted by James Clark's XT. The indexes of XML collections occupy approximately 10% of the collection size as both structural and positional information is compressed.

## Acknowledgments

The author wishes to thank members of Sun's XML Technology Center for support and many useful suggestions.

## Bibliography

- [JRA01] Jacek Ambroziak, XSLT in document indexing, XTech 2000, San Jose, CA, March 2000.
- [XSLT] XSL Transformations (XSLT), Version 1.0, W3C Recommendation, 11/16/1999
- [JavaHelp] <http://www.java.sun.com/products/javahelp>
- [JRA02] Jacek Ambroziak and William A. Woods, Natural Language Technology in Precision Content Retrieval, Sun Labs Report, 1998.
- [XPATH] XML Path Language (XPath), Version 1.0, W3C Recommendation, 11/16/1999
- [JINI] <http://www.sun.com/jini>

## Author

### Jacek Ambroziak

Staff Engineer  
Sun Microsystems  
Postal Address:  
XML Technology Center  
UBUR02-201  
01720 Burlington  
MA  
USA  
Telephon: 781-442-0189  
Fax: 781-442-1437  
E-mail: Jacek.Ambroziak@east.sun.com  
Web: www.sun.com/xml

**Jacek Ambroziak** - Jacek Ambroziak joined Sun Microsystems' XML Technology Center in 1999, where he's been applying his extensive background in natural language technologies, Java, Jini, and XML to the problems of XML document search and retrieval. Jacek has been with Sun Microsystems since 1992, when he became a Research Scientist in the Knowledge Technology Group of Sun Laboratories. While in this position, Jacek worked on an approach to text search called "conceptual indexing," which combines techniques from knowledge representation and natural language processing to enable a computer to systematically organize relationships among concepts. Jacek architected and implemented modular semantic lexicons, co-developed a modular text processing pipeline, and built a highly efficient conceptual database and search engine. Subsequent to this work he designed and implemented in Java a full-text search engine that became part of Sun's JavaHelp facility. Jacek received his Ph.D. in Computer Sciences in 1990 from the Polish Academy of Sciences in Warsaw. He worked as a Research Associate in the Academy's Institute of Fundamental Technological Research and, before joining Sun, was a Visiting Scientist in the College of Engineering at Cornell University. Jacek's recent publications include a presentation at the XTech 2000 Conference earlier this year entitled "An XML-Based Approach to the Control of XML Document Indexing," a 1998 SunLabs Report with William Woods on "Natural Language Technology in Precision Content Retrieval," and a presentation on "Conceptually Assisted Web Browsing" given at the WWW6 Conference in 1997.