Search for: Search

Under the Hood

This document goes into moderate depth on the actual operation of the e**X**tensible **T**ext **F**ramework (***XTF***.) This document is *not* an overview of the ***XTF*** system, but rather delves into some nitty-gritty details of how the system performs its tasks. The reader is encouraged to first read the XTF Deployment Guide and have a solid understanding of how data flows through the ***XTF*** servlets, both ***crossQuery*** and ***dynaXML***, as well as through the ***textIndexer*** tool.

Document Processing

The bulk of the ***XTF*** system is concerned with searching XML documents for text in various ways, and displaying the results in several forms. A brute-force search of each word in each document, every time a user made a query, would be extremely inefficient, so the ***textIndexer*** tool is used to compile all of the documents into what is called an "*inverted index*". In essence, this index is similar to that in the back of a book: for each word, it points to all the locations that word appears in all of the documents that have been indexed.

The following sections discuss the details of how the ***textIndexer*** dissects and digests documents, and cover a few basic concepts (such as *term*, *proximity*, and *stop word*) necessary to understand the entire system.

But first, some useful definitions are needed. The ***XTF*** system views each XML document as containing two major types of data: (1) Full text, and (2) Meta-data.

XTF Definition

Full text, n. All text within an XML document, except actual element definitions and their attributes.

XTF Definition:

Meta-data, n. Short data fields within an XML document which describe the entire document, for example: title, author, subject, publication date and publisher, access rights, etc.

This distinction mainly reflects the way the two types of data are used. Typically, meta-data fields are searched "database style". For instance, if one were looking for any book about Mark Twain published after 1912, then one would search the publication date and subject meta-data fields.

By contrast, the full text is typically searched "shotgun style", when one is looking for any use of a word or phrase in any book. For example, one might be interested in every reference to Mark Twain. Of course, the two types of queries can be combined in useful ways, for instance if one were interested in any mention of Mark Twain in books published after 1960.

Text Processing

The ***textIndexer*** extracts the text from each XML document, locating each word and storing its position in the inverted index. These words are called "terms".

XTF Definition:

term, n. A single indexable word. Most punctuation marks are not considered terms, but may occasionally appear within terms. For example, consider the string "O'Reilly". ***XTF*** considers this to be a single term, not two terms as in "O" and "Reilly". See Tokenizing for details of how terms are parsed.

In the following sample XML fragment, all the terms are **underlined**.

```
<head id="chapter2"> <p>"This is the day of man's greatest peril."</p> <footnote>Bekins, 1986</footnote> </head>
```

Because many documents may be book-length, this body of text could be extremely large, containing tens or hundreds of thousands of terms. ***XTF*** imposes no limits on the length of the text or the number of terms within it.

Of course the text exists within the context of XML elements; these elements and their attributes are not considered part of the document text and are not indexed, with one exception. A special attribute (`xtf:sectionType`) can be used to associate a type with block of text; the section type is recorded with all the terms in that block, and text queries can later be restricted to certain blocks based on their section types.

Meta-data Processing

As mentioned earlier, meta-data consists of small fields describing an XML document as a whole. Examples might be a book's publication date and publisher, author(s), subject keywords, etc.

XTF provides a very simple model of meta-data: each document may have any number of meta-data fields, each with a name and a textual value. A given field name may be repeated; each associated text value will be considered one unit of meta-data. Note that structured meta-data (i.e. sub-fields within fields) is not supported.

Each meta-data field is scanned for terms, and each term and its position are recorded in the inverted index. Typically these fields contain dozens or (rarely) hundreds of terms. Longer blocks, while supported, are discouraged as they are inefficient to process.

Tokenizing

As mentioned earlier, all text (whether part of a meta-data field or the full text of a document) is broken into **terms**.

XTF Definition:

tokenize, v., to break a string of text into discrete tokens, or "terms".

XTF, as it is based on the Lucene search toolkit, uses an XTF specific derivative of the Lucene standard tokenizer. This tokenizer makes a fair effort at identifying terms in the source text regardless of language. For the most part, a single term consists of one or more of the following characters:

- Western alphabetic characters such as **A**, **B**, **C**, **d**, **e**, **f**, **Ã–**, **Ã‡**,…
- Arabic numerals such as **1**, **2**, **3**, …
- Non-breaking symbols, such as **&**, **@**, and **(apostrophe)**
- **The underscore character (_)**

By contrast, the XTF system considers all traditional Chinese, Japanese, and Korean characters to be *logograms* **(complete words in and of themselves) and treats each separate character as a term. Some Western symbols are also treated as logograms by the XTF system and are treated as separate terms. These symbols include:**

- Fractions symbols, such as **Â¼**, **Â½**, **Â¾**, …
- Monetary symbols, such as **$**, **Â£**, **Â¥**, …
- Mathematical symbols, such as **+**, **>**, **=**, …
- Trademark or copyright symbology, such as **Â©**, **Â®**, and **.**
- The paragraph symbol **§**

Three other western symbols, the period (**.**), forward slash (**/**), and dash (**-**) are treated differently depending on the context in which they are used. For example, if these characters appear in an acronym like **U.S.A** or in a serial or model number like **v1.2-a** they will be treated as part of the word rather than as separate punctuation.

The following table gives examples of character strings that the tokenizer recognizes as terms. The exact specification is somewhat complex; for details see **XTFTokenizer.jj** from the XTF Distribution.

Category	Examples
Basic word: sequence of digits and letters	`boat`, `Washington`, `6`, `1895`, `Java2`
Words with internal apostrophes	`O'Reilly`, `you're`, `O'Reilly's`
Acronyms *(internal dots are removed)*	`U.S.A`, `I.B.M.`
Company names	`AT&T`, `Excite@Home`
Email addresses	`wer@all-one.com`
Computer host names	`texts.cdlib.org`, `main.server`
Floating-point numbers	`3.14159`, `.6`, `7`, `23`
Dates	`12/3/89`, `3-Jan-02`
Serial and model numbers	`12-6A`, `127.0.1.1`, `270_ES`, `FX/7`

After tokenizing, all upper-case letters within tokens are converted to lower-case, which allows queries to be case-insensitive. Optional processing on tokens can remove distinctions of plural vs. singular, and can remove diacritic accent characters.

Proximity and “Slop”

As mentioned earlier, the inverted index not only maintains a list of the documents each term appears in, but also each term’s position. This information is then used to support proximity-based queries, for example, searching for a pair of terms within 10 words of each other. In ***XTF***, proximity queries are viewed as a “sloppy” match, and thus are specified in terms of a “slop” value.

XTF Definition:

slop, n. The sum, for each term in the query, of the distance between its position in the query and its position (relative to the start of the match) in the source text. Note that slop is similar to “edit distance” in Computer Science, but slop is easier to compute.

For example, consider the query `man`

NEAR
`war` compared to this sentence in a document: “The man went to war.” The potential match will be on the words “man went to war”. In this case, “man” is at position 1 in both, and thus contributes nothing to the slop. However, “war” is at position 2 in the query but position 4 in the match; thus it contributes 2 to the slop, making the total slop 2. Thus for this to be considered a match, the proximity query would have to specify a slop of 2 or greater. This can be summarized in a table:

Term	Position in Query	Position in Text	Difference
`man`	1	1	0
`war`	2	4	2

		total slop	***2***

The definition of slop penalizes terms that are found out-of-order in the document text. So for example, consider a query for `dog` *`NEAR`* `house` compared to the sentence "Looking at his house, our dog despaired." Even though there's only one word between the terms, the slop is actually 3:

Term	Position in Query	Position in Text	Difference
`dog`	1	3	2
`house`	2	1	1
		total slop	***3***

It is easy to see that if the slop were zero, then all the words must appear in the source document in the same order with no intervening terms. This is considered an exact match, generally referred to in this document as a "phrase".

XTF Definition:

phrase, n. A proximity query with slop equal to zero.

When processing the source document, the ***textIndexer*** increments the position for each term it encounters in the source document. However, at sentence boundaries it incremented by five (this default can be changed), which gives the effect of penalizing matches that cross sentence boundaries. Additionally, special index pre-filter tags can increase the position (see the XTF Programming Guide for details.)

Chunking

The ***textIndexer*** tool splits source documents into smaller chunks of text when adding the document to its index. Doing so makes proximity searches and calculating their resulting summary blurbs or "snippets" faster by limiting how much of the source document must be read into memory.

XTF Definition:

snippet, n. A section of source text surrounding and including a match (or "hit").

The **Lucene** search engine forms the foundation of ***XTF***'s search capabilities. From Lucene's point of view, each of these text chunks is a single searchable unit, independent of all other chunks. You might wonder then, "How are proximity searches performed that span the boundary between two chunks?" For instance, consider the following two chunks representing the sentence "The quick brown fox jumped over the lazy dog."

Chunk 1:	`the`	`quick`	`brown`	`fox`	`jumped`			
Chunk 2:						`over`	`the`	`lazy`	`dog`

If one searched for the phrase "fox jumped over", the text engine would return zero results. This is clearly unacceptable.

The answer is that each chunk overlaps the previous one by a certain number of terms, called the "chunk overlap". This overlapping area allows a match in one chunk to extend into the next chunk by the overlapping number of words. Thus the overlap limits and is equal to the maximum proximity the system can handle.

Chunk 1:	the	quick	brown	fox	jumped				
Chunk 2:			brown	fox	jumped	over	the		
Chunk 3:					jumped	over	the	lazy	dog

The chunk size and chunk overlap are both configurable. If the selected chunk overlap is large relative to the chunk size, space and processing time will be wasted because many more chunks will be created. Conversely, making the overlap very small limits the effective maximum "slop" value for all proximity queries. Selecting these values is a trade-off between performance and maximum proximity.

The default values in the XTF distribution define a chunk size of 200 words and an overlap of 20 words. These seem to give an adequate maximum proximity, while minimizing processing time and disk space. One final note: chunking is not performed on meta-data fields, as they are assumed to be relatively small in size.

Stop Words and Bi-grams

Recall that ***XTF*** builds an *inverted index* using Lucene. For each term found in any document, the index stores a list of each occurrence of that term.

Now consider a term like "the". It occurs so commonly in English-language texts that the list of all occurrences becomes very large and thus takes a long time to process. Common words like "the" are called stop words; other examples are "a", "an", "it", "and", "in", "is".

A key observation is that these stop words, because they're so common, are uninteresting to search for. One's initial tendency might be then to simply ignore them, and this solution indeed speeds up searching.

XTF Definition:

stop word, n. A very common word that is generally uninteresting to search for.

For instance, a search for "man in war" would be interpreted as "man war". Unfortunately, this will turn up occurrences of "man of war" (a kind of jellyfish, and not what the user intended.)

So the second key observation is that stop words are useful *in conjunction with non-stop words*. While "in" is a very common word, the combination "man-in" is much less common, and is thus much faster to search for. This leads us directly to the idea of bi-grams, which ***XTF*** implements to get almost the speed of eliminating stop words, but still providing good query results.

XTF Definition:

bi-gram, n. A single term in the index, composed of a stop word and a non-stop word fused together.

Consider the sentence "A friend in need is a friend indeed." Scanning for stop- words and combining them with adjacent words, we get the following sequence of terms (regular terms are marked in **bold**, while bi-grams are underlined):

Index: a-friend

friend

friend-in
in-need
need
need-is
is-a

a-friend
indeed

As you can see, the index is quite different when bi-grams are added into it. Consequently, a similar transformation must be performed when a query is made, essentially rewriting the query. For instance, a search for a *phrase* "**friend** in **need**" is re-written to search for the phrase "friend-in

in-need".

More complex transformations are required for *NEAR* queries. For instance, consider the proximity query: "**friend**
NEAR
in

NEAR
need". The engine rewrites the query with and without stop words, so that if any exact matches are found, they will be ranked higher, but if not, any matches containing "friend" near "need" will be found. The resulting query looks like this:

Query: (**friend**
OR

friend-in) *NEAR* (in-need
OR
need)

In summary, transforming stop words into bi-grams speeds up query processing, while retaining the ability to include stop words in a query.

Query Operations and What They Do

This section gives details on how queries are interpreted, and specifies how the various query operators work. Note that meta-data and text queries are treated somewhat differently. This is due to the fact that meta-data fields are assumed to be short, while the full text of a document is assumed to be very large.

Interpreting User Queries

The job of translating queries from an input URL to a form that ***XTF*** can understand is undertaken by the Query Parser stylesheet. The query parser included in the base distribution is relatively simple: by default, it simply forms an AND query consisting of all the input terms. All non-word terms (such as the "++" in "C++") are ignored. Optionally, the operation can be changed to OR or NEAR. In addition, terms can be excluded if desired.

Here are some sample URLS, and how the default query parser interprets each one:

`http://`*`yourserver`*`:8080/xtf/search?title=apartheid+mind`

Interpreted as: title:(apartheid AND mind)

(Note that "+" is the URL coding for a space.)

```
http://yourserver:8080/search?title=apartheid+mind;title-exclude=mandela
```

Interpreted as: title:((apartheid AND mind) NOT (mandela))

```
http://yourserver:8080/search?title=apartheid+mind;title-join=or
```

Interpreted as: title:(apartheid OR mind)

```
http://yourserver:8080/search?title=apartheid;text=mandela
```

Interpreted as: (title:apartheid) AND (text:mandela)

```
http://yourserver:8080/search?text=%22Nelson+Mandela%22;subject=africa
```

Interpreted as: (text:PHRASE "nelson mandela") AND (subject:africa)

(Note that "%22" is the URL coding for a quote character.)

```
http://yourserver:8080/search?text=Mandela+Apartheid;text-join=5
```

Interpreted as: (text:Mandela NEAR{proximity=5} Apartheid)

Of course much more complex query parsing is possible, since the **Text Engine** can handle arbitrarily complex queries consisting any combination of boolean query operators. Creating such a system is, however, left to the system designer setting up ***XTF***, as it intermeshes closely with whatever HTML form or other mechanism is used to input the query, and is highly dependent on the skill level and needs of the final users of the system.

Text Query Operations

XTF implements a full complement of "boolean" operators used to form complex queries: AND, OR, NEAR, PHRASE, RANGE, and NOT, and supports wildcard search characters. This section covers the details of how these queries are interpreted within the very large documents ***XTF*** can handle.

TERM and Wildcard Queries on Text

A **TERM** query matches every occurrence of the specified term in the document. Upper-case vs. lower-case distinctions are ignored. Additionally, the term may contain special wildcard characters:

- `?` The question-mark character matches terms with any character at that position. For example, `lo?e` would match `love` or `lose`, but not `loe`.
- `*` The asterisk character matches terms with any number (including zero) characters at that position. For example, `dog*` would match any of the following terms: `dog`, `dogs`, `doggie`, `doggerel`, etc.

Depending on the particular wildcard, hundreds or even thousands of terms might match, so care should be taken when using these. To avoid allowing such queries to occupy the engine for long periods of time, ***XTF*** allows queries to specify a limit on the maximum number of terms to match (controlled by the workLimit attribute.) Queries that exceed the limit produce an error.

AND Query on Text

What does it mean to search for "man AND war" in the full text of all documents? Perhaps the most obvious answer would be to search for any document containing both words. But consider a document where "man" appeared in Chapter 1 and "war" appeared in Chapter 7. Would that be a document the user really wanted to find? Probably not. More likely they'd be interested in a document where "man" and "war" appear *close together*.

Thus ***XTF*** interprets **AND** queries on the full text as **NEAR** queries instead, with the slop factor set to the maximum for that index.

More formally, when used with terms, the **AND** query will match any section of text that contains *all* of the terms, in any order, as long as they are close together (that is, within the maximum proximity defined for the index, or 20 words in the default configuration.)

When used to group sub-queries together, **AND** will match text where all of the sub-queries match, in any order, as long as the matches are close together (i.e. within the maximum proximity for the index.)

OR Query on Text

When used to group terms, an **OR** query matches each occurrence of every term contained within it. If used to group sub-queries together, the **OR** query matches each occurrence of every sub-query.

PHRASE Query on Text

A **PHRASE** query generally contains two or more terms, and it matches any span of text where all the terms appear together, in order, with no other terms between them.

Less frequently, it can be used to group sub-queries. It matches a span of text where all of the sub-queries match, in order, without any intervening non-matching terms.

Note that a **PHRASE** query is equivalent to a **NEAR** query with a slop factor of zero.

NEAR Query on Text

Each **NEAR** query requires a "slop" factor. In rough terms, this factor can be thought of as limiting the amount of sloppiness when matching. A slop of zero indicates very tight control; in fact, a **NEAR** query with

zero slop is equivalent to a **PHRASE** query. A large slop value indicates that terms may appear far apart, or out of order, or both. Note however that the slop value is silently constrained to the maximum proximity defined by the chunk overlap of an index.

For more details on how slop is computed, see the section on Proximity and Slop. For information on chunk overlap and how it relates to proximity searching, see the section on Chunking.

The **NEAR** query, when used with terms, matches a span of text containing all of the terms, where the match's slop is less than or equal to the slop factor specified for the query.

When used to group sub-queries, it matches a span of text where all of the sub-queries match, and the complete match's slop is less than or equal to the slop factor specified for the **NEAR** query.

NOT Clause on Text

A **NOT** clause may be specified as a sub-query of any boolean query (**OR**, **AND**, **PHRASE**, or **NEAR**). Any matches in the **NOT** clause will suppress outer matches within the maximum proximity of the index. This can be thought of as a "kill zone": each match within the **NOT** clause kills off nearby matches.

Meta-data Query Operations

The following query operations can be applied to any meta-data field. Queries are applied to meta-data and full text in like fashion, with a few exceptions: **AND** queries are not proximity-based in meta-data fields, **NOT** clauses on meta-data can eliminate whole documents, and a new operator, **RANGE**, is available on meta-data fields.

TERM and Wildcard Queries on Meta-data

A **TERM** query matches every occurrence of the specified term in the meta-data field. Upper-case vs. lower-case distinctions are ignored. Additionally, the term may contain special wildcard characters:

? The question-mark character matches terms with any character at that position. For example, `lo?e` would match `love` *or* `lose`, but *not* `loe`.

* The asterisk character matches terms with any number (including zero) characters at that position. For example, `dog*` would match any of the following terms:`dog`, `dogs`, `doggie`, `doggerel`, etc.

Depending on the particular wildcard, hundreds or even thousands of terms might match, so care should be taken when using these. To avoid allowing such queries to occupy the engine for long periods of time, ***XTF*** allows queries to specify a limit on the maximum number of terms to match (controlled by the workLimit attribute.) Queries that exceed the limit produce an error.

RANGE Queries on Meta-data

A **RANGE** query is similar to a wildcard term query in that it matches a (possibly large) number of terms. Lower and upper bounding terms are specified, and *every* term that appears in the index lexicographically

between the two bounds is matched.

For example, if the lower bound were "1895″ and the upper bound were "1900″, a range query would match any of the terms 1895, 1896, 1897, 1898, 1899, and 1900. Optionally, the query can exclude the bounds, in which case it wouldn't match 1895 nor 1900.

As in the case of wildcard queries, care must be taken to avoid searching a huge number of terms. This can happen easily: in the case of the example above, if dates were encoded in the index in the form `YYYY-MM-DD`, then all the days from 1895 to 1900 would match… potentially 2,190 of them. And of course a range query from A to Z would match practically every term in the index. Again, each query can specify a limit on the maximum number of terms to match, to avoid bogging down the engine.

However, when searching numeric data (for example, file time and date stamps) the above wildcard approach simply is not sufficient. To handle this, XTF provides a special *numeric* range searching capability. This is specified as an attribute to the normal <range> query operator, but it tells XTF that the data is numeric, and in a rigid format (such as `YYYY-MM-DD:HH-MM-SS`; any rigid format is acceptable). When the first such query is made, XTF loads a table of all the data values and converts them to 64-bit integers. This table is then cached in memory, and range queries on that field are processed very quickly, avoiding any wildcard-like expansion.

AND Query on Meta-data

Unlike in full-text queries, an **AND** query on meta-data implies no proximity restrictions. When used with terms, it matches documents where *every* term appears somewhere in the field, in any order.

When used to group sub-queries, it matches documents where *all* of the sub-queries match (note that the sub-queries may be on several different fields.)

OR Query on Meta-data

When used to group terms, an **OR** query matches a document where any of the terms occurs within the meta-data field.

If used to group sub-queries together, the **OR** query matches documents that match by *any* of the sub-queries (note that the sub-queries may involve several different fields.)

PHRASE Query on Meta-data

A **PHRASE** query generally contains two or more terms, and it matches any document where the terms appear together in the field, in order, with no other terms between them.

Less frequently, it can be used to group sub-queries. It matches any document where all of the sub-queries match, in order, without any intervening non-matching terms.

Note that a **PHRASE** query is equivalent to a **NEAR** query with a slop factor of zero.

NEAR Query on Meta-data

Each **NEAR** query requires a "slop" factor. In rough terms, this factor can be thought of as limiting the amount of sloppiness when matching. A slop of zero indicates very tight control; in fact, a **NEAR** query with zero slop is equivalent to a **PHRASE** query. A large slop value indicates that terms (or sub-queries) may appear far apart, or out of order, or both. There is no upper bound on the slop factor. For more details on how

slop is computed, see the section on Proximity and Slop. The **NEAR** query, when used with terms, matches any document where all of the terms appear in the field and their group slop is less than or equal to the slop factor specified for the query.

When used to group sub-queries, it matches any document where all of the sub-queries match, and the complete match's slop is less than or equal to the slop factor specified for the **NEAR** query.

NOT Clause on Meta-data

A **NOT** clause may be specified as a sub-query of any boolean query (**OR**, **AND**, **PHRASE**, or **NEAR**). Any documents matching the **NEAR** clause will be removed from the outer set of matches.

Stop Words in Queries

If a query contains one or more stop words, the query will be internally rewritten to work properly with the bi-gram system. Recall from the section on Stop Words and Bi-grams that using bi-grams allows ***XTF*** to support queries containing stop words while avoiding the usual severe impact on performance that they might have.

Here are some details on how stop words are handled in various query situations:

- In the absence of any grouping operator, querying for a single stop word in a **TERM** query will produce an empty result set.
- Wildcards queries will skip stop words; for example, searching for `th?` would not match "the" (which is a stop word), but would still match "thy".
- Stop words are stripped out of **OR** queries and **NOT** clauses.
- By contrast, **PHRASE** queries retain all stop words, because users would be dismayed if a query on "man of war" returned matches on "man in war".
- In **AND** and **NEAR** queries, stop words are joined with adjacent words to form bi-grams. The resulting query will effectively prefer matches containing the stop words in the correct places, but will allow matches where they don't appear.

Term and Hit Marking

When a meta-data field, snippet, or marked up document is fed into a ***crossQuery*** or ***dynaXML*** formatting stylesheet, ***XTF*** inserts XML tags to indicate matched terms in context, and also indicates the extent of full-text matches, also called "text hits".

XTF Definition:

text hit, n. A consecutive span of words in a document that matches a text query. (Note that there may be many hits per document.)

This section covers details of which terms are marked where, what "snippets" are and how they are formed, and how hits are marked, both within snippets and in their original context.

Snippet Formation and Marking

Hits found in the full text of a document are, of course, surrounded by other text that can help the user decide if the hit is useful to them. ***XTF*** provides this context by calculating a "snippet" for each hit within a document.

XTF Definition:

snippet, n. A section of the source text or a meta-data field, surrounding and including a match (or "hit").

A query may specify the optimal length (in characters) of a snippet, and the system will get as close as it can to that length without exceeding it. The default is 80 characters.

The process used to form snippets is fairly simple:

- First, the **Text Engine** locates the matching text and surrounds it with a `<hit>` tag.
- Next, the engine adds words found in the source document before and after the hit, until it cannot add any more without exceeding the specified snippet size. It attempts to equalize the amount of context added before vs. after the hit.
- Finally, each matching term is marked with a `<term>` tag.

Note that many hits will contain terms that aren't part of the query and thus won't be marked with `<term>`. For instance, a search for "dog NEAR skeleton NEAR bone" on the text "The dog chewed on the skeleton's leg bone." would yield:

```
The <hit><term>dog</term> chewed on the

<term>skeleton's</term> leg <term>bone</term></hit>
```

Snippets are provided to the **Document Formatter** stylesheet in ***crossQuery*** for displaying a summary of hits in all documents, and to the **Document Formatter** stylesheet in ***dynaXML*** to display a ranked list of snippets in a single document.

Hits in their Original Context

In addition to snippets, ***dynaXML*** also inserts `<hit>` tags and `<term>` tags into the original XML document before feeding it to the **Document Formatter** stylesheet. This allows the stylesheet to display the document contents with the hits and terms highlighted in their original context.

The tags are identical to those contained in snippets (except of course that surrounding text need not be inserted, as the hits are marked in the original context.) Additional attributes are added to each hit, giving its score, rank, and hit number.

Note that query terms are marked everywhere they appear in the document text, not just within `<hit>` markers. This gives the stylesheet the option of highlighting them inside and/or outside hits.

Spanning XML Tags

Complication arises when a hit crosses the boundary between two XML elements in the original source text. ***XTF*** takes care not to alter the structure of the document, so hit tags in this circumstance are inserted in a special way. Essentially, the hit is divided into several stretches of unbroken text. The first stretch is marked with an `<xtf:hit>` tag with its continues attribute set to "yes". The subsequent stretches are each marked with an `<xtf:more>` tag; the continues attribute for each is "yes" except for the last, which is "no".

The goal is to allow the **Document Formatter** stylesheet to present a seamless interface to the end-user, who is probably unaware of the underlying structure of a document and is only concerned with where the hits fall.

For example, say we had the following source text containing <i> to mark an italicized section:

```
The hungry plant yearned for <i>human flesh</i> to fill its bottomless gullet.
```

If the user searched for “plant NEAR human” they would expect results something like this:

```
The hungry plant yearned for human flesh to fill its bottomless gullet.
```

To support this sort of behavior, ***XTF*** would mark up the source document like this (broken into multiple lines and indented for clarity):

```
The hungry
<hit hitNum="1" continues="yes">

  <term>plant</term> yearned for
</hit>
<i>
  <more hitNum="1" continues="no">

    <term>human</term>

  </more> flesh
</i>
to fill its bottomless gullet.
```

Let’s consider another example. If the user searched for “plant NEAR bottomless”, the resulting hit would completely span across the <i> tag. Reasonable results from the **Document Formatter** could be:

```
The hungry plant yearned for human flesh to fill its bottomless gullet.
```

Here is how ***XTF*** would mark up this example:

```
The hungry
<hit hitNum="1" continues="yes">

  <term>plant</term> yearned for
</hit>
<i>
  <more hitNum="1" continues="yes">

    human flesh
  </more>

</i>
<more hitNum="1" continues="no">
  to fill its <term>bottomless</term>
</more> gullet.
```

Special Rules for Marking Stop Words

When stop words are part of the query, special rules are applied when marking them in snippets and in their original document context:

- Within a snippet, stop words are only marked with a <term> tag when they appear within a <hit>, and then only as part of an adjoining non-stop-word that actually contributed to the match.
- Likewise, when the original XML document is marked up for the **Document Formatter**, stop words are only marked within a <hit>, and only as part of an adjoining term that contributed to the match.

Hit Scoring

XTF uses and extends Lucene's built-in scoring mechanism to provide a relevance score for each hit, and to return hits in ranked order (i.e. highest score first.) This section describes briefly how the **Text Engine** determines the score for hits in the full text and hits on meta-data fields.

You can observe the scoring engine in action by enabling the `explainScores` attribute on the `<query>` element produced by your **Query Parser** stylesheet. See the Tag Reference for more information on how to enable this.

The following sections break down XTF's scoring calculation like this: first we cover the common aspects shared by both meta-data and text chunk scoring, then talk about the differences, and finally take a look at how the final combined score for a document is computed.

Individual Hit Scoring

Whether a hit (another name for a single match) is in a meta-data field or within the full text of a document, the scoring for that particular hit is the same. How the scores are combined differs, and those differences are covered in later sections.

For those intimately familiar with Lucene, it will be helpful to know that XTF makes extensive use of Lucene's "span" queries, to enable the exact identification of particular matches within a large document. XTF's implementation of spans includes enhancements that calculate the score of each span in addition to its "slop".

Queries on the contents of an XTF index are scored using an enhanced version of Lucene's standard formula. The structure of the scoring formula is fixed, but one can override the calculation of the various factors by providing a Java implementation of the **Similarity** interface.

Plain English

- If a term appears many times in a field or chunk of a document, the match will rank higher; if it only appears a few times, the match will rank lower.
- Rare terms are given more weight than common terms.
- Any field or section of text in the document can be boosted at index time; hits in boosted sections will rank higher.
- For **AND** and **NEAR** queries, more exact matches will rank higher than sloppy matches. That is, a hit where the terms appear in order without intervening words will rank higher than an out-of-order match with many other words interspersed.
- Matches in short fields are ranked higher than those in long fields.
- In an **OR** query, hits that match many of the terms will rank higher than those that only match a few.

Mathematical Details

For a given query **q**, the score for a matching span **s** consisting of terms **t**, in field (or text chunk) **f** of document **d**, is calculated as follows:

> spanScore(q,s) = sloppyFreq(s) * boost(f,d) * lengthNorm(f,d) * coord(q,s) * (sum for t in s: idf(t))

where

- **sloppyFreq(s)** is a factor that decreases as the sloppiness of the matching span increases. The effect is to favor more exact matches. Default implementation: 1 / (slop + 1). In the special case of **OR** queries, or **AND** queries where proximity has been disabled, slop is ignored and 1.0 is used instead.
- **boost(f,d)** is the boost factor (if any) applied to the field (or section of text) containing the match.
- **lengthNorm(f,d)** is a factor that decreases as the amount of text in the field or text chunk increases, since matches in longer fields are generally less precise. Default implementation: square root of the number of terms in the field/chunk.
- **coord(q,s)** is a factor based on the fraction of all query terms that are matched by the span. Spans with a higher ratio of matching terms will be ranked higher. Default implementation: number of terms matched / number of terms in query.
- **idf(t)** is a factor based on the number of documents or text chunks containing **t**. Terms that occur in fewer documents/chunks are better indicators of topic, so **idf** is high when **t** appears seldom in the index, and low when **t** appears often in the index. Default implementation: log(number of docs or chunks in index / number of docs or chunks containing **t**)

Text Hit Scoring

The full text of a document might contain thousands of individual matching spans, each of which will be scored according to the method above. How are these scores combined into a single score for the text?

Plain English

- Documents with more matches will score higher than those with few matches.
- Matches in short texts are ranked higher than those in long texts.

Mathematical Details

For a given query **q**, the score for all matching spans **s** in all text chunks of document **d** is calculated as follows:

textScore(q,d) = lengthNorm(d,text) * tf(sum for s in d: spanScore(q,s))

where

- **lengthNorm(d,text)** is a factor that decreases as the amount of text in the document increases, since matches in longer texts are generally less precise. Default implementation: square root of the number of chunks in the document.
- **tf(…)** is a score factor that helps to equalize the scoring between documents with many matches and those with few matches. Default implementation: square root of the total score of the matches in **d**.

Meta-data Hit Scoring

The scores for multiple hits within a single meta-data field are combined in a similar manner to text hits, above.

Plain English

- Fields with more matches will score higher than those with few matches.
- Matches in short fields are ranked higher than those in long fields.

Mathematical Details

For a given query **q**, the score for all matching spans **s** in field **f** of document **d** is calculated as follows:

metaScore(q,d,f) = lengthNorm(d,f) * tf(sum for s in d.f: spanScore(q,s))

where

- **lengthNorm(d,f)** is a factor that decreases as the length of the field increases, since matches in longer fields are generally less precise. Default implementation: square root of the number of terms in the field.
- **tf(…)** is a score factor that helps to equalize the scoring between fields with many matches and those with few matches. Default implementation: square root of the total score of the matches in **d.f**.

Combined Document Score

The final type of scoring ***XTF*** performs is to combine the scores of all text hits with a document with that document's meta-data scores, to form the final score for that document. Again, the structure of this computation is fixed, but the calculations can be overridden by providing a Java implementation of the **Similarity** interface.

Plain English

- A document's score is based on the sum of all the text hits within it, plus its meta-data field scores.

Mathematical Details

For a given query **q** consisting of meta-data queries **qmf** on fields **f**, and a text query **qt**, the score for a specific document **d** is as follows:

docScore(q,d) = textScore(qt,d) + (sum for f in d: metaScore(qmf,d,f))

where **metaScore** and **textScore** are computed as outlined in the previous two sections.

Spelling Correction (under the hood)

Everybody likes Google's "did you mean" suggestions. Users often misspell words when they're querying an ***XTF*** index, and it would be nice if the system could catch the most obvious errors and automatically suggest an appropriate spelling correction. The XTF team did extensive work to create a fast and accurate facility for doing this, involving minimal work for those deploying XTF.

In the following sections, we'll discuss the guts of XTF's spelling correction system, detailing some strategies that were considered and the final strategy selected, the methods that XTF uses to come up with high-quality suggestions, and how the dictionary is created and stored. If you're looking for information on configuring and using the system, see the Programming Guide.

Choosing an Index-based Strategy

We considered three strategies for spelling correction, each deriving suggestions from a different kind of source data.

1. *Fixed Dictionary*: Available software, such as GNU Aspell, makes spelling corrections based on one or more fixed, pre-constructed dictionaries. But the bibliographic data in our test bed and our sample

queries from a live university library catalog were multilingual and contained a substantial proportion of proper nouns (e.g. names of people or places). This ruled out a fixed dictionary, since many of these proper nouns and foreign words wouldn't be present in any standard dictionary.
2. *Query Logs*: In this method, we would compare the user's query to an extensive set of prior queries, and suggest alternative queries which resulted in more hits. Unfortunately, our test system had a limited number of users, and there was no feasible way to get hold of the millions of representative queries this strategy would require, so it was ruled out as well.
3. *Index-based Dictionary*: Here we would make suggestions using an algorithm that draws on terms and term frequency data from the XTF index itself. It resembles the first approach except that the dictionary is dynamically derived from the actual documents and their text.

Because of the issues identified with the other strategies, we opted to pursue the index-based approach. We feel it is best for most collections in most situations, as it adapts to the documents most germane to the application and users, and doesn't require a long query history to become effective.

We set ourselves a goal of getting the correct suggestion in the #1 position 80% of the time, which seemed a good threshold for the system to be truly useful. With several iterations and many tweaks to the algorithm, we were able to achieve this goal for our sample data set and our sample queries (drawn from real query logs). We have a fair degree of confidence that the algorithm is quite general and should be applicable to many other data sets and types of queries.

Correction Algorithm

XTF builds a spelling correction dictionary at index-time. If a query is sent to ***crossQuery*** and results in only a small number of hits (the threshold is configurable), XTF consults the dictionary to make an alternative spelling suggestion. Here is a brief outline of the algorithm for making a suggestion:

1. For each word in the query, make a list of candidate suggestions.
2. Rank the suggestions and pick the best one (details below).
3. For multi-word queries, consider pair frequencies to improve quality.
4. Suppress near-identical suggestions, and ensure that the final suggestion increases the number of hits.

Each of these will be considered in more detail below.

Which Words to Consider?

This algorithm relies (as most other spelling algorithms do) on a sort of "shot-gun" approach: for each potentially misspelled word in the query, they make a long list of "candidate" words, that is, words that could be suggested as a correction for the original misspelled word. Then the list of candidates is ranked using some sort of scoring formula, and finally the top-ranked candidate is presented to the user.

One might naively attempt to scan *every* word in the dictionary as a candidate. Unfortunately, the cost of scoring each one becomes prohibitive when the dictionary grows beyond about ten thousand words. So a strategy is needed to quickly come up with a list of a few hundred pretty good candidates. XTF uses a novel approach that gives good speed and accuracy.

We began with a base of existing Java spelling correction code that had been contributed to the Lucene project (written by Nicolas Maisonneuve, based on code originally contributed by David Spencer). The base Lucene algorithm first breaks up the word we're looking for into 2, 3, or 4-character **n-grams** (for instance, the word primer might end up as: `~pri prim rime imer mer~`). Next, it performs a Lucene OR query on the dictionary (also built with n-grams), retaining the top 100 hits (where a hit represents a correctly spelled

word that shares some n-grams with the target word). Finally, it ranks the suggestions according to their **edit distance** to the original misspelled word. Those that are closest appear at the top. ("Edit distance" is a standard numerical measure of how far two words are from each other and is defined as the number of insert, replace, and delete operations needed to transform one word into the other.)

Unfortunately, the base method was quite slow, and often didn't find the correct word, especially in the case of short words. However, we made one critical observation: In perhaps 85-90% of the cases we examined, the correct word had an edit distance of 1 or 2 from the misspelled query word; another 5-10% had an edit distance of 3, and the extra edit was usually toward the end of the word. This observation intuitively rings true: the suggested word should be relatively "close" to the query word and those that are "far" away needn't even be considered.

Still, one wouldn't want to consider all possible one- and two-letter edits as it would still take a long time to check if all those possible edits were actually in the dictionary. Instead, XTF checks all words in which four of the first six characters match in order. This effectively checks for an edit distance of two or less at the start of the word.

Take for example the word `GLOBALISM`. Here are the 15 keys that can be created by deleting two of the first six characters:

- `OBAL`

  (deleted characters 1 and 2)

- `LBAL`
  (deleted characters 1 and 3)

- `LOAL`
  (deleted characters 1 and 4)

- `LOBL`
  (deleted characters 1 and 5)
- `LOBA`
  (deleted characters 1 and 6)
- `GBAL`
  (deleted characters 2 and 3)
- `GOAL`
  (deleted characters 2 and 4)
- `GOBL`

  (deleted characters 2 and 5)

- `GOBA`
  (deleted characters 2 and 6)

- `GLAL`
  (deleted characters 3 and 4)

- `GLBL`
  (deleted characters 3 and 5)
- `GLBA`
  (deleted characters 3 and 6)
- `GLOL`

(deleted characters 4 and 5)

- GLOA

(deleted characters 4 and 6)

- GLOB

(deleted characters 5 and 6)

So, XTF checks each of the 15 possible 4-letter keys for a given query word, and makes a merged list of all the words that share those same keys. This combined list is usually only a few hundred words long, and almost always contains within it the golden "correct" word we're looking for.

Ranking the Candidates

Given a list of candidate words, how does one find the "right" suggestion? This is the heart of most spelling algorithms and the area that needs the most "tweaking" to achieve good results. XTF's ranking algorithm is no exception, and makes decisions by assigning a score to each candidate word. The score is a sum of the following factors:

1. **Edit distance.** The base score is calculated from the edit distance between the query word and the candidate word. The usual edit distance algorithm (which considers only insertion, deletion, and replacement) is supplemented by reducing the "cost" of transposition and double-letter errors. Those types of errors are very common spelling mistakes, and reducing their cost was shown by our testing to improve suggestion accuracy. In math terms, the base score is calculated as:

   1.0 – (editDistance / queryWord.length).

1. **Metaphone.** Originally developed by Lawrence Philips, the Double Metaphone algorithm is a simple phonetic mapping that transforms any English word into a 4 character code (called a "metaphone"). Words that have the same code are assumed to be phonetically similar. In XTF's spelling system, if the metaphone of the candidate word matches that of the query word, the score is nudged upward by 0.1. (Note: Philips' algorithm actually produces two codes, primary and secondary, for a given input word; XTF only compares the primary code.)
2. **First/last letter.** If the first and last letters of the candidate word match those of the query word, then the score is nudged upward by 0.1. Again, testing showed that this approach boosted overall accuracy.
3. **Frequency.** Because the indexes do contain misspelled words, we further boost the score if the suggested word is very common. The boost ranges from 0.01 to 0.2, and is arrived at by a slightly complex method, but basically more frequent candidate words get higher boosts. This has the effect of favoring common words over uncommon words (other factors being equal), which helps to avoid ridiculous suggestions.

The original query word itself is always considered as one of the candidates. This is to reduce the problem of suggesting replacements for correctly spelled words. However, the score of the query word is reduced slightly in case a more common word is found that is very similar.

In summary, for a single-word query, the list of all candidates is ranked by score, and the one with the highest score wins and is considered the "best" suggestion for the user.

Multi-word Correction

But what about queries with multiple words? Testing showed that considering each word of a phrase independently and concatenating the result got plenty of important cases wrong. For instance, consider the

query **`untied states`**. Actually, each of these words is spelled correctly, but it's clear the user probably meant to query for **`united states`**. Also, consider the word **`harrypotter`**… the best single-word replacement might be **`hairy`**, but that's not what the user meant. How do we go beyond single-word suggestions?

We need to know more than just the frequency of the words in the index; we need to know how often they occur *together*. So when XTF builds the spelling dictionary, it additionally tracks the frequency of each pair of words as they occur in sequence.
Using the pair frequency data, we can take a more sophisticated approach to multi-word correction. Specifically, XTF tries the following additional strategies to see if they yield better suggestions:

1. For each single word (e.g. `harrypotter`) we consider splitting it up into pairs and see if any of those pairs has a high frequency. For example we'd check <`har rypotter`>, <`harr ypotter`>, <`harry potter`>, etc. and get a solid hit on that last one. That "hitting" takes the form of a higher score boost based on the pair frequency.
2. For each consecutive pair of words, we fetch the top 100 single-word suggestions for both words in the pair. Then we consider all possible combinations of a word from the first list vs. a word from the second list (that's 10,000 combinations) to see if any of the combinations are high-frequency pairs. So in the example of <`untied states`>, one of the suggestions for `untied` will certainly be `united` and we'll discover that <`united states`> as a pair has a high frequency. So we boost the score for `united`, and it ends up winning out against the other candidates.
3. Finally, we consider joining each pair of words together. So if a user accidentally queries for <`uni lateralism`>, we have a good chance of suggesting `unilateralism`.

All of these strategies are attempted and, just as in single-word candidate selection, each phrase candidate is scored and ranked. The highest scoring phrase wins.

Sanity Checks

Despite all the above efforts, sometimes the spelling system makes bad suggestions. A couple of methods are used to minimize these.

First, a filter is applied to avoid suggesting "equivalent" words. In XTF, the indexer performs several mapping functions, such as converting plural words to singular words, and mapping accented characters to their unaccented equivalents. It would be silly for the spelling system to suggest `cat` if the user entered `cats`, even if `cat` would normally be considered a better suggestion because it has higher frequency. The final suggestion is checked, and if it's equivalent (i.e. maps to the same words) to the user's initial query, no suggestion is made.

Second, it's quite possible for the spelling correction system to make a suggestion that will yield fewer results than the user's original query. While this isn't common, it happens often enough that it could be annoying. So after the spelling system comes up with a suggestion, XTF automatically runs the resulting modified query, and if fewer results are obtained, the suggestion is simply suppressed.

Dictionary Creation

Now we turn to the dictionary used by the algorithm above to produce suggestions. How is it created during the XTF indexing process? Let's find out.

Incremental Approach

One of the best features of the XTF indexer is incremental indexing, the ability to quickly add a few documents to an existing index. So we needed an incremental spelling dictionary build process to stay true to the indexer's design. To do this we phase the work, with a low-overhead collection pass added to the main indexing process, and then a phase of intensive dictionary generation, optimized as much as possible.

During the main index run, XTF simply collects information on which words are present, their frequencies, and the frequency of pairs. Data is collected in a fairly small RAM cache and periodically sorted and written to disk. Two filters assure that we avoid accumulating counts for rare words and rare word pairs. Words that occur only once or twice are disregarded (though this limit is configurable); likewise, pairs that occur only once or twice are not written to disk. Collection adds minimal CPU overhead to indexing.

Then comes the dictionary creation phase, which processes the queued word and pair counts to form the final dictionary (this can optionally be delayed until another index run). Here are the processing steps:

1. Read in the list of words and their frequencies, merging any existing list with new words added since the last dictionary build.
2. Sort the word list and sum up frequency counts for identical words.
3. Sample the frequency distribution of the words. This enables the correction algorithm to quickly decide how much boost a given word should receive by ranking its frequency in relation to the rest of the words in the dictionary.
4. Create the "edit map" file. For each word, calculate each of the 15 edit keys, and add the word to the list for that key. Key calculation was discussed above. This edit map file is created and then sorted on disk with lists for duplicate entries being merged together.
5. Finally, any existing pair data from a previous dictionary creation run is read in and then new pairs are added from the most recent main index phase. Unlike the other parts of dictionary creation, this work is all done in RAM, for reasons outlined in the next section covering data structures.

Data Structures

The data structures used to store the dictionary are motivated by the needs of the spelling correction algorithm. In particular, it needs the following information to make good suggestions:

1. The list of words that are "similar" to a given word (i.e. whose beginning is an edit distance of 2 or less.)
2. The frequency of a given candidate word (i.e. how often that word occurs within the documents in an index.)
3. The frequency of a given pair of words (i.e. how often the two words appear together, in sequence, in the index.)

To support these needs, the dictionary consists of three main data structures, each of which is discussed below.

Edit Map

Since at most 15 keys need to be read for a given input word, this data structure is mainly disk-based (it's never read entirely into RAM.) The disk file consists of one line per 4-letter key, listing all the words that share that key. At the end of the file is an index giving the length (in bytes) for each key, so that the correction engine can quickly and randomly access the entries.

The words in each list are *prefix-compressed* to conserve disk space. This is a way of compressing a list of words when many of them share prefixes. For example, say we have three words in the list for a key:

apple application aplomb

We always store the first word in its entirety; each word after that is stored as the number of characters it has in common with the previous word, plus the characters it doesn't share. For long lists of similar words, the compression becomes quite significant. In our example the compressed list is:

```
apple 4ication 2lomb
```

Here are some lines from a real edit map file, with the keys in **bold**:

```
abrd|aboard|2road|2surd|6ist|7ty|6ly


abre|abbreviated|9ions|0fabre|0laborer|7s

abrg|aboriginal

abrh|abraham|7son


abri|aboriginal|4tion|8s|0cambridge|0fabric|6ated|0labyrinth
```

… and the random-access index at the end of the file …

```
abrd|37


abre|42

abrg|16

abrh|18


abri|61
```

Word Frequency Table On disk this is stored as a simple text file with one line per word giving its frequency. The lines are sorted in ascending word order. This structure is read completely into RAM by the correction engine, as we need to potentially evaluate tens of thousands of candidate words per second in a high-volume application.

Here are some lines from a real word frequency file:

```
aboard|10
```

```
abolished|19

abolishing|9

abolish|8

abolition|34

abomination|6
```

Pair Frequency Table

The correction engine needs to check the frequency of hundreds of thousands of word pairs per second. This implies a need for extremely fast access, so we need to pull the entire data structure into RAM and search it very quickly.

The table exists only in binary (rather than text) form, in a very simple structure. A "hash code" is computed for each pair of words. The hash code is a 64-bit integer that does a good job of characterizing the pair; for two different pairs, the chance of getting the same hash code is vanishingly small. The structure consists of a large array of the all the pairs' hash codes, sorted numerically, plus a frequency count per pair. This sorted data is amenable to a fast in-memory binary search.

Here's the disk layout:

# bytes	Description
8	Magic number (identifies this as a pair frequency file)
4	Total number of pairs in the file
8	Hash code of pair 1
4	… and frequency of pair 1
8	Hash code of pair 2
4	… and frequency of pair 2
8	Hash code of pair 3
…	etc.

As you can see, we store each pair with exactly 12 bytes: 8 bytes for the 64-bit hash code, and 4 bytes for the 32-bit count. Working with fixed-size chunks makes the code simple, and also keeps the pair data file (and corresponding RAM footprint) relatively small.

Lazy Trees

To accelerate processing of large documents and to enable highlighting search results within a document, ***XTF*** creates and utilizes *lazy trees*. This section describes what lazy trees are, how they speed up processing, and how stylesheets should be optimize to take advantage of lazy trees.

The Problem of Large Documents

The user usually browses a document page by page . for instance, reading a section, then jumping to an appendix, then another section, etc. Each time the servlet produces one of these .pages., it must read and process the entire source document, even though only a small portion contributes to the output. This is the key observation underlying lazy trees.

When a document is processed through a set of stylesheets, the XSLT processor spends significant time reading and parsing the XML document, building indexes of parent-child relationships, creating identifier cross-references, and so on. Most of this work is wasted in generating the page for a single section, so what if we could eliminate the data for all the other sections?

Let.s take an example. Consider the source document below (simplified for discussion). If the user wants to see Chapter 1, only the indicated elements actually needed.

```
<front>

  <titlePage>The opening of the Apartheid Mind</titlePage>
  blah. blah. blah.

</front>
  <div1 id="ack">
  <head>Acknowledgements</head>

  <p>I would like to thank.</p>
  <!-- blah, blah, blah -->

  <!-- blah, blah, blah -->
</div1>

<?NOTE: The "ch1" div1 element is needed for this request... ?>

<div1 id="ch1">
  <head>Chapter 1</head>
  <p>It is now conventional wisdom that.</p>

  <p>See Hanlon (1981) for an overview. <ref target="bib12"/></p>

  <!-- blah, blah, blah -->
  <!-- blah, blah, blah -->
</div1>

<div1 id="ch2">

  <head>Chapter 2</head>

  <p>One of the more striking aspects of contemporary.</p>
  <!-- blah, blah, blah -->
  <!-- blah, blah, blah -->
</div1>

<div1 id="bib">
  <bibl id="bib1">  <author>Adams, Heribert</author> </bibl>

  <!-- blah, blah, blah -->
  <!-- blah, blah, blah -->

  <?NOTE: This "bib12" element is also needed... ?>
  <bibl id="bib12"> <author>Hanlon, Joseph</author> </bibl>

  <!-- blah, blah, blah -->
```

```
  <!-- blah, blah, blah -->
</div1>
```

Now that might not look like much of a savings, until you remember that each “blah, blah, blah” stands for a great deal of data.

The `<div1 id="ch1">` element contains the actual data for Chapter 1. But the `<bibl id="bib12">` element must also be included. Why? Notice that Chapter 1 contains a bibliographic reference; in order to properly generate a hyperlink to it, that reference element must be included.

In general, the only parts of the document needed are those the stylesheet actually references while generating a given page. This leads to the central idea of lazy trees: only load those parts of the document that are needed.

What is a “Lazy Tree”?

Unfortunately it’s not easy to randomly load small pieces of a large XML document. In general, one can’t tell where the end tag for an element is without scanning all the text until it’s found. So ***XTF*** creates a binary version of each XML document, called a “lazy tree”. The tree is stored in a file containing all the original contents of the document, plus an index telling XTF where each element starts and ends.

Processing begins by loading only the root element of the document. If the stylesheet references the children of that element (in an XPath expression for instance), then XTF knows right where to find them in the lazy tree’s file. It loads only those children, but not their descendants. Processing then proceeds again until the stylesheet needs another node that hasn’t been loaded yet. In this way, a typical page generation ends up loading only a small portion of the document. This strategy makes it possible to quickly generate pages for any size document (tested up to at least 60 megabytes).

Lazy tree files are stored under the index directory, in a folder hierarchy parallel to that of the original data directories. They are typically a bit smaller than the source document, since the text within them is compressed.

Search Results in Context

The other main benefit of lazy trees is that they enable ***XTF*** to show search results in context. At first the link might seem unobvious. Remember that XTF reports, for each element, the total number of hits found in that element *and all of its descendants*. Without a lazy tree, the system would have to read in the entire XML document to determine the parent-child relationships. With a lazy tree, each search hit can be directly attributed to the correct XML elements by simply looking them up in the cross-index stored in the lazy tree’s file.

For this reason and for speed of processing, ***dynaXML*** always uses the lazy tree if one has been created. If it doesn’t exist yet, dynaXML will attempt to create it at run-time. This introduces a pause, which can be avoided by having the ***textIndexer*** can create lazy trees at index time (and in fact, this is the default behavior.) One caution: if pre-building of lazy trees by the textIndexer is disabled, be sure that `docSelector.xsl` and `docReqParser.xsl` both specify the same pre-filter to use. Otherwise, dynaXML will build a lazy tree that doesn’t match the index, and strange random errors will occur when highlighting search hits in the document.

Stylesheet Considerations

Though the process of loading elements and other pieces of an XML file is generally invisible to the stylesheet programmer, there are certain best practices to obtain maximum processing speed.

First, avoid using XSL constructs that scan the entire input document. Generally, any XPath instruction beginning with "`//`" will scan every element of the document, defeating any gains of using a lazy tree. Also, using "`descendant::`" and "`descendant-or-self::`" in an XPath expression will generally cause all of the descendants to be loaded, again counteracting the benefit of lazy trees.

Instead, try to replace these constructs with the use of XSL keys, declared with the `<xsl:key>` element at the top-level of the stylesheet. You might ask, "Doesn't `xsl:key` need to scan every element of the document to build the key?" The answer is yes, but the result is stored in the lazy document, so that subsequent page views using the same key don't need to scan the document again. In other words, only the first page view causes XSL keys to be built.

Even the overhead of building XSL keys at runtime can be avoided by having the ***textIndexer*** do that work. This is accomplished simply by specifying the *displayStyle* attribute in the `docSelector.xsl` stylesheet used by the indexer. If specified, the stylesheet referenced by *displayStyle* will be scanned for `<xsl:key>` declarations, and all of the keys will be pre-built at index time. See the Document Selector Programming section in the **XTF Programming Guide**.
One final note: a good way to locate parts of the stylesheet that need to be optimized is to use ***dynaXML***'s stylesheet profiling configuration option. When enabled, a summary of how many XML document nodes were accessed by each line of the stylesheet will be printed. Find a particular request that runs unacceptably long, examine the
profile, and take aim at the lines which access the most nodes.

"More Like This" (Similar Documents Query)

XTF contains a query operator that allows one to query for documents that are "similar to" a given target document. This sort of feature is often activated in user interfaces using a "More like this…" link.

A sample implementation has been provided in the default stylesheets that come with XTF. Simply perform a query in crossQuery, and click on the **Similar Items: Find** link. A similarity query will be executed asynchronously, and a summary of the resulting documents inserted directly into your search page.

Similarity Algorithm in Brief

The current algorithm analyzes the content of the bibliographic metadata for the target item, chooses the most important terms in the record, and formulates a new query. Top-ranking items resulting from the new query are presented as recommendations.

While simple in theory, the number of permutations and complications to this approach are vast. There are many methods for choosing and ordering the top terms, and many approaches to formulating the new query. Moreover, bibliographic records are inconsistent. Some records are catalogued exhaustively, others are sparse. Particularly in sparse records, the choice of a single subject heading can significantly affect the choices and weights of terms. In extreme cases, this can cause unexpected results: two versions of a book, sparsely catalogued and with slight differences in subject headings, can yield very different recommendations. We experimented with various approaches to try to balance these complexities.

In our final iteration, each term of each metadata field in the source document is considered in turn. The number of occurrences of that term in the field, *tf*, is computed. Also, the total number of documents containing that term in that field, *df*, is fetched from the Lucene index. Terms are filtered out if they occur in

too few or too many documents (the limits are adjustable.) Next, a score is calculated for the term by multiplying *tf* * *idf*, where *idf* is the standard log(numDocs / *df*) + 1. Finally, the score for each term is totaled across all fields it occurs in. The resulting term list is ranked by score and the top-scoring 25 terms are chosen (also adjustable.)

The chosen terms are turned into what we call an "Or-Near" query. Each term is searched in each field and document, increasing the score of documents it is found in. Documents with more terms appearing in a single field receive an extra boost. In this way, a score is calculated for each matching document, and the top 5 scoring documents are output as the query results.

Activating Similarity Query

To activate the similar documents query, replace your main query output from the **Query Parser** with a `<moreLike>` tag. The XTF Tag Reference has more information on this tag.

Edit this entry.

Latest XTF News

- XTF 3.0 beta
- XTF Website Launched
- XTF Community Preview
- XTF 2.2 released

Subscribe to XTF News

- RSS

The **eXtensible Text Framework (XTF)** is supported by the California Digital Library