

Searching & Indexing XML documents

Jamie Høglund

<http://www.geniegate.com>

Formerly XML Global Technologies, Inc.

ABSTRACT

Following is a retrospective case study in the rapid development of a contextually aware search tool. While not entirely successful, it did provide many lessons as well as an effective demonstration of the potential usefulness of XML.

1. Building the "Information Super Haystack".

It was the late 90's in the Canadian city of Vancouver BC. The dot-com era was in full swing, gopher had all but evaporated while a virtually unheard of technology was quietly approaching the horizon.

The problem as we saw it, current search engines such as Yahoo or AltaVista had become saturated with irrelevant results. (Google had not yet gained the prominence it now enjoys).

1.1. Looks Can Kill (data that is)

At this point in time, *CSS* had not yet been widely adopted. To make web pages appear cosmetically appealing, webmasters would include formatting markup *directly in the HTML* as opposed to specifying it via an external style sheet.

For example, to create thin borders, web designers would overlay tables on top of black background tables. (if you were designing web pages in that era, you probably remember how impossible this was to maintain)

While this worked, it provided absolutely nothing for the content of the information. The best we could do was examine the `<title>` tags. Headings were frowned upon, most browsers would render them in large blocky letters.

A whole range of conflicting browser features made for some extremely challenging problems. Some browsers supported *server push* others didn't, all of them handled HTML display in a slightly different manner. This situation did nothing for the search engine spider trying to make sense of it all.

1.2. Structure to Save the Weary Searcher

XML was the new blockbuster technology that promised to solve the vast information wasteland by providing a *context* which could be used for refined searching.

Our solution was to make use of this context as people gradually adopted the new XML approach to web design.

```
<document>
  <company>
    <name>
      Apple
    </name>
  </company>
</document>
```

A context could be used to narrow the search results to a subset of documents. For example, a search of *apple* might reveal millions of pages about *fruit*, *diet*, *bible scripture*, *apple records* and *apple computer*. Our thinking was that XML would allow us to specify a context, for example "*company*" which would narrow the results to documents which had the word *apple* appearing only within the context of companies.

2. Implementation Challenges

Parsing and searching XML involved several challenges in addition to inverted indexing.

The time to completion for this project was a very tight **60 days**. For this reason, we opted to use a relational database as the primary storage medium.

2.1. Unknown Tags

It has been claimed that without an *xml vocabulary* an XML document offers little more than any other plain text.

To that end, our implementation of an XML search engine would first produce a list of elements, the user would then select a tag and narrow the search to results within that element. (much like any other web directory)

To solve the problem of two distinct elements having the same meaning, our plan was to implement a thesaurus at a later point in time.

2.2. Load Balanced Indexing

Obviously, to scale well, our system had to fetch web pages from multiple hosts at the same time.

We broke the process down into separate stages:

Fetching

Retrieve XML documents from remote hosts.

Parsing

This stage would break documents down into *lexical IDs*.

Inserting

This stage was responsible for inserting the lexical IDs into the database.

2.2.1. Multiple Concurrent Spiders

One problem with spidering hosts in parallel is that of basic politeness. Care had to be taken not to fetch more than one document from any one host at the same time.

To accomplish this, we employed a system of *host locking*. We created a semaphore for each host currently being spidered. When a host was "locked" no other spiders could access url's from the same website.

As these were the days prior to *HTTP/1.1*, it was sufficient to lock the hostname, as checking DNS to ensure we weren't spidering the same host by a different name was not an issue. †

† Today, with HTTP/1.1, subdomains and shared IP numbers being common, this system would not work.

2.2.2. Parsing XML

Fortunately, parsing an XML document proved to be a simple affair. Even in these early days, perl had an excellent module to assist with this task.

What was not so obvious was the task of converting words and tags into *lexical IDs*. A separate database query was required for each word. †

We soon discovered that using a local database (via perl's *tied hashes*) could save the milliseconds that would otherwise have been spent querying the database for the lexicon ID.

While it may seem as though shaving a few milliseconds would be insignificant, bear in mind this had to be done for *each* element and word in the document. Couple this with the fact that in another process, the same database would be loaded with repeated inserts and you start to understand how important a local cache of lexical IDs can be.

2.2.3. Inserting Resources into the Database

The problem with XML is that it's a tree structure and does not map well into the relational model. Inserting a document took quite a bit of processing time and resources due to the sheer volume of data produced as a result of performing this mapping between the two models.

Creating the lexical IDs in advance helped this situation to some extent, as did *bulk loading* of the data, however we were never able to completely overcome the limitations of the relational model that would soon come back to haunt us.

2.3. How Many is a Million?

People don't realize just how large a million is until they've tried to index a million documents. A million documents containing just 500 words each is *half-a-billion* words, far more information than could possibly be scanned sequentially.

While this is an enormous challenge even with plain text, it is a much larger challenge with XML.

```
<document>
  <company>
    <name>
      Apple
    </name>
  </company>
</document>
```

The problem is the nesting of the elements, the word **apple** exists within the element *document* as well as the elements *company* and *name*.

Using a purely relational model in the above example, the word "apple" would require *three separate entries* in the database (document, company, name). You can quickly see how this unfolds into a massive size when elements are nested deep or when a document contains many words. Even with lexical IDs, we're still talking about distinct records for each word, *for each element*. ‡

2.4. Investors, Bull Markets and Competition

Perhaps the single most memorable event of the late 90's was the technology stock market. To raise funding for our venture, we needed investors. This is where the 60 day goal came in.

In order to woo investors and gain funding, we had to be the very first with an XML search engine. Therefore, we had promised our investors this project would be completed and operational within a

† A lexical ID is a means of converting a word into a number. For example, the word "apple" could be converted to the number 5.

‡ Although it would be possible to *scan* the database, looking for strings (such as /document/company/name) these would not have been effectively indexed.

time frame of only 60 days.

We certainly met the goal of being first and were able to deliver within the 60 day time frame, however this was not without its costs.

3. Lessons Learned

In many respects, the project was a success. While we may have accomplished a great deal in a hurry, the overall response times of a given query was simply too slow to be practical.

It did however, serve as an example of what this new "XML technology" was capable of. We had used it to spider and index product catalogs and then use this information as a most effective demonstration of contextual searching.

3.1. A Relational Database isn't an XML Database

The most significant problem with our system was the data model, cramming the tree structure of an XML document into a strictly row based relational database involved a tremendous amount of *link-age data*.

3.2. Partition Workloads

While our spidering and indexing had been designed to utilize multiple hosts, a query had not utilized this approach. In retrospect, the URLs should have been hashed and forwarded to dedicated blocks of machines for indexing in such manner as to permit queries to be done in parallel.

3.3. HTML is Here to Stay

While we had every reason to believe that one day web pages would be written in XML, thus far this has proven not to be the case.

Today, HTML is still widely used.

In the end, being first isn't everything. Had we spent more time initially (and taken more risks in the area of experimental indexing strategies) the product would have had a much greater response time and we may have been equipped to deal with today's HTML rather than banking on tomorrow's XML.

Ultimately, the search engine had been re-written by a team of developers, the relational database had been replaced by a *btree* index. Instead of storing the words and elements in a traditional sense, the words and elements were treated equally with only their *positions* recorded. However, by this time it was becoming clear that XML would not replace HTML any time soon, therefore the goals had been adjusted for more specific applications.

Unfortunately, the project was shutdown during the crash of the stock market, all that remains of this XML search engine is goxml.com, a domain owned by a company not involved in any way with the original product.