Under Construction: Website Indexing, Part 2

by Bob Swart

```
<META NAME="author" CONTENT="Bob Swart (aka Dr.Bob - www.drbob42.com)">
<META NAME="title" CONTENT="Dr.Bob's Delphi Clinic">
<META NAME="keywords" CONTENT="Dr.Bob Delphi ObjectPascal Pascal Experts Wizards
  Components C++Builder Programming Borland Knowledge Bolesian Clinic RAD
  Efficiency Performance Source Code Tools Utilities Star Trek">
```

➤ *Listing 1*

```
procedure ScanPage(const FileName: ShortString; WebPage: TNumPage);
var
  f: Text;
  Keyword: ShortString;
begin
  assign(f,FileName);
  reset(f);
  if IOResult = 0 then begin
    Keyword := '';
    while (Keyword = '') and not eof(f) do begin
      readln(f,Keyword);
      if Pos('<TITLE>',UpperCase(Keyword)) > 0 then begin
        Delete(Keyword,1,Pos('<TITLE>',UpperCase(Keyword))+6);
        // remove title prefix
        Delete(Keyword,Pos('</TITLE>',UpperCase(Keyword)),255)
        // remove title postfix
      end else
        Keyword = '' { no title, so clear Keyword again }
    end;
    Titles[WebPages] := Keyword; { Title of Webpage }
    close(f);
    ...
  end
end {ScanPage};
```

➤ *Listing 2*

This month, we'll continue our Delphi internet solutions coverage with the sequel of our 2-part implementation of a mini website search engine, this time focusing on results presentation, search flexibility and finally some query optimisations.

If you've ever visited Alta Vista or Yahoo you already know what a Search Engine is: a place where you can specify keywords in an editbox (sometimes using special combining keywords such as `AND`, `OR`, `NOT` and `NEAR`), and a `Search` button that looks in a big database for you and comes back with a list of webpages (URLs) that contain the specified keywords, most of the time even with a short description, or the first few lines of the webpage itself.

In the last issue, we started to write our own dedicated website search engine. The initial result was a website scanner, indexer and search engine (IndexBob) that could be used to specify a single keyword and search for all the webpages in which this keyword occurs.

Did You Ever Meta Tag?

Last time, we ended with a simple list of URLs that pointed to the webpages in which the specified keyword was used. Unfortunately, nothing else was said about these webpages. Thinking about how to specify some more information made me think back about another way to set up a search engine: using `META` tags. A `META` tag is a HTML tag, to be placed in the `HEADER` of an HTML document, and usually picked up by webcrawlers (sent out by index databases and big web search engines). A few actual `META` tags from my own homepage are shown in Listing 1.

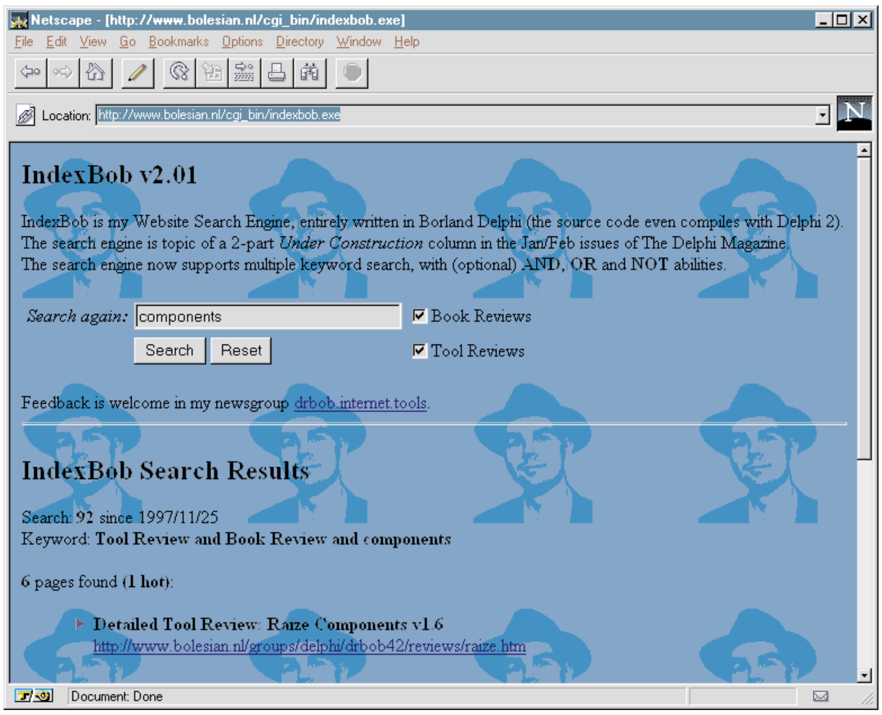
Note the list of keywords in the third `META` tag. These keywords could also have been used to build a keyword index database (instead of actually parsing every webpage). Surely, the listed keywords are accurate, to the point and define precisely what the webpage is all about? On the other hand, synonym keywords (like *Windows Control*, or even simply *Component* or *Expert*) may not be found. And what about keywords that, for some reason, are not even listed, such as *Internet* and *Intranet*? (and I know for a fact my website contains a lot of information about these two topics, but it seems that I just haven't updated my `META` tags in a while).

And that's not even the biggest problem. The biggest problem is the fact that there's no real standard format for using `META` tags. I specified the tagnames "`author`", "`title`" and "`keywords`", but someone else might use "`webmaster`", "`writer`", "`description`", "`topics`" etc and give them the same tag content values. So, for these two main reasons, I've decided *not* to incorporate `META` tags in my search engine, nor use them to describe the webpages themselves (after all, how many of your webpages actually contain sensible `META` tags anyway?).

Entitled

As an alternative, we can either list the first few lines (or first 255 characters) from a webpage, or, perhaps even more suitable, simply use the assigned title of the webpage, which is to be found between the `<TITLE>` and `</TITLE>` tags in the header of the HTML document. For example, if this article were published on my website, I would probably give it the following title in HTML:

```
<HTML>
<HEAD>
<TITLE>Under Construction #20:
  WebSite Indexing (2)</TITLE>
</HEAD>
```


Netscape - [http://www.bolesian.nl/cgi_bin/indexbob.exe]
File Edit View Go Bookmarks Options Directory Window Help
Location: http://www.bolesian.nl/cgi_bin/indexbob.exe

IndexBob v2.01

IndexBob is my Website Search Engine, entirely written in Borland Delphi (the source code even compiles with Delphi 2). The search engine is topic of a 2-part *Under Construction* column in the Jan/Feb issues of The Delphi Magazine. The search engine now supports multiple keyword search, with (optional) AND, OR and NOT abilities.

Search again: components ☑ Book Reviews ☑ Tool Reviews

Search Reset

Feedback is welcome in my newsgroup drbob.internet.tools.

IndexBob Search Results

Search: 92 since 1997/11/25
Keyword: **Tool Review and Book Review and components**

6 pages found (**1 hot**):

Detailed Tool Review: Raize Components v1.6
http://www.bolesian.nl/groups/delphi/drbob42/reviews/raize.htm

➤ *Figure 1*

Using this HTML construct, we only need to define a variable `Titles: Array of Strings` to hold the individual titles (just like the `WebPage: Array of Strings` which holds the URLs) and add some additional parsing code to the procedure `ScanPage` in the version of the unit `Index` that we ended with last month, as shown in Listing 2.

The `Array of Titles` is saved to the file title.bob, so now we have three files that make up the website search engine database: index.bob (keywords and bitset of webpages), pages.bob (actual URLs for each webpage) and title.bob (the title of each webpage).

One more gimmick is to try to locate the first specified keyword that we're looking for in the title of the webpage as well. If the keyword is also present in the title, then we can call it a "hot hit" instead of a regular hit, and I've decided to display this as shown in Figure 1, splitting the "hot hits" and regular hits and also showing the titles just above the URLs themselves.

Figure 1 is from Bolesian's intranet and also illustrates the use of `AND` (which is always optional) and `NOT` (which means use the complement set of the next keyword). We'll get to them shortly.

With good use of the title of a webpage, we can even use this information to add some "meta knowledge" to IndexBob, by putting special information in the title. For example, on my website the title of webpages that contain a book review always starts with "Book Review:" or "Detailed Book Review:" and the title of a third-party tool review always starts with "Tool Review:" or "Detailed Tool Review:".

Based on this information, we can "pre-insert" a few keywords in the list of specified keywords if the user specifies that s/he is only looking for book (or tool) reviews. You can see this as well in Figure 1, as both the "Book Review" and "Tool Review" options have been checked.

AND OR NOT

Looking for just a single keyword doesn't make for a very effective search engine. Often, we'd like to search for a combination of keywords, using `AND` or `NOT`. Sometimes we would even want to list alternatives using `OR`, although this can also be realised by performing a second search. The search for a single keyword results in a set of URLs (the IDs of the webpages). So, we can use the `*` operator to `AND` two result sets and the `+` operator to `OR` (add) two result sets. When using the `NOT` operator, we can

➤ *Listing 3*

```
var
  Found,SubSet: TPageSet;
  _Not, _Or : Boolean = False;
begin
  Keywords := LowerCase(Value('Keyword'));
  Found := [0..WebPages-1]; { initially, everything is found }
  Query := 0; { no queries performed, yet }
  _Not := False;
  _Or := False;
  if root <> nil then
  repeat
    Keyword := GetNextKeyword;
    if Keyword = 'and' then
      { skip }
    else begin
      if Keyword = 'or' then
        _Or := True
      else begin
        if Keyword = 'not' then
          _Not := True
        else begin
          if Length(Keyword) > 2 then begin
            SubSet :=  root.FindKeywordInPages(Keyword);
            if SubSet = [] then
              writeln('<BR>Invalid keyword: <I>',Keyword,'</I>')
            else begin
              if _Not then
                 SubSet := [0..WebPages-1] - SubSet;
              _Not := False;
              if _Or then
                Found := Found + SubSet
              else { and }
                Found := Found * SubSet;
              Inc(Query)
            end
          end
        end
      end
    end
  until Keywords = '';
  if Query = 0 then
    Found := []; { no SubSet found }
  ...
```

simply subtract the result set from a completely filled set, leaving only the bits that weren't in the result set to begin with. This effectively implements the `AND`, `OR` and `NOT` special keyword behaviors as can be seen in Listing 3.

Apart from this processing of the supplied keywords, very little has changed in the original source of IndexBob. So far, that is, because it's time to investigate some speed optimisation techniques for IndexBob to increase the efficiency.

IndexBob Efficiency

Those of you who have known me a little longer than today probably remember that I've always been keen on finding speed optimisations. Although IndexBob only takes about 0.4 seconds to identify the webpages where a single keyword appears, I just want to investigate if we can decrease this time even further. For that purpose, I used calls to `TimeGetTime` (from WINMM.DLL) at different points of execution within IndexBob. It wasn't a big surprise that the one big bottleneck turned out to be loading the index rather than the search itself. Of course, the index consists of over 9,000 keywords that form a binary tree, while locating a given keyword only takes log(9,000) or 14 string compares at maximum.

Generating the resulting output also takes some time, but is dwarfed by the time it actually takes to send the output from the web server back to the client machine, of course. So, to optimise the IndexBob program itself, and minimise the possible strain its execution has on the web server, we need to focus our attention on the index.bob indexfile and especially loading this file.

Last time, we saw that the results of running the Scanner CGI application on my website listed 199 webpages with the longest keyword being 30 characters long. Based on that information, I defined the storage structure for a node in the binary tree to use a set (32 bytes) for the URL id, and a `String[31]` (also 32 bytes, including the length byte) for the keyword, yielding a total `TNode` size of 64 bytes, as can be seen in Listing 4.

Since then, my website has grown a bit (I've started to make the on-line book *Delphi Internet Solutions* available) to 214 webpages. Using the old approach (Listing 4), the complete generated file index.bob would be 582Kb, listing a whopping total of 9094 different keywords.

If we take a look at the size of these different keywords, combined with the number of

```
const
  MaxPage = 255;
  MaxKeyword = 31-8;
type
  TNumPage = 0..MaxPage; { max number of webpages in site }
  TPageSet = Set of TNumPage;   { 255 }
  TKeyword = String[MaxKeyword]; { 31 }
  TNode = record
    Keyword: TKeyword; { 32 bytes }
    URLs: TPageSet;    { 32 bytes }
  end {TNode};
```

➤ *Listing 4*

Len.	1	2	3-4	5-8	9-16	17-32	33-64	65-128	129-214	*Total*
1	0	0	0	0	0	0	0	0	0	*0*
2	0	0	0	0	0	0	0	0	0	*0*
3	220	70	69	53	36	22	31	13	10	*524*
4	267	89	89	90	78	50	55	24	6	*748*
5	380	112	121	102	85	73	40	10	1	*924*
6	452	142	168	143	95	51	26	6	3	*1086*
7	519	141	161	121	95	63	27	4	2	*1133*
8	656	172	148	99	76	40	22	5	1	*1219*
9	519	97	102	74	60	19	7	3	1	*882*
10	412	90	77	62	38	13	6	1	0	*699*
11	384	68	55	32	16	5	2	4	0	*566*
12	262	59	33	11	11	4	2	1	0	*383*
13	189	32	16	5	5	2	1	0	0	*250*
14	166	23	11	1	0	1	0	0	0	*202*
15	123	8	6	2	1	0	0	0	0	*140*
16	87	4	0	0	0	0	1	0	0	*92*
17	73	3	0	0	0	0	1	0	0	*77*
18	48	2	2	0	0	0	0	0	0	*52*
19	23	2	1	0	0	0	0	0	0	*26*
20	29	1	0	0	0	0	0	0	0	*30*
21	19	1	0	0	0	0	0	0	0	*20*
22	13	1	0	0	0	0	0	0	0	*14*
23	10	0	0	0	0	0	0	0	0	*10*
24	5	0	0	0	0	0	1	0	0	*6*
25	3	0	0	0	0	0	0	0	0	*3*
26	3	0	0	0	0	0	0	0	0	*3*
27	2	0	0	0	0	0	0	0	0	*2*
28	1	0	0	0	0	0	0	0	0	*1*
29	1	0	0	0	0	0	0	0	0	*1*
30	1	0	0	0	0	0	0	0	0	*1*
31	0	0	0	0	0	0	0	0	0	*0*
Total	4867	1117	1059	795	596	343	222	71	24	**9094**

➤ *Table 1*

webpages in which each keyword appears, we get the results shown in Table 1.

The rows list the length of the keywords, while the columns specify the (range of) number of pages in which the keywords appears in. It's a little bit surprising at first to actually see that most keywords (4867 out of 9094) uniquely identify a webpage. On the other hand, 24 keywords almost appear on every webpage, and are clearly too common to be of real use for this search engine. As we can also clearly see, most keywords are between 5 and 10 characters in size (note that I already skipped all keywords smaller than 3 characters). In fact, there are only 17 keywords longer than 23 characters.

These analysis results can be used to bring down the size of the indexfile. First of all, let's skip the most common words that appear in, say, more than 100 out of the 214 webpages. That would eliminate a total of 36 most commonly used keywords in my webpages (if you're curious, the list of the most common 36 words on my website is as follows: *aka, all, and, are, bob, book, but, can, code, com, database, delphi, drbob42, drs, for, from, how, more, not, reserved, review, rights, robert, some, swart, that, the, this, using, webmaster, webpage, which, with, www, you* and *your*).

However, if we look closely, we see that some of these words will actually be used in a search query, and while each of them might not be discriminating enough on its own, combined they might just be discriminating enough. So, I've decided *not* to remove all common keywords, but only the ones that consist of a mere 3 letters. This means that I've eliminated *aka, all, and, are, bob, but, can, com, drs, for, how, not, the, www* and *you*; 15 keywords in total.

A further, and by far more significant, reduction can be made by eliminating the 17 *longest* keywords that have a size between 24 and 30 characters. This not only means we end up with a total of 9062 keywords (32 less than we started with, only a 0.5 percent decrease), but also implies that we can now use a `String[23]` to store the keywords, instead of a `String[31]`, which is a 25% decrease in storage space per keyword, or a 12.5% percent decrease in total index.bob filesize to 508 Kb. Hence roughly a 12.5% increase in loadtime and overall efficiency! (Table 2).

In short: the number of keywords went from 9094 to 9062 (or by less than half a percent), but the size of the index.bob file, and hence the loadtime, decreased by 12.5%, as `MaxKeyWord` is now 23 instead of 31 and the size of `TNode` goes from 64 to 56 bytes.

Of course, I must make sure that any word that isn't found (ie isn't in the list of "known" keywords) does not result in an empty result set when combined with keywords that do exist. So a search for "Bob Swart" (ie Bob and Swart) should still yield all webpages that include the word "Swart", as "Bob" is one of the most common keywords in my website and hence removed from the keyword tree. I've implemented this by checking the value of `SubSet` and if it's empty I don't combine it with the `Found` set of URLs, but just ignore it, and list the keyword as being invalid.

Dynamic Strings

There's another way to decrease the index.bob filesize: by making sure each keyword string on disk is only using the exact number of characters it needs. As we noted before, most keywords are between 5 and 10 characters in size, so surely we could reduce the size of index.bob considerably by only writing those string characters to disk, thereby making the size of `TNode` dynamic between 36 and 56 bytes instead of a static 56 bytes. First of all, we need to change the `WriteTree` method (see Listing 5) in the unit `Index` (to

Table 2

Len.	1	2	3-4	5-8	9-16	17-32	33-64	65-100	*Total*
1	0	0	0	0	0	0	0	0	*0*
2	0	0	0	0	0	0	0	0	*0*
3	220	70	69	53	36	22	31	8	*509*
4	267	89	89	90	78	50	55	30	*748*
5	380	112	121	102	85	73	40	11	*924*
6	452	142	168	143	95	51	26	9	*1086*
7	519	141	161	121	95	63	27	6	*1133*
8	656	172	148	99	76	40	22	6	*1219*
9	519	97	102	74	60	19	7	4	*882*
10	412	90	77	62	38	13	6	1	*699*
11	384	68	55	32	16	5	2	4	*566*
12	262	59	33	11	11	4	2	1	*383*
13	189	32	16	5	5	2	1	0	*250*
14	166	23	11	1	0	1	0	0	*202*
15	123	8	6	2	1	0	0	0	*140*
16	87	4	0	0	0	0	1	0	*92*
17	73	3	0	0	0	0	1	0	*77*
18	48	2	2	0	0	0	0	0	*52*
19	23	2	1	0	0	0	0	0	*26*
20	29	1	0	0	0	0	0	0	*30*
21	19	1	0	0	0	0	0	0	*20*
22	13	1	0	0	0	0	0	0	*14*
23	10	0	0	0	0	0	0	0	*10*
Total	4851	1117	1059	795	596	343	221	80	**9062**

generate the index). The compiler conditional BLOCK is defined to enable the new code to compile, otherwise we'll revert to the old code of a TNode of 56 bytes.

Using this method, we end up with an index.bob indexfile of no more than 372,621 bytes! Another size reduction of 25%. The only question is: *will this new method result in a faster loadtime as well?* To test this, we also need to update the ReadNode method in unit IndexBob (see Listing 6), to make sure it can handle the new dynamic layout of the index.bob indexfile.

Unfortunately, the three BlockReads are significantly slower than the single Read we used last month. So, while the new algorithm results in a further 25% reduction in index filesize, the performance actually decreases, and this alternative turns out to be less useful after all.

```
procedure WriteTree(var IndexFile: TIndexFile; root: TTree);
begin
  if root.Prev <> nil then
    WriteTree(IndexFile,root.Prev);
  if (Length(root.node.Keyword) > 3) or
    (Pages(root.node.URLs) <= MaxHits) then begin
   {$IFDEF BLOCK}
    BlockWrite(IndexFile,root.Node.Keyword[0],Ord(root.node.Keyword[0])+1);
    BlockWrite(IndexFile,root.Node.URLs,SizeOf(root.Node.URLs));
   {$ELSE}
    write(IndexFile,root.Node); { for "fixed size" TNodes }
   {$ENDIF}
  end;
  if root.Next <> nil then
    WriteTree(IndexFile,root.Next)
end {WriteTree};
```

➤ *Listing 5*

```
procedure ReadNode(var IndexFile: TIndexFile; root: TTree);
begin
  if root.Prev <> nil then
    ReadNode(IndexFile, root.Prev);
 {$IFDEF BLOCK}
  BlockRead(IndexFile,root.Node.Keyword[0],1);
  BlockRead(IndexFile,root.Node.Keyword[1],Ord(root.Node.Keyword[0]));
  BlockRead(IndexFile,root.Node.URLs,SizeOf(root.Node.URLs));
 {$ELSE}
  Read(IndexFile,root.Node);
 {$ENDIF}
  Inc(Keywords);
  if root.Next <> nil then ReadNode(IndexFile, root.Next)
end {ReadNode};
```

➤ *Listing 6*

Hashing

Another, theoretical, technique we can use to optimise the loadtime of the indexfile is to use a hashing algorithm. This means that we just convert every keyword (24 bytes at this time) to a 4 or 8-byte hashing value. We would need a good hashing algorithm to guarantee that (almost) all keywords are converted to a unique hash value, and indexing the website also means calculating the hash values for each keyword, so indexing could become much slower. However, it should also be clear that the potential benefits for IndexBob are great: for an 8-byte hashing value, a TNode only takes 32 + 8 = 40 bytes, which results in an even smaller filesize for index.bob than when using the dynamic string method. And since comparing binary hash values is much faster than comparing string values, that would surely more than compensate for the calculation of the has value for each specified keyword. In short: for time critical search engines, a good hashing algorithm might be the ideal solution.

Which is where this article ends, since I don't have a perfect hashing routine at hand (and the current efficiency is more than satisfactory anyway). If you are interested in this technique check out Julian Bucknall's article on page 18 and his follow-up next month!

TNeverendingstory...

In this second part of the website search engine article we've seen how to add more descriptive information to the found URL, how to search for more than one keyword (using AND, OR and NOT) and how we could increase the efficiency by almost 15% without sacrificing much in functionality or keywords, and how we can potentially increase the efficiency by at least another 25% by using hashing techniques. The new and complete source code for the scanner, analyser and indexer (Index) and the search engine itself (IndexBob) is on this month's disk. By the time you read this article, IndexBob v2.01 has already been operational for a number on weeks on my website, with some possible additional enhancements since the time I wrote this article (early January 1998). Check my website at www.drbob42.com for updates.

If you have any other ideas for further improvements, or just want to talk about IndexBob, do send me some feedback in my special newsgroup drbob.internet.tools at news.shoresoft.com (there's a link on my website). I welcome any and all feedback.

Next Time Dr.Bob Says...

Next month we'll move away from the internet for a (short) while and start to investigate spelling checkers, *how* do they work, *what* is needed, and of course: *how do I make my own spelling checker for, in and with Delphi?*

Stay tuned...

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a professional knowledge engineer technical consultant using Delphi, C++Builder and JBuilder for Bolesian (at www.bolesian.com), a freelance technical author for *The Delphi Magazine*, co-author of *The Revolutionary Guide to Delphi 2* and the electronic knowledge base *Delphi Internet Solutions*, with topics about Delphi and the internet/intranet. In his spare time, Bob likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 3.5-year old son Erik Mark Pascal and his 1-year old daughter Natasha Louise Delphine.