



**CODE
PROJECT®**
For those who code

Articles » Database » Database » General

hOoT - full text search engine



Mehdi Gholam, 12 Aug 2016

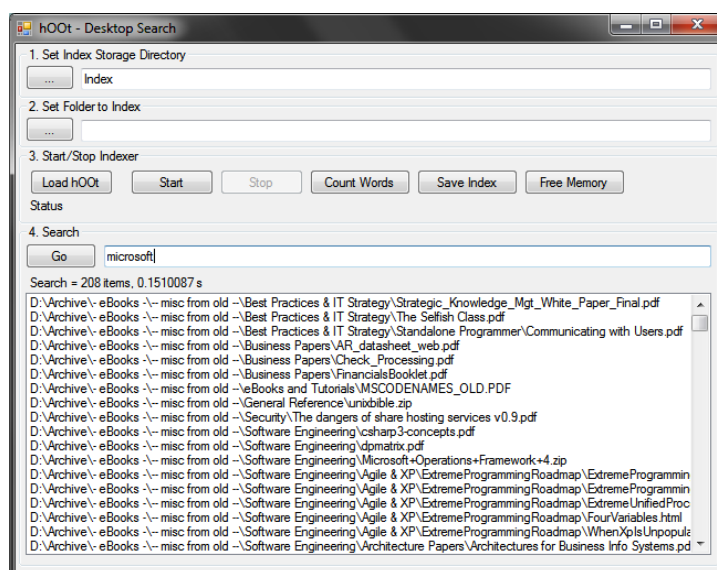
Smallest full text search engine (lucene replacement) built from scratch using inverted WAH bitmap index, highly compact storage, operating in database and document modes



[Download hOoT_v3.3.8.zip](#)
[Download SampleApp_v3.3.8.zip](#)
[Old revisions \(below\)](#)

• Introduction

- Why the name 'hOoT'?
- What is full text searching?
- Why Another Full Text Indexer?
- Features
 - Limitations
- Performance Tests
- Using the Sample Application
 - Code Behind the Sample Application
- Using hOoT in your Code
- How It Works
 - Some Statistics...
 - Index Engine
 - Query Engine
 - What's an inverted index?
 - The power of bitmap indexes
 - Files generated
 - File Formats : *.WORDS
 - File Formats : *.DOCS
 - File Formats : *.BITMAP
 - File Formats : *.DELETED
 - File Formats : *.REC
 - File Formats : *.IDX
- Appendix v2.0
- Appendix v3.3.7
- Revisions
- History



Introduction

hOoT is a extremely small size and fast embedded full text search engine for .net built from scratch using an inverted WAH bitmap index. Most people are familiar with an Apache project by the name of Lucene.net which is a port of the original java version. Many people have complained in the past why the .net version of lucene is not maintained, and many unsupported ports of the original exists. To circumvent this I have created this project which does the same job, is smaller, simpler and faster.

hOoT is part of my upcoming **RaptorDB document store database**, and was so successful that I decided to release it as a separate entity in the meantime.

hOoT uses the following articles :

- WAH compressed BitArray found here ([WAHBitArray.aspx](#))
- mini Log4net replacement found here (<http://www.codeproject.com/KB/miscctrl/minilog4net.aspx>)
- MurMur2 hash index and storage file from **RaptorDB** found here ([RaptorDB.aspx](#))
- **fastJSON** serializer found here (<http://www.codeproject.com/KB/IP/fastJSON.aspx>)
- IFilter without COM by **Eyal Post** found here (<http://www.codeproject.com/KB/cs/IFilter.aspx>) for the sample application

Based on the response and reaction of users to this project, I will upgrade and enhance **hOoT** to full feature compatibility with lucene.net, so show your love.

Why the name 'hOoT'?

The name came from the famous Google search footer and while looking for owl logos (goooooogle ~ hOooooOOt).

What is full text searching?

Full text searching is the process of searching for words in a block of text. There are 2 aspects to full text indexers / searchers :

1. **Existence** : means finding words that exist in the many blocks of texts stored (e.g. 'bob' is in the PDF documents stored).
2. **Relevance** : meaning the text blocks returned are delivered by a ranking system which sorts the most relevant first.

The first part is easy, the second part is difficult and there is much contention as to the ranking formula to use. **hOoT** only implements the first part in this version, as most of use and need the existence more in our applications than relevance, especially for database applications.

Why Another Full Text Indexer?

I was always fascinated by how Google searches in general and lucene indexing technique and its internal algorithms, but it was just too difficult to follow and anyone who has worked with lucene.net will attest that it is a **complicated** and **convoluted** piece of code. While some people are trying to create a more .net optimized version, the fact of the matter is that it is not easy to do with that code base. What amazes me is that nobody has rewritten it from scratch. **hOoT** is much simpler, smaller and faster than lucene.net.

One of the reasons for creating **hOoT** was for implementing full text search on string columns in RaptorDB - the document store version. Hopefully more people will be able to use and extend **hOoT** instead of lucene.net as it is much easier to understand and change.

Features

hOoT has been built with the following features in mind:

- Blazing fast operating speed (see performance test section)
- Incredibly small code size.
- Uses WAH compressed BitArrays to store information.
- Multi-threaded implementation meaning you can query while indexing.
- Tiny size only 38kb DLL (lucene.net is ~300kb).
- Highly optimized storage, typically ~60% smaller than lucene.net (the more in the index the greater the difference).
- Query strings are parsed on spaces with the **AND** operator (e.g. all words must exist).
- Wildcard characters are supported (*,?) in queries.
- **OR** operations are done by default (like lucene).
- **AND** operations require a (+) prefix (like lucene).
- **NOT** operations require a (-) prefix (like lucene).

Limitations

The following limitations are in this release:

- File paths in document mode is limited to 255 bytes or equivalent in UTF8.
- Exact strings are not currently supported (e.g. "alice in wonderland").
- ~~Wildcards (*,?) are not currently supported.~~
- ~~OR not currently supported in queries (e.g. bob OR alice).~~
- ~~NOT not currently supported in queries (e.g. bob NOT alice).~~
- Parenthesis not currently supported in queries.
- Ranking and relevance is not currently supported.
- Searching user defined document fields are not currently supported.

Performance Tests

Tests were performed on my notebook computer with the following specifications: AMD K625 1.5Ghz, 4Gb Ram DDRII, Windows 7 Home Premium 64bit, Win Index 3.9. **hOOT** generates a lot of log information which you can use to see what is going on inside the engine. Using the sample application I performed a crawl over a collector of around 4600 files of about 5.8Gb of data in various formats here are the results using grep on the output files (bear in mind that the application was compiled under .net 4 and running 64bit, 64bit IFilterers were installed also) :

Document Count	4,490
Total time	~22 min
Index file size	29 Mb
Text size total	632,827,458 characters
Total hOOT Indexing time	56767.2471 ms ~ 56.7 secs
Total hOOT writing document info time (fastJSON serialize)	110632.3282 ms ~ 110 secs
Total words in hOOT	~290,000

As you can see the indexing engine is blazing fast going through **632mb of text in 57secs**, the difference in time to the total 22 minutes is to do with the IFilter extraction time. On the query side for the above document count all **queries perform in about 1 ms** on the engine side and as you can see in the sample picture a search for "microsoft" gives back 208 items which took 0.151 seconds, again the difference is to do with the document deserialization time.

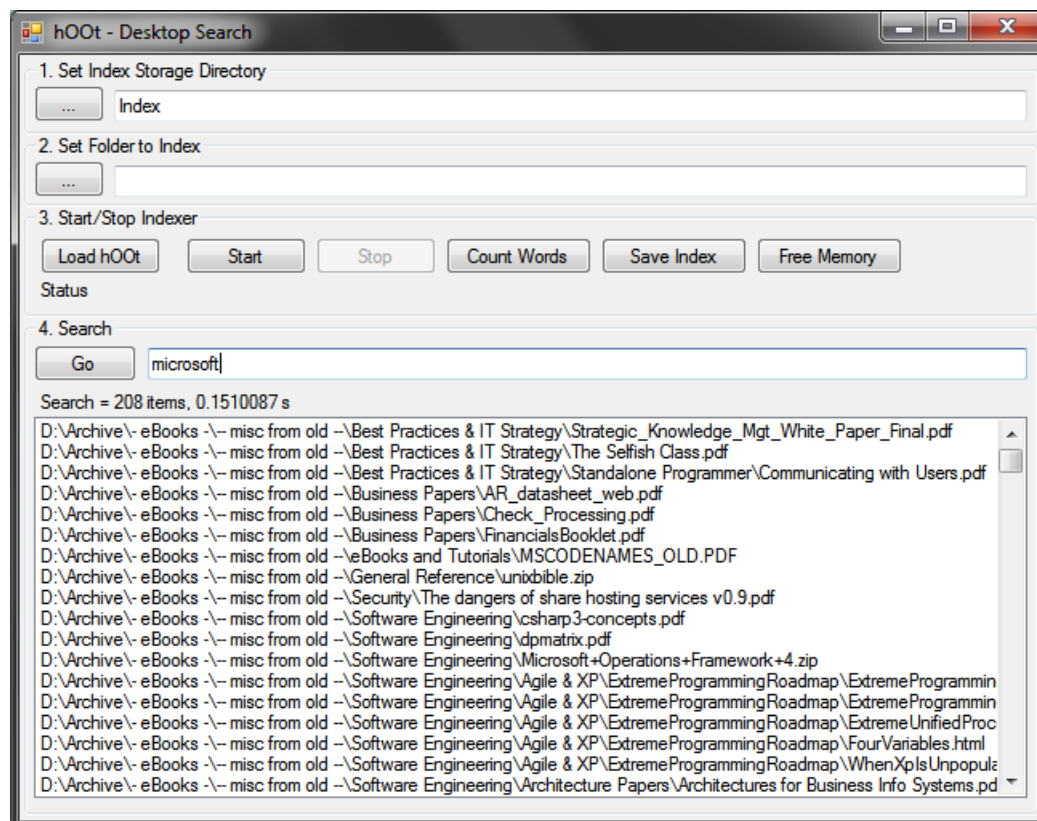
By comparison **lucene.net** has the following :

Index file size	70.5 Mb
Total time	~28 min

Performance Tests v1.2

Using a better word extractor the index size is reduced by about **19% to 24Mb**.

Using the Sample Application



The picture above is the obligatory desktop search application built on **hOot** . To use the application do the following :

1. Set the index storage directory in the (1) group box, this will store all the index information.
2. Choose a folder where you want to crawl for content in the (2) group box.
3. In the (3) group box you can do the following :
 1. Load hOot : This will load **hOot** so you can query an existing index.
 2. Start Indexing : This will load **hOot** and start indexing the directory you specified.
 3. Stop Indexing : This will come active after you have started indexing so you can stop the process.
 4. Count Words : This will show the number of words in **hOots** dictionary
 5. Save Index : This will save anything in memory to disk.
 6. Free memory : This will call the internal free memory method on **hOot** (this will only free the bitmap storage and not release the cache).
4. You can search for content in the (4) group box, the label will show the count and time taken.
 1. To open the file just double click on the file path in the list box.

This is just a demo that show cases **hOot** , although it works as is, it does need some bells and whistles for a real application. To use this sample effectively please instal the following IFilter handlers beforehand :

- Foxit PDF filter found here (<http://www.foxitsoftware.com/>) [never use Adobe as it is dismally slow].
- Microsoft Office 2010 IFilter found here (<http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=17062>)

Changes in v1.5

As of version 1.5 queries follow the lucene style of requiring prefixes and support for wildcards have been added as follows:

prefix (+) character for **AND** operations :

```
+microsoft +testing
```

Search for "microsoft" and "testing" and all words must exist

prefix (-) for **NOT** operations :

```
microsoft -testing
```

Search for "microsoft" excluding "testing"

default is the **OR** operation :

```
microsoft testing
```

Search for "microsoft" or "testing" and any word can exist

for wild cards use (*,?) like old style DOS searches and all the results will be **OR**ed together:

```
m?cro*
```

Search for "macro" or "micro" or "microsoft" or "microphone" or ...

Code Behind the Sample Application

The code behind the sample application is pretty easy to understand, it uses the **BackgroundWorker** class to to the indexing with progress events for the UI to show the files done.

```
private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
{
    string[] files = e.Argument as string[];
    BackgroundWorker wrk = sender as BackgroundWorker;
    int i = 0;
    foreach (string fn in files)
    {
        if (wrk.CancellationPending)
        {
            e.Cancel = true;
            break;
        }
        backgroundWorker1.ReportProgress(1, fn);
        try
        {
            TextReader tf = new EPocalipse.IFilter.FilterReader(fn);
            string s = "";
```

```

        if (tf != null)
            s = tf.ReadToEnd();

        h.Index(new Document(fn, s), true);
    }
    catch { }
    i++;
    if (i > 300)
    {
        i = 0;
        h.Save();
    }
}
h.Save();
h.OptimizeIndex();
}

```

Using hOot in your Code

Using **hOot** is simple and straight forward.

```

// specify the index folder to store information and the file name for the index files
Hoot hoot = new Hoot("d:\\IndexFolder" , "documents");

hoot.FreeMemory(false); // will free bitmap memory

// database mode of operation
hoot.Index(10, "some string from a database"); // index the contents of rec# 10

// document indexing mode of operation
hoot.Index(new Document("d:\\filename.ext",string_from_file));

// optimize the bitmap index size
hoot.OptimizeIndex();

```

You can inherit from the Document object and store any extra information your application needs as properties (lucene uses dictionary values which isn't nice at compile time and requires debugging at run time if you misspelled etc.).

```

public class myDoc : Hoot.Document
{
    public string Extension {get;set;}
    // any other properties you need
}

```

Because **hOot** uses **fastJSON** to store the document you created, it gives you back what ever you saved, as a first class object.

New in v1.2

```

foreach(string filename in hoot.FindDocumentNames("microsoft"))
{
    // a faster way to get just the filename instead of a Document object
}

```

How It Works

hOot operates in the following 2 modes :

1. **Database mode** : where you want to index columns in a database and you supply the row number yourself from the database.
2. **Document mode** : where you give **hOot** a document file which will be serialized and stored and document numbers generated for it.

hOot has 2 main parts which are the following :

1. **Indexer Engine** : is responsible for updating and handling the index generated.
2. **Query Engine** : is responsible for parsing the query text and generating an execution plan.

To optimize memory usage the indexer can free up memory in the following 2 stages :

1. Compress bitmap indexes in memory and free the BitArray storage.
2. Unload cached word bitmap indexes completely.

For the most part you would use stage one, but if you are going to index 100's millions of documents the second stage is going to be necessary.

Some Statistics...

A quick search in the internet reveals the following statistics about the English language:

- There are about 250,000 English words of which around 175,000 are in use (Oxford English Dictionary).
- The maximum length of a word in English is 28 characters.

After going through the words extracted from the IFilters it is easy to see that there are problems with some document formatting as some sentences don't have spaces and the words are stuck together. Also for programming documents CamelCase words are prevalent.

For this reason as of version 1.2 the word extractor in **hOot** will do the following:

- CamelCase words are broken up.
- Word lengths greater than 60 characters are ignored
- Word lengths less than 2 are ignored

Index Engine

The index engine is responsible for the following :

- Load words into memory.
- Load bitmap index data on demand.
- Extract word statistics from the input text.
- Update the word bitmap index.
- Free memory and bitmap cache.
- Write the words to disk.
- Write the bitmap index to disk.

Query Engine

The query engine will do the following :

1. Parse the input string of what to search for into words
2. Extract the bitmap index for all those words used in the search string
3. Execute the bitmap arithmetic logic
4. AND the result with the NOT of the deleted documents bitmap, to filter out deleted documents.
5. Enumerate the resulting bitmap
 - in the case of database mode give you a list of record numbers
 - in the case of document mode seek the document number and give you a list of documents from the documents storage.

What's an inverted index?

An inverted index is a special index which stores the words to bitmap conversion data. In a normal index you would store what words are in a certain document, an inverted index is the opposite given a word 'x' what documents have this word is stored.

For example if you have the following :

- Document number 1 : "to be or not to be, that is the question ..."
- Document number 2 : "Romeo! Romeo! where art thou Romeo! ..."

Would generate the following when parsed by the **GenerateWordFreq** method:

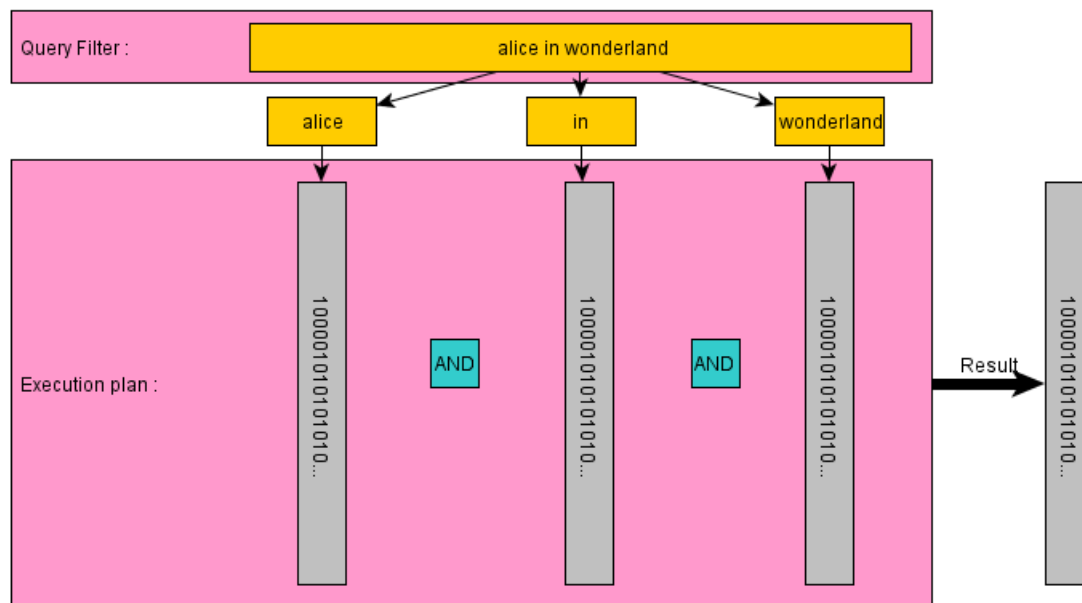
- words : {"to" count=2, doc#=1 }, {"be" count=2, doc#=1}, {"or" count=1, doc#=1}
- words : {"romeo" count=3, doc#=2}, {"where" count=1, doc#=2} ..

And the following word bitmaps would be updated (1 being the existence of that word in the index position of the document number, 0 being not found):

- to : 1000000...
- be : 1000000...
- or : 1000000...
- romeo : 0100000...
- where : 0100000...

The power of bitmap indexes

To see the power of bitmap indexes take a look at the following diagram :



As you can see for the query string "alice in wonderland" (without the quotes) **hOot** will do the following :

- The filter parser extracts the words "alice", "in", "wonderland".
- Seek the associated bitmap indexes for the words.
- Execute the query arithmetic logic in this case a logical AND operation to yield a resulting bitmap index.

The resulting bitmap is the index based record numbers to the document, for example if the result is 0001100... then the documents 4,5... (starting from index number 1) contain all the words "alice", "in", "wonderland".

This is in marked contrast to lucene indexing scheme which saves the record numbers literally albeit in an optimized notation. **hOot** offers huge space savings in index size especially with the WAH compression used, and as an added bonus, performance is extremely fast.

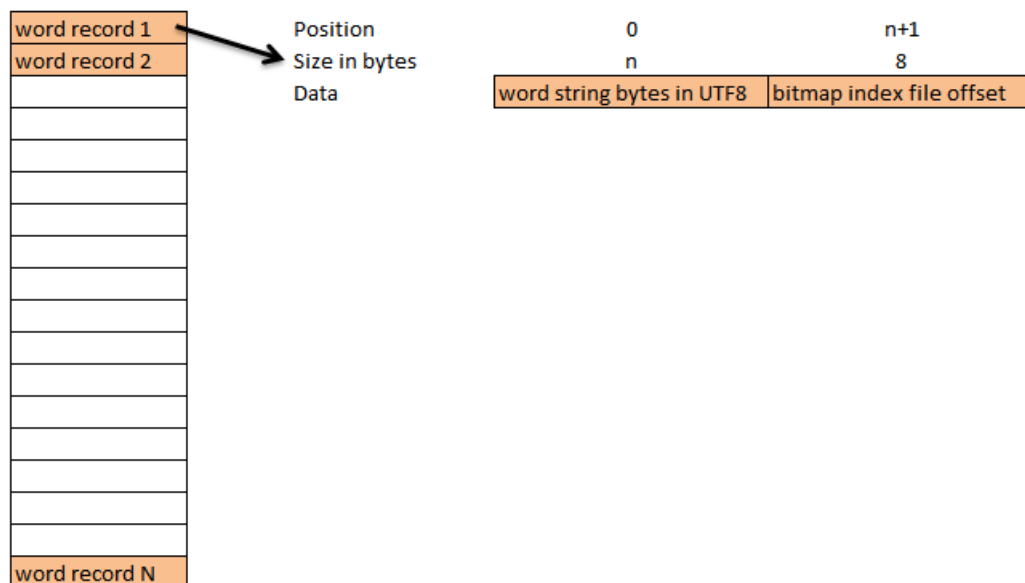
Files generated

The following files are generated in the Index directory:

- filename.WORDS : stores the words extracted from the input text.
- filename.DOCS : stores the document information.
- filename.BITMAP : stores the bitmap information for the words stored.
- filename.DELETED : stores the deleted document indexes.
- filename.REC : stores the data file offsets for document numbers.
- filename.IDX : stores the mapping between document file names and document numbers
- log.txt : stores the logging debug and error messages.

File Formats : *.WORDS

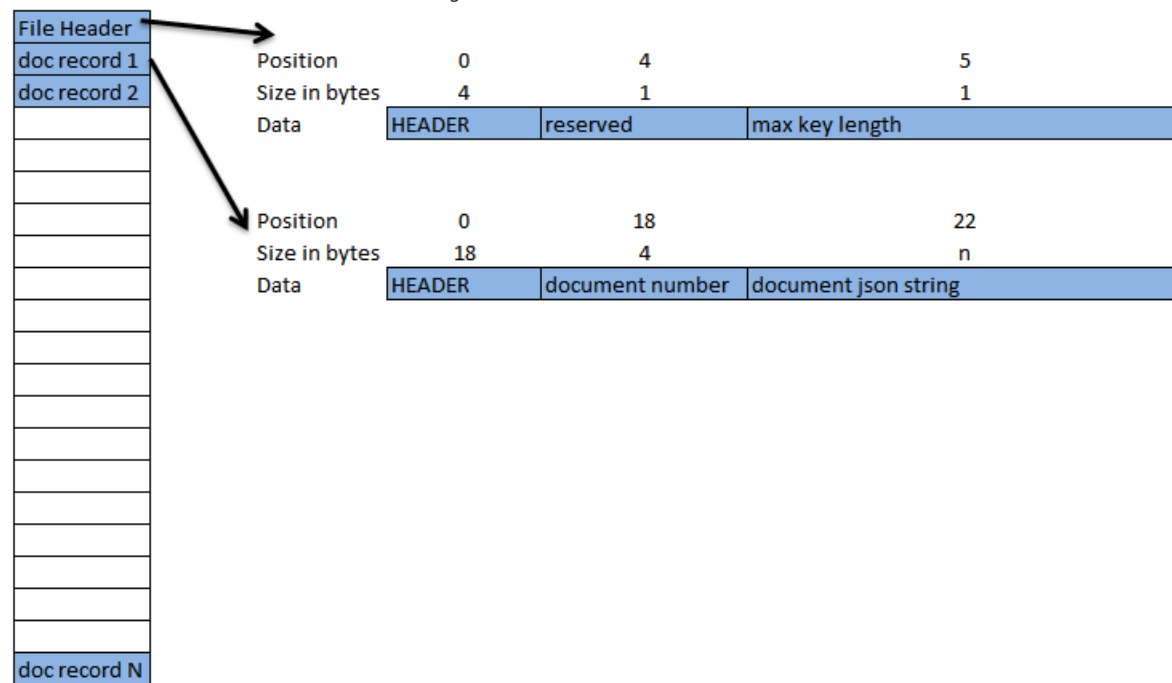
Words are stored in the following format on disk :



The bitmap index offset will point to a record in the *.bitmap file.

File Formats : *.DOCS

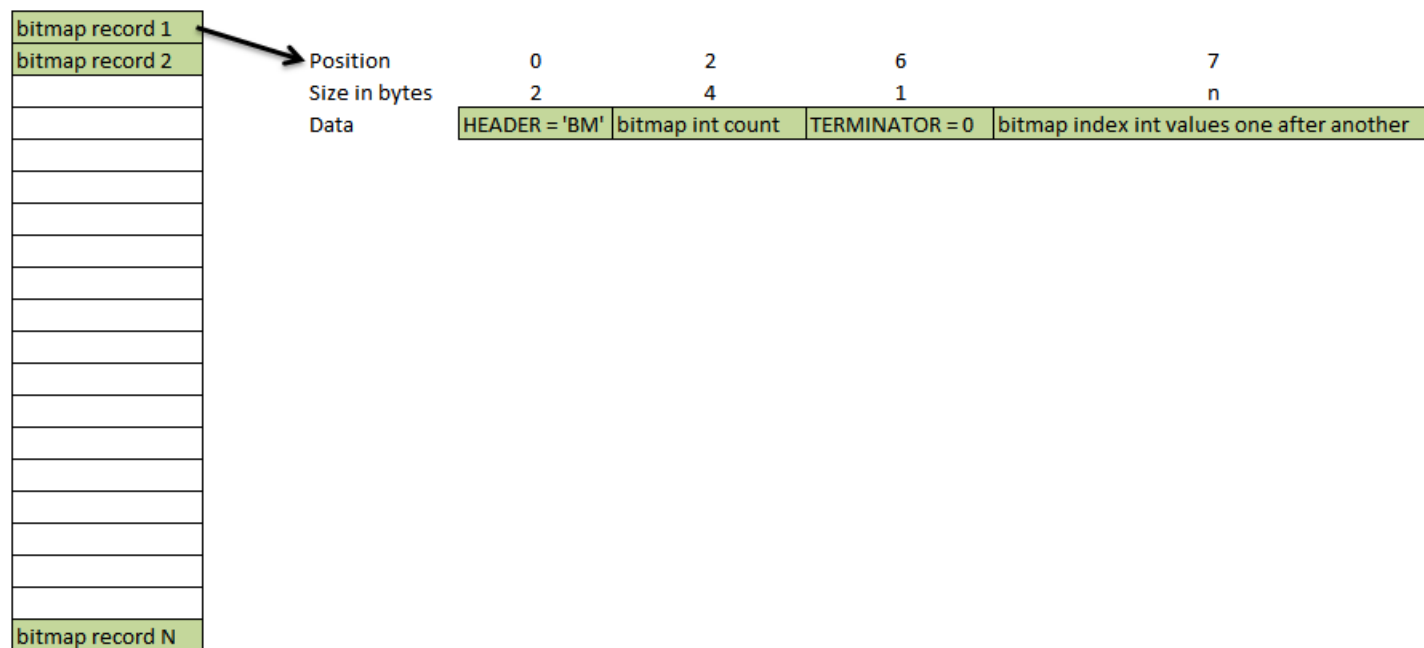
Documents are stored in JSON format in the following structure on disk:



This format came straight from **RaptorDB** and is being used there, the only modification in **hOot** is that the **MaxKeyLength** is set to 1. Also because the records are variable in length the *.RECS file maps the record number to an offset in this file for data retrieval.

File Formats : *.BITMAP

Bitmap indexes are stored in the following format on disk :



The bitmap row is variable in length and will be reused if the new data fits in the record size on disk, if not another record will be created. For this reason a periodic index compaction might be needed to remove unuesd records left from previous updates.

File Formats : *.DELETED

Deleted index file is just a series of **int** values that represent deleted files with no special formatting.

File Formats : *.REC

Rec file is a series of **long** values written to disk with no special formatting. These values map the record number to an offset in the DOCS storage file.

File Formats : *.IDX

IDX file is MurMur2 Hash dictionary for mapping document file names to document numbers. This is used in document mode.

Appendix v2.0

Finally I got round to updaing **hOot**, most of the updates are post backs from **RaptorDB** which are for stability, bug fixes and performance improvements, things like locks and multithreading issues.

The bitmap file format has changed to support index offsets and the storage file has been updated to support deleted items, so the previous files will not work in this release.

hOot now supports incremental indexing of documents and will check if the files exists in the index before indexing. You can **RemoveDocument(filename)** now and the file will be removed from the index results.

File information like dates and size is now stored in the storage file, so in the future it will be possible to check changed files and index updated documents.

Appendix v3.3.7






















Finally found some time to give **hOot** some love and post back a lot of changes made in the meantime in **RaptorDB**. Most of which are bug fixes. Notable added changes are the ability to inherit your own class from **Document** and add properties, so you can now do :

```
/// <summary>
/// Sample doc override
/// </summary>
public class myDoc : Document
{
    public myDoc(FileInfo fileinfo, string text) : base(fileinfo, text)
    {
        now = DateTime.Now;
    }
}
```

```
// other data I want to save
public DateTime now;
}
```

Also the word extracting portion has now been extracted into it's own class so going forward you can more easily change the tokenizer.

Revisions

-  [Download Hoot_v1.0.zip - 52.78 KB](#)
-  [Download Hoot_v1.1.zip - 75.21 KB](#)
-  [Download SampleApp_EXE_v1.1.zip - 41.36 KB](#)
-  [Download Hoot_v1.2.zip - 54.09 KB](#)
-  [Download SampleApp_EXE_v1.2.zip - 42.16 KB](#)
-  [Download Hoot_v1.3.zip - 54.15 KB](#)
-  [Download SampleApp_EXE_v1.3.zip - 42.26 KB](#)
-  [Download Hoot_v1.4.zip - 55.98 KB](#)
-  [Download SampleApp_EXE_v1.4.zip - 42.57 KB](#)
-  [Download Hoot_v1.5.zip - 56.35 KB](#)
-  [Download SampleApp_EXE_v1.5.zip - 42.96 KB](#)
-  [Download Hoot_v2.0.zip - 65.6 KB](#)
-  [Download SampleApp_EXE_v2.0.zip - 51.6 KB](#)
-  [Download Hoot_v2.1.zip - 66.8 KB](#)
-  [Download SampleApp_EXE_v2.1.zip - 52.1 KB](#)
-  [Download Hoot_v2.2.zip](#)
-  [Download SampleApp_EXE_v2.2.zip - 53 KB](#)
-  [Download Hoot_v2.2.1.zip - 66.8 KB](#)
-  [Download SampleApp_EXE_v2.2.1.zip - 53 KB](#)
-  [Download hOot_v3.3.7.zip](#)
-  [Download SampleApp.exe v3.3.7](#)

History

- **Initial release v1.0:** 12th July 2011
- **Update v1.1 :** 16th July 2011
 - tweaked parameters and reduced index size by 46% (now less than half of lucene)
 - speed increase bitmap index save 5x
 - bug fix sample ui
 - bug fix bitarray resize
 - thread safe internals
 - code refactoring
 - OptimizeIndex() implemented
- **Update v1.2 :** 21st July 2011
 - FindDocumentFileNames() for faster string only return
 - Better word extractor ~19% smaller index
 - breaks up camel case compound words
 - ignores strings >60 chars and less than 2 chars
 - Updated the statistics section
- **Update v1.3 :** 23rd July 2011
 - bug fix bitarray arithmetic
 - fix UI listbox flicker
- **Update v1.4 :** 26th July 2011
 - replaced WAHBitarray with v2
 - ~9x bitmap save speed increase

- ** index must be rebuilt from previous version **

- **Update v1.5** : 7th August 2011

- Thanks to **Dave Dolan** for the query evaluation logic
- added support for wildcard characters (*,?)
- added support for AND (+) , NOT (-) queries

- **Update v2.0** : 24th December 2012

- updated and embedded fastjson v2.0.11 in the project
- post back code from RaptorDB
- thread safe locks updates
- bug fix logger threading
- restructured source code
- fixed sample form tab order (thanks to **Sergey Kryukov**)
- added the ability for incremental indexing
- storing document file information for future checking

- **Update v2.1** : 24th May 2013

- upgrade to fastJSON v2.0.15
- bug fix last word missing last character

- **Update v2.2** : 15th June 2013

- bug fix WAHBitArray
- code sync with RaptorDB

- **Update v2.2.1** : 22nd June 2013

- bug fix WAHBitArray

- **Update v3.3.7** : 5th August 2016

- synced code with **RaptorDB v3.3.7**
- changed **FindDocuments()** to generic
- fixed the ability to inherit from **Document** for your own properties
- fixed registry read **IFilter** on 64bit
- search fixes "microsoft -oracle +google -app"
- extracted **tokenizer** to own class with better word parsing
- added build version with auto increment
- diversified log messages
- log level implementation

- **Update v3.3.8** : 12th August 2016

- bug fix search terms
- tokenizer breaks a.b.c words and numbers

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author



Mehdi Gholam

Architect -
United Kingdom

Mehdi first started programming when he was 8 on BBC+128k machine in 6512 processor language, after various hardware and software changes he eventually came across .net and c# which he has been using since v1.0. He is formally educated as a system analyst Industrial engineer, but his programming passion continues.

* Mehdi is the 5th person to get 6 out of 7 Platinum's on Code-Project (13th Jan'12)

* Mehdi is the 3rd person to get 7 out of 7 Platinum's on Code-Project (26th Aug'16)

Comments and Discussions

 **372 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/224722/hOot-full-text-search-engine> to post and view comments on this article, or click [here](#) to get a print view with messages.