"Analyzing two platforms .NET and Java for developing a search engine"

A Thesis
Submitted to the Department of Computer Science and Engineering
Of
BRAC University
By

Shoman Bhuiyan
ID: 02201046

In partial fulfillment of the
Requirements for the Degree
Of
Bachelor of Science in Computer Science
May 2008

Supervisor: Abdussamad Ahmed Muntahi (Lecturer)
Department of Computer Science and Engineering
BRAC University

BRAC University, Dhaka, Bangladesh

Thesis Topic: Analyzing two platforms – .NET and Java for developing a search engine

Abstract:

There are several platforms available for developing a search engine. We can create search engine with php, Visual Basic, C#, J#, Java, DotNet and with other platforms as well. Among those, DotNet and Java are chosen for popularity. Both Java and DotNet use Apache lucene tools for search engine development. The criteria for this analysis were availability, readiness and depth of tools and features from user perspective. Java found to be a better choice and was used to develop a search engine for BRAC University website.

INTRODUCTION

A software that builds an index on text and answer queries using that index.
Searching can be performed on local file system or during web surfing.
A **Web search engine** is a search engine designed to search for information on the World Wide Web. Information may consist of web pages, images and other types of files.
Some search engines also provides data available in newsgroups, databases, or open directories.
Unlike Web directories, which are maintained by human editors, search engines operate algorithmically or are a mixture of algorithmic and human input.
A search engine offers scalability, relevance ranking, integrates different data sources (email, web pages, files, database and so on)

The very first tool used for searching on the Internet was Archie. It was created in 1990 by Alan Emtage, a student at McGill University in Montreal. The program downloaded the directory listings of all the files located on public anonymous FTP (File Transfer Protocol) sites, creating a searchable database of file names; however, Archie did not index the contents of these files.

Search Engine

Search engine takes a file or website as an input and perform parsing operation upon that file. It creates index upon the parse file. Based on the created index, search engine responses to user input.

Search Engines for the general web (like all those listed above) do not really search the World Wide Web directly. Each one searches a database of the full text of web pages automatically harvested from the billions of web pages out there residing on servers. When you search the web using a search engine, you are always searching a somewhat stale copy of the real web page. When you click on links provided in a search engine's search results, you retrieve from the server the current version of the page.

Search engine databases are selected and built by computer robot programs called spiders. These "crawl" the web, finding pages for potential inclusion by following the links in the pages they already have in their database (i.e., already "know about"). They cannot think or type a URL or use judgment to "decide" to go look something up and see what's on the web about it. (Computers are getting more sophisticated all the time, but they are still brainless.)

If a web page is never linked to in any other page, search engine spiders cannot find it. The only way a brand new page - one that no other page has ever linked to - can get into a search engine is for its URL to be sent by some human to the search engine companies as a request that the new page be included. All search engine companies offer ways to do this.

After spiders find pages, they pass them on to another computer program for "indexing." This program identifies the text, links, and other content in the page and stores it in the search engine database's files so that the database can be searched by keyword and whatever more advanced approaches are offered, and the page will be found if your search matches its content.

Many web pages are excluded from most search engines by policy. The contents of most of the searchable databases mounted on the web, such as library catalogs and article databases, are excluded because search engine spiders cannot access them. All this material is referred to as the "Invisible Web" -- what you don't see in search engine results.

SEARCH ENGINES	BOOLEAN	DEFAULT	PROXIMITY	TRUNCATION	FIELDS	LIMITS	STOP	SORTING
Google Review	-, OR	And	Phrase	No (stems) word in phrase	intitle, inurl, link, site, more	Language, filetype, date, domain	Few, + searches	Relevance, site
Yahoo! Review	AND, OR, NOT, ( ), -	And	Phrase	No word in phrase	intitle, inurl, link, site, more	Language, file type, date, domain	No	Relevance, site
Ask Review	-, OR	And	Phrase	No	intitle, inurl, site	Language, site, date	Yes, + searches	Relevance, metasites
Live Search Review	AND, OR, NOT, ( ), -	And	Phrase	No	intitle, site, loc, url	Language, site	Varies, + searches	Relevance, site, sliders
Gigablast Review	AND, OR, AND NOT, ( ), +, -	And	Phrase	No	title, site, ip, more	Domain, type	Varies, + searches	Relevance
Exalead Review	AND, OR, NOT, ( ),-	And	Phrase, NEAR	Yes and stems	intitle, inurl, link, site	Language, file type, date, domain	Varies, + searches	Relevance, date

Figure 1: Different popular search engines with supported parameters

Google has one of the largest databases of Web pages, including many other types of web documents (blog posts, wiki pages, group discussion threads and document formats (e.g.,

PDFs, Word or Excel documents, PowerPoints). Despite the presence of all these formats, Google's popularity ranking often makes pages worth looking at rise near the top of search results.

Things You CAN Do in Google, Yahoo!, and Ask.com	Things NOT Supported in Google, Yahoo!, or Ask.com
• **Phrase Searching** by enclosing terms in double quotes • **OR** searching with capitalized OR • **- excludes, + requires** exact form of word • **Limit** results by language in Advanced Search	• **Truncation** - use OR searches for variants (airline OR airlines) • **Case sensitivity** capitalization does not matter

Figure 2: Supported and non supported features for Google and Yahoo search engines

Search Engine	Google www.google.com	Yahoo! Search search.yahoo.com	Ask.com www.ask.com
Links to help	Google help	Yahoo! help	Ask.com help
Size, type See tests and more charts.	HUGE. Size not disclosed in any way that allows comparison. Probably the biggest.	HUGE. Claims over 20 billion total "web objects."	LARGE. Claims to have 2 billion fully indexed, searchable pages.
Noteworthy features	Popularity ranking using PageRank™ emphasizes pages most heavily linked from other pages. Many additional databases including Book Search, Scholar (journal articles), Blog Search, Patents, Images, etc.	Shortcuts give quick access to dictionary, synonyms, patents, traffic, stocks, encyclopedia, and more.	Subject-Specific Popularity™ ranking. Suggests broader and narrower terms. AskEraser privacy option.
Boolean logic (what's this?)	Partial. AND assumed between words. Capitalize OR. ( ) accepted but not required. In Advanced Search, partial Boolean available in	Accepts AND, OR, NOT or AND NOT. *Must be capitalized.* ( ) accepted but not required.	Partial. AND assumed between words. Capitalize OR. - excludes. No ( ) or nesting.

	boxes.		
+Requires/ -Excludes (what's this?)	- excludes + will allow you to retrieve "stop words" (e.g., +in)	- excludes + will allow you to search common words: "+in truth"	- excludes + will allow you to retrieve "stop words" (e.g., +in)
Sub-Searching (what's this?)	The search box at the top of the results page shows your current search. Modify this (e.g., add more terms at the end.)	The search box at the top of the results page shows your current search. Modify this (e.g., add more terms at the end.)	The search box at the top of the results page shows your current search. Modify this (e.g., add more terms at the end.)
Results Ranking (what's this?)	Based on page popularity measured in links to it from other pages: high rank if a lot of other pages link to it. Fuzzy AND also invoked. Matching and ranking based on "cached" version of pages that may not be the most recent version.	Automatic Fuzzy AND.	Based on Subject-Specific Popularity™, links to a page by related pages.
Field limiting (what's this?)	link: site: intitle: inurl: Offers U.S.Gov't Search and other special searches. Patent search.	link: site: intitle: inurl: url: hostname: (Explanation of these distinctions.)	intitle: inurl: site: last:[time period] (Details)
Truncation, Stemming (what's this?)	No truncation. Stems some words. Search variant endings and synonyms separately, separating with OR (capitalized): *airline OR airlines*	Neither. Search with OR as in Google.	Neither. Search with OR as in Google.
Language	Yes. Major Romanized and non-Romanized languages in Advanced Search.	Yes. Major Romanized and non-Romanized languages.	Yes. Major Romanized languages. Use Advanced Search to limit.
Translation	Yes, in Translate this page link following some pages. To and sometimes from English and major European languages and Chinese, Japanese, Korean. Ues its own	Yes.	N

	translation software with user feedback.		

Figure 3: Some Ways the Recommended Search Engines Differ

Developing a Search Engine:

There are some steps that might be followed for developing search engines. We have to add documents and create index upon the files.

Steps for developing a search engine for .net platform and java

- ☐ Creating the index
- ☐ Adding the documents
- ☐ Optimizing and saving the document
- ☐ Opening the index for searching
- ☐ Searching
- ☐ Query highlighting

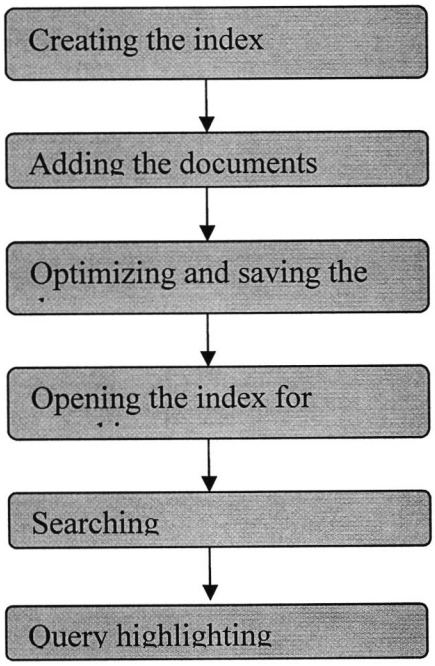

Figure4: Block diagram for developing a search engine.

DotNet platform based indexing with SearchBlackBox:

90% of the time, modern office software operates using textual data. Various pieces of text are spread everywhere on our disks, local and global networks, etc. Thousands of developers spend months trying to provide search capabilities to end-users. Some of them organize their textual data via folder trees, and create searchable indexes. Others use third-party database engines and try to adapt them for their specific needs. That's all in the past now. SearchBlackBox SDK introduces an API that achieves a brand new level of flexibility. The SearchBlackBox API allows you to add search capabilities to your own applications in minutes.

Index each and every document. SearchBlackBox SDK automatically indexes all kinds of textual documents whether they are structured or unstructured, and regardless of their format and location.
Customize searching with modern search techniques. Fuzzy, proximity, and range queries are all at your service.

Control the output. SearchBlackBox SDK allows you to control the way results are provided to the user. You can easily add filtering, grouping, sorting and automatically highlight matches in the search results.

SearchBlackBox SDK is written in C#. It is fully optimized for use in .NET applications and is natively integrated as a .NET Framework Class Library. Thus, the SDK can be used in all kinds of .NET applications, from Windows Forms GUI projects to WEB services.

SearchBlackBox SDK has all the features you need to quickly add a search engine to your applications without the need to study the subject or to write some extra code. It is both extremely easy-to-use and to deploy. With an open architecture and a reasonable price, this is one of the most competitive solutions on the market today.

Full text search

In text retrieval, full text search (also called free search text refers to a technique for searching a computer-stored document or database; in a full text search, the search engine examines all of the words in every stored document as it tries to match search words supplied by the user.

Some Web search engines, such as AltaVista employ full text search techniques, while others index only a portion of the Web pages examined by its indexing system.

When dealing with a small number of documents it is possible for the full-text search engine to directly scan the contents of the documents with each query, a strategy called **serial scanning**. This is what some rudimentary tools, such as grep (in linux), do when searching.

Full Text_ Precision versus Tradeoff:

Due to the ambiguities of ***natural language***, a full text search typically produces a retrieval list that has low precision: most of the items retrieved are irrelevant. ***Controlled-vocabulary*** searching solves this problem by ***tagging*** the documents in such a way that the ambiguities are eliminated. However, a controlled vocabulary search may have low recall: it may fail to retrieve some documents that are actually relevant to the search question. Despite the presence of many irrelevant documents in a free text search's retrieval list, a free text search may be able to locate a document that a controlled vocabulary search failed to retrieve.

Available data types for full text search:

The full-text indexes can be created on the following SQL Server character-based data types:

- ☐ char
- ☐ nchar
- ☐ varchar
- ☐ nvarchar
- ☐ text
- ☐ Ntext

Full text Index versus regular SQL server Index:

Full Text indexes	Regular SQL server indexes
Stored in the file system but administered through database.	Stored under the control of the database in which they are defined.
Only one full text index are allowed per table.	Several regular indexes allowed per table.
Addition of data to full-text indexes, called population, can be requested through either a schedule or a specific request, or can occur automatically with the addition of new data.	Updated automatically when the data upon which they are based is inserted, updated, or deleted.
Grouped within the same database into one or more full-text catalogs.	Not grouped.

SQL-SERVER: Full Text Indexing

Microsoft SQL Server supports T-SQL, an implementation of ANSI standard SQL. T-SQL is designed to (among other things) search for matches in your data. For example, if you've created a table with a column named Notes you could construct these queries:

```
SELECT * FROM MyTable WHERE Notes = 'Deliver Tuesday'
SELECT * FROM MyTable WHERE Notes LIKE '%caution%'
```

But what if you're not looking for an exact match, either to the full text of the column or a part of the column? That's when you need to go beyond the standard SQL predicates and use SQL Server's full-text search capabilities. With full-text searching, you can perform many other types of search:
Two words near each other
Any word derived from a particular root (for example run, ran, or running)
Multiple words with distinct weightings
A word or phrase close to the search word or phrase

In this article, I'll show you how to set up and use full-text searching in SQL Server 2000, and give you a sneak peek of the changes that are coming in this area when SQL Server 2005 ships next year.

Setting Up SQL Server 2005 Full-Text Searching

If you want to play with SQL Server 2005 full-text search capabilities, you need to do the following:

1) Once a database has been created, support for full-text must be enabled before any full-text operations can be performed. To enable full-text support the sp_fulltext_database stored procedure is used. This stored procedure updates the currently selected database, so be sure to USE the correct database before issuing this statement:

```
EXEC sp_fulltext_database 'enable';
```

If you are using SQL Server Management Studio and use the interactive New Database dialog to create your database, you can check the "Use full-text indexing" checkbox, which causes the above mentioned stored procedure to be automatically executed.

2) SQL Server stores full-text data in a catalog (a file that needs to be created). A single catalog can be used for multiple tables and indexes. If a catalog does not already exist, create one using CREATE FULLTEXT CATALOG:

```
CREATE FULLTEXT CATALOG my_catalog;
```

This creates a catalog named my_catalog in the default catalog location. To specify the actual file location the IN PATH attribute can be specified.

3) Once a catalog has been created you can define the actual full-text indexes for each table containing columns that you want searchable. Indexes are created using CREATE FULLTEXT INDEX like this:

CREATE FULLTEXT INDEX ON table(column)KEY INDEX table_primary_keyON my_catalog;

When creating a full-text index you specify the table and column to be indexed as table(column). More than one column may be indexed if needed, to do this simply specify the column names (comma delimited). The key with which to uniquely identify rows is required, and so KEY INDEX is used to provide the name of the table's primary key. And finally, the ON clause specifies the catalog to be used to store full-text data, and here the just created catalog is used (unless a default catalog has been defined).

Once set up, you'll be ready to use the CONTAINS and FREETEXT functions in your WHERE clauses to perform full-text searching.

Performing SQL Server FREETEXT Searches

FREETEXT provides a simple mechanism by which to perform SQL Server 2005 full-text searches, matching by meaning as opposed to exact text match.
Here is a simple example:

SELECT *FROM my_table WHERE FREETEXT (column1, 'rabbit food');

FREETEXT (column1, 'rabbit food') means perform a FREETEXT lookup on column column1 looking for anything that could mean rabbit food (but not necessarily those two exact words, and not necessarily as a phrase).

You can also search across all columns indexed for full-text search by using FREETEXT (*, 'search text').

If double quotes surround a search term then that exact phrase is matched, not the meaning.

Performing SQL Server CONTAINS Searches
SQL Server 2005 supports two forms of full-text search, FREETEXT and CONTAINS. CONTAINS is used to search for rows that contain words, phrases, partial phrases, words with the same stem, proximity searches, synonyms (using a thesaurus lookup), and more.
Here is a simple example:

SELECT note_id, note_textFROM productnotes WHERE CONTAINS(note_text, 'handsaw');

WHERE CONTAINS(note_text, 'handsaw') means find the word handsaw in column note_text. CONTAINS also supports the use of wildcards:

SELECT note_id, note_textFROM productnotesWHERE CONTAINS(note_text, '"anvil*"');

'"anvil*"' means match any word that starts with anvil. Note that unlike LIKE, full-text searching uses * as the wildcard character (instead of %). Wildcards may be used at the beginning or end of a string. Also, when passing simple text to CONTAINS then that text is enclosed within single quotes. When passing wildcards each search phrase must be enclosed within double quotes inside those outer single quotes. Failing to do this will likely cause your searches to return no matches.

CONTAINS also supports Boolean operators AND, OR, and NOT. Here are a couple of examples:

SELECT note_id, note_text FROM productnotes WHERE CONTAINS(note_text, 'safe AND handsaw');
SELECT note_id, note_textFROM productnotesWHERE CONTAINS(note_text, 'rabbit AND NOT food');

When searching through extremely long text there is a greater likelihood of matches being found if search terms are near each other in the saved data? A simple AND search matches terms anywhere in the text, but NEAR can be used to instruct the full-text search engine to only match terms when they are close together.
Here is an example:

SELECT note_id, note_textFROM productnotesWHERE CONTAINS(note_text, 'detonate NEAR quickly');

'detonate NEAR quickly' means match only rows that contain the words detonate and quickly near each other.

Sometimes you may want to match a word that is part of the same family (based on the same stem). For example, if you were searching for "life" you'd also want to match "lives". Obviously, a wildcard of life* could not help here, and using li* would likely match too many false positives. This is where inflectional matching helps.
Here is an example:
SELECT note_id, note_textFROM productnotesWHERE CONTAINS(note_text, 'FORMSOF(INFLECTIONAL, life)');

'FORMSOF(INFLECTIONAL, life)' instructs the full-text engine to look for any words that share the same stem as the specified word, in this case "life".
FORMSOF() also supports THESAURUS searches, where words can match synonyms. To use this functionality you must first populate an XML thesaurus file with words and their synonyms.

Ranking SQL Server 2005 Full-Text Search Results

When performing full-text searches you usually want not just results, but a ranking indicating how close a match is to what you are looking for. In SQL Server 2005, ranks are accessed via

ranking functions - FULLTEXT searches are ranked using function FULLTEXTTABLE() and CONTAINS searches are ranked using function CONTAINSTABLE(). Both of these functions are used the same way, and both accept search patterns, the same search patterns supported by the FULLTEXT and CONTAINS predicates themselves.
Here is an example:

SELECT f.rank, note_id, note_text FROM productnotes,FREETEXTTABLE(productnotes, note_text, 'rabbit food') fWHERE productnotes.note_id=f.[key]ORDER BY rank DESC;

This example performs a FREETEXT type search. Instead of filtering using the WHERE clause, the FREETEXTTABLE() function is used and given a search pattern instructing the full-text engine to match any rows that contain words meaning rabbit and food. FREETEXTTABLE() returns a table which is given an alias of "f" (so as to be able to refer to it in column selection and the join), this table contains a column named "key" which will contain the primary key value of the table that was indexed (productnotes in this example), and "rank" which is the rank value assigned. And finally, results here are sorted by rank descending, as the higher the rank the greater the match.

It is also possible to assign weight values to search patterns and words. The rankings assigned in the example used here assumed that all words were equally important and relevant. If this is not the case, and some words are more important than others, then the ISABOUT() function can be used to assign relative weights, and the full-text search engine will then use these values when determining rankings.

Classes for .net and Java for developing a search engine

Both the platform works with the similar classes for developing search engine.
IndexReader, IndexWriter, IndexSearcher, IndexResult , QueryParser, SearchDocument and StandardAnalyzer are default classes for search engine development. If we need to add additional features with the search engine developments then we have to implement other classes to get the desired output.

IndexReader:

IndexReader is an abstract class, providing an interface for accessing an index. Search of an index is done entirely through this abstract interface, so that any subclass which implements it is searchable.
Concrete subclasses of IndexReader are usually constructed with a call to one of the static open() methods, e.g. open(String).
For efficiency, in this API documents are often referred to via *document numbers*, non-negative integers which each name a unique document in the index. These document numbers are ephemeral--they may change as documents are added to and deleted from an index. Clients should thus not rely on a given document having the same number between sessions.
An IndexReader can be opened on a directory for which an IndexWriter is opened already, but it cannot be used to delete documents from the index then.

For backwards API compatibility, several methods are not listed as abstract, but have no useful implementations in this base class and instead always throw UnsupportedOperationException. Subclasses are strongly encouraged to override these methods, but in many cases may not need to.

IndexSearcher:

Implements search over a single IndexReader.

Applications usually need only call the inherited `Searcher.search(Query)` or **`Searcher.search(Query,Filter)`** methods. For performance reasons it is recommended to open only one IndexSearcher and use it for all of your searches.

Note that you can only access Hits from an IndexSearcher as long as it is not yet closed, otherwise an IOException will be thrown.

IndexWriter:

An IndexWriter creates and maintains an index.
The create argument to the **constructor** determines whether a new index is created, or whether an existing index is opened. Note that you can open an index with create=true even while readers are using the index. The old readers will continue to search the "point in time" snapshot they had opened, and won't see the newly created index until they re-open. There are also **constructors** with no create argument which will create a new index if there is not already an index at the provided path and otherwise open the existing index.
In either case, documents are added with **addDocument** and removed with **deleteDocuments** or **deleteDocuments**. A document can be updated with **updateDocument** (which just deletes and then adds the entire document). When finished adding, deleting and updating documents, **close** should be called.

An overview on SearchResult and SearchDocument class under .NET platform based on C#

System.Object
  SearchBlackBox.SDK.SearchResult

```
[C#]
public sealed class SearchResult :

ICustomTypeDescriptor
```

Thread Safety
Public static (**Shared** in Visual Basic) members of this type are safe for multithreaded operations. Instance members are **not** guaranteed to be thread-safe.
Remarks
A **SearchResult** is a SearchDocument and additional information about the document as a search result, particularly:
Score - the score (rank) of the result
Index - the number of the result among other results
Highlight - highlighted text (only if HighlightingSettings are specified for SearchResults)
You can use a **SearchResult** to get the document itself as well as the values of its fields.
Requirements
Namespace: SearchBlackBox.SDK
Assembly: SearchBlackBox.SDK (in SearchBlackBox.SDK.dll)

Represents a search document.
For a list of all members of this type, see SearchDocument Members
System.Object
SearchBlackBox.SDK.SearchDocument

```
[C#]
public sealed class SearchDocument :IEnumerable
```

Thread Safety
Remarks
A `SearchDocument` is a set of the DocumentFields
Each field has a name, a text value and the attributes
A field may be stored with the document. In this case it is returned with search hits on the document. Thus, each document should typically contain stored fields which uniquely identify it.
Example
The following example creates a SearchDocument by FileInfo

```
SearchDocument CreateSearchDocument(FileInfo fileInfo)
{
  SearchDocument doc = new SearchDocument();
    doc.AddField("path", fileInfo.FullName,
                 FieldAttributes.Stored | FieldAttributes.Indexed);
    doc.AddField("name", fileInfo.Name,
                 FieldAttributes.Stored | FieldAttributes.Indexed);
    doc.AddField("length", fileInfo.Length.ToString(),
                 FieldAttributes.Stored | FieldAttributes.Indexed);
    return doc;
}
```

Requirements
Namespace: SearchBlackBox.SDK
Assembly: SearchBlackBox.SDK (in SearchBlackBox.SDK.dll)

FieldAttributes Enumeration
Provides attributes for the DocumentField
This enumeration has a FlagsAttribute attribute that allows a bitwise combination of its member values.

[C#]
public enum FieldAttributes
Remarks
Attributes define the method for processing values.
The following combinations are the most frequently used ones:
Indexed | Stored - Usually for fields whose original value should be preserved. For example, URL, email, date and other non-text fields.

Indexed | Analyzed | Stored - Useful for short text fields, like "title" or "subject".
Indexed | Analyzed - Useful for long text fields, like "body".
Stored - The value will be stored in the index and returned with hits. Useful for additional information that should be shown for search results.

Members

Member Name	Description	Value
Indexed	The value of the field will be indexed and available for searching. A field must have this attribute if you need to search this field	1
Analyzed	The value of the field will be analyzed. Unanalyzed fields are indexed as a single word.	2
Stored	The value of the field will be stored in the index and can be returned with hits on the document.	4
IndexedAndAnalyzed	Indexed and Analyzed	3
IndexedAndStored	Indexed and Stored	5
IndexedAndAnalyzedAndStored	Indexed and Analyzed and Stored	7
AnalyzedAndStored	Analyzed and Stored	6

Requirements
Namespace: SearchBlackBox.SDK
Assembly: SearchBlackBox.SDK (in SearchBlackBox.SDK.dll)

HighlightingSettings Class
Specifies information about how a search result is highlighted.
For a list of all members of this type, see HighlightingSettings Members.
System.Object
  SearchBlackBox.SDK.HighlightingSettings

[C#]
```
public sealed class HighlightingSettings
```
Thread Safety
Public static (Shared in Visual Basic) members of this type are safe for multithreaded operations. Instance members are not guaranteed to be thread-safe.
Example
This example shows how to specify HighlightingSettings for highlighting the *entire text* in the HTML format. The source text is taken from the "name" field of the document.
```
HighlightingSettings settings = new HighlightingSettings("name","<b>","</b>");
```
This example shows how to specify HighlightingSettings for highlighting the results for the *most relevant fragments* from the text in the HTML format. The source text is taken from the "name" field of the document.
```
HighlightingSettings settings = new HighlightingSettings("name","<b>","</b>",3);
```
Requirements
Namespace: SearchBlackBox.SDK
Assembly: SearchBlackBox.SDK (in SearchBlackBox.SDK.dll)

Analysis

We have to analyze between two platforms Java and .NET on the basis of available tools and features for choosing the best platform for developing a search engine.

Apache lucene –query parser syntax:
Although Lucene provides the ability to create your own queries through its API, it also provides a rich query language through the Query Parser, a lexer which interprets a string into a Lucene Query using JavaCC.

Dotlucene features for .net platform
Dotlucene features for .net platform are given below

- ☐ Very good performance
- ☐ Ranked **search** results
- ☐ Search query highlighting in results
- ☐ Searches structured and unstructured data
- ☐ Metadata searching (query by date, search custom fields…)
- ☐ Index size approximately 30% of the indexed text
- ☐ Can store also full indexed documents
- ☐ Pure managed **.NET** in a single assembly
- ☐ Very friendly licensing (Apache Software License 2.0)
- ☐ Localizable (support for Brazilian, Czech, Chinese, Dutch, English, French, German, Japanese, Korean and Russian included in DotLucene National Language Support Pack)
- ☐ Extensible (source code included)
- ☐ DotLucene is a port of **Jakarta Lucene** to **.NET** (**C#**) maintained by George Aroush
- ☐ Index is compatible with the **Java** version (**Lucene**)

Dotlucene platform for creating index:
Steps required for creating index under dotlucene

- ☐ Pre-generated.
- ☐ Read-only (locks disabled).
- ☐ Compound format.
- ☐ There are 1981 HTML documents indexed.
- ☐ The documents have 5.4 MB size in total.
- ☐ The index doesn't contain the full original files.
- ☐ The index size is 513 kB.
- ☐ Documents contain three fields: title, text and path

Dotnet adding document to index:
We have to add documents to provide searching operation after building the index.

- ☐ Once you have opened the index you can start adding documents.

- ☐ Every object you want to insert into the index is a Document.
- ☐ Each document in index has fields that contain some information about it.
- ☐ For each of the fields you need to specify what the indexer should do with it.
- ☐ That means whether the field content should be:

- ☐ Stored in the index (you will be able to get any stored value from index any time later;
- ☐ This is useful for short fields, like author, title, etc.)
- ☐ Indexed (indexing is required for all fields that you want to query)
- ☐ Tokenized (i.e. split into words before indexing)

Dotlucene file parsing:

Supported document formats for file parsing under dotlucene.

- ☐ Lucene.NET can index only pure text. If you need to index PPT, DOC, RTF, PDF, HTML, XLS or whatever, you need to parse it and extract the text first.

- ☐ Here are some common document formats:

* Microsoft Office documents (XLS, DOC, PPT)
* HTML files
* RTF files
* PDF files

SearchResult class for dotnet platform:
Represents a result of the search
For a list of all members of this type, see SearchResult Members

Java platform for search engine:

- ☐ IndexFiles class creates a Lucene Index.
- ☐ The IndexWriter is the main class responsible for creating indices.

Indexing documents java:

- ☐ The recursive function indexDocs() code simply crawls the directories and uses FileDocument to create Document objects. The Document is simply a data object to represent the content in the file as well as its creation time and location. These instances are added to the indexWriter.

Java tools for developing search engine:

Apache lucene 2.3.1 is used to develop search engine for BRAC University web site.

- **Apache lucene 2.3.1**
- Apache lucene 2.2.8 (intranet web server application)
- Apache cocoon 2.0.4(based XML, XSLT technologies) and so on.

Java Parsing:

The particular Analyzer we are using, StandardAnalyzer, is little more than a standard Java Tokenizer, converting all strings to lowercase and filtering out useless words and characters from the index. By useless words and characters it means common language words such as articles (a, an, the, etc.) and other strings that would be useless for searching (e.g. 's) .

Java parser features:

- Stream based file parsing allowing for parsing of large files without storing the entire file contents in memory
- Simple reader pattern movement through file
- Simple access to field data by index or by field name
- Internal parsing using arrays instead of strings for fast looping over large files
- Uses built in framework functionality to handle most encoding issues instead of guessing or hardcoding
- Includes a CSV writer to ease the worries while creating CSV data. No more needing to worry about what to replace with what to match up with delimiters, text qualifiers, etc.

Lucene Query Syntax:

Lucene Query syntaxes are given below.

- **Lucene Query Syntax**
- Query examples:
- jazoon
- jazoon AND java <=> +jazoon +java
- jazoon OR java
- jazoon NOT php <=> jazoon -php
- conference AND (java OR j2ee)
- "Java conference"
- title:jazoon
- j?zoon
- jaz*
- schmidt~ schmidt, schmit, schmitt
- price:[000 TO 050]
- + more

Apache lucene java:

Features available under apache lucene for Java platform

- ☐ **Lucene Java**
- ☐ Java library for indexing and searching
- ☐ No dependencies (not even a logging framework)
- ☐ Works with Java 1.4 or later
- ☐ Input for indexing: Document objects
- ☐ Each document: set of Fields, field name: field content (plain text)
- ☐ Input for searching: query strings or Query objects
- ☐ Stores its index as files on disk
- ☐ No web crawler
- ☐ Powerful query syntax
- ☐ Create queries from user input or programmatically
- ☐ Fast indexing
- ☐ Fast searching
- ☐ Sorting by relevance or other fields
- ☐ Large and active community
- ☐ Apache License 2.0

Apache lucene-term modifiers:

Supported term modifiers under apache lucene are given below

- ☐ Wildcard Searches
- ☐ Lucene supports single and multiple character wildcard searches within single terms (not within phrase queries).
- ☐ To perform a single character wildcard search use the "?" symbol.
- ☐ To perform a multiple character wildcard search use the "*" symbol.
- ☐ The single character wildcard search looks for terms that match that with the single character replaced. For example, to search for "text" or "test" you can use the search:
- ☐ te?t Multiple character wildcard searches looks for 0 or more characters. For example, to search for test, tests or tester, you can use the search:
- ☐ test* You can also use the wildcard searches in the middle of a term.
- ☐ te*t Note: You cannot use a * or ? symbol as the first character of a search.

Search parameters:

Supported search parameters for both Java and .NET are given below.

Apache lucene - Range Searches:

- ☐ Range Queries allow one to match documents whose field(s) values are between the lower and upper bound specified by the Range Query. Range

Queries can be inclusive or exclusive of the upper and lower bounds. Sorting is done lexicographically.

- ☐ mod_date:[20020101 TO 20030101] This will find documents whose mod_date fields have values between 20020101 and 20030101, inclusive. Note that Range Queries are not reserved for date fields. You could also use range queries with non-date fields:
- ☐ title:{Aida TO Carmen} This will find all documents whose titles are between Aida and Carmen, but not including Aida and Carmen.
- ☐ Inclusive range queries are denoted by square brackets. Exclusive range queries are denoted by curly brackets.

Apache lucene – Boosting:

- ☐ Lucene provides the relevance level of matching documents based on the terms found. To boost a term use the caret, "^", symbol with a boost factor (a number) at the end of the term you are searching. The higher the boost factor, the more relevant the term will be.
- ☐ Boosting allows you to control the relevance of a document by boosting its term. For example, if you are searching for jakarta apache and you want the term "jakarta" to be more relevant boost it using the ^ symbol along with the boost factor next to the term. You would type: jakarta^4 apache

Apache lucene – Boolean operators:

- ☐ Boolean operators allow terms to be combined through logic operators. Lucene supports AND, "+", OR, NOT and "-" as Boolean operators
- ☐ The OR operator is the default conjunction operator. This means that if there is no Boolean operator between two terms, the OR operator is used. The OR operator links two terms and finds a matching document if either of the terms exist in a document. This is equivalent to a union using sets. The symbol || can be used in place of the word OR.
- ☐ To search for documents that contain either "jakarta apache" or just "jakarta" use the query:

"jakarta apache" jakarta or "jakarta apache" OR jakarta

Apache lucene - Boolean "And":

- The AND operator matches documents where both terms exist anywhere in the text of a single document. This is equivalent to an intersection using sets. The symbol && can be used in place of the word AND.
- To search for documents that contain "jakarta apache" and "Apache Lucene" use the query:
- "jakarta apache" AND "Apache Lucene"

Boolean +:

- The "+" or required operator requires that the term after the "+" symbol exist somewhere in a the field of a single document.
- To search for documents that must contain "jakarta" and may contain "lucene" use the query:
- +jakarta lucene

Boolean Not:

The NOT operator excludes documents that contain the term after NOT. This is equivalent to a difference using sets. The symbol ! can be used in place of the word NOT.

- To search for documents that contain "jakarta apache" but not "Apache Lucene" use the query:
  "jakarta apache" NOT "Apache Lucene"

Grouping:

- Lucene supports using parentheses to group clauses to form sub queries. This can be very useful if you want to control the boolean logic for a query.
- To search for either "jakarta" or "apache" and "website" use the query:
- (jakarta OR apache) AND website

Field Grouping:

- Lucene supports using parentheses to group multiple clauses to a single field.
- To search for a title that contains both the word "return" and the phrase "pink panther" use the query:
- title:(+return +"pink panther")

Escaping special characters:

- Lucene supports escaping special characters that are part of the query syntax. The current list special characters are
- + - && || ! ( ) { } [ ] ^ " ~ * ? : \
- To escape these character use the \ before the character. For example to search for (1+1):2 use the query:
  \(1\+1\)\:2

We have discussed so far about the features and tools available in both the platform. We found that Java supports more functionality and features than .NET. Now we are going to discuss about lucene pitfalls.

Lucene pitfalls:

- Limit to 10.000 tokens by default – IndexWriter.setMaxFieldLength()
- There's no error if a field doesn't exist
- You cannot update single fields
- You cannot "join" tables (Lucene is based on documents, not tables)
- Lucene works on strings only -> 42 is between 1 and 9
  Use "0042"

From user to user platforms might vary for developing search engine. For me the criteria for developing a search engine were Availabilty, Readiness, Depth of tools and features.

1. **Helps are more available and readily found:**

Java posses a huge community and the members are more active than in the dotlucene.
Tools are easily found and number of available tools is greater in Java than .NET.
Java tools support lots features than .NET.

2. **Java more Flexible and Open:**

Java is more flexible as it has greater number of features than .NET. It is open and more readily found to be used for developing search engine. More function supporting features of Java provide additional flexibility to developers.

3. **Java tools are more available:**

Java tools are more available than .NET and tools are usually free. You do not need to pay for using their tools and it is pretty easy to be a part of an active community for sharing or collecting information about developing search engine.

4. **Easy to find:**
There are huge number of websites are available for developing search engine.
All tools are free and better than .NET tools on the basis of supporting features and functions.

On the basis of the criteria, Java found to be a better option for developing search engine at current time.

Conclusion:

Our world is becoming a huge source of data. And from this ever increasing data source, search engine helps us to retrieve necessary information. Modern search engines simply bring information from a massive data source. We still need faster and more precise searching techniques. The search engines those are available are not quite user friendly as well. We get lots of false positive unwanted result which waste user time. We have to develop search engine on the basis of correctness, accuracy and faster response to user.