Lucene Search Engine

Important: The Sitecore.Data.Indexing API will be deprecated in Sitecore CMS 6.5 and in 7.0 it will be completely removed.

Author: Sitecore Corporation

Date: Thursday, 09 September 2009

Release: Rev. 1.0

Language: English

Table of Contents

Chapter 1

Lucene.Net in Sitecore

This article describes the implementation of Lucene.Net open-source search engine in the Sitecore CMS. For more information about Lucene.Net, follow these links:

1. Overview
2. API documentation
3. Bug tracking system
4. Query syntax
5. Scoring

Sitecore CMS implements a wrapper for the Lucene engine. The wrapper has its own API and the original API is accessible as well.

Lucene builds its own indexes by scanning the Sitecore items. The following list describes some issues which should be taken into account when implementing Lucene search.

1. Index for the web database is not set up by default. For information about index setup for the web database, see Section
2. If too much data is added into the indexes they can occupy a lot of additional space.
3. Updating of indexes adds some load on the CPU.

Important Note:

The Sitecore.Data.Indexing API, widely referred to in this document, will be deprecated in Sitecore CMS 6.5 and in 7.0 it will be completely removed.

In future developers should use the Sitecore.Search API when configuring Sitecore Search or Lucene search indexes.

For more information on the Sitecore.Search API see the Sitecore Search and Indexing document on the Sitecore Developers Network (SDN).

Chapter 2

Index configuration

Indexes are defined in the <indexes> section of the web.config file. For example, take a look at the "system" index

```
      <index id="system" singleInstance="true"
type="Sitecore.Data.Indexing.Index, Sitecore.Kernel">
        <param desc="name">$(id)</param>
        <fields hint="raw:AddField">
          <field target="created">__created</field>
          <field target="updated">__updated</field>
          <field target="author">__updated by</field>
          <field target="published">__published</field>
          <field target="name">@name</field>
          <field storage="unstored">@name</field>
          <field target="template">@tid</field>
          <field target="id" storage="unstored">@id</field>
          <type storage="unstored">memo</type>
          <type storage="unstored">text</type>
          <type storage="unstored" stripTags="true">html</type>
          <type storage="unstored" stripTags="true">rich text</type>
        </fields>
      </index>
```

The id attribute is used to distinguish different indexes.

The **fields** definition contains the index description. The definition can contain the tags **field** and **type**. The fields in the <fields> tag are parsed and indexed by the AddFields method. For information about the modification of this method, see **Section 2.3, Implementing custom indexing**.

The **field** tag defines how to transform item field values into the fields of index. The **target** attribute of this tag defines the field name in the index, where the indexed information will be stored. The value of the tag defines the field in an item or the property of an item which will be indexed. If the value begins with the "@" character, then the item's property should be get. Valid property names are @id, @lang, @mid (master ID), @name, @tid (template ID).

The **type** tag's value specifies the Sitecore field type. The indexes for all fields of this type will be stored in one and the same field.

The field and the type tags can have the following attributes:

- **target** – defines the field in the index where a value should be stored, if a tag does not have this attribute, the indexed information will be stored in the "_content" field of index.
- **stripTags** – defines whether html tags should be removed from the field value before indexing. Useful for html and rich text fields.

- **storage** – can take on the following values:

Value	Indexed value (can be used for search by Lucene)	Stored value (value can be retrieved from the index)	Notes
text (default)	+	+	The data is stored, indexed, and tokenized – good for searchable small text (Title).
keyword	+	+	The data is stored and indexed but not tokenized - good for last modified date, filename.
unstored	+	-	The data is not stored but it is indexed and tokenized. Good for big HTML and Memo fields.
unindexed	-	+	The data is stored but not indexed or tokenized. Good for URLs.

2.1 Configuring database indexes

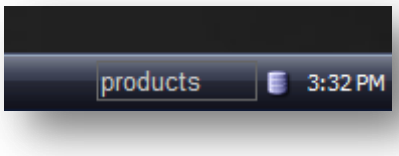
Each database can have an index which is defined by the <indexes> tag within the database definition in web.config. For example, the following lines define that the index called "system" will be used for the core database:

```
<database id="core" singleInstance="true" type="Sitecore.Data.Database,
Sitecore.Kernel">
...
        <indexes hint="list:AddIndex">
             <index path="indexes/index[@id='system']" />
        </indexes>
...
</database>
```

It is possible to define multiple indexes for a database. Just note that the "system" index is needed if you want to search in the database from the back-end:

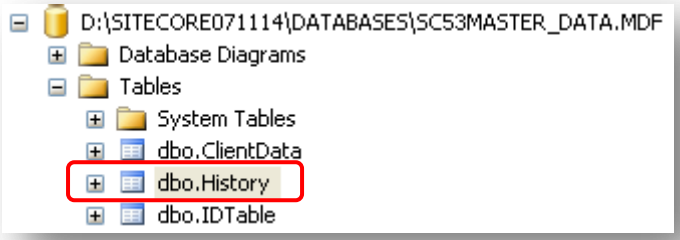
```
        <indexes hint="list:AddIndex">
             <index path="indexes/index[@id='system']" />
             <index path="indexes/index[@id='YourIndexID']"/>
        </indexes>
```

Users can search among the Sitecore items (not only the site items, but also optionally templates and systems items) from Sitecore back-end using the built in Search application, which can be accessed by selecting **Sitecore » Search** or by typing a search string in the search bar and pressing Enter:



2.2 Updating indexes

Lucene indexes are updated right after any entry is added to the History table of the appropriate database. Any modification made to items causes an addition of a record to History table.



As a result, the IndexingManager.HistoryEngine_AddedEntry event is fired and the UpdateIndexAsync job is started, which in turn updates the Lucene indexes. The job indexes only newly added items or changed items. The History table is empty when an index for a database is created for the first time, so it is important to rebuild the indexes (manually or programmatically) before letting the system to update indexes on its own. The Properties table has a record with the key "IndexingProvider_LastUpdate". This record defines the date, which is used as the start point for item indexing (all items which were modified before this date will not be updated by the UpdateIndexAsync job, this is why it is important to rebuild the indexes after they have been defined for a database).

Sitecore also updates Lucene indexes on a regular basis according to the following web.config parameter:

```
<setting name="Indexing.UpdateInterval" value="00:05:00" />
```

If an item has not been indexed after a modification for some reason, then it will be indexed on a regular basis (in our example, within the next five minutes). This mechanism is implemented for safety purposes – if the index update process breaks, the appropriate items will be indexed within the "Indexing.UpdateInterval".

So, when is the History table updated?

It is updated by the Engines.HistoryEngine.Storage engine, which is defined for every database, where indexing is required. For example, this is how this engine defined for the master database:

```
<!-- master -->
<database id="master" singleInstance="true" type="Sitecore.Data.Database,
Sitecore.Kernel">
...
<Engines.HistoryEngine.Storage>
      <obj type="Sitecore.Data.$(database).$(database)HistoryStorage,
Sitecore.$(database)">
          <param desc="connection" ref="connections/$(id)">
          </param>
          <EntryLifeTime>30.00:00:00</EntryLifeTime>
      </obj>
</Engines.HistoryEngine.Storage>
<Engines.HistoryEngine.SaveDotNetCallStack>false</Engines.HistoryEngine.SaveDotN
etCallStack>
...
</database>
```

The **SqlServerHistoryStorage** class adds the history of changing, adding and deleting items to table **_CURENT_DATABASE_ » History**. Each addition starts the index update job, and another job runs in a specified interval to check if there is something left to index.

The process described in this section updates the indexes automatically, but you can also rebuild the indexes manually by selecting **Sitecore » Control Panel » Database » Rebuild the Search Index**.

It is also possible to index only items based on a give template. This is done by adding the <templates hint="list:AddTemplate"> definition to an index. For instance, the <index> definition for the archive database contains the <templates> definition:

```
<index id="archive" singleInstance="true" type="Sitecore.Data.Indexing.Index,
Sitecore.Kernel">
        <param desc="name">$(id)</param>
        <templates hint="list:AddTemplate">
          <!-- Archived item template -->
          <template>{BF2B8DA2-3CBA-485D-8F85-3788B8AFBDBF}</template>
        </templates>
        <fields hint="raw:AddField">
          <field target="name">@name</field>
...
```

2.3 Implementing custom indexing

If you want to implement custom indexing (for example, index PDF or MS Word files), you should write your own class and derive it from Sitecore.Data.Indexing.Index. You should override the AddFields method, as long as item parsing is performed in this method.

The following example shows how to implement PDF file indexing. This example uses four third party dlls (PDFBox, IKVM.GNU.Classpath, IKVM.Runtime, FontBox-0.1.0-dev). For appropriate download links, refer to **Chapter 5, Downloads**.

```
using System;
using System.Collections.Generic;
using System.Text;
using Sitecore.Data.Items;
using Lucene.Net.Documents;
using System.IO;
using org.pdfbox.pdmodel;
using org.pdfbox.util;

namespace Sitecore.PdfIndexer
{
    public class PdfIndex : Sitecore.Data.Indexing.Index
    {
        public PdfIndex(string indexName)
            : base(indexName)
        {
        }
        protected override void AddFields(Item item, Document document)
        {
            document.Add(Field.Text("_scIsMedia", item.Paths.IsMediaItem ? "1" :
"0"));
            if (item.Paths.IsMediaItem)
            {
                MediaItem mediaItem = (MediaItem)item;
                if (mediaItem != null && mediaItem.Extension == "pdf")
                {
                    string mediaContent = ParsePDF(mediaItem);
                    document.Add(Field.Text("_content", item.Name, true));
                    document.Add(Field.Text("_content", mediaContent, true));
```

```
                }
            }
            else
            {
                base.AddFields(item, document);
            }
        }
        private string ParsePDF(MediaItem mediaItem)
        {
            Stream stream = mediaItem.GetMediaStream();
            ikvm.io.InputStreamWrapper wrapper = new
ikvm.io.InputStreamWrapper(stream);
            PDDocument doc = PDDocument.load(wrapper);
            PDFTextStripper stripper = new PDFTextStripper();
            return stripper.getText(doc);
        }
    }
}
```

When the class is written, add your index in the web.config file:

```
<index id="myindex" singleInstance="true"
type="_your_custom_namespace_._CustomIndex_, _custom_library_">
              <param desc="name">$(id)</param>
              <fields hint="raw:AddField">
                  ...
              </fields>
</index>
```

This code is also available for downloading among other code examples. For download links to this code, please refer to **Chapter 5, Downloads**.

Chapter 3

Website search implementing

To implement the search you should perform the following steps:

1. Create a search form.
2. Parse user's input and create Lucene query. (For information about Lucene query syntax, please refer to the Query Parser Syntax article).
3. Execute search using Lucene API and query from the previous step.
4. Format Search results.

3.1 Creating a search form and a codebehind class

The Downloads section of this document provides links to the following files.

- Searchform.ascx
  This file provides a simple search form including a calendar which allows to search within a given time span.
- Search.ascx.cs
  This file provides the codebehind class which implements the search logic including the ranged search and simple search output rendering.
- PdfIndex.cs
  The code for implementing PDF indexing.

For download links, please refer to **Chapter 5, Downloads**.

3.2 Lucene search in staged environments

Lucene can work on the slave server in a staged setup with multiple web farm servers if slave web farm servers share the same production (Web) database.

In such setup, set the Indexing.ServerSpecificProperties parameter to "true" in web.config:

```
<!--  INDEX PROPERTIES PER SERVER
      Indicates if server specific keys should be used for property values (such
as 'last updated').
      Default value: false
-->
<setting name="Indexing.ServerSpecificProperties" value="true" />
```

Then follow these steps:

1. Add these entries on the Master server for production (web) database which is shared between the Slave and the Master servers:

```
    <!-- Production -->
      <database id="production" singleInstance="true"
type="Sitecore.Data.Database, Sitecore.Kernel">
        <param desc="name">$(id)</param>
        <securityEnabled>true</securityEnabled>
        <dataProviders hint="list:AddDataProvider">
          <dataProvider ref="dataProviders/main" param1="$(id)">
            <disableGroup>publishing</disableGroup>
            <prefetch hint="raw:AddPrefetch">
              <sc.include file="/App_Config/Prefetch/Common.config" />
              <sc.include file="/App_Config/Prefetch/Web.config" />
            </prefetch>
          </dataProvider>
        </dataProviders>
        <proxiesEnabled>false</proxiesEnabled>
        <proxyDataProvider ref="proxyDataProviders/main" param1="$(id)"
/>
        <!-- Add for Lucene -->
        <Engines.HistoryEngine.Storage>
          <obj type="Sitecore.Data.$(database).$(database)HistoryStorage,
Sitecore.$(database)">
            <param desc="connection" ref="connections/$(id)">
            </param>
            <EntryLifeTime>30.00:00:00</EntryLifeTime>
          </obj>
        </Engines.HistoryEngine.Storage>

<Engines.HistoryEngine.SaveDotNetCallStack>false</Engines.HistoryEngine.S
aveDotNetCallStack>
        <!-- End Lucene -->
        <cacheSizes hint="setting">
```

2. Add these entries for the same production database on the Slave server:

```
      <!-- Production -->
      <database id="production" singleInstance="true"
type="Sitecore.Data.Database, Sitecore.Kernel">
        <param desc="name">$(id)</param>
        <securityEnabled>true</securityEnabled>
        <dataProviders hint="list:AddDataProvider">
          <dataProvider ref="dataProviders/main" param1="$(id)">
            <disableGroup>publishing</disableGroup>
            <prefetch hint="raw:AddPrefetch">
              <sc.include file="/App_Config/Prefetch/Common.config" />
              <sc.include file="/App_Config/Prefetch/Web.config" />
            </prefetch>
          </dataProvider>
        </dataProviders>
        <proxiesEnabled>false</proxiesEnabled>
        <proxyDataProvider ref="proxyDataProviders/main" param1="$(id)"
/>
         <!-- Add for Lucene -->
         <indexes hint="list:AddIndex">
           <index path="indexes/index[@id='system']" />
         </indexes>
         <Engines.HistoryEngine.Storage>
           <obj
```

```
type="Sitecore.Data.$(database).$(database)HistoryStorage,
Sitecore.$(database)">
              <param desc="connection" ref="connections/$(id)">
              </param>
              <EntryLifeTime>30.00:00:00</EntryLifeTime>
            </obj>
          </Engines.HistoryEngine.Storage>

<Engines.HistoryEngine.SaveDotNetCallStack>false</Engines.HistoryEngine.S
aveDotNetCallStack>
          <!-- End Lucene -->
```

3. Make the full publishing from Master to Slave server. This will update the production (web) database that is shared between the Master and the Slave servers.

4. On the Slave server create the Lucene index for the production (web) database via **Control Panel » Databases » Rebuild Search Indexes**. This will create the index for the production database.
   In case the back-end is not available on the slave server, you can use the following snippet to rebuild the indexes:

```
Database database = Factory.GetDatabase("web");
if (database != null)
   {
   for (int i = 0; i < database.Indexes.Count; i++)
      {
      database.Indexes[i].Rebuild(database);
      }
   }
```

Chapter 4

Troubleshooting

You may find yourself in a situation when you try to find some information and you are sure that this information should be there, but the search doesn't return any results. In this case you should perform the following steps:

1. Make sure that this information is really available. If you use two databases (master and web) make sure that this information is available in the appropriate database.
2. Check that you use the appropriate index for searching.
3. Check that a needed index is attached to the database.
4. If you use your own indexing class, make sure that the data is put and searched in the same index field.
5. Rebuild indexes for databases.

Chapter 5

Downloads

The following files are available for downloading.

- Search form and codebehind for CMS 5.3.zip and Search form and codebehind for CMS 6.zip
  This file provides a simple search form including a calendar which allows to search within a given time span and the codebehind class which implements the search logic including the ranged search (not relevant for CMS 6) and simple search output rendering.
- PdfIndex.zip
  The code for implementing PDF indexing.
- PDF indexing libraries.zip
  The libraries required for PDF indexing im plementation.

  Note
  Sometimes retrieving text from a PDF file can fail. If you get an Object reference exception in the PDFBox library, this may be caused by incompatible formats of the PDF file. In this case, you can search the web for a newer version of the PDFBox library (or for one that will allow you to get text from the file) and then use it in the sample described in chapter 2.3 of this document.