Sitecore CMS 7.0 or later

Sitecore Search and Indexing Guide

A Guide for Developers and Administrators

Table of Contents

Chapter 1

Introduction

This guide is for Sitecore partners and developers who want to implement search functionality in Sitecore CMS.

- Overview of Search and Indexing

- Maintaining Indexes in Sitecore

1.1   Overview

Lucene is an open source search engine used in Sitecore CMS for indexing and searching the contents of a Web site. Sitecore CMS 7.0 uses Lucene 3.0.3. Large, distributed Sitecore installations which require increased search performance can implement the SOLR search module. For more information, see the *Sitecore Search Scaling Guide*.

Sitecore implements a wrapper for the Lucene engine which has its own API. The original API (`Lucene.Net`) and the Sitecore API (`Sitecore.ContentSearch`) are both accessible to developers that want to extend their indexing and search capabilities.

However, before you start to use *Lucene.Net or* the *Sitecore.Search* API, it is important to understand some key concepts.

Important Note
The `Sitecore.Data.Indexing` API was deprecated in Sitecore CMS 6.5. The `Sitecore.Search` API works with Sitecore CMS 6 and 7, but is not recommended for new development in version 7.0. Developers should use the `Sitecore.ContentSearch` API when using Sitecore Search or Lucene search indexes in Sitecore 7.0.

Sitecore Search is already implemented in a number of different ways in the Sitecore Desktop. These features use the older Sitecore.Search API.These features are:

- Content Editor — the search box above the content tree (has the same functionality as Quick Search).

- Quick Search — the search box to the bottom right of the Sitecore Desktop (has the same functionality as Content Editor Search).

- Classic Search — this is available from the Sitecore Start button and in the Navigate tab on the ribbon.

You can also use the `Sitecore.ContentSearch` API to extend these capabilities to create custom search functionality for your web site.

For more information, see the manual *Developer's Guide to Item Buckets and Search*.

1.2 Maintaining Search Indexes in Sitecore

In Sitecore 7 the index maintenance infrastructure has been completely revised. Instead of one global way of maintaining the indexes, there is a more transparent and flexible mechanism of Index Update Strategies. Each strategy attached to an index provides a unique way of hooking up to the Sitecore events to keep your index updated.

For more information, see the section *Index Update Strategies*.

Chapter 2

Configuring Search and Indexing

This section describes the configuration files are used to control search and indexing in Sitecore CMS 7.0

- Configuration files that are shipped with Sitecore CMS 7.0

- Default configuration

2.1 Configuration Files

The following files are shipped with Sitecore CMS 7.0. They are located in the `Website\App-Config\Include` folder.
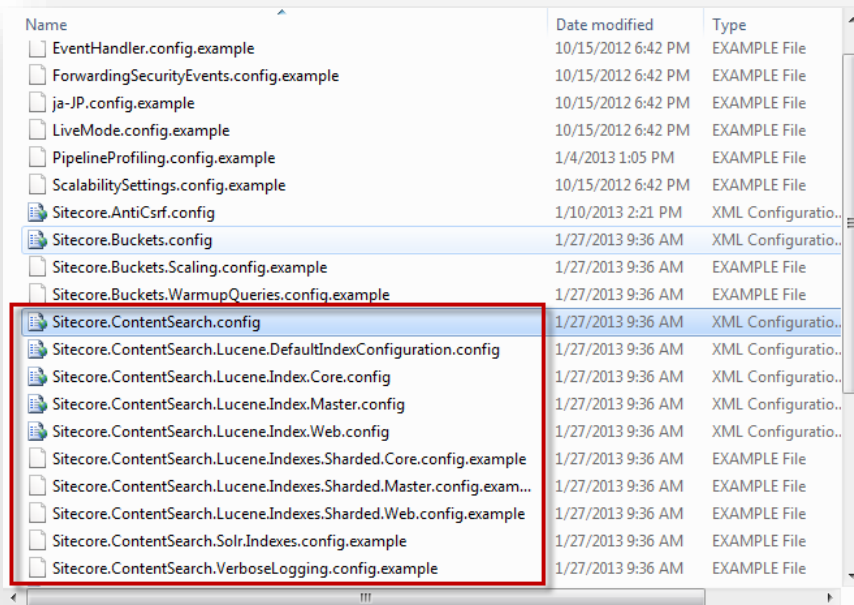
2.1.1 Sitecore.ContentSearch Configuration File

Index Element	Description
`<events>`	Specifies event handlers for indexing:start, indexing:start:remote, indexing:end and indexing:end:remote events.
`<hooks>`	This will run necessary initialization processes, such as EventHub registration, warmup queries for you index. You can extend this class or add additional hooks.
`<pipelines>`	Specifies all pipeline processors related to search and indexing
`<boostingManager>`	Specifies the manager class controlling the search result boosting logic.
`<searchManager>`	Specifies the search manager and provider classes.
`<scheduling>`	Specifies the interval at which indexing will occur.
`<settings>`	Specifies settings such as parallel indexing, maximum search results and date format.
`<commands>`	Specifies handlers for indexing events.

2.2   Default Index Configuration

Index Element	Description
`<indexUpdateStrategies>`	Specifies which update strategy will be used.
`<databasePropertyStore>`	Specifies key values that can be used in search.
`<configuration>`	Settings for index configuration.
`<DefaultIndexConfiguration>`	These settings are used if no custom settings are available for an index.
`<IndexAllFields>`	Default value is true.
`<Analyzer>`	The Lucene Standard Analyzer is specified by default. A custom analyzer may speed searches.
`<fieldMap>`	This allows you to map a Sitecore field name to the index and store it in an appropriate way, including original value storage, boost value, and data type.
`<virtualFieldProcessors>`	Specifies custom query processing for a named field.
`<exclude>`	Exclude items from the index based on template type. Exclude specific fields from the index, if IndexAllFields is set to true.
`<include>`	Include items from the index based on template type, if IndexAllFields is set to false.
`<fields>`	Allows you to format the way that field values are stored in the index, remove inbuilt Sitecore fields and store computed fields.

Chapter 3

Log Files

This chapter describes the log files and logging options that are available for search and indexing in Sitecore 7.0.

- Verbose Logger

- Crawling Log

- Search Log

3.1 Logging Search and Indexing Operations

Sitecore CMS 7.0 provides log files for tracking search and indexing operations. The *Crawling* log tracks information about indexing operations. The *Searching* log shows queries that were generated and run.

3.1.1 Verbose Logger

In order to facilitate the transparency of the indexing mechanism, in addition to standard logging embedded within various elements of the `Sitecore.ContentSearch` namespace, additional verbose logging can be enabled.

The `VerboseLogger` is instantiated conditionally during initial search configuration.

The `VerboseLogger` is instantiated only when the following setting is set to `true`:

```
"Indexing.VerboseLogging"
```

Since this setting is not present in the configuration out of the box and falls back to `false`, the `VerboseLogger` is disabled by default. See the `Sitecore.ContentSearch.VerboseLogging.config.example` file if you wish to enable Verbose Logging.

It is important to enable the `VerboseLogger` component only in special circumstances and never run it for long periods in production. Otherwise, this would result in an extremely large log file, which may have performance implications.

This feature is designed to facilitate search index configuration and provide necessary insight in troubleshooting scenarios. For example, if a particular item is not getting indexed, `VerboseLogger` can provide more context and help figure out the problem.

The `VerboseLogger` is using a rich set of indexing events:

```
indexing:excludedfromindex
indexing:start
indexing:end
indexing:addingrecursive
indexing:addedrecursive
indexing:adding
indexing:added
indexing:refreshstart
indexing:refreshend
indexing:deleteitem
indexing:deletegroup
indexing:updatingitem
indexing:updateditem
indexing:updatedependents
indexing:refreshstart
indexing:refreshend
indexing:propertyset
indexing:propertyget
indexing:propertyadd
```

3.2 Crawling Log

The *Crawling Log* is designed to provide more insight into what's going on during the indexing process.

Similar to other loggers, the *Crawling Log* is defined in the `web.config` file, in the `<log4net />` section. The default logging level is `INFO`:

```
<logger name="Sitecore.Diagnostics.Crawling" additivity="false">
      <level value="INFO" />
      <appender-ref ref="CrawlingLogFileAppender" />
</logger>
```

The `appender` for this log is defined in the same section, and by default is setup to write to a `.txt` file under the folder `data/logs`:

```
<appender name="CrawlingLogFileAppender" type=
   "log4net.Appender.SitecoreLogFileAppender, Sitecore.Logging">
      <file value="$(dataFolder)/logs/Crawling.log.{date}.txt" />
      <appendToFile value="true" />
      <layout type="log4net.Layout.PatternLayout">
        <conversionPattern value="%4t %d{ABSOLUTE} %-5p %m%n" />
      </layout>
</appender>
```

Since the implementation is based on Log4Net, you can tweak the `appender` to log to a Windows Event Log or database, or any other location. For more information, see the *Log4Net* documentation.

Here is a sample of the output from the *Crawling Log*. As you can see, when you start Sitecore, the log renders information on how each index is configured and initialized.

```
INFO  [Index=sitecore_core_index] Initializing IntervalAsynchronousUpdateStrategy
      with interval '00:01:00'.
INFO  [Index=sitecore_core_index] Initializing LuceneDatabaseCrawler.
      DB:core / Root:/sitecore
INFO  [Index=sitecore master index] Initializing SynchronousStrategy.
INFO  [Index=sitecore master index] Initializing LuceneDatabaseCrawler.
      DB:master / Root:/sitecore
INFO  [Index=sitecore_web_index] Initializing OnPublishEndAsynchronousStrategy.
INFO  [Index=sitecore web index] Initializing LuceneDatabaseCrawler.
      DB:web / Root:/sitecore
INFO  [Index=custom master] Initializing IntervalAsynchronousUpdateStrategy
      with interval '00:00:05'.
INFO  [Index=custom_master] Initializing LuceneDatabaseCrawler. DB:master /
      Root:{D70CBEED-6DCF-483F-978F-6FC3C8049512}
INFO  [Index=custom_web] Initializing OnPublishEndAsynchronousStrategy.
INFO  [Index=custom_web] Initializing LuceneDatabaseCrawler.
      DB:web / Root:{D70CBEED-6DCF-483F-978F-6FC3C8049512}
INFO  [Index=custom_web] Creating primary and secondary directories
INFO  [Index=custom_web] Resolving directories from index property
      store for index 'custom_web'
INFO [Index=custom_master] IntervalAsynchronousUpdateStrategy executing.
```

The *Crawling Log* will also output every time an index update strategy is hit for a particular index, when a full rebuild is triggered on a particular index.

Because the *Crawling Log* is set to output at the `INFO` level by default, the output information will be limited.

If you require more detailed logs of indexing activity for troubleshooting purposes, you can change the logging level to `DEBUG`:

```
<logger name="Sitecore.Diagnostics.Crawling" additivity="false">
      <level value="DEBUG" />
      <appender-ref ref="CrawlingLogFileAppender" />
</logger>
```

This configuration change, along with enabling the Verbose Logger, will produce a very detailed, item level indexing log. For obvious reasons, this setting is meant to be used only for troubleshooting at very short times in any environment.

Here are some use cases when the *Crawling Log* can be helpful:

- One of my indexes is not getting updated.

- I have determined that some items are not being indexed, but I don't know why.

- For some reason, full rebuild is triggered for my index. I need to understand why.

- I would like to explore indexing activity on a particular server.

- I am setting up a multi-server environment, which is heavily relying on ContentSearch. I need to test how my index is being updated across all servers.

3.3 Search Log

The Search Log provides an ultimate insight on how your search queries are executed.

Similar to other loggers, the *SearchLog* is defined in the `<log4net />` section of `web.config`, with the default logging level set to `INFO`:

```
<logger name="Sitecore.Diagnostics.Search" additivity="false">
  <level value="INFO" />
  <appender-ref ref="SearchLogFileAppender" />
</logger>
```

The `appender` for this log is defined in the same section, and by default is setup to write to a `.txt` file under the folder `data/logs`:

```
<appender name="SearchLogFileAppender"
    type="log4net.Appender.SitecoreLogFileAppender, Sitecore.Logging">
    <file value="$(dataFolder)/logs/Search.log.{date}.txt" />
    <appendToFile value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%4t %d{ABSOLUTE} %-5p %m%n" />
    </layout>
</appender>
```

Since the implementation is based on Log4Net, you can configure the `appender` to log to a Windows Event Log or database, or any other location. For more information, see the *Log4Net* documentation.

Here is a sample output of the SearchLog:

```
3212 19:31:56 INFO  ExecuteQueryAgainstLucene :
 +_datasource:sitecore +(+(+_path:11111111111111111111111111111111 +_latestversion:1)
 +mileagehwy:[1 TO 4mileagecity]) - Filter : 3212 19:31:56
INFO  Results from web database :8818
```

If you need to enable full level debug of content searches, enable this setting in the `Sitecore.ContentSearch..config` file and change the logging level for the SearchLogger to `DEBUG`:
`ContentSearch.EnableSearchDebug = true`

```
<logger name="Sitecore.Diagnostics.Search" additivity="false">
    <level value="DEBUG" />
    <appender-ref ref="SearchLogFileAppender" />
</logger>
```

It is important to understand the value of the SearchLog in context of a particular persona:

For developers, it is important to understand how the Linq code translates into native search queries passed onto the Search Provider (Lucene or Solr out of the box).

For the administrators, it is important to understand which search queries are performed on a particular server. This information can be used for further optimization, such as inclusion into the `queryWarmup` pipeline.

Chapter 4

Index Management

This chapter explains how to rebuild, manage, and optimize indexing in Sitecore 7.0.

- Rebuilding Indexes

- Index Property Store

- Index Dependent Html Cache Management

- Index Update Strategies

- Boosting Search Results

4.1 Rebuilding Indexes

In certain situations, such as, when deploying a site to a production environment or when indexes are out of sync or corrupted, it may be necessary to perform a full rebuild operation on a particular index.

4.1.1 Rebuilding Search Indexes in Sitecore

To rebuild indexes from the Sitecore Client:

1. Log in to the **Sitecore Desktop**.
2. Open the **Control Panel**.
3. Click **Indexing** and then click **Indexing Manager***.*
4. In the wizard select the indexes you want to rebuild and click **Rebuild**.

4.1.2 Rebuilding Search Indexes using Custom Code

To rebuild indexes from custom code, run one of these scripts from a custom `.aspx` page:

```
// To rebuild "new" search indexes, use this piece of code for every "new" index
IndexCustodian.FullRebuild(ContentSearchManager.GetIndex("[INDEX NAME]", true);

// Or to rebuild all indexes, use the following piece of code:
IndexCustodian.RebuildAll();
```

4.1.3 Rebuilding Search Indexes using Content Editor

In order to facilitate quick index management, in the **Content Editor**, on the **Developer** tab, an **Indexing Power Tools** group has been added:
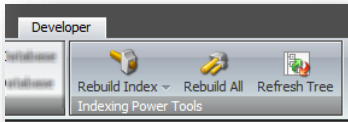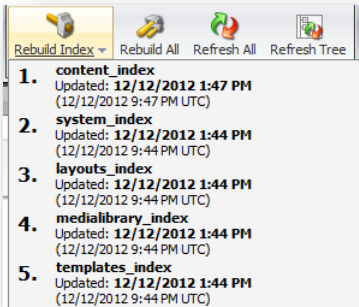

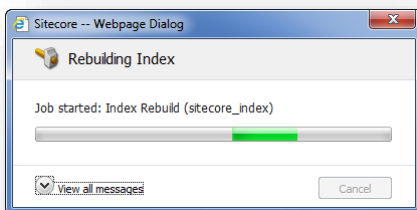This group contains the following commands:

Rebuild Index Command

When you click **Rebuild Index**, a list of all indexes registered within the system is displayed:

This list contains the id of the index, the last time index was updated in local time, and the same timestamp in UTC.

To rebuild an index, click it in the list and a progess dialog box is displayed:

Rebuild All

Performs a full rebuild on all the indexes registered in the system. A progress dialog box is displayed for each index.

Refresh Tree

This operation will process the selected item and all its descendants recursively and will force an update operation on all indexes that are responsible for the selected content area. This command is reserved for special cases when the incremental indexing is either not working as expected, or when the Manual update strategy is used. Use this feature only when absolutely necessary.

4.1.4   SwitchOnRebuildLuceneIndex

This class inherits from `LuceneIndex` and adds important capability of maintaining two directories for a particular index. This solves the problem of `LuceneIndex` implementation which resets (deletes) the index directory before a full index rebuild. Since this problem is important only for production environments, you can reconfigure your custom index with the `SwitchOnRebuildLuceneIndex` implementation during testing and before moving to a production environment.

When the `SwitchOnRebuildLuceneIndex` object is initialized, the presence of a secondary folder is verified. If the folder does not exist, it is created with the baseline directory name + `_sec`:

The index uses the `IndexReader.LastModified(Directory)` method to choose a primary directory based on the last modified date. The most up to date directory is used as the primary directory. Once you have enabled `SwitchOnRebuildIndex`, you must perform the Post-Activation steps to rebuild the indexes

Note
The primary directory is used for index read and update operations. The secondary directory is a fallback for read operations during a full index rebuild.

The information about which directory is primary and which one is secondary is written to the *Index Property Store*.

If the index has already been initialized, the information about which directory is primary and which is secondary is retrieved from the *Index Property Store*.

During the initialization of the `SwitchOnRebuildLuceneIndex object`, the following entries are written to the `CrawlingLog`:

```
"Resolving directories from index meta data store"
"Resolving directories by last time modified"
"Primary directory last modified = '…'"
"Secondary directory last modified = '…'"
"ReadUpdateDirectory is set to …"
"FullRebuildDirectory is set to '{0}'"
```

When the full index rebuild is completed, the primary and secondary directories are switched.

How to activate

To use this implementation, change the type reference on a particular search index to `Sitecore.ContentSearch.LuceneProvider.SwitchOnRebuildLuceneIndex`:

```
<indexes hint="list:AddIndex">
    <index id="content_index"
            type="Sitecore.ContentSearch.LuceneProvider.SwitchOnRebuildLuceneIndex,
                Sitecore.ContentSearch.LuceneProvider">

        <param desc="name">$(id)</param>
        <param desc="folder">$(id)</param>
 ...
```

Post-Activation-steps

After the configuration file has been adjusted, and the search index has been adjusted to use the `SwitchOnRebuildLuceneIndex` method, your website will use indexes from the primary directory. Each time you perform a full index rebuild, it is carried out in the secondary directory. The secondary directory then becomes the primary one after the rebuild.

4.2 Index Property Store

The purpose of the index property store is to persist additional meta data for indexes such as last updated timestamp.

Configuration

The out of the box index property store implementation is defined within the `sitecore/contentSearch` area within the `Sitecore.ContentSearch.Lucene.DefaultIndexConfiguration.config` file:

```
<databasePropertyStore
        type="Sitecore.ContentSearch.Maintenance.IndexDatabasePropertyStore,
        Sitecore.ContentSearch">
            <Key>$(1)</Key>
            <Database>core</Database>
</databasePropertyStore>
```

As the name suggests, this index property store implementation is using the Sitecore database as persistent store, specifically, the `Properties` table of the designated database. As can be seen from the parameters, this component is set to use the core database out of the box.

The property store is assigned to each index using the following configuration:

```
<index id="content" type="Sitecore.ContentSearch.LuceneProvider.LuceneIndex,
        Sitecore.ContentSearch.LuceneProvider">
                <param desc="name">$(id)</param>
                <param desc="folder">$(id)</param>
                <param desc="propertyStore"
                    ref="contentSearch/databasePropertyStore" param1="$(id)" />
```

The first `param1` attribute must be set to $(id) or something else that is unique within all indexes.

Essentially, the index property store is a key/value based storage. The key consists of the Master Key and the key of the record that needs to be saved. The Master Key depends on configuration. In the example above, the Key parameter on the `databasePropertyStore` definition node (`<Key>$(1)</Key>`) will pick up the first parameter from the place where `databasePropertyStore` is referenced:

```
<param desc="propertyStore" ref="contentSearch/databasePropertyStore" param1="$(id)" />
```

Since this record is defined within the `<index />` element, `$(id)` will be equal to "content":

```
<index id="content">
```

This way the `databasePropertyStore` can guarantee key uniqueness across all indexes. In addition, the InstanceName gets appended to the MasterKey, which guarantees key uniqueness across all environments.

So taking the index configuration above into account, if we attempt to save a custom property to the index property store using this code:

```
index.PropertyStore.Set("docs-count", "123");
```

The value "123" will be saved with key = content_CM01_docs-count
        where content is index id, CM01 is instance name.

How to create a custom index property store implementation:

If the out of the box implementation does not meet the implementation needs, a custom index property store implementation can be created by implementing the `IIndexPropertyStore` interface.

After the custom index property store implementation is created, it needs to be defined somewhere within the `<contentSearch />` area and assigned to appropriate indexes.

For example:

```
<contentSearch>
        <fileSystemPropertyStore type="Custom.IndexFileSystemPropertyStore, Custom">
           <Key>$(1)</Key>
           <FilePath>$(dataFolder)/indexpropertystore</FilePath>
        </fileSystemPropertyStore>
...
<index id="content" type="Sitecore.ContentSearch.LuceneProvider.
        LuceneIndex, Sitecore.ContentSearch.LuceneProvider">
                <param desc="name">$(id)</param>
                <param desc="folder">$(id)</param>
                <param desc="propertyStore"
                         ref="contentSearch/fileSystemPropertyStore"
                         param1="$(id)"/>
```

4.3 Index Dependent Html Cache Management

Some Index Update Strategies are designed to be invoked either on the publish-end event or on an interval basis. If the Sitecore instance is leveraging html cache for renderings/controls/sublayouts which contain code that depends on one or more indexes, this can create a race condition between html cache clearing and index update operations.

To solve this issue, the `renderings/controls/sublayouts` that depend on an index need to be marked with the **Clear on Index Update** flag (along with the "Cacheable" checkbox. They will be removed from cache and can be reloaded with current data when the index update is completed
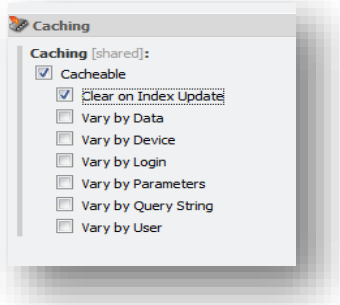
Similar to how other Vary by settings work, this can either be done on the rendering definition item or on the layout details for particular content item.

This flag will make the html cache of renderings be cleared on index update by the `IndexDependentCacheManager`. The execution of this component is triggered from the `indexing:end` and `indexing:end:remote` events:

```
<event name="indexing:end">
    <handler type="Sitecore.ContentSearch.Maintenance.
        IndexDependentHtmlCacheManager, Sitecore.ContentSearch" method="Clear" />
</event>
<event name="indexing:end:remote">
    <handler type="Sitecore.ContentSearch.Maintenance.
        IndexDependentHtmlCacheManager, Sitecore.ContentSearch" method="Clear" />
</event>
```

These events are defined within `Sitecore.ContentSearch.config`.

4.4 Index Update Strategies

Index Update Strategies are designed to provide a transparent and flexible model for index maintenance. Each index can be configured with unique set of index update strategies, however, it is not recommended to have more than 3 update strategies per index. Also, it is strongly not recommended to leverage update strategies of a similar nature within one index (more on this below).

Sitecore 7.0 ships with a diverse set of Index Update Strategies, however developers can extend this set even further. All out of the box update strategies are defined under the following node within the `Sitecore.ContentSearch` configuration files:

```
sitecore/contentSearch/indexUpdateStrategies
<manual type="Sitecore.ContentSearch.Maintenance.Strategies.ManualStrategy,
  Sitecore.ContentSearch" />
```

4.4.1 RebuildAfterFullPublish Strategy

This is how this strategy is defined in configuration:

```
<rebuildAfterFullPublish type="Sitecore.ContentSearch.Maintenance.Strategies.
  RebuildAfterFullPublishStrategy, Sitecore.ContentSearch" />
```

This strategy does exactly what the name implies. For example, in environments where a full publish is required to run regularly, it makes no sense to trigger an incremental index rebuild, as it will be quite taxing. Instead, this strategy will trigger a full index rebuild after the completion of the full publish process, which will be much more efficient. In a distributed environment, the index rebuild will be triggered on all remote servers where this strategy is configured. In this case, the Event Queue must be enabled for this environment. Please refer to the *Sitecore Search Scaling Guide* for more information.

During the initialization, it subscribes to the `OnFullPublishEnd` event and triggers a full index rebuild.

When this strategy is attached to an index and initialized, the following message will be seen in the `CrawlingLog` file:

```
Initializing RebuildAfterFullPublishStrategy for index '<index_name>'
```

When this strategy is triggered, the following message will be seen in the `CrawlingLog` file:

```
RebuildAfterFullPublishStrategy triggered on index '<index_name>'
```

Attaching to an index

This is how you can attach this strategy to an index:

```
<index id="sitecore_index" type="Sitecore.ContentSearch.LuceneProvider.
        LuceneIndex, Sitecore.ContentSearch.LuceneProvider">
  <param desc="name">$(id)</param>
  <param desc="folder">$(id)</param>
  <strategies hint="list:AddStrategy">
     <strategy ref="contentSearch/indexUpdateStrategies/rebuildAfterFullPublish" />
  </strategies>
  <Analyzer ref="search/analyzer" />
  ...
```

Recommendation

This strategy should not be combined with the Synchronous Strategy, while it may be combined with others.

Since the execution of this strategy effectively causes a full index rebuild, it is recommended to combine this strategy with the usage of the `SwitchOnRebuildLuceneIndex` implementation.

When this strategy is used together with the "onPublishEndAsync" strategy, it needs to be registered as the first entry in the list in order to be triggered first:

```
<index id="sitecore index" type="Sitecore.ContentSearch.LuceneProvider.LuceneIndex,
        Sitecore.ContentSearch.LuceneProvider">
  <param desc="name">$(id)</param>
  <param desc="folder">$(id)</param>
  <strategies hint="list:AddStrategy">
      <strategy ref="contentSearch/indexUpdateStrategies/rebuildAfterFullPublish" />
      <strategy ref="contentSearch/indexUpdateStrategies/onPublishEndAsync" />
  </strategies>
  <Analyzer ref="search/analyzer" />
```

4.4.2 OnPublishEndAsync Strategy

This is how this strategy is defined in configuration, notice the additional parameter `database` that is passed to the constructor of the `OnPublishEndAsynchronousStrategy` class:

```
<onPublishEndAsync type="Sitecore.ContentSearch.Maintenance.Strategies.
        OnPublishEndAsynchronousStrategy, Sitecore.ContentSearch">
        <param desc="database">web</param>
        <CheckForThreshold>true</CheckForThreshold>
</onPublishEndAsync>
```

The "database" parameter defines the database from where to look up the item changes for the processing. See more information below about the `CheckForThreshold` property.

This strategy does exactly what the name implies. During the initialization, it subscribes to the `OnPublishEnd` event and triggers an incremental index rebuild. With separate CM and CD servers, this event will be triggered via the `EventQueue` object, meaning that the `EventQueue` object needs to be enabled for this strategy to work in such environment.

When this strategy is attached to an index and initialized, the following message will be seen in the `CrawlingLog` file:

```
Initializing OnPublishEndAsynchronousStrategy for index '<index_name>'.
```

When this strategy is triggered, the following message will be seen in the `CrawlingLog` file:

```
OnPublishEndAsynchronousStrategy triggered on index '<index_name>'
```

Processing

The strategy will use the `EventQueue` object from the database it was initialized with:

```
<param desc="database">web</param>
```

This means that there are multiple criteria towards successful execution for this strategy:

- This database must be specified in the `<databases />` section of the configuration file.
- The `EnableEventQueues` setting must be set to `true`.
- The `EventQueue` table within the preconfigured database should have entries dated later than index's last update timestamp.

In order to prevent excessive processing of the Event Queue, the strategy will force a full index rebuild when the number of entries in the history table exceeds the number defined in the following setting: Indexing.`FullRebuildItemCountThreshold`. In most cases, this means that a substantial publishing or deployment occurred, which should always trigger a full index rebuild. This behavior will only be triggered when the following property in configuration is set to `true` (which is the default):

```
<CheckForThreshold>true</CheckForThreshold>
```

If this setting is set to `true` it is recommended to use the `SwitchOnRebuildLuceneIndex` implementation for an index that is using this strategy.

The `Indexing.FullRebuildItemCountThreshold` setting is not set out of the box and defaults to 100000.

Attaching to an index

This is how you can attach this strategy to an index:

```
<index id="sitecore index" type="Sitecore.ContentSearch.LuceneProvider.LuceneIndex,
        Sitecore.ContentSearch.LuceneProvider">
   <param desc="name">$(id)</param>
   <param desc="folder">$(id)</param>
   <strategies hint="list:AddStrategy">
      <strategy ref="contentSearch/indexUpdateStrategies/onPublishEndAsync" />
   </strategies>
   <Analyzer ref="search/analyzer" />
   ...
```

Recommendation

This strategy should not be combined with the following other strategies:

- SynchronousStrategy
- intervalAsync

This strategy can be combined with the following other strategies:

- rebuildAfterFullPublish
- remoteRebuild

This strategy is recommended for multi-server/multi-instance environments where the EventQueue is already enabled as a part of the Scaling Guide configuration. See the *Sitecore Search Scaling Guide* for more information.

4.4.3   IntervalAsynchronous Strategy

This is how this strategy is defined in configuration, notice two parameters: `database` and `interval`:

```
<intervalAsyncMaster type="Sitecore.ContentSearch.Maintenance.Strategies.
        IntervalAsynchronousStrategy, Sitecore.ContentSearch">
   <param desc="database">master</param>
   <param desc="interval">00:00:10</param>
   <CheckForThreshold>true</CheckForThreshold>
</intervalAsyncMaster>
```

The `database` parameter defines the database from where to look up the item changes for the processing.

The `interval` parameter defines the frequency of the strategy trigger.

The `CheckForThreshold` parameter is described further down.

When this strategy is attached to an index and initialized, the following message will be seen in the `CrawlingLog` file:

```
Initializing IntervalAsynchronousUpdateStrategy for index '<index_name>'.
```

When this strategy is triggered, the following message will be seen in the `CrawlingLog` file:

```
IntervalAsynchronousUpdateStrategy triggered on index '<index_name>'
```

Processing

This strategy is triggered on a time interval instead of the `OnPublishEnd` event, and is relying on the History Engine Store to process item changes.

- This referenced database must exist.

- This referenced database must have History Engine enabled.

- The History Engine should have entries dated later than index's last update timestamp.

It is leveraging an internal timer that is initialized with a predefined `interval` value and is triggered when the timer fires (every 10 seconds as shown below).

```
<intervalAsync type="Sitecore.ContentSearch.Maintenance.Strategies.
       IntervalAsynchronousStrategy, Sitecore.ContentSearch">
  <param desc="database">web</param>
  <param desc="interval">00:00:10</param>
  <CheckForThreshold>true</CheckForThreshold>
</intervalAsync>
```

In order to prevent excessive processing of the History Engine Store, the strategy will force full index rebuild when the number of entries in the history table exceeds the number defined in the following setting: `Indexing.FullRebuildItemCountThreshold`. In most cases this means that a substantial publishing or deployment occurred which should always trigger a full index rebuild.

This behavior will only be triggered when the following property in configuration is set to `true` (which is the default):

```
<CheckForThreshold>true</CheckForThreshold>
```

If this setting is set to `true`, it is recommended to use the `SwitchOnRebuildLuceneIndex` implementation.

The `Indexing.FullRebuildItemCountThreshold` setting is not enabled out of the box and defaults to 100000.

Attaching to an index

This is how you can attach this strategy to an index:

```
<index id="sitecore_index" type="Sitecore.ContentSearch.LuceneProvider.LuceneIndex,
       Sitecore.ContentSearch.LuceneProvider">
  <param desc="name">$(id)</param>
  <param desc="folder">$(id)</param>
  <strategies hint="list:AddStrategy">
     <strategy ref="contentSearch/indexUpdateStrategies/intervalAsync" />
  </strategies>
  <Analyzer ref="search/analyzer" />
  ...
```

Recommendation

This strategy should not be combined with the following other strategies:

- SynchronousStrategy
- onPublishEndAsync

This strategy can be combined with the following other strategies:

- rebuildAfterFullPublish
- remoteRebuild

This strategy is recommended for the master database indexes or for single-server environments where it is important to conserve as many resources as possible.

Also, the strategy makes sense on less critical indexes which may not need to be updated as frequently. Implementers can adjust the interval as they see fit.

Out of the box, this strategy is created specifically for the `core` and `master` databases:

```
<intervalAsyncCore type="Sitecore.ContentSearch.Maintenance.Strategies.
        IntervalAsynchronousStrategy, Sitecore.ContentSearch">
   <param desc="database">core</param>
   <param desc="interval">00:01:00</param>
   <CheckForThreshold>true</CheckForThreshold>
</intervalAsyncCore>
<intervalAsyncMaster type="Sitecore.ContentSearch.Maintenance.Strategies.
        IntervalAsynchronousStrategy, Sitecore.ContentSearch">
   <param desc="database">master</param>
   <param desc="interval">00:00:10</param>
   <CheckForThreshold>true</CheckForThreshold>
</intervalAsyncMaster>
```

4.4.4   Synchronous Strategy

This strategy provides the most close-to-real-time index update as possible and is the most expensive both in terms of CPU and IO. Before using this strategy, please read the *Recommendation* section carefully.

This is how this strategy is defined in configuration:

```
<sync type="Sitecore.ContentSearch.Maintenance.Strategies.SynchronousStrategy,
        Sitecore.ContentSearch" />
```

When this strategy is attached to an index and initialized, the following message will be seen in the `CrawlingLog` file:

```
Initializing SynchronousStrategy for index '<index_name>'.
```

When this strategy is triggered, the following message will be seen in the `CrawlingLog` file:

```
SynchronousStrategy triggered on index '<index_name>'
```

Processing

This strategy hooks up to the low-level `DataEngine` events such as `ItemSaved` and `ItemSavedRemote`. When used on a single server instance, the strategy guarantees an index update immediately after item update. On a multi-server environment, the strategy will work along with the `EventQueue` that will broadcast remote `ItemSavedRemote` events. The moment an item is published and `ItemSavedRemote` event is raised, the strategy will be triggered.

Requirements

Sitecore must be configured according to the Scaling Guide, i.e. with the `EventQueue` enabled. See the document *Sitecore Search Scaling Guide* for more information.

Attaching to an index

This is how you can attach this strategy to an index:

```
<index id="sitecore_index" type="Sitecore.ContentSearch.LuceneProvider.LuceneIndex,
        Sitecore.ContentSearch.LuceneProvider">
   <param desc="name">$(id)</param>
   <param desc="folder">$(id)</param>
```

```
<strategies hint="list:AddStrategy">
    <strategy ref="contentSearch/indexUpdateStrategies/sync" />
</strategies>
<Analyzer ref="search/analyzer" />
...
```

Recommendation

This strategy can only be combined with the following strategy:

- remoteRebuild

This strategy should be used only when the implementation requires immediate index updates and where a dedicated indexing server infrastructure with plenty of processing resources is in place. Since the *OnPublishEndAsync* strategy already provides an effective way of maintaining the search indexes on CD servers after publishing, the only application of this strategy that can be recommended is on CM servers for the indexes that process the master database, and where the timing of the index update is absolutely critical. Such CM environments should account for the extra CPU and IO intensity of the strategy. Using this strategy on a CM server with lots of content entry activity can severely impact performance of the system. For most cases, the *IntervalAsyncronous* strategy configured for the master database should be more than sufficient.

4.4.5 RemoteRebuildStrategy

This strategy is subscribing to the `OnIndexingEndedRemote` event which is triggered when a particular index is rebuilt. The strategy will react only when a full index rebuild is performed.

This mechanism allows for rebuilding remote indexes when an index is forced to be rebuilt from the Control Panel, for example.

```
<remoteRebuild type="Sitecore.ContentSearch.Maintenance.Strategies.
        RemoteRebuildStrategy, Sitecore.ContentSearch" />
```

Attaching to an index

This is how you can attach this strategy to an index:

```
<index id="sitecore index" type="Sitecore.ContentSearch.LuceneProvider.LuceneIndex,
        Sitecore.ContentSearch.LuceneProvider">
  <param desc="name">$(id)</param>
  <param desc="folder">$(id)</param>
  <strategies hint="list:AddStrategy">
     <strategy ref="contentSearch/indexUpdateStrategies/remoteRebuild" />
  </strategies>
  <Analyzer ref="search/analyzer" />
  ...
```

Recommendation

This strategy can be combined with any other strategy and can be quite handy within multi-server environments where each Sitecore instance maintains its own copy of the index. This way full rebuild can be triggered from one CM server, and this event will be raised on all remote servers where the index is configured with this strategy.

Important
In order for this strategy to work on remote servers, the index name should be identical to the one for which the index rebuild was forced.

Important
This strategy requires the Event Queue to be enabled within the environment. Please refer to the *Sitecore Search Scaling Guide* for information on how to enable the Event Queue.

Important
In order for this strategy to work, the database that is assigned for system event queue storage (`core` by default) should be shared between the Sitecore instance where the rebuild happened and where it needs to be replayed.

4.4.6   Manual Strategy

This is a special kind of strategy that essentially disables any automatic index update. Indexes that rely on this strategy will have to be rebuilt manually.

```
<manual type="Sitecore.ContentSearch.Maintenance.Strategies.ManualStrategy,
        Sitecore.ContentSearch" />
```

When this strategy is attached to an index and initialized, the following message will be seen in the `CrawlingLog` file:

```
Initializing ManualStrategy for index '<index_name>'.
Index will have to be rebuilt manually
```

Attaching to an index

This is how you can attach this strategy to an index:

```
<index id="sitecore index" type="Sitecore.ContentSearch.LuceneProvider.LuceneIndex,
        Sitecore.ContentSearch.LuceneProvider">
   <param desc="name">$(id)</param>
   <param desc="folder">$(id)</param>
   <strategies hint="list:AddStrategy">
      <strategy ref="contentSearch/indexUpdateStrategies/manual" />
   </strategies>
   <Analyzer ref="search/analyzer" />
   ...
```

Recommendation

It does not make sense to combine this strategy with any other strategy. It is reserved for special cases when the whole indexing process must be outsourced to a dedicated server and no index update should happen on a particular Sitecore instance.

4.5 Boosting Search Results at Indexing Time

Items and specific fields can be boosted at indexing time to score these specific items better than others. This can, for example, be used to promote popular/most sold books in a bookshop and show less sold books as ranked lower than popular books in a search result.

Boosting can be applied on the item/document level or at field level.

Field level boosting can be used to make matches on a specific field more or less important. For example, in a bookshop, the book product items have three fields: Title, Summary and Foreword. The searches are still performed on all three fields, but matches in the Foreword field should be less important that Title and Summary.

Note
Boosting can also be applied at query time and will also take effect on items or fields that have been boosted at indexing time.

Configuration

```
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
  <sitecore>
    ...
    <pipelines>
      <!-- RESOLVE FIELD LEVEL BOOSTING
           Pipeline for resolving boosting rules on fields.
           Arguments: (Item) Item being indexed
           Example : Boost search results by a field value.
      -->
      <indexing.resolveFieldBoost help="Processors should derive from
        Sitecore.ContentSearch.Pipelines.ResolveBoost.ResolveFieldBoost.
        BaseResolveFieldBoostPipelineProcessor">
         <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost
        .ResolveFieldBoost.SystemFieldFilter, Sitecore.ContentSearch"/>
         <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
         ResolveFieldBoost.FieldDefinitionItemResolver, Sitecore.ContentSearch"/>
         <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
         ResolveFieldBoost.StaticFieldBoostResolver, Sitecore.ContentSearch"/>
      </indexing.resolveFieldBoost>

      <!-- RESOLVE ITEM LEVEL BOOSTING
           Pipeline for resolving boosting rules on items.
           Arguments: (Item) Item being indexed
           Example : Boost search results by an Item Template.
      -->
      <indexing.resolveItemBoost help="Processors should derive from
       Sitecore.ContentSearch.Pipelines.ResolveBoost.
       ResolveItemBoost.BaseResolveItemBoostPipelineProcessor">
        <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
         ResolveItemBoost.ItemLocationFilter, Sitecore.ContentSearch">
           <includedLocations hint="list">
               <content>/sitecore/content</content>
               <media>/sitecore/media library</media>
           </includedLocations>
        </processor>
        <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
         ResolveItemBoost.StaticItemBoostResolver, Sitecore.ContentSearch"/>
        <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
         ResolveItemBoost.LocalRuleBasedItemBoostResolver, Sitecore.ContentSearch"/>
        <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
         ResolveItemBoost.GlobalRuleBasedItemBoostResolver, Sitecore.ContentSearch"/>
      </indexing.resolveItemBoost>
    </pipelines>
```

```
                <!-- BOOSTING MANAGER
                      The manager class controlling the boosting resolution logic
                -->
                <boostingManager defaultProvider="default" enabled="true">
                  <providers>
                    <clear/>
                    <add name="default" type="Sitecore.ContentSearch.Boosting.
                     PipelineBasedBoostingProvider, Sitecore.ContentSearch"/>
                  </providers>
                </boostingManager>
                ...
            </sitecore>
        </configuration>
```

Rules for Boost

- Default value for Boost is 1

- Values greater than 1 push the results to the top.

- Values less than 1 drag the results to the bottom (this is not often used).

4.5.1   Field Level Boosting

Field level boosting is resolved within the `indexing.resolveFieldBoost` pipeline that has the following processors enabled:

```
                <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
                  ResolveFieldBoost.SystemFieldFilter, Sitecore.ContentSearch"/>
                <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
                  ResolveFieldBoost.FieldDefinitionItemResolver, Sitecore.ContentSearch"/>
                <processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
                  ResolveFieldBoost.StaticFieldBoostResolver, Sitecore.ContentSearch"/>
```

The `SystemFieldFilter` processor makes sure that the feature ignores all the system fields — whose names start with "__". This ensures that the impact on performance of the indexing operation is minimized.

The `FieldDefinitionItemResolver` takes the instance of the `Sitecore.Data.Fields.Field` and resolves the Field Definition Item that is expected to provide the boost value.

Once the template definition item is resolved, the final processor, `StaticFieldBoostResolver` is fired which simply reads the boost value out of the Field Definition Item, which is referred to as "static field boost".

To activate the Indexing section with the Boost Value field, on the **View** tab toggle the **Standard Fields** checkbox:
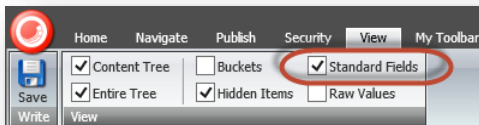
Out of the box, the Field Level Boosting feature is only enabled for static boost resolution. You can enable the Rule-Based Boosting for fields yourself. For more information about this, see the *Item Level Boosting* section.

4.5.2 Item Level Boosting

Item level boosting is resolved within the `indexing.resolveItemBoost` pipeline that has the following processors enabled:

```
<processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
 ResolveItemBoost.ItemLocationFilter, Sitecore.ContentSearch">
   <includedLocations hint="list">
      <content>/sitecore/content</content>
      <media>/sitecore/media library</media>
   </includedLocations>
</processor>
<processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
   ResolveItemBoost.StaticItemBoostResolver, Sitecore.ContentSearch"/>
<processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
   ResolveItemBoost.LocalRuleBasedItemBoostResolver, Sitecore.ContentSearch"/>
<processor type="Sitecore.ContentSearch.Pipelines.ResolveBoost.
   ResolveItemBoost.GlobalRuleBasedItemBoostResolver, Sitecore.ContentSearch"/>
```

The `ItemLocationFilter` processor makes sure that only items in the specified locations are processed by the pipeline. This ensures that the impact on performance of the indexing operation is minimized. If you need to extend this list, make sure to use the syntax below

```
<includedLocations hint="list">
   <content>/sitecore/content</content>
   <media>/sitecore/media library</media>
   <custom>/sitecore/custom</custom>
</includedLocations>
```

The `StaticItemBoostResolver` processor simply reads the boost value out of the indexed item, which is referred to as "static item boost".

This value can also be set for all items based on the Sample Item template via the __Standard Values facility:

To activate the Indexing section with the Boost Value field, on the **View** tab toggle the **Standard Fields** checkbox:
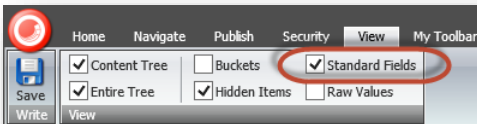
The `LocalRuleBasedItemBoostResolver` and `GlobalRuleBasedItemBoostResolver` processors execute Rule-Based Boosting for the indexed item. Both are executed in a similar fashion, using the Rules Engine, with the difference being the location from where the boosting rules are looked up. For more information, see the *Rules Engine Cookbook*.

The `LocalRuleBasedItemBoostResolver` picks up the rules from the **Boosting Rules** field in the Indexing section:

The source of the **Boosting Rules** field is restricted to the following location where all boosting rules are managed:

`/sitecore/system/Settings/Indexing and Search/Boosting Rules`

This set of boosting rules can also be set on the __Standard Values level, just as any other field value.

The `GlobalRuleBasedItemBoostResolver` will process all boosting rules created under `/sitecore/system/Settings/Indexing and Search/Boosting Rules/Global Rules`.

In this example, the boosting rule called "Item has new in title" will be executed for every item that is being indexed. So it is important to keep absolutely necessary global boosting rules. Otherwise, this could have a drastically negative impact on indexing time. In other words, use the local boosting rules as much as possible.

For more information on how the boosting rules are evaluated, see the *Rule-Based Boosting section*.

4.5.3 Rule-Based Boosting

This feature is based on the Rules Engine allowing for easy creation and extension of boosting rules. For more information, see the *Rules Engine Cookbook*.

In order to create a boosting rule, locate the following item:

`/sitecore/system/Settings/Indexing and Search/Boosting Rules`

First, you need to decide whether it's a boosting rule that will be used locally per item or per template, or it's a global rule. Each has a designated location. For example, in order to create a new local item rule, right click on the Item Rules folder and Insert a new Boosting Rule:
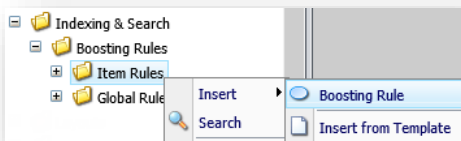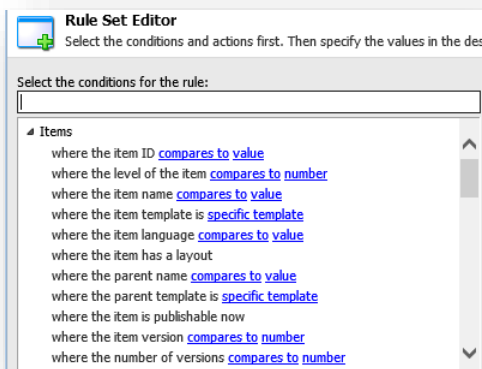

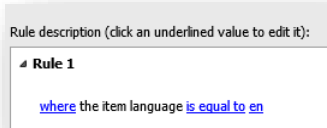Let's create a boosting rule that will boost all items in English language.
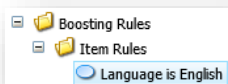
After the rule is created, click **Edit Rule**.

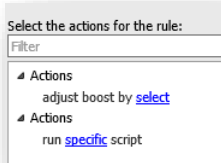The **Rule Set Editor** dialog box contains a set of familiar conditions:

Select the *where the item language compares to value* rule from the list and set the macro as follows:

The action we need is called *adjust boost by select*:

When you click on the "select" macro the following dialog will appear where you can specify the boost value. The boost value can be either a positive or negative whole number.

Once the boosting rule is saved, we need to associate it with either an item or a template.

4.5.4 Rule-Based Boosting for Fields

Since the Rule-Based Boosting for fields is not implemented out of the box, you can implement this feature with very little development effort. See the *Developer's Guide to Item Buckets and Search* for more information.

4.5.5 Troubleshooting Boosting

For Rule-Based Boosting, one common reason why boosting may not be resolved properly is that the boosting rule was not published to the target database.

For field level boosting, you have to make sure that the Field Definition Item was published as well.

Since this feature is used at indexing time, you have to make sure the item was re-indexed properly for the new boosting values to be picked up. Depending on the Index Update Strategy of choice, this could happen either immediately after an item is saved, on an interval basis or after publishing.

If the item or field level boosting efforts are not giving the desired result, the first thing you should do is debug the boosting values by changing the logger level of the CrawlingLog from INFO to DEBUG:

```
<logger name="Sitecore.Diagnostics.Crawling" additivity="false">
  <level value="DEBUG" />
  <appender-ref ref="CrawlingLogFileAppender" />
</logger>
```

This will restart the application pool and generate a new CrawlingLog file.

Next time you force an item re-index, entries will be shown in the log file, which should provide you with the necessary insight on what resolved boosting values are being set.

If the resolved item boost is not what you expect, the indexed item may pick it up from the Global Rules container, or maybe the Local Boosting Rule is not evaluating properly.

If you do not see any entries about item boosting resolving in the Crawling Log, you can enable the Verbose Logger component and review why the item is not being indexed.

Chapter 5 Pipelines

This chapter explains the pipeline processors that are used for search and indexing in Sitecore CMS 7.0.

- Overview of pipeline processors

- Indexing.getDependencies pipeline

5.1   Pipelines Overview

The following list shows the pipelines that are invoked for search procedures:

```
contentSearch.stripQueryStringParameters
contentSearch.getContextIndex
contentSearch.getGlobalSearchFilters
contentSearch.getFacets
contentSearch.processFacets
contentSearch.queryWarmup
contentSearch.translateQuery
indexing.filterIndex.inbound
indexing.filterIndex.outbound
indexing.getDependencies
indexing.resolveFieldBoost
indexing.resolveItemBoost
```

More information about these pipeline processors are listed in this section.

5.2 contentSearch.queryWarmup

This pipeline is critical for the production environment, and allows specifying a list of raw queries to be executed during Sitecore initialization. This early execution will make sure the queries are cached, so the first visitor does not have to a delay. Be aware, that this pipeline will have a negative effect on the application startup time. So it is recommended to use it only in a production environment.

To enable this feature, in the `App_Config/Include` folder, remove `.example` from the extension of the `Sitecore.Buckets.WarmupQueries.config.example` file.

To extend the list of queries, in the `sitecore/search/warmup` item, add more `<query />` elements.

5.3   indexing.getDependencies Pipeline

This pipeline is designed to address issues when a search document is built from the data coming from more than one item. Consider example with Item Cloning and two items. When the value of the title field changes on the original item, all clones need to be notified and updated. Note that the processors within the pipeline are responsible for the collection of the "indexing-dependent items", and that's it. Once the pipeline returns, all "indexing-dependent items" will be processed and updated within the index.

Please note that this processor is disabled by default and can be enabled if a particular solution uses the Item Cloning feature.

5.3.1   How to Enable/Disable

The pipeline is executed from within each crawler if the crawler's `ProcessDependencies` property is set to `true`, which is the default. To disable this feature, add the following parameter to the appropriate index under the `<Configuration />` section.

```
<index id="content" ...>
            ...

   <Configuration type="...">
       <IndexAllFields>true</IndexAllFields>
       <ProcessDependencies>false</ProcessDependencies>
```

Alternatively, if the indexes don't override default configuration with a local one (e.g., the Configuration section is missing), you can also globally change this setting under `<DefaultIndexConfiguration />`:

```
<DefaultIndexConfiguration type="...">
           <IndexAllFields>true</IndexAllFields>
           <ProcessDependencies>false</ProcessDependencies>
```

Note that if a particular index has its own "local" `<Configuration />` section, this setting will need to be defined there in order for this to take affect for a particular index. The indexes without local configuration will use the setting from the `<DefaultIndexConfiguration />` section.

5.3.2   Usage

This pipeline can be used for other scenarios as well, such as composite page design where child pages are included into the search document of the parent item. This way you can insert an additional processor that will contain the logic that determines whether the currently indexed item is a child item, so the code can return the parent as "indexing-dependent item".

This specific scenario would most likely be complemented by one or more custom Computed Fields that would perform the reverse logic of reading the child pages and adding the content to the search document created for the currently indexed item.

Each custom processor must implement
`Sitecore.ContentSearch.Pipelines.GetDependencies BaseProcessor`:

```
public abstract class BaseProcessor
    {
        public abstract void Process(GetDependenciesArgs args);
    }
```

The pipeline arguments represented by the `GetDependenciesArgs` class contain the instance of the indexed item and the list of `ItemUris` for each indexing-dependent item.

This pipeline is defined within `Sitecore.ContentSearch.config`.

5.3.3 How to troubleshoot

Consider enabling Verbose Logger to monitor whether this pipeline is triggered for a particular index. If it is enabled and the Verbose Logger is on, you should see the following messages in Crawler log:

```
    INFO  [Index=content_index] UpdateDependents content_index|sitecore://master/{604A57CD-
5FF7-4A9B-AE27-3C962CBB9E3A}?lang=en&ver=1
    // the dependent items will be updated below
    INFO  [Index=content_index] ItemUpdating content_index|sitecore://master/{4C9B6DFA-A22F-
4755-81D6-DAFE1AFDE58E}?lang=en&ver=1
    INFO  [Index=content_index] ItemUpdated content_index|sitecore://master/{4C9B6DFA-A22F-
4755-81D6-DAFE1AFDE58E}?lang=en&ver=1
    INFO  Setting Index Property 'content_index_LT-AS-CGSXDS1-
elbrus.local.net_last_updated'='20130116T202350'
```

For more information about verbose logging, see the section *Verbose Logger.*