collective.solr Documentation

Release 4.0

Plone Foundation

Dec 31, 2021

Contents

CHAPTER 1

Introduction

`collective.solr` integrates the Plone CMS with the Solr search engine.

Apache Solr is based on Lucene and is *the* enterprise open source search engine. It powers the search of sites like Twitter, the Apple and iTunes Stores, Wikipedia, Netflix and many more.

Solr does not only scale to any level of content, but provides rich search functionality, like faceting, geospatial search, suggestions, spelling corrections, indexing of binary formats and a whole variety of powerful tools to configure custom search solutions. It has integrated clustering and load-balancing to provide a high level of robustness.

`collective.solr` comes with a default configuration and setup of Solr that makes it extremely easy to get started, yet provides a vastly superior search quality compared to Plone's integrated text search based on `ZCTextIndex`.

CHAPTER 2

Current Status

The code is used in production in many sites and considered stable. This add-on can be installed in a Plone 4.1 (or later) site to enable indexing operations as well as searching (site and live search) using Solr. Doing so will not only significantly improve search quality and performance - especially for a large number of indexed objects, but also reduce the memory footprint of your Plone instance by allowing you to remove the `SearchableText`, `Description` and `Title` indexes from the catalog. In large sites with 100000 content objects and more, searches using `ZCTextIndex` often taken 10 seconds or more and require a good deal of memory from ZODB caches. Solr will typically answer these requests in 10ms to 50ms at which point network latency and the rendering speed of Plone's page templates are a more dominant factor.

CHAPTER 3

Contents

This documentation will describe all necessary information to setup and use Solr in combination with Plone.

3.1 Base Information how Solr and the Integration of Solr and Plone work

3.1.1 Architecture

When working with Solr it's good to keep some things about it in mind. This information is targeted at developers and integrators trying to use and extend Solr in their Plone projects.

3.1.1.1 Dependencies

Currently we depend on *collective.indexing* as a means to hook into the normal catalog machinery of Plone to detect content changes. *c.indexing* before version two had some persistent data structures that frequently caused problems when removing the add-on. These problems have been fixed in version two. Unfortunately *c.indexing* still has to hook the catalog machinery in various evil ways, as the machinery lacks the required hooks for its use-case. Going forward it is expected for *c.indexing* to be merged into the underlying *ZCatalog* implementation, at which point *collective.solr* can use those hooks directly.

3.1.1.2 Indexing

Solr is not transactional aware and does not support any kind of rollback or undo. We therefor only send data to Solr at the end of any successful request. This is done via collective.indexing, a transaction manager and an end request transaction hook. This means you won't see any changes done to content inside a request when doing Solr searches later on in the same request. Inside tests you need to either commit real transactions or otherwise flush the Solr connection. There's no transaction concept, so one request doing a search might get some results in its beginning, than a different request might add new information to Solr. If the first request is still running and does the same search again it might get different results taking the changes from the second request into account.

Solr is not a real time search engine. While there's work under way to make Solr capable of delivering real time results, there's currently always a certain delay up to some minutes from the time data is sent to Solr to when it is available in searches.

Search results are returned in Solr by distinct search threads. These search threads hold a great number of caches which are crucial for Solr to perform. When index or unindex operations are sent to Solr, it will keep those in memory until a commit is executed on its own search index. When a commit occurs, all search threads and thus all caches are thrown away and new threads are created reflecting the data after the commit. While there's a certain amount of cache data that is copied to the new search threads, this data has to be validated against the new index which takes some time. The *useColdSearcher* and *maxWarmingSearchers* options of the Solr recipe relate to this aspect. While cache data is copied over and validated for a new search thread, the searcher is *warming up*. If the warming up is not yet completed the searcher is considered to be *cold*.

In order to get real good performance out of Solr, we need to minimize the number of commits against the Solr index. We can achieve this by turning off *auto-commit* and instead use *commitWithin*. So we don't sent a *commit* to Solr at the end of each index/unindex request on the Plone side. Instead we tell Solr to commit the data to its index at most after a certain time interval. Values of 15 minutes to 1 minute work well for this interval. The larger you can make this interval, the better the performance of Solr will be, at the cost of search results lagging behind a bit. In this setup we also need to configure the *autoCommitMaxTime* option of the Solr server, as *commitWithin* only works for index but not unindex operations. Otherwise a large number of unindex operations without any index operations occurring could not be reflected in the index for a long time.

As a result of all the above, the Solr index and the Plone site will always have slightly diverging contents. If you use Solr to do searches you need to be aware of this, as you might get results for objects that no longer exist. So any *brain/getObject* call on the Plone side needs to have error handling code around it as the object might not be there anymore and traversing to it can throw an exception.

When adding new or deleting old content or changing the workflow state of it, you will also not see those actions reflected in searches right away, but only after a delay of at most the *commitWithin* interval. After a *commitWithin* operation is sent to Solr, any other operations happening during that time window will be executed after the first interval is over. So with a 15 minute interval, if document A is indexed at 5:15, B at 5:20 and C at 5:35, both A & B will be committed at 5:30 and C at 5:50.

3.1.1.3 Searching

Information retrieval is a complex science. We try to give a very brief explanation here, refer to the literature and documentation of Lucene/Solr for much more detailed information.

If you do searches in normal Plone, you have a search term and query the SearchableText index with it. The SearchableText is a simple concatenation of all searchable fields, by default title, description and the body text.

The default ZCTextIndex in Plone uses a simplified version of the Okapi BM25 algorithm described in papers in 1998. It uses two metrics to score documents:

- Term frequency: How often does a search term occur in a document
- Inverse document frequency: The inverse of in how many documents a term occurs. Terms only occurring in a few documents are scored higher than those occurring in many documents.

It calculates the sum of all scores, for every term common to the query and any document. So for a query with two terms, a document is likely to score higher if it contains both terms, except if one of them is a very common term and the other document contains the non-common term more often.

The similarity function used in Solr/Lucene uses a different algorithm, based on a combination of a boolean and vector space model, but taking the same underlying metrics into account. In addition to the term frequency and inverse document frequency Solr respects some more metrics:

- length normalization: The number of all terms in a field. Shorter fields contribute higher scores compared to long fields.

- boost values: There's a variety of boost values that can be applied, both index-time document boost values as well as boost values per search field or search term

In its pre 2.0 versions, collective.solr used a naive approach and mirrored the approach taken by ZCTextIndex. So it sent each search query as one query and matched it against the full SearchableText field inside Solr. By doing that Solr basically used the same algorithm as ZCTextIndex as it only had one field to match with the entire text in it. The only difference was the use of the length normalization, so shorter documents ranked higher than those with longer texts. This actually caused search quality to be worse, as you'd frequently find folders, links or otherwise rather empty documents. The Okapi BM25 implementation in ZCTextIndex deliberately ignores the document length for that reason.

In order to get good or better search quality from Solr, we have to query it in a different way. Instead of concatenating all fields into one big text, we need to preserve the individual fields and use their intrinsic importance. We get the main benefit be realizing that matches on the title and description are more important than matches on the body text or other fields in a document. collective.solr 2.0+ does exactly that by introducing a *search-pattern* to be used for text searches. In its default form it causes each query to work against the title, description and full searchable text fields and boosts the title by a high and the description by a medium value. The length normalization already provides an improvement for these fields, as the title is likely short, the description a bit longer and the full text even longer. By using explicit boost values the effect gets to be more pronounced.

If you do custom searches or want to include more fields into the full text search you need to keep the above in mind. Simply setting the *searchable* attribute on the schema of a field to *True* will only include it in the big searchable text stream. If you for example include a field containing tags, the simple tag names will likely 'drown' in the full body text. You might want to instead change the search pattern to include the field and potentially put a boost value on it - though it will be more important as it's likely to be extremely short. Similarly extracting the full text of binary files and simply appending them into the search stream might not be the best approach. You should rather index those in a separate field and then maybe use a boost value of less than one to make the field less important. Given two documents with the same content, one as a normal page and one as a binary file, you'll likely want to find the page first, as it's faster to access and read than the file.

There's a good number of other improvements you can do using query time and index time boost values. To provide index time boost values, you can provide a skin script called *solr_boost_index_values* which gets the object to be indexed and the data sent to Solr as arguments and returns a dictionary of field names to boost values for each document. The safest is to return a boost value for the empty string, which results in a document boost value. Field level boost values don't work with all searches, especially wildcard searches as done by most simple web searches. The index time boost allows you to implement policies like boosting certain content types over others, taking into account ratings or number of comments as a measure of user feedback or anything else that can be derived from each content item.

3.2 Installation, Setup and Usage of Solr Integration

3.2.1 Installation

3.2.1.1 Installing collective.solr for a Plone buildout / project

The following buildout configuration may be used to get started quickly:

```
[buildout]
extends =
  buildout.cfg
  https://raw.githubusercontent.com/collective/collective.solr/master/solr.cfg
  https://raw.githubusercontent.com/collective/collective.solr/master/solr-4.10.x.cfg

[instance]
eggs += collective.solr
```

After saving this to let's say `solr.cfg` the buildout can be run and the Solr server and Plone instance started:

```
$ python bootstrap.py
$ bin/buildout -c solr.cfg
...
$ bin/solr-instance start
$ bin/instance start
```

Next you should activate the `collective.solr (site search)` add-on in the add-on control panel of Plone. After activation you should review the settings in the new `Solr Settings` control panel. To index all your content in Solr you can call the provided maintenance view:

```
http://localhost:8080/plone/@@solr-maintenance/reindex
```

Creating the initial index can take some considerable time. A typical indexing rate for a Plone site running of a local disk is 20 index operations per second. While Solr scales to orders of magnitude more than that, the limiting factor is database access time in Plone.

If you have an existing site with a large volume of content, you can create an initial Solr index on a staging server or development machine, then rsync it over to the live machine, enable Solr and call *@@solr-maintenance/sync*. The sync will usually take just a couple of minutes for catching up with changes in the live database. You can also use this approach when making changes to the index structure or changing the settings of existing fields.

Note that the example `solr.cfg` is bound to change. Always copy the file to your local buildout. In general you should never rely on extending buildout config files from servers that aren't under your control.

3.2.2 Setup Solr

3.2.2.1 Solr Schema

Solr Field Types

Autocomplete suggestions with Solr

http://wiki.apache.org/solr/Suggester

Simple autocomplete configuration using the "Title" field (buildout.cfg):

```
additional-solrconfig =
  <searchComponent name="suggest" class="solr.SpellCheckComponent">
    <lst name="spellchecker">
      <str name="name">suggest</str>
      <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
      <str name="lookupImpl">org.apache.solr.spelling.suggest.fst.WFSTLookupFactory</
↪str>
      <str name="field">Title</str>
      <float name="threshold">0.005</float>
      <str name="buildOnCommit">true</str>
    </lst>
  </searchComponent>

  <requestHandler name="/autocomplete" class="org.apache.solr.handler.component.
↪SearchHandler">
     <lst name="defaults">
      <str name="spellcheck">true</str>
```

(continues on next page)

(continued from previous page)

```
      <str name="spellcheck.dictionary">suggest</str>
      <str name="spellcheck.count">10</str>
      <str name="spellcheck.onlyMorePopular">true</str>
     </lst>
     <arr name="components">
      <str>suggest</str>
     </arr>
  </requestHandler>
```

More complex example with custom field/filters:

```
index +=
    name:title_autocomplete type:text_auto indexed:true stored:true

additional-solrconfig =
  <searchComponent name="suggest" class="solr.SpellCheckComponent">
    <lst name="spellchecker">
      <str name="name">suggest</str>
      <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
      <str name="lookupImpl">org.apache.solr.spelling.suggest.fst.WFSTLookupFactory</
↪str>
      <str name="field">title_autocomplete</str>
      <float name="threshold">0.005</float>
      <str name="buildOnCommit">true</str>
    </lst>
  </searchComponent>

  <requestHandler name="/autocomplete" class="org.apache.solr.handler.component.
↪SearchHandler">
     <lst name="defaults">
      <str name="spellcheck">true</str>
      <str name="spellcheck.dictionary">suggest</str>
      <str name="spellcheck.count">10</str>
      <str name="spellcheck.onlyMorePopular">true</str>
     </lst>
     <arr name="components">
      <str>suggest</str>
     </arr>
  </requestHandler>

extra-field-types =
  <fieldType class="solr.TextField" name="text_auto">
    <analyzer>
      <tokenizer class="solr.WhitespaceTokenizerFactory"/>
      <filter class="solr.ShingleFilterFactory" maxShingleSize="4" outputUnigrams=
↪"true"/>
    </analyzer>
  </fieldType>

additional-schema-config =
  <copyField source="Title" dest="title_autocomplete" />
```

3.2.2.2 Solr Base Schema for Plone

3.2.3 Configuring collective.solr

3.2.3.1 Solr-Connection Configuration

ZCML Configuration (prefered)

The connections settings for Solr can be configured in ZCML and thus in buildout. This makes it easier when copying databases between multiple Zope instances with different Solr servers. Example:

```
zcml-additional =
    <configure xmlns:solr="http://namespaces.plone.org/solr">
        <solr:connection host="localhost" port="8983" base="/solr/plone"/>
   </configure>
```

TTW Configuration

3.2.3.2 TTW Configuration of Solr-Settings

3.2.4 Considerations for a production Setup

3.2.4.1 Java Settings

Make sure you are using a *server* version of Java in production. The output of:

```
$ java -version
```

should include *Java HotSpot(TM) Server VM* or *Java HotSpot(TM) 64-Bit Server VM*. You can force the Java VM into server mode by calling it with the *-server* command. Do not try to run Solr with versions of OpenJDK or other non-official Java versions. They tend to not work well or at all.

Depending on the size of your Solr index, you need to configure the Java VM to have enough memory. Good starting values are *-Xms128M -Xmx256M*, as a rule of thumb keep *Xmx* double the size of *Xms*.

You can configure these settings via the *java_opts* value in the *collective.recipe.solrinstance* recipe section like:

```
java_opts =
  -server
  -Xms128M
  -Xmx256M
```

3.2.4.2 Monitoring

Java has a general monitoring framework called JMX. You can use this to get a huge number of details about the Java process in general and Solr in particular. Some hints are at http://wiki.apache.org/solr/SolrJmx. The default *collective.recipe.solrinstance* config uses *<jmx />*, so we can use command line arguments to configure it. Our example *buildout/solr.cfg* includes all the relevant values in its *java_opts* variable.

To view all the available metrics, start Solr and then the *jconsole* command included in the Java SDK and connect to the local process named *start.jar*. Solr specific information is available from the MBeans tab under the *solr* section. For example you'll find *avgTimePerRequest* within *search/org.apache.solr.handler.component.SearchHandler* under *Attributes*.

If you want to integrate with munin, you can install the JMX plugin at: http://exchange.munin-monitoring.org/plugins/jmx/details

Follow its install instructions and tweak the included examples to query the information you want to track. To track the average time per search request, add a file called *solr_avg_query_time.conf* into */usr/share/munin/plugins* with the following contents:

```
graph_title Average Query Time
graph_vlabel ms
graph_category Solr

solr_average_query_time.label time per request
solr_average_query_time.jmxObjectName solr/:type=search,id=org.apache.solr.handler.
↪component.SearchHandler
solr_average_query_time.jmxAttributeName avgTimePerRequest
```

Then add a symlink to add the plugin:

```
$ ln -s /usr/share/munin/plugins/jmx_ /etc/munin/plugins/jmx_solr_avg_query_time
```

Point the jmx plugin to the Solr process, by opening */etc/munin/plugin-conf.d/munin-node.conf* and adding something like:

```
[jmx_*]
env.jmxurl service:jmx:rmi:///jndi/rmi://127.0.0.1:8984/jmxrmi
```

The host and port need to match those passed via *java_opts* to Solr. To check if the plugins are working do:

```
$ export jmxurl="service:jmx:rmi:///jndi/rmi://127.0.0.1:8984/jmxrmi"
$ cd /etc/munin/plugins
```

And call the plugin you configured directly, like for example:

```
$ ./solr_avg_query_time
solr_average_query_time.value NaN
```

We include a number of useful configurations inside the package, in the *collective/solr/munin_config* directory. You can copy all of them into the */usr/share/munin/plugins* directory and create the symlinks for all of them.

3.2.4.3 Replication

At this point Solr doesn't yet allow for a full fault tolerance setup. You can read more about the Solr Cloud effort which aims to provide this.

But we can setup a simple master/slave replication using Solr's built-in Solr Replication support, which is a first step in the right direction.

In order to use this, you can setup a Solr master server and give it some extra config:

```
[solr-instance]
additional-solrconfig =
  <requestHandler name="/replication" class="solr.ReplicationHandler" >
    <lst name="master">
      <str name="replicateAfter">commit</str>
      <str name="replicateAfter">startup</str>
```

(continues on next page)

(continued from previous page)

```
      <str name="replicateAfter">optimize</str>
    </lst>
  </requestHandler>
```

Then you can point one or multiple slave servers to the master. Assuming the master runs on *solr-master.domain.com* at port *8983*, we could write:

```
[solr-instance]
additional-solrconfig =
  <requestHandler name="/replication" class="solr.ReplicationHandler" >
    <lst name="slave">
      <str name="masterUrl">http://solr-master.domain.com:8983/solr/replication</str>
      <str name="pollInterval">00:00:30</str>
    </lst>
  </requestHandler>
```

A poll interval of 30 seconds should be fast enough without creating too much overhead.

At this point *collective.solr* does not yet have support for connecting to multiple servers and using the slaves as a fallback for querying. As there's no master-master setup yet, fault tolerance for index changes cannot be provided.

3.2.4.4 SolrCloud

You can read more about the Solr Cloud effort which aims to provide this.

3.3 Development

Releases can be found on the Python Package Index at http://pypi.python.org/pypi/collective.solr. The code and issue trackers can be found on GitHub at https://github.com/collective/collective.solr.

For outstanding issues and features remaining to be implemented please see the to-do list included in the package as well as it's issue tracker.

3.3.1 TODOs:

- Migrate tests to use plone.app.testing
- Migrate control panel to use plone.autoform and plone.app.registry
- support for date facets
- result iterator (with __len__ on results object)
- support for "navtree" and "level" arguments for path queries
- provide decorator for solr exceptions
- add signal handlers (see store.py)
- add a configurable queue limit for large transactions
- mapping from accessor name to attribute name, i.e. getTitle -> title, preferably via <copyField> tags in the solr schema

- evaluate http://www.gnuenterprise.org/~jcater/solr.py as a replacement (also see http://tinyurl.com/2zcogf)
- evaluate sunburnet as a replacement https://pypi.python.org/pypi/sunburnt
- evaluate mysolr as backend https://pypi.python.org/pypi/mysolr
- implement LocalParams to have a nicer facet view http://wiki.apache.org/solr/SimpleFacetParameters#Multi-Select_Faceting_and_LocalParams
- Use current search view and get rid of ancient search override
- Implement a push only and read only mode
- Play nice with eea.facetednavigation

3.3.2 Search widget:

The search widget in the @@search view is implemented using React. It provides the default Plone search behavior. It uses the search REST API provided by the plone.restapi package.

The sources and distribution files can be found in the static directory. The Webpack build files are also provided in case you need to improve, customize or extend the default features of the widget.

You can setup the widget development environment by:

```
$ cd src/collective/solr/static/
$ npm install
```

Then also given a collective.solr development instance running in port 8080 and solr running:

```
$ npm start
```

Then you can access to the URL http://localhost:3000

Once you've finished developing, you should create a bundle by:

```
$ npm run build
```

CHAPTER 4

Credits

This code was inspired by enfold.solr by Enfold Systems as well as work done at the Snow Sprint 2008. The *solr.py* module is based on the original python integration package from Solr itself.

Development was kindly sponsored by Elkjop and the Nordic Council and Nordic Council of Ministers.

CHAPTER 5

Contributors

- Hanno Schlichting (hannosh)
- Tom Gross (tomgross)
- Timo Stollenwerk (tisto)
- Manuel Reinhardt (reinhardt)
- Patrick Gerken (do3cc)
- Andreas Zeidler (witsch)
- Martijn Pieters (mjpieters)
- Carsten Senger (csenger)
- Andrea Cecchi (cekk)
- Florian Schulze (fschulze)
- Mauro Amico (mamico)
- Giacomo Spettoli (giacomos)
- (jkubaile)
- Luca Fabbri (keul)
- Witek (witekdev)
- Laurence Rowe (lrowe)
- JC Brand (jcbrand)
- Daniel Widerin (saily)
- Wolfgang Thomas (pysailor)
- Philip Bauer (pbauer)
- Cédric Messiant (cedricmessiant)
- Rodrigo (rristow)

- (tschorr)
- Alexander Pilz (pilz)
- Jean Jordaan (jean)
- Alexander Loechel (loechel)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search