Haystack Documentation

Release 2.5.0

Daniel Lindsley

May 24, 2017

Contents

Haystack provides modular search for Django. It features a unified, familiar API that allows you to plug in different search backends (such as Solr, Elasticsearch, Whoosh, Xapian, etc.) without having to modify your code.

Note: This documentation represents the current version of Haystack. For old versions of the documentation:

- v2.4.X: https://django-haystack.readthedocs.io/en/v2.4.1/
- v2.3.X: https://django-haystack.readthedocs.io/en/v2.3.0/
- v2.2.X: https://django-haystack.readthedocs.io/en/v2.2.0/
- v2.1.X: https://django-haystack.readthedocs.io/en/v2.1.0/
- v2.0.X: https://django-haystack.readthedocs.io/en/v2.0.0/
- v1.2.X: https://django-haystack.readthedocs.io/en/v1.2.7/
- v1.1.X: https://django-haystack.readthedocs.io/en/v1.1/

CHAPTER 1

Getting Started

If you're new to Haystack, you may want to start with these documents to get you up and running:

Getting Started with Haystack

Search is a topic of ever increasing importance. Users increasing rely on search to separate signal from noise and find what they're looking for quickly. In addition, search can provide insight into what things are popular (many searches), what things are difficult to find on the site and ways you can improve the site better.

To this end, Haystack tries to make integrating custom search as easy as possible while being flexible/powerful enough to handle more advanced use cases.

Haystack is a reusable app (that is, it relies only on its own code and focuses on providing just search) that plays nicely with both apps you control as well as third-party apps (such as `django.contrib.*`) without having to modify the sources.

Haystack also does pluggable backends (much like Django's database layer), so virtually all of the code you write ought to be portable between whichever search engine you choose.

Note: If you hit a stumbling block, there is both a mailing list and #haystack on irc.freenode.net to get help.

Note: You can participate in and/or track the development of Haystack by subscribing to the development mailing list.

This tutorial assumes that you have a basic familiarity with the various major parts of Django (models/forms/views/settings/URLconfs) and tailored to the typical use case. There are shortcuts available as well as hooks for much more advanced setups, but those will not be covered here.

For example purposes, we'll be adding search functionality to a simple note-taking application. Here is `myapp/models.py`:

```
from django.db import models
from django.contrib.auth.models import User


class Note(models.Model):
    user = models.ForeignKey(User)
    pub_date = models.DateTimeField()
    title = models.CharField(max_length=200)
    body = models.TextField()

    def __unicode__(self):
        return self.title
```

Finally, before starting with Haystack, you will want to choose a search backend to get started. There is a quick-start guide to *Installing Search Engines*, though you may want to defer to each engine's official instructions.

Installation

Use your favorite Python package manager to install the app from PyPI, e.g.

Example:

```
pip install django-haystack
```

Configuration

Add Haystack To `INSTALLED_APPS`

As with most Django applications, you should add Haystack to the `INSTALLED_APPS` within your settings file (usually `settings.py`).

Example:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',

    # Added.
    'haystack',

    # Then your usual apps...
    'blog',
]
```

Modify Your `settings.py`

Within your `settings.py`, you'll need to add a setting to indicate where your site configuration file will live and which backend to use, as well as other settings for that backend.

`HAYSTACK_CONNECTIONS` is a required setting and should be at least one of the following:

Solr

Example:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://127.0.0.1:8983/solr'
        # ...or for multicore...
        # 'URL': 'http://127.0.0.1:8983/solr/mysite',
    },
}
```

Elasticsearch

Example (ElasticSearch 1.x):

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.elasticsearch_backend.ElasticsearchSearchEngine',
        'URL': 'http://127.0.0.1:9200/',
        'INDEX_NAME': 'haystack',
    },
}
```

Example (ElasticSearch 2.x):

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.elasticsearch2_backend.Elasticsearch2SearchEngine
↪',
        'URL': 'http://127.0.0.1:9200/',
        'INDEX_NAME': 'haystack',
    },
}
```

Whoosh

Requires setting `PATH` to the place on your filesystem where the Whoosh index should be located. Standard warnings about permissions and keeping it out of a place your webserver may serve documents out of apply.

Example:

```
import os
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': os.path.join(os.path.dirname(__file__), 'whoosh_index'),
    },
}
```

Xapian

First, install the Xapian backend (via http://github.com/notanumber/xapian-haystack/tree/master) per the instructions included with the backend.

Requires setting `PATH` to the place on your filesystem where the Xapian index should be located. Standard warnings about permissions and keeping it out of a place your webserver may serve documents out of apply.

Example:

```
import os
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'xapian_backend.XapianEngine',
        'PATH': os.path.join(os.path.dirname(__file__), 'xapian_index'),
    },
}
```

Simple

The `simple` backend using very basic matching via the database itself. It's not recommended for production use but it will return results.

Warning: This backend does *NOT* work like the other backends do. Data preparation does nothing & advanced filtering calls do not work. You really probably don't want this unless you're in an environment where you just want to silence Haystack.

Example:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.simple_backend.SimpleEngine',
    },
}
```

Handling Data

Creating `SearchIndexes`

`SearchIndex` objects are the way Haystack determines what data should be placed in the search index and handles the flow of data in. You can think of them as being similar to Django `Models` or `Forms` in that they are field-based and manipulate/store data.

You generally create a unique `SearchIndex` for each type of `Model` you wish to index, though you can reuse the same `SearchIndex` between different models if you take care in doing so and your field names are very standardized.

To build a `SearchIndex`, all that's necessary is to subclass both `indexes.SearchIndex` & `indexes.Indexable`, define the fields you want to store data with and define a `get_model` method.

We'll create the following `NoteIndex` to correspond to our `Note` model. This code generally goes in a `search_indexes.py` file within the app it applies to, though that is not required. This allows Haystack to automatically pick it up. The `NoteIndex` should look like:

```
import datetime
from haystack import indexes
from myapp.models import Note


class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        """Used when the entire index for model is updated."""
        return self.get_model().objects.filter(pub_date__lte=datetime.datetime.now())
```

Every `SearchIndex` requires there be one (and only one) field with `document=True`. This indicates to both Haystack and the search engine about which field is the primary field for searching within.

Warning: When you choose a `document=True` field, it should be consistently named across all of your `SearchIndex` classes to avoid confusing the backend. The convention is to name this field `text`.

There is nothing special about the `text` field name used in all of the examples. It could be anything; you could call it `pink_polka_dot` and it won't matter. It's simply a convention to call it `text`.

Additionally, we're providing `use_template=True` on the `text` field. This allows us to use a data template (rather than error-prone concatenation) to build the document the search engine will index. You'll need to create a new template inside your template directory called `search/indexes/myapp/note_text.txt` and place the following inside:

```
{{ object.title }}
{{ object.user.get_full_name }}
{{ object.body }}
```

In addition, we added several other fields (`author` and `pub_date`). These are useful when you want to provide additional filtering options. Haystack comes with a variety of `SearchField` classes to handle most types of data.

A common theme is to allow admin users to add future content but have it not display on the site until that future date is reached. We specify a custom `index_queryset` method to prevent those future items from being indexed.

Setting Up The Views

Add The `SearchView` To Your URLconf

Within your URLconf, add the following line:

```
url(r'^search/', include('haystack.urls')),
```

This will pull in the default URLconf for Haystack. It consists of a single URLconf that points to a `SearchView` instance. You can change this class's behavior by passing it any of several keyword arguments or override it entirely with your own view.

Search Template

Your search template (`search/search.html` for the default case) will likely be very simple. The following is enough to get going (your template/block names will likely differ):

```
{% extends 'base.html' %}

{% block content %}
    <h2>Search</h2>

    <form method="get" action=".">
        <table>
            {{ form.as_table }}
            <tr>
                <td> </td>
                <td>
                    <input type="submit" value="Search">
                </td>
            </tr>
        </table>

        {% if query %}
            <h3>Results</h3>

            {% for result in page.object_list %}
                <p>
                    <a href="{{ result.object.get_absolute_url }}">{{ result.object.
↪title }}</a>
                </p>
            {% empty %}
                <p>No results found.</p>
            {% endfor %}

            {% if page.has_previous or page.has_next %}
                <div>
                    {% if page.has_previous %}<a href="?q={{ query }}&amp;page={{␣
↪page.previous_page_number }}">{% endif %}&laquo; Previous{% if page.has_previous %}
↪</a>{% endif %}
                    |
                    {% if page.has_next %}<a href="?q={{ query }}&amp;page={{ page.
↪next_page_number }}">{% endif %}Next &raquo;{% if page.has_next %}</a>{% endif %}
                </div>
            {% endif %}
        {% else %}
            {# Show some example queries to run, maybe query syntax, something else?
↪#}
        {% endif %}
    </form>
{% endblock %}
```

Note that the `page.object_list` is actually a list of `SearchResult` objects instead of individual models. These objects have all the data returned from that record within the search index as well as score. They can also directly access the model for the result via `{{ result.object }}`. So the `{{ result.object.title }}` uses the actual `Note` object in the database and accesses its `title` field.

Reindex

The final step, now that you have everything setup, is to put your data in from your database into the search index. Haystack ships with a management command to make this process easy.

Note: If you're using the Solr backend, you have an extra step. Solr's configuration is XML-based, so you'll need to manually regenerate the schema. You should run `./manage.py build_solr_schema` first, drop the XML output in your Solr's `schema.xml` file and restart your Solr server.

Simply run `./manage.py rebuild_index`. You'll get some totals of how many models were processed and placed in the index.

Note: Using the standard `SearchIndex`, your search index content is only updated whenever you run either `./manage.py update_index` or start afresh with `./manage.py rebuild_index`.

You should cron up a `./manage.py update_index` job at whatever interval works best for your site (using `--age=<num_hours>` reduces the number of things to update).

Alternatively, if you have low traffic and/or your search engine can handle it, the `RealtimeSignalProcessor` automatically handles updates/deletes for you.

Complete!

You can now visit the search section of your site, enter a search query and receive search results back for the query! Congratulations!

What's Next?

This tutorial just scratches the surface of what Haystack provides. The `SearchQuerySet` is the underpinning of all search in Haystack and provides a powerful, `QuerySet`-like API (see *SearchQuerySet API*). You can use much more complicated `SearchForms`/`SearchViews` to give users a better UI (see *Views & Forms*). And the *Best Practices* provides insight into non-obvious or advanced usages of Haystack.

Views & Forms

Note: As of version 2.4 the views in `haystack.views.SearchView` are deprecated in favor of the new generic views in `haystack.generic_views.SearchView` which use the standard Django class-based views which are available in every version of Django which is supported by Haystack.

Haystack comes with some default, simple views & forms as well as some django-style views to help you get started and to cover the common cases. Included is a way to provide:

- Basic, query-only search.
- Search by models.
- Search with basic highlighted results.
- Faceted search.

- Search by models with basic highlighted results.

Most processing is done by the forms provided by Haystack via the `search` method. As a result, all but the faceted types (see *Faceting*) use the standard `SearchView`.

There is very little coupling between the forms & the views (other than relying on the existence of a `search` method on the form), so you may interchangeably use forms and/or views anywhere within your own code.

Forms

SearchForm

The most basic of the form types, this form consists of a single field, the `q` field (for query). Upon searching, the form will take the cleaned contents of the `q` field and perform an `auto_query` on either the custom `SearchQuerySet` you provide or off a default `SearchQuerySet`.

To customize the `SearchQuerySet` the form will use, pass it a `searchqueryset` parameter to the constructor with the `SearchQuerySet` you'd like to use. If using this form in conjunction with a `SearchView`, the form will receive whatever `SearchQuerySet` you provide to the view with no additional work needed.

The `SearchForm` also accepts a `load_all` parameter (`True` or `False`), which determines how the database is queried when iterating through the results. This also is received automatically from the `SearchView`.

All other forms in Haystack inherit (either directly or indirectly) from this form.

HighlightedSearchForm

Identical to the `SearchForm` except that it tags the `highlight` method on to the end of the `SearchQuerySet` to enable highlighted results.

ModelSearchForm

This form adds new fields to form. It iterates through all registered models for the current `SearchSite` and provides a checkbox for each one. If no models are selected, all types will show up in the results.

HighlightedModelSearchForm

Identical to the `ModelSearchForm` except that it tags the `highlight` method on to the end of the `SearchQuerySet` to enable highlighted results on the selected models.

FacetedSearchForm

Identical to the `SearchForm` except that it adds a hidden `selected_facets` field onto the form, allowing the form to narrow the results based on the facets chosen by the user.

Creating Your Own Form

The simplest way to go about creating your own form is to inherit from `SearchForm` (or the desired parent) and extend the `search` method. By doing this, you save yourself most of the work of handling data correctly and stay API compatible with the `SearchView`.

For example, let's say you're providing search with a user-selectable date range associated with it. You might create a form that looked as follows:

```
from django import forms
from haystack.forms import SearchForm


class DateRangeSearchForm(SearchForm):
    start_date = forms.DateField(required=False)
    end_date = forms.DateField(required=False)

    def search(self):
        # First, store the SearchQuerySet received from other processing.
        sqs = super(DateRangeSearchForm, self).search()

        if not self.is_valid():
            return self.no_query_found()

        # Check to see if a start_date was chosen.
        if self.cleaned_data['start_date']:
            sqs = sqs.filter(pub_date__gte=self.cleaned_data['start_date'])

        # Check to see if an end_date was chosen.
        if self.cleaned_data['end_date']:
            sqs = sqs.filter(pub_date__lte=self.cleaned_data['end_date'])

        return sqs
```

This form adds two new fields for (optionally) choosing the start and end dates. Within the `search` method, we grab the results from the parent form's processing. Then, if a user has selected a start and/or end date, we apply that filtering. Finally, we simply return the `SearchQuerySet`.

Views

Note: As of version 2.4 the views in `haystack.views.SearchView` are deprecated in favor of the new generic views in `haystack.generic_views.SearchView` which use the standard Django class-based views which are available in every version of Django which is supported by Haystack.

New Django Class Based Views

New in version 2.4.0.

The views in `haystack.generic_views.SearchView` inherit from Django's standard FormView. The example views can be customized like any other Django class-based view as demonstrated in this example which filters the search results in `get_queryset`:

```
# views.py
from datetime import date

from haystack.generic_views import SearchView

class MySearchView(SearchView):
    """My custom search view."""
```

```
    def get_queryset(self):
        queryset = super(MySearchView, self).get_queryset()
        # further filter queryset based on some set of criteria
        return queryset.filter(pub_date__gte=date(2015, 1, 1))

    def get_context_data(self, *args, **kwargs):
        context = super(MySearchView, self).get_context_data(*args, **kwargs)
        # do something
        return context

# urls.py

urlpatterns = [
    url(r'^/search/?$', MySearchView.as_view(), name='search_view'),
]
```

Upgrading

Upgrading from basic usage of the old-style views to new-style views is usually as simple as:

1. Create new views under `views.py` subclassing `haystack.generic_views.SearchView` or `haystack.generic_views.FacetedSearchView`
2. Move all parameters of your old-style views from your `urls.py` to attributes on your new views. This will require renaming `searchqueryset` to `queryset` and `template` to `template_name`
3. Review your templates and replace the `page` variable with `page_obj`

Here's an example:

```
### old-style views...
# urls.py

sqs = SearchQuerySet().filter(author='john')

urlpatterns = [
    url(r'^$', SearchView(
        template='my/special/path/john_search.html',
        searchqueryset=sqs,
        form_class=SearchForm
    ), name='haystack_search'),
]

### new-style views...
# views.py

class JohnSearchView(SearchView):
    template_name = 'my/special/path/john_search.html'
    queryset = SearchQuerySet().filter(author='john')
    form_class = SearchForm

# urls.py
from myapp.views import JohnSearchView

urlpatterns = [
    url(r'^$', JohnSearchView.as_view(), name='haystack_search'),
]
```

If your views overrode methods on the old-style SearchView, you will need to refactor those methods to the equivalents on Django's generic views. For example, if you previously used `extra_context()` to add additional template variables or preprocess the values returned by Haystack, that code would move to `get_context_data`

Old Method	New Method
`extra_context()`	get_context_data()
`create_response()`	dispatch() or `get()` and `post()`
`get_query()`	get_queryset()

Old-Style Views

Deprecated since version 2.4.0.

Haystack comes bundled with three views, the class-based views (`SearchView` & `FacetedSearchView`) and a traditional functional view (`basic_search`).

The class-based views provide for easy extension should you need to alter the way a view works. Except in the case of faceting (again, see *Faceting*), the `SearchView` works interchangeably with all other forms provided by Haystack.

The functional view provides an example of how Haystack can be used in more traditional settings or as an example of how to write a more complex custom view. It is also thread-safe.

SearchView(template=None, load_all=True, form_class=None, searchqueryset=None, results_per_page=None)

The `SearchView` is designed to be easy/flexible enough to override common changes as well as being internally abstracted so that only altering a specific portion of the code should be easy to do.

Without touching any of the internals of the `SearchView`, you can modify which template is used, which form class should be instantiated to search with, what `SearchQuerySet` to use in the event you wish to pre-filter the results. what `Context`-style object to use in the response and the `load_all` performance optimization to reduce hits on the database. These options can (and generally should) be overridden at the URLconf level. For example, to have a custom search limited to the 'John' author, displaying all models to search by and specifying a custom template (`my/special/path/john_search.html`), your URLconf should look something like:

```
from django.conf.urls import url
from haystack.forms import ModelSearchForm
from haystack.query import SearchQuerySet
from haystack.views import SearchView

sqs = SearchQuerySet().filter(author='john')

# Without threading...
urlpatterns = [
    url(r'^$', SearchView(
        template='my/special/path/john_search.html',
        searchqueryset=sqs,
        form_class=SearchForm
    ), name='haystack_search'),
]

# With threading...
from haystack.views import SearchView, search_view_factory

urlpatterns = [
    url(r'^$', search_view_factory(
        view_class=SearchView,
```

```
        template='my/special/path/john_search.html',
        searchqueryset=sqs,
        form_class=ModelSearchForm
    ), name='haystack_search'),
]
```

Warning: The standard `SearchView` is not thread-safe. Use the `search_view_factory` function, which returns thread-safe instances of `SearchView`.

By default, if you don't specify a `form_class`, the view will use the `haystack.forms.ModelSearchForm` form.

Beyond this customizations, you can create your own `SearchView` and extend/override the following methods to change the functionality.

`__call__(self, request)`

Generates the actual response to the search.

Relies on internal, overridable methods to construct the response. You generally should avoid altering this method unless you need to change the flow of the methods or to add a new method into the processing.

`build_form(self, form_kwargs=None)`

Instantiates the form the class should use to process the search query.

Optionally accepts a dictionary of parameters that are passed on to the form's `__init__`. You can use this to lightly customize the form.

You should override this if you write a custom form that needs special parameters for instantiation.

`get_query(self)`

Returns the query provided by the user.

Returns an empty string if the query is invalid. This pulls the cleaned query from the form, via the `q` field, for use elsewhere within the `SearchView`. This is used to populate the `query` context variable.

`get_results(self)`

Fetches the results via the form.

Returns an empty list if there's no query to search with. This method relies on the form to do the heavy lifting as much as possible.

`build_page(self)`

Paginates the results appropriately.

In case someone does not want to use Django's built-in pagination, it should be a simple matter to override this method to do what they would like.

extra_context(self)

Allows the addition of more context variables as needed. Must return a dictionary whose contents will add to or overwrite the other variables in the context.

create_response(self)

Generates the actual HttpResponse to send back to the user. It builds the page, creates the context and renders the response for all the aforementioned processing.

basic_search(request, template='search/search.html', load_all=True, form_class=ModelSearchForm, searchqueryset=None, extra_context=None, results_per_page=None)

The `basic_search` tries to provide most of the same functionality as the class-based views but resembles a more traditional generic view. It's both a working view if you prefer not to use the class-based views as well as a good starting point for writing highly custom views.

Since it is all one function, the only means of extension are passing in kwargs, similar to the way generic views work.

Creating Your Own View

As with the forms, inheritance is likely your best bet. In this case, the `FacetedSearchView` is a perfect example of how to extend the existing `SearchView`. The complete code for the `FacetedSearchView` looks like:

```
class FacetedSearchView(SearchView):
    def extra_context(self):
        extra = super(FacetedSearchView, self).extra_context()

        if self.results == []:
            extra['facets'] = self.form.search().facet_counts()
        else:
            extra['facets'] = self.results.facet_counts()

        return extra
```

It updates the name of the class (generally for documentation purposes) and adds the facets from the `SearchQuerySet` to the context as the `facets` variable. As with the custom form example above, it relies on the parent class to handle most of the processing and extends that only where needed.

Template Tags

Haystack comes with a couple common template tags to make using some of its special features available to templates.

highlight

Takes a block of text and highlights words from a provided query within that block of text. Optionally accepts arguments to provide the HTML tag to wrap highlighted word in, a CSS class to use with the tag and a maximum length of the blurb in characters.

The defaults are `span` for the HTML tag, `highlighted` for the CSS class and 200 characters for the excerpt.

Syntax:

```
{% highlight <text_block> with <query> [css_class "class_name"] [html_tag "span"]␣
↪[max_length 200] %}
```

Example:

```
# Highlight summary with default behavior.
{% highlight result.summary with query %}

# Highlight summary but wrap highlighted words with a div and the
# following CSS class.
{% highlight result.summary with query html_tag "div" css_class "highlight_me_please"
↪%}

# Highlight summary but only show 40 words.
{% highlight result.summary with query max_length 40 %}
```

The highlighter used by this tag can be overridden as needed. See the *Highlighting* documentation for more information.

more_like_this

Fetches similar items from the search index to find content that is similar to the provided model's content.

Note: This requires a backend that has More Like This built-in.

Syntax:

```
{% more_like_this model_instance as varname [for app_label.model_name,app_label.model_
↪name,...] [limit n] %}
```

Example:

```
# Pull a full SearchQuerySet (lazy loaded) of similar content.
{% more_like_this entry as related_content %}

# Pull just the top 5 similar pieces of content.
{% more_like_this entry as related_content limit 5  %}

# Pull just the top 5 similar entries or comments.
{% more_like_this entry as related_content for "blog.entry,comments.comment" limit 5
↪%}
```

This tag behaves exactly like `SearchQuerySet.more_like_this`, so all notes in that regard apply here as well.

Glossary

Search is a domain full of its own jargon and definitions. As this may be an unfamiliar territory to many developers, what follows are some commonly used terms and what they mean.

Engine An engine, for the purposes of Haystack, is a third-party search solution. It might be a full service (i.e. Solr) or a library to build an engine with (i.e. Whoosh)

Index The datastore used by the engine is called an index. Its structure can vary wildly between engines but commonly they resemble a document store. This is the source of all information in Haystack.

Document A document is essentially a record within the index. It usually contains at least one blob of text that serves as the primary content the engine searches and may have additional data hung off it.

Corpus A term for a collection of documents. When talking about the documents stored by the engine (rather than the technical implementation of the storage), this term is commonly used.

Field Within the index, each document may store extra data with the main content as a field. Also sometimes called an attribute, this usually represents metadata or extra content about the document. Haystack can use these fields for filtering and display.

Term A term is generally a single word (or word-like) string of characters used in a search query.

Stemming A means of determining if a word has any root words. This varies by language, but in English, this generally consists of removing plurals, an action form of the word, et cetera. For instance, in English, 'giraffes' would stem to 'giraffe'. Similarly, 'exclamation' would stem to 'exclaim'. This is useful for finding variants of the word that may appear in other documents.

Boost Boost provides a means to take a term or phrase from a search query and alter the relevance of a result based on if that term is found in the result, a form of weighting. For instance, if you wanted to more heavily weight results that included the word 'zebra', you'd specify a boost for that term within the query.

More Like This Incorporating techniques from information retrieval and artificial intelligence, More Like This is a technique for finding other documents within the index that closely resemble the document in question. This is useful for programmatically generating a list of similar content for a user to browse based on the current document they are viewing.

Faceting Faceting is a way to provide insight to the user into the contents of your corpus. In its simplest form, it is a set of document counts returned with results when performing a query. These counts can be used as feedback for the user, allowing the user to choose interesting aspects of their search results and "drill down" into those results.

An example might be providing a facet on an `author` field, providing back a list of authors and the number of documents in the index they wrote. This could be presented to the user with a link, allowing the user to click and narrow their original search to all results by that author.

Management Commands

Haystack comes with several management commands to make working with Haystack easier.

`clear_index`

The `clear_index` command wipes out your entire search index. Use with caution. In addition to the standard management command options, it accepts the following arguments:

`--noinput`: If provided, the interactive prompts are skipped and the index is unceremoniously wiped out.

`--verbosity`: Accepted but ignored.

`--using`: Update only the named backend (can be used multiple times). By default, all backends will be updated.

`--nocommit`: If provided, it will pass commit=False to the backend. This means that the update will not become immediately visible and will depend on another explicit commit or the backend's commit strategy to complete the update.

By default, this is an **INTERACTIVE** command and assumes that you do **NOT** wish to delete the entire index.

Note: The `--nocommit` argument is only supported by the Solr backend.

Warning: Depending on the backend you're using, this may simply delete the entire directory, so be sure your `HAYSTACK_CONNECTIONS[<alias>]['PATH']` setting is correctly pointed at just the index directory.

update_index

Note: If you use the `--start/--end` flags on this command, you'll need to install dateutil to handle the datetime parsing.

The `update_index` command will freshen all of the content in your index. It iterates through all indexed models and updates the records in the index. In addition to the standard management command options, it accepts the following arguments:

`--age`: Number of hours back to consider objects new. Useful for nightly reindexes (`--age=24`). Requires `SearchIndexes` to implement the `get_updated_field` method. Default is `None`.

`--start`: The start date for indexing within. Can be any dateutil-parsable string, recommended to be YYYY-MM-DDTHH:MM:SS. Requires `SearchIndexes` to implement the `get_updated_field` method. Default is `None`.

`--end`: The end date for indexing within. Can be any dateutil-parsable string, recommended to be YYYY-MM-DDTHH:MM:SS. Requires `SearchIndexes` to implement the `get_updated_field` method. Default is `None`.

`--batch-size`: Number of items to index at once. Default is 1000.

`--remove`: Remove objects from the index that are no longer present in the database.

`--workers`: Allows for the use multiple workers to parallelize indexing. Requires `multiprocessing`.

`--verbosity`: If provided, dumps out more information about what's being done.

- `0` = No output
- `1` = Minimal output describing what models were indexed and how many records.
- `2` = Full output, including everything from `1` plus output on each batch that is indexed, which is useful when debugging.

`--using`: Update only the named backend (can be used multiple times). By default, all backends will be updated.

`--nocommit`: If provided, it will pass commit=False to the backend. This means that the updates will not become immediately visible and will depend on another explicit commit or the backend's commit strategy to complete the update.

Note: The `--nocommit` argument is only supported by the Solr and ElasticSearch backends.

Examples:

```
# Update everything.
./manage.py update_index --settings=settings.prod

# Update everything with lots of information about what's going on.
./manage.py update_index --settings=settings.prod --verbosity=2

# Update everything, cleaning up after deleted models.
./manage.py update_index --remove --settings=settings.prod

# Update everything changed in the last 2 hours.
./manage.py update_index --age=2 --settings=settings.prod

# Update everything between Dec. 1, 2011 & Dec 31, 2011
./manage.py update_index --start='2011-12-01T00:00:00' --end='2011-12-31T23:59:59' --
↪settings=settings.prod

# Update just a couple apps.
./manage.py update_index blog auth comments --settings=settings.prod

# Update just a single model (in a complex app).
./manage.py update_index auth.User --settings=settings.prod

# Crazy Go-Nuts University
./manage.py update_index events.Event media news.Story --start='2011-01-01T00:00:00 --
↪remove --using=hotbackup --workers=12 --verbosity=2 --settings=settings.prod
```

Note: This command *ONLY* updates records in the index. It does *NOT* handle deletions unless the `--remove` flag is provided. You might consider a queue consumer if the memory requirements for `--remove` don't fit your needs. Alternatively, you can use the `RealtimeSignalProcessor`, which will automatically handle deletions.

rebuild_index

A shortcut for `clear_index` followed by `update_index`. It accepts any/all of the arguments of the following arguments:

`--age`: Number of hours back to consider objects new. Useful for nightly reindexes (`--age=24`). Requires `SearchIndexes` to implement the `get_updated_field` method.

`--batch-size`: Number of items to index at once. Default is 1000.

`--site`: The site object to use when reindexing (like *search_sites.mysite*).

`--noinput`: If provided, the interactive prompts are skipped and the index is unceremoniously wiped out.

`--remove`: Remove objects from the index that are no longer present in the database.

`--verbosity`: If provided, dumps out more information about what's being done.

- `0` = No output
- `1` = Minimal output describing what models were indexed and how many records.
- `2` = Full output, including everything from `1` plus output on each batch that is indexed, which is useful when debugging.

`--using`: Update only the named backend (can be used multiple times). By default, all backends will be updated.

--nocommit: If provided, it will pass commit=False to the backend. This means that the update will not become immediately visible and will depend on another explicit commit or the backend's commit strategy to complete the update.

For when you really, really want a completely rebuilt index.

build_solr_schema

Once all of your `SearchIndex` classes are in place, this command can be used to generate the XML schema Solr needs to handle the search data. Generates a Solr schema and solrconfig file that reflects the indexes using templates under a Django template dir 'search_configuration/*.xml'. If none are found, then provides defaults suitable for Solr 6.4.

It accepts the following arguments:

--filename: If provided, renders schema.xml from the template directory directly to a file instead of stdout. Does not render solrconfig.xml

--using: Update only the named backend (can be used multiple times). By default all backends will be updated.

--configure-directory: If provided, attempts to configure a core located in the given directory by removing the `managed-schema.xml` (renaming if it exists), configuring the core by rendering the `schema.xml` and `solrconfig.xml` templates provided in the Django project's `TEMPLATE_DIR/search_configuration` directories.

--reload-core: If provided, attempts to automatically reload the solr core via the urls in the `URL` and `ADMIN_URL` settings of the Solr entry in `HAYSTACK_CONNECTIONS`. Both *must* be provided.

Note: `build_solr_schema --configure-directory=<dir>` can be used in isolation to drop configured files anywhere one might want for staging to one or more solr instances through arbitrary means. It will render all template files in the directory into the `configure-directory`

`build_solr_schema --configure-directory=<dir> --reload-core` can be used together to reconfigure and reload a core located on a filesystem accessible to Django in a one-shot mechanism with no further requirements (assuming there are no errors in the template or configuration)

Note: `build_solr_schema` uses templates to generate the output files. Haystack provides default templates for `schema.xml` and `solrconfig.xml` that are solr 6.5 compatible using some sensible defaults. If you would like to provide your own template, you will need to place it in `search_configuration/` inside a directory specified by your app's template directories settings. Examples:

```
/myproj/myapp/templates/search_configuration/schema.xml
/myproj/myapp/templates/search_configuration/sorlconfig.xml
/myproj/myapp/templates/search_configuration/otherfile.xml
# ...or...
/myproj/templates/search_configuration/schema.xml
/myproj/templates/search_configuration/sorlconfig.xml
/myproj/myapp/templates/search_configuration/otherfile.xml
```

Warning: This command does NOT automatically update the `schema.xml` file for you all by itself. You must use –filename or –configure-directory to achieve this.

haystack_info

Provides some basic information about how Haystack is setup and what models it is handling. It accepts no arguments. Useful when debugging or when using Haystack-enabled third-party apps.

(In)Frequently Asked Questions

What is Haystack?

Haystack is meant to be a portable interface to a search engine of your choice. Some might call it a search framework, an abstraction layer or what have you. The idea is that you write your search code once and should be able to freely switch between backends as your situation necessitates.

Why should I consider using Haystack?

Haystack is targeted at the following use cases:

- If you want to feature search on your site and search solutions like Google or Yahoo search don't fit your needs.
- If you want to be able to customize your search and search on more than just the main content.
- If you want to have features like drill-down (faceting) or "More Like This".
- If you want a interface that is non-search engine specific, allowing you to change your mind later without much rewriting.

When should I not be using Haystack?

- Non-Model-based data. If you just want to index random data (flat files, alternate sources, etc.), Haystack isn't a good solution. Haystack is very `Model`-based and doesn't work well outside of that use case.
- Ultra-high volume. Because of the very nature of Haystack (abstraction layer), there's more overhead involved. This makes it portable, but as with all abstraction layers, you lose a little performance. You also can't take full advantage of the exact feature-set of your search engine. This is the price of pluggable backends.

Why was Haystack created when there are so many other search options?

The proliferation of search options in Django is a relatively recent development and is actually one of the reasons for Haystack's existence. There are too many options that are only partial solutions or are too engine specific.

Further, most use an unfamiliar API and documentation is lacking in most cases.

Haystack is an attempt to unify these efforts into one solution. That's not to say there should be no alternatives, but Haystack should provide a good solution to 80%+ of the search use cases out there.

What's the history behind Haystack?

Haystack started because of my frustration with the lack of good search options (before many other apps came out) and as the result of extensive use of Djangosearch. Djangosearch was a decent solution but had a number of shortcomings, such as:

- Tied to the models.py, so you'd have to modify the source of third-party (or django.contrib) apps in order to effectively use it.

- All or nothing approach to indexes. So all indexes appear on all sites and in all places.
- Lack of tests.
- Lack of documentation.
- Uneven backend implementations.

The initial idea was to simply fork Djangosearch and improve on these (and other issues). However, after stepping back, I decided to overhaul the entire API (and most of the underlying code) to be more representative of what I would want as an end-user. The result was starting afresh and reusing concepts (and some code) from Djangosearch as needed.

As a result of this heritage, you can actually still find some portions of Djangosearch present in Haystack (especially in the `SearchIndex` and `SearchBackend` classes) where it made sense. The original authors of Djangosearch are aware of this and thus far have seemed to be fine with this reuse.

Why doesn't <search engine X> have a backend included in Haystack?

Several possibilities on this.

1. Licensing

   A common problem is that the Python bindings for a specific engine may have been released under an incompatible license. The goal is for Haystack to remain BSD licensed and importing bindings with an incompatible license can technically convert the entire codebase to that license. This most commonly occurs with GPL'ed bindings.

2. Lack of time

   The search engine in question may be on the list of backends to add and we simply haven't gotten to it yet. We welcome patches for additional backends.

3. Incompatible API

   In order for an engine to work well with Haystack, a certain baseline set of features is needed. This is often an issue when the engine doesn't support ranged queries or additional attributes associated with a search record.

4. We're not aware of the engine

   If you think we may not be aware of the engine you'd like, please tell us about it (preferably via the group - http://groups.google.com/group/django-haystack/). Be sure to check through the backends (in case it wasn't documented) and search the history on the group to minimize duplicates.

Sites Using Haystack

The following sites are a partial list of people using Haystack. I'm always interested in adding more sites, so please find me (`daniellindsley`) via IRC or the mailing list thread.

LJWorld/Lawrence.com/KUSports

For all things search-related.

Using: Solr

- http://www2.ljworld.com/search/
- http://www2.ljworld.com/search/vertical/news.story/

- http://www2.ljworld.com/marketplace/
- http://www.lawrence.com/search/
- http://www.kusports.com/search/

AltWeeklies

Providing an API to story aggregation.

Using: Whoosh

- http://www.northcoastjournal.com/altweeklies/documentation/

Trapeze

Various projects.

Using: Xapian

- http://www.trapeze.com/
- http://www.windmobile.ca/
- http://www.bonefishgrill.com/
- http://www.canadiantire.ca/ (Portions of)

Vickerey.com

For (really well done) search & faceting.

Using: Solr

- http://store.vickerey.com/products/search/

Eldarion

Various projects.

Using: Solr

- http://eldarion.com/

Sunlight Labs

For general search.

Using: Whoosh & Solr

- http://sunlightlabs.com/
- http://subsidyscope.com/

NASA

For general search.

Using: Solr

- An internal site called SMD Spacebook 1.1.
- http://science.nasa.gov/

AllForLocal

For general search.

- http://www.allforlocal.com/

HUGE

Various projects.

Using: Solr

- http://hugeinc.com/
- http://houselogic.com/

Brick Design

For search on Explore.

Using: Solr

- http://bricksf.com/
- http://explore.org/

Winding Road

For general search.

Using: Solr

- http://www.windingroad.com/

Reddit

For Reddit Gifts.

Using: Whoosh

- http://redditgifts.com/

Pegasus News

For general search.

Using: Xapian

- http://www.pegasusnews.com/

Rampframe

For general search.

Using: Xapian

- http://www.rampframe.com/

Forkinit

For general search, model-specific search and suggestions via MLT.

Using: Solr

- http://forkinit.com/

Structured Abstraction

For general search.

Using: Xapian

- http://www.structuredabstraction.com/
- http://www.delivergood.org/

CustomMade

For general search.

Using: Solr

- http://www.custommade.com/

University of the Andes, Dept. of Political Science

For general search & section-specific search. Developed by Monoku.

Using: Solr

- http://www.congresovisible.org/
- http://www.monoku.com/

Christchurch Art Gallery

For general search & section-specific search.

Using: Solr

- http://christchurchartgallery.org.nz/search/
- http://christchurchartgallery.org.nz/collection/browse/

DevCheatSheet.com

For general search.

Using: Xapian

- http://devcheatsheet.com/

TodasLasRecetas

For search, faceting & More Like This.

Using: Solr

- http://www.todaslasrecetas.es/receta/s/?q=langostinos
- http://www.todaslasrecetas.es/receta/9526/brochetas-de-langostinos

AstroBin

For general search.

Using: Solr

- http://www.astrobin.com/

European Paper Company

For general search.

Using: ???

- http://europeanpaper.com/

mtn-op

For general search.

Using: ???

- http://mountain-op.com/

Crate

Crate is a PyPI mirror/replacement. It's using Haystack to power all search & faceted navigation on the site.

Using: Elasticsearch

- https://crate.io/

Pix Populi

Pix Populi is a popular French photo sharing site.

Using: Solr

- http://www.pix-populi.fr/

LocalWiki

LocalWiki is a tool for collaborating in local, geographic communities. It's using Haystack to power search on every LocalWiki instance.

Using: Solr

- http://localwiki.org/

Pitchup

For faceting, geo and autocomplete.

Using: ???

- http://www.pitchup.com/search/

Gidsy

Gidsy makes it easy for anyone to organize and find exciting things to do everywhere in the world.

For activity search, area pages, forums and private messages.

Using: Elasticsearch

- https://gidsy.com/
- https://gidsy.com/search/
- https://gidsy.com/forum/

GroundCity

Groundcity is a Romanian dynamic real estate site.

For real estate, forums and comments.

Using: Whoosh

- http://groundcity.ro/cautare/

Docket Alarm

Docket Alarm allows people to search court dockets across the country. With it, you can search court dockets in the International Trade Commission (ITC), the Patent Trial and Appeal Board (PTAB) and All Federal Courts.

Using: Elasticsearch

- https://www.docketalarm.com/search/ITC
- https://www.docketalarm.com/search/PTAB
- https://www.docketalarm.com/search/dockets

Educreations

Educreations makes it easy for anyone to teach what they know and learn what they don't with a recordable whiteboard. Haystack is used to provide search across users and lessons.

Using: Solr

- http://www.educreations.com/browse/

Haystack-Related Applications

Sub Apps

These are apps that build on top of the infrastructure provided by Haystack. Useful for essentially extending what Haystack can do.

queued_search

http://github.com/toastdriven/queued_search (2.X compatible)

Provides a queue-based setup as an alternative to `RealtimeSignalProcessor` or constantly running the `update_index` command. Useful for high-load, short update time situations.

celery-haystack

https://github.com/jezdez/celery-haystack (1.X and 2.X compatible)

Also provides a queue-based setup, this time centered around Celery. Useful for keeping the index fresh per model instance or with the included task to call the `update_index` management command instead.

haystack-rqueue

https://github.com/mandx/haystack-rqueue (2.X compatible)

Also provides a queue-based setup, this time centered around RQ. Useful for keeping the index fresh using `./manage.py rqworker`.

django-celery-haystack

https://github.com/mixcloud/django-celery-haystack-SearchIndex

Another queue-based setup, also around Celery. Useful for keeping the index fresh.

saved_searches

http://github.com/toastdriven/saved_searches (2.X compatible)

Adds personalization to search. Retains a history of queries run by the various users on the site (including anonymous users). This can be used to present the user with their search history and provide most popular/most recent queries on the site.

saved-search

https://github.com/DirectEmployers/saved-search

An alternate take on persisting user searches, this has a stronger focus on locale-based searches as well as further integration.

haystack-static-pages

http://github.com/trapeze/haystack-static-pages

Provides a simple way to index flat (non-model-based) content on your site. By using the management command that comes with it, it can crawl all pertinent pages on your site and add them to search.

django-tumbleweed

http://github.com/mcroydon/django-tumbleweed

Provides a tumblelog-like view to any/all Haystack-enabled models on your site. Useful for presenting date-based views of search data. Attempts to avoid the database completely where possible.

Haystack-Enabled Apps

These are reusable apps that ship with `SearchIndexes`, suitable for quick integration with Haystack.

- django-faq (freq. asked questions app) - http://github.com/benspaulding/django-faq
- django-essays (blog-like essay app) - http://github.com/bkeating/django-essays
- gtalug (variety of apps) - http://github.com/myles/gtalug
- sciencemuseum (science museum open data) - http://github.com/simonw/sciencemuseum
- vz-wiki (wiki) - http://github.com/jobscry/vz-wiki
- ffmff (events app) - http://github.com/stefreak/ffmff
- Dinette (forums app) - http://github.com/uswaretech/Dinette
- fiftystates_site (site) - http://github.com/sunlightlabs/fiftystates_site
- Open-Knesset (site) - http://github.com/ofri/Open-Knesset

Installing Search Engines

Solr

Official Download Location: http://www.apache.org/dyn/closer.cgi/lucene/solr/

Solr is Java but comes in a pre-packaged form that requires very little other than the JRE and Jetty. It's very performant and has an advanced featureset. Haystack suggests using Solr 6.x, though it's possible to get it working on Solr 4.x+ with a little effort. Installation is relatively simple:

For Solr 6.X:

```
curl -LO https://archive.apache.org/dist/lucene/solr/x.Y.0/solr-X.Y.0.tgz
tar -C solr -xf solr-X.Y.0.tgz --strip-components=1
cd solr
./bin/solr create -c tester -n basic_config
```

By default this will create a core with a managed schema. This setup is dynamic but not useful for haystack, and we'll need to configure solr to use a static (classic) schema. Haystack can generate a viable schema.xml and solrconfig.xml for you from your application and reload the core for you (once Haystack is installed and setup). To do this run: `./manage.py build_solr_schema --configure-directory=<CoreConfigDif> --reload-core`. In this example CoreConfigDir is something like `../solr-6.5.0/server/solr/tester/conf`, and `--reload-core` is what triggers reloading of the core. Please refer to `build_solr_schema` in the management-commands for required configuration.

For Solr 4.X:

```
curl -LO https://archive.apache.org/dist/lucene/solr/4.10.2/solr-4.10.2.tgz
tar xvzf solr-4.10.2.tgz
cd solr-4.10.2
cd example
java -jar start.jar
```

You'll need to revise your schema. You can generate this from your application (once Haystack is installed and setup) by running `./manage.py build_solr_schema`. Take the output from that command and place it in `solr-4.10.2/example/solr/collection1/conf/schema.xml`. Then restart Solr.

Warning: Please note; the template filename, the file YOU supply under TEMPLATE_DIR/search_configuration has changed to schema.xml from solr.xml. The previous template name solr.xml was a legacy holdover from older versions of solr.

You'll also need a Solr binding, `pysolr`. The official `pysolr` package, distributed via PyPI, is the best version to use (2.1.0+). Place `pysolr.py` somewhere on your `PYTHONPATH`.

Note: `pysolr` has its own dependencies that aren't covered by Haystack. See https://pypi.python.org/pypi/pysolr for the latest documentation. Simplest approach is to install using `pip install pysolr`

More Like This

To enable the "More Like This" functionality in Haystack, you'll need to enable the `MoreLikeThisHandler`. Add the following line to your `solrconfig.xml` file within the `config` tag:

```
<requestHandler name="/mlt" class="solr.MoreLikeThisHandler" />
```

Spelling Suggestions

To enable the spelling suggestion functionality in Haystack, you'll need to enable the `SpellCheckComponent`.

The first thing to do is create a special field on your `SearchIndex` class that mirrors the `text` field, but uses `FacetCharField`. This disables the post-processing that Solr does, which can mess up your suggestions. Something like the following is suggested:

```
class MySearchIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    # ... normal fields then...
    suggestions = indexes.FacetCharField()

    def prepare(self, obj):
        prepared_data = super(MySearchIndex, self).prepare(obj)
        prepared_data['suggestions'] = prepared_data['text']
        return prepared_data
```

Then, you enable it in Solr by adding the following line to your `solrconfig.xml` file within the `config` tag:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">

    <str name="queryAnalyzerFieldType">textSpell</str>

    <lst name="spellchecker">
      <str name="name">default</str>
      <str name="field">suggestions</str>
      <str name="spellcheckIndexDir">./spellchecker1</str>
      <str name="buildOnCommit">true</str>
    </lst>
</searchComponent>
```

Then change your default handler from:

```
<requestHandler name="standard" class="solr.StandardRequestHandler" default="true" />
```

... to ...:

```
<requestHandler name="standard" class="solr.StandardRequestHandler" default="true">
    <arr name="last-components">
        <str>spellcheck</str>
    </arr>
</requestHandler>
```

Be warned that the `<str name="field">suggestions</str>` portion will be specific to your `SearchIndex` classes (in this case, assuming the main field is called `text`).

Elasticsearch

Official Download Location: http://www.elasticsearch.org/download/

Elasticsearch is Java but comes in a pre-packaged form that requires very little other than the JRE. It's also very performant, scales easily and has an advanced featureset. Haystack currently only supports Elasticsearch 1.x and 2.x. Elasticsearch 5.x is not supported yet, if you would like to help, please see #1383.

Installation is best done using a package manager:

```
# On Mac OS X...
brew install elasticsearch

# On Ubuntu...
apt-get install elasticsearch

# Then start via:
elasticsearch -f -D es.config=<path to YAML config>

# Example:
elasticsearch -f -D es.config=/usr/local/Cellar/elasticsearch/0.90.0/config/
↪elasticsearch.yml
```

You may have to alter the configuration to run on `localhost` when developing locally. Modifications should be done in a YAML file, the stock one being `config/elasticsearch.yml`:

```
# Unicast Discovery (disable multicast)
discovery.zen.ping.multicast.enabled: false
discovery.zen.ping.unicast.hosts: ["127.0.0.1"]

# Name your cluster here to whatever.
# My machine is called "Venus", so...
cluster:
  name: venus

network:
  host: 127.0.0.1

path:
  logs: /usr/local/var/log
  data: /usr/local/var/data
```

You'll also need an Elasticsearch binding: elasticsearch (**NOT** `pyes`). Place `elasticsearch` somewhere on your `PYTHONPATH` (usually `python setup.py install` or `pip install elasticsearch`).

Note: `elasticsearch` has its own dependencies that aren't covered by Haystack. You'll also need `urllib3`.

Whoosh

Official Download Location: http://bitbucket.org/mchaput/whoosh/

Whoosh is pure Python, so it's a great option for getting started quickly and for development, though it does work for small scale live deployments. The current recommended version is 1.3.1+. You can install via PyPI using `sudo easy_install whoosh` or `sudo pip install whoosh`.

Note that, while capable otherwise, the Whoosh backend does not currently support "More Like This" or faceting. Support for these features has recently been added to Whoosh itself & may be present in a future release.

Xapian

Official Download Location: http://xapian.org/download

Xapian is written in C++ so it requires compilation (unless your OS has a package for it). Installation looks like:

```
curl -O http://oligarchy.co.uk/xapian/1.2.18/xapian-core-1.2.18.tar.xz
curl -O http://oligarchy.co.uk/xapian/1.2.18/xapian-bindings-1.2.18.tar.xz

unxz xapian-core-1.2.18.tar.xz
unxz xapian-bindings-1.2.18.tar.xz

tar xvf xapian-core-1.2.18.tar
tar xvf xapian-bindings-1.2.18.tar

cd xapian-core-1.2.18
./configure
make
sudo make install

cd ..
cd xapian-bindings-1.2.18
./configure
make
sudo make install
```

Xapian is a third-party supported backend. It is not included in Haystack proper due to licensing. To use it, you need both Haystack itself as well as `xapian-haystack`. You can download the source from http://github.com/notanumber/xapian-haystack/tree/master. Installation instructions can be found on that page as well. The backend, written by David Sauve (notanumber), fully implements the *SearchQuerySet* API and is an excellent alternative to Solr.

Debugging Haystack

There are some common problems people run into when using Haystack for the first time. Some of the common problems and things to try appear below.

Note: As a general suggestion, your best friend when debugging an issue is to use the `pdb` library included with Python. By dropping a `import pdb; pdb.set_trace()` in your code before the issue occurs, you can step through and examine variable/logic as you progress through. Make sure you don't commit those `pdb` lines though.

"No module named haystack."

This problem usually occurs when first adding Haystack to your project.

- Are you using the `haystack` directory within your `django-haystack` checkout/install?
- Is the `haystack` directory on your `PYTHONPATH`? Alternatively, is `haystack` symlinked into your project?
- Start a Django shell (`./manage.py shell`) and try `import haystack`. You may receive a different, more descriptive error message.
- Double-check to ensure you have no circular imports. (i.e. module A tries importing from module B which is trying to import from module A.)

"No results found." (On the web page)

Several issues can cause no results to be found. Most commonly it is either not running a `rebuild_index` to populate your index or having a blank `document=True` field, resulting in no content for the engine to search on.

- Do you have a `search_indexes.py` located within an installed app?
- Do you have data in your database?
- Have you run a `./manage.py rebuild_index` to index all of your content?
- Try running `./manage.py rebuild_index -v2` for more verbose output to ensure data is being processed/inserted.
- Start a Django shell (`./manage.py shell`) and try:

```
>>> from haystack.query import SearchQuerySet
>>> sqs = SearchQuerySet().all()
>>> sqs.count()
```

- You should get back an integer > 0. If not, check the above and reindex.

```
>>> sqs[0] # Should get back a SearchResult object.
>>> sqs[0].id # Should get something back like 'myapp.mymodel.1'.
>>> sqs[0].text # ... or whatever your document=True field is.
```

- If you get back either `u''` or `None`, it means that your data isn't making it into the main field that gets searched. You need to check that the field either has a template that uses the model data, a `model_attr` that pulls data directly from the model or a `prepare/prepare_FOO` method that populates the data at index time.
- Check the template for your search page and ensure it is looping over the results properly. Also ensure that it's either accessing valid fields coming back from the search engine or that it's trying to access the associated model via the `{{ result.object.foo }}` lookup.

"LockError: [Errno 17] File exists: '/path/to/whoosh_index/_MAIN_LOCK"'

This is a Whoosh-specific traceback. It occurs when the Whoosh engine in one process/thread is locks the index files for writing while another process/thread tries to access them. This is a common error when using `RealtimeSignalProcessor` with Whoosh under any kind of load, which is why it's only recommended for small sites or development.

The only real solution is to set up a cron job that runs `./manage.py rebuild_index` (optionally with `--age=24`) that runs nightly (or however often you need) to refresh the search indexes. Then disable the use of the `RealtimeSignalProcessor` within your settings.

The downside to this is that you lose real-time search. For many people, this isn't an issue and this will allow you to scale Whoosh up to a much higher traffic. If this is not acceptable, you should investigate either the Solr or Xapian backends.

"Failed to add documents to Solr: [Reason: None]"

This is a Solr-specific traceback. It generally occurs when there is an error with your `HAYSTACK_CONNECTIONS[<alias>]['URL']`. Since Solr acts as a webservice, you should test the URL in your web browser. If you receive an error, you may need to change your URL.

This can also be caused when using old versions of pysolr (2.0.9 and before) with httplib2 and including a trailing slash in your `HAYSTACK_CONNECTIONS[<alias>]['URL']`. If this applies to you, please upgrade to the current version of pysolr.

"Got an unexpected keyword argument 'boost'"

This is a Solr-specific traceback. This can also be caused when using old versions of pysolr (2.0.12 and before). Please upgrade your version of pysolr (2.0.13+).

Changelog

v2.6.0 (2017-01-02)

- Merge #1460: backend support for Elasticsearch 2.x. [Chris Adams]

  Thanks to João Junior (@joaojunior) and Bruno Marques (@ElSaico) for the patch

  Closes #1460 Closes #1391 Closes #1336 Closes #1247

- Docs: update Elasticsearch support status. [Chris Adams]
- Tests: avoid unrelated failures when elasticsearch is not installed. [Chris Adams]

  This avoids spurious failures in tests for other search engines when the elasticsearch client library is not installed at all but the ES backend is still declared in the settings.

- Tests: friendlier log message for ES version checks. [Chris Adams]

  This avoids a potentially scary-looking ImportError flying by in the test output for what's expected in normal usage.

- Tests: update ES version detection in settings. [Chris Adams]

  This allows the tests to work when run locally or otherwise outside of our Travis / Tox scripts by obtaining the version from the installed *elasticsearch* client library.

- Tests: update ES1 client version check message. [Chris Adams]

  The name of the Python module changed over time and this now matches the ES2 codebase behaviour of having the error message give you the exact package to install including the version.

- Update travis script with ES documentation. [Chris Adams]

  Add a comment for anyone wondering why this isn't a simple *add-apt-repository* call

- Fixed More Like This test with deferred query on Elasticsearch 2.x. [Bruno Marques]
- Fixed expected query behaviour on ES2.x test. [Bruno Marques]
- Install elasticsearch2.0 via apt. [joaojunior]
- Install elasticsearch2.0 via apt. [joaojunior]
- Remove typo. [joaojunior]
- Remove services elasticsearch. [joaojunior]
- Fix typo. [joaojunior]
- Sudo=true in .travis.yml to install elasticsearch from apt-get. [joaojunior]
- Fix .travis. [joaojunior]
- Add logging in __init__ tests elasticsearch. [joaojunior]
- Get changes from Master to resolve conflicts. [joaojunior]
- Install elasticsearch1.7 via apt. [joaojunior]

- Update Files to run tests in Elasticsearch2.x. [joaojunior]
- Refactoring the code in pull request #1336 . This pull request is to permit use ElasticSearch 2.X. [joaojunior]
- Improved custom object identifier test. [Chris Adams]

  This provides an example for implementors and ensures that failing to use the custom class would cause a test failure.
- Update management backend documentation for *–using* [flinkflonk]

  Thanks to @flinkflonk for the patch!

  Closes #1215
- Fix filtered "more like this" queries (#1459) [David Cook]

  Now the Solr backend correctly handles a *more_like_this()* query which is subsequently *filter()*-ed.

  Thanks to @divergentdave for the patch and tests!
- ReStructuredText link format fixes. (#1458) [John Heasly]
- Add note to Backend Support docs about lack of ES 5.X support. (#1457) [John Heasly]
- Replace deprecated Point.get_coords() calls. [Chris Adams]

  This works as far back as Django 1.8, which is the earliest which we support.

  See #1454
- Use setuptools_scm to manage package version numbers. [Chris Adams]

v2.5.1 (2016-10-28)

New

- Support for Django 1.10. [Chris Adams]

  Thanks to Morgan Aubert (@ellmetha) for the patch

  Closes #1434 Closes #1437 Closes #1445

Fix

- Contains filter, add endswith filter. [Antony]
  - *__contains* now works in a more intuitive manner (the previous behaviour remains the default for = shortcut queries and can be requested explicitly with *__content*)
  - *__endswith* is now supported as the logical counterpart to *__startswith*

  Thanks to @antonyr for the patch and @sebslomski for code review and testing.

Other

- V2.5.1. [Chris Adams]
- Add support for Django 1.10 (refs: #1437, #1434) [Morgan Aubert]
- Docs: fix Sphinx hierarchy issue. [Chris Adams]

- Fix multiprocessing regression in update_index. [Chris Adams]

  4e1e2e1c5df1ed1c5432b9d26fcb9dc1abab71f4 introduced a bug because it used a property name which exists on haystack.ConnectionHandler but not the Django ConnectionHandler class it's modeled on. Long-term, we should rename the Haystack class to something like *SearchConnectionHandler* to avoid future confusion.

  Closes #1449

- Doc: cleanup searchindex_api.rst. [Jack Norman]

  Thanks to Jack Norman (@jwnorman) for the patch

- Merge pull request #1444 from jeremycline/master. [Chris Adams]

  Upgrade setuptools in Travis so urllib3-1.18 installs

- Upgrade setuptools in Travis so urllib3-1.18 installs. [Jeremy Cline]

  The version of setuptools in Travis is too old to handle <= as an environment marker.

- Tests: accept Solr/ES config from environment. [Chris Adams]

  This makes it easy to override these values for e.g. running test instances using Docker images with something like this:

  ```
  ` TEST_ELASTICSEARCH_1_URL="http://$(docker port elasticsearch-1.7
  9200/tcp)/" TEST_SOLR_URL="http://$(docker port solr-6 8983/tcp)/solr/"
  test_haystack/run_tests.py `
  ```

  See #1408

- Merge pull request #1418 from Alkalit/master. [Steve Byerly]

  Added link for 2.5.x version docs

- Added link for 2.5.x version. [Alexey Kalinin]

- Merge pull request #1432 from farooqaaa/master. [Steve Byerly]

  Added missing *–batch-size* argument for *rebuild_index* management command.

- Added missing –batch-size argument. [Farooq Azam]

- Merge pull request #1036 from merwok/patch-1. [Steve Byerly]

  Documentation update

- Use ellipsis instead of pass. [Éric Araujo]

- Fix code to enable highlighting. [Éric Araujo]

- Merge pull request #1392 from browniebroke/bugfix/doc-error. [Steve Byerly]

  Fix Sphinx errors in the changelog

- Fix Sphinx errors in the changelog. [Bruno Alla]

- Merge pull request #1341 from tymofij/solr-hl-options. [Steve Byerly]

- Merge master > tymofij/solr-hl-options. [Steve Byerly]

- Make solr backend accept both shortened and full-form highlighting options. [Tim Babych]

- Autoprefix 'hl.' for solr options. [Tim Babych]

- Update gitignore to not track test artifacts. [Steve Byerly]

- Merge pull request #1413 from tymofij/patch-2. [Steve Byerly]

  typo: suite -> suit

- Typo: suite -> suit. [Tim Babych]
- Merge pull request #1412 from SteveByerly/highlight_sqs_docs. [Steve Byerly]

  improve sqs highlight docs - illustrate custom parameters
- Improve highlight docs for custom options. [Steve Byerly]

v2.5.0 (2016-07-11)

New

- SearchQuerySet.set_spelling_query for custom spellcheck. [Chris Adams]

  This makes it much easier to customize the text sent to the backend search engine for spelling suggestions independently from the actual query being executed.
- Support ManyToManyFields in model_attr lookups. [Arjen Verstoep]

  Thanks to @Terr for the patch
- *update_index* will retry after backend failures. [Gilad Beeri]

  Now *update_index* will retry failures multiple times before aborting with a progressive time delay.

  Thanks to Gilad Beeri (@giladbeeri) for the patch
- *highlight()* accepts custom values on Solr and ES. [Chris Adams]

  This allows the default values to be overriden and arbitrary backend-specific parameters may be provided to Solr or ElasticSearch.

  Thanks to @tymofij for the patch

  Closes #1334
- Allow Routers to return multiple indexes. [Chris Adams]

  Thanks to Hugo Chargois (@hchargois) for the patch

  Closes #1337 Closes #934
- Support for newer versions of Whoosh. [Chris Adams]
- Split SearchView.create_response into get_context. [Chris Adams]

  This makes it easy to override the default *create_response* behaviour if you don't want a standard HTML response.

  Thanks @seocam for the patch

  Closes #1338
- Django 1.9 support thanks to Claude Paroz. [Chris Adams]
- Create a changelog using gitchangelog. [Chris Adams]

  This uses gitchangelog to generate docs/changelog.rst from our Git commit history using the tags for each version. The configuration is currently tracking upstream exactly except for our version tags being prefixed with "v".

Changes

- Support for Solr 5+ spelling suggestion format. [Chris Adams]
- Set install requirements for Django versions. [Chris Adams]

  This will prevent accidentally breaking apps when Django 1.10 is released.

  Closes #1375
- Avoid double-query for queries matching no results. [Chris Adams]
- Update supported/tested Django versions. [Chris Adams]
  - setup.py install_requires uses *>=1.8* to match our current test matrix
  - Travis allows failures for Django 1.10 so we can start tracking the upcoming release
- Make backend subclassing easier. [Chris Adams]

  This change allows the backend build_search_kwargs to accept arbitrary extra arguments, making life easier for authors of *SearchQuery* or *SearchBackend* subclasses when they can directly pass a value which is directly supported by the backend search client.
- Update_index logging & multiprocessing improvements. [Chris Adams]
  - Since older versions of Python are no longer supported we no longer conditionally import multiprocessing (see #1001)
  - Use multiprocessing.log_to_stderr for all messages
  - Remove previously-disabled use of the multiprocessing workers for index removals, allowing the worker code to be simplified
- Moved signal processor loading to app_config.ready. [Chris Adams]

  Thanks to @claudep for the patch

  Closes #1260
- Handle *__in=[]* gracefully on Solr. [Chris Adams]

  This commit avoids the need to check whether a list is empty to avoid an error when using it for an *__in* filter.

  Closes #358 Closes #1311

Fix

- Attribute resolution on models which have a property named *all* (#1405) [Henrique Chehad]

  Thanks to Henrique Chehad (@henriquechehad) for the patch

  Closes #1404
- Tests will fall back to the Apache archive server. [Chris Adams]

  The Apache 4.10.4 release was quietly removed from the mirrors without a redirect. Until we have time to add newer Solr releases to the test suite we'll download from the archive and let the Travis build cache store it.
- Whoosh backend support for RAM_STORE (closes #1386) [Martin Owens]

  Thanks to @doctormo for the patch
- Unsafe update_worker multiprocessing sessions. [Chris Adams]

  The *update_index* management command does not handle the *multiprocessing* environment safely. On POSIX systems, *multiprocessing* uses *fork()* which means that when called in a context such as the test suite where

the connection has already been used some backends like pysolr or ElasticSearch may have an option socket connected to the search server and that leaves a potential race condition where HTTP requests are interleaved, producing unexpected errors.

This commit resets the backend connection inside the workers and has been stable across hundreds of runs, unlike the current situation where a single-digit number of runs would almost certainly have at least one failure.

Other improvements: * Improved sanity checks for indexed documents in management

command test suite. This wasn't actually the cause of the problem above but since I wrote it while tracking down the real problem there's no reason not to use it.

- update_index now checks that each block dispatched was executed to catch any possible silent failures.

Closes #1376 See #1001

- Tests support PyPy. [Chris Adams]

  PyPy has an optimization which causes it to call __len__ when running a list comprehension, which is the same thing Python does for *list(iterable)*. This commit simply changes the test code to always use *list* the PyPy behaviour matches CPython.

- Avoid an extra query on empty spelling suggestions. [Chris Adams]

  None was being used as a placeholder to test whether to run a spelling suggestion query but was also a possible response when the backend didn't return a suggestion, which meant that calling *spelling_suggestion()* could run a duplicate query.

- MultiValueField issues with single value (#1364) [Arjen Verstoep]

  Thanks to @terr for the patch!

- Queryset slicing and reduced code duplication. [Craig de Stigter]

  Now pagination will not lazy-load all earlier pages before returning the result.

  Thanks to @craigds for the patch

  Closes #1269 Closes #960

- Handle negative timestamps returned from ES. [Chris Adams]

  Elastic search can return negative timestamps for histograms if the dates are pre-1970. This PR properly handles these pre-1970 dates.

  Thanks to @speedplane for the patch

  Closes #1239

- SearchMixin allows form initial values. [Chris Adams]

  Thanks to @ahoho for the patch

  Closes #1319

- Graceful handling of empty __in= lists on ElasticSearch. [Chris Adams]

  Thanks to @boulderdave for the ES version of #1311

  Closes #1335

Other

- Docs: update unsupported backends notes. [Chris Adams]
  - Officially suggest developing backends as separate projects

  - Recommend Sphinx users consider django-sphinxql

- V2.5.0. [Chris Adams]
- Bump version to 2.5.dev2. [Chris Adams]
- AUTHORS. [Tim Babych]
- Expand my username into name in changelog.txt. [Tim Babych]
- Corrected non-ascii characters in comments. (#1390) [Mark Walker]
- Add lower and upper bounds for django versions. [Simon Hanna]
- Convert readthedocs link for their .org -> .io migration for hosted projects. [Adam Chainz]

  As per [their blog post of the 27th April](https://blog.readthedocs.com/securing-subdomains/) 'Securing subdomains':

  > Starting today, Read the Docs will start hosting projects from subdomains on the domain readthedocs.io, instead of on readthedocs.org. This change addresses some security concerns around site cookies while hosting user generated data on the same domain as our dashboard.

  Test Plan: Manually visited all the links I've modified.
- V2.5.dev1. [Chris Adams]
- Merge pull request #1349 from sbussetti/master. [Chris Adams]

  Fix logging call in *update_index*
- Fixes improper call to logger in mgmt command. [sbussetti]
- Merge pull request #1340 from claudep/manage_commands. [Chris Adams]

  chg: migrate management commands to argparse
- Updated management commands from optparse to argparse. [Claude Paroz]

  This follows Django's same move and prevents deprecation warnings. Thanks Mario César for the initial patch.
- Merge pull request #1225 from gregplaysguitar/patch-1. [Chris Adams]

  fix: correct docstring for ModelSearchForm.get_models !minor
- Fix bogus docstring. [Greg Brown]
- Merge pull request #1328 from claudep/travis19. [Chris Adams]

  Updated test configs to include Django 1.9
- Updated test configs to include Django 1.9. [Claude Paroz]
- Merge pull request #1313 from chrisbrooke/Fix-elasticsearch-2.0-meta- data-changes. [Chris Adams]
- Remove boost which is now unsupported. [Chris Brooke]
- Fix concurrency issues when building UnifiedIndex. [Chris Adams]

  We were getting this error a lot when under load in a multithreaded wsgi environment:

  > Model '%s' has more than one 'SearchIndex'' handling it.

  Turns out the connections in haystack.connections and the UnifiedIndex instance were stored globally. However there is a race condition in UnifiedIndex.build() when multiple threads both build() at once, resulting in the above error.

  Best fix is to never share the same engine or UnifiedIndex across multiple threads. This commit does that.

  Closes #959 Closes #615

- Load connection routers lazily. [Chris Adams]

  Thanks to Tadas Dailyda (@skirsdeda) for the patch

  Closes #1034 Closes #1296

- DateField/DateTimeField accept strings values. [Chris Adams]

  Now the convert method will be called by default when string values are received instead of the normal date/datetime values.

  Closes #1188

- Fix doc ReST warning. [Chris Adams]
- Merge pull request #1297 from martinsvoboda/patch-1. [Sam Peka]

  Highlight elasticsearch 2.X is not supported yet

- Highlight in docs that elasticsearch 2.x is not supported yet. [Martin Svoboda]
- Start updating compatibility notes. [Chris Adams]
  - Deprecate versions of Django which are no longer supported by the Django project team
  - Update ElasticSearch compatibility messages
  - Update Travis / Tox support matrix
- Merge pull request #1287 from ses4j/patch-1. [Sam Peka]

  Remove duplicated SITE_ID from test_haystack/settings.py

- Remove redundant SITE_ID which was duplicated twice. [Scott Stafford]
- Add `fuzzy` operator to SearchQuerySet. [Chris Adams]

  This exposes the backends' native fuzzy query support.

  Thanks to Ana Carolina (@anacarolinats) and Steve Bussetti (@sbussetti) for the patch.

- Merge pull request #1281 from itbabu/python35. [Justin Caratzas]

  Add python 3.5 to tests

- Add python 3.5 to tests. [Marco Badan]

  ref: https://docs.djangoproject.com/en/1.9/faq/install/#what-python-version-can-i-use-with-django

- SearchQuerySet: don't trigger backend access in __repr__ [Chris Adams]

  This can lead to confusing errors or performance issues by triggering backend access at unexpected locations such as logging.

  Closes #1278

- Merge pull request #1276 from mariocesar/patch-1. [Chris Adams]

  Use compatible get_model util to support new django versions

  Thanks to @mariocesar for the patch!

- Reuse haystack custom get model method. [Mario César Señoranis Ayala]
- Removed unused import. [Mario César Señoranis Ayala]
- Use compatible get_model util to support new django versions. [Mario César Señoranis Ayala]
- Merge pull request #1263 from dkarchmer/patch-1. [Chris Adams]

  Update views_and_forms.rst

- Update views_and_forms.rst. [David Karchmer]

  After breaking my head for an hour, I realized the instructions to upgrade to class based views is incorrect. It should indicate that switch from *page* to *page_obj* and not *page_object*

v2.3.2 (2015-11-11)

- V2.3.2 maintenance update. [Chris Adams]
- Fix #1253. [choco]
- V2.3.2 pre-release version bump. [Chris Adams]
- Allow individual records to be skipped while indexing. [Chris Adams]

  Previously there was no easy way to skip specific objects other than filtering the queryset. This change allows a prepare method to raise *SkipDocument* after calling methods or making other checks which cannot easily be expressed as database filters.

  Thanks to Felipe Prenholato (@chronossc) for the patch

  Closes #380 Closes #1191

v2.4.1 (2015-10-29)

- V2.4.1. [Chris Adams]
- Minimal changes to the example project to allow test use. [Chris Adams]
- Merge remote-tracking branch 'django-haystack/pr/1261' [Chris Adams]

  The commit in #1252 / #1251 was based on the assumption that the tutorial used the new generic views, which is not yet correct.

  This closes #1261 by restoring the wording and adding some tests to avoid regressions in the future before the tutorial is overhauled.
- Rename 'page_obj' with 'page' in the tutorial, section Search Template as there is no 'page_obj' in the controller and this results giving 'No results found' in the search. [bboneva]
- Style cleanup. [Chris Adams]
  - Remove duplicate & unused imports
  - PEP-8 indentation & whitespace
  - Use *foo not in bar* instead of *not foo in bar*
- Update backend logging style. [Chris Adams]
  - Make Whoosh message consistent with the other backends
  - Pass exception info to loggers in except: blocks
  - PEP-8
- Avoid unsafe default value on backend clear() methods. [Chris Adams]

  Having a mutable structure like a list as a default value is unsafe; this commit changes that to the standard None.
- Merge pull request #1254 from chocobn69/master. [Chris Adams]

  Update for API change in elasticsearch 1.8 (closes #1253)

  Thanks to @chocobn69 for the patch

- Fix #1253. [choco]
- Tests: update Solr launcher for changed mirror format. [Chris Adams]

  The Apache mirror-detection script appears to have changed its response format recently. This change handles that and makes future error messages more explanatory.
- Bump doc version numbers - closes #1105. [Chris Adams]
- Merge pull request #1252 from rhemzo/master. [Chris Adams]

  Update tutorial.rst (closes #1251)

  Thanks to @rhemzo for the patch
- Update tutorial.rst. [rhemzo]

  change page for page_obj
- Merge pull request #1240 from speedplane/improve-cache-fill. [Chris Adams]

  Use a faster implementation of query result cache
- Use a faster implementation of this horrible cache. In my tests it runs much faster and uses far less memory. [speedplane]
- Merge pull request #1149 from lovmat/master. [Chris Adams]

  FacetedSearchMixin bugfixes and improvements

  - Updated documentation & example code
  - Fixed inheritance chain
  - Added facet_fields

  Thanks to @lovmat for the patch
- Updated documentation, facet_fields attribute. [lovmat]
- Added facet_fields attribute. [lovmat]

  Makes it easy to include facets into FacetedSearchVIew
- Bugfixes. [lovmat]
- Merge pull request #1232 from dlo/patch-1. [Chris Adams]

  Rename elasticsearch-py to elasticsearch in docs

  Thanks to @dlo for the patch
- Rename elasticsearch-py to elasticsearch in docs. [Dan Loewenherz]
- Update wording in SearchIndex get_model exception. [Chris Adams]

  Thanks to Greg Brown (@gregplaysguitar) for the patch

  Closes #1223
- Corrected exception wording. [Greg Brown]
- Allow failures on Python 2.6. [Chris Adams]

  Some of our test dependencies like Mock no longer support it. Pinning Mock==1.0.1 on Python 2.6 should avoid that failure but the days of Python 2.6 are clearly numbered.
- Travis: stop testing unsupported versions of Django on Python 2.6. [Chris Adams]

- Use Travis' matrix support rather than tox. [Chris Adams]

  This avoids a layer of build setup and makes the Travis console reports more useful

- Tests: update the test version of Solr in use. [Chris Adams]

  4.7.2 has disappeared from most of the Apache mirrors

v2.4.0 (2015-06-09)

- Release 2.4.0. [Chris Adams]
- Merge pull request #1208 from ShawnMilo/patch-1. [Chris Adams]

  Fix a typo in the faceting docs

- Possible typo fix. [Shawn Milochik]

  It seems that this was meant to be results.

- 2.4.0 release candidate 2. [Chris Adams]
- Fix Django 1.9 deprecation warnings. [Ilan Steemers]
  - replaced get_model with haystack_get_model which returns the right function depending on the Django version
  - get_haystack_models is now compliant with > Django 1.7

  Closes #1206

- Documentation: update minimum versions of Django, Python. [Chris Adams]
- V2.4.0 release candidate. [Chris Adams]
- Bump version to 2.4.0.dev1. [Chris Adams]
- Travis: remove Django 1.8 from allow_failures. [Chris Adams]
- Tests: update test object creation for Django 1.8. [Chris Adams]

  Several of the field tests previously assigned a related test model instance before saving it:

```
mock_tag = MockTag(name='primary')
mock = MockModel()
mock.tag = mock_tag
```

  Django 1.8 now validates this dodgy practice and throws an error.

  This commit simply changes it to use *create()* so the mock_tag will have a pk before assignment.

- Update AUTHORS. [Chris Adams]
- Tests: fix deprecated Manager.get_query_set call. [Chris Adams]
- Updating haystack to test against django 1.8. [Chris Adams]

  Updated version of @troygrosfield's patch updating the test-runner for Django 1.8

  Closes #1175

- Travis: allow Django 1.8 failures until officially supported. [Chris Adams]

  See #1175

- Remove support for Django 1.5, add 1.8 to tox/travis. [Chris Adams]

  The Django project does not support 1.5 any more and it's the source of most of our false-positive test failures

- Use db.close_old_connections instead of close_connection. [Chris Adams]

  Django 1.8 removed the *db.close_connection* method.

  Thanks to Alfredo Armanini (@phingage) for the patch

- Fix mistake in calling super TestCase method. [Ben Spaulding]

  Oddly this caused no issue on Django <= 1.7, but it causes numerous errors on Django 1.8.

- Correct unittest imports from commit e37c1f3. [Ben Spaulding]

- Prefer stdlib unittest over Django's unittest2. [Ben Spaulding]

  There is no need to fallback to importing unittest2 because Django 1.5 is the oldest Django we support, so django.utils.unittest is guaranteed to exist.

- Prefer stdlib OrderedDict over Django's SortedDict. [Ben Spaulding]

  The two are not exactly they same, but they are equivalent for Haystack's needs.

- Prefer stdlib importlib over Django's included version. [Ben Spaulding]

  The app_loading module had to shuffle things a bit. When it was importing the function it raised a [RuntimeError][]. Simply importing the module resolved that.

  [RuntimeError]: https://gist.github.com/benspaulding/f36eaf483573f8e5f777

- Docs: explain how field boosting interacts with filter. [Chris Adams]

  Thanks to @amjoconn for contributing a doc update to help newcomers

  Closes #1043

- Add tests for values/values_list slicing. [Chris Adams]

  This confirms that #1019 is fixed

- Update_index: avoid gaps in removal logic. [Chris Adams]

  The original logic did not account for the way removing records interfered with the pagination logic.

  Closes #1194

- Update_index: don't use workers to remove stale records. [Chris Adams]

  There was only minimal gain to this because, unlike indexing, removal is a simple bulk operation limited by the search engine.

  See #1194 See #1201

- Remove lxml dependency. [Chris Adams]

  pysolr 3.3.2+ no longer requires lxml, which saves a significant install dependency

- Allow individual records to be skipped while indexing. [Chris Adams]

  Previously there was no easy way to skip specific objects other than filtering the queryset. This change allows a prepare method to raise *SkipDocument* after calling methods or making other checks which cannot easily be expressed as database filters.

  Thanks to Felipe Prenholato (@chronossc) for the patch

  Closes #380 Closes #1191

- Update_index: avoid "MySQL has gone away error" with workers. [Eric Bressler (Platform)]

  This fixes an issue with a stale database connection being passed to a multiprocessing worker when using *–remove*

Thanks to @ebressler for the patch

Closes #1201

- Depend on pysolr 3.3.1. [Chris Adams]
- Start-solr-test-server: avoid Travis dependency. [Chris Adams]

  This will now fall back to the current directory when run outside of our Travis-CI environment
- Fix update_index –remove handling. [Chris Adams]
  - Fix support for custom keys by reusing the stored value rather than regenerating following the default pattern
  - Batch remove operations using the total number of records in the search index rather than the database

  Closes #1185 Closes #1186 Closes #1187
- Merge pull request #1177 from paulshannon/patch-1. [Chris Adams]

  Update TravisCI link in README
- Update TravisCI link. [Paul Shannon]

  I think the repo got changed at some point and the old project referenced at travisci doesn't exist anymore...
- Travis: enable containers. [Chris Adams]
  - Move apt-get installs to the addons/apt_packages: http://docs.travis-ci.com/user/apt-packages/
  - Set *sudo: false* to enable containers: http://docs.travis-ci.com/user/workers/container-based-infrastructure/
- Docs: correct stray GeoDjango doc link. [Chris Adams]
- Document: remove obsolete Whoosh Python 3 warning. [Chris Adams]

  Thanks to @gitaarik for the pull request

  Closes #1154 Fixes #1108
- Remove method_decorator backport (closes #1155) [Chris Adams]

  This was no longer used anywhere in the Haystack source or documentation
- Travis: enable APT caching. [Chris Adams]
- Travis: update download caching. [Chris Adams]
- App_loading cleanup. [Chris Adams]
  - Add support for Django 1.7+ AppConfig
  - Rename internal app_loading functions to have **haystack_** prefix to make it immediately obvious that they are not Django utilities and start
  - Add tests to avoid regressions for apps nested with multiple levels of module hierarchy like *raven.contrib.django.raven_compat*
  - Refactor app_loading logic to make it easier to remove the legacy compatibility code when we eventually drop support for older versions of Django

  Fixes #1125 Fixes #1150 Fixes #1152 Closes #1153
- Switch defaults closer to Python 3 defaults. [Chris Adams]
  - Add __future__ imports:

  isort –add_import 'from __future__ import absolute_import, division, print_function, unicode_literals'

  - Add source encoding declaration header

- Setup.py: use strict PEP-440 dev version. [Chris Adams]

  The previous version was valid as per PEP-440 but triggers a warning in pkg_resources

- Merge pull request #1146 from kamilmowinski/patch-1. [Chris Adams]

  Fix typo in SearchResult documentation

- Update searchresult_api.rst. [kamilmowinski]

- Merge pull request #1143 from wicol/master. [Chris Adams]

  Fix deprecation warnings in Django 1.6.X (thanks @wicol)

- Fix deprecation warnings in Django 1.6.X. [Wictor]

  Options.model_name was introduced in Django 1.6 together with a deprecation warning: https://github.com/django/django/commit/ec469ade2b04b94bfeb59fb0fc7d9300470be615

- Travis: move tox setup to before_script. [Chris Adams]

  This should cause dependency installation problems to show up as build errors rather than outright failures

- Update ElasticSearch defaults to allow autocompleting numbers. [Chris Adams]

  Previously the defaults for ElasticSearch used the *lowercase* tokenizer, which prevented numbers from being autocompleted.

  Thanks to Phill Tornroth (@phill-tornroth) for contributing a patch which changes the default settings to use the *standard* tokenizer with the *lowercase* filter

  Closes #1056

- Update documentation for new class-based views. [Chris Adams]

  Thanks to @troygrosfield for the pull-request

  Closes #1139 Closes #1133 See #1130

- Added documentation for configuring facet behaviour. [Chris Adams]

  Thanks to Philippe Luickx for the contribution

  Closes #1111

- UnifiedIndex has a stable interface to get all indexes. [Chris Adams]

  Previously it was possible for UnifiedIndexes.indexes to be empty when called before the list had been populated. This change deprecates accessing *.indexes* directly in favor of a *get_indexes()* accessor which will call *self.build()* first if necessary.

  Thanks to Phill Tornroth for the patch and tests.

  Closes #851

- Add support for SQ in SearchQuerySet.narrow() (closes #980) [Chris Adams]

  Thanks to Andrei Fokau (@andreif) for the patch and tests

- Disable multiprocessing on Python 2.6 (see #1001) [Chris Adams]

  multiprocessing.Pool.join() hangs reliably on Python 2.6 but not any later version tested. Since this is an optional feature we'll simply disable it

- Bump version number to 2.4.0-dev. [Chris Adams]

- Update_index: wait for all pool workers to finish. [Chris Adams]

  There was a race condition where update_index() would return before all of the workers had finished updating Solr. This manifested itself most frequently as Travis failures for the multiprocessing test (see #1001).

- Tests: Fix ElasticSearch index setup (see #1093) [Chris Adams]

  Previously when clear_elasticsearch_index() was called to reset the tests, this could produce confusing results because it cleared the mappings without resetting the backend's setup_complete status and thus fields which were expected to have a specific type would end up being inferred

  With this changed test_regression_proper_start_offsets and test_more_like_this no longer fail

- Update rebuild_index –nocommit handling and add tests. [Chris Adams]

  rebuild_index builds its option list by combining the options from clear_index and update_index. This previously had a manual exclude list for options which were present in both commands to avoid conflicts but the nocommit option wasn't in that list.

  This wasn't tested because our test suite uses call_command rather than invoking the option parser directly.

  This commit also adds tests to confirm that –nocommit will actually pass commit=False to clear_index and update_index.

  Closes #1140 See #1090

- Support ElasticSearch 1.x distance filter syntax (closes #1003) [Chris Adams]

  The elasticsearch 1.0 release was backwards incompatible with our previous usage.

  Thanks to @dulaccc for the patch adding support.

- Docs: add Github style guide link to pull request instructions. [Chris Adams]

  The recent Github blog post makes a number of good points:

  https://github.com/blog/1943-how-to-write-the-perfect-pull-request

- Fixed exception message when resolving model_attr. [Wictor]

  This fixes the error message displayed when model_attr references an unknown attribute.

  Thanks to @wicol for the patch

  Closes #1094

- Compatibility with Django 1.7 app loader (see #1097) [Chris Adams]

  - Added wrapper around get_model, so that Django 1.7 uses the new app loading mechanism.
  - Added extra model check to prevent that a simple module is treated as model.

  Thanks to Dirk Eschler (@deschler) for the patch.

- Fix index_fieldname to match documentation (closes #825) [Chris Adams]

  @jarig contributed a fix to ensure that index_fieldname renaming does not interfere with using the field name declared on the index.

- Add tests for Solr/ES spatial order_by. [Chris Adams]

  This exists primarily to avoid the possibility of breaking compatibility with the inconsistent lat, lon ordering used by Django, Solr and ElasticSearch.

- Remove undocumented *order_by_distance* [Chris Adams]

  This path was an undocumented artifact of the original geospatial feature-branch back in the 1.X era. It wasn't documented and is completely covered by the documented API.

- ElasticSearch tests: PEP-8 cleanup. [Chris Adams]
- Implement managers tests for spatial features. [Chris Adams]

  This is largely shadowed by the actual spatial tests but it avoids surprises on the query generation

  - Minor PEP-8
- Remove unreferenced add_spatial methods. [Chris Adams]

  SolrSearchQuery and ElasticsearchSearchQuery both defined an *add_spatial* method which was neither called nor documented.
- Remove legacy httplib/httplib2 references. [Chris Adams]

  We've actually delegated the actual work to requests but the docs & tests had stale references
- Tests: remove legacy spatial backend code. [Chris Adams]

  This has never run since the solr_native_distance backend did not exist and thus the check always failed silently
- ElasticSearch backend: minor PEP-8 cleanup. [Chris Adams]
- Get-solr-download-url: fix Python 3 import path. [Chris Adams]

  This allows the scripts to run on systems where Python 3 is the default version
- Merge pull request #1130 from troygrosfield/master. [Chris Adams]

  Added generic class based search views

  (thanks @troygrosfield)
- Removed "expectedFailure". [Troy Grosfield]
- Minor update. [Troy Grosfield]
- Added tests for the generic search view. [Troy Grosfield]
- Hopefully last fix for django version checking. [Troy Grosfield]
- Fix for django version check. [Troy Grosfield]
- Adding fix for previously test for django 1.7. [Troy Grosfield]
- Adding py34-django1.7 to travis. [Troy Grosfield]
- Test for the elasticsearch client. [Troy Grosfield]
- Added unicode_literals import for py 2/3 compat. [Troy Grosfield]
- Added generic class based search views. [Troy Grosfield]
- Merge pull request #1101 from iElectric/nothandledclass. [Chris Adams]

  Report correct class when raising NotHandled
- Report correct class when raising NotHandled. [Domen Kožar]
- Merge pull request #1090 from andrewschoen/feature/no-commit-flag. [Chris Adams]

  Adds a –nocommit arg to the update_index, clear_index and rebuild_index management command.
- Adds a –nocommit arg to the update_index, clear_index and rebuild_index management commands. [Andrew Schoen]
- Merge pull request #1103 from pkafei/master. [Chris Adams]

  Update documentation to reference Solr 4.x
- Changed link to official archive site. [Portia Burton]

- Added path to schema.xml. [Portia Burton]
- Added latest version of Solr to documentation example. [Portia Burton]
- Update ElasticSearch version requirements. [Chris Adams]
- Elasticsearch's python api by default has _source set to False, this causes keyerror mentioned in bug #1019. [xsamurai]
- Solr: clear() won't call optimize when commit=False. [Chris Adams]

  An optimize will trigger a commit implicitly so we'll avoid calling it when the user has requested not to commit
- Bumped __version__ (closes #1112) [Dan Watson]
- Travis: allow PyPy builds to fail. [Chris Adams]

  This is currently unstable and it's not a first-class supported platform yet
- Tests: fix Solr server tarball test. [Chris Adams]

  On a clean Travis instance, the tarball won't exist
- Tests: have Solr test server startup script purge corrupt tarballs. [Chris Adams]

  This avoids tests failing if a partial download is cached by Travis
- Merge pull request #1084 from streeter/admin-mixin. [Daniel Lindsley]

  Document and add an admin mixin
- Document support for searching in the Django admin. [Chris Streeter]
- Add some spacing. [Chris Streeter]
- Create an admin mixin for external use. [Chris Streeter]

  There are cases where one might have a different base admin class, and wants to use the search features in the admin as well. Creating a mixin makes this a bit cleaner.

v2.3.1 (2014-09-22)

- V2.3.1. [Chris Adams]
- Tolerate non-importable apps like django-debug-toolbar. [Chris Adams]

  If your installed app isn't even a valid Python module, haystack will issue a warning but continue.

  Thanks to @gojomo for the patch

  Closes #1074 Closes #1075
- Allow apps without models.py on Django <1.7. [Chris Adams]

  This wasn't officially supported by Django prior to 1.7 but is used by some third-party apps such as Grappelli

  This commit adds a somewhat contrived test app to avoid future regressions by ensuring that the test suite always has an application installed which does not have models.py

  See #1073

v2.3.0 (2014-09-19)

- Travis: Enable IRC notifications. [Chris Adams]
- Fix app loading call signature. [Chris Adams]

  Updated code from #1016 to ensure that get_models always returns a list (previously on Django 1.7 it would return the bare model when called with an argument of the form *app.modelname*)

  Add some basic tests
- App loading: use ImproperlyConfigured for bogus app names. [Chris Adams]

  This never worked but we'll be more consistent and return ImproperlyConfigured instead of a generic LookupError
- App Loading: don't suppress app-registry related exceptions. [Chris Adams]

  This is just asking for trouble in the future. If someone comes up with an edge case, we should add a test for it
- Remove Django version pin from install_requires. [Chris Adams]
- Django 1.7 support for app discovery. [Chris Adams]
  - Refactored @Xaroth's patch from #1015 into a separate utils module
  - PEP-8 cleanup
- Start the process of updating for v2.3 release. [Chris Adams]
- Django 1.7 compatibility for model loading. [Chris Adams]

  This refactors the previous use of model _meta.module_name and updates the tests so the previous change can be tested safely.

  Closes #981 Closes #982
- Update tox Django version pins. [Chris Adams]
- Mark expected failures for Django 1.7 (see #1069) [Chris Adams]
- Django 1.7: ensure that the app registry is ready before tests are loaded. [Chris Adams]

  The remaining test failures are due to some of the oddities in model mocking, which can be solved by overhauling the way we do tests and mocks.
- Tests: Whoosh test overhaul. [Chris Adams]
  - Move repetitive filesystem reset logic into WhooshTestCase which cleans up after itself
  - Use mkdtemp instead of littering up the current directory with a 'tmp' subdirectory
  - Use skipIf rather than expectFailure on test_writable to disable it only when STORAGE=ram rather than always
- Unpin elasticsearch library version for testing. [Chris Adams]
- Tests: add MIDDLEWARE_CLASSES for Django 1.7. [Chris Adams]
- Use get_model_ct_tuple to generate template name. [Chris Adams]
- Refactor simple_backend to use get_model_ct_tuple. [Chris Adams]
- Haystack admin: refactor to use get_model_ct_tuple. [Chris Adams]
- Consolidate model meta references to use get_model_ct (see #981) [Chris Adams]

  This use of a semi-public Django interface will break in Django 1.7 and we can start preparing by using the existing haystack.utils.get_model_ct function instead of directly accessing it everywhere.

- Refactor get_model_ct to handle Django 1.7, add tuple version. [Chris Adams]

  We have a mix of model _meta access which usually expects strings but in a few places needs raw values. This change adds support for Django 1.7 (see https://code.djangoproject.com/ticket/19689) and allows raw tuple access to handle other needs in the codebase

- Add Django 1.7 warning to Sphinx docs as well. [Chris Adams]

v2.2.1 (2014-09-03)

- Mark 2.2.X as incompatible with Django 1.7. [Chris Adams]
- Tests: don't suppress Solr stderr logging. [Chris Adams]

  This will make easier to tell why Solr sometimes goes away on Travis

- Update Travis & Tox config. [Chris Adams]
  - Tox: wait for Solr to start before running tests
  - Travis: allow solr & pip downloads to be cached
  - Travis now uses start-solr-test-server.sh instead of travis-solr
  - Test Solr configuration uses port 9001 universally as per the documentation
  - Change start-solr-test-server.sh to change into its containing directory, which also allows us to remove the realpath dependency
  - **Test Solr invocation matches pysolr**
    - Use get-solr-download-url script to pick a faster mirror
    - Upgrade to Solr 4.7.2
- Travis, Tox: add Django 1.7 targets. [Chris Adams]
- Merge pull request #1055 from andreif/feature/realpath-fallback-osx. [Chris Adams]
- Fallback to pwd if realpath is not available. [Andrei Fokau]
- Merge pull request #1053 from gandalfar/patch-1. [Chris Adams]
- Update example for Faceting to reference page.object_list. [Jure Cuhalev]

  Instead of *results* - ref #1052

- Add PyPy targets to Tox & Travis. [Chris Adams]

  Closes #1049

- Merge pull request #1044 from areski/patch-1. [Chris Adams]

  Update Xapian install instructions (thanks @areski)

- Update Xapian install. [Areski Belaid]
- Docs: fix signal processors link in searchindex_api. [Chris Adams]

  Correct a typo in b676b17dbc4b29275a019417e7f19f531740f05e

- Merge pull request #1050 from jogwen/patch-2. [Chris Adams]
- Link to 'signal processors' [Joanna Paulger]
- Merge pull request #1047 from g3rd/patch-1. [Chris Adams]

  Update the installing search engine documentation URL (thanks @g3rd)

- Fixed the installing search engine doc URL. [Chad Shrock]
- Merge pull request #1025 from reinout/patch-1. [Chris Adams]

  Fixed typo in templatetag docs example (thanks to @reinout)
- Fixed typo in example. [Reinout van Rees]

  It should be *css_class* in the template tag example instead of just *class*. (It is mentioned correctly in the syntax line earlier).

v2.2.0 (2014-08-03)

- Release v2.2.0. [Chris Adams]
- Test refactor - merge all the tests into one test suite (closes #951) [Chris Adams]

  Major refactor by @honzakral which stabilized the test suite, makes it easier to run and add new tests and somewhat faster, too.

  - Merged all the tests
  - Mark tests as skipped when a backend is not available (e.g. no ElasticSearch or Solr connection)
  - Massively simplified test runner (`python setup.py test`)

  Minor updates: * Travis:

  - Test Python 3.4
  - Use Solr 4.6.1
  - Simplified legacy test code which can now be replaced by the test utilities in newer versions of Django
  - Update ElasticSearch client & tests for ES 1.0+
  - Add option for SearchModelAdmin to specify the haystack connection to use
  - Fixed a bug with RelatedSearchQuerySet caching using multiple instances (429d234)
- RelatedSearchQuerySet: move class globals to instance properties. [Chris Adams]

  This caused obvious failures in the test suite and presumably elsewhere when multiple RelatedSearchQuerySet instances were in use
- Merge pull request #1032 from maikhoepfel/patch-1. [Justin Caratzas]

  Drop unused variable when post-processing results
- Drop unused variable when post-processing results. [Maik Hoepfel]

  original_results is not used in either method, and can be safely removed.
- 404 when initially retrieving mappings is ok. [Honza Král]
- Ignore 400 (index already exists) when creating an index in Elasticsearch. [Honza Král]
- ElasticSearch: update clear() for 1.x+ syntax. [Chris Adams]

  As per http://www.elasticsearch.org/guide/en/elasticsearch/reference/1.x/docs-delete-by-query.html this should be nested inside a top-level query block:

  {“query”: {“query_string”: . . . }}
- Add setup.cfg for common linters. [Chris Adams]

- ElasticSearch: avoid KeyError for empty spelling. [Chris Adams]

  It was possible to get a KeyError when spelling suggestions were requested but no suggestions are returned by the backend.

  Thanks to Steven Skoczen (@skoczen) for the patch

- Merge pull request #970 from tobych/patch-3. [Justin Caratzas]

  Improve punctuation in super-scary YMMV warning

- Improve punctuation in super-scary YMMV warning. [Toby Champion]
- Merge pull request #969 from tobych/patch-2. [Justin Caratzas]

  Fix typo; clarify purpose of search template

- Fix typo; clarify purpose of search template. [Toby Champion]
- Merge pull request #968 from tobych/patch-1. [Justin Caratzas]

  Fix possessive “its” in tutorial.rst

- Fix possessive “its” [Toby Champion]
- Merge pull request #938 from Mbosco/patch-1. [Daniel Lindsley]

  Update tutorial.rst

- Update tutorial.rst. [BoscoMW]
- Fix logging call in SQS post_process_results (see #648) [Chris Adams]

  This was used in an except: handler and would only be executed when a load_all() queryset retrieved a model which wasn’t registered with the index.

- Merge pull request #946 from gkaplan/spatial-docs-fix. [Daniel Lindsley]

  Small docs fix for spatial search example code

- Fix typo with instantiating Distance units. [Graham Kaplan]
- Solr backend: correct usage of pysolr delete. [Chris Adams]

  We use HAYSTACK_ID_FIELD in other places but the value passed to pysolr’s delete() method must use the keyword argument `id`:

  https://github.com/toastdriven/pysolr/blob/v3.1.0/pysolr.py#L756

  Although the value is passed to Solr an XML tag named `<id>` it will always be checked against the actual `uniqueKey` field even if it uses a custom name:

  https://wiki.apache.org/solr/UpdateXmlMessages#A.22delete.22_documents_by_ID_and_by_Query

  Closes #943

- Add a note on elasticsearch-py versioning with regards to 1.0. [Honza Král]
- Ignore 404 when removing a document from elasticsearch. [Honza Král]

  Fixes #942

- Ignore missing index during .clear() [Honza Král]

  404 in indices.delete can only mean that the index is there, no issue for a delete operation

  Fixes #647

- Tests: remove legacy targets. [Chris Adams]

  - Django 1.4 is no longer supported as per the documentation
  - Travis: use Python 3.3 targets instead of 3.2
- Tests: update pysolr requirement to 3.1.1. [Chris Adams]

  3.1.1 shipped a fix for a change in the Solr response format for the content extraction handler
- Merge pull request #888 from acdha/888-solr-field-list-regression. [Chris Adams]

  Solr / ElasticSearch backends: restore run() kwargs handling

  This fixes an earlier regression which did not break functionality but made *.values()* and *.values_list()* much less of an optimization than intended.

  #925 will be a more comprehensive refactor but this is enough of a performance win to be worth including if a point release happens before #925 lands.
- ElasticSearch backend: run() kwargs are passed directly to search backend. [Chris Adams]

  This allows customization by subclasses and also fixes #888 by ensuring that the custom field list prepared by *ValuesQuerySet* and *ValuesListQuerySet* is actually used.
- Solr backend: run() kwargs are passed directly to search backend. [Chris Adams]

  This allows customization by subclasses and also fixes #888 by ensuring that the custom field list prepared by *ValuesQuerySet* and *ValuesListQuerySet* is actually used.
- Tests: skip Solr content extraction with old PySolr. [Chris Adams]

  Until pysolr 3.1.1 ships there's no point in running the Solr content extraction tests because they'll fail:

  https://github.com/toastdriven/pysolr/pull/104
- Make sure DJANGO_CT and DJANGO_ID fields are not analyzed. [Honza Král]
- No need to store fields separately in elasticsearch. [Honza Král]

  That will justlead to fields being stored once - as part of _source as well as in separate index that would never be used by haystack (would be used only in special cases when requesting just that field, which can be, with minimal overhead, still just extracted from the _source as it is).
- Remove extra code. [Honza Král]
- Simplify mappings for elasticsearch fields. [Honza Král]
  - don't specify defaults (index:analyzed for strings, boost: 1.0)
  - omit extra settings that have little or negative effects (term_vector:with_positions_offsets)
  - only use type-specific settings (not_analyzed makes no sense for non-string types)

  Fixes #866
- Add narrow queries as individual subfilter to promote caching. [Honza Král]

  Each narrow query will be cached individually which means more cache reuse
- Doc formatting fix. [Honza Král]
- Allow users to pass in additional kwargs to Solr and Elasticsearch backends. [Honza Král]

  Fixes #674, #862
- Whoosh: allow multiple order_by() fields. [Chris Adams]

  The Whoosh backend previously prevented the use of more than one order_by field. It now allows multiple fields as long as every field uses the same sort direction.

Thanks to @qris, @overflow for the patch

Closes #627 Closes #919

- Fix bounding box calculation for spatial queries (closes #718) [Chris Adams]

  Thanks @jasisz for the fix

- Docs: fix ReST syntax error in searchqueryset_api.rst. [Chris Adams]
- Tests: update test_more_like_this for Solr 4.6. [Chris Adams]
- Tests: update test_quotes_regression exception test. [Chris Adams]

  This was previously relying on the assumption that a query would not match, which is Solr version dependent, rather than simply confirming that no exception is raised

- Tests: update Solr schema to match current build_solr_schema. [Chris Adams]
  - Added fields used in spatial tests: location, username, comment
  - Updated schema for recent Solr
  - Ran *xmllint –c14n "$*" | xmllint –format –encode "utf-8" -*
- Tests: update requirements to match tox. [Chris Adams]
- Move test Solr instructions into a script. [Chris Adams]

  These will just rot horribly if they're not actually executed on a regular basis...

- Merge pull request #907 from gam-phon/patch-1. [Chris Adams]
- Fix url for solr 3.5.0. [Yaser Alraddadi]
- Merge pull request #775 from stefanw/avoid-pks-seen-on-update. [Justin Caratzas]

  Avoid unnecessary, potentially huge db query on index update

- Merge branch 'master' into avoid-pks-seen-on-update. [Stefan Wehrmeyer]

  Change smart_text into smart_bytes as in master

  Conflicts: haystack/management/commands/update_index.py

- Upgraded python3 in tox to 3.3. [justin caratzas]

  3.3 is a better target for haystack than 3.2, due to PEP414

- Merge pull request #885 from HonzaKral/elasticsearch-py. [Justin Caratzas]

  Use elasticsearch-py instead of pyelasticsearch.

- Use elasticsearch-py instead of pyelasticsearch. [Honza Král]

  elasticsearch-py is the official Python client for Elasticsearch.

- Merge pull request #899 from acdha/html5-input-type=search. [Justin Caratzas]

  Search form <input type="search">

- Use HTML5 <input type=search> (closes #899) [Chris Adams]
- Update travis config so that unit tests will run with latest solr + elasticsearch. [justin caratzas]
- Merge remote-tracking branch 'HonzaKral/filtered_queries' Fixes #886. [Daniel Lindsley]
- Use terms filter for DJANGO_CT, *much* faster. [Honza Král]
- Cleaner query composition when it comes to filters in ES. [Honza Král]

- Fixed typo in AUTHORS. [justin caratzas]
- Added pabluk to AUTHORS. [Pablo SEMINARIO]
- Fixed ValueError exception when SILENTLY_FAIL=True. [Pablo SEMINARIO]
- Merge pull request #882 from benspaulding/docs/issue-607. [Justin Caratzas]

  Remove bit about SearchQuerySet.load_all_queryset deprecation
- Remove bit about SearchQuerySet.load_all_queryset deprecation. [Ben Spaulding]

  That method was entirely removed in commit b8048dc0e9e3.

  Closes #607. Thanks to @bradleyayers for the report.
- Merge pull request #881 from benspaulding/docs/issue-606. [Justin Caratzas]

  Fix documentation regarding ModelSearchIndex to match current behavior
- Fix documentation regarding ModelSearchIndex to match current behavior. [Ben Spaulding]

  Closes #606. Thanks to @bradleyayers for the report.
- Fixed #575 & #838, where a change in Whoosh 2.5> required explicitly setting the Searcher.search() limit to None to restore correct results. [Keryn Knight]

  Thanks to scenable and Shige Abe (typeshige) for the initial reports, and to scenable for finding the root issue in Whoosh.
- Removed python 1.4 / python 3.2 tox env because thats not possible. [justin caratzas]

  also pinned versions of requirements for testing
- Added test for autocomplete whitespace fix. [justin caratzas]
- Fixed autocomplete() method: spaces in query. [Ivan Virabyan]
- Fixed basepython for tox envs, thanks –showconfig. [justin caratzas]

  also, added latest django 1.4 release, which doesn't error out currently.

  Downgraded python3.3 to python3.2, as thats what the lastest debian stable includes. I'm working on compiling pypy and python3.3 on the test box, so those will probably be re-added as time allows.

  failing tests: still solr context extraction + spatial
- Fixed simple backend for django 1.6, _fields was removed. [justin caratzas]
- [tox] run tests for 1.6, fix test modules so they are found by the new test runner. [justin caratzas]

  These changes are backwards-compatible with django 1.5. As of this commit, the only failing tests are the Solr extractraction test, and the spatial tests.
- Switch solr configs to solr 4. [justin caratzas]

  almost all tests passing, but spatial not working
- Update solr schema template to fix stopwords_en.txt relocation. [Patrick Altman]

  Seems that in versions >3.6 and >4 stopwords_en.txt moved to a new location. This won't be backwards compatible for older versions of solr.

  Addresses issues #558, #560 In addition, issue #671 references this problem
- Pass *using* to index_queryset for update. [bigjust]
- Update tox to test pypy, py26, py27, py33, django1.5 and django1.6. [bigjust]

  django 1.6 doesn't actually work yet, but there are other efforts to get that working

- Fixed my own spelling test case. How embarrassing. [Dan Watson]
- Added a spelling test case for ElasticSearch. [Dan Watson]
- More ElasticSearch test fixes. [Dan Watson]
- Added some faceting tests for ElasticSearch. [Dan Watson]
- Fixed ordering issues in the ElasticSearch tests. [Dan Watson]
- Merge remote-tracking branch 'infoxchange/fix-elasticsearch-index- settings-reset' [Daniel Lindsley]
- Test ensuring recreating the index does not remove the mapping. [Alexey Kotlyarov]
- Reset backend state when deleting index. [Alexey Kotlyarov]

  Reset setup_complete and existing_mapping when an index is deleted. This ensures create_index is called later to restore the settings properly.
- Use Django's copy of six. [Dan Watson]
- Merge pull request #847 from luisbarrueco/mgmtcmd-fix. [Dan Watson]

  Fixed an update_index bug when using multiple connections
- Fixed an update_index bug when using multiple connections. [Luis Barrueco]
- Fixed a missed raw_input call on Python 3. [Dan Watson]
- Merge pull request #840 from postatum/fix_issue_807. [Justin Caratzas]

  Fixed issue #807
- Fixed issue #807. [postatum]
- Merge pull request #837 from nicholasserra/signals-docs-fix. [Justin Caratzas]

  Tiny docs fix in signal_processors example code
- Tiny docs fix in signal_processors example code. [Nicholas Serra]
- Merge pull request #413 from phill-tornroth/patch-1. [Justin Caratzas]

  Silly little change, I know.. but I actually ran into a case where I acci
- Silly little change, I know.. but I actually ran into a case where I accidentally passed a list of models in without *ing them. When that happens, we get a string formatting exception (not all arguments were formatted) instead of the useful "that ain't a model, kid" business. [Phill Tornroth]
- Merge pull request #407 from bmihelac/patch-1. [Justin Caratzas]

  Fixed doc, `query` is context variable and not in request.
- Fixed doc, `query` is context variable and not in request. [bmihelac]
- Merge pull request #795 from davesque/update_excluded_indexes_error_message. [Justin Caratzas]

  Improve error message for duplicate index classes
- Improve error message for duplicate index classes. [David Sanders]

  To my knowledge, the 'HAYSTACK_EXCLUDED_INDEXES' setting is no longer used.
- Started the v2.1.1 work. [Daniel Lindsley]
- Avoid unnecessary db query on index update. [Stefan Wehrmeyer]

  pks_seen is only needed if objects are removed from index, so only compute it if necessary. Improve pks_seen to not build an intermediary list.

v2.1.0 (2013-07-28)

- Bumped to v2.1.0! [Daniel Lindsley]
- Python 3 support is done, thanks to RevSys & the PSF! Updated requirements in the docs. [Daniel Lindsley]
- Added all the new additions to AUTHORS. [Daniel Lindsley]
- Merge branch 'py3' [Daniel Lindsley]
- Added Python 3 compatibility notes. [Daniel Lindsley]
- Whoosh mostly working under Python 3. See docs for details. [Daniel Lindsley]
- Backported things removed from Django 1.6. [Daniel Lindsley]
- Final core changes. [Daniel Lindsley]
- Solr tests all but passing under Py3. [Daniel Lindsley]
- Elasticsearch tests passing under Python 3. [Daniel Lindsley]

  Requires git master (ES 1.0.0 beta) to work properly when using suggestions.
- Overrides passing under Py3. [Daniel Lindsley]
- Simple backend ported & passing. [Daniel Lindsley]
- Whoosh all but fully working under Python 3. [Daniel Lindsley]
- Closer on porting ES. [Daniel Lindsley]
- Core tests mostly pass on Py 3. o/ [Daniel Lindsley]

  What's left are 3 failures, all ordering issues, where the correct output is present, but ordering is different between Py2 / Py3.
- More porting to Py3. [Daniel Lindsley]
- Started porting to py3. [Daniel Lindsley]
- Merge pull request #821 from knightzero/patch-1. [Justin Caratzas]

  Update autocomplete.rst
- Update autocomplete.rst. [knightzero]
- Merge pull request #744 from trigger-corp/master. [Justin Caratzas]

  Allow for document boosting with elasticsearch
- Update the current elasticsearch boost test to also test document boosting. [Connor Dunn]
- Map boost field to _boost in elasticsearch. [Connor Dunn]

  Means that including a boost field in a document will cause document level boosting.
- Added ethurgood to AUTHORS. [Daniel Lindsley]
- Add test__to_python for elastisearch backend. [Eric Thurgood]
- Fix datetime instantiation in elasticsearch backend's _to_python. [Eric Thurgood]
- Merge pull request #810 from pabluk/minor-docs-fix. [Chris Adams]

  Updated description for TIMEOUT setting - thanks @pabluk
- Updated description for TIMEOUT setting. [Pablo SEMINARIO]
- Updated the backend support docs. Thanks to kezabelle & dimiro1 for the report! [Daniel Lindsley]

- Added haystack-rqueue to “Other Apps”. [Daniel Lindsley]
- Updated README & index. [Daniel Lindsley]
- Added installation instructions. [bigjust]
- Merge pull request #556 from h3/master. [Justin Caratzas]

  Updated to ‘xapian_backend.XapianEngine’ docs & example
- Updated XapianEngine module path. [h3]
- Updated XapianEngine module path. [h3]
- Merge pull request #660 from seldon/master. [Justin Caratzas]

  Some minor docs fixes
- Fixed a few typos in docs. [Lorenzo Franceschini]
- Add Educreations to who uses Haystack. [bigjust]
- Merge pull request #692 from stephenpaulger/master. [Justin Caratzas]

  Change the README link to latest 1.2 release.
- Update README.rst. [Stephen Paulger]

  Update 1.2.6 link to 1.2.7
- Merge pull request #714 from miracle2k/patch-1. [Justin Caratzas]

  Note enabling INCLUDE_SPELLING requires a reindex.
- Note enabling INCLUDE_SPELLING requires a reindex. [Michael Elsdörfer]
- Unicode support in SimpleSearchQuery (closes #793) [slollo]
- Merge pull request #790 from andrewschoen/feature/haystack-identifier- module. [Andrew Schoen]

  Added a new setting, HAYSTACK_IDENTIFIER_METHOD, which will allow a cust...
- Added a new setting, `HAYSTACK_IDENTIFIER_METHOD`, which will allow a custom method to be provided for `haystack.utils.get_identifier`. [Schoen]
- Fixed an exception log message in elasticsearch backend, and added a loading test for elasticsearch. [Dan Watson]
- Changed exception log message in whoosh backend to use __class__.__name__ instead of just __name__ (closes #641) [Jeffrey Tratner]
- Further bumped the docs on installing engines. [Daniel Lindsley]
- Update docs/installing_search_engines.rst. [Tom Dyson]

  grammar, Elasticsearch version and formatting consistency fixes.
- Added GroundCity & Docket Alarm to the Who Uses docs. [Daniel Lindsley]
- Started the development on v2.0.1. [Daniel Lindsley]

v2.0.0 (2013-05-12)

- Bumped to v2.0.0! [Daniel Lindsley]
- Changed how `Raw` inputs are handled. Thanks to kylemacfarlane for the (really good) report. [Daniel Lindsley]
- Added a (passing) test trying to verify #545. [Daniel Lindsley]

- Fixed a doc example on custom forms. Thanks to GrivIN and benspaulding for patches. [Daniel Lindsley]
- Added a reserved character for Solr (v4+ supports regexes). Thanks to RealBigB for the initial patch. [Daniel Lindsley]
- Merge branch 'master' of github.com:toastdriven/django-haystack. [Jannis Leidel]
- Fixed the stats tests. [Daniel Lindsley]
- Adding description of stats support to docs. [Ranjit Chacko]
- Adding support for stats queries in Solr. [Ranjit Chacko]
- Added tests for the previous kwargs patch. [Daniel Lindsley]
- Bug fix to allow object removal without a commit. [Madan Thangavelu]
- Do not refresh the index after it has been deleted. [Kevin Tran]
- Fixed naming of manager for consistency. [Jannis Leidel]
  - renamed *HaystackManager* to *SearchIndexManager*
  - renamed *get_query_set* to *get_search_queryset*
- Updated the docs on running tests. [Daniel Lindsley]
- Merge branch 'madan' [Daniel Lindsley]
- Fixed the case where index_name isn't available. [Daniel Lindsley]
- Fixing typo to allow manager to switch between different index_labels. [Madan Thangavelu]
- Haystack manager and tests. [Madan Thangavelu]
- Removing unwanted spaces. [Madan Thangavelu]
- Object query manager for searchindex. [Madan Thangavelu]
- Added requirements file for testing. [Daniel Lindsley]
- Added a unit test for #786. [Dan Watson]
- Fixed a bug when passing "using" to SearchQuerySet (closes #786). [Rohan Gupta]
- Ignore the env directory. [Daniel Lindsley]
- Allow for setuptools as well as distutils. [Daniel Lindsley]
- Merge pull request #785 from mattdeboard/dev-mailing-list. [Chris Adams]

  Add note directing users to django-haystack-dev mailing list.
- Add note directing users to django-haystack-dev mailing list. [Matt DeBoard]
- Spelling suggestions for ElasticSearch (closes #769 and #747) [Dan Watson]
- Added support for sending facet options to the backend (closes #753) [Dan Watson]
- More_like_this: honor .models() restriction. [Chris Adams]

  Original patch by @mattdeboard updated to remove test drift since it was originally submitted

  Closes #593 Closes #543
- Removed commercial support info. [Daniel Lindsley]
- Merge pull request #779 from pombredanne/pep386_docfixes. [Jannis Leidel]

  Update version to 2.0.0b0 in doc conf

- Update version to 2.0.0b0 in doc conf .. to redeem myself of the unlucky #777 minimess. [pombredanne]
- Merge pull request #778 from falinsky/patch-1. [Justin Caratzas]

  Fix bug in setup.py
- Fix bug. [Sergey Falinsky]
- Merge pull request #777 from pombredanne/patch-1. [Justin Caratzas]

  Update version to be a PEP386 strict with a minor qualifier of 0 for now...
- Update version to be a PEP386 strict with a minor qualifier of 0 for now. [pombredanne]

  This version becomes a "strict" version under PEP386 and should be recognized by install/packaging tools (such as distribute/distutils/setuptools) as newer than 2.0.0-beta. This will also help making small increments of the version which brings some sanity when using an update from HEAD and ensure that things will upgrade alright.
- Update_index: display Unicode model names (closes #767) [Chris Adams]

  The model's verbose_name_plural value is included as Unicode but under Python 2.x the progress message it was included in was a regular byte-string. Now it's correctly handled as Unicode throughout.
- Merge pull request #731 from adityar7/master. [Jannis Leidel]

  Setup custom routers before settings up signal processor.
- Setup custom routers before settings up signal processor. [Aditya Rajgarhia]

  Fixes https://github.com/toastdriven/django-haystack/issues/727
- Port the *from_python* method from pyelasticsearch to the Elasticsearch backend, similar to *to_python* in 181bbc2c010a135b536e4d1f7a1c5ae4c63e33db. [Jannis Leidel]

  Fixes #762. Refs #759.
- Merge pull request #761 from stefanw/simple-models-filter. [Justin Caratzas]

  Make models filter work on simple backend
- Make model filter for simple backend work. [Stefan Wehrmeyer]

  Adds Stefan Wehrmeyer to AUTHORS for patch
- Merge pull request #746 from lazerscience/fix-update-index-output. [Justin Caratzas]

  Using force_text for indexing message
- Replacing *force_text* with *force_unicode*. #746. [Bernhard Vallant]
- Using force_text for indexing message. [Bernhard Vallant]

  verbose_name_plural may be a functional proxy object from ugettext_lazy, it should be forced to be a string!
- Support pyelasticsearch 0.4 change (closes #759) [Chris Adams]

  pyelasticsearch 0.4 removed the *to_python* method Haystack used.

  Thanks to @erikrose for the quick patch
- Merge pull request #755 from toastdriven/issue/754-doc-build-warning. [Chris Adams]
- Add preceding dots to hyperlink target; fixes issue 754. [Ben Spaulding]

  This error was introduced in commit faacbcb.
- Merge pull request #752 from bigjust/master. [Justin Caratzas]

  Fix Simple Score field collision

- Simple: Fix bug in score field collision. [bigjust]

  Previous commit 0a9c919 broke the simple backend for models that didn't have an indexed score field. Added a test to cover regression.

- Set zip_safe in setup.py to prevent egg creation. [Jannis Leidel]

  This is a work around for a bug in Django that prevents detection of management commands embedded in packages installed as setuptools eggs.

- Merge pull request #740 from acdha/simplify-search-view-name-property. [Chris Adams]

  Remove redundant __name__ assignment on SearchView

- Remove redundant __name__ assignment on SearchView. [Chris Adams]

  __name__ was being explicitly set to a value which was the same as the default value.

  Additionally corrected the obsolete __name__ method declaration in the documentation which reflected the code prior to SHA:89d8096 in 2010.

- Merge pull request #698 from gjb83/master. [Chris Adams]

  Fixed deprecation warning for url imports on Django 1.3

  Thanks to @gjb83 for the patch.

- Removed star imports. [gjb83]
- Maintain Django 1.3 compatibility. [gjb83]
- Fixed deprecation warning. [gjb83]

  django.conf.urls.defaults is now deprecated. Use django.conf.urls instead.

- Merge pull request #743 from bigjust/solr-managementcmd-fix. [Justin Caratzas]

  Solr build_solr_schema: fixed a bug in build_solr_schema. Thanks to mjum...

- Solr build_solr_schema: fixed a bug in build_solr_schema. Thanks to mjumbewu for the report! [Justin Caratzas]

  If you tried to run build_solr_schema with a backend that supports schema building, but was not Solr (like Whoosh), then you would get an invalid schema. This fix raises the ImproperlyConfigured exception with a proper message.

- Merge pull request #742 from bigjust/simple-backend-score-fix. [Justin Caratzas]
- Simple: removed conflicting score field from raw result objects. [Justin Caratzas]

  This keeps consistency with the Solr backend, which resolves this conflict in the same manner.

- ElasticSearch: fix AltParser test. [Chris Adams]

  AltParser queries are still broken but that fucntionality has only been listed as supported on Solr.

- Better Solr AltParser quoting (closes #730) [Chris Adams]

  Previously the Solr AltParser implementation embedded the search term as an attribte inside the {!...} construct, which required it to be doubly escaped.

  This change contributed by @ivirabyan moves the value outside the query, requiring only our normal quoting:

  ```
  q=(_query_:"{!edismax}Assassin's Creed")
  ```

  instead of:

  ```
  q=(_query_:"{!edismax v='Assassin's Creed'}")
  ```

Thanks @ivirabyan for the patch!

- Solr: use nested query syntax for AltParser queries. [Chris Adams]

  The previous implementation would, given a query like this:

```
sqs.filter(content=AltParser('dismax', 'library', qf="title^2 text" mm=1))
```

  generate a query like this:

```
{!dismax v=library qf="title^2 text" mm=1}
```

  This works in certain situations but causes Solr to choke while parsing it when Haystack wraps this term in parentheses:

```
org.apache.lucene.queryParser.ParseException: Cannot parse '({!dismax mm=1 qf=
↪'title^2 text institution^0.8' v=library})':
Encountered " &lt;RANGEEX_GOOP&gt; "qf=\'title^1.25 "" at line 1, column 16.
```

  The solution is to use the nested query syntax described here:

  http://searchhub.org/2009/03/31/nested-queries-in-solr/

  This will produce a query like this, which works with Solr 3.6.2:

```
(_query_:"{!edismax mm=1 qf='title^1.5 text institution^0.5' v=library}")
```

  Leaving the actual URL query string looking like this:

```
q=%28_query_%3A%22%7B%21edismax+mm%3D1+qf%3D%27title%5E1.5+text+institution%5E0.5
↪%27+v%3Dlibrary%7D%22%29
```

  - Tests updated for the new query generation output
  - A Solr backend task was added to actually run the dismax queries and verify that we're not getting Solr 400s errors due to syntax gremlins

- Pass active backend to index queryset calls (closes #534) [Chris Adams]

  Now the Index index_queryset() and read_queryset() methods will be called with the active backend name so they can optionally perform backend-specific filtering.

  This is extremely useful when using something like Solr cores to maintain language specific backends, allowing an Index to select the appropriate documents for each language:

```
def index_queryset(self, using=None):
    return Post.objects.filter(language=using)
```

  Changes:

  - clear_index, update_index and rebuild_index all default to processing *every* backend. `--using` may now be provided multiple times to select a subset of the configured backends.
  - Added examples to the Multiple Index documentation page

- Because Windows. [Daniel Lindsley]
- Fixed the docs on debugging to cover v2. Thanks to eltesttox for the report. [Daniel Lindsley]
- That second colon matters. [Daniel Lindsley]
- Further docs on autocomplete. [Daniel Lindsley]

- Fixed the imports that would stomp on each other. [Daniel Lindsley]

  Thanks to codeinthehole, Attorney-Fee & imacleod for pointing this out.

- BACKWARD-INCOMPATIBLE: Removed `RealTimeSearchIndex` in favor of `SignalProcessors`. [Daniel Lindsley]

  This only affects people who were using `RealTimeSearchIndex` (or a queuing variant) to perform near real-time updates. Those users should refer to the Migration documentation.

- Updated ignores. [Daniel Lindsley]
- Merge pull request #552 from hadesgames/master. [Jannis Leidel]

  Fixes process leak when using update_index with workers.

- Fixed update_index process leak. [Tache Alexandru]
- Merge branch 'master' of github.com:toastdriven/django-haystack. [Jannis Leidel]
- Merge pull request #682 from acdha/682-update_index-tz-support. [Chris Adams]

  update_index should use non-naive datetime when settings.USE_TZ=True

- Tests for update_index timezone support. [Chris Adams]
  - Confirm that update_index –age uses the Django timezone-aware now support function
  - Skip this test on Django 1.3
- Update_index: use tz-aware datetime where applicable. [Chris Adams]

  This will allow Django 1.4 users with USE_TZ=True to use update_index with time windowing as expected - otherwise the timezone offset needs to be manually included in the value passed to -a

- Tests: mark expected failures in Whoosh suite. [Chris Adams]

  This avoids making it painful to run the test suite and flags the tests which need attention

- Tests: mark expected failures in ElasticSearch suite. [Chris Adams]

  This avoids making it painful to run the test suite and flags the tests which need attention

- Multiple index tests: correct handling of Whoosh teardown. [Chris Adams]

  We can't remove the Whoosh directory per-test - only after every test has run...

- Whoosh tests: use a unique tempdir. [Chris Adams]

  This ensures that there's no way for results to persist across runs and lets the OS clean up the mess if we fail catastrophically

  The multiindex and regular whoosh tests will have different prefixes to ease debugging

- Merge pull request #699 from acdha/tox-multiple-django-versions. [Chris Adams]

  Minor tox.ini & test runner tidying

- Test runner: set exit codes on failure. [Chris Adams]
- Tox: refactor envlist to include Django versions. [Chris Adams]
  - Expanded base dependencies
  - Set TEST_RUNNER_ARGS=-v0 to reduce console noise
  - Add permutations of python 2.5, 2.6, 2.7 and django 1.3 and 1.4

- Test runner: add $TEST_RUNNER_ARGS env. variable. [Chris Adams]

  This allows you to export TEST_RUNNER_ARGS=-v0 to affect all 9 invocations

- Tox: store downloads in tmpdir. [Chris Adams]
- Be a bit more careful when resetting connections in the multiprocessing updater. Fixes #562. [Jannis Leidel]
- Fixed distance handling in result parser of the elasticsearch backend. This is basically the second part of #566. Thanks to Josh Drake for the initial patch. [Jannis Leidel]
- Merge pull request #670 from dhan88/master. [Jannis Leidel]

  Elasticsearch backend using incorrect coordinates for geo_bounding_box (within) filter

- Elasticsearch geo_bounding_box filter expects top_left (northwest) and bottom_right (southeast). Haystack's elasticsearch backend is passing northeast and southwest coordinates instead. [Danny Han]
- Merge pull request #666 from caioariede/master. [Jannis Leidel]

  Fixes incorrect call to put_mapping on ElasticSearch backend

- Fixes incorrect call to put_mapping on elasticsearch backend. [Caio Ariede]
- Added ericholscher to AUTHORS. [Daniel Lindsley]
- Add a title for the support matrix so it's linkable. [Eric Holscher]
- Tests: command-line help and coverage.py support. [Chris Adams]

  This makes run_all_tests.sh a little easier to use and simplifies the process of running under coverage.py

  Closes #683

- Tests: basic help and coverage.py support. [Chris Adams]

  run_all_tests.sh now supports –help and –with-coverage

- Add a CONTRIBUTING.md file for Github. [Chris Adams]

  This is a migrated copy of docs/contributing.rst so Github can suggest it when pull requests are being created

- Fix combination logic for complex queries. [Chris Adams]

  Previously combining querysets which used a mix of logical AND and OR operations behaved unexpectedly.

  Thanks to @mjl for the patch and tests in SHA: 9192dbd

  Closes #613, #617

- Added rz to AUTHORS. [Daniel Lindsley]
- Fixed string joining bug in the simple backend. [Rodrigo Guzman]
- Added failing test case for #438. [Daniel Lindsley]
- Fix Solr more-like-this tests (closes #655) [Chris Adams]
  - Refactored the MLT tests to be less brittle in checking only the top 5 results without respect to slight ordering variations.
  - Refactored LiveSolrMoreLikeThisTestCase into multiple tests
  - Convert MLT templatetag tests to rely on mocks for stability and to avoid hard-coding backend assumptions, at the expense of relying completely on the backend MLT queryset-level tests to exercise that code.
  - Updated MLT code to always assume deferred querysets are available (introduced in Django 1.1) and removed a hard-coded internal attr check

- All backends: fixed more_like_this & deferreds. [Chris Adams]

  Django removed the get_proxied_model helper function in the 1.3 dev cycle:

  https://code.djangoproject.com/ticket/17678

  This change adds support for the simple new property access used by 1.3+

  BACKWARD INCOMPATIBLE: Django 1.2 is no longer supported
- Updated elasticsearch backend to use a newer pyelasticsearch release that features an improved API , connection pooling and better exception handling. [Jannis Leidel]
- Added Gidsy to list of who uses Haystack. [Jannis Leidel]
- Increased the number of terms facets returned by the Elasticsearch backend to 100 from the default 10 to work around an issue upstream. [Jannis Leidel]

  This is hopefully only temporary until it's fixed in Elasticsearch, see https://github.com/elasticsearch/elasticsearch/issues/1776.
- Merge pull request #643 from stephenmcd/master. [Chris Adams]

  Fixed logging in simple_backend
- Fixed logging in simple_backend. [Stephen McDonald]
- Added Pitchup to Who Uses. [Daniel Lindsley]
- Merge branch 'unittest2-fix' [Chris Adams]
- Better unittest2 detection. [Chris Adams]

  This supports Python 2.6 and earlier by shifting the import to look towards the future name rather than the past
- Merge pull request #652 from acdha/solr-content-extraction-test-fix. [Chris Adams]

  Fix the Solr content extraction handler tests
- Add a minimal .travis.yml file to suppress build spam. [Chris Adams]

  Until the travis-config branch is merged in, this can be spread around to avoid wasting time running builds before we're ready
- Tests: enable Solr content extraction handler. [Chris Adams]

  This is needed for the test_content_extraction test to pass
- Tests: Solr: fail immediately on config errors. [Chris Adams]
- Solr tests: clean unused imports. [Chris Adams]
- Suppress console DeprecationWarnings. [Chris Adams]
- Merge pull request #651 from acdha/unittest2-fix. [Chris Adams]

  Update unittest2 import logic so the tests can actually be run
- Update unittest2 import logic. [Chris Adams]

  We'll try to get it from Django 1.3+ but Django 1.2 users will need to install it manually
- Merge pull request #650 from bigjust/patch-1. [Chris Adams]

  Fix typo in docstring
- Fix typo. [Justin Caratzas]
- Refactor to use a dummy logger that lets you turn off logging. [Travis Swicegood]

- A bunch of Solr testing cleanup. [Chris Adams]
- Skip test is pysolr isn't available. [Travis Swicegood]
- Updated Who Uses to correct a backend usage. [Daniel Lindsley]
- Updated documentation about using the main pyelasticsearch release. [Jannis Leidel]
- Merge pull request #628 from kjoconnor/patch-1. [Jannis Leidel]

  Missing '
- Missing ' [Kevin O'Connor]
- Fixed a mostly-empty warning in the `SearchQuerySet` docs. Thanks to originell for the report! [Daniel Lindsley]
- Fixed the "Who Uses" entry on AstroBin. [Daniel Lindsley]
- Use the match_all query to speed up performing filter only queries dramatically. [Jannis Leidel]
- Fixed typo in docs. Closes #612. [Jannis Leidel]
- Updated link to celery-haystack repository. [Jannis Leidel]
- Fixed the docstring of SearchQuerySet.none. Closes #435. [Jannis Leidel]
- Fixed the way quoting is done in the Whoosh backend when using the `__in` filter. [Jason Kraus]
- Added the solrconfig.xml I use for testing. [Daniel Lindsley]
- Fixed typo in input types docs. Closes #551. [Jannis Leidel]
- Make sure an search engine's backend isn't instantiated on every call to the backend but only once. Fixes #580. [Jannis Leidel]
- Restored sorting to ES backend that was broken in d1fa95529553ef8d053308159ae4efc455e0183f. [Jannis Leidel]
- Prevent spatial filters from stomping on existing filters in ElasticSearch backend. [Josh Drake]
- Merge branch 'mattdeboard-sq-run-refactor' [Jannis Leidel]
- Fixed an ES test that seems like a change in behavior in recent ES versions. [Jannis Leidel]
- Merge branch 'sq-run-refactor' of https://github.com/mattdeboard /django-haystack into mattdeboard-sq-run-refactor. [Jannis Leidel]
- Refactor Solr & ES SearchQuery subclasses to use the `build_params` from `BaseSearchQuery` to build the kwargs to be passed to the search engine. [Matt DeBoard]

  This refactor is made to make extending Haystack simpler. I only ran the Solr tests which invoked a `run` call (via `get_results`), and those passed. I did not run the ElasticSearch tests; however, the `run` method for both Lucene-based search engines were identical before, and are identical now. The test I did run – `LiveSolrSearchQueryTestCase.test_log_query` – passed.
- Merge branch 'master' of https://github.com/toastdriven/django- haystack. [Jannis Leidel]
- Merge pull request #568 from duncm/master. [Jannis Leidel]

  Fix exception in SearchIndex.get_model()
- Fixed `SearchIndex.get_model()` to raise exception instead of returning it. [Duncan Maitland]
- Merge branch 'master' of https://github.com/toastdriven/django- haystack. [Jannis Leidel]
- Fixed Django 1.4 compatibility. Thanks to bloodchild for the report! [Daniel Lindsley]

- Refactored `SearchBackend.search` so that kwarg-generation operations are in a discrete method. [Matt DeBoard]

  This makes it much simpler to subclass `SearchBackend` (& the engine-specific variants) to add support for new parameters.

- Added witten to AUTHORS. [Daniel Lindsley]
- Fix for #378: Highlighter returns unexpected results if one term is found within another. [dan]
- Removed jezdez's old entry in AUTHORS. [Daniel Lindsley]
- Added Jannis to Primary Authors. [Daniel Lindsley]
- Merge branch 'master' of github.com:jezdez/django-haystack. [Jannis Leidel]
- Fixed a raise condition when using the simple backend (e.g. in tests) and changing the DEBUG setting dynamically (e.g. in integration tests). [Jannis Leidel]
- Add missing *ImproperlyConfigured* import from django's exceptions. [Luis Nell]

  l178 failed.

- Commercial support is now officially available for Haystack. [Daniel Lindsley]
- Using multiple workers (and resetting the connection) causes things to break when the app is finished and it moves to the next and does qs.count() to get a count of the objects in that app to index with psycopg2 reporting a closed connection. Manually closing the connection before each iteration if using multiple workers before building the queryset fixes this issue. [Adam Fast]
- Removed code leftover from v1.X. Thanks to kossovics for the report! [Daniel Lindsley]
- Fixed a raise condition when using the simple backend (e.g. in tests) and changing the DEBUG setting dynamically (e.g. in integration tests). [Jannis Leidel]
- All backends let individual documents fail, rather than failing whole chunks. Forward port of acdha's work on 1.2.X. [Daniel Lindsley]
- Added ikks to AUTHORS. [Daniel Lindsley]
- Fixed `model_choices` to use `smart_unicode`. [Igor Tamara]
- +localwiki.org. [Philip Neustrom]
- Added Pix Populi to "Who Uses". [Daniel Lindsley]
- Added contribution guidelines. [Daniel Lindsley]
- Updated the docs to reflect the supported version of Django. Thanks to catalanojuan for the original patch! [Daniel Lindsley]
- Fix PYTHONPATH Export and add Elasticsearch example. [Craig Nagy]
- Updated the Whoosh URL. Thanks to cbess for the original patch! [Daniel Lindsley]
- Reset database connections on each process on update_index when using –workers. [Diego Búrigo Zacarão]
- Moved the `build_queryset` method to `SearchIndex`. [Alex Vidal]

  This method is used to build the queryset for indexing operations. It is copied from the build_queryset function that lived in the update_index management command.

  Making this change allows developers to modify the queryset used for indexing even when a date filter is necessary. See *tests/core/indexes.py* for tests.

- Fixed a bug where `Indexable` could be mistakenly recognized as a discoverable class. Thanks to twoolie for the original patch! [Daniel Lindsley]

- Fixed a bug with query construction. Thanks to dstufft for the report! [Daniel Lindsley]

  This goes back to erroring on the side of too many parens, where there weren't enough before. The engines will no-op them when they're not important.

- Fixed a bug where South would cause Haystack to setup too soon. Thanks to adamfast for the report! [Daniel Lindsley]
- Added Crate.io to "Who Uses"! [Daniel Lindsley]
- Fixed a small typo in spatial docs. [Frank Wiles]
- Logging: avoid forcing string interpolation. [Chris Adams]
- Fixed docs on using a template for Solr schema. [Daniel Lindsley]
- Add note to 'Installing Search Engines' doc explaining how to override the template used by 'build_solr_schema' [Matt DeBoard]
- Better handling of `.models`. Thanks to zbyte64 for the report & HonzaKral for the original patch! [Daniel Lindsley]
- Added Honza to AUTHORS. [Daniel Lindsley]
- Handle sorting for ElasticSearch better. [Honza Kral]
- Update docs/backend_support.rst. [Issac Kelly]
- Fixed a bug where it's possible to erroneously try to get spelling suggestions. Thanks to bigjust for the report! [Daniel Lindsley]
- The `dateutil` requirement is now optional. Thanks to arthurnn for the report. [Daniel Lindsley]
- Fixed docs on Solr spelling suggestion until the new Suggester support can be added. Thanks to zw0rk & many others for the report! [Daniel Lindsley]
- Bumped to beta. [Daniel Lindsley]

  We're not there yet, but we're getting close.

- Added saved-search to subproject docs. [Daniel Lindsley]
- Search index discovery no longer swallows errors with reckless abandon. Thanks to denplis for the report! [Daniel Lindsley]
- Elasticsearch backend officially supported. [Daniel Lindsley]

  All tests passing.

- Back down to 3 on latest pyelasticsearch. [Daniel Lindsley]
- And then there were 3 (Elasticsearch test failures). [Daniel Lindsley]
- Solr tests now run faster. [Daniel Lindsley]
- Improved the tutorial docs. Thanks to denplis for the report! [Daniel Lindsley]
- Down to 9 failures on Elasticsearch. [Daniel Lindsley]
- Because the wishlist has changed. [Daniel Lindsley]
- A few small fixes. Thanks to robhudson for the report! [Daniel Lindsley]
- Added an experimental Elasticsearch backend. [Daniel Lindsley]

  Tests are not yet passing but it works in basic hand-testing. Passing test coverage coming soon.

- Fixed a bug related to the use of `Exact`. [Daniel Lindsley]

- Removed accidental indent. [Daniel Lindsley]
- Ensure that importing fields without the GeoDjango kit doesn't cause an error. Thanks to dimamoroz for the report! [Daniel Lindsley]
- Added the ability to reload a connection. [Daniel Lindsley]
- Fixed `rebuild_index` to properly have all options available. [Daniel Lindsley]
- Fixed a bug in pagination. Thanks to sgoll for the report! [Daniel Lindsley]
- Added an example to the docs on what to put in `INSTALLED_APPS`. Thanks to Dan Krol for the suggestion. [Daniel Lindsley]
- Changed imports so the geospatial modules are only imported as needed. [Dan Loewenherz]
- Better excluded index detection. [Daniel Lindsley]
- Fixed a couple of small typos. [Sean Bleier]
- Made sure the toolbar templates are included in the source distribution. [Jannis Leidel]
- Fixed a few documentation issues. [Jannis Leidel]
- Moved my contribution for the geospatial backend to a attribution of Gidsy which funded my work. [Jannis Leidel]
- Small docs fix. [Daniel Lindsley]
- Added input types, which enables advanced querying support. Thanks to CMGdigital for funding the development! [Daniel Lindsley]
- Added geospatial search support! [Daniel Lindsley]

  I have anxiously waited to add this feature for almost 3 years now. Support is finally present in more than one backend & I was generously given some paid time to work on implementing this.

  Thanks go out to:

  - CMGdigital, who paid for ~50% of the development of this feature & were awesomely supportive.
  - Jannis Leidel (jezdez), who did the original version of this patch & was an excellent sounding board.
  - Adam Fast, for patiently holding my hand through some of the geospatial confusions & for helping me verify GeoDjango functionality.
  - Justin Bronn, for the great work he originally did on GeoDjango, which served as a point of reference/inspiration on the API.

  And thanks to all others who have submitted a variety of patches/pull requests/interest throughout the years trying to get this feature in place.
- Added .values() / .values_list() methods, for fetching less data. Thanks to acdha for the original implementation! [Daniel Lindsley]
- Reduced the number of queries Haystack has to perform in many cases (pagination/facet_counts/spelling_suggestions). Thanks to acdha for the improvements! [Daniel Lindsley]
- Spruced up the layout on the new DjDT panel. [Daniel Lindsley]
- Fixed compatibility with Django pre-1.4 trunk. * The MAX_SHOW_ALL_ALLOWED variable is no longer available, and hence causes an ImportError with Django versions higher 1.3. * The "list_max_show_all" attribute on the ChangeList object is used instead. * This patch maintains compatibility with Django 1.3 and lower by trying to import the MAX_SHOW_ALL_ALLOWED variable first. [Aram Dulyan]
- Updated `setup.py` for the new panel bits. [Daniel Lindsley]

- Added a basic DjDT panel for Haystack. Thanks to robhudson for planting the seed that Haystack should bundle this! [Daniel Lindsley]
- Added the ability to specify apps or individual models to `update_index`. Thanks to CMGdigital for funding this development! [Daniel Lindsley]
- Added `--start/--end` flags to `update_index` to allow finer- grained control over date ranges. Thanks to CMGdigital for funding this development! [Daniel Lindsley]
- I hate Python packaging. [Daniel Lindsley]
- Made `SearchIndex` classes thread-safe. Thanks to craigds for the report & original patch. [Daniel Lindsley]
- Added a couple more uses. [Daniel Lindsley]
- Bumped reqs in docs for content extraction bits. [Daniel Lindsley]
- Added a long description for PyPI. [Daniel Lindsley]
- Solr backend support for rich-content extraction. [Chris Adams]

  This allows indexes to use text extracted from binary files as well as normal database content.
- Fixed errant `self.log`. [Daniel Lindsley]

  Thanks to terryh for the report!
- Fixed a bug with index inheritance. [Daniel Lindsley]

  Fields would seem to not obey the MRO while method did. Thanks to ironfroggy for the report!
- Fixed a long-time bug where the Whoosh backend didn't have a `log` attribute. [Daniel Lindsley]
- Fixed a bug with Whoosh's edge n-gram support to be consistent with the implementation in the other engines. [Daniel Lindsley]
- Added celery-haystack to Other Apps. [Daniel Lindsley]
- Changed `auto_query` so it can be run on other, non-`content` fields. [Daniel Lindsley]
- Removed extra loops through the field list for a slight performance gain. [Daniel Lindsley]
- Moved `EXCLUDED_INDEXES` to a per-backend setting. [Daniel Lindsley]
- BACKWARD-INCOMPATIBLE: The default filter is now `__contains` (in place of `__exact`). [Daniel Lindsley]

  If you were relying on this behavior before, simply add `__exact` to the fieldname.
- BACKWARD-INCOMPATIBLE: All "concrete" `SearchIndex` classes must now mixin `indexes.Indexable` as well in order to be included in the index. [Daniel Lindsley]
- Added tox to the mix. [Daniel Lindsley]
- Allow for less configuration. Thanks to jeromer & cyberdelia for the reports! [Daniel Lindsley]
- Fixed up the management commands to show the right alias & use the default better. Thanks to jeromer for the report! [Daniel Lindsley]
- Fixed a bug where signals wouldn't get setup properly, especially on `RealTimeSearchIndex`. Thanks to byoungb for the report! [Daniel Lindsley]
- Fixed formatting in the tutorial. [Daniel Lindsley]
- Removed outdated warning about padding numeric fields. Thanks to mchaput for pointing this out! [Daniel Lindsley]

- Added a silent failure option to prevent Haystack from suppressing some failures. [Daniel Lindsley]

  This option defaults to `True` for compatibility & to prevent cases where lost connections can break reindexes/searches.

- Fixed the simple backend to not throw an exception when handed an `SQ`. Thanks to diegobz for the report! [Daniel Lindsley]
- Whoosh now supports More Like This! Requires Whoosh 1.8.4. [Daniel Lindsley]
- Deprecated `get_queryset` & fixed how indexing happens. Thanks to Craig de Stigter & others for the report! [Daniel Lindsley]
- Fixed a bug where `RealTimeSearchIndex` was erroneously included in index discovery. Thanks to dedsm for the report & original patch! [Daniel Lindsley]
- Added Vickery to "Who Uses". [Daniel Lindsley]
- Require Whoosh 1.8.3+. It's for your own good. [Daniel Lindsley]
- Added multiprocessing support to `update_index`! Thanks to CMGdigital for funding development of this feature. [Daniel Lindsley]
- Fixed a bug where `set` couldn't be used with `__in`. Thanks to Kronuz for the report! [Daniel Lindsley]
- Added a `DecimalField`. [Daniel Lindsley]
- Fixed a bug where a different style of import could confuse the collection of indexes. Thanks to groovecoder for the report. [Daniel Lindsley]
- Fixed a typo in the autocomplete docs. Thanks to anderso for the catch! [Daniel Lindsley]
- Fixed a backward-incompatible query syntax change Whoosh introduced between 1.6.1 & 1.6.2 that causes only one model to appear as though it is indexed. [Daniel Lindsley]
- Updated AUTHORS to reflect the Kent's involvement in multiple index support. [Daniel Lindsley]
- BACKWARD-INCOMPATIBLE: Added multiple index support to Haystack, which enables you to talk to more than one search engine in the same codebase. Thanks to: [Daniel Lindsley]
  - Kent Gormat for funding the development of this feature.
  - alex, freakboy3742 & all the others who contributed to Django's multidb feature, on which much of this was based.
  - acdha for inspiration & feedback.
  - dcramer for inspiration & feedback.
  - mcroydon for patch review & docs feedback.

  This commit starts the development efforts for Haystack v2.

v1.2.7 (2012-04-06)

- Bumped to v1.2.7! [Daniel Lindsley]
- Solr: more informative logging when full_prepare fails during update. [Chris Adams]
  - Change the exception handler to record per-object failures
  - Log the precise object which failed in a manner which tools like Sentry can examine
- Added ikks to AUTHORS. [Daniel Lindsley]
- Fixed `model_choices` to use `smart_unicode`. Thanks to ikks for the patch! [Daniel Lindsley]

- Fixed compatibility with Django pre-1.4 trunk. * The MAX_SHOW_ALL_ALLOWED variable is no longer available, and hence causes an ImportError with Django versions higher 1.3. * The "list_max_show_all" attribute on the ChangeList object is used instead. * This patch maintains compatibility with Django 1.3 and lower by trying to import the MAX_SHOW_ALL_ALLOWED variable first. [Aram Dulyan]
- Fixed a bug in pagination. Thanks to sgoll for the report! [Daniel Lindsley]
- Added an example to the docs on what to put in `INSTALLED_APPS`. Thanks to Dan Krol for the suggestion. [Daniel Lindsley]
- Added .values() / .values_list() methods, for fetching less data. [Chris Adams]
- Reduced the number of queries Haystack has to perform in many cases (pagination/facet_counts/spelling_suggestions). [Chris Adams]
- Fixed compatibility with Django pre-1.4 trunk. * The MAX_SHOW_ALL_ALLOWED variable is no longer available, and hence causes an ImportError with Django versions higher 1.3. * The "list_max_show_all" attribute on the ChangeList object is used instead. * This patch maintains compatibility with Django 1.3 and lower by trying to import the MAX_SHOW_ALL_ALLOWED variable first. [Aram Dulyan]

v1.2.6 (2011-12-09)

- I hate Python packaging. [Daniel Lindsley]
- Bumped to v1.2.6! [Daniel Lindsley]
- Made `SearchIndex` classes thread-safe. Thanks to craigds for the report & original patch. [Daniel Lindsley]
- Added a long description for PyPI. [Daniel Lindsley]
- Fixed errant `self.log`. [Daniel Lindsley]

  Thanks to terryh for the report!
- Started 1.2.6. [Daniel Lindsley]

v1.2.5 (2011-09-14)

- Bumped to v1.2.5! [Daniel Lindsley]
- Fixed a bug with index inheritance. [Daniel Lindsley]

  Fields would seem to not obey the MRO while method did. Thanks to ironfroggy for the report!
- Fixed a long-time bug where the Whoosh backend didn't have a `log` attribute. [Daniel Lindsley]
- Fixed a bug with Whoosh's edge n-gram support to be consistent with the implementation in the other engines. [Daniel Lindsley]
- Added tswicegood to AUTHORS. [Daniel Lindsley]
- Fixed the `clear_index` management command to respect the `--site` option. [Travis Swicegood]
- Removed outdated warning about padding numeric fields. Thanks to mchaput for pointing this out! [Daniel Lindsley]
- Added a silent failure option to prevent Haystack from suppressing some failures. [Daniel Lindsley]

  This option defaults to `True` for compatibility & to prevent cases where lost connections can break reindexes/searches.
- Fixed the simple backend to not throw an exception when handed an `SQ`. Thanks to diegobz for the report! [Daniel Lindsley]

- Bumped version post-release. [Daniel Lindsley]
- Whoosh now supports More Like This! Requires Whoosh 1.8.4. [Daniel Lindsley]

v1.2.4 (2011-05-28)

- Bumped to v1.2.4! [Daniel Lindsley]
- Fixed a bug where the old `get_queryset` wouldn't be used during `update_index`. Thanks to Craig de Stigter & others for the report. [Daniel Lindsley]
- Bumped to v1.2.3! [Daniel Lindsley]
- Require Whoosh 1.8.3+. It's for your own good. [Daniel Lindsley]

v1.2.2 (2011-05-19)

- Bumped to v1.2.2! [Daniel Lindsley]
- Added multiprocessing support to `update_index`! Thanks to CMGdigital for funding development of this feature. [Daniel Lindsley]
- Fixed a bug where `set` couldn't be used with `__in`. Thanks to Kronuz for the report! [Daniel Lindsley]
- Added a `DecimalField`. [Daniel Lindsley]

v1.2.1 (2011-05-14)

- Bumped to v1.2.1. [Daniel Lindsley]
- Fixed a typo in the autocomplete docs. Thanks to anderso for the catch! [Daniel Lindsley]
- Fixed a backward-incompatible query syntax change Whoosh introduced between 1.6.1 & 1.6.2 that causes only one model to appear as though it is indexed. [Daniel Lindsley]

v1.2.0 (2011-05-03)

- V1.2.0! [Daniel Lindsley]
- Added `request` to the `FacetedSearchView` context. Thanks to dannercustommade for the report! [Daniel Lindsley]
- Fixed the docs on enabling spelling suggestion support in Solr. [Daniel Lindsley]
- Fixed a bug so that `ValuesListQuerySet` now works with the `__in` filter. Thanks to jcdyer for the report! [Daniel Lindsley]
- Added the new `SearchIndex.read_queryset` bits. [Sam Cooke]
- Changed `update_index` so that it warns you if your `SearchIndex.get_queryset` returns an unusable object. [Daniel Lindsley]
- Removed Python 2.3 compat code & bumped requirements for the impending release. [Daniel Lindsley]
- Added treyhunner to AUTHORS. [Daniel Lindsley]
- Improved the way selected_facets are handled. [Chris Adams]
  - `selected_facets` may be provided multiple times.

  - Facet values are quoted to avoid backend confusion (i.e. *author:Joe Blow* is seen by Solr as *author:Joe AND Blow* rather than the expected *author:"Joe Blow"*)

- Add test for Whoosh field boost. [Trey Hunner]
- Enable field boosting with Whoosh backend. [Trey Hunner]
- Fixed the Solr & Whoosh backends to use the correct `site` when processing results. Thanks to Madan Thangavelu for the original patch! [Daniel Lindsley]
- Added lukeman to AUTHORS. [Daniel Lindsley]
- Updating Solr download and installation instructions to reference version 1.4.1 as 1.3.x is no longer available. Fixes #341. [lukeman]
- Revert "Shifted `handle_registrations` into `models.py`." [Daniel Lindsley]

  This seems to be breaking for people, despite working here & passing tests. Back to the drawing board...

  This reverts commit 106758f88a9bc5ab7e505be62d385d876fbc52fe.
- Shifted `handle_registrations` into `models.py`. [Daniel Lindsley]

  For historical reasons, it was (wrongly) kept & run in `__init__.py`. This should help fix many people's issues with it running too soon.
- Pulled out `EmptyResults` for testing elsewhere. [Daniel Lindsley]
- Fixed a bug where boolean filtering wouldn't work properly on Whoosh. Thanks to alexrobbins for pointing it out! [Daniel Lindsley]
- Added link to 1.1 version of the docs. [Daniel Lindsley]
- Whoosh 1.8.1 compatibility. [Daniel Lindsley]
- Added TodasLasRecetas to "Who Uses". Thanks Javier! [Daniel Lindsley]
- Added a new method to `SearchQuerySet` to allow you to specify a custom `result_class` to use in place of `SearchResult`. Thanks to aaronvanderlip for getting me thinking about this! [Daniel Lindsley]
- Added better autocomplete support to Haystack. [Daniel Lindsley]
- Changed `SearchForm` to be more permissive of missing form data, especially when the form is unbound. Thanks to cleifer for pointing this out! [Daniel Lindsley]
- Ensured that the primary key of the result is a string. Thanks to gremmie for pointing this out! [Daniel Lindsley]
- Fixed a typo in the tutorial. Thanks to JavierLopezMunoz for pointing this out! [Daniel Lindsley]
- Added appropriate warnings about `HAYSTACK_<ENGINE>_PATH` settings in the docs. [Daniel Lindsley]
- Added some checks for badly-behaved backends. [Daniel Lindsley]
- Ensure `use_template` can't be used with `MultiValueField`. [Daniel Lindsley]
- Added n-gram fields for auto-complete style searching. [Daniel Lindsley]
- Added `django-celery-haystack` to the subapp docs. [Daniel Lindsley]
- Fixed the the faceting docs to correctly link to narrowed facets. Thanks to daveumr for pointing that out! [Daniel Lindsley]
- Updated docs to reflect the `form_kwargs` that can be used for customization. [Daniel Lindsley]
- Whoosh backend now explicitly closes searchers in an attempt to use fewer file handles. [Daniel Lindsley]
- Changed fields so that `boost` is now the parameter of choice over `weight` (though `weight` has been retained for backward compatibility). Thanks to many people for the report! [Daniel Lindsley]

- Bumped revision. [Daniel Lindsley]

v1.1 (2010-11-23)

- Bumped version to v1.1! [Daniel Lindsley]
- The `build_solr_schema` command can now write directly to a file. Also includes tests for the new overrides. [Daniel Lindsley]
- Haystack's reserved field names are now configurable. [Daniel Lindsley]
- BACKWARD-INCOMPATIBLE: `auto_query` has changed so that only double quotes cause exact match searches. Thanks to craigds for the report! [Daniel Lindsley]
- Added docs on handling content-type specific output in results. [Daniel Lindsley]
- Added tests for `content_type`. [Daniel Lindsley]
- Added docs on boosting. [Daniel Lindsley]
- Updated the `searchfield_api` docs. [Daniel Lindsley]
- `template_name` can be a list of templates passed to `loader.select_template`. Thanks to zifot for the suggestion. [Daniel Lindsley]
- Moved handle_facet_parameters call into FacetField's __init__. [Travis Cline]
- Updated the pysolr dependency docs & added a debugging note about boost support. [Daniel Lindsley]
- Starting the beta. [Daniel Lindsley]
- Fixed a bug with `FacetedSearchForm` where `cleaned_data` may not exist. Thanks to imageinary for the report! [Daniel Lindsley]
- Added the ability to build epub versions of the docs. [Alfredo]
- Clarified that the current supported version of Whoosh is the 1.1.1+ series. Thanks to glesica for the report & original patch! [Daniel Lindsley]
- The SearchAdmin now correctly uses SEARCH_VAR instead of assuming things. [Rob Hudson]
- Added the ability to "weight" individual fields to adjust their relevance. [David Sauve]
- Fixed facet fieldname lookups to use the proper fieldname. [Daniel Lindsley]
- Removed unneeded imports from the Solr backend. [Daniel Lindsley]
- Further revamping of faceting. Each field type now has a faceted variant that's created either with `faceted=True` or manual initialization. [Daniel Lindsley]

  This should also make user-created field types possible, as many of the gross `isinstance` checks were removed.
- Fixes SearchQuerySet not pickleable. Patch by oyiptong, tests by toastdriven. [oyiptong]
- Added the ability to remove objects from the index that are no longer in the database to the `update_index` management command. [Daniel Lindsley]
- Added a `range` filter type. Thanks to davisp & lukesneeringer for the suggestion! [Daniel Lindsley]

  Note that integer ranges are broken on the current Whoosh (1.1.1). However, date & character ranges seem to work fine.
- Consistency. [Daniel Lindsley]

- Ensured that multiple calls to `count` don't result in multiple queries. Thanks to Nagyman and others for the report! [Daniel Lindsley]
- Ensure that when fetching the length of a result set that the whole index isn't consumed (especially on Whoosh & Xapian). [Daniel Lindsley]
- Really fixed dict ordering bugs in SearchSite. [Travis Cline]
- Changed how you query for facets and how how they are presented in the facet counts. Allows customization of facet field names in indexes. [Travis Cline]

  Lightly backward-incompatible (git only).
- Made it easier to override `SearchView/SearchForm` behavior when no query is present. [Daniel Lindsley]

  No longer do you need to override both `SearchForm` & `SearchView` if you want to return all results. Use the built-in `SearchView`, provide your own custom `SearchForm` subclass & override the `no_query_found` method per the docstring.
- Don't assume that any pk castable to an integer should be an integer. [Carl Meyer]
- Fetching a list of all fields now produces correct results regardless of dict-ordering. Thanks to carljm & veselosky for the report! [Daniel Lindsley]
- Added notes about what is needed to make schema-building independent of dict-ordering. [Daniel Lindsley]
- Sorted model order matters. [Daniel Lindsley]
- Prevent Whoosh from erroring if the `end_offset` is less than or equal to 0. Thanks to zifot for the report! [Daniel Lindsley]
- Removed insecure use of `eval` from the Whoosh backend. Thanks to SmileyChris for pointing this out. [Daniel Lindsley]
- Disallow `indexed=False` on `FacetFields`. Thanks to jefftriplett for the report! [Daniel Lindsley]
- Added `FacetField` & changed the way facets are processed. [Daniel Lindsley]

  Facet data is no longer quietly duplicated just before it goes into the index. Instead, full fields are created (with all the standard data & methods) to contain the faceted information.

  This change is backward-compatible, but allows for better extension, not requiring data duplication into an unfaceted field and a little less magic.
- EmptyQuerySet.facet_counts() won't hit the backend. [Chris Adams]

  This avoids an unnecessary extra backend query displaying the default faceted search form.
- TextMate fail. [Daniel Lindsley]
- Changed `__name__` to an attribute on `SearchView` to work with decorators. Thanks to trybik for the report! [Daniel Lindsley]
- Changed some wording on the tutorial to indicate where the data template should go. Thanks for the suggestion Davepar! [Daniel Lindsley]
- Merge branch 'whoosh-1.1' [Daniel Lindsley]
- Final cleanup before merging Whoosh 1.1 branch! [Daniel Lindsley]
- Final Whoosh 1.1.1 fixes. Waiting for an official release of Whoosh & hand testing, then this ought to be merge-able. [Daniel Lindsley]
- Upgraded the Whoosh backend to 1.1. Still one remaining test failure and two errors. Waiting on mchaput's thoughts/patches. [Daniel Lindsley]
- Mistakenly committed this change. This bug is not fixed. [Daniel Lindsley]

- Better handling of attempts at loading backends when the various supporting libraries aren't installed. Thanks to traviscline for the report. [Daniel Lindsley]
- Fixed random test failures from not running the Solr tests in awhile. [Daniel Lindsley]
- Changed mlt test to use a set comparison to eliminate failures due to ordering differences. [Travis Cline]
- Sped up Solr backend tests by moving away from RealTimeSearchIndex since it was adding objects to Solr when loading fixtures. [Travis Cline]
- Automatically add `suggestion` to the context if `HAYSTACK_INCLUDE_SPELLING` is set. Thanks to notanumber for the suggestion! [Daniel Lindsley]
- Added apollo13 to AUTHORS for the `SearchForm.__init__` cleanup. [Daniel Lindsley]
- Use kwargs.pop instead of try/except. [Florian Apolloner]
- Added Rob to AUTHORS for the admin cleanup. [Daniel Lindsley]
- Fixed selection_note text by adding missing zero. [Rob Hudson]
- Fixed full_result_count in admin search results. [Rob Hudson]
- Fixed admin actions in admin search results. [Rob Hudson]
- Added DevCheatSheet to "Who Uses". [Daniel Lindsley]
- Added Christchurch Art Gallery to "Who Uses". [Daniel Lindsley]
- Forgot to include ghostrocket as submitting a patch on the previous commit. [Daniel Lindsley]
- Fixed a serious bug in the `simple` backend that would flip the object instance and class. [Daniel Lindsley]
- Updated Whoosh to 0.3.18. [Daniel Lindsley]
- Updated NASA's use of Haystack in "Who Uses". [Daniel Lindsley]
- Changed how `ModelSearchIndex` introspects to accurately use `IntegerField` instead of `FloatField` as it was using. [Daniel Lindsley]
- Added CongresoVisible to Who Uses. [Daniel Lindsley]
- Added a test to verify a previous change to the `simple` backend. [Daniel Lindsley]
- Fixed the new admin bits to not explode on Django 1.1. [Daniel Lindsley]
- Added `SearchModelAdmin`, which enables Haystack-based search within the admin. [Daniel Lindsley]
- Fixed a bug when not specifying a `limit` when using the `more_like_this` template tag. Thanks to symroe for the original patch. [Daniel Lindsley]
- Fixed the error messages that occur when looking up attributes on a model. Thanks to acdha for the patch. [Daniel Lindsley]
- Added pagination to the example search template in the docs so it's clear that it is supported. [Daniel Lindsley]
- Fixed copy-paste foul in `Installing Search Engines` docs. [Daniel Lindsley]
- Fixed the `simple` backend to return `SearchResult` instances, not just bare model instances. Thanks to Agos for the report. [Daniel Lindsley]
- Fixed the `clear_index` management command to respect `--verbosity`. Thanks to kylemacfarlane for the report. [Daniel Lindsley]
- Altered the `simple` backend to only search textual fields. This makes the backend work consistently across all databases and is likely the desired behavior anyhow. Thanks to kylemacfarlane for the report. [Daniel Lindsley]

- Fixed a bug in the `Highlighter` which would double-highlight HTML tags. Thanks to EmilStenstrom for the original patch. [Daniel Lindsley]
- Updated management command docs to mention all options that are accepted. [Daniel Lindsley]
- Altered the Whoosh backend to correctly clear the index when using the `RAMStorage` backend. Thanks to kylemacfarlane for the initial patch. [Daniel Lindsley]
- Changed `SearchView` to allow more control over how many results are shown per page. Thanks to simonw for the suggestion. [Daniel Lindsley]
- Ignore `.pyo` files when listing out the backend options. Thanks to kylemacfarlane for the report. [Daniel Lindsley]
- Added CustomMade to Who Uses. [Daniel Lindsley]
- Moved a backend import to allow changing the backend Haystack uses on the fly. [Daniel Lindsley]

  Useful for testing.
- Added more debugging information to the docs. [Daniel Lindsley]
- Added DeliverGood.org to the "Who Uses" docs. [Daniel Lindsley]
- Added an settings override on `HAYSTACK_LIMIT_TO_REGISTERED_MODELS` as a possible performance optimization. [Daniel Lindsley]
- Added the ability to pickle `SearchResult` objects. Thanks to dedsm for the original patch. [Daniel Lindsley]
- Added docs and fixed tests on the backend loading portions. Thanks to kylemacfarlane for the report. [Daniel Lindsley]
- Fixed bug with `build_solr_schema` where `stored=False` would be ignored. Thanks to johnthedebs for the report. [Daniel Lindsley]
- Added debugging notes for Solr. Thanks to smccully for reporting this. [Daniel Lindsley]
- Fixed several errors in the `simple` backend. Thanks to notanumber for the original patch. [Daniel Lindsley]
- Documentation fixes for Xapian. Thanks to notanumber for the edits! [Daniel Lindsley]
- Fixed a typo in the tutorial. Thanks to cmbeelby for pointing this out. [Daniel Lindsley]
- Fixed an error in the tutorial. Thanks to bencc for pointing this out. [Daniel Lindsley]
- Added a warning to the docs that `SearchQuerySet.raw_search` does not chain. Thanks to jacobstr for the report. [Daniel Lindsley]
- Fixed an error in the documentation on providing fields for faceting. Thanks to ghostmob for the report. [Daniel Lindsley]
- Fixed a bug where a field that's both nullable & faceted would error if no data was provided. Thanks to LarryEitel for the report. [Daniel Lindsley]
- Fixed a regression where the built-in Haystack fields would no longer facet correctly. Thanks to traviscline for the report. [Daniel Lindsley]
- Fixed last code snippet on the `SearchIndex.prepare_FOO` docs. Thanks to sk1p for pointing that out. [Daniel Lindsley]
- Fixed a bug where the schema could be built improperly if similar fieldnames had different options. [Daniel Lindsley]
- Added to existing tests to ensure that multiple faceted fields are included in the index. [Daniel Lindsley]
- Finally added a README. [Daniel Lindsley]

- Added a note about versions of the docs. [Daniel Lindsley]
- Go back to the default Sphinx theme. The custom Haystack theme is too much work and too little benefit. [Daniel Lindsley]
- Added a note in the tutorial about building the schema when using Solr. Thanks to trey0 for the report! [Daniel Lindsley]
- Fixed a bug where using `SearchQuerySet.models()` on an unregistered model would be silently ignored. [Daniel Lindsley]

  It is still silently ignored, but now emits a warning informing the user of why they may receive more results back than they expect.
- Added notes about the `simple` backend in the docs. Thanks to notanumber for catching the omission. [Daniel Lindsley]
- Removed erroneous old docs about Lucene support, which never landed. [Daniel Lindsley]
- Merge branch 'master' of github.com:toastdriven/django-haystack. [Daniel Lindsley]
- Fixed typo in the tutorial. Thanks fxdgear for pointing that out! [Daniel Lindsley]
- Fixed a bug related to Unicode data in conjunction with the `dummy` backend. Thanks to kylemacfarlane for the report! [Daniel Lindsley]
- Added Forkinit to Who Uses. [Daniel Lindsley]
- Added Rampframe to Who Uses. [Daniel Lindsley]
- Added other apps documentation for Haystack-related apps. [Daniel Lindsley]
- Unified the way `DEFAULT_OPERATOR` is setup. [Daniel Lindsley]
- You can now override `ITERATOR_LOAD_PER_QUERY` with a setting if you're consuming big chunks of a `SearchQuerySet`. Thanks to kylemacfarlane for the report. [Daniel Lindsley]
- Moved the preparation of faceting data to a `SearchIndex.full_prepare()` method for easier overriding. Thanks to xav for the suggestion! [Daniel Lindsley]
- The `more_like_this` tag now silently fails if things go south. Thanks to piquadrat for the patch! [Daniel Lindsley]
- Added a fleshed out `simple_backend` for basic usage + testing. [David Sauve]
- `SearchView.build_form()` now accepts a dict to pass along to the form. Thanks to traviscline for the patch! [Daniel Lindsley]
- Fixed the `setup.py` to include `haystack.utils` and added to the `MANIFEST.in`. Thanks to jezdez for the patch! [Daniel Lindsley]
- Fixed date faceting in Solr. [Daniel Lindsley]

  No more OOMs and very fast over large data sets.
- Added the `search_view_factory` function for thread-safe use of `SearchView`. [Daniel Lindsley]
- Added more to the docs about the `SearchQuerySet.narrow()` method to describe when/why to use it. [Daniel Lindsley]
- Fixed Whoosh tests. [Daniel Lindsley]

  Somewhere, a reference to the old index was hanging around causing incorrect failures.
- The Whoosh backed now uses the `AsyncWriter`, which ought to provide better performance. Requires Whoosh 0.3.15 or greater. [Daniel Lindsley]

- Added a way to pull the correct fieldname, regardless if it's been overridden or not. [Daniel Lindsley]
- Added docs about adding new fields. [Daniel Lindsley]
- Removed a painful `isinstance` check which should make non-standard usages easier. [Daniel Lindsley]
- Updated docs regarding reserved field names in Haystack. [Daniel Lindsley]
- Pushed some of the new faceting bits down in the implementation. [Daniel Lindsley]
- Removed unnecessary fields from the Solr schema template. [Daniel Lindsley]
- Revamped how faceting is done within Haystack to make it easier to work with. [Daniel Lindsley]
- Add more sites to Who Uses. [Daniel Lindsley]
- Fixed a bug in `ModelSearchIndex` where the `index_fieldname` would not get set. Also added a way to override it in a general fashion. Thanks to traviscline for the patch! [Daniel Lindsley]
- Backend API standardization. Thanks to batiste for the report! [Daniel Lindsley]
- Removed a method that was supposed to have been removed before 1.0. Oops. [Daniel Lindsley]
- Added the ability to override field names within the index. Thanks to traviscline for the suggestion and original patch! [Daniel Lindsley]
- Corrected the AUTHORS because slai actually provided the patch. Sorry about that. [Daniel Lindsley]
- Refined the internals of `ModelSearchIndex` to be a little more flexible. Thanks to traviscline for the patch! [Daniel Lindsley]
- The Whoosh backend now supports `RamStorage` for use with testing or other non-permanent indexes. [Daniel Lindsley]
- Fixed a bug in the `Highlighter` involving repetition and regular expressions. Thanks to alanzoppa for the original patch! [Daniel Lindsley]
- Fixed a bug in the Whoosh backend when a `MultiValueField` is empty. Thanks to alanwj for the original patch! [Daniel Lindsley]
- All dynamic imports now use `importlib`. Thanks to bfirsh for the original patch mentioning this. [Daniel Lindsley]

  A backported version of `importlib` is included for compatibility with Django 1.0.
- Altered `EmptySearchQuerySet` so it's usable from templates. Thanks to bfirsh for the patch! [Daniel Lindsley]
- Added tests to ensure a Whoosh regression is no longer present. [Daniel Lindsley]
- Fixed a bug in Whoosh where using just `.models()` would create an invalid query. Thanks to ricobl for the original patch. [Daniel Lindsley]
- Forms with initial data now display it when used with SearchView. Thanks to osirius for the original patch. [Daniel Lindsley]
- App order is now consistent with INSTALLED_APPS when running `update_index`. [Daniel Lindsley]
- Updated docs to reflect the recommended way to do imports in when defining `SearchIndex` classes. [Daniel Lindsley]

  This is not my preferred style but reduces the import errors some people experience.
- Fixed omission of Xapian in the settings docs. Thanks to flebel for pointing this out. [Daniel Lindsley]
- Little bits of cleanup related to testing. [Daniel Lindsley]
- Fixed an error in the docs related to pre-rendering data. [Daniel Lindsley]

- Added Pegasus News to Who Uses. [Daniel Lindsley]
- Corrected an import in forms for consistency. Thanks to bkonkle for pointing this out. [Daniel Lindsley]
- Fixed bug where passing a customized `site` would not make it down through the whole stack. Thanks to Peter Bengtsson for the report and original patch. [Daniel Lindsley]
- Bumped copyright years. [Daniel Lindsley]
- Changed Whoosh backend so most imports will raise the correct exception. Thanks to shabda for the suggestion. [Daniel Lindsley]
- Refactored Solr's tests to minimize reindexes. Runs ~50% faster. [Daniel Lindsley]
- Fixed a couple potential circular imports. [Daniel Lindsley]
- The same field can now have multiple query facets. Thanks to bfirsh for the original patch. [Daniel Lindsley]
- Added schema for testing Solr. [Daniel Lindsley]
- Fixed a string interpolation bug when adding an invalid data facet. Thanks to simonw for the original patch. [Daniel Lindsley]
- Fixed the default highlighter to give slightly better results, especially with short strings. Thanks to Robert-Gawron for the original patch. [Daniel Lindsley]
- Changed the `rebuild_index` command so it can take all options that can be passed to either `clear_index` or `update_index`. Thanks to brosner for suggesting this. [Daniel Lindsley]
- Added `--noinput` flag to `clear_index`. Thanks to aljosa for the suggestion. [Daniel Lindsley]
- Updated the example in the template to be a little more real-world and user friendly. Thanks to j0hnsmith for pointing this out. [Daniel Lindsley]
- Fixed a bug with the Whoosh backend where scores weren't getting populated correctly. Thanks to horribtastic for the report. [Daniel Lindsley]
- Changed `EmptySearchQuerySet` so it returns an empty list when slicing instead of mistakenly running queries. Thanks to askfor for reporting this bug. [Daniel Lindsley]
- Switched `SearchView` & `FacetedSearchView` to use `EmptySearchQuerySet` (instead of a regular list) when there are no results. Thanks to acdha for the original patch. [Daniel Lindsley]
- Added RedditGifts to "Who Uses". [Daniel Lindsley]
- Added Winding Road to "Who Uses". [Daniel Lindsley]
- Added ryszard's full name to AUTHORS. [Daniel Lindsley]
- Added initialization bits to part of the Solr test suite. Thanks to notanumber for pointing this out. [Daniel Lindsley]
- Started the 1.1-alpha work. Apologies for not doing this sooner. [Daniel Lindsley]
- Added an advanced setting for disabling Haystack's initialization in the event of a conflict with other apps. [Daniel Lindsley]
- Altered `SearchForm` to use `.is_valid()` instead of `.clean()`, which is a more idiomatic/correct usage. Thanks to askfor for the suggestion. [Daniel Lindsley]
- Added MANIFEST to ignore list. [Daniel Lindsley]
- Fixed Django 1.0 compatibility when using the Solr backend. [Daniel Lindsley]
- Marked Haystack as 1.0 final. [Daniel Lindsley]

- Incorrect test result from changing the documented way the `highlight` template tag gets called. [Daniel Lindsley]
- Updated the example in faceting documentation to provide better results and explanation on the reasoning. [Daniel Lindsley]
- Added further documentation about `SearchIndex`/`RealTimeSearchIndex`. [Daniel Lindsley]
- Added docs about *SearchQuerySet.highlight*. [toastdriven]
- Added further docs on *RealTimeSearchIndex*. [toastdriven]
- Added documentation on the `RealTimeSearchIndex` class. [toastdriven]
- Fixed the documentation for the arguments on the *highlight* tag. Thanks to lucalenardi for pointing this out. [Daniel Lindsley]
- Fixed tutorial to mention where the *NoteSearchIndex* should be placed. Thanks to bkeating for pointing this out. [Daniel Lindsley]
- Marked Haystack as 1.0.0 release candidate 1. [Daniel Lindsley]
- Haystack now requires Whoosh 0.3.5. [Daniel Lindsley]
- Last minute documentation cleanup. [Daniel Lindsley]
- Added documentation about the management commands that come with Haystack. [Daniel Lindsley]
- Added docs on the template tags included with Haystack. [Daniel Lindsley]
- Added docs on highlighting. [Daniel Lindsley]
- Removed some unneeded legacy code that was causing conflicts when Haystack was used with apps that load all models (such as *django- cms2*, *localemiddleware* or *django-transmeta*). [Daniel Lindsley]
- Removed old code from the *update_index* command. [Daniel Lindsley]
- Altered spelling suggestion test to something a little more consistent. [Daniel Lindsley]
- Added tests for slicing the end of a *RelatedSearchQuerySet*. [Daniel Lindsley]
- Fixed case where *SearchQuerySet.more_like_this* would fail when using deferred Models. Thanks to Alex Gaynor for the original patch. [Daniel Lindsley]
- Added default logging bits to prevent "No handlers found" message. [Daniel Lindsley]
- BACKWARD-INCOMPATIBLE: Renamed *reindex* management command to *update_index*, renamed *clear_search_index* management command to *clear_index* and added a *rebuild_index* command to both clear & reindex. [Daniel Lindsley]
- BACKWARD-INCOMPATIBLE: *SearchIndex* no longer hooks up *post_save/post_delete* signals for the model it's registered with. [Daniel Lindsley]

  If you use *SearchIndex*, you will have to manually cron up a *reindex* (soon to become *update_index*) management command to periodically refresh the data in your index.

  If you were relying on the old behavior, please use *RealTimeSearchIndex* instead, which does hook up those signals.
- Ensured that, if a *MultiValueField* is marked as *indexed=False* in Whoosh, it ought not to post-process the field. [Daniel Lindsley]
- Ensured data going into the indexes round-trips properly. Fixed *DateField*/*DateTimeField* handling for all backends and *MultiValueField* handling in Whoosh. [Daniel Lindsley]
- Added a customizable *highlight* template tag plus an underlying *Highlighter* implementation. [Daniel Lindsley]

- Added more documentation about using custom *SearchIndex.prepare_FOO* methods. [Daniel Lindsley]
- With Whoosh 0.3.5+, the number of open files is greatly reduced. [Daniel Lindsley]
- Corrected example in docs about *RelatedSearchQuerySet*. Thanks to askfor for pointing this out. [Daniel Lindsley]
- Altered *SearchResult* objects to fail gracefully when the model/object can't be found. Thanks to akrito for the report. [Daniel Lindsley]
- Fixed a bug where *auto_query* would fail to escape strings that pulled out for exact matching. Thanks to jefftriplett for the report. [Daniel Lindsley]
- Added Brick Design to Who Uses. [Daniel Lindsley]
- Updated backend support docs slightly. [Daniel Lindsley]
- Added the ability to combine *SearchQuerySet's via '&* or |. Thanks to reesefrancis for the suggestion. [Daniel Lindsley]
- Revised the most of the tutorial. [Daniel Lindsley]
- Better documented how user-provided data should be sanitized. [Daniel Lindsley]
- Fleshed out the *SearchField* documentation. [Daniel Lindsley]
- Fixed formatting on `SearchField` documentation. [Daniel Lindsley]
- Added basic `SearchField` documentation. [Daniel Lindsley]

  More information about the kwargs and usage will be eventually needed.
- Bumped the *ulimit* so Whoosh tests pass consistently on Mac OS X. [Daniel Lindsley]
- Fixed the *default* kwarg in *SearchField* (and subclasses) to work properly from a user's perspective. [Daniel Lindsley]
- BACKWARD-INCOMPATIBLE: Fixed `raw_search` to cooperate when paginating/slicing as well as many other conditions. [Daniel Lindsley]

  This no longer immediately runs the query, nor pokes at any internals. It also now takes into account other details, such as sorting & faceting.
- Fixed a bug in the Whoosh backend where slicing before doing a hit count could cause strange results when paginating. Thanks to kylemacfarlane for the original patch. [Daniel Lindsley]
- The Whoosh tests now deal with the same data set as the Solr tests and cover various aspects better. [Daniel Lindsley]
- Started to pull out the real-time, signal-based updates out of the main *SearchIndex* class. Backward compatible for now. [Daniel Lindsley]
- Fixed docs to include *utils* documentation. [Daniel Lindsley]
- Updated instructions for installing *pysolr*. Thanks to sboisen for pointing this out. [Daniel Lindsley]
- Added acdha to AUTHORS for previous commit. [Daniel Lindsley]
- Added exception handling to the Solr Backend to silently fail/log when Solr is unavailable. Thanks to acdha for the original patch. [Daniel Lindsley]
- The *more_like_this* tag is now tested within the suite. Also has lots of cleanup for the other Solr tests. [Daniel Lindsley]
- On both the Solr & Whoosh backends, don't do an update if there's nothing being updated. [Daniel Lindsley]

- Moved Haystack's internal fields out of the backends and into *SearchIndex.prepare*. [Daniel Lindsley]

  This is both somewhat more DRY as well as a step toward Haystack being useful to non-Django projects.

- Fixed a bug in the *build_schema* where fields that aren't supposed to be indexed are still getting post-procesed by Solr. Thanks to Jonathan Slenders for the report. [Daniel Lindsley]
- Added HUGE to Who Uses. [Daniel Lindsley]
- Fixed bug in Whoosh where it would always generate spelling suggestions off the full query even when given a different query string to check against. [Daniel Lindsley]
- Simplified the SQ object and removed a limitation on kwargs/field names that could be passed in. Thanks to traviscline for the patch. [Daniel Lindsley]
- Documentation on *should_update* fixed to match the new signature. Thanks to kylemacfarlane for pointing this out. [Daniel Lindsley]
- Fixed missing words in Best Practices documentation. Thanks to frankwiles for the original patch. [Daniel Lindsley]
- The *update_object* method now passes along kwargs as needed to the *should_update* method. Thanks to askfor for the suggestion. [Daniel Lindsley]
- Updated docs about the removal of the Whoosh fork. [Daniel Lindsley]
- Removed extraneous *BadSearchIndex3* from test suite. Thanks notanumber! [Daniel Lindsley]
- We actually want *repr*, not *str*. [Daniel Lindsley]
- Pushed the *model_attr* check lower down into the `SearchField`s and make it occur later, so that exceptions come at a point where Django can better deal with them. [Daniel Lindsley]
- Fixed attempting to access an invalid *model_attr*. Thanks to notanumber for the original patch. [Daniel Lindsley]
- Added SQ objects (replacing the QueryFilter object) as the means to generate queries/query fragments. Thanks to traviscline for all the hard work. [Daniel Lindsley]

  The SQ object is similar to Django's Q object and allows for arbitrarily complex queries. Only backward incompatible if you were relying on the SearchQuery/QueryFilter APIs.

- Reformatted debugging docs a bit. [Daniel Lindsley]
- Added debugging information about the Whoosh lock error. [Daniel Lindsley]
- Brought the TODO up to date. [Daniel Lindsley]
- Added a warning to the documentation about how *__startswith* may not always provide the expected results. Thanks to codysoyland for pointing this out. [Daniel Lindsley]
- Added debugging documentation, with more examples coming in the future. [Daniel Lindsley]
- Added a new *basic_search* view as a both a working example of how to write traditional views and as a thread-safe view, which the class- based ones may/may not be. [Daniel Lindsley]
- Fixed sample template in the documentation. Thanks to lemonad for pointing this out. [Daniel Lindsley]
- Updated documentation to include a couple more Sphinx directives. Index is now more useful. [Daniel Lindsley]
- Made links more obvious in documentation. [Daniel Lindsley]
- Added an *example_project* demonstrating how a sample project might be setup. [Daniel Lindsley]
- Fixed *load_backend* to use the argument passed instead of always the *settings.HAYSTACK_SEARCH_ENGINE*. Thanks to newgene for the report. [Daniel Lindsley]

- Regression where sometimes *narrow_queries* got juggled into a list when it should be a set everywhere. Thanks tcline & ericholscher for the report. [Daniel Lindsley]
- Updated the Whoosh backend's version requirement to reflect the fully working version of Whoosh. [Daniel Lindsley]
- With the latest SVN version of Whoosh (r344), *SearchQuerySet()* now works properly in Whoosh. [Daniel Lindsley]
- Added a *FacetedModelSearchForm*. Thanks to mcroydon for the original patch. [Daniel Lindsley]
- Added translation capabilities to the *SearchForm* variants. Thanks to hejsan for pointing this out. [Daniel Lindsley]
- Added AllForLocal to Who Uses. [Daniel Lindsley]
- The underlying caching has been fixed so it no longer has to fill the entire cache before it to ensure consistency. [Daniel Lindsley]

  This results in significantly faster slicing and reduced memory usage. The test suite is more complete and ensures this functionality better.

  This also removes *load_all_queryset* from the main *SearchQuerySet* implementation. If you were relying on this behavior, you should use *RelatedSearchQuerySet* instead.
- Log search queries with *DEBUG = True* for debugging purposes, similar to what Django does. [Daniel Lindsley]
- Updated LJ's Who Uses information. [Daniel Lindsley]
- Added Sunlight Labs & NASA to the Who Uses list. [Daniel Lindsley]
- Added Eldarion to the Who Uses list. [Daniel Lindsley]
- When more of the cache is populated, provide a more accurate *len()* of the *SearchQuerySet*. This ought to only affect advanced usages, like excluding previously-registered models or *load_all_queryset*. [Daniel Lindsley]
- Fixed a bug where *SearchQuerySet's longer than 'REPR_OUTPUT_SIZE* wouldn't include a note about truncation when *__repr__* is called. [Daniel Lindsley]
- Added the ability to choose which site is used when reindexing. Thanks to SmileyChris for pointing this out and the original patch. [Daniel Lindsley]
- Fixed the lack of a *__unicode__* method on *SearchResult* objects. Thanks to mint_xian for pointing this out. [Daniel Lindsley]
- Typo'd the setup.py changes. Thanks to jlilly for catching that. [Daniel Lindsley]
- Converted all query strings to Unicode for Whoosh. Thanks to simonw108 for pointing this out. [Daniel Lindsley]
- Added template tags to *setup.py*. Thanks to Bogdan for pointing this out. [Daniel Lindsley]
- Added two more tests to the Whoosh backend, just to make sure. [Daniel Lindsley]
- Corrected the way Whoosh handles *order_by*. Thanks to Rowan for pointing this out. [Daniel Lindsley]
- For the Whoosh backend, ensure the directory is writable by the current user to try to prevent failed writes. [Daniel Lindsley]
- Added a better label to the main search form field. [Daniel Lindsley]
- Bringing the Whoosh backend up to version 0.3.0b14. This version of Whoosh has better query parsing, faster indexing and, combined with these changes, should cause fewer disruptions when used in a multiprocess/multithreaded environment. [Daniel Lindsley]

- Added optional argument to *spelling_suggestion* that lets you provide a different query than the one built by the SearchQuerySet. [Daniel Lindsley]

  Useful for passing along a raw user-provided query, especially when there is a lot of post-processing done.

- SearchResults now obey the type of data chosen in their corresponding field in the SearchIndex if present. Thanks to evgenius for the original report. [Daniel Lindsley]
- Fixed a bug in the Solr backend where submitting an empty string to search returned an ancient and incorrect datastructure. Thanks kapa77 for the report. [Daniel Lindsley]
- Fixed a bug where the cache would never properly fill due to the number of results returned being lower than the hit count. This could happen when there were results excluded due to being in the index but the model NOT being registered in the *SearchSite*. Thanks akrito and tcline for the report. [Daniel Lindsley]
- Altered the docs to look more like the main site. [Daniel Lindsley]
- Added a (short) list of who uses Haystack. Would love to have more on this list. [Daniel Lindsley]
- Fixed docs on preparing data. Thanks fud. [Daniel Lindsley]
- Added the *ModelSearchIndex* class for easier *SearchIndex* generation. [Daniel Lindsley]
- Added a note about using possibly unsafe data with *filter/exclude*. Thanks to ryszard for pointing this out. [Daniel Lindsley]
- Standardized the API on *date_facet*. Thanks to notanumber for the original patch. [Daniel Lindsley]
- Moved constructing the schema down to the *SearchBackend* level. This allows more flexibility when creating a schema. [Daniel Lindsley]
- Fixed a bug where a hyphen provided to *auto_query* could break the query string. Thanks to ddanier for the report. [Daniel Lindsley]
- BACKWARD INCOMPATIBLE - For consistency, *get_query_set* has been renamed to *get_queryset* on *SearchIndex* classes. [Daniel Lindsley]

  A simple search & replace to remove the underscore should be all that is needed.

- Missed two bits while updating the documentation for the Xapian backend. [Daniel Lindsley]
- Updated documentation to add the Xapian backend information. A big thanks to notatnumber for all his hard work on the Xapian backend. [Daniel Lindsley]
- Added *EmptySearchQuerySet*. Thanks to askfor for the suggestion! [Daniel Lindsley]
- Added "Best Practices" documentation. [Daniel Lindsley]
- Added documentation about the *HAYSTACK_SITECONF* setting. [Daniel Lindsley]
- Fixed erroneous documentation on Xapian not supporting boost. Thanks notanumber! [Daniel Lindsley]
- BACKWARD INCOMPATIBLE - The *haystack.autodiscover()* and other site modifications now get their own configuration file and should no longer be placed in the *ROOT_URLCONF*. Thanks to SmileyChris for the original patch and patrys for further feedback. [Daniel Lindsley]
- Added *verbose_name_plural* to the *SearchResult* object. [Daniel Lindsley]
- Added a warning about ordering by integers with the Whoosh backend. [Daniel Lindsley]
- Added a note about ordering and accented characters. [Daniel Lindsley]
- Updated the *more_like_this* tag to allow for narrowing the models returned by the tag. [Daniel Lindsley]
- Fixed *null=True* for *IntegerField* and *FloatField*. Thanks to ryszard for the report and original patch. [Daniel Lindsley]

- Reverted aabdc9d4b98edc4735ed0c8b22aa09796c0a29ab as it would cause mod_wsgi environments to fail in conjunction with the admin on Django 1.1. [Daniel Lindsley]
- Added the start of a glossary of terminology. [Daniel Lindsley]
- Various documentation fixes. Thanks to sk1p & notanumber. [Daniel Lindsley]
- The *haystack.autodiscover()* and other site modifications may now be placed in ANY URLconf, not just the *ROOT_URLCONF*. Thanks to SmileyChris for the original patch. [Daniel Lindsley]
- Fixed invalid/empty pages in the SearchView. Thanks to joep and SmileyChris for patches. [Daniel Lindsley]
- Added a note and an exception about consistent fieldnames for the document field across all *SearchIndex* classes. Thanks **sk1p_**! [Daniel Lindsley]
- Possible thread-safety fix related to registration handling. [Daniel Lindsley]
- BACKWARD INCOMPATIBLE - The 'boost' method no longer takes kwargs. This makes boost a little more useful by allowing advanced terms. [Daniel Lindsley]

  To migrate code, convert multiple kwargs into separate 'boost' calls, quote what was the key and change the '=' to a ','.
- Updated documentation to match behavioral changes to MLT. [Daniel Lindsley]
- Fixed a serious bug in MLT on Solr. Internals changed a bit and now things work correctly. [Daniel Lindsley]
- Removed erroneous 'zip_safe' from setup.py. Thanks ephelon. [Daniel Lindsley]
- Added *null=True* to fields, allowing you to ignore/skip a field when indexing. Thanks to Kevin for the original patch. [Daniel Lindsley]
- Fixed a standing test failure. The dummy setup can't do *load_all* due to mocking. [Daniel Lindsley]
- Added initial *additional_query* to MLT to allow for narrowing results. [Daniel Lindsley]
- Fixed nasty bug where results would get duplicated due to cached results. [Daniel Lindsley]
- Altered *ITERATOR_LOAD_PER_QUERY* from 20 to 10. [Daniel Lindsley]
- Corrected tutorial when dealing with fields that have *use_template=True*. [Daniel Lindsley]
- Updated documentation to reflect basic Solr setup. [Daniel Lindsley]
- Fix documentation on grabbing Whoosh and on the 'load_all' parameter for SearchForms. [Daniel Lindsley]
- Fixed bug where the '__in' filter wouldn't work with phrases or data types other than one-word string/integer. [Daniel Lindsley]
- Fixed bug so that the 'load_all' option in 'SearchView' now actually does what it says it should. How embarrassing... [Daniel Lindsley]
- Added ability to specify custom QuerySets for loading records via 'load_all'/'load_all_queryset'. [Daniel Lindsley]
- Fixed a bug where results from non-registered models could appear in the results. [Daniel Lindsley]
- BACKWARD INCOMPATIBLE - Changed 'module_name' to 'model_name' throughout Haystack related to SearchResult objects. Only incompatible if you were relying on this attribute. [Daniel Lindsley]
- Added the ability to fetch additional and stored fields from a SearchResult as well as documentation on the SearchResult itself. [Daniel Lindsley]
- Added the ability to look through relations in SearchIndexes via '__'. [Daniel Lindsley]
- Added note about the 'text' fieldname convention. [Daniel Lindsley]
- Added an 'update_object' and 'remove_object' to the SearchSite objects as a shortcut. [Daniel Lindsley]

- Recover gracefully from queries Whoosh judges to be invalid. [Daniel Lindsley]
- Missed test from previous commit. [Daniel Lindsley]
- Added stemming support to Whoosh. [Daniel Lindsley]
- Removed the commented version. [Daniel Lindsley]
- Django 1.0.X compatibility fix for the reindex command. [Daniel Lindsley]
- Reindexes should now consume a lot less RAM. [Daniel Lindsley]

  Evidently, when you run a ton of queries touching virtually everything in your DB, you need to clean out the "logged" queries from the connection. Sad but true.
- Altered *SearchBackend.remove* and *SearchBackend.get_identifier* to accept an object or a string identifier (in the event the object is no longer available). [Daniel Lindsley]

  This is useful in an environment where you no longer have the original object on hand and know what it is you wish to delete.
- Added a simple (read: ghetto) way to run the test suite without having to mess with settings. [Daniel Lindsley]
- Added a setting *HAYSTACK_BATCH_SIZE* to control how many objects are processed at once when running a reindex. [Daniel Lindsley]
- Fixed import that was issuing a warning. [Daniel Lindsley]
- Further tests to make sure *unregister* works appropriately as well, just to be paranoid. [Daniel Lindsley]
- Fixed a bizarre bug where backends may see a different site object than the rest of the application code. THIS REQUIRES SEARCH & REPLACING ALL INSTANCES OF *from haystack.sites import site* TO *from haystack import site*. [Daniel Lindsley]

  No changes needed if you've been using *haystack.autodiscover()*.
- Pushed save/delete signal registration down to the SearchIndex level. [Daniel Lindsley]

  This should make it easier to alter how individual indexes are setup, allowing you to queue updates, prevent deletions, etc. The internal API changed slightly.
- Created a default 'clean' implementation, as the first three (and soon fourth) backends all use identical code. [Daniel Lindsley]
- Updated tests to match new 'model_choices'. [Daniel Lindsley]
- Added timeout support to Solr. [Daniel Lindsley]
- Capitalize the Models in the model_choices. [Daniel Lindsley]
- Removed unnecessary import. [Daniel Lindsley]
- No longer need to watch for DEBUG in the 'haystack_info' command. [Daniel Lindsley]
- Fixed bug in Whoosh backend when spelling suggestions are disabled. [Daniel Lindsley]
- Added a "clear_search_index" management command. [Daniel Lindsley]
- Removed comments as pysolr now supports timeouts and the other comment no longer applies. [Daniel Lindsley]
- Removed Solr-flavored schema bits. [Daniel Lindsley]

  Still need to work out a better way to handle user created fields that don't fit neatly into subclassing one of the core Field types.
- Moved informational messages to a management command to behave better when using dumpdata or wsgi. [Daniel Lindsley]

- Changed some Solr-specific field names. Requires a reindex. [Daniel Lindsley]
- Typo'd docstring. [Daniel Lindsley]
- Removed empty test file from spelling testing. [Daniel Lindsley]
- Documentation for getting spelling support working on Solr. [Daniel Lindsley]
- Initial spelling support added. [Daniel Lindsley]
- Added a 'more_like_this' template tag. [Daniel Lindsley]
- Removed an unnecessary 'run'. This cause MLT (and potentially 'raw_search') to fail by overwriting the results found. [Daniel Lindsley]
- Added Whoosh failure. Needs inspecting. [Daniel Lindsley]
- Finally added views/forms documentation. A touch rough still. [Daniel Lindsley]
- Fixed a bug in FacetedSearchView where a SearchQuerySet method could be called on an empty list instead. [Daniel Lindsley]
- More faceting documentation. [Daniel Lindsley]
- Started faceting documentation. [Daniel Lindsley]
- Updated docs to finally include details about faceting. [Daniel Lindsley]
- Empty or one character searches in Whoosh returned the wrong data structure. Thanks for catching this, silviogutierrez! [Daniel Lindsley]
- Added scoring to Whoosh now that 0.1.20+ support it. [Daniel Lindsley]
- Fixed a bug in the Solr tests due to recent changes in pysolr. [Daniel Lindsley]
- Added documentation on the 'narrow' method. [Daniel Lindsley]
- Added additional keyword arguments on raw_search. [Daniel Lindsley]
- Added 'narrow' support in Whoosh. [Daniel Lindsley]
- Fixed Whoosh backend's handling of pre-1900 dates. Thanks JoeGermuska! [Daniel Lindsley]
- Backed out the Whoosh quoted dates patch. [Daniel Lindsley]

  Something still seems amiss in the Whoosh query parser, as ranges and dates together don't seem to get parsed together properly.
- Added a small requirements section to the docs. [Daniel Lindsley]
- Added notes about enabling the MoreLikeThisHandler within Solr. [Daniel Lindsley]
- Revised how tests are done so each backend now gets its own test app. [Daniel Lindsley]

  All tests pass once again.
- Added 'startswith' filter. [Daniel Lindsley]
- Fixed the __repr__ method on QueryFilters. Thanks JoeGermuska for the original patch! [Daniel Lindsley]
- BACKWARDS INCOMPATIBLE - Both the Solr & Whoosh backends now provide native Python types back in SearchResults. [Daniel Lindsley]

  This also allows Whoosh to use native types better from the 'SearchQuerySet' API itself.

  This unfortunately will also require all Whoosh users to reindex, as the way some data (specifically datetimes/dates but applicable to others) is stored in the index.
- SearchIndexes now support inheritance. Thanks smulloni! [Daniel Lindsley]

- Added FacetedSearchForm to make handling facets easier. [Daniel Lindsley]
- Heavily refactored the SearchView to take advantage of being a class. [Daniel Lindsley]

  It should now be much easier to override bits without having to copy-paste the entire __call__ method, which was more than slightly embarrassing before.
- Fixed Solr backend so that it properly converts native Python types to something Solr can handle. Thanks smulloni for the original patch! [Daniel Lindsley]
- SearchResults now include a verbose name for display purposes. [Daniel Lindsley]
- Fixed reverse order_by's when using Whoosh. Thanks matt_c for the original patch. [Daniel Lindsley]
- Handle Whoosh stopwords behavior when provided a single character query string. [Daniel Lindsley]
- Lightly refactored tests to only run engines with their own settings. [Daniel Lindsley]
- Typo'd the tutorial when setting up your own SearchSite. Thanks mcroydon! [Daniel Lindsley]
- Altered loading statements to only display when DEBUG is True. [Daniel Lindsley]
- Write to STDERR where appropriate. Thanks zerok for suggesting this change. [Daniel Lindsley]
- BACKWARD INCOMPATIBLE - Altered the search query param to 'q' instead of 'query'. Thanks simonw for prompting this change. [Daniel Lindsley]
- Removed the Whoosh patch in favor of better options. Please see the documentation. [Daniel Lindsley]
- Added Whoosh patch for 0.1.15 to temporarily fix reindexes. [Daniel Lindsley]
- Altered the reindex command to handle inherited models. Thanks smulloni! [Daniel Lindsley]
- Removed the no longer needed Whoosh patch. [Daniel Lindsley]

  Whoosh users should upgrade to the latest Whoosh (0.1.15) as it fixes the issues that the patch covers as well as others.
- Documented the 'content' shortcut. [Daniel Lindsley]
- Fixed an incorrect bit of documentation on the default operator setting. Thanks benspaulding! [Daniel Lindsley]
- Added documentation about Haystack's various settings. [Daniel Lindsley]
- Corrected an issue with the Whoosh backend that can occur when no indexes are registered. Now provides a better exception. [Daniel Lindsley]
- Documentation fixes. Thanks benspaulding! [Daniel Lindsley]
- Fixed Whoosh patch, which should help with the "KeyError" exceptions when searching with models. Thanks Matias Costa! [Daniel Lindsley]
- Improvements to the setup.py. Thanks jezdez & ask! [Daniel Lindsley]
- Fixed the .gitignore. Thanks ask! [Daniel Lindsley]
- FacetedSearchView now inherits from SearchView. Thanks cyberdelia! [Daniel Lindsley]

  This will matter much more soon, as SearchView is going to be refactored to be more useful and extensible.
- Documentation fixes. [Daniel Lindsley]
- Altered the whoosh patch. Should apply cleanly now. [Daniel Lindsley]
- Better linking to the search engine installation notes. [Daniel Lindsley]
- Added documentation on setting up the search engines. [Daniel Lindsley]

- Provide an exception when importing a backend dependency fails. Thanks brosner for the initial patch. [Daniel Lindsley]
- Yay stupid typos! [Daniel Lindsley]
- Relicensing under BSD. Thanks matt_c for threatening to use my name in an endorsement of a derived product! [Daniel Lindsley]
- Fixed a bug in ModelSearchForm. Closes #1. Thanks dotsphinx! [Daniel Lindsley]
- Added link to pysolr binding. [Daniel Lindsley]
- Refined documentation on preparing SearchIndex data. [Daniel Lindsley]
- Changed existing references from 'model_name' to 'module_name'. [Daniel Lindsley]

  This was done to be consistent both internally and with Django. Thanks brosner!
- Documentation improvements. Restyled and friendlier intro page. [Daniel Lindsley]
- Added documentation on preparing data. [Daniel Lindsley]
- Additions and re-prioritizing the TODO list. [Daniel Lindsley]
- Added warnings to Whoosh backend in place of silently ignoring unsupported features. [Daniel Lindsley]
- Corrected Xapian's capabilities. Thanks richardb! [Daniel Lindsley]
- BACKWARD INCOMPATIBLE - Altered all settings to be prefixed with **HAYSTACK_**. Thanks Collin! [Daniel Lindsley]
- Test cleanup from previous commits. [Daniel Lindsley]
- Changed the DEFAULT_OPERATOR back to 'AND'. Thanks richardb! [Daniel Lindsley]
- Altered the way registrations get handled. [Daniel Lindsley]
- Various fixes. Thanks brosner! [Daniel Lindsley]
- Added new 'should_update' method to documentation. [Daniel Lindsley]
- Added 'should_update' method to SearchIndexes. [Daniel Lindsley]

  This allows you to control, on a per-index basis, what conditions will cause an individual object to reindex. Useful for models that update frequently with changes that don't require indexing.
- Added FAQ docs. [Daniel Lindsley]
- Alter Whoosh backend to commit regardless. This avoids locking issues that can occur on higher volume sites. [Daniel Lindsley]
- A more efficient implementation of index clearing in Whoosh. [Daniel Lindsley]
- Added details about settings needed in settings.py. [Daniel Lindsley]
- Added setup.py. Thanks cyberdelia for prompting it. [Daniel Lindsley]
- Reindex management command now can reindex a limited range (like last 24 hours). Thanks traviscline. [Daniel Lindsley]
- More things to do. [Daniel Lindsley]
- Documentation formatting fixes. [Daniel Lindsley]
- Added SearchBackend docs. [Daniel Lindsley]
- Corrected reST formatting. [Daniel Lindsley]
- Additional TODO's. [Daniel Lindsley]

- Initial SearchIndex documentation. [Daniel Lindsley]
- Formally introduced the TODO. [Daniel Lindsley]
- Updated backend support list. [Daniel Lindsley]
- Added initial documentation for SearchSites. [Daniel Lindsley]
- Changed whoosh backend to fix limiting sets. Need to revisit someday. [Daniel Lindsley]
- Added patch for Whoosh backend and version notes in documentation. [Daniel Lindsley]
- Initial Whoosh backend complete. [Daniel Lindsley]

  Does not yet support highlighting or scoring.
- Removed some unnecessary dummy code. [Daniel Lindsley]
- Work on trying to get the default site to load reliably in all cases. [Daniel Lindsley]
- Trimmed down the urls for tests now that the dummy backend works correctly. [Daniel Lindsley]
- Dummy now correctly loads the right SearchBackend. [Daniel Lindsley]
- Removed faceting from the default SearchView. [Daniel Lindsley]
- Refactored tests so they are no longer within the haystack app. [Daniel Lindsley]

  Further benefits include less mocking and haystack's tests no longer contributing overall testing of end-user apps. Documentation included.
- Removed old comment. [Daniel Lindsley]
- Fixed a potential race condition. Also, since there's no way to tell when everything is ready to go in Django, adding an explicit call to SearchQuerySet's __init__ to force the site to load if it hasn't already. [Daniel Lindsley]
- More tests on models() support. [Daniel Lindsley]
- Pulled schema building out into the site to leverage across backends. [Daniel Lindsley]
- Altered backend loading for consistency with Django and fixed the long-incorrect-for-non-obvious-and-tedious-reasons version number. Still beta but hopefully that changes soon. [Daniel Lindsley]
- Missed a spot when fixing SearchSites. [Daniel Lindsley]
- BACKWARD INCOMPATIBLE - Created a class name conflict during the last change (double use of `SearchIndex`). Renamed original `SearchIndex` to `SearchSite`, which is slightly more correct anyhow. [Daniel Lindsley]

  This will only affect you if you've custom built sites (i.e. not used `autodiscover()`.
- More documentation. Started docs on SearchQuery. [Daniel Lindsley]
- Further fleshed out SearchQuerySet documentation. [Daniel Lindsley]
- BACKWARD INCOMPATIBLE (2 of 2) - Altered autodiscover to search for 'search_indexes.py' instead of 'indexes.py' to prevent collisions and be more descriptive. [Daniel Lindsley]
- BACKWARD INCOMPATIBLE (1 of 2) - The ModelIndex class has been renamed to be SearchIndex to make room for future improvements. [Daniel Lindsley]
- Fleshed out a portion of the SearchQuerySet documentation. [Daniel Lindsley]
- SearchQuerySet.auto_query now supports internal quoting for exact matches. [Daniel Lindsley]
- Fixed semi-serious issue with SearchQuery objects, causing bits to leak from one query to the next when cloning. [Daniel Lindsley]
- Altered Solr port for testing purposes. [Daniel Lindsley]

- Now that Solr and core feature set are solid, moved haystack into beta status. [Daniel Lindsley]
- Added simple capabilities for retrieving facets back. [Daniel Lindsley]
- Bugfix to make sure model choices don't get loaded until after the IndexSite is populated. [Daniel Lindsley]
- Initial faceting support complete. [Daniel Lindsley]
- Query facets tested. [Daniel Lindsley]
- Bugfix to (field) facets. [Daniel Lindsley]

  Using a dict is inappropriate, as the output from Solr is sorted by count. Now using a two-tuple.
- Backward-incompatible changes to faceting. Date-based faceting is now present. [Daniel Lindsley]
- Solr implementation of faceting started. Needs more tests. [Daniel Lindsley]
- Initial faceting support in place. Needs more thought and a Solr implementation. [Daniel Lindsley]
- Unbreak iterables in queries. [Daniel Lindsley]
- Bugfixes for Unicode handling and loading deleted models. [Daniel Lindsley]
- Fixed bug in Solr's run method. [Daniel Lindsley]
- Various bug fixes. [Daniel Lindsley]
- Backward-Incompatible: Refactored ModelIndexes to allow greater customization before indexing. See "prepare()" methods. [Daniel Lindsley]
- Updated "build_solr_schema" command for revised fields. [Daniel Lindsley]
- Refactored SearchFields. Lightly backwards-incompatible. [Daniel Lindsley]
- No more duplicates from the "build_solr_schema" management command. [Daniel Lindsley]
- Removed the kwargs. Explicit is better than implicit. [Daniel Lindsley]
- Tests for highlighting. [Daniel Lindsley]
- Added initial highlighting support. Needs tests and perhaps a better implementation. [Daniel Lindsley]
- Started "build_solr_schema" command. Needs testing with more than one index. [Daniel Lindsley]
- Argh. ".select_related()" is killing reindexes. Again. [Daniel Lindsley]
- Stored fields now come back as part of the search result. [Daniel Lindsley]
- Fixed Solr's SearchQuery.clean to handle reserved words more appropriately. [Daniel Lindsley]
- Filter types seem solid and have tests. [Daniel Lindsley]
- App renamed (for namespace/sanity/because it's really different reasons). [Daniel Lindsley]
- Started trying to support the various filter types. Needs testing and verification. [Daniel Lindsley]
- Fixed tests in light of the change to "OR". [Daniel Lindsley]
- Readded "select_related" to reindex command. [Daniel Lindsley]
- I am a moron. [Daniel Lindsley]
- "OR" is now the default operator. Also, "auto_query" now handles not'ed keywords. [Daniel Lindsley]
- "More Like This" now implemented and functioning with Solr backend. [Daniel Lindsley]
- Removed broken references to __name__. [Daniel Lindsley]
- Internal documentation fix. [Daniel Lindsley]

- Solr backend can now clear on a per-model basis. [Daniel Lindsley]
- Solr backend tests fleshed out. Initial stability of Solr. [Daniel Lindsley]

  This needs more work (as does everything) but it seems to be working reliably from my testing (both unit and "real-world"). Onward and upward.
- Massive renaming/refactoring spree. Tests 100% passing again. [Daniel Lindsley]
- Renamed BaseSearchQuerySet to SearchQuerySet. Now requires instantiation. [Daniel Lindsley]
- Standardizing syntax. [Daniel Lindsley]
- Backend support update. [Daniel Lindsley]
- An attempt to make sure the main IndexSite is always setup, even outside web requests. Also needs improvement. [Daniel Lindsley]
- Reindexes now work. [Daniel Lindsley]
- Some painful bits to make things work for now. Needs improvement. [Daniel Lindsley]
- Support kwargs on the search. [Daniel Lindsley]
- Move solr backend tests in prep for fully testing the backend. [Daniel Lindsley]
- Some ContentField/StoredField improvements. [Daniel Lindsley]

  StoredFields now have a unique template per field (as they should have from the start) and there's a touch more checking. You can also now override the template name for either type of field.
- Fixed backend loading upon unpickling SearchBackend. [Daniel Lindsley]
- Tweak internal doc. [Daniel Lindsley]
- MOAR DOCS. [Daniel Lindsley]
- Internal documentation and cleanup. Also alters the behavior of SearchQuerySet's "order_by" method slightly, bringing it more in-line with QuerySet's behavior. [Daniel Lindsley]
- Documentation/license updates. [Daniel Lindsley]
- Fixed ModelIndexes and created tests for them. 100% tests passing again. [Daniel Lindsley]
- Started refactoring ModelIndexes. Needs tests (and possibly a little love). [Daniel Lindsley]
- Implemented Solr's boost, clean, multiple order-by. Fixed Solr's score retrieval (depends on custom pysolr) and exact match syntax. [Daniel Lindsley]
- Minor changes/cleanup. [Daniel Lindsley]
- Updated docs and a FIXME. [Daniel Lindsley]
- SearchView/SearchForm tests passing. [Daniel Lindsley]
- Changed BaseSearchQuery to accept a SearchBackend instance instead of the class. [Daniel Lindsley]
- Better dummy implementation, a bugfix to raw_search and SearchView/SearchForm tests. [Daniel Lindsley]
- Temporarily changed the Solr backend to ignore fields. Pysolr will need a patch and then reenable this. [Daniel Lindsley]
- Merge branch 'master' of ssh://daniel@mckenzie/home/daniel/djangosearch_refactor into HEAD. [Daniel Lindsley]
- Started SearchView tests and added URLconf. [Daniel Lindsley]
- Started SearchView tests and added URLconf. [Daniel Lindsley]

- Added note about basic use. Needs refactoring. [Matt Croydon]
- Merged index.rst. [Matt Croydon]
- Fixed result lookups when constructing a SearchResult. [Daniel Lindsley]
- Added more docs. [Daniel Lindsley]
- Added FIXME for exploration on Solr backend. [Daniel Lindsley]
- Solr's SearchQuery now handles phrases (exact match). [Daniel Lindsley]
- More work on the Solr backend. [Daniel Lindsley]
- Added more imports for future test coverage. [Daniel Lindsley]
- Added stubs for backend tests. [Daniel Lindsley]
- Documentation updates. [Daniel Lindsley]
- Refactored forms/views. Needs tests. [Daniel Lindsley]
- Removed old entries in .gitignore. [Daniel Lindsley]
- Implemented load_all. [Daniel Lindsley]
- Fixed query result retrieval. [Daniel Lindsley]
- Updated documentation index and tweaked overview formatting. [Matt Croydon]
- Slight docs improvements. [Daniel Lindsley]
- Started work on Solr backend. [Daniel Lindsley]
- Ignore _build. [Matt Croydon]
- Refactored documentation to format better in Sphinx. [Matt Croydon]
- Added _build to .gitignore. [Matt Croydon]
- Added sphinx config for documentation. [Matt Croydon]
- Verified _fill_cache behavior. 100% test pass. [Daniel Lindsley]
- Added a couple new desirable bits of functionality. Mostly stubbed. [Daniel Lindsley]
- Removed fixme and updated docs. [Daniel Lindsley]
- Removed an old reference to SearchPaginator. [Daniel Lindsley]
- Updated import paths to new backend Base* location. [Daniel Lindsley]
- Relocated base backend classes to __init__.py for consistency with Django. [Daniel Lindsley]
- BaseSearchQuerySet initial API complete and all but working. One failing test related to caching results. [Daniel Lindsley]
- Added new (improved?) template path for index templates. [Daniel Lindsley]
- Removed SearchPaginator, as it no longer provides anything over the standard Django Paginator. [Daniel Lindsley]
- Added len/iter support to BaseSearchQuerySet. Need to finish getitem support and test. [Daniel Lindsley]
- Started to update ModelIndex. [Daniel Lindsley]
- Started to alter dummy to match new class names/API. [Daniel Lindsley]
- Little bits of cleanup. [Daniel Lindsley]

- Added overview of where functionality belongs in djangosearch. This should likely make it's way into other docs and go away eventually. [Daniel Lindsley]
- BaseSearchQuery now tracks filters via QueryFilter objects. Tests complete for QueryFilter and nearly complete for BaseSearchQuery. [Daniel Lindsley]
- Started docs on creating new backends. [Daniel Lindsley]
- Started tests for BaseSearchQuery and BaseSearchQuerySet. [Daniel Lindsley]
- Fixed site loading. [Daniel Lindsley]
- More work on the Base* classes. [Daniel Lindsley]
- Started docs on creating new backends. [Daniel Lindsley]
- Yet more work on BaseSearchQuerySet. Now with fewer FIXMEs. [Daniel Lindsley]
- More work on BaseSearchQuerySet and added initial BaseSearchQuery object. [Daniel Lindsley]
- Removed another chunk of SearchPaginator as SearchQuerySet becomes more capable. Hopefully, SearchPaginator will simply go away soon. [Daniel Lindsley]
- Fixed ModelSearchForm to check the site's registered models. [Daniel Lindsley]
- Reenabled how other backends might load. [Daniel Lindsley]
- Added ignores. [Daniel Lindsley]
- Started documenting what backends are supported and what they can do. [Daniel Lindsley]
- More work on SearchQuerySet. [Daniel Lindsley]
- More renovation and IndexSite's tests pass 100%. [Daniel Lindsley]
- Fleshed out sites tests. Need to setup environment in order to run them. [Daniel Lindsley]
- Started adding tests. [Daniel Lindsley]
- First blush at SearchQuerySet. Non-functional, trying to lay out API and basic funationality. [Daniel Lindsley]
- Removed old results.py in favor of the coming SearchQuerySet. [Daniel Lindsley]
- Noted future improvements on SearchPaginator. [Daniel Lindsley]
- Removed old reference to autodiscover and added default site a la NFA. [Daniel Lindsley]
- Commented another use of RELEVANCE. [Daniel Lindsley]
- Little backend tweaks. [Daniel Lindsley]
- Added autodiscover support. [Daniel Lindsley]
- Readded management command. [Daniel Lindsley]
- Added SearchView and ModelSearchForm back in. Needs a little work. [Daniel Lindsley]
- Readded results. Need to look at SoC for ideas. [Daniel Lindsley]
- Readded paginator. Needs docs/tests. [Daniel Lindsley]
- Readded core backends + solr. Will add others as they reach 100% functionality. [Daniel Lindsley]
- Added ModelIndex back in. Customized to match new setup. [Daniel Lindsley]
- Added signal registration as well as some introspection capabilities. [Daniel Lindsley]
- Initial commit. Basic IndexSite implementation complete. Needs tests. [Daniel Lindsley]

Contributing

Haystack is open-source and, as such, grows (or shrinks) & improves in part due to the community. Below are some guidelines on how to help with the project.

Philosophy

- Haystack is BSD-licensed. All contributed code must be either
  - the original work of the author, contributed under the BSD, or...
  - work taken from another project released under a BSD-compatible license.
- GPL'd (or similar) works are not eligible for inclusion.
- Haystack's git master branch should always be stable, production-ready & passing all tests.
- Major releases (1.x.x) are commitments to backward-compatibility of the public APIs. Any documented API should ideally not change between major releases. The exclusion to this rule is in the event of either a security issue or to accommodate changes in Django itself.
- Minor releases (x.3.x) are for the addition of substantial features or major bugfixes.
- Patch releases (x.x.4) are for minor features or bugfixes.

Guidelines For Reporting An Issue/Feature

So you've found a bug or have a great idea for a feature. Here's the steps you should take to help get it added/fixed in Haystack:

- First, check to see if there's an existing issue/pull request for the bug/feature. All issues are at https://github.com/toastdriven/django-haystack/issues and pull reqs are at https://github.com/toastdriven/django-haystack/pulls.
- If there isn't one there, please file an issue. The ideal report includes:
  - A description of the problem/suggestion.
  - How to recreate the bug.
  - If relevant, including the versions of your:
    - Python interpreter
    - Django
    - Haystack
    - Search engine used (as well as bindings)
    - Optionally of the other dependencies involved
  - Ideally, creating a pull request with a (failing) test case demonstrating what's wrong. This makes it easy for us to reproduce & fix the problem. Instructions for running the tests are at *Welcome to Haystack!*

You might also hop into the IRC channel (`#haystack` on `irc.freenode.net`) & raise your question there, as there may be someone who can help you with a work-around.

Guidelines For Contributing Code

If you're ready to take the plunge & contribute back some code/docs, the process should look like:

- Fork the project on GitHub into your own account.
- Clone your copy of Haystack.
- Make a new branch in git & commit your changes there.
- Push your new branch up to GitHub.
- Again, ensure there isn't already an issue or pull request out there on it. If there is & you feel you have a better fix, please take note of the issue number & mention it in your pull request.
- Create a new pull request (based on your branch), including what the problem/feature is, versions of your software & referencing any related issues/pull requests.

In order to be merged into Haystack, contributions must have the following:

- A solid patch that:
  - is clear.
  - works across all supported versions of Python/Django.
  - follows the existing style of the code base (mostly PEP-8).
  - comments included as needed.
- A test case that demonstrates the previous flaw that now passes with the included patch.
- If it adds/changes a public API, it must also include documentation for those changes.
- Must be appropriately licensed (see "Philosophy").
- Adds yourself to the AUTHORS file.

If your contribution lacks any of these things, they will have to be added by a core contributor before being merged into Haystack proper, which may take substantial time for the all-volunteer team to get to.

Guidelines For Core Contributors

If you've been granted the commit bit, here's how to shepherd the changes in:

- Any time you go to work on Haystack, please use `git pull --rebase` to fetch the latest changes.
- Any new features/bug fixes must meet the above guidelines for contributing code (solid patch/tests passing/docs included).
- Commits are typically cherry-picked onto a branch off master.
  - This is done so as not to include extraneous commits, as some people submit pull reqs based on their git master that has other things applied to it.
- A set of commits should be squashed down to a single commit.
  - `git merge --squash` is a good tool for performing this, as is `git rebase -i HEAD~N`.
  - This is done to prevent anyone using the git repo from accidently pulling work-in-progress commits.
- Commit messages should use past tense, describe what changed & thank anyone involved. Examples:

```
"""Added support for the latest version of Whoosh (v2.3.2)."""
"""Fixed a bug in ``solr_backend.py``. Thanks to joeschmoe for the report!"""
"""BACKWARD-INCOMPATIBLE: Altered the arguments passed to ``SearchBackend``.

Further description appears here if the change warrants an explanation
as to why it was done."""
```

- For any patches applied from a contributor, please ensure their name appears in the AUTHORS file.
- When closing issues or pull requests, please reference the SHA in the closing message (i.e. `Thanks! Fixed in SHA: 6b93f6`). GitHub will automatically link to it.

Python 3 Support

As of Haystack v2.1.0, it has been ported to support both Python 2 & Python 3 within the same codebase. This builds on top of what six & Django provide.

No changes are required for anyone running an existing Haystack installation. The API is completely backward-compatible, so you should be able to run your existing software without modification.

Virtually all tests pass under both Python 2 & 3, with a small number of expected failures under Python (typically related to ordering, see below).

Supported Backends

The following backends are fully supported under Python 3. However, you may need to update these dependencies if you have a pre-existing setup.

- Solr (pysolr>=3.1.0)
- Elasticsearch

Notes

Testing

If you were testing things such as the query generated by a given `SearchQuerySet` or how your forms would render, under Python 3.3.2+, hash randomization is in effect, which means that the ordering of dictionaries is no longer consistent, even on the same platform.

Haystack took the approach of abandoning making assertions about the entire structure. Instead, we either simply assert that the new object contains the right things or make a call to `sorted(...)` around it to ensure order. It is recommended you take a similar approach.

Migrating From Haystack 1.X to Haystack 2.X

Haystack introduced several backward-incompatible changes in the process of moving from the 1.X series to the 2.X series. These were done to clean up the API, to support new features & to clean up problems in 1.X. At a high level, they consisted of:

- The removal of `SearchSite` & `haystack.site`.
- The removal of `handle_registrations` & `autodiscover`.

- The addition of multiple index support.
- The addition of `SignalProcessors` & the removal of `RealTimeSearchIndex`.
- The removal/renaming of various settings.

This guide will help you make the changes needed to be compatible with Haystack 2.X.

Settings

Most prominently, the old way of specifying a backend & its settings has changed to support the multiple index feature. A complete Haystack 1.X example might look like:

```
HAYSTACK_SEARCH_ENGINE = 'solr'
HAYSTACK_SOLR_URL = 'http://localhost:9001/solr/default'
HAYSTACK_SOLR_TIMEOUT = 60 * 5
HAYSTACK_INCLUDE_SPELLING = True
HAYSTACK_BATCH_SIZE = 100

# Or...
HAYSTACK_SEARCH_ENGINE = 'whoosh'
HAYSTACK_WHOOSH_PATH = '/home/search/whoosh_index'
HAYSTACK_WHOOSH_STORAGE = 'file'
HAYSTACK_WHOOSH_POST_LIMIT = 128 * 1024 * 1024
HAYSTACK_INCLUDE_SPELLING = True
HAYSTACK_BATCH_SIZE = 100

# Or...
HAYSTACK_SEARCH_ENGINE = 'xapian'
HAYSTACK_XAPIAN_PATH = '/home/search/xapian_index'
HAYSTACK_INCLUDE_SPELLING = True
HAYSTACK_BATCH_SIZE = 100
```

In Haystack 2.X, you can now supply as many backends as you like, so all of the above settings can now be active at the same time. A translated set of settings would look like:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://localhost:9001/solr/default',
        'TIMEOUT': 60 * 5,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
    },
    'autocomplete': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': '/home/search/whoosh_index',
        'STORAGE': 'file',
        'POST_LIMIT': 128 * 1024 * 1024,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
    },
    'slave': {
        'ENGINE': 'xapian_backend.XapianEngine',
        'PATH': '/home/search/xapian_index',
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
```

```
    },
}
```

You are required to have at least one connection listed within `HAYSTACK_CONNECTIONS`, it must be named `default` & it must have a valid `ENGINE` within it. Bare minimum looks like:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.simple_backend.SimpleEngine'
    }
}
```

The key for each backend is an identifier you use to describe the backend within your app. You should refer to the *Multiple Indexes* documentation for more information on using the new multiple indexes & routing features.

Also note that the `ENGINE` setting has changed from a lowercase "short name" of the engine to a full path to a new `Engine` class within the backend. Available options are:

- `haystack.backends.solr_backend.SolrEngine`
- `haystack.backends.whoosh_backend.WhooshEngine`
- `haystack.backends.simple_backend.SimpleEngine`

Additionally, the following settings were outright removed & will generate an exception if found:

- `HAYSTACK_SITECONF` - Remove this setting & the file it pointed to.
- `HAYSTACK_ENABLE_REGISTRATIONS`
- `HAYSTACK_INCLUDE_SPELLING`

Backends

The `dummy` backend was outright removed from Haystack, as it served very little use after the `simple` (pure-ORM-powered) backend was introduced.

If you wrote a custom backend, please refer to the "Custom Backends" section below.

Indexes

The other major changes affect the `SearchIndex` class. As the concept of `haystack.site` & `SearchSite` are gone, you'll need to modify your indexes.

A Haystack 1.X index might've looked like:

```
import datetime
from haystack.indexes import *
from haystack import site
from myapp.models import Note


class NoteIndex(SearchIndex):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_queryset(self):
```

```
        """Used when the entire index for model is updated."""
        return Note.objects.filter(pub_date__lte=datetime.datetime.now())


site.register(Note, NoteIndex)
```

A converted Haystack 2.X index should look like:

```
import datetime
from haystack import indexes
from myapp.models import Note


class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        """Used when the entire index for model is updated."""
        return self.get_model().objects.filter(pub_date__lte=datetime.datetime.now())
```

Note the import on `site` & the registration statements are gone. Newly added are is the `NoteIndex.get_model` method. This is a **required** method & should simply return the `Model` class the index is for.

There's also a new, additional class added to the `class` definition. The `indexes.Indexable` class is a simple mixin that serves to identify the classes Haystack should automatically discover & use. If you have a custom base class (say `QueuedSearchIndex`) that other indexes inherit from, simply leave the `indexes.Indexable` off that declaration & Haystack won't try to use it.

Additionally, the name of the `document=True` field is now enforced to be `text` across all indexes. If you need it named something else, you should set the `HAYSTACK_DOCUMENT_FIELD` setting. For example:

```
HAYSTACK_DOCUMENT_FIELD = 'pink_polka_dot'
```

Finally, the `index_queryset` method should supplant the `get_queryset` method. This was present in the Haystack 1.2.X series (with a deprecation warning in 1.2.4+) but has been removed in Haystack v2.

Finally, if you were unregistering other indexes before, you should make use of the new `EXCLUDED_INDEXES` setting available in each backend's settings. It should be a list of strings that contain the Python import path to the indexes that should not be loaded & used. For example:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://localhost:9001/solr/default',
        'EXCLUDED_INDEXES': [
            # Imagine that these indexes exist. They don't.
            'django.contrib.auth.search_indexes.UserIndex',
            'third_party_blog_app.search_indexes.EntryIndex',
        ]
    }
}
```

This allows for reliable swapping of the index that handles a model without relying on correct import order.

Removal of `RealTimeSearchIndex`

Use of the `haystack.indexes.RealTimeSearchIndex` is no longer valid. It has been removed in favor of `RealtimeSignalProcessor`. To migrate, first change the inheritance of all your `RealTimeSearchIndex` subclasses to use `SearchIndex` instead:

```
# Old.
class MySearchIndex(indexes.RealTimeSearchIndex, indexes.Indexable):
    # ...


# New.
class MySearchIndex(indexes.SearchIndex, indexes.Indexable):
    # ...
```

Then update your settings to enable use of the `RealtimeSignalProcessor`:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'
```

Done!

For most basic uses of Haystack, this is all that is necessary to work with Haystack 2.X. You should rebuild your index if needed & test your new setup.

Advanced Uses

Swapping Backend

If you were manually swapping the `SearchQuery` or `SearchBackend` being used by `SearchQuerySet` in the past, it's now preferable to simply setup another connection & use the `SearchQuerySet.using` method to select that connection instead.

Also, if you were manually instantiating `SearchBackend` or `SearchQuery`, it's now preferable to rely on the connection's engine to return the right thing. For example:

```
from haystack import connections
backend = connections['default'].get_backend()
query = connections['default'].get_query()
```

Custom Backends

If you had written a custom `SearchBackend` and/or custom `SearchQuery`, there's a little more work needed to be Haystack 2.X compatible.

You should, but don't have to, rename your `SearchBackend` & `SearchQuery` classes to be more descriptive/less collide-y. For example, `solr_backend.SearchBackend` became `solr_backend.SolrSearchBackend`. This prevents non-namespaced imports from stomping on each other.

You need to add a new class to your backend, subclassing `BaseEngine`. This allows specifying what `backend` & `query` should be used on a connection with less duplication/naming trickery. It goes at the bottom of the file (so that the classes are defined above it) and should look like:

```
from haystack.backends import BaseEngine
from haystack.backends.solr_backend import SolrSearchQuery

# Code then...

class MyCustomSolrEngine(BaseEngine):
    # Use our custom backend.
    backend = MySolrBackend
    # Use the built-in Solr query.
    query = SolrSearchQuery
```

Your `HAYSTACK_CONNECTIONS['default']['ENGINE']` should then point to the full Python import path to your new `BaseEngine` subclass.

Finally, you will likely have to adjust the `SearchBackend.__init__` & `SearchQuery.__init__`, as they have changed significantly. Please refer to the commits for those backends.

CHAPTER 2

Advanced Uses

Once you've got Haystack working, here are some of the more complex features you may want to include in your application.

Best Practices

What follows are some general recommendations on how to improve your search. Some tips represent performance benefits, some provide a better search index. You should evaluate these options for yourself and pick the ones that will work best for you. Not all situations are created equal and many of these options could be considered mandatory in some cases and unnecessary premature optimizations in others. Your mileage may vary.

Good Search Needs Good Content

Most search engines work best when they're given corpuses with predominantly text (as opposed to other data like dates, numbers, etc.) in decent quantities (more than a couple words). This is in stark contrast to the databases most people are used to, which rely heavily on non-text data to create relationships and for ease of querying.

To this end, if search is important to you, you should take the time to carefully craft your `SearchIndex` subclasses to give the search engine the best information you can. This isn't necessarily hard but is worth the investment of time and thought. Assuming you've only ever used the `BasicSearchIndex`, in creating custom `SearchIndex` classes, there are some easy improvements to make that will make your search better:

- For your `document=True` field, use a well-constructed template.
- Add fields for data you might want to be able to filter by.
- If the model has related data, you can squash good content from those related models into the parent model's `SearchIndex`.
- Similarly, if you have heavily de-normalized models, it may be best represented by a single indexed model rather than many indexed models.

Well-Constructed Templates

A relatively unique concept in Haystack is the use of templates associated with `SearchIndex` fields. These are data templates, will never been seen by users and ideally contain no HTML. They are used to collect various data from the model and structure it as a document for the search engine to analyze and index.

Note: If you read nothing else, this is the single most important thing you can do to make search on your site better for your users. Good templates can make or break your search and providing the search engine with good content to index is critical.

Good templates structure the data well and incorporate as much pertinent text as possible. This may include additional fields such as titles, author information, metadata, tags/categories. Without being artificial, you want to construct as much context as you can. This doesn't mean you should necessarily include every field, but you should include fields that provide good content or include terms you think your users may frequently search on.

Unless you have very unique numbers or dates, neither of these types of data are a good fit within templates. They are usually better suited to other fields for filtering within a `SearchQuerySet`.

Additional Fields For Filtering

Documents by themselves are good for generating indexes of content but are generally poor for filtering content, for instance, by date. All search engines supported by Haystack provide a means to associate extra data as attributes/fields on a record. The database analogy would be adding extra columns to the table for filtering.

Good candidates here are date fields, number fields, de-normalized data from related objects, etc. You can expose these things to users in the form of a calendar range to specify, an author to look up or only data from a certain series of numbers to return.

You will need to plan ahead and anticipate what you might need to filter on, though with each field you add, you increase storage space usage. It's generally **NOT** recommended to include every field from a model, just ones you are likely to use.

Related Data

Related data is somewhat problematic to deal with, as most search engines are better with documents than they are with relationships. One way to approach this is to de-normalize a related child object or objects into the parent's document template. The inclusion of a foreign key's relevant data or a simple Django `{% for %}` templatetag to iterate over the related objects can increase the salient data in your document. Be careful what you include and how you structure it, as this can have consequences on how well a result might rank in your search.

Avoid Hitting The Database

A very easy but effective thing you can do to drastically reduce hits on the database is to pre-render your search results using stored fields then disabling the `load_all` aspect of your `SearchView`.

> **Warning:** This technique may cause a substantial increase in the size of your index as you are basically using it as a storage mechanism.

To do this, you setup one or more stored fields (*indexed=False*) on your `SearchIndex` classes. You should specify a template for the field, filling it with the data you'd want to display on your search results pages. When the model

attached to the `SearchIndex` is placed in the index, this template will get rendered and stored in the index alongside the record.

Note: The downside of this method is that the HTML for the result will be locked in once it is indexed. To make changes to the structure, you'd have to reindex all of your content. It also limits you to a single display of the content (though you could use multiple fields if that suits your needs).

The second aspect is customizing your `SearchView` and its templates. First, pass the `load_all=False` to your `SearchView`, ideally in your URLconf. This prevents the `SearchQuerySet` from loading all models objects for results ahead of time. Then, in your template, simply display the stored content from your `SearchIndex` as the HTML result.

Warning: To do this, you must absolutely avoid using `{{ result.object }}` or any further accesses beyond that. That call will hit the database, not only nullifying your work on lessening database hits, but actually making it worse as there will now be at least query for each result, up from a single query for each type of model with `load_all=True`.

Content-Type Specific Templates

Frequently, when displaying results, you'll want to customize the HTML output based on what model the result represents.

In practice, the best way to handle this is through the use of `include` along with the data on the `SearchResult`.

Your existing loop might look something like:

```
{% for result in page.object_list %}
    <p>
        <a href="{{ result.object.get_absolute_url }}">{{ result.object.title }}</a>
    </p>
{% empty %}
    <p>No results found.</p>
{% endfor %}
```

An improved version might look like:

```
{% for result in page.object_list %}
    {% if result.content_type == "blog.post" %}
    {% include "search/includes/blog/post.html" %}
    {% endif %}
    {% if result.content_type == "media.photo" %}
    {% include "search/includes/media/photo.html" %}
    {% endif %}
{% empty %}
    <p>No results found.</p>
{% endfor %}
```

Those include files might look like:

```
# search/includes/blog/post.html
<div class="post_result">
    <h3><a href="{{ result.object.get_absolute_url }}">{{ result.object.title }}</a></
↪h3>
```

```
    <p>{{ result.object.tease }}</p>
</div>

# search/includes/media/photo.html
<div class="photo_result">
    <a href="{{ result.object.get_absolute_url }}">
    <img src="http://your.media.example.com/media/{{ result.object.photo.url }}"></a>
    <p>Taken By {{ result.object.taken_by }}</p>
</div>
```

You can make this even better by standardizing on an includes layout, then writing a template tag or filter that generates the include filename. Usage might looks something like:

```
{% for result in page.object_list %}
    {% with result|search_include as fragment %}
    {% include fragment %}
    {% endwith %}
{% empty %}
    <p>No results found.</p>
{% endfor %}
```

Real-Time Search

If your site sees heavy search traffic and up-to-date information is very important, Haystack provides a way to constantly keep your index up to date.

You can enable the `RealtimeSignalProcessor` within your settings, which will allow Haystack to automatically update the index whenever a model is saved/deleted.

You can find more information within the *Signal Processors* documentation.

Use Of A Queue For A Better User Experience

By default, you have to manually reindex content, Haystack immediately tries to merge it into the search index. If you have a write-heavy site, this could mean your search engine may spend most of its time churning on constant merges. If you can afford a small delay between when a model is saved and when it appears in the search results, queuing these merges is a good idea.

You gain a snappier interface for users as updates go into a queue (a fast operation) and then typical processing continues. You also get a lower churn rate, as most search engines deal with batches of updates better than many single updates. You can also use this to distribute load, as the queue consumer could live on a completely separate server from your webservers, allowing you to tune more efficiently.

Implementing this is relatively simple. There are two parts, creating a new `QueuedSignalProcessor` class and creating a queue processing script to handle the actual updates.

For the `QueuedSignalProcessor`, you should inherit from `haystack.signals.BaseSignalProcessor`, then alter the `setup/teardown` methods to call an enqueuing method instead of directly calling `handle_save/handle_delete`. For example:

```
from haystack import signals


class QueuedSignalProcessor(signals.BaseSignalProcessor):
    # Override the built-in.
    def setup(self):
```

```
        models.signals.post_save.connect(self.enqueue_save)
        models.signals.post_delete.connect(self.enqueue_delete)

    # Override the built-in.
    def teardown(self):
        models.signals.post_save.disconnect(self.enqueue_save)
        models.signals.post_delete.disconnect(self.enqueue_delete)

    # Add on a queuing method.
    def enqueue_save(self, sender, instance, **kwargs):
        # Push the save & information onto queue du jour here
        ...

    # Add on a queuing method.
    def enqueue_delete(self, sender, instance, **kwargs):
        # Push the delete & information onto queue du jour here
        ...
```

For the consumer, this is much more specific to the queue used and your desired setup. At a minimum, you will need to periodically consume the queue, fetch the correct index from the `SearchSite` for your application, load the model from the message and pass that model to the `update_object` or `remove_object` methods on the `SearchIndex`. Proper grouping, batching and intelligent handling are all additional things that could be applied on top to further improve performance.

Highlighting

Haystack supports two different methods of highlighting. You can either use `SearchQuerySet.highlight` or the built-in `{% highlight %}` template tag, which uses the `Highlighter` class. Each approach has advantages and disadvantages you need to weigh when deciding which to use.

If you want portable, flexible, decently fast code, the `{% highlight %}` template tag (or manually using the underlying `Highlighter` class) is the way to go. On the other hand, if you care more about speed and will only ever be using one backend, `SearchQuerySet.highlight` may suit your needs better.

Use of `SearchQuerySet.highlight` is documented in the *SearchQuerySet API* documentation and the `{% highlight %}` tag is covered in the *Template Tags* documentation, so the rest of this material will cover the `Highlighter` implementation.

Highlighter

The `Highlighter` class is a pure-Python implementation included with Haystack that's designed for flexibility. If you use the `{% highlight %}` template tag, you'll be automatically using this class. You can also use it manually in your code. For example:

```
>>> from haystack.utils import Highlighter

>>> my_text = 'This is a sample block that would be more meaningful in real life.'
>>> my_query = 'block meaningful'

>>> highlight = Highlighter(my_query)
>>> highlight.highlight(my_text)
u'...<span class="highlighted">block</span> that would be more <span class=
↪"highlighted">meaningful</span> in real life.'
```

The default implementation takes three optional kwargs: `html_tag`, `css_class` and `max_length`. These allow for basic customizations to the output, like so:

```
>>> from haystack.utils import Highlighter

>>> my_text = 'This is a sample block that would be more meaningful in real life.'
>>> my_query = 'block meaningful'

>>> highlight = Highlighter(my_query, html_tag='div', css_class='found', max_
↪length=35)
>>> highlight.highlight(my_text)
u'...<div class="found">block</div> that would be more <div class="found">meaningful</
↪div>...'
```

Further, if this implementation doesn't suit your needs, you can define your own custom highlighter class. As long as it implements the API you've just seen, it can highlight however you choose. For example:

```
# In ``myapp/utils.py``...
from haystack.utils import Highlighter

class BorkHighlighter(Highlighter):
    def render_html(self, highlight_locations=None, start_offset=None, end_
↪offset=None):
        highlighted_chunk = self.text_block[start_offset:end_offset]

        for word in self.query_words:
            highlighted_chunk = highlighted_chunk.replace(word, 'Bork!')

        return highlighted_chunk
```

Then set the `HAYSTACK_CUSTOM_HIGHLIGHTER` setting to `myapp.utils.BorkHighlighter`. Usage would then look like:

```
>>> highlight = BorkHighlighter(my_query)
>>> highlight.highlight(my_text)
u'Bork! that would be more Bork! in real life.'
```

Now the `{% highlight %}` template tag will also use this highlighter.

Faceting

What Is Faceting?

Faceting is a way to provide users with feedback about the number of documents which match terms they may be interested in. At its simplest, it gives document counts based on words in the corpus, date ranges, numeric ranges or even advanced queries.

Faceting is particularly useful when trying to provide users with drill-down capabilities. The general workflow in this regard is:

1. You can choose what you want to facet on.
2. The search engine will return the counts it sees for that match.
3. You display those counts to the user and provide them with a link.

4. When the user chooses a link, you narrow the search query to only include those conditions and display the results, potentially with further facets.

Note: Faceting can be difficult, especially in providing the user with the right number of options and/or the right areas to be able to drill into. This is unique to every situation and demands following what real users need.

You may want to consider logging queries and looking at popular terms to help you narrow down how you can help your users.

Haystack provides functionality so that all of the above steps are possible. From the ground up, let's build a faceted search setup. This assumes that you have been to work through the *Getting Started with Haystack* and have a working Haystack installation. The same setup from the *Getting Started with Haystack* applies here.

1. Determine Facets And `SearchQuerySet`

Determining what you want to facet on isn't always easy. For our purposes, we'll facet on the `author` field.

In order to facet effectively, the search engine should store both a standard representation of your data as well as exact version to facet on. This is generally accomplished by duplicating the field and storing it via two different types. Duplication is suggested so that those fields are still searchable in the standard ways.

To inform Haystack of this, you simply pass along a `faceted=True` parameter on the field(s) you wish to facet on. So to modify our existing example:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user', faceted=True)
    pub_date = DateTimeField(model_attr='pub_date')
```

Haystack quietly handles all of the backend details for you, creating a similar field to the type you specified with `_exact` appended. Our example would now have both a `author` and `author_exact` field, though this is largely an implementation detail.

To pull faceting information out of the index, we'll use the `SearchQuerySet.facet` method to setup the facet and the `SearchQuerySet.facet_counts` method to retrieve back the counts seen.

Experimenting in a shell (`./manage.py shell`) is a good way to get a feel for what various facets might look like:

```
>>> from haystack.query import SearchQuerySet
>>> sqs = SearchQuerySet().facet('author')
>>> sqs.facet_counts()
{
    'dates': {},
    'fields': {
        'author': [
            ('john', 4),
            ('daniel', 2),
            ('sally', 1),
            ('terry', 1),
        ],
    },
    'queries': {}
}
```

Note: Note that, despite the duplication of fields, you should provide the regular name of the field when faceting. Haystack will intelligently handle the underlying details and mapping.

As you can see, we get back a dictionary which provides access to the three types of facets available: `fields`, `dates` and `queries`. Since we only faceted on the `author` field (which actually facets on the `author_exact` field managed by Haystack), only the `fields` key has any data associated with it. In this case, we have a corpus of eight documents with four unique authors.

Note: Facets are chainable, like most `SearchQuerySet` methods. However, unlike most `SearchQuerySet` methods, they are *NOT* affected by `filter` or similar methods. The only method that has any effect on facets is the `narrow` method (which is how you provide drill-down).

Configuring facet behaviour

You can configure the behaviour of your facets by passing options for each facet in your SearchQuerySet. These options can be backend specific.

limit *tested on Solr*

The `limit` parameter limits the results for each query. On Solr, the default facet.limit is 100 and a negative number removes the limit.

Example usage:

```
>>> from haystack.query import SearchQuerySet
>>> sqs = SearchQuerySet().facet('author', limit=-1)
>>> sqs.facet_counts()
{
    'dates': {},
    'fields': {
        'author': [
            ('abraham', 1),
            ('benny', 2),
            ('cindy', 1),
            ('diana', 5),
        ],
    },
    'queries': {}
}

>>> sqs = SearchQuerySet().facet('author', limit=2)
>>> sqs.facet_counts()
{
    'dates': {},
    'fields': {
        'author': [
            ('abraham', 1),
            ('benny', 2),
        ],
    },
    'queries': {}
}
```

sort *tested on Solr*

The `sort` parameter will sort the results for each query. Solr's default facet.sort is `index`, which will sort the facets alphabetically. Changing the parameter to `count` will sort the facets by the number of results for each facet value.

Example usage:

```
>>> from haystack.query import SearchQuerySet
>>> sqs = SearchQuerySet().facet('author', sort='index', )
>>> sqs.facet_counts()
{
    'dates': {},
    'fields': {
        'author': [
            ('abraham', 1),
            ('benny', 2),
            ('cindy', 1),
            ('diana', 5),
        ],
    },
    'queries': {}
}

>>> sqs = SearchQuerySet().facet('author', sort='count', )
>>> sqs.facet_counts()
{
    'dates': {},
    'fields': {
        'author': [
            ('diana', 5),
            ('benny', 2),
            ('abraham', 1),
            ('cindy', 1),
        ],
    },
    'queries': {}
}
```

Now that we have the facet we want, it's time to implement it.

2. Switch to the `FacetedSearchView` and `FacetedSearchForm`

There are three things that we'll need to do to expose facets to our frontend. The first is construct the `SearchQuerySet` we want to use. We should have that from the previous step. The second is to switch to the `FacetedSearchView`. This view is useful because it prepares the facet counts and provides them in the context as `facets`.

Optionally, the third step is to switch to the `FacetedSearchForm`. As it currently stands, this is only useful if you want to provide drill-down, though it may provide more functionality in the future. We'll do it for the sake of having it in place but know that it's not required.

In your URLconf, you'll need to switch to the `FacetedSearchView`. Your URLconf should resemble:

```
from django.conf.urls import url
from haystack.forms import FacetedSearchForm
from haystack.views import FacetedSearchView


urlpatterns = [
```

```
    url(r'^$', FacetedSearchView(form_class=FacetedSearchForm, facet_fields=['author
↪']), name='haystack_search'),
]
```

The `FacetedSearchView` will now instantiate the `FacetedSearchForm`. The specified `facet_fields` will be present in the context variable `facets`. This is added in an overridden `extra_context` method.

3. Display The Facets In The Template

Templating facets involves simply adding an extra bit of processing to display the facets (and optionally to link to provide drill-down). An example template might look like this:

```
<form method="get" action=".">
    <table>
        <tbody>
            {{ form.as_table }}
            <tr>
                <td> </td>
                <td><input type="submit" value="Search"></td>
            </tr>
        </tbody>
    </table>
</form>

{% if query %}
    <!-- Begin faceting. -->
    <h2>By Author</h2>

    <div>
        <dl>
            {% if facets.fields.author %}
                <dt>Author</dt>
                {# Provide only the top 5 authors #}
                {% for author in facets.fields.author|slice:":5" %}
                    <dd><a href="{{ request.get_full_path }}&amp;selected_
↪facets=author_exact:{{ author.0|urlencode }}">{{ author.0 }}</a> ({{ author.1 }})</
↪dd>
                {% endfor %}
            {% else %}
                <p>No author facets.</p>
            {% endif %}
        </dl>
    </div>
    <!-- End faceting -->

    <!-- Display results... -->
    {% for result in page.object_list %}
        <div class="search_result">
            <h3><a href="{{ result.object.get_absolute_url }}">{{ result.object.title␣
↪}}</a></h3>

            <p>{{ result.object.body|truncatewords:80 }}</p>
        </div>
    {% empty %}
        <p>Sorry, no results found.</p>
    {% endfor %}
{% endif %}
```

Displaying the facets is a matter of looping through the facets you want and providing the UI to suit. The `author.0` is the facet text from the backend and the `author.1` is the facet count.

4. Narrowing The Search

We've also set ourselves up for the last bit, the drill-down aspect. By appending on the `selected_facets` to the URLs, we're informing the `FacetedSearchForm` that we want to narrow our results to only those containing the author we provided.

For a concrete example, if the facets on author come back as:

```
{
    'dates': {},
    'fields': {
        'author': [
            ('john', 4),
            ('daniel', 2),
            ('sally', 1),
            ('terry', 1),
        ],
    },
    'queries': {}
}
```

You should present a list similar to:

```
<ul>
    <li><a href="/search/?q=Haystack&selected_facets=author_exact:john">john</a> (4)</
↪li>
    <li><a href="/search/?q=Haystack&selected_facets=author_exact:daniel">daniel</a>␣
↪(2)</li>
    <li><a href="/search/?q=Haystack&selected_facets=author_exact:sally">sally</a> (1)
↪</li>
    <li><a href="/search/?q=Haystack&selected_facets=author_exact:terry">terry</a> (1)
↪</li>
</ul>
```

Warning: Haystack can automatically handle most details around faceting. However, since `selected_facets` is passed directly to narrow, it must use the duplicated field name. Improvements to this are planned but incomplete.

This is simply the default behavior but it is possible to override or provide your own form which does additional processing. You could also write your own faceted `SearchView`, which could provide additional/different facets based on facets chosen. There is a wide range of possibilities available to help the user navigate your content.

Autocomplete

Autocomplete is becoming increasingly common as an add-on to search. Haystack makes it relatively simple to implement. There are two steps in the process, one to prepare the data and one to implement the actual search.

Step 1. Setup The Data

To do autocomplete effectively, the search backend uses n-grams (essentially a small window passed over the string). Because this alters the way your data needs to be stored, the best approach is to add a new field to your `SearchIndex` that contains the text you want to autocomplete on.

You have two choices: `NgramField` and `EdgeNgramField`. Though very similar, the choice of field is somewhat important.

- If you're working with standard text, `EdgeNgramField` tokenizes on whitespace. This prevents incorrect matches when part of two different words are mashed together as one n-gram. **This is what most users should use.**
- If you're working with Asian languages or want to be able to autocomplete across word boundaries, `NgramField` should be what you use.

Example (continuing from the tutorial):

```
import datetime
from haystack import indexes
from myapp.models import Note


class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')
    # We add this for autocomplete.
    content_auto = indexes.EdgeNgramField(model_attr='content')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        """Used when the entire index for model is updated."""
        return Note.objects.filter(pub_date__lte=datetime.datetime.now())
```

As with all schema changes, you'll need to rebuild/update your index after making this change.

Step 2. Performing The Query

Haystack ships with a convenience method to perform most autocomplete searches. You simply provide a field and the query you wish to search on to the `SearchQuerySet.autocomplete` method. Given the previous example, an example search would look like:

```
from haystack.query import SearchQuerySet

SearchQuerySet().autocomplete(content_auto='old')
# Result match things like 'goldfish', 'cuckold' and 'older'.
```

The results from the `SearchQuerySet.autocomplete` method are full search results, just like any regular filter.

If you need more control over your results, you can use standard `SearchQuerySet.filter` calls. For instance:

```
from haystack.query import SearchQuerySet

sqs = SearchQuerySet().filter(content_auto=request.GET.get('q', ''))
```

This can also be extended to use `SQ` for more complex queries (and is what's being done under the hood in the `SearchQuerySet.autocomplete` method).

Example Implementation

The above is the low-level backend portion of how you implement autocomplete. To make it work in browser, you need both a view to run the autocomplete and some Javascript to fetch the results.

Since it comes up often, here is an example implementation of those things.

> **Warning:** This code comes with no warranty. Don't ask for support on it. If you copy-paste it and it burns down your server room, I'm not liable for any of it.
>
> It worked this one time on my machine in a simulated environment.
>
> And yeah, semicolon-less + 2 space + comma-first. Deal with it.

A stripped-down view might look like:

```
# views.py
import simplejson as json
from django.http import HttpResponse
from haystack.query import SearchQuerySet


def autocomplete(request):
    sqs = SearchQuerySet().autocomplete(content_auto=request.GET.get('q', ''))[:5]
    suggestions = [result.title for result in sqs]
    # Make sure you return a JSON object, not a bare list.
    # Otherwise, you could be vulnerable to an XSS attack.
    the_data = json.dumps({
        'results': suggestions
    })
    return HttpResponse(the_data, content_type='application/json')
```

The template might look like:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Autocomplete Example</title>
</head>
<body>
  <h1>Autocomplete Example</h1>

  <form method="post" action="/search/" class="autocomplete-me">
    <input type="text" id="id_q" name="q">
    <input type="submit" value="Search!">
  </form>

  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.3/jquery.min.js"></
↪script>
  <script type="text/javascript">
    // In a perfect world, this would be its own library file that got included
    // on the page and only the ``$(document).ready(...)`` below would be present.
```

```
    // But this is an example.
    var Autocomplete = function(options) {
      this.form_selector = options.form_selector
      this.url = options.url || '/search/autocomplete/'
      this.delay = parseInt(options.delay || 300)
      this.minimum_length = parseInt(options.minimum_length || 3)
      this.form_elem = null
      this.query_box = null
    }

    Autocomplete.prototype.setup = function() {
      var self = this

      this.form_elem = $(this.form_selector)
      this.query_box = this.form_elem.find('input[name=q]')

      // Watch the input box.
      this.query_box.on('keyup', function() {
        var query = self.query_box.val()

        if(query.length < self.minimum_length) {
          return false
        }

        self.fetch(query)
      })

      // On selecting a result, populate the search field.
      this.form_elem.on('click', '.ac-result', function(ev) {
        self.query_box.val($(this).text())
        $('.ac-results').remove()
        return false
      })
    }

    Autocomplete.prototype.fetch = function(query) {
      var self = this

      $.ajax({
        url: this.url
      , data: {
          'q': query
        }
      , success: function(data) {
          self.show_results(data)
        }
      })
    }

    Autocomplete.prototype.show_results = function(data) {
      // Remove any existing results.
      $('.ac-results').remove()

      var results = data.results || []
      var results_wrapper = $('<div class="ac-results"></div>')
      var base_elem = $('<div class="result-wrapper"><a href="#" class="ac-result"></
↪a></div>')
```

```
      if(results.length > 0) {
        for(var res_offset in results) {
          var elem = base_elem.clone()
          // Don't use .html(...) here, as you open yourself to XSS.
          // Really, you should use some form of templating.
          elem.find('.ac-result').text(results[res_offset])
          results_wrapper.append(elem)
        }
      }
      else {
        var elem = base_elem.clone()
        elem.text("No results found.")
        results_wrapper.append(elem)
      }

      this.query_box.after(results_wrapper)
    }

    $(document).ready(function() {
      window.autocomplete = new Autocomplete({
        form_selector: '.autocomplete-me'
      })
      window.autocomplete.setup()
    })
  </script>
</body>
</html>
```

Boost

Scoring is a critical component of good search. Normal full-text searches automatically score a document based on how well it matches the query provided. However, sometimes you want certain documents to score better than they otherwise would. Boosting is a way to achieve this. There are three types of boost:

- Term Boost
- Document Boost
- Field Boost

Note: Document & Field boost support was added in Haystack 1.1.

Despite all being types of boost, they take place at different times and have slightly different effects on scoring.

Term boost happens at query time (when the search query is run) and is based around increasing the score if a certain word/phrase is seen.

On the other hand, document & field boosts take place at indexing time (when the document is being added to the index). Document boost causes the relevance of the entire result to go up, where field boost causes only searches within that field to do better.

Warning: Be warned that boost is very, very sensitive & can hurt overall search quality if over-zealously applied. Even very small adjustments can affect relevance in a big way.

Term Boost

Term boosting is achieved by using `SearchQuerySet.boost`. You provide it the term you want to boost on & a floating point value (based around `1.0` as 100% - no boost).

Example:

```
# Slight increase in relevance for documents that include "banana".
sqs = SearchQuerySet().boost('banana', 1.1)

# Big decrease in relevance for documents that include "blueberry".
sqs = SearchQuerySet().boost('blueberry', 0.8)
```

See the *SearchQuerySet API* docs for more details on using this method.

Document Boost

Document boosting is done by adding a `boost` field to the prepared data `SearchIndex` creates. The best way to do this is to override `SearchIndex.prepare`:

```
from haystack import indexes
from notes.models import Note


class NoteSearchIndex(indexes.SearchIndex, indexes.Indexable):
    # Your regular fields here then...

    def prepare(self, obj):
        data = super(NoteSearchIndex, self).prepare(obj)
        data['boost'] = 1.1
        return data
```

Another approach might be to add a new field called `boost`. However, this can skew your schema and is not encouraged.

Field Boost

Field boosting is enabled by setting the `boost` kwarg on the desired field. An example of this might be increasing the significance of a `title`:

```
from haystack import indexes
from notes.models import Note


class NoteSearchIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    title = indexes.CharField(model_attr='title', boost=1.125)

    def get_model(self):
        return Note
```

Note: Field boosting only has an effect when the SearchQuerySet filters on the field which has been boosted. If you are using a default search view or form you will need override the search method or other include the field in your search query. This example CustomSearchForm searches the automatic `content` field and the `title` field which has been boosted:

```
from haystack.forms import SearchForm

class CustomSearchForm(SearchForm):

    def search(self):
        if not self.is_valid():
            return self.no_query_found()

        if not self.cleaned_data.get('q'):
            return self.no_query_found()

        q = self.cleaned_data['q']
        sqs = self.searchqueryset.filter(SQ(content=AutoQuery(q)) |␣
↪SQ(title=AutoQuery(q)))

        if self.load_all:
            sqs = sqs.load_all()

        return sqs.highlight()
```

Signal Processors

Keeping data in sync between the (authoritative) database & the (non-authoritative) search index is one of the more difficult problems when using Haystack. Even frequently running the `update_index` management command still introduces lag between when the data is stored & when it's available for searching.

A solution to this is to incorporate Django's signals (specifically `models.db.signals.post_save` & `models.db.signals.post_delete`), which then trigger *individual* updates to the search index, keeping them in near-perfect sync.

Older versions of Haystack (pre-v2.0) tied the `SearchIndex` directly to the signals, which caused occasional conflicts of interest with third-party applications.

To solve this, starting with Haystack v2.0, the concept of a `SignalProcessor` has been introduced. In it's simplest form, the `SignalProcessor` listens to whatever signals are setup & can be configured to then trigger the updates without having to change any `SearchIndex` code.

> **Warning:** Incorporating Haystack's `SignalProcessor` into your setup **will** increase the overall load (CPU & perhaps I/O depending on configuration). You will need to capacity plan for this & ensure you can make the tradeoff of more real-time results for increased load.

Default - `BaseSignalProcessor`

The default setup is configured to use the `haystack.signals.BaseSignalProcessor` class, which includes all the underlying code necessary to handle individual updates/deletes, **BUT DOES NOT HOOK UP THE SIGNALS**.

This means that, by default, **NO ACTION IS TAKEN BY HAYSTACK** when a model is saved or deleted. The `BaseSignalProcessor.setup` & `BaseSignalProcessor.teardown` methods are both empty to prevent anything from being setup at initialization time.

This usage is configured very simply (again, by default) with the `HAYSTACK_SIGNAL_PROCESSOR` setting. An example of manually setting this would look like:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.BaseSignalProcessor'
```

This class forms an excellent base if you'd like to override/extend for more advanced behavior. Which leads us to...

Realtime - `RealtimeSignalProcessor`

The other included `SignalProcessor` is the `haystack.signals.RealtimeSignalProcessor` class. It is an extremely thin extension of the `BaseSignalProcessor` class, differing only in that in implements the `setup/teardown` methods, tying **ANY** Model `save/delete` to the signal processor.

If the model has an associated `SearchIndex`, the `RealtimeSignalProcessor` will then trigger an update/delete of that model instance within the search index proper.

Configuration looks like:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'
```

This causes **all** `SearchIndex` classes to work in a realtime fashion.

Note: These updates happen in-process, which if a request-response cycle is involved, may cause the user with the browser to sit & wait for indexing to be completed. Since this wait can be undesirable, especially under load, you may wish to look into queued search options. See the *Haystack-Related Applications* documentation for existing options.

Custom `SignalProcessors`

The `BaseSignalProcessor` & `RealtimeSignalProcessor` classes are fairly simple/straightforward to customize or extend. Rather than forking Haystack to implement your modifications, you should create your own subclass within your codebase (anywhere that's importable is usually fine, though you should avoid `models.py` files).

For instance, if you only wanted `User` saves to be realtime, deferring all other updates to the management commands, you'd implement the following code:

```
from django.contrib.auth.models import User
from django.db import models
from haystack import signals


class UserOnlySignalProcessor(signals.BaseSignalProcessor):
    def setup(self):
        # Listen only to the ``User`` model.
        models.signals.post_save.connect(self.handle_save, sender=User)
        models.signals.post_delete.connect(self.handle_delete, sender=User)

    def teardown(self):
        # Disconnect only for the ``User`` model.
        models.signals.post_save.disconnect(self.handle_save, sender=User)
        models.signals.post_delete.disconnect(self.handle_delete, sender=User)
```

For other customizations (modifying how saves/deletes should work), you'll need to override/extend the `handle_save/handle_delete` methods. The source code is your best option for referring to how things currently work on your version of Haystack.

Multiple Indexes

Much like Django's multiple database support, Haystack has "multiple index" support. This allows you to talk to several different engines at the same time. It enables things like master-slave setups, multiple language indexing, separate indexes for general search & autocomplete as well as other options.

Specifying Available Connections

You can supply as many backends as you like, each with a descriptive name. A complete setup that accesses all backends might look like:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://localhost:9001/solr/default',
        'TIMEOUT': 60 * 5,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'SILENTLY_FAIL': True,
    },
    'autocomplete': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': '/home/search/whoosh_index',
        'STORAGE': 'file',
        'POST_LIMIT': 128 * 1024 * 1024,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'SILENTLY_FAIL': True,
    },
    'slave': {
        'ENGINE': 'xapian_backend.XapianEngine',
        'PATH': '/home/search/xapian_index',
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'SILENTLY_FAIL': True,
    },
    'db': {
        'ENGINE': 'haystack.backends.simple_backend.SimpleEngine',
        'SILENTLY_FAIL': True,
    }
}
```

You are required to have at least one connection listed within `HAYSTACK_CONNECTIONS`, it must be named `default` & it must have a valid `ENGINE` within it.

Management Commands

All management commands that manipulate data use **ONLY** one connection at a time. By default, they use the `default` index but accept a `--using` flag to specify a different connection. For example:

```
./manage.py rebuild_index --noinput --using=whoosh
```

Automatic Routing

To make the selection of the correct index easier, Haystack (like Django) has the concept of "routers". All provided routers are checked whenever a read or write happens, in the order in which they are defined.

For read operations (when a search query is executed), the `for_read` method of each router is called, until one of them returns an index, which is used for the read operation.

For write operations (when a delete or update is executed), the `for_write` method of each router is called, and the results are aggregated. All of the indexes that were returned are then updated.

Haystack ships with a `DefaultRouter` enabled. It looks like:

```
class DefaultRouter(BaseRouter):
    def for_read(self, **hints):
        return DEFAULT_ALIAS

    def for_write(self, **hints):
        return DEFAULT_ALIAS
```

This means that the default index is used for all read and write operations.

If the `for_read` or `for_write` method doesn't exist or returns `None`, that indicates that the current router can't handle the data. The next router is then checked.

The `for_write` method can return either a single string representing an index name, or an iterable of such index names. For example:

```
class UpdateEverythingRouter(BaseRouter):
    def for_write(self, **hints):
        return ('myindex1', 'myindex2')
```

The `hints` passed can be anything that helps the router make a decision. This data should always be considered optional & be guarded against. At current, `for_write` receives an `index` option (pointing to the `SearchIndex` calling it) while `for_read` may receive `models` (being a list of `Model` classes the `SearchQuerySet` may be looking at).

You may provide as many routers as you like by overriding the `HAYSTACK_ROUTERS` setting. For example:

```
HAYSTACK_ROUTERS = ['myapp.routers.MasterRouter', 'myapp.routers.SlaveRouter',
↪'haystack.routers.DefaultRouter']
```

Master-Slave Example

The `MasterRouter` & `SlaveRouter` might look like:

```
from haystack import routers


class MasterRouter(routers.BaseRouter):
    def for_write(self, **hints):
        return 'master'

    def for_read(self, **hints):
        return None


class SlaveRouter(routers.BaseRouter):
```

```
    def for_write(self, **hints):
        return None

    def for_read(self, **hints):
        return 'slave'
```

The observant might notice that since the methods don't overlap, this could be combined into one `Router` like so:

```
from haystack import routers


class MasterSlaveRouter(routers.BaseRouter):
    def for_write(self, **hints):
        return 'master'

    def for_read(self, **hints):
        return 'slave'
```

Manually Selecting

There may be times when automatic selection of the correct index is undesirable, such as when fixing erroneous data in an index or when you know exactly where data should be located.

For this, the `SearchQuerySet` class allows for manually selecting the index via the `SearchQuerySet.using` method:

```
from haystack.query import SearchQuerySet

# Uses the routers' opinion.
sqs = SearchQuerySet().auto_query('banana')

# Forces the default.
sqs = SearchQuerySet().using('default').auto_query('banana')

# Forces the slave connection (presuming it was setup).
sqs = SearchQuerySet().using('slave').auto_query('banana')
```

Warning: Note that the models a `SearchQuerySet` is trying to pull from must all come from the same index. Haystack is not able to combine search queries against different indexes.

Custom Index Selection

If a specific backend has been selected, the `SearchIndex.index_queryset` and `SearchIndex.read_queryset` will receive the backend name, giving indexes the opportunity to customize the returned queryset.

For example, a site which uses separate indexes for recent items and older content might define `index_queryset` to filter the items based on date:

```
def index_queryset(self, using=None):
    qs = Note.objects.all()
    archive_limit = datetime.datetime.now() - datetime.timedelta(days=90)

    if using == "archive":
```

```
        return qs.filter(pub_date__lte=archive_limit)
    else:
        return qs.filter(pub_date__gte=archive_limit)
```

Multi-lingual Content

Most search engines require you to set the language at the index level. For example, a multi-lingual site using Solr can use multiple cores and corresponding Haystack backends using the language name. Under this scenario, queries are simple:

```
sqs = SearchQuerySet.using(lang).auto_query(...)
```

During index updates, the Index's `index_queryset` method will need to filter the items to avoid sending the wrong content to the search engine:

```
def index_queryset(self, using=None):
    return Post.objects.filter(language=using)
```

Rich Content Extraction

For some projects it is desirable to index text content which is stored in structured files such as PDFs, Microsoft Office documents, images, etc. Currently only Solr's ExtractingRequestHandler is directly supported by Haystack but the approach below could be used with any backend which supports this feature.

Extracting Content

`SearchBackend.extract_file_contents()` accepts a file or file-like object and returns a dictionary containing two keys: `metadata` and `contents`. The `contents` value will be a string containing all of the text which the backend managed to extract from the file contents. `metadata` will always be a dictionary but the keys and values will vary based on the underlying extraction engine and the type of file provided.

Indexing Extracted Content

Generally you will want to include the extracted text in your main document field along with everything else specified in your search template. This example shows how to override a hypothetical `FileIndex`'s `prepare` method to include the extract content along with information retrieved from the database:

```
def prepare(self, obj):
    data = super(FileIndex, self).prepare(obj)

    # This could also be a regular Python open() call, a StringIO instance
    # or the result of opening a URL. Note that due to a library limitation
    # file_obj must have a .name attribute even if you need to set one
    # manually before calling extract_file_contents:
    file_obj = obj.the_file.open()

    extracted_data = self.get_backend().extract_file_contents(file_obj)

    # Now we'll finally perform the template processing to render the
    # text field with *all* of our metadata visible for templating:
```

```
    t = loader.select_template(('search/indexes/myapp/file_text.txt', ))
    data['text'] = t.render(Context({'object': obj,
                                     'extracted': extracted_data}))

    return data
```

This allows you to insert the extracted text at the appropriate place in your template, modified or intermixed with database content as appropriate:

```
{{ object.title }}
{{ object.owner.name }}

...

{% for k, v in extracted.metadata.items %}
    {% for val in v %}
        {{ k }}: {{ val|safe }}
    {% endfor %}
{% endfor %}

{{ extracted.contents|striptags|safe }}
```

Spatial Search

Spatial search (also called geospatial search) allows you to take data that has a geographic location & enhance the search results by limiting them to a physical area. Haystack, combined with the latest versions of a couple engines, can provide this type of search.

In addition, Haystack tries to implement these features in a way that is as close to GeoDjango as possible. There are some differences, which we'll highlight throughout this guide. Additionally, while the support isn't as comprehensive as PostGIS (for example), it is still quite useful.

Additional Requirements

The spatial functionality has only one non-included, non-available-in-Django dependency:

- `geopy` - `pip install geopy`

If you do not ever need distance information, you may be able to skip installing `geopy`.

Support

You need the latest & greatest of either Solr or Elasticsearch. None of the other backends (specifially the engines) support this kind of search.

For Solr, you'll need at least **v3.5+**. In addition, if you have an existing install of Haystack & Solr, you'll need to upgrade the schema & reindex your data. If you're adding geospatial data, you would have to reindex anyhow.

For Elasticsearch, you'll need at least v0.17.7, preferably v0.18.6 or better. If you're adding geospatial data, you'll have to reindex as well.

Lookup Type	Solr	Elasticsearch	Whoosh	Xapian	Simple
within	X	X			
dwithin	X	X			
distance	X	X			
order_by('distance')	X	X			
polygon		X			

For more details, you can inspect http://wiki.apache.org/solr/SpatialSearch or http://www.elasticsearch.org/guide/reference/query-dsl/geo-bounding-box-filter.html.

Geospatial Assumptions

Points

Haystack prefers to work with `Point` objects, which are located in `django.contrib.gis.geos.Point` but conviently importable out of `haystack.utils.geo.Point`.

`Point` objects use **LONGITUDE, LATITUDE** for their construction, regardless if you use the parameters to instantiate them or WKT/`GEOSGeometry`.

Examples:

```
# Using positional arguments.
from haystack.utils.geo import Point
pnt = Point(-95.23592948913574, 38.97127105172941)

# Using WKT.
from django.contrib.gis.geos import GEOSGeometry
pnt = GEOSGeometry('POINT(-95.23592948913574 38.97127105172941)')
```

They are preferred over just providing `latitude, longitude` because they are more intelligent, have a spatial reference system attached & are more consistent with GeoDjango's use.

Distance

Haystack also uses the `D` (or `Distance`) objects from GeoDjango, implemented in `django.contrib.gis.measure.Distance` but conveniently importable out of `haystack.utils.geo.D` (or `haystack.utils.geo.Distance`).

`Distance` objects accept a very flexible set of measurements during instantiaton and can convert amongst them freely. This is important, because the engines rely on measurements being in kilometers but you're free to use whatever units you want.

Examples:

```
from haystack.utils.geo import D

# Start at 5 miles.
imperial_d = D(mi=5)

# Convert to fathoms...
fathom_d = imperial_d.fathom

# Now to kilometers...
km_d = imperial_d.km
```

```
# And back to miles.
mi = imperial_d.mi
```

They are preferred over just providing a raw distance because they are more intelligent, have a well-defined unit system attached & are consistent with GeoDjango's use.

WGS-84

All engines assume WGS-84 (SRID 4326). At the time of writing, there does **not** appear to be a way to switch this. Haystack will transform all points into this coordinate system for you.

Indexing

Indexing is relatively simple. Simply add a `LocationField` (or several) onto your `SearchIndex` class(es) & provide them a `Point` object. For example:

```
from haystack import indexes
from shops.models import Shop


class ShopIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    # ... the usual, then...
    location = indexes.LocationField(model_attr='coordinates')

    def get_model(self):
        return Shop
```

If you must manually prepare the data, you have to do something slightly less convenient, returning a string-ified version of the coordinates in WGS-84 as `lat,long`:

```
from haystack import indexes
from shops.models import Shop


class ShopIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    # ... the usual, then...
    location = indexes.LocationField()

    def get_model(self):
        return Shop

    def prepare_location(self, obj):
        # If you're just storing the floats...
        return "%s,%s" % (obj.latitude, obj.longitude)
```

Alternatively, you could build a method/property onto the `Shop` model that returns a `Point` based on those coordinates:

```
# shops/models.py
from django.contrib.gis.geos import Point
from django.db import models
```

```
class Shop(models.Model):
    # ... the usual, then...
    latitude = models.FloatField()
    longitude = models.FloatField()

    # Usual methods, then...
    def get_location(self):
        # Remember, longitude FIRST!
        return Point(self.longitude, self.latitude)


# shops/search_indexes.py
from haystack import indexes
from shops.models import Shop


class ShopIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    location = indexes.LocationField(model_attr='get_location')

    def get_model(self):
        return Shop
```

Querying

There are two types of geospatial queries you can run, `within` & `dwithin`. Like their GeoDjango counterparts (within & dwithin), these methods focus on finding results within an area.

within

`SearchQuerySet`.**`within`**(*self*, *field*, *point_1*, *point_2*)

`within` is a bounding box comparison. A bounding box is a rectangular area within which to search. It's composed of a bottom-left point & a top-right point. It is faster but slighty sloppier than its counterpart.

Examples:

```
from haystack.query import SearchQuerySet
from haystack.utils.geo import Point

downtown_bottom_left = Point(-95.23947, 38.9637903)
downtown_top_right = Point(-95.23362278938293, 38.973081081164715)

# 'location' is the fieldname from our ``SearchIndex``...

# Do the bounding box query.
sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right)

# Can be chained with other Haystack calls.
sqs = SearchQuerySet().auto_query('coffee').within('location', downtown_bottom_left,␣
↪downtown_top_right).order_by('-popularity')
```

Note: In GeoDjango, assuming the `Shop` model had been properly geo-ified, this would have been implemented as:

```
from shops.models import Shop
Shop.objects.filter(location__within=(downtown_bottom_left, downtown_top_right))
```

Haystack's form differs because it yielded a cleaner implementation, was no more typing than the GeoDjango version & tried to maintain the same terminology/similar signature.

---

dwithin

`SearchQuerySet.`**`dwithin`**(*self*, *field*, *point*, *distance*)

`dwithin` is a radius-based search. A radius-based search is a circular area within which to search. It's composed of a center point & a radius (in kilometers, though Haystack will use the `D` object's conversion utilities to get it there). It is slower than''within'' but very exact & can involve fewer calculations on your part.

Examples:

```
from haystack.query import SearchQuerySet
from haystack.utils.geo import Point, D

ninth_and_mass = Point(-95.23592948913574, 38.96753407043678)
# Within a two miles.
max_dist = D(mi=2)

# 'location' is the fieldname from our ``SearchIndex``...

# Do the radius query.
sqs = SearchQuerySet().dwithin('location', ninth_and_mass, max_dist)

# Can be chained with other Haystack calls.
sqs = SearchQuerySet().auto_query('coffee').dwithin('location', ninth_and_mass, max_
↪dist).order_by('-popularity')
```

Note: In GeoDjango, assuming the `Shop` model had been properly geo-ified, this would have been implemented as:

```
from shops.models import Shop
Shop.objects.filter(location__dwithin=(ninth_and_mass, D(mi=2)))
```

Haystack's form differs because it yielded a cleaner implementation, was no more typing than the GeoDjango version & tried to maintain the same terminology/similar signature.

---

distance

`SearchQuerySet.`**`distance`**(*self*, *field*, *point*)

By default, search results will come back without distance information attached to them. In the concept of a bounding box, it would be ambiguous what the distances would be calculated against. And it is more calculation that may not be necessary.

So like GeoDjango, Haystack exposes a method to signify that you want to include these calculated distances on results.

Examples:

```
from haystack.query import SearchQuerySet
from haystack.utils.geo import Point, D

ninth_and_mass = Point(-95.23592948913574, 38.96753407043678)

# On a bounding box...
downtown_bottom_left = Point(-95.23947, 38.9637903)
downtown_top_right = Point(-95.23362278938293, 38.973081081164715)

sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right).
↪distance('location', ninth_and_mass)

# ...Or on a radius query.
sqs = SearchQuerySet().dwithin('location', ninth_and_mass, D(mi=2)).distance('location
↪', ninth_and_mass)
```

You can even apply a different field, for instance if you calculate results of key, well-cached hotspots in town but want distances from the user's current position:

```
from haystack.query import SearchQuerySet
from haystack.utils.geo import Point, D

ninth_and_mass = Point(-95.23592948913574, 38.96753407043678)
user_loc = Point(-95.23455619812012, 38.97240128290697)

sqs = SearchQuerySet().dwithin('location', ninth_and_mass, D(mi=2)).distance('location
↪', user_loc)
```

Note: The astute will notice this is Haystack's biggest departure from GeoDjango. In GeoDjango, this would have been implemented as:

```
from shops.models import Shop
Shop.objects.filter(location__dwithin=(ninth_and_mass, D(mi=2))).distance(user_loc)
```

Note that, by default, the GeoDjango form leaves *out* the field to be calculating against (though it's possible to override it & specify the field).

Haystack's form differs because the same assumptions are difficult to make. GeoDjango deals with a single model at a time, where Haystack deals with a broad mix of models. Additionally, accessing `Model` information is a couple hops away, so Haystack favors the explicit (if slightly more typing) approach.

Ordering

Because you're dealing with search, even with geospatial queries, results still come back in **RELEVANCE** order. If you want to offer the user ordering results by distance, there's a simple way to enable this ordering.

Using the standard Haystack `order_by` method, if you specify `distance` or `-distance` **ONLY**, you'll get geographic ordering. Additionally, you must have a call to `.distance()` somewhere in the chain, otherwise there is no distance information on the results & nothing to sort by.

Examples:

```
from haystack.query import SearchQuerySet
from haystack.utils.geo import Point, D
```

```
ninth_and_mass = Point(-95.23592948913574, 38.96753407043678)
downtown_bottom_left = Point(-95.23947, 38.9637903)
downtown_top_right = Point(-95.23362278938293, 38.973081081164715)

# Non-geo ordering.
sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right).
↪order_by('title')
sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right).
↪distance('location', ninth_and_mass).order_by('-created')

# Geo ordering, closest to farthest.
sqs = SearchQuerySet().within('location', downtown_bottom_left, downtown_top_right).
↪distance('location', ninth_and_mass).order_by('distance')
# Geo ordering, farthest to closest.
sqs = SearchQuerySet().dwithin('location', ninth_and_mass, D(mi=2)).distance('location
↪', ninth_and_mass).order_by('-distance')
```

Note: This call is identical to the GeoDjango usage.

Warning: You can not specify both a distance & lexicographic ordering. If you specify more than just `distance` or `-distance`, Haystack assumes `distance` is a field in the index & tries to sort on it. Example:

```
# May blow up!
sqs = SearchQuerySet().dwithin('location', ninth_and_mass, D(mi=2)).distance(
↪'location', ninth_and_mass).order_by('distance', 'title')
```

This is a limitation in the engine's implementation.

If you actually **have** a field called `distance` (& aren't using calculated distance information), Haystack will do the right thing in these circumstances.

Caveats

In all cases, you may call the `within/dwithin/distance` methods as many times as you like. However, the **LAST** call is the information that will be used. No combination logic is available, as this is largely a backend limitation.

Combining calls to both `within` & `dwithin` may yield unexpected or broken results. They don't overlap when performing queries, so it may be possible to construct queries that work. Your Mileage May Vary.

Django Admin Search

Haystack comes with a base class to support searching via Haystack in the Django admin. To use Haystack to search, inherit from `haystack.admin.SearchModelAdmin` instead of `django.contrib.admin.ModelAdmin`.

For example:

```
from haystack.admin import SearchModelAdmin
from .models import MockModel


class MockModelAdmin(SearchModelAdmin):
    haystack_connection = 'solr'
```

```
    date_hierarchy = 'pub_date'
    list_display = ('author', 'pub_date')


admin.site.register(MockModel, MockModelAdmin)
```

You can also specify the Haystack connection used by the search with the `haystack_connection` property on the model admin class. If not specified, the default connection will be used.

If you already have a base model admin class you use, there is also a mixin you can use instead:

```
from django.contrib import admin
from haystack.admin import SearchModelAdminMixin
from .models import MockModel


class MyCustomModelAdmin(admin.ModelAdmin):
    pass


class MockModelAdmin(SearchModelAdminMixin, MyCustomModelAdmin):
    haystack_connection = 'solr'
    date_hierarchy = 'pub_date'
    list_display = ('author', 'pub_date')


admin.site.register(MockModel, MockModelAdmin)
```

CHAPTER 3

Reference

If you're an experienced user and are looking for a reference, you may be looking for API documentation and advanced usage as detailed in:

`SearchQuerySet` API

class `SearchQuerySet` (*using=None*, *query=None*)

The `SearchQuerySet` class is designed to make performing a search and iterating over its results easy and consistent. For those familiar with Django's ORM `QuerySet`, much of the `SearchQuerySet` API should feel familiar.

Why Follow `QuerySet`?

A couple reasons to follow (at least in part) the `QuerySet` API:

1. Consistency with Django
2. Most Django programmers have experience with the ORM and can use this knowledge with `SearchQuerySet`.

And from a high-level perspective, `QuerySet` and `SearchQuerySet` do very similar things: given certain criteria, provide a set of results. Both are powered by multiple backends, both are abstractions on top of the way a query is performed.

Quick Start

For the impatient:

```
from haystack.query import SearchQuerySet
all_results = SearchQuerySet().all()
hello_results = SearchQuerySet().filter(content='hello')
hello_world_results = SearchQuerySet().filter(content='hello world')
```

```
unfriendly_results = SearchQuerySet().exclude(content='hello').filter(content='world')
recent_results = SearchQuerySet().order_by('-pub_date')[:5]

# Using the new input types...
from haystack.inputs import AutoQuery, Exact, Clean
sqs = SearchQuerySet().filter(content=AutoQuery(request.GET['q']), product_type=Exact(
↪'ancient book'))

if request.GET['product_url']:
    sqs = sqs.filter(product_url=Clean(request.GET['product_url']))
```

For more on the `AutoQuery`, `Exact`, `Clean` classes & friends, see the *Input Types* documentation.

SearchQuerySet

By default, `SearchQuerySet` provide the documented functionality. You can extend with your own behavior by simply subclassing from `SearchQuerySet` and adding what you need, then using your subclass in place of `SearchQuerySet`.

Most methods in `SearchQuerySet` "chain" in a similar fashion to `QuerySet`. Additionally, like `QuerySet`, `SearchQuerySet` is lazy (meaning it evaluates the query as late as possible). So the following is valid:

```
from haystack.query import SearchQuerySet
results = SearchQuerySet().exclude(content='hello').filter(content='world').order_by(
↪'-pub_date').boost('title', 0.5)[10:20]
```

The `content` Shortcut

Searching your document fields is a very common activity. To help mitigate possible differences in `SearchField` names (and to help the backends deal with search queries that inspect the main corpus), there is a special field called `content`. You may use this in any place that other fields names would work (e.g. `filter`, `exclude`, etc.) to indicate you simply want to search the main documents.

For example:

```
from haystack.query import SearchQuerySet

# This searches whatever fields were marked ``document=True``.
results = SearchQuerySet().exclude(content='hello')
```

This special pseudo-field works best with the `exact` lookup and may yield strange or unexpected results with the other lookups.

SearchQuerySet Methods

The primary interface to search in Haystack is through the `SearchQuerySet` object. It provides a clean, programmatic, portable API to the search backend. Many aspects are also "chainable", meaning you can call methods one after another, each applying their changes to the previous `SearchQuerySet` and further narrowing the search.

All `SearchQuerySet` objects implement a list-like interface, meaning you can perform actions like getting the length of the results, accessing a result at an offset or even slicing the result list.

Methods That Return A `SearchQuerySet`

`all`

`SearchQuerySet.all(self)`:

Returns all results for the query. This is largely a no-op (returns an identical copy) but useful for denoting exactly what behavior is going on.

`none`

`SearchQuerySet.none(self)`:

Returns an `EmptySearchQuerySet` that behaves like a `SearchQuerySet` but always yields no results.

`filter`

`SearchQuerySet.`**`filter`**(*self*, ***kwargs*)

Filters the search by looking for (and including) certain attributes.

The lookup parameters (`**kwargs`) should follow the *Field lookups* below. If you specify more than one pair, they will be joined in the query according to the `HAYSTACK_DEFAULT_OPERATOR` setting (defaults to `AND`).

You can pass it either strings or a variety of *Input Types* if you need more advanced query behavior.

> **Warning:** Any data you pass to `filter` gets auto-escaped. If you need to send non-escaped data, use the `Raw` input type (*Input Types*).
>
> Also, if a string with one or more spaces in it is specified as the value, the string will get passed along **AS IS**. This will mean that it will **NOT** be treated as a phrase (like Haystack 1.X's behavior).
>
> If you want to match a phrase, you should use either the `__exact` filter type or the `Exact` input type (*Input Types*).

Examples:

```
sqs = SearchQuerySet().filter(content='foo')

sqs = SearchQuerySet().filter(content='foo', pub_date__lte=datetime.date(2008, 1, 1))

# Identical to the previous example.
sqs = SearchQuerySet().filter(content='foo').filter(pub_date__lte=datetime.date(2008,␣
↪1, 1))

# To send unescaped data:
from haystack.inputs import Raw
sqs = SearchQuerySet().filter(title=Raw(trusted_query))

# To use auto-query behavior on a non-``document=True`` field.
from haystack.inputs import AutoQuery
sqs = SearchQuerySet().filter(title=AutoQuery(user_query))
```

exclude

SearchQuerySet.**exclude**(*self*, ***kwargs*)

Narrows the search by ensuring certain attributes are not included.

> **Warning:** Any data you pass to `exclude` gets auto-escaped. If you need to send non-escaped data, use the `Raw` input type (*Input Types*).

Example:

```
sqs = SearchQuerySet().exclude(content='foo')
```

filter_and

SearchQuerySet.**filter_and**(*self*, ***kwargs*)

Narrows the search by looking for (and including) certain attributes. Join behavior in the query is forced to be `AND`. Used primarily by the `filter` method.

filter_or

SearchQuerySet.**filter_or**(*self*, ***kwargs*)

Narrows the search by looking for (and including) certain attributes. Join behavior in the query is forced to be `OR`. Used primarily by the `filter` method.

order_by

SearchQuerySet.**order_by**(*self*, **args*)

Alters the order in which the results should appear. Arguments should be strings that map to the attributes/fields within the index. You may specify multiple fields by comma separating them:

```
SearchQuerySet().filter(content='foo').order_by('author', 'pub_date')
```

Default behavior is ascending order. To specify descending order, prepend the string with a `-`:

```
SearchQuerySet().filter(content='foo').order_by('-pub_date')
```

Note: In general, ordering is locale-specific. Haystack makes no effort to try to reconcile differences between characters from different languages. This means that accented characters will sort closely with the same character and **NOT** necessarily close to the unaccented form of the character.

If you want this kind of behavior, you should override the `prepare_FOO` methods on your `SearchIndex` objects to transliterate the characters as you see fit.

---

highlight

SearchQuerySet.**highlight**(*self*)

If supported by the backend, the SearchResult objects returned will include a highlighted version of the result:

```
sqs = SearchQuerySet().filter(content='foo').highlight()
result = sqs[0]
result.highlighted['text'][0] # u'Two computer scientists walk into a bar. The␣
↪bartender says "<em>Foo</em>!".'
```

The default functionality of the highlighter may not suit your needs. You can pass additional keyword arguments to highlight that will ultimately be used to build the query for your backend. Depending on the available arguments for your backend, you may need to pass in a dictionary instead of normal keyword arguments:

```
# Solr defines the fields to higlight by the ``hl.fl`` param. If not specified, we
# would only get `text` back in the ``highlighted`` dict.
kwargs = {
    'hl.fl': 'other_field',
    'hl.simple.pre': '<span class="highlighted">',
    'hl.simple.post': '</span>'
}
sqs = SearchQuerySet().filter(content='foo').highlight(**kwargs)
result = sqs[0]
result.highlighted['other_field'][0] # u'Two computer scientists walk into a bar. The␣
↪bartender says "<span class="highlighted">Foo</span>!".'
```

models

SearchQuerySet.**models**(*self*, **models*)

Accepts an arbitrary number of Model classes to include in the search. This will narrow the search results to only include results from the models specified.

Example:

```
SearchQuerySet().filter(content='foo').models(BlogEntry, Comment)
```

result_class

SearchQuerySet.**result_class**(*self*, *klass*)

Allows specifying a different class to use for results.

Overrides any previous usages. If None is provided, Haystack will revert back to the default SearchResult object.

Example:

```
SearchQuerySet().result_class(CustomResult)
```

boost

SearchQuerySet.**boost**(*self*, *term*, *boost_value*)

Boosts a certain term of the query. You provide the term to be boosted and the value is the amount to boost it by. Boost amounts may be either an integer or a float.

Example:

```
SearchQuerySet().filter(content='foo').boost('bar', 1.5)
```

facet

SearchQuerySet.**facet**(*self*, *field*, ***options*)

Adds faceting to a query for the provided field. You provide the field (from one of the SearchIndex classes) you like to facet on. Any keyword options you provide will be passed along to the backend for that facet.

Example:

```
# For SOLR (setting f.author.facet.*; see http://wiki.apache.org/solr/
↪SimpleFacetParameters#Parameters)
SearchQuerySet().facet('author', mincount=1, limit=10)
# For Elasticsearch (see http://www.elasticsearch.org/guide/reference/api/search/
↪facets/terms-facet.html)
SearchQuerySet().facet('author', size=10, order='term')
```

In the search results you get back, facet counts will be populated in the SearchResult object. You can access them via the facet_counts method.

Example:

```
# Count document hits for each author within the index.
SearchQuerySet().filter(content='foo').facet('author')
```

date_facet

SearchQuerySet.**date_facet**(*self*, *field*, *start_date*, *end_date*, *gap_by*, *gap_amount=1*)

Adds faceting to a query for the provided field by date. You provide the field (from one of the SearchIndex classes) you like to facet on, a start_date (either datetime.datetime or datetime.date), an end_date and the amount of time between gaps as gap_by (one of 'year', 'month', 'day', 'hour', 'minute' or 'second').

You can also optionally provide a gap_amount to specify a different increment than 1. For example, specifying gaps by week (every seven days) would be gap_by='day', gap_amount=7).

In the search results you get back, facet counts will be populated in the SearchResult object. You can access them via the facet_counts method.

Example:

```
# Count document hits for each day between 2009-06-07 to 2009-07-07 within the index.
SearchQuerySet().filter(content='foo').date_facet('pub_date', start_date=datetime.
↪date(2009, 6, 7), end_date=datetime.date(2009, 7, 7), gap_by='day')
```

query_facet

SearchQuerySet.**query_facet**(*self*, *field*, *query*)

Adds faceting to a query for the provided field with a custom query. You provide the field (from one of the `SearchIndex` classes) you like to facet on and the backend-specific query (as a string) you'd like to execute.

Please note that this is **NOT** portable between backends. The syntax is entirely dependent on the backend. No validation/cleansing is performed and it is up to the developer to ensure the query's syntax is correct.

In the search results you get back, facet counts will be populated in the `SearchResult` object. You can access them via the `facet_counts` method.

Example:

```
# Count document hits for authors that start with 'jo' within the index.
SearchQuerySet().filter(content='foo').query_facet('author', 'jo*')
```

within

SearchQuerySet.within(self, field, point_1, point_2):

Spatial: Adds a bounding box search to the query.

See the *Spatial Search* docs for more information.

dwithin

SearchQuerySet.dwithin(self, field, point, distance):

Spatial: Adds a distance-based search to the query.

See the *Spatial Search* docs for more information.

stats

SearchQuerySet.stats(self, field):

Adds stats to a query for the provided field. This is supported on Solr only. You provide the field (from one of the `SearchIndex` classes) you would like stats on.

In the search results you get back, stats will be populated in the `SearchResult` object. You can access them via the '' stats_results'' method.

Example:

```
# Get stats on the author field.
SearchQuerySet().filter(content='foo').stats('author')
```

stats_facet

SearchQuerySet.stats_facet(self, field,

Adds stats facet for the given field and facet_fields represents the faceted fields. This is supported on Solr only.

Example:

```
# Get stats on the author field, and stats on the author field
faceted by bookstore.
SearchQuerySet().filter(content='foo').stats_facet('author','bookstore')
```

distance

SearchQuerySet.distance(self, field, point):

Spatial: Denotes results must have distance measurements from the provided point.

See the *Spatial Search* docs for more information.

narrow

`SearchQuerySet.`**narrow**(*self*, *query*)

Pulls a subset of documents from the search engine to search within. This is for advanced usage, especially useful when faceting.

Example:

```
# Search, from recipes containing 'blend', for recipes containing 'banana'.
SearchQuerySet().narrow('blend').filter(content='banana')

# Using a fielded search where the recipe's title contains 'smoothie', find all␣
↪recipes published before 2009.
SearchQuerySet().narrow('title:smoothie').filter(pub_date__lte=datetime.datetime(2009,
↪ 1, 1))
```

By using `narrow`, you can create drill-down interfaces for faceting by applying `narrow` calls for each facet that gets selected.

This method is different from `SearchQuerySet.filter()` in that it does not affect the query sent to the engine. It pre-limits the document set being searched. Generally speaking, if you're in doubt of whether to use `filter` or `narrow`, use `filter`.

Note: This method is, generally speaking, not necessarily portable between backends. The syntax is entirely dependent on the backend, though most backends have a similar syntax for basic fielded queries. No validation/cleansing is performed and it is up to the developer to ensure the query's syntax is correct.

---

raw_search

`SearchQuerySet.`**raw_search**(*self*, *query_string*, ***kwargs*)

Passes a raw query directly to the backend. This is for advanced usage, where the desired query can not be expressed via `SearchQuerySet`.

This method is still supported, however it now uses the much more flexible `Raw` input type (*Input Types*).

Warning: Different from Haystack 1.X, this method no longer causes immediate evaluation & now chains appropriately.

Example:

```
# In the case of Solr... (this example could be expressed with SearchQuerySet)
SearchQuerySet().raw_search('django_ct:blog.blogentry "However, it is"')
```

```
# Equivalent.
from haystack.inputs import Raw
sqs = SearchQuerySet().filter(content=Raw('django_ct:blog.blogentry "However, it is"
↪'))
```

Please note that this is **NOT** portable between backends. The syntax is entirely dependent on the backend. No validation/cleansing is performed and it is up to the developer to ensure the query's syntax is correct.

Further, the use of `**kwargs` are completely undocumented intentionally. If a third-party backend can implement special features beyond what's present, it should use those `**kwargs` for passing that information. Developers should be careful to make sure there are no conflicts with the backend's `search` method, as that is called directly.

load_all

`SearchQuerySet.`**`load_all`**(*self*)

Efficiently populates the objects in the search results. Without using this method, DB lookups are done on a per-object basis, resulting in many individual trips to the database. If `load_all` is used, the `SearchQuerySet` will group similar objects into a single query, resulting in only as many queries as there are different object types returned.

Example:

```
SearchQuerySet().filter(content='foo').load_all()
```

auto_query

`SearchQuerySet.`**`auto_query`**(*self*, *query_string*, *fieldname=None*)

Performs a best guess constructing the search query.

This method is intended for common use directly with a user's query. This method is still supported, however it now uses the much more flexible `AutoQuery` input type (*Input Types*).

It handles exact matches (specified with single or double quotes), negation (using a - immediately before the term) and joining remaining terms with the operator specified in `HAYSTACK_DEFAULT_OPERATOR`.

Example:

```
sqs = SearchQuerySet().auto_query('goldfish "old one eye" -tank')

# Equivalent.
from haystack.inputs import AutoQuery
sqs = SearchQuerySet().filter(content=AutoQuery('goldfish "old one eye" -tank'))

# Against a different field.
sqs = SearchQuerySet().filter(title=AutoQuery('goldfish "old one eye" -tank'))
```

autocomplete

A shortcut method to perform an autocomplete search.

Must be run against fields that are either `NgramField` or `EdgeNgramField`.

Example:

```
SearchQuerySet().autocomplete(title_autocomplete='gol')
```

more_like_this

SearchQuerySet.**more_like_this**(*self*, *model_instance*)

Finds similar results to the object passed in.

You should pass in an instance of a model (for example, one fetched via a `get` in Django's ORM). This will execute a query on the backend that searches for similar results. The instance you pass in should be an indexed object. Previously called methods will have an effect on the provided results.

It will evaluate its own backend-specific query and populate the `SearchQuerySet` in the same manner as other methods.

Example:

```
entry = Entry.objects.get(slug='haystack-one-oh-released')
mlt = SearchQuerySet().more_like_this(entry)
mlt.count() # 5
mlt[0].object.title # "Haystack Beta 1 Released"

# ...or...
mlt = SearchQuerySet().filter(public=True).exclude(pub_date__lte=datetime.date(2009,␣
↪7, 21)).more_like_this(entry)
mlt.count() # 2
mlt[0].object.title # "Haystack Beta 1 Released"
```

using

SearchQuerySet.**using**(*self*, *connection_name*)

Allows switching which connection the `SearchQuerySet` uses to search in.

Example:

```
# Let the routers decide which connection to use.
sqs = SearchQuerySet().all()

# Specify the 'default'.
sqs = SearchQuerySet().all().using('default')
```

Methods That Do Not Return A `SearchQuerySet`

count

SearchQuerySet.**count**(*self*)

Returns the total number of matching results.

This returns an integer count of the total number of results the search backend found that matched. This method causes the query to evaluate and run the search.

Example:

```
SearchQuerySet().filter(content='foo').count()
```

best_match

SearchQuerySet.**best_match**(*self*)

Returns the best/top search result that matches the query.

This method causes the query to evaluate and run the search. This method returns a `SearchResult` object that is the best match the search backend found:

```
foo = SearchQuerySet().filter(content='foo').best_match()
foo.id # Something like 5.

# Identical to:
foo = SearchQuerySet().filter(content='foo')[0]
```

latest

SearchQuerySet.**latest**(*self*, *date_field*)

Returns the most recent search result that matches the query.

This method causes the query to evaluate and run the search. This method returns a `SearchResult` object that is the most recent match the search backend found:

```
foo = SearchQuerySet().filter(content='foo').latest('pub_date')
foo.id # Something like 3.

# Identical to:
foo = SearchQuerySet().filter(content='foo').order_by('-pub_date')[0]
```

facet_counts

SearchQuerySet.**facet_counts**(*self*)

Returns the facet counts found by the query. This will cause the query to execute and should generally be used when presenting the data (template-level).

You receive back a dictionary with three keys: `fields`, `dates` and `queries`. Each contains the facet counts for whatever facets you specified within your `SearchQuerySet`.

Note: The resulting dictionary may change before 1.0 release. It's fairly backend-specific at the time of writing. Standardizing is waiting on implementing other backends that support faceting and ensuring that the results presented will meet their needs as well.

Example:

```
# Count document hits for each author.
sqs = SearchQuerySet().filter(content='foo').facet('author')

sqs.facet_counts()
# Gives the following response:
```

```
# {
#     'dates': {},
#     'fields': {
#         'author': [
#             ('john', 4),
#             ('daniel', 2),
#             ('sally', 1),
#             ('terry', 1),
#         ],
#     },
#     'queries': {}
# }
```

stats_results

SearchQuerySet.stats_results(self):

Returns the stats results found by the query.

This will cause the query to execute and should generally be used when presenting the data (template-level).

You receive back a dictionary with three keys: `fields`, `dates` and `queries`. Each contains the facet counts for whatever facets you specified within your `SearchQuerySet`.

Note: The resulting dictionary may change before 1.0 release. It's fairly backend-specific at the time of writing. Standardizing is waiting on implementing other backends that support faceting and ensuring that the results presented will meet their needs as well.

Example:

```
# Count document hits for each author.
sqs = SearchQuerySet().filter(content='foo').stats('price')

sqs.stats_results()

# Gives the following response
# {
#    'stats_fields':{
#       'author:{
#            'min': 0.0,
#            'max': 2199.0,
#            'sum': 5251.2699999999995,
#            'count': 15,
#            'missing': 11,
#            'sumOfSquares': 6038619.160300001,
#            'mean': 350.08466666666664,
#            'stddev': 547.737557906113
#        }
#    }
#
# }
```

set_spelling_query

SearchQuerySet.**set_spelling_query**(*self*, *spelling_query*)

This method allows you to set the text which will be passed to the backend search engine for spelling suggestions. This is helpful when the actual query being sent to the backend has complex syntax which should not be seen by the spelling suggestion component.

In this example, a Solr `edismax` query is being used to boost field and document weights and `set_spelling_query` is being used to send only the actual user-entered text to the spellchecker:

```
alt_q = AltParser('edismax', self.query,
                  qf='title^4 text provider^0.5',
                  bq='django_ct:core.item^6.0')
sqs = sqs.filter(content=alt_q)
sqs = sqs.set_spelling_query(self.query)
```

spelling_suggestion

SearchQuerySet.**spelling_suggestion**(*self*, *preferred_query=None*)

Returns the spelling suggestion found by the query.

To work, you must set `INCLUDE_SPELLING` within your connection's settings dictionary to `True`, and you must rebuild your index afterwards. Otherwise, `None` will be returned.

This method causes the query to evaluate and run the search if it hasn't already run. Search results will be populated as normal but with an additional spelling suggestion. Note that this does *NOT* run the revised query, only suggests improvements.

If provided, the optional argument to this method lets you specify an alternate query for the spelling suggestion to be run on. This is useful for passing along a raw user-provided query, especially when there are many methods chained on the `SearchQuerySet`.

Example:

```
sqs = SearchQuerySet().auto_query('mor exmples')
sqs.spelling_suggestion() # u'more examples'

# ...or...
suggestion = SearchQuerySet().spelling_suggestion('moar exmples')
suggestion # u'more examples'
```

values

SearchQuerySet.**values**(*self*, **fields*)

Returns a list of dictionaries, each containing the key/value pairs for the result, exactly like Django's `ValuesQuerySet`.

This method causes the query to evaluate and run the search if it hasn't already run.

You must provide a list of one or more fields as arguments. These fields will be the ones included in the individual results.

Example:

```
sqs = SearchQuerySet().auto_query('banana').values('title', 'description')
```

values_list

SearchQuerySet.**values_list**(*self*, **fields*, ***kwargs*)

Returns a list of field values as tuples, exactly like Django's `ValuesListQuerySet`.

This method causes the query to evaluate and run the search if it hasn't already run.

You must provide a list of one or more fields as arguments. These fields will be the ones included in the individual results.

You may optionally also provide a `flat=True` kwarg, which in the case of a single field being provided, will return a flat list of that field rather than a list of tuples.

Example:

```
sqs = SearchQuerySet().auto_query('banana').values_list('title', 'description')

# ...or just the titles as a flat list...
sqs = SearchQuerySet().auto_query('banana').values_list('title', flat=True)
```

Field Lookups

The following lookup types are supported:

- content
- contains
- exact
- gt
- gte
- lt
- lte
- in
- startswith
- endswith
- range
- fuzzy

Except for `fuzzy` these options are similar in function to the way Django's lookup types work. The actual behavior of these lookups is backend-specific.

Warning: The `startswith` filter is strongly affected by the other ways the engine parses data, especially in regards to stemming (see *Glossary*). This can mean that if the query ends in a vowel or a plural form, it may get stemmed before being evaluated.

This is both backend-specific and yet fairly consistent between engines, and may be the cause of sometimes unexpected results.

Warning: The `content` filter became the new default filter as of Haystack v2.X (the default in Haystack v1.X was `exact`). This changed because `exact` caused problems and was unintuitive for new people trying to use Haystack. `content` is a much more natural usage.

If you had an app built on Haystack v1.X & are upgrading, you'll need to sanity-check & possibly change any code that was relying on the default. The solution is just to add `__exact` to any "bare" field in a `.filter(...)` clause.

Example:

```
SearchQuerySet().filter(content='foo')

# Identical to:
SearchQuerySet().filter(content__content='foo')

# Phrase matching.
SearchQuerySet().filter(content__exact='hello world')

# Other usages look like:
SearchQuerySet().filter(pub_date__gte=datetime.date(2008, 1, 1), pub_date__
↪lt=datetime.date(2009, 1, 1))
SearchQuerySet().filter(author__in=['daniel', 'john', 'jane'])
SearchQuerySet().filter(view_count__range=[3, 5])
```

EmptySearchQuerySet

Also included in Haystack is an `EmptySearchQuerySet` class. It behaves just like `SearchQuerySet` but will always return zero results. This is useful for places where you want no query to occur or results to be returned.

RelatedSearchQuerySet

Sometimes you need to filter results based on relations in the database that are not present in the search index or are difficult to express that way. To this end, `RelatedSearchQuerySet` allows you to post-process the search results by calling `load_all_queryset`.

Warning: `RelatedSearchQuerySet` can have negative performance implications. Because results are excluded based on the database after the search query has been run, you can't guarantee offsets within the cache. Therefore, the entire cache that appears before the offset you request must be filled in order to produce consistent results. On large result sets and at higher slices, this can take time.

This is the old behavior of `SearchQuerySet`, so performance is no worse than the early days of Haystack.

It supports all other methods that the standard `SearchQuerySet` does, with the addition of the `load_all_queryset` method and paying attention to the `load_all_queryset` method of `SearchIndex` objects when populating the cache.

load_all_queryset

RelatedSearchQuerySet.**load_all_queryset**(*self*, *model_class*, *queryset*)

Allows for specifying a custom QuerySet that changes how load_all will fetch records for the provided model. This is useful for post-processing the results from the query, enabling things like adding select_related or filtering certain data.

Example:

```
sqs = RelatedSearchQuerySet().filter(content='foo').load_all()
# For the Entry model, we want to include related models directly associated
# with the Entry to save on DB queries.
sqs = sqs.load_all_queryset(Entry, Entry.objects.all().select_related(depth=1))
```

This method chains indefinitely, so you can specify QuerySets for as many models as you wish, one per model. The SearchQuerySet appends on a call to in_bulk, so be sure that the QuerySet you provide can accommodate this and that the ids passed to in_bulk will map to the model in question.

If you need to do this frequently and have one QuerySet you'd like to apply everywhere, you can specify this at the SearchIndex level using the load_all_queryset method. See *SearchIndex API* for usage.

SearchIndex API

class SearchIndex

The SearchIndex class allows the application developer a way to provide data to the backend in a structured format. Developers familiar with Django's Form or Model classes should find the syntax for indexes familiar.

This class is arguably the most important part of integrating Haystack into your application, as it has a large impact on the quality of the search results and how easy it is for users to find what they're looking for. Care and effort should be put into making your indexes the best they can be.

Quick Start

For the impatient:

```
import datetime
from haystack import indexes
from myapp.models import Note


class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def index_queryset(self, using=None):
        "Used when the entire index for model is updated."
        return self.get_model().objects.filter(pub_date__lte=datetime.datetime.now())
```

Background

Unlike relational databases, most search engines supported by Haystack are primarily document-based. They focus on a single text blob which they tokenize, analyze and index. When searching, this field is usually the primary one that is searched.

Further, the schema used by most engines is the same for all types of data added, unlike a relational database that has a table schema for each chunk of data.

It may be helpful to think of your search index as something closer to a key-value store instead of imagining it in terms of a RDBMS.

Why Create Fields?

Despite being primarily document-driven, most search engines also support the ability to associate other relevant data with the indexed document. These attributes can be mapped through the use of fields within Haystack.

Common uses include storing pertinent data information, categorizations of the document, author information and related data. By adding fields for these pieces of data, you provide a means to further narrow/filter search terms. This can be useful from either a UI perspective (a better advanced search form) or from a developer standpoint (section-dependent search, off-loading certain tasks to search, et cetera).

Warning: Haystack reserves the following field names for internal use: `id`, `django_ct`, `django_id` & `content`. The `name` & `type` names used to be reserved but no longer are.

You can override these field names using the `HAYSTACK_ID_FIELD`, `HAYSTACK_DJANGO_CT_FIELD` & `HAYSTACK_DJANGO_ID_FIELD` if needed.

Significance Of `document=True`

Most search engines that were candidates for inclusion in Haystack all had a central concept of a document that they indexed. These documents form a corpus within which to primarily search. Because this ideal is so central and most of Haystack is designed to have pluggable backends, it is important to ensure that all engines have at least a bare minimum of the data they need to function.

As a result, when creating a `SearchIndex`, one (and only one) field must be marked with `document=True`. This signifies to Haystack that whatever is placed in this field while indexing is to be the primary text the search engine indexes. The name of this field can be almost anything, but `text` is one of the more common names used.

Stored/Indexed Fields

One shortcoming of the use of search is that you rarely have all or the most up-to-date information about an object in the index. As a result, when retrieving search results, you will likely have to access the object in the database to provide better information.

However, this can also hit the database quite heavily (think `.get(pk=result.id)` per object). If your search is popular, this can lead to a big performance hit. There are two ways to prevent this. The first way is `SearchQuerySet.load_all`, which tries to group all similar objects and pull them through one query instead of many. This still hits the DB and incurs a performance penalty.

The other option is to leverage stored fields. By default, all fields in Haystack are either indexed (searchable by the engine) or stored (retained by the engine and presented in the results). By using a stored field, you can store commonly used data in such a way that you don't need to hit the database when processing the search result to get more information.

For example, one great way to leverage this is to pre-render an object's search result template DURING indexing. You define an additional field, render a template with it and it follows the main indexed record into the index. Then, when that record is pulled when it matches a query, you can simply display the contents of that field, which avoids the database hit.:

Within `myapp/search_indexes.py`:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')
    # Define the additional field.
    rendered = CharField(use_template=True, indexed=False)
```

Then, inside a template named `search/indexes/myapp/note_rendered.txt`:

```
<h2>{{ object.title }}</h2>

<p>{{ object.content }}</p>
```

And finally, in `search/search.html`:

```
...

{% for result in page.object_list %}
    <div class="search_result">
        {{ result.rendered|safe }}
    </div>
{% endfor %}
```

Keeping The Index Fresh

There are several approaches to keeping the search index in sync with your database. None are more correct than the others and which one you use depends on the traffic you see, the churn rate of your data, and what concerns are important to you (CPU load, how recent, et cetera).

The conventional method is to use `SearchIndex` in combination with cron jobs. Running a `./manage.py update_index` every couple hours will keep your data in sync within that timeframe and will handle the updates in a very efficient batch. Additionally, Whoosh (and to a lesser extent Xapian) behaves better when using this approach.

Another option is to use `RealtimeSignalProcessor`, which uses Django's signals to immediately update the index any time a model is saved/deleted. This yields a much more current search index at the expense of being fairly inefficient. Solr & Elasticsearch are the only backends that handles this well under load, and even then, you should make sure you have the server capacity to spare.

A third option is to develop a custom `QueuedSignalProcessor` that, much like `RealtimeSignalProcessor`, uses Django's signals to enqueue messages for updates/deletes. Then writing a management command to consume these messages in batches, yielding a nice compromise between the previous two options.

For more information see *Signal Processors*.

Note: Haystack doesn't ship with a `QueuedSignalProcessor` largely because there is such a diversity of lightweight queuing options and that they tend to polarize developers. Queuing is outside of Haystack's goals (provide good, powerful search) and, as such, is left to the developer.

Additionally, the implementation is relatively trivial & there are already good third-party add-ons for Haystack to enable this.

Advanced Data Preparation

In most cases, using the *model_attr* parameter on your fields allows you to easily get data from a Django model to the document in your index, as it handles both direct attribute access as well as callable functions within your model.

Note: The `model_attr` keyword argument also can look through relations in models. So you can do something like `model_attr='author__first_name'` to pull just the first name of the author, similar to some lookups used by Django's ORM.

However, sometimes, even more control over what gets placed in your index is needed. To facilitate this, `SearchIndex` objects have a 'preparation' stage that populates data just before it is indexed. You can hook into this phase in several ways.

This should be very familiar to developers who have used Django's `forms` before as it loosely follows similar concepts, though the emphasis here is less on cleansing data from user input and more on making the data friendly to the search backend.

1. `prepare_FOO(self, object)`

The most common way to affect a single field's data is to create a `prepare_FOO` method (where FOO is the name of the field). As a parameter to this method, you will receive the instance that is attempting to be indexed.

Note: This method is analogous to Django's `Form.clean_FOO` methods.

To keep with our existing example, one use case might be altering the name inside the `author` field to be "firstname lastname <email>". In this case, you might write the following code:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def prepare_author(self, obj):
        return "%s <%s>" % (obj.user.get_full_name(), obj.user.email)
```

This method should return a single value (or list/tuple/dict) to populate that field's data upon indexing. Note that this method takes priority over whatever data may come from the field itself.

Just like `Form.clean_FOO`, the field's `prepare` runs before the `prepare_FOO`, allowing you to access `self.prepared_data`. For example:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')
```

```
    def get_model(self):
        return Note

    def prepare_author(self, obj):
        # Say we want last name first, the hard way.
        author = u''

        if 'author' in self.prepared_data:
            name_bits = self.prepared_data['author'].split()
            author = "%s, %s" % (name_bits[-1], ' '.join(name_bits[:-1]))

        return author
```

This method is fully function with `model_attr`, so if there's no convenient way to access the data you want, this is an excellent way to prepare it:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    categories = MultiValueField()
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def prepare_categories(self, obj):
        # Since we're using a M2M relationship with a complex lookup,
        # we can prepare the list here.
        return [category.id for category in obj.category_set.active().order_by('-
↪created')]
```

2. `prepare(self, object)`

Each `SearchIndex` gets a `prepare` method, which handles collecting all the data. This method should return a dictionary that will be the final data used by the search backend.

Overriding this method is useful if you need to collect more than one piece of data or need to incorporate additional data that is not well represented by a single `SearchField`. An example might look like:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def prepare(self, object):
        self.prepared_data = super(NoteIndex, self).prepare(object)

        # Add in tags (assuming there's a M2M relationship to Tag on the model).
        # Note that this would NOT get picked up by the automatic
        # schema tools provided by Haystack.
        self.prepared_data['tags'] = [tag.name for tag in object.tags.all()]

        return self.prepared_data
```

If you choose to use this method, you should make a point to be careful to call the `super()` method before altering the data. Without doing so, you may have an incomplete set of data populating your indexes.

This method has the final say in all data, overriding both what the fields provide as well as any `prepare_FOO` methods on the class.

Note: This method is roughly analogous to Django's `Form.full_clean` and `Form.clean` methods. However, unlike these methods, it is not fired as the result of trying to access `self.prepared_data`. It requires an explicit call.

---

3. Overriding `prepare(self, object)` On Individual `SearchField` Objects

The final way to manipulate your data is to implement a custom `SearchField` object and write its `prepare` method to populate/alter the data any way you choose. For instance, a (naive) user-created `GeoPointField` might look something like:

```
from django.utils import six
from haystack import indexes

class GeoPointField(indexes.CharField):
    def __init__(self, **kwargs):
        kwargs['default'] = '0.00-0.00'
        super(GeoPointField, self).__init__(**kwargs)

    def prepare(self, obj):
        return six.text_type("%s-%s" % (obj.latitude, obj.longitude))
```

The `prepare` method simply returns the value to be used for that field. It's entirely possible to include data that's not directly referenced to the object here, depending on your needs.

Note that this is NOT a recommended approach to storing geographic data in a search engine (there is no formal suggestion on this as support is usually non-existent), merely an example of how to extend existing fields.

Note: This method is analagous to Django's `Field.clean` methods.

Adding New Fields

If you have an existing `SearchIndex` and you add a new field to it, Haystack will add this new data on any updates it sees after that point. However, this will not populate the existing data you already have.

In order for the data to be picked up, you will need to run `./manage.py rebuild_index`. This will cause all backends to rebuild the existing data already present in the quickest and most efficient way.

Note: With the Solr backend, you'll also have to add to the appropriate `schema.xml` for your configuration before running the `rebuild_index`.

`Search Index`

get_model

SearchIndex.**get_model**(*self*)

Should return the Model class (not an instance) that the rest of the SearchIndex should use.

This method is required & you must override it to return the correct class.

index_queryset

SearchIndex.**index_queryset**(*self*, *using=None*)

Get the default QuerySet to index when doing a full update.

Subclasses can override this method to avoid indexing certain objects.

read_queryset

SearchIndex.**read_queryset**(*self*, *using=None*)

Get the default QuerySet for read actions.

Subclasses can override this method to work with other managers. Useful when working with default managers that filter some objects.

build_queryset

SearchIndex.**build_queryset**(*self*, *start_date=None*, *end_date=None*)

Get the default QuerySet to index when doing an index update.

Subclasses can override this method to take into account related model modification times.

The default is to use SearchIndex.index_queryset and filter based on SearchIndex.get_updated_field

prepare

SearchIndex.**prepare**(*self*, *obj*)

Fetches and adds/alters data before indexing.

get_content_field

SearchIndex.**get_content_field**(*self*)

Returns the field that supplies the primary document to be indexed.

update

SearchIndex.**update**(*self*, *using=None*)

Updates the entire index.

If using is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

update_object

SearchIndex.**update_object**(*self*, *instance*, *using=None*, ***kwargs*)

Update the index for a single object. Attached to the class's post-save hook.

If `using` is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

remove_object

SearchIndex.**remove_object**(*self*, *instance*, *using=None*, ***kwargs*)

Remove an object from the index. Attached to the class's post-delete hook.

If `using` is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

clear

SearchIndex.**clear**(*self*, *using=None*)

Clears the entire index.

If `using` is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

reindex

SearchIndex.**reindex**(*self*, *using=None*)

Completely clears the index for this model and rebuilds it.

If `using` is provided, it specifies which connection should be used. Default relies on the routers to decide which backend should be used.

get_updated_field

SearchIndex.**get_updated_field**(*self*)

Get the field name that represents the updated date for the model.

If specified, this is used by the reindex command to filter out results from the `QuerySet`, enabling you to reindex only recent records. This method should either return None (reindex everything always) or a string of the `Model`'s `DateField`/`DateTimeField` name.

should_update

SearchIndex.**should_update**(*self*, *instance*, ***kwargs*)

Determine if an object should be updated in the index.

It's useful to override this when an object may save frequently and cause excessive reindexing. You should check conditions on the instance and return False if it is not to be indexed.

The kwargs passed along to this method can be the same as the ones passed by Django when a Model is saved/delete, so it's possible to check if the object has been created or not. See django.db.models.signals.post_save for details on what is passed.

By default, returns True (always reindex).

load_all_queryset

SearchIndex.**load_all_queryset**(*self*)

Provides the ability to override how objects get loaded in conjunction with RelatedSearchQuerySet.load_all. This is useful for post-processing the results from the query, enabling things like adding select_related or filtering certain data.

> **Warning:** Utilizing this functionality can have negative performance implications. Please see the section on RelatedSearchQuerySet within *SearchQuerySet API* for further information.

By default, returns all() on the model's default manager.

Example:

```
class NoteIndex(SearchIndex, indexes.Indexable):
    text = CharField(document=True, use_template=True)
    author = CharField(model_attr='user')
    pub_date = DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note

    def load_all_queryset(self):
        # Pull all objects related to the Note in search results.
        return Note.objects.all().select_related()
```

When searching, the RelatedSearchQuerySet appends on a call to in_bulk, so be sure that the QuerySet you provide can accommodate this and that the ids passed to in_bulk will map to the model in question.

If you need a specific QuerySet in one place, you can specify this at the RelatedSearchQuerySet level using the load_all_queryset method. See *SearchQuerySet API* for usage.

ModelSearchIndex

The ModelSearchIndex class allows for automatic generation of a SearchIndex based on the fields of the model assigned to it.

With the exception of the automated introspection, it is a SearchIndex class, so all notes above pertaining to SearchIndexes apply. As with the ModelForm class in Django, it employs an inner class called Meta, which should contain a model attribute. By default all non-relational model fields are included as search fields on the index, but fields can be restricted by way of a fields whitelist, or excluded with an excludes list, to prevent certain fields from appearing in the class.

In addition, it adds a *text* field that is the document=True field and has *use_template=True* option set, just like the BasicSearchIndex.

Warning: Usage of this class might result in inferior `SearchIndex` objects, which can directly affect your search results. Use this to establish basic functionality and move to custom *SearchIndex* objects for better control.

At this time, it does not handle related fields.

Quick Start

For the impatient:

```
import datetime
from haystack import indexes
from myapp.models import Note

# All Fields
class AllNoteIndex(indexes.ModelSearchIndex, indexes.Indexable):
    class Meta:
        model = Note

# Blacklisted Fields
class LimitedNoteIndex(indexes.ModelSearchIndex, indexes.Indexable):
    class Meta:
        model = Note
        excludes = ['user']

# Whitelisted Fields
class NoteIndex(indexes.ModelSearchIndex, indexes.Indexable):
    class Meta:
        model = Note
        fields = ['user', 'pub_date']

    # Note that regular ``SearchIndex`` methods apply.
    def index_queryset(self, using=None):
        "Used when the entire index for model is updated."
        return Note.objects.filter(pub_date__lte=datetime.datetime.now())
```

Input Types

Input types allow you to specify more advanced query behavior. They serve as a way to alter the query, often in backend-specific ways, without altering your Python code; as well as enabling use of more advanced features.

Input types currently are only useful with the `filter/exclude` methods on `SearchQuerySet`. Expanding this support to other methods is on the roadmap.

Available Input Types

Included with Haystack are the following input types:

Raw

class `haystack.inputs.`**Raw**

Raw allows you to specify backend-specific query syntax. If Haystack doesn't provide a way to access special query functionality, you can make use of this input type to pass it along.

Example:

```
# Fielded.
sqs = SearchQuerySet().filter(author=Raw('daniel OR jones'))

# Non-fielded.
# See ``AltParser`` for a better way to construct this.
sqs = SearchQuerySet().filter(content=Raw('{!dismax qf=author mm=1}haystack'))
```

Clean

class haystack.inputs.**Clean**

`Clean` takes standard user (untrusted) input and sanitizes it. It ensures that no unintended operators or special characters make it into the query.

This is roughly analogous to Django's `autoescape` support.

Note: By default, if you hand a `SearchQuerySet` a bare string, it will get wrapped in this class.

Example:

```
# This becomes "daniel or jones".
sqs = SearchQuerySet().filter(content=Clean('daniel OR jones'))

# Things like ``:`` & ``/`` get escaped.
sqs = SearchQuerySet().filter(url=Clean('http://www.example.com'))

# Equivalent (automatically wrapped in ``Clean``).
sqs = SearchQuerySet().filter(url='http://www.example.com')
```

Exact

class haystack.inputs.**Exact**

`Exact` allows for making sure a phrase is exactly matched, unlike the usual `AND` lookups, where words may be far apart.

Example:

```
sqs = SearchQuerySet().filter(author=Exact('n-gram support'))

# Equivalent.
sqs = SearchQuerySet().filter(author__exact='n-gram support')
```

Not

class haystack.inputs.**Not**

`Not` allows negation of the query fragment it wraps. As `Not` is a subclass of `Clean`, it will also sanitize the query.

This is generally only used internally. Most people prefer to use the `SearchQuerySet.exclude` method.

Example:

```
sqs = SearchQuerySet().filter(author=Not('daniel'))
```

AutoQuery

class haystack.inputs.**AutoQuery**

`AutoQuery` takes a more complex user query (that includes simple, standard query syntax bits) & forms a proper query out of them. It also handles sanitizing that query using `Clean` to ensure the query doesn't break.

`AutoQuery` accommodates for handling regular words, NOT-ing words & extracting exact phrases.

Example:

```
# Against the main text field with an accidental ":" before "search".
# Generates a query like ``haystack (NOT whoosh) "fast search"``
sqs = SearchQuerySet().filter(content=AutoQuery('haystack -whoosh "fast :search"'))

# Equivalent.
sqs = SearchQuerySet().auto_query('haystack -whoosh "fast :search"')

# Fielded.
sqs = SearchQuerySet().filter(author=AutoQuery('daniel -day -lewis'))
```

AltParser

class haystack.inputs.**AltParser**

`AltParser` lets you specify that a portion of the query should use a separate parser in the search engine. This is search-engine-specific, so it may decrease the portability of your app.

Currently only supported under Solr.

Example:

```
# DisMax.
sqs = SearchQuerySet().filter(content=AltParser('dismax', 'haystack', qf='text',␣
↪mm=1))

# Prior to the spatial support, you could do...
sqs = SearchQuerySet().filter(content=AltParser('dismax', 'haystack', qf='author',␣
↪mm=1))
```

Creating Your Own Input Types

Building your own input type is relatively simple. All input types are simple classes that provide an `__init__` & a `prepare` method.

The `__init__` may accept any `args/kwargs`, though the typical use usually just involves a query string.

The `prepare` method lets you alter the query the user provided before it becomes of the main query. It is lazy, called as late as possible, right before the final query is built & shipped to the engine.

A full, if somewhat silly, example looks like:

```
from haystack.inputs import Clean


class NoShoutCaps(Clean):
    input_type_name = 'no_shout_caps'
    # This is the default & doesn't need to be specified.
    post_process = True

    def __init__(self, query_string, **kwargs):
        # Stash the original, if you need it.
        self.original = query_string
        super(NoShoutCaps, self).__init__(query_string, **kwargs)

    def prepare(self, query_obj):
        # We need a reference to the current ``SearchQuery`` object this
        # will run against, in case we need backend-specific code.
        query_string = super(NoShoutCaps, self).prepare(query_obj)

        # Take that, capital letters!
        return query_string.lower()
```

`SearchField` API

class `SearchField`

The `SearchField` and its subclasses provides a way to declare what data you're interested in indexing. They are used with `SearchIndexes`, much like `forms.*Field` are used within forms or `models.*Field` within models.

They provide both the means for storing data in the index, as well as preparing the data before it's placed in the index. Haystack uses all fields from all `SearchIndex` classes to determine what the engine's index schema ought to look like.

In practice, you'll likely never actually use the base `SearchField`, as the subclasses are much better at handling real data.

Subclasses

Included with Haystack are the following field types:

- `BooleanField`
- `CharField`
- `DateField`
- `DateTimeField`
- `DecimalField`
- `EdgeNgramField`
- `FloatField`
- `IntegerField`
- `LocationField`

- `MultiValueField`
- `NgramField`

And equivalent faceted versions:

- `FacetBooleanField`
- `FacetCharField`
- `FacetDateField`
- `FacetDateTimeField`
- `FacetDecimalField`
- `FacetFloatField`
- `FacetIntegerField`
- `FacetMultiValueField`

Note: There is no faceted variant of the n-gram fields. Because of how the engine generates n-grams, faceting on these field types (`NgramField` & `EdgeNgram`) would make very little sense.

Usage

While `SearchField` objects can be used on their own, they're generally used within a `SearchIndex`. You use them in a declarative manner, just like fields in `django.forms.Form` or `django.db.models.Model` objects. For example:

```
from haystack import indexes
from myapp.models import Note


class NoteIndex(indexes.SearchIndex, indexes.Indexable):
    text = indexes.CharField(document=True, use_template=True)
    author = indexes.CharField(model_attr='user')
    pub_date = indexes.DateTimeField(model_attr='pub_date')

    def get_model(self):
        return Note
```

This will hook up those fields with the index and, when updating a `Model` object, pull the relevant data out and prepare it for storage in the index.

Field Options

default

`SearchField.default`

Provides a means for specifying a fallback value in the event that no data is found for the field. Can be either a value or a callable.

document

SearchField.**document**

A boolean flag that indicates which of the fields in the SearchIndex ought to be the primary field for searching within. Default is False.

Note: Only one field can be marked as the document=True field, so you should standardize this name and the format of the field between all of your SearchIndex classes.

---

indexed

SearchField.**indexed**

A boolean flag for indicating whether or not the data from this field will be searchable within the index. Default is True.

The companion of this option is stored.

index_fieldname

SearchField.**index_fieldname**

The index_fieldname option allows you to force the name of the field in the index. This does not change how Haystack refers to the field. This is useful when using Solr's dynamic attributes or when integrating with other external software.

Default is variable name of the field within the SearchIndex.

model_attr

SearchField.**model_attr**

The model_attr option is a shortcut for preparing data. Rather than having to manually fetch data out of a Model, model_attr allows you to specify a string that will automatically pull data out for you. For example:

```
# Automatically looks within the model and populates the field with
# the ``last_name`` attribute.
author = CharField(model_attr='last_name')
```

It also handles callables:

```
# On a ``User`` object, pulls the full name as pieced together by the
# ``get_full_name`` method.
author = CharField(model_attr='get_full_name')
```

And can look through relations:

```
# Pulls the ``bio`` field from a ``UserProfile`` object that has a
# ``OneToOneField`` relationship to a ``User`` object.
biography = CharField(model_attr='user__profile__bio')
```

null

SearchField.**null**

A boolean flag for indicating whether or not it's permissible for the field not to contain any data. Default is `False`.

Note: Unlike Django's database layer, which injects a `NULL` into the database when a field is marked nullable, `null=True` will actually exclude that field from being included with the document. This is more efficient for the search engine to deal with.

---

stored

SearchField.**stored**

A boolean flag for indicating whether or not the data from this field will be stored within the index. Default is `True`.

This is useful for pulling data out of the index along with the search result in order to save on hits to the database.

The companion of this option is `indexed`.

template_name

SearchField.**template_name**

Allows you to override the name of the template to use when preparing data. By default, the data templates for fields are located within your `TEMPLATE_DIRS` under a path like `search/indexes/{app_label}/{model_name}_{field_name}.txt`. This option lets you override that path (though still within `TEMPLATE_DIRS`).

Example:

```
bio = CharField(use_template=True, template_name='myapp/data/bio.txt')
```

You can also provide a list of templates, as `loader.select_template` is used under the hood.

Example:

```
bio = CharField(use_template=True, template_name=['myapp/data/bio.txt', 'myapp/bio.txt
↪', 'bio.txt'])
```

use_template

SearchField.**use_template**

A boolean flag for indicating whether or not a field should prepare its data via a data template or not. Default is False.

Data templates are extremely useful, as they let you easily tie together different parts of the `Model` (and potentially related models). This leads to better search results with very little effort.

Method Reference

__init__

SearchField.**__init__**(*self*, *model_attr=None*, *use_template=False*, *template_name=None*, *document=False*, *indexed=True*, *stored=True*, *faceted=False*, *default=NOT_PROVIDED*, *null=False*, *index_fieldname=None*, *facet_class=None*, *boost=1.0*, *weight=None*)

Instantiates a fresh `SearchField` instance.

has_default

SearchField.**has_default**(*self*)

Returns a boolean of whether this field has a default value.

prepare

SearchField.**prepare**(*self*, *obj*)

Takes data from the provided object and prepares it for storage in the index.

prepare_template

SearchField.**prepare_template**(*self*, *obj*)

Flattens an object for indexing.

This loads a template (`search/indexes/{app_label}/{model_name}_{field_name}.txt`) and returns the result of rendering that template. `object` will be in its context.

convert

SearchField.**convert**(*self*, *value*)

Handles conversion between the data found and the type of the field.

Extending classes should override this method and provide correct data coercion.

SearchResult API

class SearchResult(*app_label*, *model_name*, *pk*, *score*, ***kwargs*)

The `SearchResult` class provides structure to the results that come back from the search index. These objects are what a `SearchQuerySet` will return when evaluated.

Attribute Reference

The class exposes the following useful attributes/properties:

- `app_label` - The application the model is attached to.
- `model_name` - The model's name.
- `pk` - The primary key of the model.

- `score` - The score provided by the search engine.
- `object` - The actual model instance (lazy loaded).
- `model` - The model class.
- `verbose_name` - A prettier version of the model's class name for display.
- `verbose_name_plural` - A prettier version of the model's *plural* class name for display.
- `searchindex` - Returns the `SearchIndex` class associated with this result.
- `distance` - On geo-spatial queries, this returns a `Distance` object representing the distance the result was from the focused point.

Method Reference

content_type

`SearchResult.`**`content_type`**(*self*)

Returns the content type for the result's model instance.

get_additional_fields

`SearchResult.`**`get_additional_fields`**(*self*)

Returns a dictionary of all of the fields from the raw result.

Useful for serializing results. Only returns what was seen from the search engine, so it may have extra fields Haystack's indexes aren't aware of.

get_stored_fields

`SearchResult.`**`get_stored_fields`**(*self*)

Returns a dictionary of all of the stored fields from the SearchIndex.

Useful for serializing results. Only returns the fields Haystack's indexes are aware of as being 'stored'.

`SearchQuery` API

class `SearchQuery`(*using=DEFAULT_ALIAS*)

The `SearchQuery` class acts as an intermediary between `SearchQuerySet`'s abstraction and `SearchBackend`'s actual search. Given the metadata provided by `SearchQuerySet`, `SearchQuery` builds the actual query and interacts with the `SearchBackend` on `SearchQuerySet`'s behalf.

This class must be at least partially implemented on a per-backend basis, as portions are highly specific to the backend. It usually is bundled with the accompanying `SearchBackend`.

Most people will **NOT** have to use this class directly. `SearchQuerySet` handles all interactions with `SearchQuery` objects and provides a nicer interface to work with.

Should you need advanced/custom behavior, you can supply your version of `SearchQuery` that overrides/extends the class in the manner you see fit. You can either hook it up in a `BaseEngine` subclass or `SearchQuerySet` objects take a kwarg parameter `query` where you can pass in your class.

SQ Objects

For expressing more complex queries, especially involving AND/OR/NOT in different combinations, you should use `SQ` objects. Like `django.db.models.Q` objects, `SQ` objects can be passed to `SearchQuerySet.filter` and use the familiar unary operators (`&`, `|` and `~`) to generate complex parts of the query.

> **Warning:** Any data you pass to `SQ` objects is passed along **unescaped**. If you don't trust the data you're passing along, you should use the `clean` method on your `SearchQuery` to sanitize the data.

Example:

```
from haystack.query import SQ

# We want "title: Foo AND (tags:bar OR tags:moof)"
sqs = SearchQuerySet().filter(title='Foo').filter(SQ(tags='bar') | SQ(tags='moof'))

# To clean user-provided data:
sqs = SearchQuerySet()
clean_query = sqs.query.clean(user_query)
sqs = sqs.filter(SQ(title=clean_query) | SQ(tags=clean_query))
```

Internally, the `SearchQuery` object maintains a tree of `SQ` objects. Each `SQ` object supports what field it looks up against, what kind of lookup (i.e. the __ filters), what value it's looking for, if it's a AND/OR/NOT and tracks any children it may have. The `SearchQuery.build_query` method starts with the root of the tree, building part of the final query at each node until the full final query is ready for the `SearchBackend`.

Backend-Specific Methods

When implementing a new backend, the following methods will need to be created:

build_query_fragment

`SearchQuery.`**`build_query_fragment`**(*self*, *field*, *filter_type*, *value*)

Generates a query fragment from a field, filter type and a value.

Must be implemented in backends as this will be highly backend specific.

Inheritable Methods

The following methods have a complete implementation in the base class and can largely be used unchanged.

build_query

`SearchQuery.`**`build_query`**(*self*)

Interprets the collected query metadata and builds the final query to be sent to the backend.

build_params

SearchQuery.**build_params**(*self*, *spelling_query=None*)

Generates a list of params to use when searching.

clean

SearchQuery.**clean**(*self*, *query_fragment*)

Provides a mechanism for sanitizing user input before presenting the value to the backend.

A basic (override-able) implementation is provided.

run

SearchQuery.**run**(*self*, *spelling_query=None*, ***kwargs*)

Builds and executes the query. Returns a list of search results.

Optionally passes along an alternate query for spelling suggestions.

Optionally passes along more kwargs for controlling the search query.

run_mlt

SearchQuery.**run_mlt**(*self*, ***kwargs*)

Executes the More Like This. Returns a list of search results similar to the provided document (and optionally query).

run_raw

SearchQuery.**run_raw**(*self*, ***kwargs*)

Executes a raw query. Returns a list of search results.

get_count

SearchQuery.**get_count**(*self*)

Returns the number of results the backend found for the query.

If the query has not been run, this will execute the query and store the results.

get_results

SearchQuery.**get_results**(*self*, ***kwargs*)

Returns the results received from the backend.

If the query has not been run, this will execute the query and store the results.

get_facet_counts

SearchQuery.**get_facet_counts**(*self*)

Returns the results received from the backend.

If the query has not been run, this will execute the query and store the results.

boost_fragment

SearchQuery.**boost_fragment**(*self*, *boost_word*, *boost_value*)

Generates query fragment for boosting a single word/value pair.

matching_all_fragment

SearchQuery.**matching_all_fragment**(*self*)

Generates the query that matches all documents.

add_filter

SearchQuery.**add_filter**(*self*, *expression*, *value*, *use_not=False*, *use_or=False*)

Narrows the search by requiring certain conditions.

add_order_by

SearchQuery.**add_order_by**(*self*, *field*)

Orders the search result by a field.

clear_order_by

SearchQuery.**clear_order_by**(*self*)

Clears out all ordering that has been already added, reverting the query to relevancy.

add_model

SearchQuery.**add_model**(*self*, *model*)

Restricts the query requiring matches in the given model.

This builds upon previous additions, so you can limit to multiple models by chaining this method several times.

set_limits

SearchQuery.**set_limits**(*self*, *low=None*, *high=None*)

Restricts the query by altering either the start, end or both offsets.

clear_limits

SearchQuery.**clear_limits**(*self*)

Clears any existing limits.

add_boost

SearchQuery.**add_boost**(*self*, *term*, *boost_value*)

Adds a boosted term and the amount to boost it to the query.

raw_search

SearchQuery.**raw_search**(*self*, *query_string*, ***kwargs*)

Runs a raw query (no parsing) against the backend.

This method causes the SearchQuery to ignore the standard query-generating facilities, running only what was provided instead.

Note that any kwargs passed along will override anything provided to the rest of the SearchQuerySet.

more_like_this

SearchQuery.**more_like_this**(*self*, *model_instance*)

Allows backends with support for "More Like This" to return results similar to the provided instance.

add_stats_query

SearchQuery.**add_stats_query**(*self*, *stats_field*, *stats_facets*)

Adds stats and stats_facets queries for the Solr backend.

add_highlight

SearchQuery.**add_highlight**(*self*)

Adds highlighting to the search results.

add_within

SearchQuery.add_within(self, field, point_1, point_2):

Adds bounding box parameters to search query.

add_dwithin

SearchQuery.add_dwithin(self, field, point, distance):

Adds radius-based parameters to search query.

add_distance

SearchQuery.add_distance(self, field, point):

Denotes that results should include distance measurements from the point passed in.

add_field_facet

SearchQuery.**add_field_facet**(*self*, *field*, ***options*)

Adds a regular facet on a field.

add_date_facet

SearchQuery.**add_date_facet**(*self*, *field*, *start_date*, *end_date*, *gap_by*, *gap_amount*)

Adds a date-based facet on a field.

add_query_facet

SearchQuery.**add_query_facet**(*self*, *field*, *query*)

Adds a query facet on a field.

add_narrow_query

SearchQuery.**add_narrow_query**(*self*, *query*)

Narrows a search to a subset of all documents per the query.

Generally used in conjunction with faceting.

set_result_class

SearchQuery.**set_result_class**(*self*, *klass*)

Sets the result class to use for results.

Overrides any previous usages. If `None` is provided, Haystack will revert back to the default `SearchResult` object.

using

SearchQuery.**using**(*self*, *using=None*)

Allows for overriding which connection should be used. This disables the use of routers when performing the query.

If `None` is provided, it has no effect on what backend is used.

SearchBackend API

class SearchBackend(*connection_alias*, ***connection_options*)

The `SearchBackend` class handles interaction directly with the backend. The search query it performs is usually fed to it from a `SearchQuery` class that has been built for that backend.

This class must be at least partially implemented on a per-backend basis and is usually accompanied by a `SearchQuery` class within the same module.

Unless you are writing a new backend, it is unlikely you need to directly access this class.

Method Reference

update

`SearchBackend.`**update**(*self*, *index*, *iterable*)

Updates the backend when given a `SearchIndex` and a collection of documents.

This method MUST be implemented by each backend, as it will be highly specific to each one.

remove

`SearchBackend.`**remove**(*self*, *obj_or_string*)

Removes a document/object from the backend. Can be either a model instance or the identifier (i.e. `app_name.model_name.id`) in the event the object no longer exists.

This method MUST be implemented by each backend, as it will be highly specific to each one.

clear

`SearchBackend.`**clear**(*self*, *models=[]*)

Clears the backend of all documents/objects for a collection of models.

This method MUST be implemented by each backend, as it will be highly specific to each one.

search

`SearchBackend.`**search**(*self*, *query_string*, *sort_by=None*, *start_offset=0*, *end_offset=None*, *fields=''*, *highlight=False*, *facets=None*, *date_facets=None*, *query_facets=None*, *narrow_queries=None*, *spelling_query=None*, *limit_to_registered_models=None*, *result_class=None*, ***kwargs*)

Takes a query to search on and returns a dictionary.

The query should be a string that is appropriate syntax for the backend.

The returned dictionary should contain the keys 'results' and 'hits'. The 'results' value should be an iterable of populated `SearchResult` objects. The 'hits' should be an integer count of the number of matched results the search backend found.

This method MUST be implemented by each backend, as it will be highly specific to each one.

extract_file_contents

SearchBackend.**extract_file_contents**(*self*, *file_obj*)

Perform text extraction on the provided file or file-like object. Returns either None or a dictionary containing the keys `contents` and `metadata`. The `contents` field will always contain the extracted text content returned by the underlying search engine but `metadata` may vary considerably based on the backend and the input file.

prep_value

SearchBackend.**prep_value**(*self*, *value*)

Hook to give the backend a chance to prep an attribute value before sending it to the search engine.

By default, just force it to unicode.

more_like_this

SearchBackend.**more_like_this**(*self*, *model_instance*, *additional_query_string=None*, *result_class=None*)

Takes a model object and returns results the backend thinks are similar.

This method MUST be implemented by each backend, as it will be highly specific to each one.

build_schema

SearchBackend.**build_schema**(*self*, *fields*)

Takes a dictionary of fields and returns schema information.

This method MUST be implemented by each backend, as it will be highly specific to each one.

build_models_list

SearchBackend.**build_models_list**(*self*)

Builds a list of models for searching.

The `search` method should use this and the `django_ct` field to narrow the results (unless the user indicates not to). This helps ignore any results that are not currently handled models and ensures consistent caching.

Architecture Overview

SearchQuerySet

One main implementation.

- Standard API that loosely follows `QuerySet`
- Handles most queries
- Allows for custom "parsing"/building through API
- Dispatches to `SearchQuery` for actual query

- Handles automatically creating a query
- Allows for raw queries to be passed straight to backend.

SearchQuery

Implemented per-backend.

- Method for building the query out of the structured data.
- Method for cleaning a string of reserved characters used by the backend.

Main class provides:

- Methods to add filters/models/order-by/boost/limits to the search.
- Method to perform a raw search.
- Method to get the number of hits.
- Method to return the results provided by the backend (likely not a full list).

SearchBackend

Implemented per-backend.

- Connects to search engine
- Method for saving new docs to index
- Method for removing docs from index
- Method for performing the actual query

SearchSite

One main implementation.

- Standard API that loosely follows `django.contrib.admin.sites.AdminSite`
- Handles registering/unregistering models to search on a per-site basis.
- Provides a means of adding custom indexes to a model, like `ModelAdmins`.

SearchIndex

Implemented per-model you wish to index.

- Handles generating the document to be indexed.
- Populates additional fields to accompany the document.
- Provides a way to limit what types of objects get indexed.
- Provides a way to index the document(s).
- Provides a way to remove the document(s).

Backend Support

Supported Backends

- Solr
- ElasticSearch
- Whoosh
- Xapian

Backend Capabilities

Solr

Complete & included with Haystack.

- Full SearchQuerySet support
- Automatic query building
- "More Like This" functionality
- Term Boosting
- Faceting
- Stored (non-indexed) fields
- Highlighting
- Spatial search
- Requires: pysolr (2.0.13+) & Solr 3.5+

ElasticSearch

Complete & included with Haystack.

- Full SearchQuerySet support
- Automatic query building
- "More Like This" functionality
- Term Boosting
- Faceting (up to 100 facets)
- Stored (non-indexed) fields
- Highlighting
- Spatial search
- Requires: elasticsearch-py 1.x or 2.x. ElasticSearch 5.X is currently unsupported: see #1383.

Whoosh

Complete & included with Haystack.

- Full SearchQuerySet support
- Automatic query building
- "More Like This" functionality
- Term Boosting
- Stored (non-indexed) fields
- Highlighting
- Requires: whoosh (2.0.0+)

Xapian

Complete & available as a third-party download.

- Full SearchQuerySet support
- Automatic query building
- "More Like This" functionality
- Term Boosting
- Faceting
- Stored (non-indexed) fields
- Highlighting
- Requires: Xapian 1.0.5+ & python-xapian 1.0.5+
- Backend can be downloaded here: xapian-haystack

Backend Support Matrix

Back-end	SearchQuery-Set Support	Auto Query Building	More Like This	Term Boost	Faceting	Stored Fields	High-lighting	Spa-tial
Solr	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Elastic-Search	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Whoosh	Yes	Yes	Yes	Yes	No	Yes	Yes	No
Xapian	Yes	Yes	Yes	Yes	Yes	Yes	Yes (plugin)	No

Unsupported Backends & Alternatives

If you have a search engine which you would like to see supported in Haystack, the current recommendation is to develop a plugin following the lead of xapian-haystack so that project can be developed and tested independently of the core Haystack release schedule.

Sphinx

This backend has been requested multiple times over the years but does not yet have a volunteer maintainer. If you would like to work on it, please contact the Haystack maintainers so your project can be linked here and, if desired, added to the django-haystack organization on GitHub.

In the meantime, Sphinx users should consider Jorge C. Leitão's django-sphinxql project.

Haystack Settings

As a way to extend/change the default behavior within Haystack, there are several settings you can alter within your `settings.py`. This is a comprehensive list of the settings Haystack recognizes.

HAYSTACK_DEFAULT_OPERATOR

Optional

This setting controls what the default behavior for chaining `SearchQuerySet` filters together is.

Valid options are:

```
HAYSTACK_DEFAULT_OPERATOR = 'AND'
HAYSTACK_DEFAULT_OPERATOR = 'OR'
```

Defaults to `AND`.

HAYSTACK_CONNECTIONS

Required

This setting controls which backends should be available. It should be a dictionary of dictionaries resembling the following (complete) example:

```
HAYSTACK_CONNECTIONS = {
    'default': {
        'ENGINE': 'haystack.backends.solr_backend.SolrEngine',
        'URL': 'http://localhost:9001/solr/default',
        'TIMEOUT': 60 * 5,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'EXCLUDED_INDEXES': ['thirdpartyapp.search_indexes.BarIndex'],
    },
    'autocomplete': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': '/home/search/whoosh_index',
        'STORAGE': 'file',
        'POST_LIMIT': 128 * 1024 * 1024,
        'INCLUDE_SPELLING': True,
        'BATCH_SIZE': 100,
        'EXCLUDED_INDEXES': ['thirdpartyapp.search_indexes.BarIndex'],
    },
    'slave': {
        'ENGINE': 'xapian_backend.XapianEngine',
        'PATH': '/home/search/xapian_index',
        'INCLUDE_SPELLING': True,
```

```
        'BATCH_SIZE': 100,
        'EXCLUDED_INDEXES': ['thirdpartyapp.search_indexes.BarIndex'],
    },
    'db': {
        'ENGINE': 'haystack.backends.simple_backend.SimpleEngine',
        'EXCLUDED_INDEXES': ['thirdpartyapp.search_indexes.BarIndex'],
    }
}
```

No default for this setting is provided.

The main keys (`default` & friends) are identifiers for your application. You can use them any place the API exposes `using` as a method or kwarg.

There must always be at least a `default` key within this setting.

The `ENGINE` option is required for all backends & should point to the `BaseEngine` subclass for the backend.

Additionally, each backend may have additional options it requires:

- Solr
  - `URL` - The URL to the Solr core. e.g. http://localhost:9001/solr/collection1
  - `ADMIN_URL` - The URL to the administrative functions. e.g. http://localhost:9001/solr/admin/cores
- Whoosh
  - `PATH` - The filesystem path to where the index data is located.
- Xapian
  - `PATH` - The filesystem path to where the index data is located.

The following options are optional:

- `INCLUDE_SPELLING` - Include spelling suggestions. Default is `False`
- `BATCH_SIZE` - How many records should be updated at once via the management commands. Default is `1000`.
- `TIMEOUT` - (Solr and ElasticSearch) How long to wait (in seconds) before the connection times out. Default is `10`.
- `STORAGE` - (Whoosh-only) Which storage engine to use. Accepts `file` or `ram`. Default is `file`.
- `POST_LIMIT` - (Whoosh-only) How large the file sizes can be. Default is `128 * 1024 * 1024`.
- `FLAGS` - (Xapian-only) A list of flags to use when querying the index.
- `EXCLUDED_INDEXES` - A list of strings (as Python import paths) to indexes you do **NOT** want included. Useful for omitting third-party things you don't want indexed or for when you want to replace an index.
- `KWARGS` - (Solr and ElasticSearch) Any additional keyword arguments that should be passed on to the underlying client library.

HAYSTACK_ROUTERS

Optional

This setting controls how routing is performed to allow different backends to handle updates/deletes/reads.

An example:

```
HAYSTACK_ROUTERS = ['search_routers.MasterSlaveRouter', 'haystack.routers.
↪DefaultRouter']
```

Defaults to `['haystack.routers.DefaultRouter']`.

HAYSTACK_SIGNAL_PROCESSOR

Optional

This setting controls what `SignalProcessor` class is used to handle Django's signals & keep the search index up-to-date.

An example:

```
HAYSTACK_SIGNAL_PROCESSOR = 'haystack.signals.RealtimeSignalProcessor'
```

Defaults to `'haystack.signals.BaseSignalProcessor'`.

HAYSTACK_DOCUMENT_FIELD

Optional

This setting controls what fieldname Haystack relies on as the default field for searching within.

An example:

```
HAYSTACK_DOCUMENT_FIELD = 'wall_o_text'
```

Defaults to `text`.

HAYSTACK_SEARCH_RESULTS_PER_PAGE

Optional

This setting controls how many results are shown per page when using the included `SearchView` and its subclasses.

An example:

```
HAYSTACK_SEARCH_RESULTS_PER_PAGE = 50
```

Defaults to `20`.

HAYSTACK_CUSTOM_HIGHLIGHTER

Optional

This setting allows you to specify your own custom `Highlighter` implementation for use with the `{% highlight %}` template tag. It should be the full path to the class.

An example:

```
HAYSTACK_CUSTOM_HIGHLIGHTER = 'myapp.utils.BorkHighlighter'
```

No default is provided. Haystack automatically falls back to the default implementation.

HAYSTACK_ITERATOR_LOAD_PER_QUERY

Optional

This setting controls the number of results that are pulled at once when iterating through a `SearchQuerySet`. If you generally consume large portions at a time, you can bump this up for better performance.

Note: This is not used in the case of a slice on a `SearchQuerySet`, which already overrides the number of results pulled at once.

An example:

```
HAYSTACK_ITERATOR_LOAD_PER_QUERY = 100
```

The default is 10 results at a time.

HAYSTACK_LIMIT_TO_REGISTERED_MODELS

Optional

This setting allows you to control whether or not Haystack will limit the search results seen to just the models registered. It should be a boolean.

If your search index is never used for anything other than the models registered with Haystack, you can turn this off and get a small to moderate performance boost.

An example:

```
HAYSTACK_LIMIT_TO_REGISTERED_MODELS = False
```

Default is `True`.

HAYSTACK_ID_FIELD

Optional

This setting allows you to control what the unique field name used internally by Haystack is called. Rarely needed unless your field names collide with Haystack's defaults.

An example:

```
HAYSTACK_ID_FIELD = 'my_id'
```

Default is `id`.

HAYSTACK_DJANGO_CT_FIELD

Optional

This setting allows you to control what the content type field name used internally by Haystack is called. Rarely needed unless your field names collide with Haystack's defaults.

An example:

```
HAYSTACK_DJANGO_CT_FIELD = 'my_django_ct'
```

Default is `django_ct`.

HAYSTACK_DJANGO_ID_FIELD

Optional

This setting allows you to control what the primary key field name used internally by Haystack is called. Rarely needed unless your field names collide with Haystack's defaults.

An example:

```
HAYSTACK_DJANGO_ID_FIELD = 'my_django_id'
```

Default is `django_id`.

HAYSTACK_IDENTIFIER_METHOD

Optional

This setting allows you to provide a custom method for `haystack.utils.get_identifier`. Useful when the default identifier pattern of <app.label>.<object_name>.<pk> isn't suited to your needs.

An example:

```
HAYSTACK_IDENTIFIER_METHOD = 'my_app.module.get_identifier'
```

Default is `haystack.utils.default_get_identifier`.

HAYSTACK_FUZZY_MIN_SIM

Optional

This setting allows you to change the required similarity when using `fuzzy` filter.

Default is `0.5`

HAYSTACK_FUZZY_MAX_EXPANSIONS

Optional

This setting allows you to change the number of terms fuzzy queries will expand to when using `fuzzy` filter.

Default is `50`

Utilities

Included here are some of the general use bits included with Haystack.

get_identifier

get_identifier(*obj_or_string*)

Gets an unique identifier for the object or a string representing the object.

If not overridden, uses `<app_label>.<object_name>.<pk>`.

CHAPTER 4

Developing

Finally, if you're looking to help out with the development of Haystack, the following links should help guide you on running tests and creating additional backends:

Running Tests

Everything

The simplest way to get up and running with Haystack's tests is to run:

```
python setup.py test
```

This installs all of the backend libraries & all dependencies for getting the tests going and runs the tests. You will still have to setup search servers (for running Solr tests, the spatial Solr tests & the Elasticsearch tests).

Cherry-Picked

If you'd rather not run all the tests, run only the backends you need since tests for backends that are not running will be skipped.

`Haystack` is maintained with all tests passing at all times, so if you receive any errors during testing, please check your setup and file a report if the errors persist.

To run just a portion of the tests you can use the script `run_tests.py` and just specify the files or directories you wish to run, for example:

```
cd test_haystack
./run_tests.py whoosh_tests test_loading.py
```

The `run_tests.py` script is just a tiny wrapper around the nose library and any options you pass to it will be passed on; including `--help` to get a list of possible options:

```
cd test_haystack
./run_tests.py --help
```

Configuring Solr

Haystack assumes that you have a Solr server running on port `9001` which uses the schema and configuration provided in the `test_haystack/solr_tests/server/` directory. For convenience, a script is provided which will download, configure and start a test Solr server:

```
test_haystack/solr_tests/server/start-solr-test-server.sh
```

If no server is found all solr-related tests will be skipped.

Configuring Elasticsearch

The test suite will try to connect to Elasticsearch on port `9200`. If no server is found all elasticsearch tests will be skipped. Note that the tests are destructive - during the teardown phase they will wipe the cluster clean so make sure you don't run them against an instance with data you wish to keep.

If you want to run the geo-django tests you may need to review the GeoDjango GEOS and GDAL settings before running these commands:

```
cd test_haystack
./run_tests.py elasticsearch_tests
```

Creating New Backends

The process should be fairly simple.

1. Create new backend file. Name is important.
2. Two classes inside.
   (a) SearchBackend (inherit from haystack.backends.BaseSearchBackend)
   (b) SearchQuery (inherit from haystack.backends.BaseSearchQuery)

SearchBackend

Responsible for the actual connection and low-level details of interacting with the backend.

- Connects to search engine
- Method for saving new docs to index
- Method for removing docs from index
- Method for performing the actual query

SearchQuery

Responsible for taking structured data about the query and converting it into a backend appropriate format.

- Method for creating the backend specific query - `build_query`.

CHAPTER 5

Requirements

Haystack has a relatively easily-met set of requirements.

- Python 2.7+ or Python 3.3+
- A supported version of Django: https://www.djangoproject.com/download/#supported-versions

Additionally, each backend has its own requirements. You should refer to *Installing Search Engines* for more details.

Index

Symbols

A

B

C

D

E

F

G

H

I