Search

A common task for web applications is to search some data in the database with user input. In a simple case, this could be filtering a list of objects by a category. A more complex use case might require searching with weighting, categorization, highlighting, multiple languages, and so on. This document explains some of the possible use cases and the tools you can use.

We'll refer to the same models used in Making queries.

Use Cases

Standard textual queries

Text-based fields have a selection of matching operations. For example, you may wish to allow lookup up an author like so:

```
>>> Author.objects.filter(name__contains="Terry")
[<Author: Terry Gilliam>, <Author: Terry Jones>]
```

This is a very fragile solution as it requires the user to know an exact substring of the author's name. A better approach could be a case-insensitive match (`icontains`), but this is only marginally better.

A database's more advanced comparison functions

If you're using PostgreSQL, Django provides a selection of database specific tools to allow you to leverage more complex querying options. Other databases have different selections of tools, possibly via plugins or user-defined functions. Django doesn't include any support for them at this time. We'll use some examples from PostgreSQL to demonstrate the kind of functionality databases may have.

> **Searching in other databases:** All of the searching tools provided by `django.contrib.postgres` are constructed entirely on public APIs such as custom lookups and database functions. Depending on your database, you should be able to construct queries to allow similar APIs. If there are specific things which cannot be achieved this way, please open a ticket.

In the above example, we determined that a case insensitive lookup would be more useful. When dealing with non-English names, a further improvement is to use `unaccented comparison`:

```
>>> Author.objects.filter(name__unaccent__icontains="Helen")
[<Author: Helen Mirren>, <Author: Helena Bonham Carter>, <Author: Hélène Joy>]
```

This shows another issue, where we are matching against a different spelling of the name. In this case we have an asymmetry though - a search for `Helen` will pick up `Helena` or `Hélène`, but not the reverse. Another option would be to use a `trigram_similar` comparison, which compares sequences of letters.

For example:

```
>>> Author.objects.filter(name__unaccent__lower__trigram_similar="Hélène")
[<Author: Helen Mirren>, <Author: Hélène Joy>]
```

Now we have a different problem - the longer name of "Helena Bonham Carter" doesn't show up as it is much longer. Trigram searches consider all combinations of three letters, and compares how many appear in both search and source strings. For the longer name, there are more combinations that don't appear in the source string, so it is no longer considered a close match.

The correct choice of comparison functions here depends on your particular data set, for example the language(s) used and the type of text being searched. All of the examples we've seen are on short strings where the user is likely to enter something close (by varying definitions) to the source data.

Document-based search

Standard database operations stop being a useful approach when you start considering large blocks of text. Whereas the examples above can be thought of as operations on a string of characters, full text search looks at the actual words. Depending on the system used, it's likely to use some of the following ideas:

- Ignoring "stop words" such as "a", "the", "and".
- Stemming words, so that "pony" and "ponies" are considered similar.
- Weighting words based on different criteria such as how frequently they appear in the text, or the importance of the fields, such as the title or keywords, that they appear in.

There are many alternatives for using searching software, some of the most prominent are Elastic and Solr. These are full document-based search solutions. To use them with data from Django models, you'll need a layer which translates your data into a textual document, including back-references to the database ids. When a search using the engine returns a certain document, you can then look it up in the database. There are a variety of third-party libraries which are designed to help with this process.

PostgreSQL support

PostgreSQL has its own full text search implementation built-in. While not as powerful as some other search engines, it has the advantage of being inside your database and so can easily be combined with other relational queries such as categorization.

The `django.contrib.postgres` module provides some helpers to make these queries. For example, a query might select all the blog entries which mention "cheese":

```
>>> Entry.objects.filter(body_text__search="cheese")
[<Entry: Cheese on Toast recipes>, <Entry: Pizza recipes>]
```

You can also filter on a combination of fields and on related models:

```
>>> Entry.objects.annotate(
...     search=SearchVector("blog__tagline", "body_text"),
... ).filter(search="cheese")
[
    <Entry: Cheese on Toast recipes>,
    <Entry: Pizza Recipes>,
    <Entry: Dairy farming in Argentina>,
]
```

See the `contrib.postgres` Full text search document for complete details.