Suggester

Suggester - a flexible "autocomplete" component.

A common need in search applications is suggesting query terms or phrases based on incomplete user input. These completions may come from a dictionary that is based upon the main index or upon any other arbitrary dictionary. It's often useful to be able to provide only top-N suggestions, either ranked alphabetically or according to their usefulness for an average user (e.g. popularity, or the number of returned results).

Solr 3.1 includes a component called Suggester that provides this functionality.

Suggester reuses much of the SpellCheckComponent infrastructure, so it also reuses many common SpellCheck parameters, such as `spellcheck=true` or `spellcheck.build=true`, etc. The way this component is configured in `solrconfig.xml` is also very similar:

```
  <searchComponent class="solr.SpellCheckComponent" name="suggest">
    <lst name="spellchecker">
      <str name="name">suggest</str>
      <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
      <str name="lookupImpl">org.apache.solr.spelling.suggest.tst.TSTLookupFactory</str>
      <!-- Alternatives to lookupImpl:
           org.apache.solr.spelling.suggest.fst.FSTLookupFactory   [finite state automaton]
           org.apache.solr.spelling.suggest.fst.WFSTLookupFactory [weighted finite state automaton]
           org.apache.solr.spelling.suggest.jaspell.JaspellLookupFactory [default, jaspell-based]
           org.apache.solr.spelling.suggest.tst.TSTLookupFactory   [ternary trees]
      -->
      <str name="field">name</str>  <!-- the indexed field to derive suggestions from -->
      <float name="threshold">0.005</float>
      <str name="buildOnCommit">true</str>
<!--
      <str name="sourceLocation">american-english</str>
-->
    </lst>
  </searchComponent>
  <requestHandler class="org.apache.solr.handler.component.SearchHandler" name="/suggest">
    <lst name="defaults">
      <str name="spellcheck">true</str>
      <str name="spellcheck.dictionary">suggest</str>
      <str name="spellcheck.onlyMorePopular">true</str>
      <str name="spellcheck.count">5</str>
      <str name="spellcheck.collate">true</str>
    </lst>
    <arr name="components">
      <str>suggest</str>
    </arr>
  </requestHandler>
```

The look-up of matching suggestions in a dictionary is implemented by subclasses of the Lookup class - the implementations that are included in Solr are:

- JaspellLookup - tree-based representation based on Jaspell,
- TSTLookup - ternary tree based representation, capable of immediate data structure updates,
- FSTLookup - automaton based representation; slower to build, but consumes far less memory at runtime (see performance notes below).
- WFSTLookup - weighted automaton representation: an alternative to FSTLookup for more fine-grained ranking. Solr 3.6+

For practical purposes all of the above implementations will most likely run at similar speed when requests are made via the HTTP stack (which will become the bottleneck). Direct benchmarks of these classes indicate that (W)FSTLookup provides better performance compared to the other two methods, at a much lower memory cost. JaspellLookup can provide "fuzzy" suggestions, though this functionality is not currently exposed (it's a one line change in JaspellLookup). Support for infix-suggestions is planned for FSTLookup (which would be the only structure to support these).

An example of an autosuggest request:

```
http://localhost:8983/solr/suggest?q=ac
```

And the corresponding response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="spellcheck">
    <lst name="suggestions">
      <lst name="ac">
        <int name="numFound">2</int>
        <int name="startOffset">0</int>
        <int name="endOffset">2</int>
        <arr name="suggestion">
          <str>acquire</str>
          <str>accommodate</str>
        </arr>
      </lst>
      <str name="collation">acquire</str>
    </lst>
  </lst>
</response>
```

Configuration

The configuration snippet above shows a few common configuration parameters. A complete list of them is best found int he source code of each Lookup class, but here is an overview:

SpellCheckComponent configuration

- `searchComponent/@name` - arbitrary name for this component
- `spellchecker` list:
    - `name` - a symbolic name of this spellchecker (can be later referred to in URL parameters and in SearchHandler configuration - see the section below)
    - `classname` - Suggester, to provide the autocomplete functionality
    - `lookupImpl` - Lookup implementation. These in-memory implementations are available:
        - `org.apache.solr.spelling.suggest.tst.TSTLookupFactory` - a simple compact ternary trie based lookup
        - `org.apache.solr.spelling.suggest.jaspell.JaspellLookupFactory` - a more complex lookup based on a ternary trie from the JaSpell project.
        - `org.apache.solr.spelling.suggest.fst.FSTLookupFactory` - automaton-based lookup
        - `org.apache.solr.spelling.suggest.fst.WFSTLookupFactory` - weighted automaton-based lookup
    - `buildOnCommit` - if set to true then the Lookup data structure will be rebuilt after commit. If false (default) then the Lookup data will be built only when requested (by URL parameter `spellcheck.build=true`). **NOTE: currently implemented Lookup-s keep their data in memory, so unlike spellchecker data this data is discarded on core reload and not available until you invoke the build command, either explicitly or implicitly via commit.**
    - `sourceLocation` - location of the dictionary file. If not empty then this is a path to a dictionary file (see below). If this value is empty then the main index will be used as a source of terms and weights.
    - `field` - if `sourceLocation` is empty then terms from this field in the index will be used when building the trie.
    - `threshold` - threshold is a value in [0..1] representing the minimum fraction of documents (of the total) where a term should appear, in order to be added to the lookup dictionary.
    - `storeDir` - where to store the index data on the disk (else use in-memory).

Dictionary

When a file-based dictionary is used (non-empty `sourceLocation` parameter above) then it's expected to be a plain text file in UTF-8 encoding. Blank lines and lines that start with a '#' are ignored. The remaining lines must consist of either a string without literal TAB (\u0007) character, or a string and a TAB separated floating-point weight.

Example:

```
# This is a sample dictionary file.

acquire
accidentally\t2.0
accommodate\t3.0
```

If weight is missing it's assumed to be equal 1.0. Weights affect the sorting of matching suggestions when `spellcheck.onlyMorePopular=true` is selected - weights are treated as "popularity" score, with higher weights preferred over suggestions with lower weights.

Please note that the format of the file is not limited to single terms but can also contain phrases - which is an improvement over the TermsComponent that you could also use for a simple version of autocomplete functionality.

FSTLookup has a built-in mechanism to discretize weights into a fixed set of buckets (to speed up suggestions). The number of buckets is configurable.

WFSTLookup does not use buckets, but instead a shortest path algorithm. Note that it expects weights to be whole numbers.

Threshold parameter

As mentioned above, if the `sourceLocation` parameter is empty then the terms from a field indicated by the `field` parameter are used. It's often the case that due to imperfect source data there are many uncommon or invalid terms that occur only once in the whole corpus (e.g. OCR errors, typos, etc). According to the Zipf's law this actually forms the majority of terms, which means that the dictionary built indiscriminately from a real-life index would consist mostly of uncommon terms, and its size would be enormous. In order to avoid this and to reduce the size of in-memory structures it's best to set the `threshold` parameter to a value slightly above zero (0.5% in the example above). This already vastly reduces the size of the dictionary by skipping "hapax legomena" while still preserving most of the common terms. This parameter has no effect when using a file-based dictionary - it's assumed that only useful terms are found there. 😉

SearchHandler configuration

In the example above we add a new handler that uses SearchHandler with a single SearchComponent that we just defined, namely the `suggest` component. Then we define a few defaults for this component (that can be overridden with URL parameters):

- `spellcheck=true` - because we always want to run the Suggester for queries submitted to this handler.
- `spellcheck.dictionary=suggest` - this is the name of the dictionary component that we configured above.
- `spellcheck.onlyMorePopular=true` - if this parameter is set to true then the suggestions will be sorted by weight ("popularity") - the `count` parameter will effectively limit this to a top-N list of best suggestions. If this is set to false then suggestions are sorted alphabetically.
- `spellcheck.count=5` - specifies to return up to 5 suggestions.
- `spellcheck.collate=true` - to provide a query collated with the first matching suggestion.

Tips and tricks

- Use (W)FSTLookup to conserve memory (unless you need a more sophisticated matching, then look at JaspellLookup). There are some benchmarks of all four implementations: SOLR-1316 (outdated) and a bit newer here:

SOLR-2378, and here: LUCENE-3714. The class to perform these benchmarks is in the source tree and is called LookupBenchmarkTest.

- Use `threshold` parameter to limit the size of the trie, to reduce the build time and to remove invalid/uncommon terms. Values below 0.01 should be sufficient, greater values can be used to limit the impact of terms that occur in a larger portion of documents. Values above 0.5 probably don't make much sense.
- Don't forget to invoke `spellcheck.build=true` after core reload. Or extend the Lookup class to do this on init(), or implement the load/save methods in Lookup to persist this data across core reloads.
- If you want to use a dictionary file that contains phrases (actually, strings that can be split into multiple tokens by the default QueryConverter) then define a different QueryConverter like this:

```
  <!--
   The SpellingQueryConverter to convert raw (CommonParams.Q) queries into tokens.  Define a simple regular
expression
   in your QueryAnalyzer chain if you want to strip off field markup, boosts, ranges, etc.
   -->
  <queryConverter name="queryConverter" class="org.apache.solr.spelling.SuggestQueryConverter"/>
```

An example for setting up a typical case of auto-suggesting phrases (e.g. previous queries from query logs with associated score) is here:

- solrconfig.xml: Example WFST config that uses the SuggestQueryConverter with the phrase_suggest fieldtype.
- schema.xml: Example phrase_suggest fieldtype that lowercases, collapses runs of whitespace, tries to remove query syntax, etc
- example phrase dictionary with weights: Tab-separated example list of phrases and weights.