# Simple Faceted Search

## Lucene.Net Simple Faceted Search

### What is faceted search?

Faceted search is the ability to find results based upon classifications of data which are not fixed. Facets can allow the user to filter data through multiple paths and different ordering. You can click on any combination of facets and arrive to the results from different paths.

e.g. e-commerce searches over clothes that have facets of price range, color, brands, sex, etc.

#### References to adding faceted to Lucene.Net.

- http://markmail.org/message/hyetqvex7bqnshek#query:+page:1+mid:zrew4dimoktd6vex+state:results
- (external file) http://markmail.org/download.xqy?id=zrew4dimoktd6vex&number=1

#### Status: Prototyping inside the Lucene.Net_2_9_4g branch

Which means anything can be a breaking change before its put into a release.

This shouldn't prevent anyone from making their own version or taking the existing code and building on top of it, improving it. In fact we encourage that you do so and submit patches and influence the design.

### Simple Faceted Search

A contrib project was created for implementing a simple faceted search inside of Lucene.Net:
https://issues.apache.org/jira/browse/LUCENENET-415
svn repository: https://svn.apache.org/repos/asf/incubator/lucene.net/branches/Lucene.Net_2_9_4g/src/contrib/SimpleFacetedSearch

The tests do show how to use the simple faceted search, look at the TestSimpleFacetedSearch.cs file and the "HowToUse" Method.
https://svn.apache.org/repos/asf/incubator/lucene.net/branches/Lucene.Net_2_9_4g/test/contrib/SimpleFacetedSearch/TestSimpleFacetedSearch.cs

#### Sample Usage

The usage is pretty straight forward. The code sample below actually comes from the tests for simple faceted search, so we encourage you to look at the rest of the tests inside the project.

```
            // Pulled from the HowToUse method.

            // create the index reader.
            // * the code sample assumes you've already done this, which is what the _Reader variable
references below

            Query query = new QueryParser(Lucene.Net.Util.Version.LUCENE_29, "text", new
            StandardAnalyzer(Lucene.Net.Util.Version.LUCENE_29)).Parse(searchString);


            // pass in the reader and the names of the facets that you've created using fields in the documents.
            // the facets are determined

            SimpleFacetedSearch sfs = new SimpleFacetedSearch(_Reader, new string[] { "language", "category" });

            // then pass in the query into the search like you normally would with a typical search class.

            SimpleFacetedSearch.Hits hits = sfs.Search(query, 10);

            // what comes back is different than normal.
            // the result documents & hits are grouped by facets.

            // you'll need to iterate over groups of hits-per-facet.

            long totalHits = hits.TotalHitCount;
            foreach (SimpleFacetedSearch.HitsPerFacet hpg in hits.HitsPerFacet)
            {
                long hitCountPerGroup = hpg.HitCount;
                SimpleFacetedSearch.FacetName facetName = hpg.Name;

                foreach (Document doc in hpg.Documents)
                {
                    string text = doc.GetField("text").StringValue();

                    // replace with logging or your desired output writer
                    System.Diagnostics.Debug.WriteLine(">>" + facetName + ": " + text);

                }
            }
```

### Under the hood.

Some things that will help understanding what the code actually does underneath the covers.

- Cartesian Product - a set of products
  - for those of you rusty on your math, a good simple example would be (x,y) coordinates.
  - http://en.wikipedia.org/wiki/Cartesian_product
- FieldValuesBitSets
  - maps a field / facet name to a bit set (OpenBitSetDISI) and its underlying values. e.g. if the facet was color, it would put all the distinct values for this field into a list like "blue","cyan","fusca","butterscotch" and all those other fun random names of colors into one flattened list.
  - This basically helps facilitate storing the bit set combinations described by example in the facet algorithm section below
- OpenBitSetDISI
  - DISI stands for Document Id Set Iterator

Basically the search class builds up and caches bitsets of possible combinations on the specified facets that are supplied in the constructor of the query.

If the data does not have a certain combination of facets, it omits these combinations from being stored.

e.g. if you only have "grey" shirts for "males" indexed and you have 0 "grey" shirts indexed for "females", it will omit the "grey" shirts for "female" combination in the possible combinations. "grey" is a valid color and "females" is a valid sex for under each facet, they are not a valid combination. These combinations will never have hits.

The SimpleFacetedSearch will consume the supplied query and then return the groups of facet combinations and their hits.

### Facet Algorithm

Suppose, we want a faceted search on fields f1 f2 f3,
and their values in index are

| | f1 | f2 | f3 |
|---|---|---|---|
| | – | – | – |
| doc1 | A | I | 1 |
| doc2 | A | I | 2 |
| doc3 | A | I | 3 |
| doc4 | A | J | 1 |
| doc5 | A | J | 2 |
| doc6 | A | J | 3 |
| doc7 | B | I | 1 |

Algorithm:
1- Find all possible values for f1 which are (A,B) , for f2 which are (I,J) and for f3 which are (1,2,3)
2- Find Cartesian Product of (A,B)X(I,J)X(1,2,3). (12 possible groups)
3- Eliminate the ones that surely result in 0 hits. (for ex, B J 2. since they have no doc. in common)

## Tips & Tricks

- SFS instances are thread-safe. So, the same instance can be shared among many threads like IndexReader
- SFS should only be created when an IndexReader is opened/reopened. Creation with every search can be performance killer
- Iterating over the thousands of results(e.g, 1000 facets x 25 Hits per facet) can be too costly. The old trick can be used here too, "just show the hit counts, and make a new search when user clicks a category".

## Memory Requirements

For each instance of SFS and search results:
MemUsage = (Document# in Index) / 8 * (Facet Count); // e.g, 1M docs & 1000 facets cost ~128MB (*So, caching results of SFS searches may not be a good idea*)

## Thanks

Thanks to DIGY and Ben West for working on this and putting this together for the community.

## Notes

if any of this information is out of date, please contact the developers list and notify them or if you have wiki access, please modify.