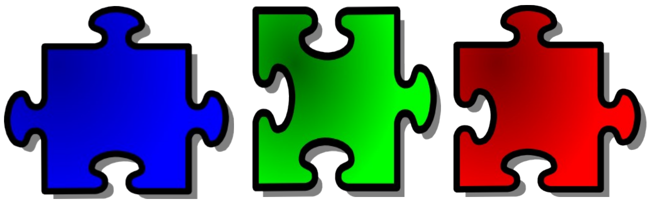
Apache Lucene



Faceted Search

USER'S GUIDE

Table of Contents

1 Introduction

A category is an aspect of indexed documents which can be used to classify the documents. For example, in a collection of books at an online bookstore, categories of a book can be its price, author, publication date, binding type, and so on.

In *faceted search,* in addition to the standard set of search results, we also get *facet* results, which are lists of subcategories for certain categories. For example, for the price facet, we get a list of relevant price ranges; for the author facet, we get a list of relevant authors; and so on. In most UIs, when users click one of these subcategories, the search is *narrowed*, or *drilled down*, and a new search limited to this subcategory (e.g., to a specific price range or author) is performed.

Note that faceted search is more than just the ordinary *fielded search*. In fielded search, users can add search keywords like `price:10` or `author:"Mark Twain"` to the query to narrow the search, but this requires knowledge of which fields are available, and which values are worth trying. This is where faceted search comes in: it provides a list of useful subcategories, which ensures that the user only drills down into useful subcategories and never into a category for which there are no results. In essence, faceted search makes it easy to *navigate through* the search results.

The list of subcategories provided for each facet is also useful to the user in itself, even when the user never drills down. This list allows the user to see at one glance some statistics on the search results, e.g., what price ranges and which authors are most relevant to the given query.

In recent years, faceted search has become a very common UI feature in search engines, especially in e-commerce websites. Faceted search makes it easy for untrained users to find the specific item they are interested in, whereas manually adding search keywords (as in the examples above) proved too cumbersome for ordinary users, and required too much guesswork, trial-and-error, or the reading of lengthy help pages.

See http://en.wikipedia.org/wiki/Faceted_search for more information on faceted search.

1. Facet Features

First and main faceted search capability that comes to mind is counting, but in fact faceted search is more than facet counting. We now briefly discuss the available faceted search features.

1.1. Facet Counting

Which of the available subcategories of a facet should a UI display? A query in a book store might yield books by a hundred different authors, but normally we'd want do display only, say, ten of those.

Most available faceted search implementations use **counts** to determine the importance of each subcategory. These implementations go over all search results for the given query, and count how many results are in each subcategory. Finally, the subcategories with the most results can be displayed. So the user sees the price ranges, authors, and so on, for which there are most results. Often, the count is displayed next to the subcategory name, in parentheses, telling the user how many results he can expect to see if he drills down into this subcategory.

The main API for obtaining facet counting is `CountFacetRequest` – as in the following code snippet:

```
new CountFacetRequest(new CategoryPath("author"), 10));
```

A detailed code example using count facet requests is shown below – see Accumulating Facets.

1.2. Facet Associations

So far we've discussed categories as binary features, where a document either belongs to a category, or not.

While counts are useful in most situations, they are sometimes not sufficiently informative for the user, with respect to deciding which subcategory is more important to display.

For this, the facets package allows to associate a value with a category. The search time interpretation of the associated value is application dependent. For example, a possible interpretation is as a “match level” (e.g., confidence level). This value can then be used so that a document that is very weakly associated with a certain category will only contribute little to this category's aggregated weight.

1.3. Multiple Facet Requests At Once

A single faceted accumulation is capable of servicing multiple facet requests. Programmatic, this is quite simple – wrap all the facet requests of interest into the facet-search-parameters which are passed to a facets accumulator/collector (more on these objects below). The results would be comprised of as many facet results as there were facet requests.

However there is a delicate **limitation**: all facets maintained in the same location in the index are required to be treated the same. See the section on Indexing Parameters for an explanation on maintaining certain facets at certain locations.

1.4. Facet Labels at Search Time

Facets results always contain the facet (internal) ID and (accumulated) value. Some of the results also contain the facet label, AKA the category name. We mention this here since computing the label is a time consuming task, and hence applications can specify with a facet request to return top 1000 facets but to compute the label only for the top 10 facets. In order to compute labels for more of the facet results it is not required to perform accumulation again.

See `FacetRequest.getNumResults()` `FacetRequest.getNumLabel()` and `FacetResultNode.getLabel(TaxonomyReader)`.

2. Indexing Categories Illustrated

In order to find facets at search time they must first be added to the index at indexing time. Recall that Lucene documents are made of fields for textual search. The addition of categories is performed by an appropriate `DocumentBuilder` – or `CategoryDocumentBuilder` in our case.

Indexing therefore usually goes like this:

- For each input document:
  - Create a fresh (empty) Lucene Document
  - Parse input text and add appropriate text search fields
  - **Gather all input categories associated with the document and create a CategoryDocumentBuilder with the list of categories**
  - **"Build" the document – this actually adds the categories to the Lucene document.**
  - Add the document to the index

Following is a code snippet for indexing categories. The complete example can be found in package `org.apache.lucene.facet.example.simple.SimpleIndexer` .

```
    IndexWriter writer = ...
    TaxonomyWriter taxo =
       new LuceneTaxonomyWriter(taxoDir, OpenMode.CREATE);
    ...
1   Document doc = new Document();
    doc.add(new Field(
       "title", titleText, Store.YES, Index.ANALYZED));
    ...
2   List<CategoryPath> categories = new ArrayList<CategoryPath>();
3   categories.add(new CategoryPath("author", "Mark Twain"));
    categories.add(new CategoryPath("year", "2010"));
    ...
4   DocumentBuilder categoryDocBuilder =
       new CategoryDocumentBuilder(taxo);
    categoryDocBuilder.setCategoryPaths(categories);
    categoryDocBuilder.build(doc);
5   writer.addDocument(doc)
```

We now explain the steps above, following the code line numbers:

1. Document contains not only text search fields but also facet search information.
2. Prepare a container for document categories.
3. Categories that should be added to the document are accumulated in the categories list.
4. A `CategoryDocumentBuilder` is created, set with the appropriate list of categories, and invoked to "build" - that is, to populate the document with categories. It is in this step that the taxonomy is updated to contain the newly added categories (if not already there) – see more on this in the section about the taxonomy index below. This line could be made more compact: one can create a single `CategoryDocumentBuilder cBuilder` and reuse it like this:

```
DocumentBuilder cBuilder = new CategoryDocumentBuilder(taxo);
...
cBuilder.setCategoryPaths(categories).build(doc);
```

5. Add the document to the index. As a result, category info is saved also in the regular search index, for supporting facet aggregation at search time (e.g. facet counting) as well as facet drill-down. For more information on indexed facet information see below the section Indexed Facet Information.

3. Accumulating Facets Illustrated

Facets accumulation reflects a set of documents over some facet requests:

- Document set – a subset of the index documents, usually documents matching a user query.
- Facet requests – facet **accumulation** specification, e.g. count a certain facet *dimension*.

`FacetRequest` is a basic component in faceted search – it describes the facet information need. Every facet **request** is made of at least two fields:

- `CategoryPath` – root category of the facet request. The categories that are returned as a result of the request will all be descendants of this root
- `Number of Results` – number of sub-categories to return (at most).

There are other parameters to a facet request, such as "how many facet results to label", "how **deep** to go from the request root when serving the facet request" and more – see the API Javadocs for `FacetRequest` and its subclasses for more information on these parameters. For labels in particular, see the section Facet Labels at Search Time.

`FacetRequest` in an abstract class, open for extensions, and users may add their own requests. The most often used request is `CountFacetRequest` – used for counting facets.

Facets accumulation is – not surprisingly – driven by a `FacetsAccumulator`. The most used one is `StandardFacetsAccumulator`, however there are also accumulators that support sampling – to be used in huge collections, and there's an adaptive facets accumulator which applies sampling conditionally on the statistics of the data. While facets accumulators are very extendible and powerful, they might be too overwhelming for beginners. For this reason, the code offers a higher level interface for facets accumulating: the `FacetsCollector`. It extends `Collector`, and as such can be passed to the search() method of Lucene's `IndexSearcher`. In case the application also needs to collect documents (in addition to accumulating/collecting facets), it can wrap multiple collectors with `MultiCollector`. Most code samples below use `FacetsCollector` due to its simple interface. It is quite likely that `FacetsCollector` should suffice the needs of most applications, therefore we recommend to start with it, and only when needing more flexibility turn to directly use facets accumulators.

Following is a code snippet from the example code – the complete example can be found under `org.apache.lucene.facet.example.simple.Searcher`

```
1  IndexReader indexReader = IndexReader.open(indexDir);
   Searcher searcher = new IndexSearcher(indexReader);
2  TaxonomyReader taxo = new LuceneTaxonomyReader(taxoDir);
   ...
3  Query q = new TermQuery(new Term(SimpleUtils.TEXT, "white"));
   TopScoreDocCollector tdc = TopScoreDocCollector.create(10, true);
   ...
4  FacetSearchParams facetSearchParams = new FacetSearchParams();
5  facetSearchParams.addFacetRequest(
      new CountFacetRequest(
         new CategoryPath("author"), 10));
```

```
  ...
6 FacetsCollector facetsCollector =
     new FacetsCollector(facetSearchParams, indexReader, taxo);
7 searcher.search(q,
     MultiCollector.wrap(topDocsCollector, facetsCollector));
8 List<FacetResult> res = facetsCollector.getFacetResults();
```

We now explain the steps above, following the code line numbers:

1. Index reader and Searcher are initialized as usual.
2. A taxonomy reader is opened – it provides access to the facet information which was stored by the Taxonomy Writer at indexing time.
3. Regular text query is created to find the documents matching user need, and a collector for collecting the top matching documents is created.
4. Facet-search-params is a container for facet requests.
5. A single facet-request – namely a count facet request – is created and added to the facet search params. The request should return top 10 Author subcategory counts.
6. Facets-Collector is the simplest interface for facets accumulation (counting in this example).
7. Lucene search takes both collectors – facets-collector and top-doc-collector, both wrapped by a multi-collector. This way, a single search operation finds both top documents and top facets. Note however that facets aggregation takes place not only over the top documents, but rather over all documents matching the query.
8. Once search completes, facet-results can be obtained from the facets-collector.

Returned facet results are organized in a list, conveniently ordered the same as the facet-requests in the facet-search-params. Each result however contains the request for which it was created.

Here is the (recursive) structure of the facet result:

- **Facet Result**
  - **Facet Request** – the request for which this result was obtained.
  - **#Valid Descendants** – how many valid descendants were encountered over the set of matching documents (some of which might have been filtered out because e.g. only top 10 results were requested).
  - **Root Result Node** – root facet result for the request
    - **Ordinal** – unique internal ID of the facet
    - **Label** – full label of the facet (possibly null)
    - **Value** – facet value, e.g. count
    - **Sub-results-nodes** – child result nodes (possibly null)

Note that not always there would be sub result nodes – this depends on the requested result mode:

- **PER_NODE_IN_TREE** – a tree, and so there may be sub results.
- **GLOBAL_FLAT** – here the results tree would be rather flat, with only (at most) leaves below the root result node.

4. Indexed Facet Information

When indexing a document to which categories were added, information on these categories is added to the search index, in two locations:

- *Category Tokens* are added to the document for each category attached to that document. These categories can be used at search time for drill-down.
- A special *Category List Token* is added to each document containing information on all the categories that were added to this document. This can be used at search time for facet accumulation, e.g. facet counting.

When a category is added to the index (that is, when a document containing a category is indexed), all its parent categories are added as well. For example, indexing a document with the category `<"author", "American", "Mark Twain">` results in creating three tokens: “`/author`”, “`/author/American`”, and “`/author/American/Mark Twain`” (the character '/' here is just a human readable separator – there's no such element in the actual index). This allows drilling-down and counting any category in the taxonomy, and not just leaf nodes, enabling a UI application to show either how many books have authors, or how many books have American authors, or how many books have Mark Twain as their (American) author.

Similarly, Drill-down capabilities are this way possible also for node categories.

In order to keep the counting list compact, it is built using category ordinal – an ordinal is an integer number attached to a category when it is added for the first time into the taxonomy.

For ways to further alter facet index see the section below on Facet Indexing Parameters.

5. Taxonomy Index

The taxonomy is an auxiliary data-structure maintained side-by-side with the regular index to support faceted search operations. It contains information about all the categories that ever existed in any document in the index. Its API is open and allows simple usage, or more advanced for the interested users.

When a category is added to a document, a corresponding node is added to the taxonomy (unless already there). In fact, sometimes more than one node is added – each parent category is added as well, so that the taxonomy is maintained as a Tree, with a virtual root.

So, for the above example, adding the category <`"author","American","Mark Twain"`> actually added two nodes: one for "/author" and one for "/author/Mark Twain".

An integer number – called ordinal is attached to each category the first time the category is added to the taxonomy. This allows for a compact representation of category list tokens in the index, for facets accumulation.

One interesting fact about the taxonomy index is worth knowing: once a category is added to the taxonomy, it is never removed, even if all related documents are removed. This differs from a regular index, where if all documents containing a certain term are removed, and their segments are merged, the term will also be removed. This might cause a performance issue: large taxonomy means large ordinal numbers for categories, and hence large categories values arrays would be maintained during accumulation. It is probably not a real problem for most applications, but be aware of this. If, for example, an application at a certain point in time removes an index entirely in order to recreate it, or, if it removed all the documents from the index in order to re-populate it, it also makes sense in this opportunity to remove the taxonomy index and create a new, fresh one, without the unused categories.

6. Facet Parameters

Facet parameters control how categories and facets are indexed and searched. Apart from specifying facet requests within facet search parameters, under default settings it is not required to provide any parameters, as there are ready to use working defaults for everything.

However many aspects are configurable and can be modified by providing altered facet parameters for either search or indexing.

6.1. Facet Indexing parameters

Facet Indexing Parameters are consulted with during indexing. Among several parameters it defines, the following two are likely to interest many applications:

- **Category list definitions** – in the index, facets are maintained in two forms: category-tokens (for drill-down) and category-list-tokens (for accumulation). This parameter allows to specify, for each category, the Lucene term used for maintaining the category-list-tokens for that category. The default implementation in `DefaultFacetIndexingParams` maintains this information for all categories under the same special dedicated term. One case where it is needed to maintain two categories in separate category lists, is when it is known that at search time it would be required to use different types of accumulation logic for each, but at the same accumulation call.
- **Partition size** – category lists can be maintained in a partitioned way. If, for example, the partition size is set to 1000, a distinct sub-term is used for maintaining each 1000 categories, e.g. term1 for categories 0 to 999, term2 for categories 1000 to 1999, etc. The default implementation in `DefaultFacetIndexingParams` maintains category lists in a single partition, hence it defines the partition size as `Integer.MAX_VALUE`. The importance of this parameter is on allowing to handle very large taxonomies without exhausting RAM resources. This is because at facet accumulation time, facet values arrays are maintained in the size of the partition. With a single partition, the size of these arrays is as the size of the taxonomy, which might be OK for most applications. Limited partition sizes allow to perform the accumulation with less RAM, but with some runtime overhead, as the matching documents are processed for each of the partitions.

See the API Javadocs of `FacetIndexingParams` for additional configuration capabilities which were not discussed here.

6.2. Facet Search Parameters

Facet Search Parameters, consulted at search time (during facets accumulation) are rather plain, providing the following:

- **Facet indexing parameters** – which were in effect at indexing time – allowing facets accumulation to “understand” how facets are maintained in the index.

- **Container of facet requests** – the requests which should be accumulated.

6.3. Category Lists, Multiple, Dimensions

Category list parameters which are accessible through the facet indexing parameters provide the information about:

- Lucene Term under which category information is maintained in the index.
- Encoding (and decoding) used for writing and reading the categories information in the index.

For cases when certain categories should be maintained in different location than others, use `PerDimensionIndexingParams`, which returns a different `CategoryListParams` object for each “dimension”. This is a good opportunity to explain about dimensions. This is just a notion: the top element – or first element – in a category path is denoted as the dimension of that category. Indeed, the dimension stands out as a top important part of the category path, such as “Location” for the category “Location/Europe/France/Paris”.

7. Advanced Faceted Examples

We now provide examples for more advanced facet indexing and search, such as drilling-down on facet values and multiple category lists.

7.1. Drill-down with regular facets

Drill-down allows users to focus on part of the results. Assume a commercial sport equipment site where a user is searching for a tennis racquet. The user issues the query “tennis racquet” and as result is shown a page with 10 tennis racquets, by various providers, of various types and prices. In addition, the site UI shows to the user a break down of all available racquets by price and make. The user now decides to focus on racquets made by “Head”, and will now be shown a new page, with 10 Head racquets, and new break down of the results into racquet types and prices. Additionally, the application can choose to display a new breakdown, by racquet weights. This step of moving from results (and facet statistics) of the entire (or larger) data set into a portion of it by specifying a certain category, is what we call “Drill-down”. We now show the required code lines for implementing such a drill-down.

```
1  Query baseQuery = queryParser.parse("tennis racquet");
   ...
2  Query q2 = DrillDown.query(baseQuery,
                              new CategoryPath("make", "head"), 10));
```

In line 1 the original user query is created and then used to obtain information on all tennis racquets.

In line 2, a specific category from within the facet results was selected by the user, and is hence used for creating the drill-down query.

Please refer to `SimpleSearcher.searchWithDrillDown()` for a more detailed code example performing drill-down.

7.2. Multiple Category Lists

The default is to maintain all categories information in a single list. While this will suit most applications, in some situations an application may wish to use multiple category lists, for example, when the distribution of some category values is different than that of other categories and calls for using a different encoding, more efficient for the specific distribution. Another example is when most facets are rarely used while some facets are used very heavily, so an application may opt to maintain the latter in memory – and in order to keep memory footprint lower it is useful to maintain only those heavily used facets in a separate category list.

First we define indexing parameters with multiple category lists:

```
PerDimensionIndexingParams iParams =
      new PerDimensionIndexingParams();
iParams.addCategoryListParams(
      new CategoryPath("Author"),
      new CategoryListParams(
             new Term("$RarelyUsed", "Facets")));
```

```
iParams.addCategoryListParams(
      new CategoryPath("Language"),
      new CategoryListParams(
             new Term("$HeavilyUsed", "Ones")));[1]
```

This will cause the Language categories to be maintained in one category list, and Author facets to be maintained in a another category list. Note that any other category, if encountered, will still be maintained in the default category list.

These non-default indexing parameters should now be used both at indexing and search time. As depicted below, at indexing time this is done when creating the category document builder, while at search time this is done when creating the search parameters. Other than that the faceted search code is unmodified.

```
DocumentBuilder categoryDocBuilder =
        new CategoryDocumentBuilder(taxo, iParams);
...
FacetSearchParams facetSearchParams = new FacetSearchParams(iParams);
```

A complete simple example can be found in package `org.apache.lucene.facet.example.multiCL` under the example code.

[1]Note that the term name has no effect, in that calling it "Heavily Used" is just for better readability.

8. Optimizations

Faceted search through a large collection of documents with large numbers of facets altogether and/or large numbers of facets per document is challenging performance wise, either in CPU, RAM, or both. A few ready to use optimizations exist to tackle these challenges.

8.1. Sampling

Facet sampling allows to accumulate facets over a sample of the matching documents set. In many cases, once top facets are found over the sample set, exact accumulations are computed for those facets only, this time over the entire matching document set.

Two kinds of sampling exist: complete support and wrapping support. The complete support is through `SamplingAccumulator` and is tied to an extension of the `StandardFacetsAccumulator` and has the benefit of automatically applying other optimizations, such as Complements. The wrapping support is through `SamplingWrapper` and can wrap any accumulator, and as such, provides more freedom for applications.

8.2. Complements

When accumulating facets over a very large matching documents set, possibly almost as large as the entire collection, it is possible to speed up accumulation by looking at the complement set of documents, and then obtaining the actual results by subtracting from the total results. It should be noted that this is available only for count requests, and that the first invocation that involves this optimization might take longer because the total counts have to be computed.

This optimization is applied automatically by `StandardFacetsAccumulator`.

8.3. Partitions

Partitions are also discussed in the section about Facet Indexing parameters.

Facets are internally accumulated by first accumulating all facets and later on extracting the results for the requested facets. During this process, accumulation arrays are maintained in the size of the taxonomy. For a very large taxonomy, with multiple simultaneous faceted search operations, this might lead to excessive memory footprint. Partitioning the faceted information allows to relax the memory usage, by maintaining the category lists in several partitions, and by processing one partition at a time. This is automatically done by `StandardFacetsAccumulator`. However the default partition size is `Integer.MAX_VALUE`, practically setting to a single partition, i.e. no partitions at all.

Decision to override this behavior and use multiple partitions must be taken at indexing time. Once the index is created and already contains category lists it is too late to modify this.

See `FacetIndexingParams.getPartitionSize()` for API to alter this default behavior.

9. Concurrent indexing and search

Sometimes, indexing is done once, and when the index is fully prepared, searching starts. However, in most real applications indexing is *incremental* (new data comes in once in a while, and needs to be indexed), and indexing often needs to happen while searching is continuing at full steam.

Luckily, Lucene supports multiprocessing – one process writing to an index while another is reading from it – in an admirable fashion. One of the key insights behind how Lucene allows multiprocessing is *Point In Time* semantics. The idea is that when an IndexReader is opened, it gets a view of the index at the *point in time* it was opened. If an IndexWriter in a different process or thread modifies the index, the reader does not know about it until a new IndexReader is opened (or the reopen() method of an existing IndexReader is called).

In faceted search, we complicate things somewhat by adding a second index – the taxonomy index. The taxonomy API also follows point-in-time semantics, but this is not quite enough. Some attention must be paid by the user to keep those two indexes consistently in sync:

The main index refers to category numbers defined in the taxonomy index. Therefore, it is important that we open the TaxonomyReader *after* opening the IndexReader. Moreover, every time an IndexReader is reopen()ed, the TaxonomyReader needs to be refresh()[1]ed as well.

But there is one extra caution: whenever the application deems it has written enough information worthy a commit, it must **first** call commit() for the TaxonomyWriter and only **after** that call commit() for the IndexWriter. Closing the indices should also be done in this order – **first** close the taxonomy, and only **after** that close the index.

To summarize, if you're writing a faceted search application where searching and indexing happens concurrently, please follow these guidelines (in addition to the usual guidelines on how to use Lucene correctly in the concurrent case):

- In the indexing process:
    1. Before a writer commit()s the IndexWriter, it must commit() the TaxonomyWriter. Nothing should be added to the index between these two commit()s.
    2. Similarly, before a writer close()s the IndexWriter, it must close() the TaxonomyWriter.
- In the searching process:
    1. Open the IndexReader first, and then the TaxonomyReader.

[1]TaxonomyReader's refresh() is simpler than IndexReader's reopen(). While the latter keeps both the old and new reader open, the former keeps only the new reader. The reason is that a new IndexReader might have modified old information (old documents deleted, for example) so a thread which is in the middle of a search needs to continue using the old information. With TaxonomyReader, however, we are guaranteed that existing categories are never deleted or modified – the only thing that can happen is that new categories are added. Since search threads do not care if new categories are added in the middle of a search, there is no reason to keep around the old object, and the new one suffices.

2. After a reopen() on the IndexReader, refresh() the TaxonomyReader. No search should be performed on the new IndexReader until refresh() has finished.

Note that the above discussion assumes that the underlying file-system on which the index and the taxonomy are stored respects ordering: if index A is written before index B, then any reader finding a modified index B will also see a modified index A.