A Skwish/Tika/Lucene Mashup

A small demo program using Skwish with Apache Tika and Lucene.

Table of contents

1. Overview

The example program discussed here demonstrates how you can use skwish in combination with such tools as Apache Lucene and Tika. In the Quick Demo article we discussed a simple program `FileImporter` for storing files in a Skwish-based "container". Here, we build on that sample program by running the files (Skwish entries) through Tika and storing the output directly in Skwish. We also index the container using Lucene and provide a crude web interface to search the contents.

Again, as mentioned elsewhere in this documentation, the example programs use an experimental HTTP server that ships with the library. You don't, of course, need to use that server in order to use Skwish in your project. We use it here because, well.., it's fun! (and performance is not too shoddy).

2. Download and setup

The example program is packaged separately from the distribution. It uses maven for build configuration. If you haven't already done so, download and install maven, first. Next download the file `skwish-eg-0.2.zip` from the project download page. (It's listed under **`Additional Files`** on that page.)

Unzip the contents of that file and build using maven. If you don't have Tika or Lucene installed in your local maven repo, the first build can take a while as maven resolves and downloads library dependencies.

3. Running the program

Run `TikLuc` first with the help option '-h'. This gives a brief description of the program and the arguments it accepts. The program's console output is displayed below. Classpath settings are omitted in the console listings below: set it to include the necessary libs as defined in the `pom.xml` file.

```
>$ java com.faunos.skwish.eg.tikluc.TikLuc -h

Usage:
 java com.faunos.skwish.eg.tikluc.TikLuc <storage dir> [-chw] [-i <lo>
<hi>] [source file/dir]*

 Arguments:
   <storage dir>       The storage directory. Required.
   [source file/dir]* Zero or more files or directories for import. Each
                      file or directory is imported in a separate thread.
```

```
Options:
  -c  Creates a new storage dir
  -h  Prints this message
  -s  Prints container statistics
  -w  Exposes storage contents via HTTP
  -i  Indexes documents in the meta id range lo, hi(exc)

Description:
  A Skwish demo program that imports files into a "storage" container.
  The container consists of 3 segment stores:
    1. cache. Contains file contents.
    2. tika.  Contains parser output.
    3. meta.  Contains meta information about the files.
  All segment stores are accessible through HTTP (option -w).

>$
```

Follow the procedures in the following subsections in the order they are presented. This is a demo program for a developer audience: it is not designed to be a bullet proof app! Once you understand what the program does, *then* go ahead and break it..

3.1. Importing files

Let's first create a "container" instance and put some files in it. Below we target a few well-populated directories. Pick some directories containing a lot of text in some form or another (e.g. html, source code, pdf, ms office, whatever) as inputs to the program. Type something like the following on the command line:

```
>$ java com.faunos.skwish.eg.tikluc.TikLuc -c tk01 ../doc ../src/ -w
```

Tip

You can combine command line options. For example, the above can also be written as `>$ java com.faunos.skwish.eg.tikluc.TikLuc -cw tk01 ../doc ../src/` Command line arguments are matched to options in the order that the options occur.

Like it's predecessor `FileStore`, the `TikLuc` program creates (writes) a new container directory `tk01`, starts a read-only web server on the container (port 8880), and imports all files under the `doc` and `src` subdirectories. As before, each argument is imported under a separate thread and is scoped within its own transactions.

Here, however, in addition to storing the raw contents of the files in the segment store called `cache`, we're then also feeding the contents into Tika in order to extract text and other meta data about the files. We pipe Tika's output directly to a separate segment store we call `tika`.

The files are not indexed yet, but we can browse them. Point your browser to an arbitrary entry in the "container":

```
http://localhost:8880/skwish/meta?id=336
```

The plain text response will look some like

```


uri=/doc/com/faunos/skwish/sys/mgr/TxnGapTable.html
cache.id=108
cache.tid=0
tika.txt.id=214
tika.tid=0
tika.link.id=215
tika.m.Content-Encoding=ISO-8859-1
tika.m.title=TxnGapTable
tika.m.Content-Language=en
tika.m.keywords=com.faunos.skwish.sys.mgr.TxnGapTable class
tika.m.Content-Type=text/html
tika.m.language=en
```

The above entry (your's will doubtless be different) contains meta information about a document named `/doc/com/faunos/skwish/sys/mgr/TxnGapTable.html`. It references other pieces of data using properties-format name/value pairs. Although not exactly spelled out this way anywhere in the actual demo program's code, entries in the `meta` store represent the root of our conceptual document model. So an entry's id in the meta store is taken to mean a document's unique identifier.

Again, as in the "Quick Start" example, you can manually browse the container. The actual file contents can be viewed at

```
http://localhost:8880/skwish/cache?id=108&tid=0&ct=text/html
```

although, relative URLs will be resolved incorrectly. (A google-cache-like hack is in the works to fix this, but is beyond what we're trying to demo here.) The text extracted by tika for this document can be found at

```
http://localhost:8880/skwish/tika?id=214&tid=0
```

and

```
http://localhost:8880/skwish/tika?id=215&tid=0
```

contains a sequence of hyperlinks Tika detected in the document.

3.2. Indexing the container contents

Stop the program (err, control-C, that is), and index the contents in a separate invocation.

```
>$ java com.faunos.skwish.eg.tikluc.TikLuc -wsi tk01 0 200000
```

This indexes the first 200000 documents in the container. (You probably threw in fewer documents in the previous step, but you get the picture.) When the console output indicates that the indexing job is done, point your browser to

```
http://localhost:8880/
```

and start playing with Lucene's impressive search capabilities. Use the `-i` option on the command line only when you want to index new documents (or after you have dropped the index by deleting the contents of the `tk01/lucene` subdirectory).

4. Remarks

Below, a number of observations regarding this demo program, and the approach taken here.

4.1. File import throughput

You'll notice that overall throughput during the file import phase increases considerably as the number of directories for import (command line arguments) is increased (within limits, of course). This may be because the "import" task is I/O bound, and since each command line argument is pushed into skwish under a separate transaction, each running under its own thread, when an I/O task blocks (as when Tika is pumping SAX events into skwish) another thread is free to do useful work. So if this speculation is right, skwish transactions can be seen as a performance booster in some scenarios.

4.2. Indexing speed

Lucene seems to index files faster when they come from skwish than directly from the file system. On small runs, say if indexing 5000 documents totaling about 27Mb of text (Tika output), throughput is at around 300 doc/sec. An "orders-of-magnitude" larger corpus of documents, of course, is necessary in order to establish benchmarks, and we should expect this number to degrade significantly on bigger runs. Still if it turns out there really *is* a performance advantage with the skwish approach at the indexing phase, then it shouldn't come as a total surprise. Here's why.

First, though we may index a lot of fields, we maintain very few stored fields. That job is off-loaded to skwish. Instead each Lucene document contains a few references to skwish entry IDs which in turn may be used to load the [conceptual] document's other fields. (The title, if present, in the search results, for example, comes from skwish, not the Lucene index.)

But perhaps more importantly, the real reason we might expect files coming from skwish to index faster than from the file system is that skwish I/O *should* be more efficient than opening and closing lots of files in a directory structure. And since the contents of consecutive documents stored in skwish enjoy better locality of reference than if they were to be accessed directly from the file system, skwish may be better suited for large I/O based batch jobs scenarios.

4.3. Append-only document model a la Lucene

Although not implemented in this demo, the approach we'd take here with regard to document updates is not that different than that taken with Lucene. Here, as there, we cannot *modify* document contents; instead, we must delete the old document and replace it with a new one. But because each document under our model consists of a graph of parts (skwish entries), some of these parts may be reusable in the new "replacement" documents. So, for example, to add a new field to a document, you replace its `meta` entry with a new one containing the old "fields" plus the added field and then "update" the Lucene index; the `cache` and `tika` entries remain untouched.

4.4. Orthogonality of Indexing, Storage and Content Model

Because the demo program does not maintain application state (i.e. documents/data) in the Lucene index itself, it's easy to re-index the data without losing a lot of pre-processing work. Conceptually, this is similar to tinkering with indexes on columns in an existing data set in an RDBMS: while the presence of an index in a relational database may represent a constraint on the content model, the index itself does not contain any data (at least, not from the user's perspective). So it is common for a database administrator to try different indexing strategies as the data and access patterns evolve. And using the approach taken here, one is able to do the same with Lucene. Nothing new here conceptually (a lot people already do this); just a means to that end.

4.5. Lazier stored field loading

The text search results of the demo program contain a link to the contents of the source document. We could have provided links to other parts of the content model--to the tika-generated links entry, for example. Conceptually, the binary contents of the resources pointed to by links in the search results can be considered *stored fields* of the document. Such externally stored fields can be lazily loaded on demand. Again, nothing new here as an *idea*; a hopefully good implementation.