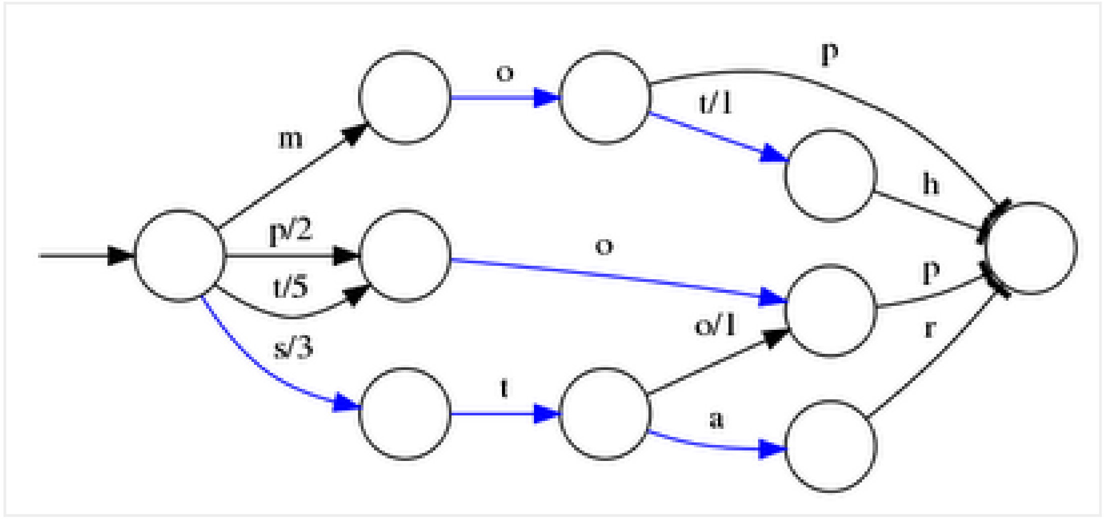
Changing Bits

Friday, December 3, 2010

Using Finite State Transducers in Lucene

FSTs are finite-state machines that map a term (byte sequence) to an arbitrary output. They also look cool:



That FST maps the sorted words *mop*, *moth*, *pop*, *star*, *stop* and *top* to their ordinal number (0, 1, 2, ...). As you traverse the arcs, you sum up the outputs, so *stop* hits 3 on the **s** and 1 on the **o**, so its output ordinal is 4. The outputs can be arbitrary numbers or byte sequences, or combinations, etc. -- it's pluggable.

Essentially, an FST is a SortedMap<ByteSequence,SomeOutput>, if the arcs are in sorted order. With the right representation, it requires far less RAM than other SortedMap implementations, but has a higher CPU cost during lookup. The low memory footprint is vital for Lucene since an index can easily have many millions (sometimes, billions!) of unique terms.

There's a great deal of theory behind FSTs. They generally support the same operations as FSMs (determinize, minimize, union, intersect, etc.). You can also compose them, where the outputs of one FST are intersected with the inputs of the next, resulting in a new FST.

There are some nice general-purpose FST toolkits (OpenFst looks great) that support all these operations, but for Lucene I decided to implement this neat algorithm which incrementally builds up the minimal unweighted FST from pre-sorted inputs. This is a perfect fit for Lucene since we already store all our terms in sorted (unicode) order.

The resulting implementation (currently a patch on LUCENE-2792) is fast and memory efficient: it builds the 9.8 million terms in a 10 million Wikipedia index in ~8 seconds (on a fast computer), requiring less than 256 MB heap. The resulting FST is 69 MB. It can also build a prefix trie, pruning by how many terms come through each node, with even less memory.

Note that because addition is commutative, an FST with numeric outputs is not guaranteed to be minimal in my implementation; perhaps if I could generalize the algorithm to a weighted FST instead, which also stores a weight on each arc, that would yield the minimal FST. But I don't expect this will be a problem in practice for Lucene.

In the patch I modified the SimpleText codec, which was loading all terms into a TreeMap mapping the BytesRef term to an int docFreq and long filePointer, to use an FST instead, and all tests pass!

There are lots of other potential places in Lucene where we could use FSTs, since we often need map the index terms to "something". For example, the terms index maps to a long file position; the field cache maps to ordinals; the terms dictionary maps to codec-specific metadata, etc. We also have multi-term queries (eg

Prefix, Wildcard, Fuzzy, Regexp) that need to test a large number of terms, that could work directly via intersection with the FST instead (many apps could easily fit their entire terms dict in RAM as an FST since the format is so compact). The FST could be used for a key/value store. Lots of fun things to try!

Many thanks to Dawid Weiss for helping me iterate on this.

Michael McCandless on 12/03/2010

Share 1

18 comments:

Michael Kleen May 24, 2013 at 10:53 PM

Hello, what does the fst with 69 mb for wikipedia contains ? Only the titles of the entries or the whole text as well ?

Reply

Michael McCandless May 25, 2013 at 10:51 AM

Hi Michael,

That FST held all unique terms (tokens) from indexing all English Wikipedia content.

Reply

kbros August 1, 2013 at 8:08 PM

Hi Mike,
As the FST has a high CPU because of seeking, how bad are its performances affected by multiplying the number of existing terms?

Do you have an estimation of how is a typical query time distributed between the FST, seek in the term dictionary, calculating frequency or positions? How can I profile that in my own index?

Thanks!

Reply

Replies

Michael McCandless August 2, 2013 at 9:48 AM

Hi kbros,

The FST seek time is typically a tiny part of the overall query execution time; usually most of the time is spent iterating through the postings once the lookup is done in the terms dict. Only terms-dict heavy queries (e.g. FuzzyQuery, DirectSpellChecker, WildcardQuery) will really stress the terms dictionary.

That said, we have a Google Summer of Code project now working on using an FST to hold all terms in the terms dict, which then relies much more on the FST performance ... https://issues.apache.org/jira/browse/LUCENE-3069 has the details.

But if you are worried about it, run a profiler and report back the results! There are always things that need fixing :)

Alessandro Benedetti October 1, 2013 at 10:58 AM

Hi Michael,
it seems that the result of the Google Summer of Code project has been committed.
Having a FST holding the complete Term Dictionary I think could be useful, for example in the SpellCheck scenario, to make a quick intersection between the Levenstein Automaton and the Index Terms Dictionary .
Am I right ?
Also in the Term component scenario, i think it can give us improvements ...
Can you briefly summarize me where do you think to use the FST Term Dictionary ? :)

Cheers

Michael McCandless October 3, 2013 at 7:28 AM

Hi Alessandro,

Unfortunately, I believe the FuzzyQuery performance was slower with the new FST-based terms dictionary, but yes, that is the idea: it ought to be faster.

DirectPostingsFormat, which stores all terms AND postings in RAM, does give a good speedup for fuzzy queries, so a DirectSpellChecker implemented on that would be faster. However, this format is VERY RAM consuming, i.e. it makes no effort to "compress" things (unlike FST).

I think there is room for a happy medium, e.g. pull out the terms dict from DirectPostingsFormat, maybe compress it a bit (but not as much as FST), and I think we'd see good speedups for terms-dict intensive operations.

We may even want to implement the Levenshtein intersection without the intermediate automaton, i.e. "on the fly", because constructing that automaton takes non-trivial time.

Reply

Alessandro Benedetti October 3, 2013 at 12:56 PM

Thank you Michael for the quick answer,
For the intermediate automaton are you referring to the Levenstein Automaton, accepting all the strings with the expected distance from the query term ?
So directly building an automaton, accepting only Index terms with the expected distance from the the query term ?

Reply

Replies

Michael McCandless October 4, 2013 at 7:06 AM

Yes, that automaton.

But it turns out you don't have to fully build it up front, in order to do the "intersection". You can expand it as-you-go, which in theory saves time because then you only expand the parts that your terms dictionary would ever have encountered. It's sort of like a lazy automaton building ...

Reply

Alessandro Benedetti October 4, 2013 at 7:15 AM

Yes , it makes sense and sounds reasonable :)
But is it feasible ?

I am starting learning about FSA and FST now, so I'm not yet into the implementation,
but one thing I remember is that Lucene FST implementation is so good and compact because it creates an immutable FST (that becomes a sort of byte array under the hood) .
So creating an expandable FST is already possible or it will be a challenge ?
Can you suggest me other material to study ragarding this really interesting topic ? (I followed your Solr revolution conference, some blog posts and videos so far :))

Reply

Replies

Michael McCandless October 11, 2013 at 7:48 AM

Hi Alessandro,

Maybe read the original paper for the FST building algorithm we implemented in Lucene? http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.3698

But note that the Levenshtein automaton that we build/intersect for FuzzyQuery is not an FST; it's an Automaton (Lucene has two separate packages for these). I suspect by not pre-building the automaton, but rather expanding it on the fly, we could see some speedups in certain cases; not sure how much.

Reply

Anonymous March 7, 2014 at 4:01 AM

Hey Mike,
Are we using fst to make inverted index? If yes, where the output of traversing a string in fst pointing to?

Reply

Replies

Michael McCandless March 7, 2014 at 6:27 AM

FSTs are used only in certain places, e.g. the terms index (held in RAM, mapping term prefixes to blocks on disk), dictionaries (hunspell), suggesters, synonyms, etc. They are also used in an optional MemoryPostingsFormat, where each term and its postings are held in a big FST (good for primary key fields). We also more recently added a terms dict that holds just the terms + pointers to on-disk postings, in an FST.

What the output contains / where it points to varies drastically with each of these uses... really it's generic and can be whatever you need it to be.

Anonymous April 10, 2014 at 8:00 AM

I am curious what is the performance of FST. And its possible to map the words with id they occur in (a simple typeahead). and this will output the ids of document in which word occur can we somehow emulate the ranking formula (Similarity, or bmk21) and bake that in the FST, of course it would not be that simple but is it possilbe. Lets say we control the preprocess of words i.e know how many times the word appeared in the doc know how times word appeared in all docsids and then
bake that in some fst`s like wordFST scoringFST and so on. is this approach will benefit over traditional, and as i red there is no incremental way of updateing a fst i has to be rebuild from ground up.

Michael McCandless April 10, 2014 at 9:55 AM

Yes, you can encode that into an FST, and e.g. Lucene's MemoryPostingsFormat does exactly that. But this is not a good fit for fields where each term may occur in many docs, because the output byte[] in that case can get quite large.

Anonymous April 10, 2014 at 10:11 AM

i see thanks. I am almost sure you have seen the talk Michael Butch(sorry if i misspelled his name) gave about their implementation of memory postings in lucene and recent improvments that can lead them to be merged back in lucene, wondering is there initial talk about that.

Reply

Anonymous September 14, 2014 at 9:54 AM

I was wondering whether it is possible to associate ids with recognized strings in the automata. Say for instance I have a lexicon and each of its entry has a unique integer id. I'd like to compile the lexicon as an union string automata and whenever it recognizes an entry in a string, it outputs the unique id. It's like using the automata as a set of primary keys.
I could not find such an example of use online.
In a previous comment you mention MemoryPostingsFormat begin able to output ids (but document id), would it be possible to use that ? Thanks

Reply

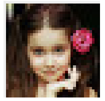
Replies

Michael McCandless September 15, 2014 at 4:24 AM

Hi Anonymous,

Associating a String to an arbitrary (including Integer) output is precisely what FSTs are for, so that use case works easily. Try looking at TestFSTs.java for an example?

Reply



Clifton Jacobs February 11, 2015 at 5:12 AM

Thats astonishing...

Reply

Enter your comment...

Comment as: Select profile...

Publish Preview

Links to this post

Create a Link

About Me

Michael McCandless

Follow 827

Michael loves building software; he's been building search engines for more than a decade. In 1999 he co-founded iPhrase Technologies, a startup providing a user-centric enterprise search application, written primarily in Python and C. After IBM acquired iPhrase in 2005, Michael fell in love with Lucene, becoming a committer in 2006 and PMC member in 2008. Michael has remained an active committer, helping to push Lucene to new places in recent years. He's co-author of Lucene in Action, 2nd edition. In his spare time Michael enjoys building his own computers, writing software to control his house (mostly in Python), encoding videos and tinkering with all sorts of other things.

View my complete profile