Quoting Hiroshi Fujinoki <hfujino@siue.edu>:

> Alex:
>
> In your document for performance evaluation (I attached the document to
> this e-mail - it is what I received from you):
>
> (a) Does an acronym "IR" (in section 1.1) mean "information retrieval"?
>
> (b) Does the term, "relevancy scoring" (in section 1.2.1) mean the same
> things as "proximity scoring" (in section 1.2.2)? If not, can you tell me
> the difference between the two terms?
>
> Thanks and see you at 5:00PM this afternoon.
>
> HF

(a) Yes, IR = information retrieval.

(b) No, relevancy scoring encompasses two types of relevancy measures: proximity and keyword/term weighting.

(1) Keyword/term weighting heuristics, commonly used in IR:

Rare terms in a document COLLECTION are more relevant. (More discriminating.) For instance, if a term in the query only matches one document in the entire collection, that increases the importance of the document.

When a query term matches a frequent term in a single document, that increases the importance of the document, e.g., if user a user searches for “fast”, documents which have multiple occurrences of “fast” should be more relevant than documents that have fewer occurrences of fast.

Both of these insights will be used. Our approach:

After reviewing IR literature I discovered there is an empirically well-performing (on typical document collections) family of heuristics known as idf-tf. I will choose one or more from variants from this family. This is the easy and fast way to do relevancy scoring.

(2) Proximity scoring. This is more challenging; it is slower, there isn't as much of an established approach (as opposed to term weighting), and it blows up the index size more.

I am experimenting with different methods. There are many possibilities. I have considered the idea of using multiple methods, and learn weight coefficients for their respective scores through training/machine learning. This is really the best approach, but I doubt we have time for it.

Anyway, we have two relevancy measures: “proximity” and “keyword/term importance”. Our relevancy tests will try different variations of this. Likely, the best one will combine both of these types of relevancy measures, in which case we will probably do something like a weighted average, e.g., alpha * proximity_score + (1-alpha) * keyword_weight_score, where 0 <= alpha <= 1.

Also, I am experimenting with different methods on how to represent proximity (block granularity) data.

(a) A compact postings list for each term using a binary string M bits long. If position k is 1, then block k has the given term. Frequent terms, like "the", if not removed (using stopwords) will be a binary string of M 1's. Infrequent terms will have a binary string with few 1's and many 0's. This works pretty well, in terms of space-efficiency. There is a "preferred words per block" option (I usually use 250 words/block, since that is the average page size), giving us a page-level granularity. If the document is especially large, like an encyclopedia, this will however cause a huge binary string to be needed for each term in the index. Thus, I also have another configuration option, "max blocks per document", which will scale the words per block up if the preferred words per block would create too many blocks.

    I usually use a max words per block of around 50 to 150. This range creates space-efficient indexes, and also allows the postings list to be iterated more quickly.

    Btw, for the binary string representation, if there are a large number of blocks, most terms will have sparse binary strings (mostly 0's), which means they are very compressible. Using stock disk compression (in Windows 7) compresses them quite a bit, and this is a very non-aggressive compressor.

(b) An issue with approach (a) is that many algorithms for postings lists use a vector of blocks/positions. I can certainly construct such a vector from the binary string, but a question of efficiency presents itself.

    I am in the process of implementing the vector of blocks postings list (like in that one paper on inverted indexes). For most documents, this will create a larger index (larger file size and larger memory footprint), but it'll probably be faster assuming it can fit in memory (and maybe even if it can’t?). We’ll see. I want to use best-of-breed algorithms whenever possible since I want the secure index database to scale up to very large databases.

Other concerns

Presently, I allow an arbitrary number of bits to be used to represent each term (which controls the false positive rate) and the number of bits to use in the binary string to represent the presence of terms in blocks.

Typically I use 10 bits (for ~ 0.001 false probability rate) to represent terms and typically a max of 100 bits to represent blocks for a total of 110 bits per term plus serialization overhead of the perfect hash. Small documents will use very few blocks, e.g.,5 to 10 blocks, which means 20 bits per term. Terms are typically unigrams and bigrams (bigrams can be used to search for an arbitrarily large phrase using a biword model, e.g., if the query is "hello doctor fujinoki's family", it will be decomposed into the following bigrams: "hello doctor", "doctor fujinoki", "Fujinoki family". All of these terms must exist in a single block to be considered a hit (phrases that span multiple blocks can be accounted for too).

So, this is a very compact representation. The perfect hash size is not down to the theoretically optimal size, but I am using the very best perfect hash construction that has been devised so I can't do much about that. Still, it's pretty small.

It's an especially good fit for when you allow for a non-minimal perfect hash. A minimal perfect hash for N members maps every member to a unique index AND there are only N index positions. A non-minimal perfect hash maps every member to a unique index, but there are more than N index positions. I have this parameterized (loading factor) – what percentage of index positions are not for a real member of the document? Since this doesn't really increase the file size much (sometimes it decreases it), AND its valuable to us to obfuscate the terms of the index (there is no way for an attacker to determine which index positions are for actual document members, and which aren't – I call this "polluting" the secure index; it doesn't increase the false positive rate any, and it causes less information about the index to be leaked), the non-minimal perfect hash is a good fit for our needs.

A problem with this entire approach is the fact that I'm not using byte-aligned data structures. It's space efficient (thus may fit secure index databases in memory), but non-byte aligned operations are much slower. Thus, I am also working towards, if time permits, a byte-aligned version of this, which will generally be larger and less customizable, but faster assuming it can fit in memory.