Thank you for the comments. I read your comments, and I agree with the points you've brought up.

Let me think about it some more and, later this evening, I will respond to them in more depth.

Metrics, refined

I'd also like to report some interesting results. I'm still in the process of refining what problem I'm trying to solve. Before, I was trying to solve a very complicated question which tried to answer the question, "How close is this search phrase to matching this approximate bloom document?" And this involved all sorts of distance metrics and state-space exploration. That's still an interesting, but I fear it won't work well in practice to give users the results they need to help them find what they're looking for. It may also be unreasonably slow for large queries.

Today, I've reduced the problem I'm solving (for now) to this:

a. For a given bloom document, how well does it match the set of keywords in the search query?

   Botton line: I'm now operating on keywords indivisibly.

   For example, if a user searches for "alex hello world", it will search for the following keywords *"alex", "hello",* and *"world"*. A match on *"alex towell"* exactly is no better than matching *"alex"* and *"towell"* separately. If the user wanted one of the keywords to be "alex towell", then he needs to do a keyword search for '"alex towell" hello' instead of 'alex towell hello'.

b. For a given block in a given bloom document, how well does said block match the keywords in the search query? This is the same principle as in (a), above, but we'll be using different metrics and weights to rank a block as opposed to an entire document.

So, here's my formula for the rank of a block within a bloom document. It's pretty simple.

First, for each keyword in the search query, we give it a weight defined by the function:

$$weight(keyword)$$

An interesting weight metric I discovered (it is similar to the previous weight I had come up with on my own--reciprocal of probability--but seems to have more empirical grounding) is [1]:

$$weight(keyword \mid bloom\ document) = -\log\frac{n}{N}$$

Where n is the number of blocks in the given bloom document for which the keyword is in, and N is the total number of blocks in the given bloom document.

Examples:

if N=16 blocks, and n=4 blocks which have keyword: weight = -log(4/16) ~ 1.4

if N=16 blocks, and n=8 blocks which have keyword: weight = -log(8/16) ~ 0.7

if N=16 blocks, and n=1 blocks which have keyword: weight = -log(1/16) ~ 2.8

Appropriately, the last example, in which the keyword was the least common among the blocks, was given the highest weight of importance.

Now, for a given block, **B**$_i$, we don't only wish to find which keywords, weighted by their respective weights, a block has, but also we would like a proximity measure. Put simply, how far away is each keyword from a given block? This too is a function:

$$distance\left(keyword \mid block = B_j\right)$$

A reasonable candidate distance function, then, is:

$$\begin{aligned}&distance(keyword \mid block = Bj)\\ &\qquad = [location(Bj) - location(nearest\ block\ to\ Bj\ with\ keyword)]^2\end{aligned}$$

Where location is a function which maps a block to a position, e.g., location(block) = index of block *(assuming a block's index maps to the actual order of the blocks in the bloom document)*

Now, to determine the cost of a block for all the keywords, we do the following:

$$cost(keywords) = \sum_{keyword \in keywords} weight(keyword) \times distance(keyword\ |block = B_j)$$

We can then normalize this result to get a value between 0 and 1. The worst-case for a block is when it is a maximal distance away from all of the keywords. In our example distance and location functions, this is simply a distance of (N-1)^2. So:

$$\begin{aligned}&normalized\_cost\left(keywords \mid block = B_j\right)\\ &\qquad = \frac{\sum_{keyword \in keywords} weight(keyword) \times distance(keyword\ |block = B_j)}{(N-1)^2 \sum_{keyword \in keywords} weight(keyword)}\end{aligned}$$

And, so, to get a block's ranking, we just do 1 – normalized_cost(keywords | block). 1 is a perfect rank to the keywords, and 0 is the worst-possible rank.

Now, if we want to rank entire bloom documents (to compare two or more bloom documents), we need to pay attention to the fact that some of the keywords mentioned in a query may not exist in a bloom document. So, now, let's rank whole bloom documents in the same way we ranked blocks in a single bloom document:

$$weight(keyword \mid set\ of\ bloom\ documents) = -\log\frac{n}{N}$$

Where n is the number of bloom documents which have the keyword, and N is the total number of bloom documents. If a keyword is in *none* of the bloom documents, then we ignore that keyword (like we did when we ranked blocks in a given bloom document).

However, this loses the notion of proximity. So, let's instead modify our weight function for blocks in a particular bloom document. In the previously mentioned cost function for a block, my example weight function only operated correctly on keywords which were present in at least one of the blocks. However, the weight function could work on non-existent keywords too, e.g., weight(keyword | bloom document) = MAXIMUM if keyword is not in the bloom document. If we do this, then a block in one document can be compared to a block in another document directly. And, indeed, the whole document can be assigned a weight by summing up the cost of all the blocks, as previously defined but with the new weight function, then dividing that total sum by the number of blocks. Do note that this approach to whole document ranking may "unfairly" penalize large documents (or documents with more blocks, depending on the distance function used). I see work-arounds to that, but it's not clear a work-around is desirable.

Anyway, I think I'm narrowing in on good candidate solutions to this problem. I'll try multiple metrics, but I think the interesting point is that *all* sorts of metrics can be applied, seemingly usefully, to the limited (approximate) knowledge provided by the bloom filter.

[1] http://www.soi.city.ac.uk/~ser/papers/RSJ76.pdf

User interface

So, I've been thinking about how someone (or some group) will use an example application of our bloom filter. We want to adhere to these design goals:

- confidentiality: who should have access to which parts of the document
    - Accomplished via encryption (hybrid design using asymmetric and symmetric ciphers)
- authorization: what permissions does a user have for working with the document (encryption)
- accountability: what has a user done with the document (1-way hashes, signatures)
- integrity: how do you know if the document has been altered? (1-way hashes, signatures
- authenticity: how do you know where the document came from? (1-way hashes, signatures)
- non-repudiation: can the signatory deny having signed the document? (1-wa hashes, signatures)

I've almost worked out, I believe, the structure to accomplish this, to varying degrees of success. It would be easier if we could trust a single source for the document (e.g., a trusted server), but that doesn't jive well with the capability of our bloom documents to be spread around to many people in a decentralized way. All of these objectives can be met, to varying degrees, in a way that does not require any centralization. That is, everyone can have an independent copy of the document and work on it in a decentralized, independent way, then combine their results later. Merging the same block can be done using merge tools, and merging different blocks into the same document can be done trivially (since the bloom document is already a logical file consisting of separate ordered blocks).

I'm leaning towards the use of a set of simple command line tools, where each command line tool is a simple, single-task-oriented program. I would also like to re-use as much existing infrastructure as possible. So, for instance, OpenPGP for key management and DSA for digital signatures.

I'm still working on the file format and how exactly I want to go about doing this. An example of command line utilities I'm considering include, but are not limited to: "bloom_make", "bloom_block_insert", "bloom_block_remove", "bloom_block_replace", "bloom_block_get", "bloom_block_check_signature", "bloom_filter_set_visualize", "bloom_filter_set_query". Each of these programs will generally require public keys and/or symmetric keys to use appropriately.

```
File format

--------------

Bloom
blocks: List[Block]
masterPublicKey: PublicKey
Block
      // a log table of who did what to this block of the bloom document
      logTable: LogTable = …

      // who should be given the symmetric keys?
      // for each authorized user, encrypt symmetric key with his public key
so only he can access it
      authorizedList = List {
            AuthorizedUser1  =
                  publicKey: PublicKey
                  symmetricKeyEncrypted: String = E_rk(symmetric_key)
            AuthorizedUser2 = …
            …
      }

// message should be encrypted with the symmetric key currently assigned to
it
encryptedMessage: String = E_k(message)

// if E_uk(encrptedMessage.privateKeyEncrypted)
//    == hash(sig.publicKey | logTable | authorizedList | encryptedMessage),
//    then block state is same as the person who signed it saw it at the
time; you trust him?
      sig =
publicKey: PublicKey
privateKeyEncrypted: String = E_rk(hash[masterPublicKey | publicKey | logTable
|
authorizedList | encryptedMessage])
```

The really high-level overview is that, a bloom document consists of 1 or more independent blocks. Each block can be assigned a symmetric key for confidentiality. Each block also optionally has a set of public keys for people who should always be able to access the block (authorization), even if the block changes its symmetric key. To do this, we just encrypt the symmetric key for the block to some people's public keys so only they can open it to see what the symmetric key is. Any time a user makes a change to the document, he or she signs it by using his or her private key (public key cryptography) to encode a hash of the encrypted message, the log table, etc. (the state of the block) which can be decrypted by others using the public key corresponding to that private key, and checking to make sure that this decrypted string is equal to the hash of the block's current state. If it's not, then it has no valid signatory and probably shouldn't be trusted. Also, if it is equal, but it's signed by someone you don't trust (see, for example: web of trust), then you probably shouldn't trust it.

Also, a block in this sense is actually not the bloom filter block levels. These blocks are blocks which represent the (encrypted) content of some part of the file. Each block can be worked on independently, so that a large document can be decomposed into a number of smaller blocks with varying degrees of security.

If you don’t trust someone in a block’s public key list (or you don’t trust the master key) – e.g., maybe they are not in your web of trust (as defined by existing open-source frameworks for this), then you can fork the document and revoke their access and edit the block in that forked document instead. In this way, we have secure, distributed editing of documents with confidentiality, non-repudiation, etc.

There will also be tools to make a bloom filters for a block for which you have proper access to, using the same security mechanisms for non-repudiation, confidentiality, etc. These bloom filters can then be distributed independently from the actual bloom document, which makes sense in the case where the document/block shouldn’t even accessible to all but a specific group of people, so you just want to give everyone else an approximate, query-able version of the document. These bloom filters can also be attached to the bloom document so that, for instance, if each block has attached bloom filters, then the whole document can be queried through these bloom filters. There can be varying levels of access, as you described, and if a block changes, only the bloom filters for that block need to be updated.

My question to you is, does this seem reasonable? Or is it not very relevant towards my thesis? This is an example application that seems to mesh well with the bloom filter document representation, but maybe it’s too much or over-complicated? (I’m far from being confident in my solution to the security framework, also – getting distributed, decentralized security right is really tricky.)

I know that’s a lot to take in. I certainly don’t expect you to get back to me anytime soon on this. My biggest update was in the first part, “Metrics, refined.” The user interface stuff is secondary, but I’m just trying to work out a good example for it.