Research / survey

# our research

### hash functions

Another way to change the hash function is to map a character to a different value, e.g., a code lookup table. It's only going to have two columns (one for the original character, and one for the integer to map to), and Z rows, where Z is size of alphabet, e.g., a through z or a through z + 0 through 9, etc.

See: Mapped Additive Shift Hashing (MASH) in paper "Extremely Fast Text Feature Extraction for Classification and Indexing"

Now, we could have a single hash function, but change the lookup table. This is also easy to modify in the context of learning parameters, e.g., a stochastic gradient descent may try some slight tweaks to the table to see if it results in better performance (fewer false negatives for a given bloom filter size, or better spread among bloom filter indices such that its more difficult to gain information by looking at index values and hash functions).

Maybe it's not even necessary to have more than one hash function? Learn good parameters for one hash function on the training data, and we're done? I don't know. It might be easier to gain information about the bloom filter though.

false negatives

distribution

In the context of bloom filters, distribution is this: for a set of words, does each word map to independent indices? If so, then a statistical analysis can be performed to infer properties of the document, even if each word unit (1-gram, 2-gram, ..., k-gram) in the document is pre-encrypted before hashed (pre-encrypted for query privacy).

So, this is another performance metric to optimize: we want member indices to overlap to some degree while still maintaining a low probability on false positives. The hash functions, in this way, serve as a form of confidentiallity even without pre-encryption since you may not be able to infer much about what members (or how many) are present by looking at the bloom filter.

## negative examples

Use negative examples from a document repo. This may unfairly advantage my approach, but ...

---

Co-occurrence can be extracted from document sources using ngram analysis.

Co-location can also be extracted from ngram analysis, e.g.:  'crystal clear', 'middle management', 'nuclear family', and 'cosmetic surgery'

## query statistics

From "http://en.wikipedia.org/wiki/Web\_search\_query"

"Research has shown that query term frequency distributions conform to the power law, or *long tail* distribution curves. That is, a small portion of the terms observed in a large query log (e.g. > 100 million queries) are used most often, while the remaining terms are used less often individually."

Evidence in support of my supposition that to train the bloom filter, we can focus on training it to not respond with a false negative on probable queries.

Source: http://link.springer.com/chapter/10.1007%2F978-3-540-31865-1\_2

However, also:

But in a recent study in 2011 it was found that the average length of queries has grown steadily over time and average length of non-English languages queries had increased more than English queries.[8]For longer queries, Natural language processing helps, since parse trees of queries can be matched with that of answers and their snippets.[9] For multi-sentence queries where keywords statistics and Tf–idf is not very helpful, Parse thicket technique comes into play to structurally represent complex questions and answers.[10]

## vertical search?

If vertical search, can learn more effective bloom filters?

http://en.wikipedia.org/wiki/Vertical\_search

## relevancy and visualization

A form of relevance feedback: http://en.wikipedia.org/wiki/Relevance\_feedback

Given a document store D, which documents are most relevant to the information need of the user? (Information need is extracted from the query.) Return a set of documents, d, ranked by relevancy that satisfy the query, e.g., tf-idf or other keyword weighting approaches.

Once we have a set of ranked documents, visualization is a further refinement to satisfy the information need of the user. This refinement stage is especially important in the context of our research since the document itself may not be immediately accessible (encrypted), and acquiring access may be costly (declassification). Thus, this visualization step may be important as the user can consider what he's looking for and visually pick out candidates that seem like a better match towards his potentially ill-defined information needs.

This visualization step can also serve to refine his information need to a specific block within a given document. Since the document itself may be sensitive (e.g., trade secrets of a company or classified government documents), being granted access to the entire document may be a hard sell. It may be an easier sell if the user only asks for access to some subset of the document, e.g., 4 blocks (blocks 3 to 7) out of 200 blocks.

### refinements

tweaking keyword weights

Let the user tweak the keyword weights while visualizing relevancy to try to narrow in on fewer blocks.

This is based on Rocchio algorithm's concept of re-weighting the terms in a vector space model. Basically, it's a form of nearest centroid classification, in which some of the documents are classified as belonging to the class (relevant to query) and others not. Then, move the centroid (or centroids) around to change the class. Move them in a direction guided by the user such that, through refinement by the user, a different set of documents compared to the initial set is included in the result set.

## applicable?

* query expansion

Query expansion is designed to increase the recall of the set of documents retrieved. That is, return more of the documents relevant to the user's information need. This may reduce the precision, e.g., ratio of number of relevant documents retrieved to the number of documents retrieved.

Difficult is more difficult to apply to our work since query expansion often relies upon words near to the query words in the source document, but since the source document is concealed (we do not see the document as a bag of words; rather, the document is like a hidden bag of words, and we only know a word is in the bag if we guess a word, and then we are told whether said word is in the bag. That said, we can still do query expansions like if query = "hello world" then an expansion might be "(hello OR hi OR ... OR greetings) AND (world OR planet OR ... OR earth). Of course, now we must be careful about not expanding it to terms which have the wrong sense; this is a natural language problem, e.g., sense disambiguation.

With regards to being more difficult to apply: this is both an advantage and a disadvantage. It is a disadvantage in the context of satisfying a user's information needs; it is an advantage in the context of satisfying a document's security needs. So, a trade-off.

* vector space model

Same reasons as for query expansion. Vector space model represents document and query as a vector, and then does simple linear algebra (e.g., angle difference) on them to score the relevance of a given document to a given query.

We do not have a bag of words in our bloom filter readily available. We have only a yes/no answer to whether a given term is a member of a bloom filter. We can, of course, do a bunch of such queries to CONSTRUCT an approximate bag of words and then apply the vector space model to the problem. However, this approach will miss terms (especially non-common terms which a dictionary wouldn't include, for instance) and, of course, falsely include terms that shouldn't be present.

More over, this is a slow way to do it. Every query is now expanded to N-queries, where N is extremely large.

It is, however, reasonable to construct a vector from the terms in a query, e.g., if a query is "hello world" and the document contains "hello", then:

query vector = <hello=1, world=1> = <1,1>, and document vector = <hello=1,vector=0>=<1,0>. Angle difference is, then, 45 degrees. Furthermore, we can weight the terms, e.g., if world is given a weight twice as large as hello, then:

<hello=1, world=2> \* <hello=1, world=0>, then angle is 63 degrees. This is larger than 45 degrees, which makes sense since we're weighing "world" more than "hello", and the document doesn't contain "world".

* navigational queries
  + not applicable to our research?

## probability of inferring words from bloom filter indices

If the hash functions do not uniformly distribute words to indices, e.g., some words have indices all to themselves, then this is a vulnerability in our solution. During training, we an also check to make sure that no word in the training set maps to indices that no other word in the training set maps to, i.e., there is no overlap.

## preprocessing

### semantic compression

## hybrid

### bloom filter for quick dirty lookups

if a hit occurs

use conventional lookup

could be a false positive

maybe need to request full document or index?

if no hit occurs

bloom filters: no false negatives

ignore

### other approaches

Take the words in the document, then encrypt them individually. You could expose this bag of encrypted words.

Problems:

(1) Frequency analysis. You can determine which encrypted words are most common.

(2) Only keyword matching. Approximate matching is expensive (try variations of a keyword, see if it matches). Also, cannot do multiple-keyword (n-gram) matching unless you also encrypt those. Again, expensive, and variations are now more problematic.

(3) You can only search it if you have the key. What if you want anyone to be able to search it?

(4) Sequential search. Unless you create an inverted index on these encrypted words.

Solutions?

... this text is obsolete I think. Revisit.

## new big ideas

### limited regular expressions

A limited form of "regular expression" searching. E.g., have two classes, # or & for numeric or alphabetical respectively. A third class is \* for alphanumeric. Three classes, then. Ok, so, for each character position in a term, insert an '&' and '\*' in addition to the original character if the character is alphabetical, and insert '#' and '\*' and the original character (numeric) if the character is numeric. That means, for a term with k characters, there are 3^k variations. This is fairly poor behavior, e.g., 3 character words have 3^3=27 variations, 4 -> 81, 5 -> 243, ..., 10 -> 59049.  If we restrict it to just one class, \* for any character, then we instead have 3->8, 4->16, 5->32, ..., 10->1024. More reasonable, but still grows quickly. As long as k is reasonably small, we may be able to train a bloom filter effectively. Of course, we can also just tolerate a very high probability of a false positive on regular expression searching; after all, if the user fails to find what he wants with a normal search, then the regular expression search may be used as a last ditch effort, where higher false probabilities may become more acceptable in turn for improved access.

Another alternative is to limit the number of variations, e.g., a term can only include a maximum of k \*'s. So, now we've reduced the problem down to, for a term with n characters, (n choose k) \* 2^k. This is much more reasonable, and still provides good regular expression matching.

### one-way pre-hashing w/secret

Encrypt terms before inserting into bloom filter. This protects query privacy in addition to data privacy. Another solution to query privacy: a one-way hash. Faster, secure if we assume the one-way hash has the property that it is hard to discover a string which has the given hash value (pre-image resistance). If we use a one-way hash, and if the algorithm (for hashing) is known, then it is easy for common words to be discovered. A solution to this is a secret, so we hash instead OneWayHash(keyword | secret). This is fast and secure as long as secret is secure.

If secret is revealed...what to do? Bloom filter is already constructed using the secret. There is a solution not involving reencryption in the paper "Shared and Searchable Encrypted Data for Untrusted Servers", but it seems overly complex and has already been done. I trivial solution is to remake the bloom filters with a new secret, but this is admittedly expensive -- the one-way hash may be fast, but relearning the bloom filter is problematic. Maybe, however, all we need to do is remember the properties of the bloom filter -- hash functions and size -- and just remake them with the same properties but a different secret. Since the one-way hash is fast, the whole operation may be fast. But, this undermines the need for learning... or does it?

### subset checks to reduce false positives

For a compound term "A B" to be in the bloom filter, terms "A" and "B" must independently be in the filter. For a compound term "A B C" to be in the bloom filter, terms "A", "B", "C", "A B", "B C" must independently be in the bloom filter. In this way, we can gain confidence in a compound terms' membership by checking if some sampling of subsets are present too. If not, we know "A B C" is a false positive. If all subset checks are true, then "A B C" may still be a false positive, but the probability of it being a false positive has diminished. I believe I can formalize this exactly using mathematics.

k = number of hash functions

n = number of members inserted

m = size of bloom filter (array size)

r = number of non-present sub-grams of the n-gram, e.g., if we're talking about a 3-gram "a b c", then there are {"a", "b", "c", "a b", "b c", "a b c"} subgrams to check, of those some may not be present. Let's call that r. At a maximum, there are n + (n - 1) + ... + 1 = n(n+1)/2 sub-grams of an n-gram to check.

P[n-gram is a false positive] = P[all non-strict subsets are false positives] = {[1 - (1 - 1/m))^(kn)]^k}^r ~ [(1 - e^{-kn/m})^k]^r = (1 - e^{-kn/m})^(kr). When we don't do the subset checks, probability of a false positive is just  (1 - e^{-kn/m})^k, so what's the ratio of these two equations?

(1-e^{-kn/m})^{k(1-r)}. Note that 0 < (1 - e^{-kn/m}) < 1, so let u = (1 - e^{-kn/m}) => u^k(1-r). u is just the false positive probability, so let's rewrite that as a fraction, a/100 => (100/a)^{k(r-1)}. Let a = 1 (1% false positive probability), then 100^{k(r-1)}. This is how much more likely we'll have a false positive on an n-gram query without sub-gram checking as opposed to an n-gram query with subgram checking where there are r negative subsets that can be checked. Sometimes, this can be quite high, but let's say there is on average only 2 negative subsets for an n-gram query. That's extremely conservative for large n. Then, 100^k more likely to have a false positive without checking! Let's say there are 2 hash functions, 100^2! So, we can definitely exploit this property. As a sanity check on math, if there is only 1 subset, that means only the n-gram itself is a negative example and the n(n+1)/2 subgrams are positive examples, then math reduces to 100^{k\*0} = 1, which is correct -- no advantage given at this point since there are no subgrams to check to reinforce a positive as a true positive.

This is a significant reduction in probability of a false positive. We can use this to have extremely high confidence in a true positive the larger the n-gram granularity becomes, and so to meet a given target false probability for each n-gram size, we can keep increasing the false positive probability in a certain way, which means the size of the bloom filter can increase very slowly as we add more and more n-grams. I'm thinking we can have very large ngrams without much little adverse effect on bloom filter array size for a given probability of a false positive.

### new fitness parameter

To give uncertainty to wether a word is a true positive, we may want to not MINIMIZE false probability, but to make it as close as possible to a certain value for P[tp]. P[tp] = P[+] - P[fp].

why a bloom filter?

We hash each word or n-gram (these are features; thus this is a form of dimensionality reduction]. So, we don't care what it hashes to, we'll just make that hash index true if we insert the given word.

Now, if we only used one hash function, what's the problem? Let's consider the fact that we don't care about collisions among members, and collisions among non-members. In fact, as a form of dimensionality reduction and space savings, we ought to look for a hash function that maximizes these sort of collisions. A perfect hash function in fact hashes all members to index 0, and hashes all non-members to index 1 (or at least most nonmembers). This is a form of drastic dimensionality reduction; it is indeed a binary classifier using the simplest possible representation. This is good! A billion word document, for querying purposes, is just 2 bits and a hash function of size M bits, so 2 + M bits in total. But notice that M must be factored into this equation too.

For instance, if the hash function is:

hash(word) -> index

    return 0 if word is in table[world]

    else return 1

Then we haven't really accomplished anything; we're just using a massive lookup table. The complexity of the document is now represented in a table and so M = O(f(size of document)), where f is probably the square root function (according to zipf's law). The only savings we had was a product of the fact that some words repeat, and we can use that to do a form of simple compression. Moreover, the lookup table (or in general, hash function) itself may have other undesirable properties, like revealing the words (no data confidentiality) of the document. So, we're looking for a hash function that can be represented, say, as a matrix or vector of weight coefficients that operate on, say, the characters of the word [although we can operate on any feature of the words/ngrams, e.g., let the weight vector apply to n bits of a word at a time], e.g.:

hash(word) -> index

    sum <- 0

    for i <- 0 to size(word):

        sum <- sum + a^i \* weight[word[i]]

Unfortunately, the perfect hash function described earlier isn't practical on any realistic document. So, instead, what'll happen is the words (or ngrams) of a document will be spread around to different indexes, sometimes colliding and sometimes not colliding. We want to make it so that the probability that a word not in the document will collide with a word in the document (a false positive) is negligible. If we just make the size of the array (number of indices) really large, say 2^20, this is somewhat trivial to do. In fact, for any given bag of words much smaller than 2^20, they'll all probably map to different indices and be spread out quite far at that. But, this has two primary disadvantages:

(1) Takes up a lot of space. In the example, 2^20 bits, in fact, which is 2^20 / 2^3 = 2^17 bytes. Ok, less than a megabyte.

(2) If you desire privacy, then since for any given bag of words much smaller than 2^20, each word has a high probability of mapping to a unique index. So, if a word maps to a true index, that means the person can be confident that the word is there. In fact, he can just look at the distribution inside the array and infer a lot of stats about your data, e.g., number of unique words.

So, in this sense, collisions is a privacy inducing feature: if two words map to the same index, you don't if the document contains one of the words, or both of the words, or maybe any of the words since, of course, another word could have mapped there (but the last point is the case with the trivial solution also).

So what do we want to do? We want to keep the size of the array kind of small, and we want to maximize collisions between members, and separately maximize collisions among nonmembers. This will be hard to do with just one function -- maybe! I think we need more degrees of freedom. This is where bloom filters come into play. There's a lot of background theory for bloom filters, e.g., how large it, as a statistical expectation, needs to be  to be under a given probability of a false positive for a given number of hash functions, etc. But I think we can do a lot better than these expectations since we'll be training our bloom filters more intelligently to discriminate between members and nonmembers using probabilities mined from corpora.

When learning a bloom filter in this way, I will start off with one function. But I won't just be trying to minimize probability of a false positive, I'll also be trying to maximize collisions to make it impossible to tell which words are in a document. You can still do some analysis, granted, like a bloom filter with H hashes and size M with N <= M true indices has an expected number of unique members, and they can still use sophisticated algorithms to discern some probability of which words are in it (that is why it's useful in the first place! otherwise we may as well spit out a 0.5 probability of a given word being in the filter).

However, even this information leakage can be reduced if we do the following: use a one-way hash with a secret and hash that. Now, some stats can still be gleamed from the bloom filter distribution, but not much! Maybe the only stat: expected number of unique elements in the bloom filter. But I suspect even this information can be minimized, or given huge error bars, through training the bloom filter to penalize a bloom filter that doesn't have huge error bars on this.

# papers

## Concept Search: Semantics Enabled Information Retrieval\*\*\*

This paper had a lot of useful information about the discipline of information retrieval in general. It gave me an appreciation for the depth of this field. I have since began pouring over more sources of information about it.

A lot of their work can be applied to our bloom filter representation, e.g., instead of inserting "dog" insert "dog-1", such that the correct sense of the word (if dog-1 is the correct sense) is used. If the user expresses an information need for a particular sense, then we won't mismatch due to synonymy (we can improve our precision without losing recall). We can go further, e.g., when seeing dog-1, also insert carnivore-1. And so on. A lot of possibilities, but some of their algorithmic sophistication doesn't seem to be able to work with out bloom filter document representation.

A lot of notes are on the paper, but here are some issues off the top of my head...

(1) IR: map a query to a set of ranked (ordered) documents according to their relevance to the query; the query represents the information need of the user.

(2) They covered a lot of different relevancy metrics. Initially, I was thinking in terms of "degree of matching to the query" but I'm persuaded that "relevance" is a better way to think about this; also, a large body of existing work has already been established for doing this.

    (a) We'll be using some of the relevance metrics, but combine it with our own peculiar needs, e.g., we're also ranking blocks (not just documents) within a document, and since we can't see what the blocks contain exactly, we provide a useful visualization feature to show things like density or how close it is to the weighted keyword (or key terms) in the query.

(3) They cover the main models: vector space, boolean, and probabilistic. Ours approach borrows a bit from these fundamental categories.

    (a) tf-idf, etc.

(4) At detph, they cover the advantages/disadvantages with syntactic search, e.g., polysemy and synonymy, related concepts, etc. I've made notes about this elsewhere.

### Semantic Search

Concept Search

basis

computation of semantic relations between concepts

needs semantic information

natural language processing

### Syntactic Search

basis

string similarity between words

syntactic matching

atomic elements

words

multi-word phrases

ngrams

big problems

polysemy

synonymy

measures

cosine similiarity

### Information Retrieval

Semantic

Syntactic

### relationship to our approach

We mostly use syntactic searching... but things like stemming (finding word roots) are somewhat related to concept searching.

It is interesting to point out, though, that a lot of their work can be used in our work.

## Practical Techniques for Searches on Encrypted Data

This is one of the first papers on encrypted searching. A lot of good info.

## secure and privacy preserving keyword searching for cloud storage services

### privacy

user query privacy

user data privacy

### natural approach to privacy

Encrypt all the data.

E.g., use public key to encrypt email + keywords before sending it to CSP then send queries in the form of encrypted keywords to retrieve email.

problems

CSP cannot determine which files meet information need of user, therefore must send all files.

Decrypting/decrypting is cpu/memory intensive. This is problematic especially for thin-client type devices, e.g., tablets or smart phones.

Another issue not address by paper: not everyone can search it; you must have the key. If you have the key,  you can read file exactly as it is given. What if you want to give people "partial" access in the form of looking up keywords or looking up n-grams? (All or nothing.)

cpu/memory intensive

bandwidth-heavy

all-or-nothing

### terminology

CSP (cloud service provider)

trusted

untrusted

Bilinear Diffie-Hellman (BDH)

random oracle model

PEKS (public key encryption with keyword searching)

EPPKS (efficient privacy preserving keyword searching scheme)

math terms

* cyclic groups
* prime order
* additive group
* multiplicative group
* Bilinear map
* bilinear
* super-singular ellypic curves
* abelian

SPKS

IND-CPA

### relevancy to our research

These results may be combined.

A disadvantage to our approach is that:

(1) A user (any user) may, if they have the bloom filter, find out which words -- with the possibility of a false positive -- are in the encrypted document. This may or may not be desirable behavior. Data privacy.

(2) Someone eavesdropping on your queries can determine what your are interested in. Query privacy.

How can this be addressed? Before inserting a word into the bloom filter, encrypt it and insert that into the bloom filter instead.

Encrypt(SecretKey, Keyword) -> Keyword'

Now, to query the document for keyword k, you must do the following:

Does the bloom filter have Encrypt(SecretKey, k)? If it tests as a positive, then (with some degree of false positive probability) his question is answered.

What about mapping Encrypt(PublicKey, Keyword) instead? The end result is still a randomized sequence of characters (encrypted keyword) that has been inserted into the bloom filter.

## on indexing and information disclosure measure for efficient cryptograph query

### similarity

Seems to be based on an idea similar to what i proposed; namely, optimizing the structure (in our case, hash functions at a given bloom filter size) such that we try to minimize the probability of a false positive on probable n-grams.

On the topic of probable n-grams, I have been considering how to do this. I have already implemented a solution which will take in whatever corpus you feed it and spit out a large tree of grams, with the ability to sample from it according to the word frequencies in log2(n) time (well, log2(n1) + log2(n2) + ... + log2(nk), where k is size of ngram we wish to sample). An ideally suited text corpus is what? I'm thinking a few different kinds. One is to sample from common sources of text, uniformly, to try to model the general distribution found "out in the wild". Another source of text is the document repository the user can choose documents from; this has a highly discriminating property -- but it assumes a user will not try queries composed of words not found in the document store.

However, I see a combination approach working. Get a list of words which represents the top 99.9%, or whatever, of words seen in the language. Now, learn to not have a probability of a false positive on these 1-grams below a specified threshold (still using information about probability of seeing a 1-gram to prefer to have a false negative on an improbable 1-gram than a probable 1-gram). Then, if desired, learn up to k-grams \*only from text in the document store\*. Now, here's the outline: for a query with a k-gram in it to be a true positive, then all the 1-grams the k-gram is composed of must be true positives. We have already taken care of the 1-grams sufficiently well, so do this as a sanity check. If the k-gram check fails, then it's not a true positive. If the k-gram check succeeds, then it may be a false positive. Now, see if the 1-gram check fails. If it does, then assume it was a false positive on the k-gram check. If both checks succeed, then assume the k-gram is a true positive.

As an extension, we can also do a 2-gram, ..., (k-1)-gram checks on subsets of the k-gram term in the query. They all must succeed for the k-gram to have the possibility of being a true positive. However, this amounts to 2^k checks, and for large k this is not reasonable, e.g., k=6 -> 2^6 = 64 checks. This doesn't seem too unreasonable since all of them are constant lookups, but 2^10 is 1024, 2^20 is over a million. However, we don't have to be exhaustive. We only want 'evidence' that the, say, 20-gram is a true positive. Checking just the 1-grams may be enough. Checking also a random sampling of other grams may be all we need, say, a constant 10 other variations.

So, now we have a method of being pretty confident in a positive result being a true positive. If the 1-gram check succeeds, it's possible it's a false positive. It's still possible for the 2-gram checks to fail, if so, and so on until we get to the k-gram. At each stage, failure for a false positive is a possibility, so this is pretty interesting. And, of course, all of these 1-grams, 2-grams, ..., k-grams also permit exact matches up to k words for a term, which is nice in its own right.

### information disclosure

query disclosure

An eavesdropper can see which keywords the user is interested in by examining the query.

A solution to this problem for our approach: mentioned elsewhere, but encrypt terms then hash the encrypted terms. This takes care of both data disclosure and query disclosure.

data disclosure

A person can query document for terms to get an idea about what it might contain, up to the granularity of the n-grams and block sizes.

A possible solution for our approach: see query disclosure. Basically, encrypt terms before inserting into bloom filter.

## Extremely Fast Text Feature Extraction for Classification and Indexing

This paper doesn't deal much with our research, but its discussion on representing strings as a hash vector had some interesting things to say, namely, that this representation serves equally as well as the bag of words (strings) representation when learning a classifier on it. They proposed a hash function which is quite simple -- a simple bit-shift -- but acquires variety by changing how a character maps to an integer. (This can be randomized.) This, it turns out, has much better distribution properties than using the value of the character itself compared to other well-known hash functions, e.g., Java's hash function.

With regards to our research, this may be an efficient way to represent hash functions: only one one (or maybe a couple), but with different character -> integer mappings. One can use a form of stochastic gradient descent to change the mappings in an advantageous way (with respect to training/validation minimizing false positives). I think this will be fast.

## Shared and Searchable Encrypted Data for Untrusted Servers

This has a lot of interesting features. I haven't had time to digest the material yet, but a few points:

(1) They assume authorized users are fully trusted. This is a bold assumption, and doesn't cover tiered access. Tiered access, especially if many people are stakeholders, is sometimes more appropriate. A simple example of tiered access:let some users search using the capabilities exposed by the document, but not open the document itself. This is a trivial case in our bloom filter example, but our example doesn't have as strong a guarantee about security, e.g., what information can be gleamed from the bloom filter distribution of values in indices? A part of our work may be to learn hash functions which optimize on this parameter in addition to reducing false negatives.

### trusting a KMS server

A "flaw" in their design: they trust a key management server. What if the server is compromised? What if the server isn't trustworthy? They make the reasonable argument that a KMS is a lot less involved than a data storage server, e.g., less data needs to be secured and it's easier to maintain. However, it is a weak link in the chain: they're solution is only as secure as the weakest link.

But why does this vulnerability exist? To assist in other interesting functionality, namely key revocation. That is, you can revoke a user's access to the encrypted documents without having to go through an expensive process of reencrypting the documents under a new encryption key. So, that seems very applicable, and there's no easy solution to this problem without, seemingly, giving the system this vulnerability.

## Secure Conjunctive Keyword Search Over Encrypted Data

Discussion on capabilities seems interesting. They point out that they may want the user to be able to search for a conjunction of two terms, A and B, but not, for instance, B by itself. This behavior is possible to implement with n-gram granularity as proposed in our research, but without as strong of a security/privacy model.

They also name some terms I've been thinking about, e.g., query privacy (not let an eavesdropper know what you've been searching for) and data privacy. In our case, query privacy is possible by hashing the encrypted terms (using an asymmetric or symmetric key) instead of the plaintext. This also prevents unauthorized users from querying the document at all.

Also, they only discuss keyword searching in terms of certain fields, e.g., an email in which the subject or the email sender's email address can be searched (has a search capability). It's interesting to point

They at times seem to create straw man arguments for simpler solutions, e.g., assuming that for a conjunctive query on a document in which there are m searchable keywords, that this would require 2^m storage space -- exponential in the number of searchable keywords. This strikes me as a little dishonest though since its unlikely all M terms will be useful in a query, therefore only a subset of m, say, k < 5, should be sufficient. Even more selectivity is possible, e.g., allow only search terms to be used in a conjunction, thus allowing very large conjunctive queries without the exponential cost, e.g., (M choose k) + (M choose k-1) + ... + (M choose 2) + 1 instead of 2^M, which has intractable space complexity with respect to M.

Another example of this is when they decry the simple set intersection approach of conjunctive queries consisting of searching on keyword key1 and keyword key2 separately and only returning those documents which have both keywords. They make the argument that this exposes a lot of extra information about the encrypted document -- and it does. But, if the keyword queries are, for instance, encrypted and we do a search on the encrypted keywords instead, the server doesn't even know what its looking for -- it just knows which documents they map to.

* nomenclature
  + PIR
  + bilinear decisional diffie-hellman (BDDH)
* distinguishing capabilities

I'm not really clear on how fruitful this line of inquiry is. What if you can, given two documents with indistinguishable capabilities (they both have the same keywords, for instance), distinguish one document from the other? What is the significance of the information that is gained because of this? It is never clearly articulated. That's not to say this is not a useful thing to have, only that I wish they had done a better job selling it. Maybe I didn't grok enough of the math to appreciate their analysis.

Also, while keywords aren't revealed, the fields that the keywords belong to are. This is a limitation of their work; that is, first, a keyword must have an associated and unique field (which is limiting), and perhaps worse, the field is exposed during the query. So, while an eavesdropper may not be able to see which emails you are interested in, they may be able to tell the distribution of unique email addresses that you search for. This by itself can reveal private information, e.g., the distribution may be similar tp the distribution they see from you in a less secure setting, and therefore be able to make the mapping that way, e.g., this encrypted document is an email from alex@nowhere.net.

* + e-advantage
  + ICC (indistinguishability of cipher from cipher)
    - adverserial game
* capabilities
* query privacy
* data privacy

## Adding Compression to BLock Addressing Inverted Indexes

See notes on paper.

A few key observations:

(1) Some similarity to our work: proposed inverted indexing to blocks instead of word positions (to save space); once they find out which blocks a word is in, they do a sequential scan. We have blocks too, for localization (i.e., ranking blocks in an encrypted document based on how close it is to all the weighted keywords, or some other metric), but we have the opposite approach: we do a sequential (or parallel) scan through the blocks (not the words in a block), and do constant lookups within each block.

(2) I liked their discussion on using a Huffman tree on words as the atomic unit (leaves in the tree) instead of bits. Also, using bytes at a time to index into the tree seemed interesting, but I'll have to re-read it: how are they doing a simple left or right decision at a branch with byets instead of bits? I think I know but will have to re-read. Anyway, I may use this structure in a different way than them: to make the tree gram, a tree such that the deeper you go, the larger the n-gram you have traversed. My implementation of this works well but takes a lot of space; this coding scheme can drastically reduce its memory footprint (ngrams are notoriously space hungry) and maybe speed it up a tiny bit (by having fewer bytes to compare less than to?).

Otherwise, their discussion on compression was irrelevant to our research. Bloom filter is made up on just boolean array elements, and its size is a function not of word lengths or other matters, but number of members (unique words/ngrams) and probability of a false positive. Unique words/ngrams is a function of n-gram granularity, text/block size, and freedom of regular expressions (e.g., how many classes to represent--alpha, alphanumeric, numeric?--and how many wildcard classes can be used in a word; I've quantified space complexity for both of these though more work needs to be done to fully formalize it).

## supervised semantic indexing\*\*\*

## Fuzzy Keyword Search over Encrypted Data in Cloud Computing

There were a few issues in this paper, but overall I thought they had a simple (clever) solution to a thorny problem: approximate matching with up to k-edit errors while avoiding the combinatorial explosion. So, here's the idea: when inserting "abc", also insert "ab\*", "a\*c", "\*bc", "\*abc", "a\*bc", "ab\*c", "abc\*".

So if the user searches for "abb", and no match is found, then try 1-edit errors, including "ab\*", which is indeed in there.

I have extended this concept to cover a limited form of regular expressions with limited classes. Hopefully, with the use of a bloom filter and, reasonably, expecting higher false positives with the regular expression matching, we can include many such variations without blowing up the bloom filter size. I think I have a good handle on how to do this. So, for instance, let's say a doc contains "abc". Then, in the bloom filter, we insert "abc", "ab\*", "a\*c", "a\*\*", "\*bc", "\*b\*", "\*\*c", "\*\*\*". 2^3. Exponential, true, but we can limit this, e.g., if a query term is of size n, then we only perform at max (n choose k) \* 2^k variations. If k = 3, then (n choose 3) \* 8 for every term, if we assume average term is n = 6, then 8\*n!/3!(n-3)!=8\*6!/3!3!=160, so for every word we insert 160 variations. I think we might be pleasantly surprised by how well the hashes deal with this constant size increase. And if this is too large, I have a lot of ideas about how to reign in things. Anyway, back to problem. So, say the user wants to search for any pattern matching "ab\*", and the document contains "abc", well, this will match.

We can extend this to multiple classes, e.g., \* = alphanumeric, # = numeric, & = alphabetical. Math is then: (n choose k) \* 3^k. Now it's (n choose 3) \* 27, so a big jump. Obviously there's a trade-off between space complexity and regular expression generality.

For "abc" we can also insert "\*abc", "\*\*abc", "\*a\*bc", "\*ab\*c", "\*abc\*", ..., "abc\*\*". Clearly, this grows quickly. But now, we can do more complicated regular-like expression searches. These searches are constant, too, but with space complexity that grows exponentially in theory. HOWEVER, we can trade space complexity for false positives. This will be an interesting trade-off to explore; combined with learning good hashes, we may get a lot of mileage out of this approach. ALSO: If a query has more than k asterisks, we can immediately reject it; we don't support larger than k wildcards. That does leave character sequences with <= k asterisks. We can construct negative training examples of these -- maybe just a subset -- and go after a desired false positive probability. But I think a higher false positive probability is reasonable for regular-like expression matching (on encrypted keyword searching on encrypted data).

# Weka

# nomenclature

* Polysemy
* Document
* information need

http://en.wikipedia.org/wiki/Information\_need

* Text Retrieval Conference (TREC)

http://en.wikipedia.org/wiki/Text\_Retrieval\_Conference

"Uniform scoring is performed so the systems can be fairly evaluated."

* Lucene

http://lucene.apache.org/

* + openrelevance

http://lucene.apache.org/openrelevance/

* As We May Think
* structured / semi-structure / structured

# Information retrieval

**Information retrieval** is the activity of obtaining information resources relevant to an information need from a collection of information resources. Searches can be based on metadata or on full-text (or other content-based) indexing.

Automated information retrieval systems are used to reduce what has been called "information overload". Many universities and public libraries use IR systems to provide access to books, journals and other documents. Web search engines are the most visible IR applications.

---

Information retrieval (IR) is ﬁnding material (usually documents) of an unstructured nature (usually text) that satisﬁes an information need from within large collections (usually stored on computers).

## query

An information retrieval process begins when a user enters a query into the system. Queries are formal statements of information needs, for example search strings in web search engines. In information retrieval a query does not uniquely identify a single object in the collection. Instead, several objects may match the query, perhaps with different degrees of relevancy.

An object is an entity that is represented by information in a database. User queries are matched against the database information. Depending on the application the data objects may be, for example, text documents, images,[1] audio,[2] mind maps[3] or videos. Often the documents themselves are not kept or stored directly in the IR system, but are instead represented in the system by document surrogates or metadata.

Most IR systems compute a numeric score on how well each object in the database matches the query, and rank the objects according to this value. The top ranking objects are then shown to the user. The process may then be iterated if the user wishes to refine the query.[4]

### refinement

### input into the system to satisfy information needs

### query language

Information retrieval query language attempts to find documents containing information that is relevant to an area of inquiry.

This is in contrast to database query languages which attempt to give factual answers to factual questions.

### complex queries

contextual query language (CQL)

## rank

## relevancy

### testing performance

See: TREC

source of docs

source of queries

source of relevancy judgements for queries

binary: relevant or not relevant

## performance measures

### precision

http://en.wikipedia.org/wiki/Precision\_(information\_retrieval)

### recall

http://en.wikipedia.org/wiki/Precision\_(information\_retrieval)

### fall-out

### F-measure

### average precision

### mean average precision

http://en.wikipedia.org/wiki/Mean\_average\_precision

### Discounted cumulative gain

## information overload

### relevant results important

## term discrimination

http://en.wikipedia.org/wiki/Term\_Discrimination

# document retrieval

Document retrieval systems find information to given criteria by matching text records (*documents*) against user queries, as opposed to expert systems that answer questions by inferring over a logical knowledge database. A document retrieval system consists of a database of documents, a classification algorithm to build a full text index, and a user interface to access the database.

A document retrieval system has two main tasks:

* Find relevant documents to user queries
* Evaluate the matching results and sort them according to relevance, using algorithms such as PageRank.

Internet search engines are classical applications of document retrieval. The vast majority of retrieval systems currently in use range from simple Boolean systems through to systems using statistical ornatural language processing techniques.

* text retrieval
* two main tasks
  + (1) find relevant docs
  + (2) rank results (relevancy)
    - page rank
* internet search
* kinds of
  + form-based (syntactic)
    - substring matching
    - suffix tree

# vector space model

Advantages (over standard boolean model)

    \* simple model based on linear algebra

    \* allows computing degree of similarity between queries and documents

    \* allows ranking of documents by relevance

    \* partial matching (approximate)

Limitations:

    \* long documents poorly represented

        - similarity is low -- small scalar and curse of dimensionality

    \* query terms must precisely match document terms

    \* semantically insensitive -- docs are just a bag of terms (words) so synonymy and polyonomy is a problem -- false negative

    \* order of words is lost in the representation

    \* assumes terms are statistically independent

    \* weighting of terms is not very formal

* cosine similarity

On the bloom filter, can only do this:

Take a query q, transform it to q', note that at this point query q' is a match (it may be a false positive, however). Then, take cosine similarity of q and q'.

* tf-idf

http://en.wikipedia.org/wiki/Tf-idf

This is a commonly used scheme in the vector space model.

term frequency-inverse document frequency model, where weights in v\_d = [w1,d, ..., wn,d] are assigned according to the tf-idf formula.