Contents

Experiment Design

Inputs

- Secure index

  The type of secure index. It is either PSIB, PSIF, PSIP, or BSIB. In most of the experiments, multiple secure indexes and their respective outputs are compared to one another. See **Background Information** for more details.

- Corpus size

  Number of documents in the corpus. A variable corpus size should effect most outputs in a linear way, e.g., MinDist* lag time should depend linearly on the number of documents (assuming documents are of fixed size). However, MinDist* scoring and BM25 scoring may be effected in a non-linear way, thus I make this variable to see how such outputs respond.

- Document size

  Number of words in each document in the corpus. A variable document size will be used to see how each *secure index* scales with document size with respect to a number of parameters.

- Terms/query

  Number of terms in each query, where a term is either a keyword or an exact phrase.

- Words/term

  Number of words in each term.

- Secrets

  Number of secrets that can be used to search for query terms in the secure index database.

- Obfuscations

  Number of extra "noise" terms added to a query. The noise terms can either be randomly generated, or sampled from a obfuscation distribution.

- Location uncertainty

  Unigram or bigrams in the document have exact positions. Exact positions reveal too much information about the contents of the document; thus, their positions are either randomly changed (up to some maximum offset from the true position) or only said approximately located within some interval (e.g., PsiBlock only reveals that a term is within some block range).

- False positive rate

  A random string that is not in the document will have a probability (equal to the false positive rate) of testing positively as belonging to the secure index which approximately represents the document. This probability can be controlled by increasing or decreasing the false positive rate.

Outputs

- Secure Index Size

  The size (e.g., bytes) of the secure index database for a corresponding corpus (collection of documents).

- Build Time

  Time taken to build the secure index database for a given corpus.

- Load Time

  Time taken to load a secure index database for a given corpus.

- Boolean query precision

  Proportion of retrieved documents are relevant in a Boolean query, in which all of the query's terms must exist in the document. See **Background Information**.

- Boolean query recall

  Proportion of relevant documents were retrieved. See **Background Information**.

- BM25/MinDist* ranking MAP (mean average precision)

  See **Background Information**.

- BM25/MinDist*/Boolean ranking lag

  Time taken for the corresponding kind of query to complete.

Global Experiment Design Parameters

- The number of unique words per corpus (corpus dictionary) is fixed at 10,000 words. The unique words in the dictionary follow a zipf distribution, and they are randomly generated with an average length of 6.5 alphabetic characters. Such a dictionary is uniquely constructed for each trial of every experiment.

- For BM25 scoring, parameter b is set to 0.75 and parameter k_1 is is set to 1.2. See Background Information for more details.

- Each document of size n in the corpus separately samples m=$12n^{\frac{1}{2}}$ unique words from the corpus dictionary, conforming to Heap's law with K=12 and β=0.5. Once m unique words are sampled, they are renormalized to make them into a proper distribution. It is this distribution that is used to generate the sequence of words for a document.

  I did this with the intention of making each document approximately follow a zipf distribution, but with a different subset of words to account for different authors with different but overlapping vocabularies. In hindsight, it would have been sufficient (and perhaps preferable) to have simply sampled n words directly from the corpus dictionary.

- When calculating the outputs for a given input, e.g., false positive rate, location uncertainty, etc., I always use a query set consisting of 30 queries. I then average the outputs over all of those queries where appropriate.

- For each query term in a query in a query set, I seed a document in the corpus with that term with probability p = 0.2 except where otherwise noted. Thus, for a query with k terms, the probability that one or more of its terms occurs in the document is $P[at\ least\ one] = 1 - P[none] = 1 - (1-p)^k$. For k = 1, P[at least one] = p = 0.2; for k = 2, P[at least one] = $2p - p^2 = 0.36$. Thus, if a query consisting of k terms seeds a corpus of size N, then on average $N(1-(1-p)^k)$ documents will be seeded with the query term.

  In general, this should mean when doing a mean average precision calculation, the expected number of documents relevant to a query with k terms will be $N(1-(1-p)^k)$. The remainder should be non-relevant, i.e., map scores of 0, which means it is irrelevant how they are ranked and can thus be ignored in the mean average precision calculation. Of course, if the secure index does score them with a score of non-zero, that is a sign of a false positive, and should and does subsequently degrade the map score. To clarify, when calculating a map score, I include as many documents for map scoring in the ranked list as there are non-zero scores for retrieved ranked documents.

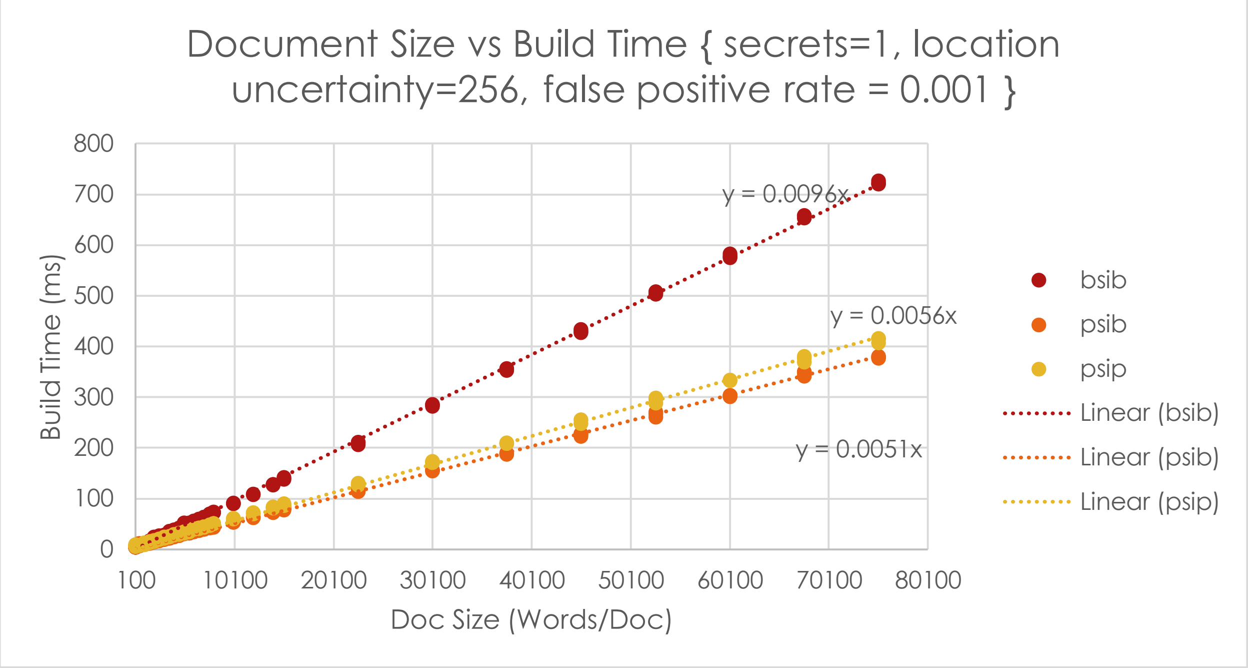
  Once a document is targeted to be seeded by a given query term, all such occurrences of the term will occur within a window size of w ~ U(min(size(doc), 2000), max(size(doc), 6000)) except where otherwise noted.
  Finally, the number of occurrences of the term within that window will be n = $\min\{1, w^{\frac{4}{5}} \times$

$UINF(0.01, 0.001)\}$.

- When measuring precision, MinDist* map, or bm25 map, each query in the query set (as previously mentioned, there are 30 queries per query set in total) is submitted 10 times for the block based indexes, and the average of those 10 is taken to be the output.

Experiments

Document Size Experiments



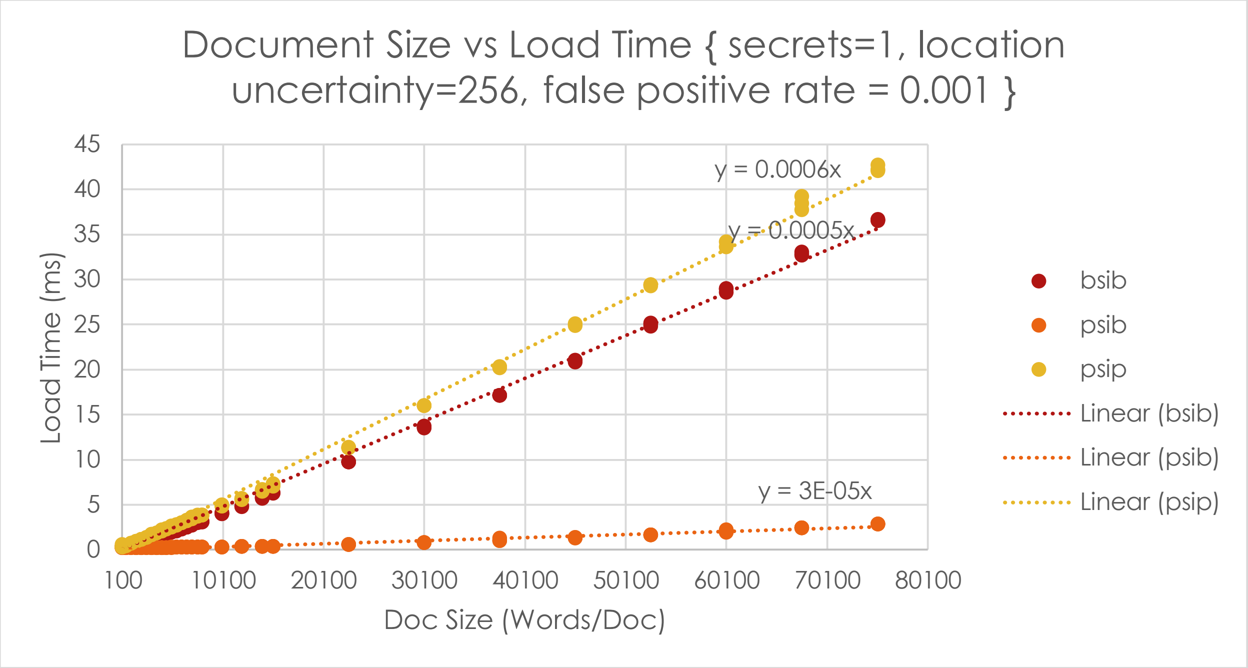
Input: Document size (Words/Doc)

Output: Build time.

In this experiment, I am interested in seeing how document size affects secure index build time. The actual size of the document in bytes is less important than the number of words in the document.

The output shows an unmistakable pattern. As the document size (in words) increases, the build time for each type of secure index also increases. However, while PSIB and PSIP are running neck and neck, BSIB is a distant third, taking nearly twice as long as the other two. I do not put much stock into this particular measurement, as I did not implement the Bloom filter in the BSIB myself, so there may be some lurking inefficiencies in the actual implementation that I am not aware of.

All three of the secure indexes do not seem unreasonably slow; even a document consisting of nearly ~ 75,000 words (300+ pages) takes less than a second to build.



Input: Document size (Words/Doc)

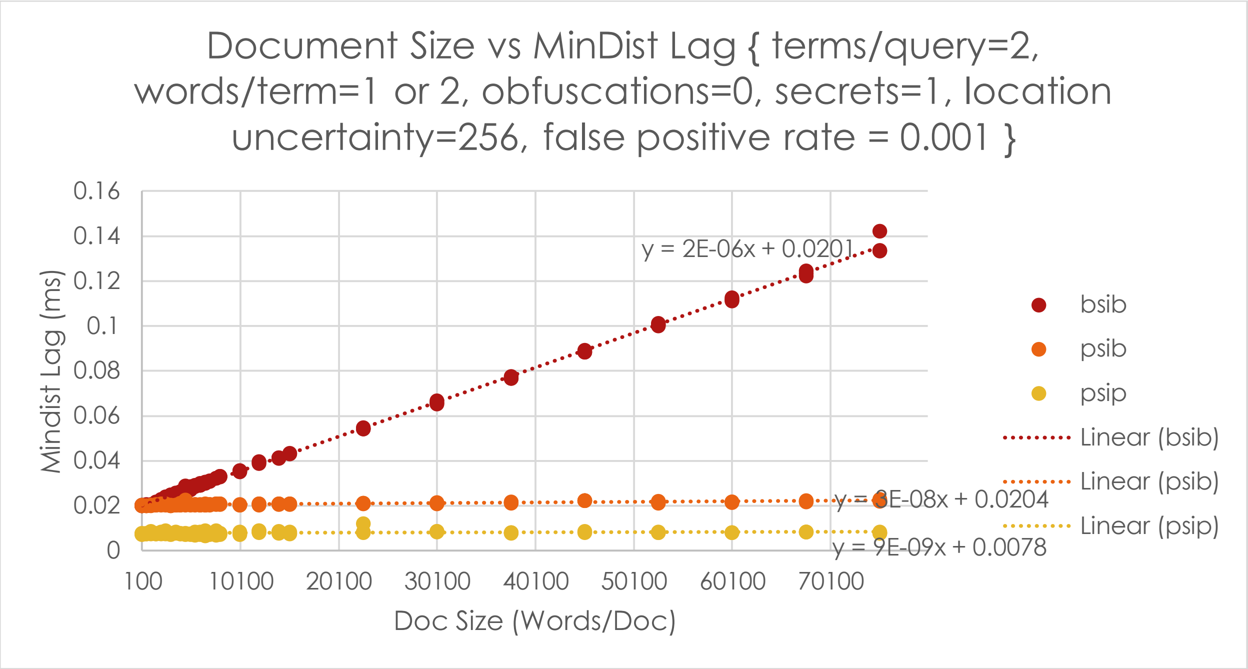
Output: Load time.

In this experiment, I am interested in seeing how document size (in words) affects secure index load time.

Remarkably, PSIB (and, to a greater extent although not shown here, PSIF) is nearly constant when representing documents from 1 page to 300 pages. The PSIB representing the 300 page documents takes only ~2.85 milliseconds to load raw from disk.

The other two seem to perform less impressively; I did not implement the Bloom filter – I used one that came recommended by others – but I imagine that, as the document size increases, the overhead of serializing a larger number of hash functions takes a toll. As far as the performance of the PSIP is concerned, I did not take any great pains to optimize it; for instance, I load the postings lists as vectors of integers, which may incur significant vector construction overhead as the number of terms in the document increases.

Note that there is nothing stopping someone from using a more efficient serialization of the PSIP's posting lists. Even a simple bit vector per term would likely see significant gains here.



Input: Document size (Words/Doc)

Output: MinDist* lag

Machine:

In this experiment, I am interested in seeing how document size (in words) affects MinDist* query lag.

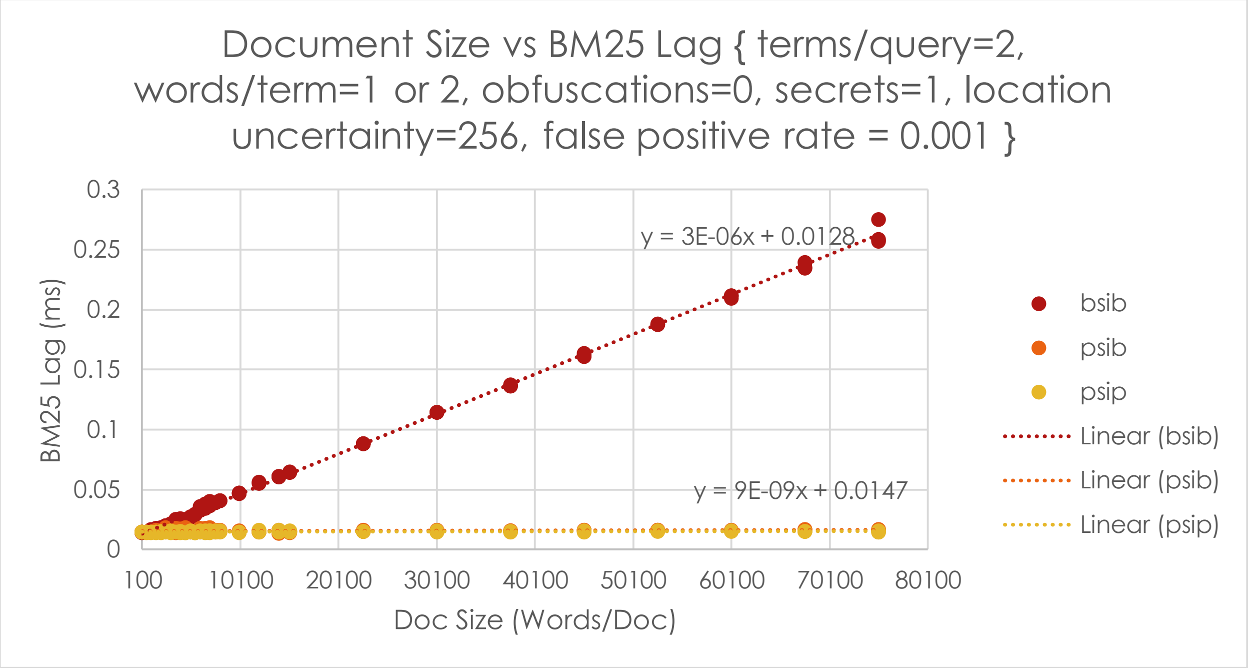
For large documents, BSIB does relatively poorly on this measure. This makes sense in that, for a fixed location uncertainty, as the document size (in words) increases, the document must segmented into more blocks and therefore more Bloom filters (and therefore more hash functions) must be queried. For the 300 page document, the lag is around 0.14 milliseconds. If the corpus consists of a million documents, this operation would require nearly 2 minutes and 30 seconds to complete, which is impractical.

The PSI-based secure indexes, on the other hand, take a nearly constant amount of time with respect to document size. Even here, however, scalability is an issue. A corpus consisting of a million such documents would require 20 seconds to complete. Even PSIP, the fastest secure index on this benchmark, would require ~7 seconds to complete the query. None of them are below the 1 second mark, which is often considered the maximum delay a typical user will tolerate. Of course, this is not a typical use case—the services provided by secure indexes are not free, and the lag is far better than the alternative: downloading the entire corpus, decompressing the documents, and conducting a local search.

However, there are a two immediately obvious things that could significantly speed up the MinDist* search operation. First and foremost, each secure index in the database re-hashes the hidden query's unigram and bigram terms with SHA256. While the re-hashing operation is desirable to ensure that a cryptographic hash of a term in one secure index looks nothing like the cryptographic hash of the same term in any other secure index, using SHA256 to perform the re-hashing is overkill; after all, the hidden query itself has already been transformed using SHA256.

Each evaluation of SHA256, according to my benchmarks, takes ~0.0024 milliseconds. This is fairly significant; a single SHA256 hash is slightly over one-third the total time taken, on average, to complete a PSIP MinDist* query. Moreover, 0.0024 milliseconds is required for each hidden unigram or bigram in the query. While the secure indexes short-circuit processing queries when appropriate, it is clear that significant savings could be had by using a much quicker non-cryptographic hash function without any loss in confidentiality.

The second way to speed up query processing is through parallel programming techniques. Each query can be independently queried, so this is what is known as an *embarrassingly parallel* problem. While building the secure indexes actually has been parallelized somewhat, nothing else has been.



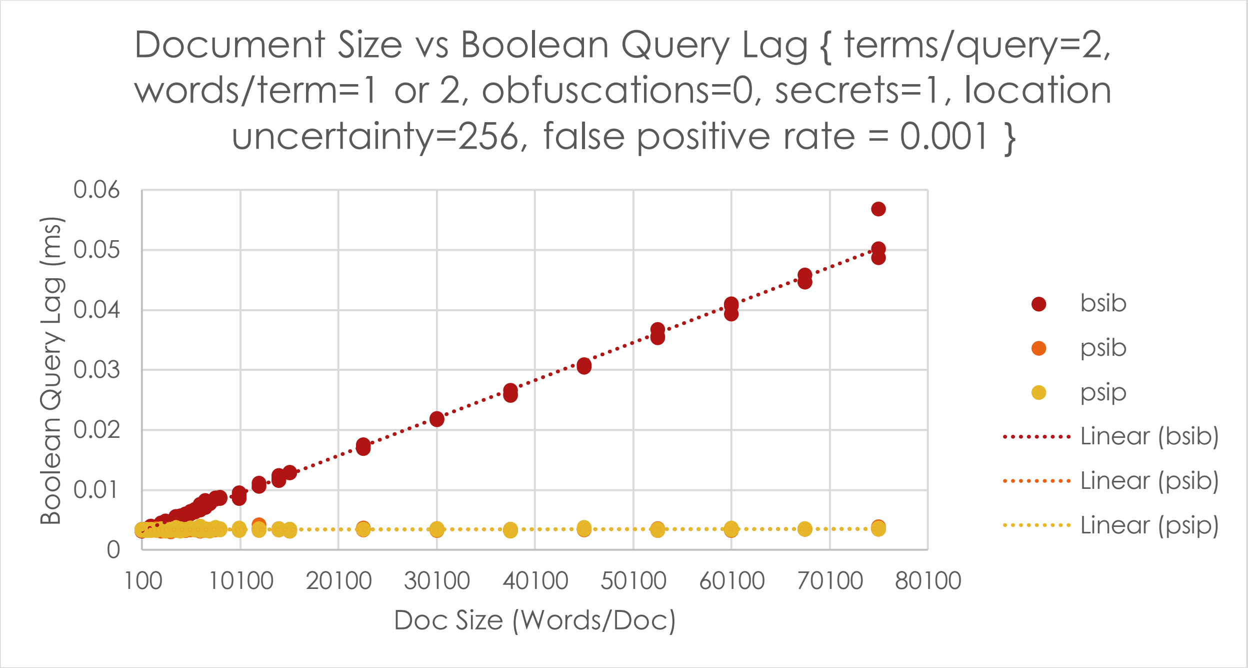
Input: Document size (Words/Doc)

Output: BM25 lag

In this experiment, I am interested in seeing how document size affects BM25 query lag. The actual size of the document in bytes is less important than the number of words in the document.

Like with MinDist* lag, BM25 lag shows a similar pattern. However, note that BM25 is even slower. The reason for this is related to the fact that BM25 queries secure indexes twice for each query term; once for calculating the number of documents which contain a query term, and once for calculating the number of times the query term appears in a document.

As mentioned in the section on caching, many of these computations can be memoized or cached. This could in practice be expected to save a significant amount of time. Moreover, in this particular case, the implementation of the BM25 algorithm can be streamlined to only require a single frequency per query term request per document, rather than the two different queries used presently.



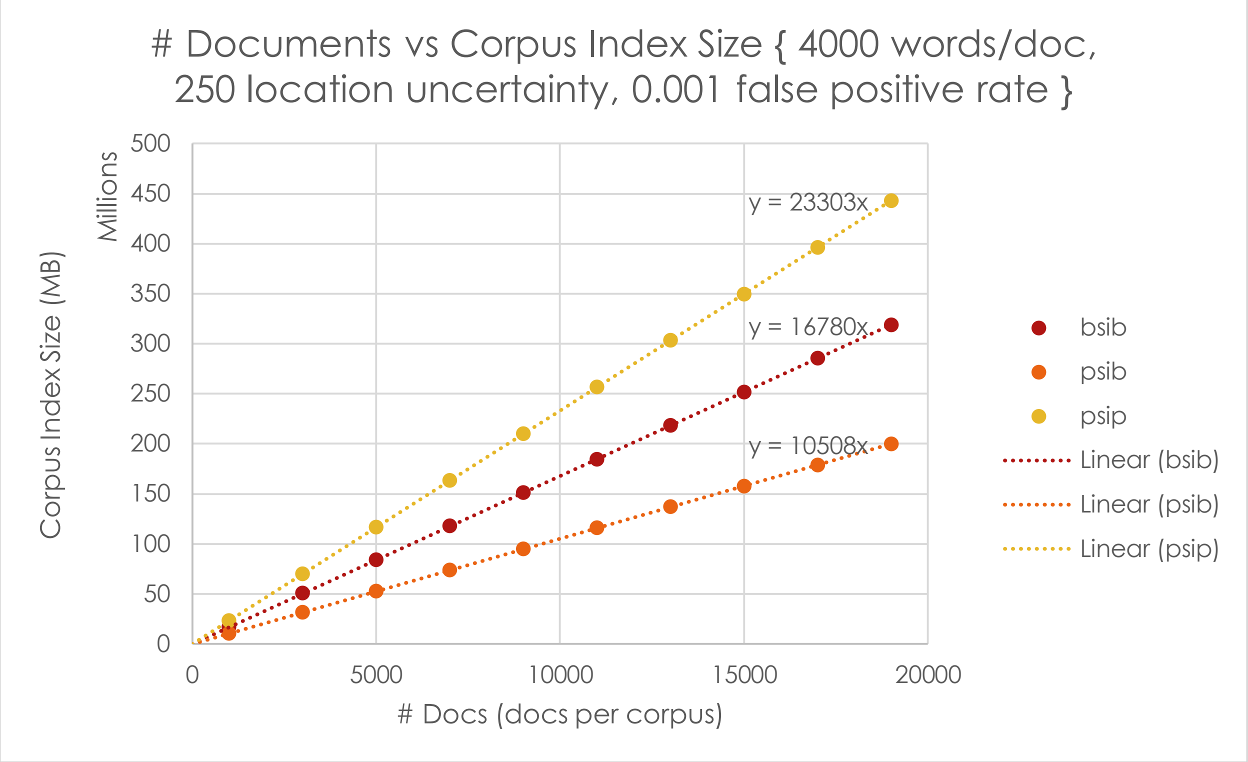
Input: Document size

Output: Boolean query lag

As with every other output related to lag time, BSIB performs comparatively poorly. That said, this is the quickest—and simplest—kind of query you can submit to a secure index database. In this case, the time required to perform the SHA256 re-hashing operation takes up the most significant portion of time.

Indeed, PSIB and PSIP can complete this operation in approximately 0.003 milliseconds, at least 0.0024 milliseconds of which is consumed by computing SHA256 hashes. In other words, this operation would take, minus the SHA256 overhead, approximately 500 nanoseconds on average per document for the PSI-based secure indexes.

Number of documents (corpus size) experiments



Input: Number of documents (docs/corpus)

Output: Total corpus index size (MB)

Background Information

Precision and recall

Precision and recall are relevant metrics for Boolean searches; they do not rank retrieved documents like BM25 or MinDist*; a document is either considered relevant (contains all of the terms in a query) or non-relevant.

Precision measures the proportion of retrieved documents which are relevant to the query. It is defined as:

$$precision = \frac{|relevant \ \cap \ retrieved|}{|retrieved|}$$

Precision has a range of [0, 1]. Recall measures the proportion of relevant documents that were retrieved. It is defined as:

$$recall = \frac{|relevant \ \cap \ retrieved|}{|relevant|}$$

Recall also has a range of [0, 1]. It is trivial to achieve a recall of 1 (100%) by retrieving every document in the corpus. This, however, comes at the cost of decreased precision. Thus, in general, there is a trade-off between precision and recall.

BM25

BM25 ranks documents according to their relevance to a specified query. It is a term weighting heuristic based on two fundamental insights. First, some of the terms in a query will occur more frequently in one relevant document compared to another relevant document. When scoring the relevancy of a document, if a query term appears in it frequently, the given document should be given more weight all things else being equal.

The second insight is that some query terms will be in a large proportion of the documents in the corpus. These terms, therefore, carry very little meaning—they have little discriminatory power since they appear in a large fraction of the documents. Conversely, many query terms will be very rare or even unique in a corpus, and thus they have significant discriminatory power. For example, the term "the" is in nearly every document—it serves as linguistic glue— but the term "acatalepsy" will be found in very few. The more discriminatory power a query term has, the more weight it should be given when scoring a document's relevancy to the query.

Mathematically, these intuitions are realized using the following standard IR functions:

$$\begin{aligned} &BM25Score(D, Q, C) \\ &\quad = \sum_{i=1}^{|Q|} InverseDocumentFrequency(q_i, C) \\ &\quad \times \frac{TermFrequency(q_i, D) \times (k_1 + 1)}{TermFrequency(q_i, D) + k_1 \times (1 - b + b \times \frac{|D|}{avgdl}}, \end{aligned}$$

where Q is the query (and |Q| is the number of terms in Q), D is the document to be ranked by *BM25Score*, C is the corpus (collection of documents D is being scored against), q_i is the i^{th} term in query Q, and avgdl is the average size (in words) of documents in corpus C.

Parameters b and k_1 are free parameters. In the experiments, b is set to 0.75 and k_1 is set to 1.2. These are typical values, although ideally these parameters would be automatically tuned for each secure index. *TermFrequency* is a function which simply returns the number of times query term q_i appears in document D. *InverseDocumentFrequency* is defined as:

$$InverseDocumentFrquency(q_i, C) = log\left\{\frac{|C| - count(q_i, C) + \frac{1}{2}}{count(q_i, C) + \frac{1}{2}}\right\},$$

where |C| is the number of documents in corpus C and *count* is a function which returns the number of documents in C which had one or more occurrences of q_i.

When using BM25 to rank search results, each document D in corpus C is ranked according to query Q by the function *BM25Score*. After giving every document a BM25 rank, the results are sorted in descending order of rank as the final output.

Note that in the experiments, no standard short-cut techniques were used to avoid processing every query term in Q for every document in C.

MinDist*

MinDist, like BM25, ranks documents according to their proximity relevance to a given query. It is a less established ranking heuristic than BM25, but in experiments [1] it had performed well compared to other proximity heuristics. In our experiments, we add additional tunable parameters to MinDist and call it MinDist*.

It is a proximity heuristic in which the minimum distance between each existent pair of query terms in the document is summed over. The intuition behind the min-pairwise summation: the more concentrated the query terms are in a document, the more relevant the document is to the query, but only up to a certain point.

For example, consider a query Q consisting of two keywords, Q = {“computer”, “science”}. Given two documents, A and B, where A contains both “hello” and “doctor” on page 7, and B contains “computer” on page 7 and “science” on the page 20. It is clear that A should be considered much more relevant to Q since the two keywords of interest are much closer together. However, consider a third document, C, in which “hello” appears on page 7 and “science” appears on page 100. Intuitively, this is not much worse than B; both documents are simply not that relevant, and B is at best only minutely more relevant.

Mathematically, these intuitions are implemented in the following way. Let Q be the set of query terms, let Q’ be the subset of Q that exist in the given document, and let *s* be the sum of the minimum pair-wise distances between terms in Q’.

$$MinDistScore(Q', s) = \ln\left(\alpha + \gamma \cdot \exp\left\{-\frac{\beta s}{|Q'|^{\theta}}\right\}\right),$$

where $\alpha, \gamma, \beta > 0$. The original MinDist scoring function is equivalent to *MinDistScore(Q’, s; γ = 1, β = 1, θ = 0)*.

To see if this function matches our expected intuition—that it should be a strictly decreasing function that flattens out as *s* increases, it may be instructive to consider the limits and partial derivative of *MinDistScore* with respect to *s*.

As *s* converges to 0, *MinDistScore* converges to $\ln(\alpha + \gamma)$. As *s* converges to ∞, *MinDistScore* converges to $\ln(\alpha)$. To see if these end points are the maximum and minimum values respectively, let us consider the partial derivative with respect to *s*.

$$\frac{\partial}{\partial s} MinDistScore(Q', s) = -\frac{1}{|Q'|^{\theta}}\left(\frac{\beta\gamma \cdot \exp\left\{-\frac{\beta s}{|Q'|^{\theta}}\right\}}{\alpha + \gamma \cdot \exp\left\{-\frac{\beta s}{|Q'|^{\theta}}\right\}}\right)$$

This function, for all positive values of *s*, is negative. The smaller *s* is, the more negative it is; and the larger *s* is, the less negative it is—it approaches $-\frac{1}{|Q'|^{\theta}}\left(\frac{\beta\gamma}{\alpha+\gamma}\right)$ as s approaches 0 and approaches 0 as *s* approaches ∞. This matches the desired intuition; for small *s*, a small increase in *s* corresponds to a large decrease in *MinDistScore*; and, for large *s*, a small increase in *s* corresponds to small decrease in *s*—in other words, its graph flattens out as *s* increases. This matches our intuition.

It is also reassuring to note that for large |Q'|, the function will decrease less rapidly than for small |Q'|, which is the desired behavior. Recall that |Q'| corresponds to matching more of the terms in the query. Thus, we do not wish to unduly penalize a document which contains more of the query's terms but spread out over a larger region.

In the experiments, for the secure indexes we bind the MinDistScore function with the parameter values MinDistScore(Q', s; $\alpha = \frac{3}{10}, \gamma = 3e, \beta = \frac{1}{100}, \theta = 1$), For the canonical index (index with perfect information), all but the β parameter is the same; β has been set to $\frac{1}{50}$ instead of $\frac{1}{100}$. This causes the canonical index's *MinDistScore* to be decrease more rapidly as s increases. However, since every experiment measuring MinDist output does not interact with any other scoring function, like BM25, the ordering of how documents are ranked is the same for any MinDistScore binding as long as β > 0, γ > 0, and they have the same value for θ. Since this is the only aspect of the output that matters in the MinDist* MAP experiments, this difference for β has no effect on the results.

MinDist* is most appropriately used as a way to add proximity sensitivity to already established scoring methods, like BM25. For example, a linear combination of their scores can be used as the final output of a scoring function that is both sensitive to proximity and term frequencies:

$$^{1}\,\mathrm{Score}(D,Q,C) = \alpha_1 \cdot MinDistScore(s,Q') + \alpha_2 \cdot BM25Score(D,Q,C)$$

Since training data is abundant—it is the output from the canonical index for a set of queries on a given corpus—ideally the free parameters (e.g., the parameters in MinDist*) for each secure index would be independently optimized using a supervised learning algorithm to minimize some error measure on actual output versus the expected output. This would make for an interesting area of future research.

Mean average precision (MAP)

MAP is a way to measure a secure index's BM25 and MinDist* output scores. It does this by measuring how closely its outputs matches a canonical (expected) output, as determined by a non-secure index that provides perfect location and frequency information.

The more approximately a secure index represents a document, the less information one can infer about the document from the secure index. Thus, to what extent the secure index can approximate a document while still achieving high MAP scores is an important question.

MAP is calculated by taking the mean of the average precisions on over 30 queries. The precision at k is:

$$precision(q,k) = \frac{|k\ most\ relevant\ docs\ to\ query\ q \cap top\ k\ retrieved\ docs\ for\ query\ q|}{k}$$

[1] If $\alpha_1 + \alpha_2 = 1, \alpha_1 > 0, \alpha_2 > 0$ then *MinDistScore* and *BM25Score* should be normalized s.t. they share the same minimum and maximum values.

The average precision for the top n documents is:

$$avg\_precision(q,n) = \frac{1}{n}\sum_{k=1}^{n} precision(q,k)$$

The mean average precision (MAP) for the top n documents over Q queries is:

$$map(Q,n) = \frac{1}{|Q|}\sum_{i=1}^{|Q|} avg\_precision(q_i,n)$$

Consider the following. Suppose the ranked list of relevant documents to a query is [3, 0, 1, 2, 4], and the retrieved ranked list (by a secure index) is [2, 4, 3, 0, 1]. The precision at k=1 is $\frac{|\{3\}\cap\{2\}|}{1} = \frac{|\emptyset|}{1} = 0$; the precision at k=2 is $\frac{|\{3,0\}\cap\{2,4\}|}{2} = \frac{|\emptyset|}{2} = 0$; the precision at k=3 is $\frac{|\{3,0,1\}\cap\{2,4,3\}|}{3} = \frac{|\{3\}|}{3} = \frac{1}{3}$, the precision at k=4 is $\frac{|\{3,0,1,2\}\cap\{2,4,3,0\}|}{4} = \frac{|\{3,0,1\}|}{4} = \frac{3}{4}$, and the precision at k=5 is$\frac{|\{3,0,1,2,4\}\cap\{2,4,3,0,1\}|}{5} = \frac{|\{3,0,1,2,4\}|}{5} = \frac{5}{5} = 1$. Thus, the average precision is $\frac{0+0+\frac{1}{3}+\frac{3}{4}+1}{5} = \frac{5}{12}$. The mean average precision would simply be the mean of the average precisions for M queries.

Note that the average precision for the last value of k is necessarily 1 if, by that iteration of k, the relevant set and the retrieved set contain the same elements. However, in general, this is not the case; for instance, if the relevant ranked list of documents to a query is [A, B], and the retrieved ranked list is [D, C, B, A], then if the mean average precision goes from k=1 to k=2, the average precision is 0. In my simulation, I do a variation of this.

Suppose the relevant ranked list of documents to a query is [A=0.9, B=0.85, C=0, D=0], and the retrieved ranked list is [A=0.9, C=0.85, D=0.5, B=0]. Then, I calculate the average precision for the top k=3 instead of the top k=4 or top k=2. In this example, document B is not included in any of the precision at k=1 to k=3 calculations.

Finally, in one of the experiments, I conduct a "page one" test, i.e., I find the mean average precision using only the top 10 results. The randomized algorithm does much more poorly in this instance, e.g., with over 85% probability, the mean average precision will be less than or equal to 0.05.

Simulating the Attacker

There are many possible ways an attacker could compromise the confidentiality of the secure indexes and the hidden queries. There is the obvious case where a secret is disclosed to a would-be attacker. Once a secret is known and the attacker has acquired authorization to query the secure indexes, it may, for instance, systematically probe the secure indexes in such a way as to try to classify the secret documents using a trigram language model.

However, I limit my attention to attacks on one of the more vulnerable parts of the system: hidden queries[2]. I consider below two general strategies to compromise hidden query

[2] It is one of the more vulnerable parts of the system because it is more susceptible to substitution cipher attacks; the secure index itself has many safe-guards against such attacks: (1) each secure index uniquely hashes its members with a salt such that the same unigram or bigram in one secure index will look completely different than in any other, (2) each Psi-based secure index maps every string in the universe of strings to a much small bit string (e.g., 10 bits), (3)

confidentiality, cryptographic hash attacks and maximum likelihood attacks. I only consider cryptographic hash attacks, but I conduct experiments using maximum likelihood attacks.

Cryptographic Hash Attacks

Cryptographic hash functions take as input an arbitrary-length string and output a fixed-length string (hash value).

In general, cryptographic hash functions have the following properties:

(1) Pre-image resistance.

Given a hash string h, finding a string *m* s.t. hash(*m*) = *h* should be intractable. Lacking this property, an attacker can observe *h* and find one or more candidate *m*.

On the one hand, in the context of hidden queries, lacking pre-image resistance, an attacker may be able to discern what a target is searching for. On the other hand, effective collision resistance means that any collisions found suggests both that (a) the guessed secret is correct (if not already known) and (b) hash(m) = h implies that the searcher was indeed looking for m.

Thus, there is a case to be made that pre-image resistance is undesirable in this context. Indeed, since most queries will consist of common terms, if the attacker knows any secrets he can hash a dictionary of common terms to discover what other users are searching for[3].

However, if too many collisions on legitimate queries occur, then this may have a negative effect on the accuracy of search results, like BM25 ranking of documents. So, collision resistance represents a trade-off between privacy and accuracy of search results for the kind of attack mentioned above.

I do not explore this trade-off in my experimental design, but a simple and effective approach to exploring it consists of changing the size of the fixed-length output of the hash; if a hash function maps all input to n bits, then a smaller n corresponds to a larger collision rate. For example, if there are 64 query terms which are mapped to 4 bits each, then on average (assuming a good uniform hash function) each term will collide with $64/2^4 = 4$ other terms in the population. How this will in practice effect outputs of interest is difficult to estimate without performing experiments.

(2) Collision resistance.

Finding strings m_1 and m_2 s.t. hash(m_1) = hash(m_2). A general purpose cryptographic hash function should make this kind of search infeasible, but in the context of hidden

frequency and location information is only approximate, and (4) entries in the secure index may be fake.

[3] Note that this assumes the attacker has access to the plaintext hidden query stream. If the hidden query steam is taking place over a secret channel, the secure index server and the attacker must share information to make this an effective kind of attack.

queries, it is irrelevant. That is, it does not compromise the hidden query stream nor the contents of the secure indexes if such collisions are discovered. Thus, for instance, birthday attacks on the hidden terms are not useful even if feasible.

Maximum Likelihood Attack

Instead of mounting an attack that depends on finding collisions, I simulate a different kind of attack.

Suppose there is a sample of n independent and identically distributed observations t – that is, a history of query terms – coming from some distribution f(t), where f is a pmf denoting how probable a randomly sampled query term t is.

$$P[randomly\ sampled\ query\ term\ is\ t] = f(t)$$

Thus, the probability of seeing a particular history of terms t_1, t_2, …, t_n is[4]:

$$P[\bar{t}] = P[t_1 \wedge t_2 \wedge \dots \wedge t_n] = f(t_1)f(t_2)\dots f(t_n)$$

Each term t is mapped to a hidden term h. The objective of the attacker, then, is to find a function g which maps each hidden term h to a term t.

$$g(h) = \begin{cases} t_{i_1}\ if\ h = h_1 \\ t_{i_2}\ if\ h = h_2 \\ \quad\quad \vdots \\ t_{i_n}\ if\ h = h_n \end{cases}, t_{i_j} \neq\ t_{i_k}\ if\ j \neq k$$

To accomplish this goal, I simulate an attacker for which the distribution f(t) is known (and, in the simulation, is a zipf distribution); since this distribution can be estimated by examining queries in an IR system that does not hide the query terms, assuming the attacker can arrive at a reasonable approximation of the true distribution is not unreasonable.

For a given $\tilde{g}$ in g, the probability of seeing a particular history of hidden terms $\bar{h}$ is:

$$P[\bar{h}] = P[h_1 \wedge h_2 \wedge \dots h_n] = \prod_{i=1}^{n} f(\tilde{g}(h_i))$$

To discover the most likely mapping function $\tilde{g}$ in g, the attacker will use maximum likelihood estimation; that is, it will explore the space of g and choose a $\tilde{g}$ which maximizes the probability[5] of seeing $\bar{h}$.

$$\tilde{g} = \operatorname*{argmax}_{g} \prod_{i=1}^{n} f(g(h_i))$$

[4] Since order is irrelevant (i.i.d. distribution), the actual probability is $\frac{n!}{(k_1!k_2!\dots k_n!)} f(t_1)f(t_2)\dots f(t_n)$, where k_i represents how many times t_i appears in the query history set. However, $\frac{n!}{(k_1!k_2!\dots k_n!)}$ is a constant for a given history of n and k, so we can safely ignore it in our maximum likelihood attack.

[5] When simulating the attacker, the log of the maximum likelihood will be used instead.

Since the space of g is O(n!), a subset of the space must be explored which has a high likelihood of finding local maxima. In my simulation, I use a hill-climbing algorithm, in which the neighbors to a point in this space are defined as the interchange any two t_{i_j} and t_{i_k} in $\tilde{g}$ (in other words, a swap).

Note that an excellent initial starting point in this space, especially given a sufficient number of samples, is to collect all of the hidden terms, sort them by frequency, and pair them up to the terms t in f sorted by probability. However, I do not use this initial estimator in my simulation. If I did, this would give an even greater advantage (with respect to mitigating maximum likelihood attacks) to simulations involving multiple secrets or obfuscations.

When we add m secrets per term, i.e., hidden terms for term *t* consist of the set {h(*t*|*secret*$_1$, *secret*$_2$, …, *secret*$_m$}, then g has the form s.t. each plaintext term maps to m hidden terms. Thus, the space of g is now (nm!) instead of O(n!), and it is expected that more samples of hidden terms will be needed for a given level of accuracy (where accuracy is defined as the percentage of hidden terms which have been correctly mapped).

Finally, when we add k obfuscations to the vocabulary of hidden terms (without secrets), g takes the form:

$$g(h) = \begin{cases} t_{i_1}\ if\ h = h_1 \\ t_{i_2}\ if\ h = h_2 \\ \quad\quad . \\ \quad\quad . \\ \quad\quad . \\ t_{i_n}\ if\ h = h_n\ , t_{i_j} \neq\ t_{i_k}\ if\ j \neq k \\ o\ if\ h = h_{n+1} \\ \quad\quad . \\ \quad\quad . \\ \quad\quad . \\ o\ if\ h = h_{n+k} \end{cases}$$

In the above function g, h_{n+1} to h_{n+k} do not actually map to any plaintext term; they all map to class *obfuscation*. Thus, if a specific set of k hidden terms map to class *obfuscation*, there is only one way for each of those hidden terms to be mapped to it. The space for g, then, is $O\left(\frac{(n+k)!}{k!}\right) = O(P(n+k,n))$. This is equal to or larger than n! for all non-negative integer values of n and k; as a degenerate case, when k = 0 (no obfuscations), it reduces to n!.

Obfuscations introduce additional unknowns that either must be given or estimated. As with the distribution of plaintext terms being given, the probability that a random hidden term is an obfuscation term will also be given, i.e., P[obfuscation] = c, 0 < c < 1.

There are many ways to complicate matters for the attacker when dealing with obfuscations, e.g., making it so that the distribution of individual obfuscation hidden terms are similar to the distribution of non-obfuscated hidden terms. However, in the design experiment, each obfuscation term has a uniform probability.

When combining both obfuscations and secrets, the space of g is $O\left(\frac{(nm+k)!}{k!}\right) = O(P(nm+k,nm))$. In any case, the space of g explodes as m or k grows (the original space was already exponential with respect to n). I only consider increasing m or k separately, i.e., if I increase k, I fix m at 1. Alternatively, if I increase m, I fix k at 0.

Secure indexes

There are several different kinds of secure indexes explored in my research. With the exception of the BSIB, none of them have been proposed as far I am aware.

PSI

Perfect-hash **S**ecure **I**ndex. A secure index only capable of answering approximate Boolean queries, i.e., do these terms exist in the given document?

PSI is based upon the perfect hash[6].

(1) Each unigram or bigram s_i in the target document D is concatenated with *n* secrets—$s'_{i,j} = s_i + secret_j, i = 1\ to\ |unigrams \in D \cup bigrams \in D|, j = 1\ to\ k$. Every unigram and bigram in the document will thus be searchable with *k* different secrets.
(2) Each $s'_{i,j}$ is cryptographically hashed—$s''_{i,j} = cryptographic\_hash(s'_{i,j})$[7]
(3) Each $s''_{i,j}$ is concatenated with the document's identifier (e.g., hash of its filename) and is then re-hashed—$s'''_{i,j} = hash$[8]$(s''_{i,j} + doc\_id)$. This prevents the same cryptographically hashed term in different documents from mapping to the same hash value[9].
(4) Each $s'''_{i,j}$ is uniquely hashed by a perfect hash function s.t. $perfect\_hash(s'''_{i,j}) \neq perfect\_hash(s'''_{x,y}) \leftrightarrow i \neq x\ or\ j \neq y$. That is, no collisions among any of the cryptographic hashes are possible. If using a minimum perfect hash, then $\forall_{i,j} perfect\_hash(s'''_{i,j}) \in [1, n \cdot k]$.
(5) Let U be a bit array with at least $n \cdot k$ indices, where each index maps to M contiguous bits. Then, $\forall_{i,j} U\left[perfect\_hash(s'''_{i,j})\right] = hash$[10]$(s'''_{i,j})\ mod\ M$.

Note that because an M bit hash of $s'''_{i,j}$ is stored in U rather than the actual value, false positives on non-member strings x are possible, i.e., if $perfect\_hash(\mathrm{x}) = perfect\_hash(s'''_{i,j})$, then with probability $\frac{1}{2^M}$, $hash(x)\ mod\ M = hash(s'''_{i,j})\ mod\ M$. This equality denotes a false positive. Therefore, PSI represents an approximate set in which false positives occur with conditional probability $P[positive | not\ a\ member] = \frac{1}{2^M}$.

There is a different way in which a false positive may occur when processing n-gram query terms, n > 2 Consider the following. If a document contains the words "A B B D", then the PSI

[6] A perfect hash function for a set S maps distinct elements in S to a set of integers with no collisions.

[7] Default implementation uses SHA256 and non-invertibly maps the hashes to N hexadecimal digits.

[8] A non-cryptographic hash function is preferable for efficiency reasons.

[9] Limits statistical inference to sampling from a single secure index rather than an entire corpus of secure indexes since each secure index has a unique and random way of mapping its unigrams and bigrams to hashes.

[10] JenkinsHash is used in my implementation, but there are many other more perhaps more suitable replacements.

will conceptually represent it as the set {“A”, “B”, “D”, “A B”, “B B”, “B D”}. To determine if “A B” exists in the document, a single set membership test will suffice.

However, determining whether the query term “A B B” exists in the document is more complicated (it does not exist in the set if only unigrams and bigrams are members). To support exact phrase searches larger than bigrams, as in the trigram “A B B”, a biword model is used in which n-gram query terms, n > 2, are decomposed into a set of *n-1* bigram tests, e.g., testing if “A B B” exists is transformed into a a conjunction of membership tests for “A B” and “B B”. If all bigrams test as positive, the n-gram term is said to exist in the document. In this case, “A B B” will correctly test positively. But if the term is “A B D”, then it will test positively both for “A B” and “B D” but nowhere in the document is the trigram “A B D” found. Therefore, this query term would cause a false positive to occur.

Notes:

1) In my PSI-based experiments, I allow for any false positive rate of the form $\frac{1}{2^{\mathrm{M}}}$, where M can be any positive integer. In a more practical implementation, it would be sensible to optimize the special case where M represents a byte-aligned number of bits, e.g., 8-bits or 16-bits, to take advantage of much faster parallel bit-wise operations.
2) PSI is not used in isolation in any of the experiments. Instead, PSIB, PSIF, and PSIP—which build on top of PSI—are tested. In hindsight, it is regrettable I did not include any experiments exclusively for it.
3) I also regret not including a PSI-based index more directly comparable to BSIB, i.e., a secure index which constructs a perfect hash for each block-of-word segments as is done in BSIB.

PsiBlock (PSIB)

PSIB uses the PSI interface to, for instance, check for the existence of query terms, and on top of that provides an interface capable of answering approximate frequency and location requests for query terms.

To construct a PSIB, first a PSI is constructed. Then, the document is segmented into N blocks, and a bit vector of size $N \cdot size(PSI)$ is constructed such that there are N bits assigned to each unigram and bigram in the document. Finally, if a term resides in a block, the corresponding index representing that block in the bit vector for the given term is set to 1. Otherwise, it is set to 0. The larger the N, the more precisely PSIB can locate terms.

Notes:

(1) Bit vectors are used so that byte-alignment in memory is not necessary. For all of the PSI and PSI-based secure indexes, this approach is used to minimize the size of the secure index resident in memory without compression.

(2) The bit vector representing the blocks a term resides in can become very sparse as the number of blocks increase. This can be very inefficient. However, sparse bit vectors can be easily compressed—see compressed bit vectors. I elected not to do this in my experiments, some of which do clearly reveal the need for compressed bit vectors or some other representation.

> Moreover, I allow for an arbitrary number of blocks per document. However, like with the PSI, a more practical implementation could see a significant performance boost if byte-aligned sizes were used instead, e.g., k-1 parallel bit-wise AND operations could determine which blocks contain all the bigrams in a k-gram query term.

The frequency for a query term is approximated by summing the binary digits of the bit vector representing that term's approximate block locations. Note that for unigram and bigram query terms, the pre-computed bit vector may be used, but if the term is an n-gram, n > 2, then the bit vector representing locations for the term is derived from an AND operation on the corresponding bit vector entries for all of the bigrams of the n-gram query term.

The location for a query term is approximated in much the same way as the frequency, except instead of summing the binary digits of the bit vector, a list of approximate locations is returned. For example, if each block is of size m (each block has m words, except the last which may have fewer) and query term *t* exists in blocks 0 and 5, then two locations will be returned, one in the range [0, m-1] and the other in the range [5m, 6m-1].

On false negatives

Unlike the PSI, false negatives are possible because we use the approximate location information to eliminate positive matches that are most likely false positives. However, if an occurrence of a true positive spans two blocks, that occurrence will be eliminated by the algorithm. One solution is to check for whether the bigrams of an n-gram query term exist in adjacent blocks, e.g., for the query term "A B C", if "A B" exists in block *i*, check for "B C" in either block *i* or block *i+1*. For sufficiently long query terms, a chain of adjacent blocks may also be acceptable.

PsiFreq (PSIF)

PSIF uses the PSI interface to, for instance, check for the existence of query terms, and on top of that provides an interface capable of answering approximate frequency requests for query terms.

To construct a PSIF, first a PSI is constructed. Then, the frequency of each member (unigram and bigram) is calculated. These frequency counts are then stored in a bit vector in a memory efficient way.

If exact frequency information for positive examples of unigrams and bigrams is not desirable, then during the construction phase an approximation of the exact frequency may be used instead, e.g., approximate frequency = exact frequency + UNIF(-r, r).

To service frequency requests for unigrams or bigrams, the PSI interface is used to index into the PSIF's frequency bit vector. If the query term is not a unigram or bigram, then the frequency is considered to be the minimum frequency of all of the bigrams making up the query term.

PsiPost (PSIP)

PSIP uses the PSI interface to, for instance, check for the existence of query terms, and on top of that provides an interface capable of answering approximate frequency and location requests for query terms.

To construct a PSIP, first a PSI is constructed. Then, a postings list (a list of positions) for each unigram and bigram in the document is constructed. Finally, the positions in the postings list are randomly changed according some random variate, i.e., approximate position = exact position + UNIF(-r, r). In the experiments, I use a triangular distribution (with a mode equal to the exact position) instead of a uniform distribution. The triangular distribution has less variance and therefore preserves more information about location information.

Frequency requests are serviced in the same way as PSIF using the size of a term's postings list as the frequency. Note that this means the PSIP does not exploit location information to eliminate false positives. This is done for the sake of speed.

If exact frequency information for positive examples of unigrams and bigrams is not desirable, then insert random positions into the postings lists and/or calculate the average position of adjacent positions for a term and use that average position in place of the two positions. Do this as many times as necessary to achieve the desired level of approximation, e.g., if a term appears N times, repeating this N-1 times would result in storing its mean position. Of course, this also effects location accuracy.

To service location requests for unigram or bigrams, the PSI interface is used to index into the PSIP postings lists. For an n-gram query term, a greedy algorithm is used to construct non-overlapping sets each with a diameter less than or equal to some constant that depends on the way in which the postings list was changed by the random variate (e.g., when a triangular distribution is used with a maximum offset of r positions in either direction, then 2r+|*number of words in the query term - 1*| is the diameter). The positions of the query term are taken to be the center of each such non-overlapping set.

Note that since a greedy algorithm is used, this operation is fairly quick as evidenced by experiments consisting of queries with a large terms. Moreover, the use of a more sophisticated algorithm, e.g., an algorithm which produces the maximum number of such non-overlapping sets, is not obviously an improvement in the context of greater accuracy.

Bloom filter secure index (BSIB)

Bloom filter secure index. A secure index capable of answering approximate frequency and location requests for query terms. BSIB uses a Bloom filter rather than a PSI.

Similar to PSIB, BISB is a block-based secure index. A document is segmented into N blocks, and for each block a Bloom filter (another kind of approximate set) is constructed such that the unigrams and bigrams residing in that block are inserted into it.

The frequency for a query term is approximated by counting how many of the N blocks it tests as positive in. If the query term is a unigram or a bigram, a test for membership in the block's Bloom filter is performed directly. Otherwise, a test for the presence of all the n-gram query's n-1 bigrams is performed, e.g., if the query term is "A B C D", then test for "A B", "B C", and "C D". If any do not test as positive, that block does not contain the query term. Otherwise, it is said to contain it.

The location for a query term is approximated by storing which blocks it tests as positive in, and returning a random location for each such block range. For example, if each block is of size m words and query term *t* exists in blocks 0 and 5, then two locations will be returned; one in the range [0, m-1] and the other in the range [5m, 6m-1].

Caching results

Caching previously calculate results would result in significant savings, e.g., whenever a query is mapped to a list of ranked documents, store the mapping in a cache; or, whenever query term *t*'s BM25 inverse document weight is computed, store the mapping in a cache. I had initially planned on using an LRU cache to memoize computations like the above, but then I reasoned that I wanted to actually test the outputs of the various secure indexes without the effect of a cache.

In a practical implementation, a simple LRU cache or some other memorization technique could be used to avoid recalculating results. Since queries will be heavily biased towards a small subset of terms, this would result in significant savings. Since the secure index database server could do this without the user's permission,

Definition of a query

A query represents an *information need*. In practice, it a string of terms, where a term is either a keyword or an exact phrase surrounded by quotes. Consider the following query: <QUERY BEGIN>"doctors without borders" volunteer<QUERY END>. This query consists of two terms: the keyword "volunteer" and the exact phrase "doctors without borders". When conducting a search on a secure index, it will look for that exact phrase and that exact keyword. It will not count "doctors with borders" as a hit since the phrase must exactly match.

So, each query consists of one or more terms and each term consists of one or more words. The actual secure index itself only stores representations of the unigrams (keywords) and bigrams found in the document it represents. Thus, any phrase term larger than two words must be converted into a chain of bigrams. More details on this are provided elsewhere.

Query confidentiality – additional measures

In my experimental design, I only entertain query confidentiality as it goes from the user to the server. Note, however, that the response from the server may also reveal (leak) information about the user and the document.

To mitigate information leaks about what the search user may be interested in, we can look to Oblivious RAM to inspiration. For instance, instead of sending a single query for each query a user is interested in, the client may submit a packet of N queries, only one of which is related to the user's actual query. The N-1 fake queries are strictly intended to obfuscate the user's actual query of interest, much like query obfuscation is designed to obfuscate the actual terms of interest in a single query.

The N-1 other queries in the query packet may consist of terms chosen from a distribution that is likely to result in a lot of document hits (i.e., high relevancy to a large set of documents). In this way, it may be impossible for an attacker to infer which queries are actually of interest to the user.

This comes at the cost of increased resource consumption, e.g., the server must process more queries per legitimate query, increasing processing and network transmission requirements.

Letter n-grams and word n-grams

In my experiments, I use word unigrams and word bigrams as the atomic, indivisible units in my secure indexes. However, this is not necessary; I could go in either direction, supporting larger word n-grams (e.g., inserting trigrams also) to support faster searching on larger phrases, or go in the opposite direction and support units smaller than whole words using letter n-grams.

Consider the string "hello world". If storing letter trigrams, then the following transformation takes place, where * denotes whitespace.

$$list[\text{“}hello\ world\text{”}] \rightarrow set\{\text{"}hel\text{"}, \text{"}ell\text{"}, \text{"}llo\text{"}, \text{"}lo * \text{"}, \text{"}o * w\text{"}, \text{"} * wo\text{"}, \text{"}wor\text{"}, \text{"}orl\text{"}, \text{"}rld\text{"}\}$$

To search for the word “hello”, check for the existence of "hel", "ell", and "llo" in the set. If all three letter trigrams exist, that word is said to exist. As in the biword model, false positives are possible.

With letter n-grams, partial word matches are automatically possible. For instance, if the user wishes to find any words matching “ello”, then simply check for the existence of “ell” and “llo”. In fact, any substring that is three characters or larger can be matched.

Wild-card searching

It is also possible to support wild-card searching, e.g., match any pattern “hello * doctor”, where * represents a word wildcard. To support word wild-cards, In addition to inserting word unigrams and word bigrams, strings of the form “first * third” must also be inserted, i.e., skip the word in the middle. For example:

$$list[\text{“}hello\ my\ good\ doctor\text{"}] \rightarrow set\{\text{“hello”}, \text{“my”}, \text{“good”}, \text{“doctor”}, \text{“hello my”}, \text{“my good”}, \text{“good doctor”}, \text{“hello} * \text{good”}, \text{“my} * \text{doctor”}\}$$

This increases the size of the secure index fractionally while facilitating rapid word wildcard searching. This can, at increasing cost to size, be extended to support multiple word wildcards, e.g., “hello * * doctor”. Character wildcards can be supported by letter n-grams in the same way that word wildcards are supported by word n-grams. All of this, of course, may inflate the secure index beyond what is desirable.

Approximate searching and error tolerance

Approximate searching is a superset of wild-card searching. Other ways approximate searching can be accommodated include hashing the terms of the document in such that like or similar terms hash to the same value, e.g., locality sensitive hashing, phonetic coding (e.g., Soundex, which tends to map words that sound alike to the same hash), or stemming (reducing words to their stem or root form, e.g., {“computer”, “computing”, “computes”, “computed”, “compute”, “computation”, “computations”, “computational”} all map to the same stem, “comput”.

These kind of transformations generally take place during a preprocessing step so that the secure index constructor only sees the final forms of terms. Note that there is nothing preventing a preprocessor from including any combination of these transformations.

Another form of approximate searching can be accomplished through online query expansion, e.g., finding synonyms for words, fixing spelling errors, and so forth. However, these tricks—stemming, synonym expansion, etc.—must be used with caution lest they come at too high a cost in terms of trading precision for recall. Ideally, both recall and precision will be improved, since documents that are relevant to the user’s information need (represented by the query) will be retrieved where otherwise they would not have been.

Experiment Platforms

Machine A	
Operating System	Windows 7 Service Pack 1
Processor	AMD A6-6400K APU 3.9GHz
Installed memory (RAM)	8.00 GB
Storage Device	Kingston SSDNow V300 Series SV300S37A/60G 2.5" 60GB SATA III SSD
Compiler	Visual Studio 2013; 32-bit target; command line = ” /MP /GS /GL /W3 /Gy /Zi /Gm- /O2 /fp:precise /GF /GT /WX- /Zc:forScope /arch:SSE2 /Gd /Oy- /Oi /MD”