Encrypted Keyword Search in a Distributed Storage System

Shay Artzi Adam Kieżun Calvin Newport David Schultz

{artzi, akiezun, cnewport, das}@csail.mit.edu

Abstract

Encrypted keyword search allows a server to perform a search over a set of encrypted documents on behalf of a client without learning the contents of the documents or the words being searched for. Designing a practical system is challenging because the privacy constraint thwarts standard indexing and ranking techniques. We present Mafdet, an encrypted keyword search system we have implemented. Our system makes the search practical even for large data sets. We evaluated Mafdet's performance on a set of queries and a large collection of documents. In these queries, Mafdet's accuracy is within 6% of Google Desktop, and the search time is on the order of seconds for document sets as large as 2.6 GB.

1 Introduction

As people and organizations increasingly rely on distributed services to store large volumes of data reliably and make it globally available, the problem of searching these data becomes more important. For instance, consider a laptop user who encrypts his files and stores them on such a service to avoid data theft or loss, or a small organization that uses distributed storage for archival. Both of these clients may need to perform searches over the documents they have stored; however, this is a challenging problem because the storage service does not and should not know the encryption key, and the clients cannot reasonably download all of the potentially relevant documents over the WAN.

Several prior schemes (e.g., [5, 10, 18]) address the encrypted search problem on a small scale, usually involving a PDA as the client and a personal computer as the storage service. We propose a widely-distributed storage service that uses encryption to keep the servers oblivious to the content of the database, while allowing clients who have the decryption key to generate *trapdoors*, which the service can use to perform keyword searches of the database on the clients' behalf. Our system provides strong privacy guarantees. Short of breaking the encryption, the service has no way to learn anything about the content of keyword queries or of the documents being stored, except possibly the lengths of the documents, the list of documents in each search result, and the limited statistics used to rank results.

The traditional technique for performing searches efficiently is to maintain an index that maps keywords to document identifiers. However, as we discuss in Section 2.3, direct application of this approach is insecure. Our solution instead relies upon per-document metadata that servers use, in conjunction with trapdoors, to determine which documents match the keyword that generated the trapdoor.

This technique introduces two main difficulties. First, the cost of performing a search is linear in the total number of documents, whereas inverted indices have performance that is linear in the number of search results. Second, the need for privacy complicates the implementation of search result ranking mechanisms. Without access to the contents of the documents, it is unclear how to decide which results best match a given query.

Our main contribution in this work is a system that addresses these two problems. Our architecture makes searches fast by taking advantage of multiple servers and caching. We also introduce several novel relevance schemes that rely on carefully selected extra values stored in our encrypted metadata to allow for accurate relevance decisions to be made by the untrusted server. We show that the system is usable at a scale of 100,000 documents, both in terms of efficiency and accuracy.

2 Encrypted Keyword Search

In this section, we describe the properties required of a secure search service, and the threat model that we use. We then exhibit a proposal that is simpler than our design and show why it fails to achieve these properties. In Section 3 and following, we describe our architecture and show how it achieves our goals. Section 4 describes the prototype of the system that we have implemented.

2.1 Properties

Since seemingly small information leaks can often be exploited by a malicious adversary to reveal significant information, we view privacy as a constraint and build on established definitions of security. Subject to these requirements, we have built a system that can scale well to large data sets and provide accurate search results. Below, we give an informal description of the desired properties, deferring a principled discussion of security to [10, 18].

2.1.1 Privacy Constraints

- **Controlled searching.** The server cannot learn anything about the contents of documents, except when the client performs a search.
- **Hidden queries.** The client can search for a set of documents containing a keyword without revealing the keyword to the server.
- **Query isolation.** From the query result, the server learns nothing about the plain text other than the set of documents that match the query (and possibly the limited statistical information used to perform ranking).
- **Update isolation.** The server learns nothing more from updates than it would if we maintained no additional metadata for the purpose of performing searches.

2.1.2 Performance Goals

- **Low query latency.** Although searching encrypted data is fundamentally harder than searching public data, queries against large data sets should take on the order of seconds, not minutes or hours.
- **High server throughput.** The system should be capable of processing many queries concurrently without significantly increasing the number of search servers.
- **Support for boolean queries.** The query language must be expressive enough to allow users to issue precise queries. As with most search engines, we dismiss regular expression searches and focus on simpler boolean queries. In practice, we focus on the `and` operator; `or` and `not` are relatively trivial.
- **Competitive search accuracy.** Information retrieval systems employ a wide variety of heuristics based on query keyword proximity and frequency within documents, and overall keyword frequency, and other metrics to identify the most relevant matches. For a large data set, it is crucial that our system be capable of supporting enough of these heuristics to produce accurate search results.

2.2 Threat Model

We model the adversary as an eavesdropper who knows everything that the storage service knows. However, the adversary is passive; she cannot compromise the integrity of the results produced by the service. This assumption may seem limiting, but it fits well with real-world systems such as BFT [4], which preserves integrity but not secrecy in the face of active attacks involving less than a threshold number of replicas.

We assume that the adversary has *no knowledge* of the contents of the documents stored by the client. Ideally, one would like to protect against a stronger adversary who knows keywords in some of the documents and would like to learn information about other documents. The trouble with this stronger model is that queries that match documents the adversary knows fundamentally leak information about documents she doesn't know.

This limitation is a legitimate concern in designing a robust system, and we attempt to mitigate it in our design. Our system decouples the file servers, which store and retrieve documents, and search servers, which store search metadata and execute searches on behalf of clients. An attacker who compromises only the search servers will be unable to use document access patterns to deduce which of the encrypted documents corresponds to the plaintexts she knows.

2.3 A Straw Man Proposal

One possible way to design a system for encrypted search involves a simple variant on an inverted index, in which the keys in the index are encrypted keywords rather than plaintext keywords. Inverted indices seem to be a natural solution because they are efficient and well-studied.

Unfortunately, inverted indices are insecure in the presence of updates. This is because adding or removing a document requires that the index be modified in the locations corresponding to the keywords in that document, so the server is able to correlate the document with other documents that share keywords with it. Hence, inverted indices violate the update isolation property of Section 2.1.1. Since every document has to be uploaded at some point, the amount of information inverted indices leak is comparable to what the server would learn if the user searched for every keyword in every document.

Without update isolation, a malicious server can employ statistical attacks against the *entire document set* to infer partial information about contents of documents. Update isolation limits statistical attacks to the specific keywords the user searches for, which is presumably a small fraction of the total dictionary size. (According to [1], at least 75% of words in web pages indexed by Web search engines are never searched for.) Furthermore, update isolation allows users to heuristically improve security by re-encrypting the database or parts thereof periodically (e.g. after a particular number of searches) to confound the would-be snoop.

Indices may be appropriate for certain applications where statistical attacks are deemed inconsequential. However, our goal in this work is to construct a robust, general-purpose system, so we eschew indices.

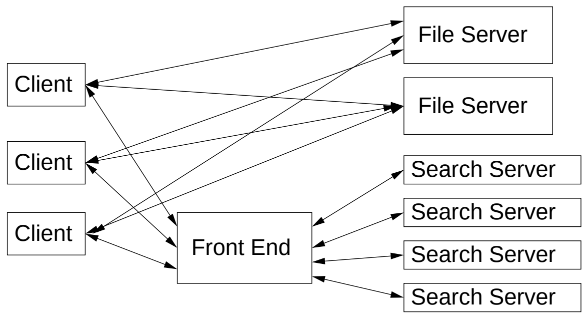

Figure 1: Mafdet physical architecture. One front end allowing several clients to securely search and retrieve documents.

3 Mafdet Architecture

In this section, we describe Mafdet, a secure keyword search service we designed. Logically, Mafdet consists of two separate services: a search service and a document storage service. The mechanisms used to implement these two services are different, so we logically view each service as being implemented by separate search servers and file servers, as shown on the right side of Figure 1. In practice, the servers could be mapped to physical machines in a way that makes most effective use of hardware resources.

3.1 System Components

Front End The front end acts as a proxy for all requests. It performs load balancing and failover and manages the mapping of documents to the file and search servers responsible for them. The **upload**(*file*,*filter*[1]), **download**(*docId*), **delete**(*docId*) operations store, retrieve, and delete documents, respectively, and **search**(*query*) performs a search.

File Servers Our system is compatible with virtually any file server architecture, and we do not consider in detail how file servers might interact with our search service. Unlike the scheme of [18], the mechanisms we use to perform searches do not require or benefit from knowledge of the encrypted file contents, so this aspect of the system is orthogonal to our main contributions.

[1]Filters are per-document metadata generated by the client, as discussed in section 3.2.

Search Servers Search servers store per-document metadata, they use this metadata to locate the documents that contain search terms, and they cache the results of previous searches. Search servers do not perform any ranking computation. They only collect the number of times each query word appears and in which parts of the document it appears. This information is later used by the front end to compute relevance ranking.

3.2 Search Metadata

Several schemes for performing private searches have been proposed. We adopt a scheme proposed in [10] because it supports our security goals, and because unlike other systems (e.g., [5, 11]), it does not require that the full set of potential keywords are known in advance, before uploading any documents.

Our system associates a Bloom filter with every document. A Bloom filter [2] is an efficient data structure for probabilistically determining set membership. It supports two operations, **insert**$(set, element)$ and **search**$(set, element)$. Naïvely, one might imagine simply inserting all the keywords that appear in a document into the filter. However, this violates the controlled searching property of Section 2.1.1 by allowing servers to search for arbitrary keywords without authorization of the client.

Instead of inserting the actual keywords into the Bloom filters, we insert keyed hashes of the keywords. The key is private to the client, so servers are unable to compute these hashes on their own. We call the hash values that make up the sets represented by the Bloom filters *trapdoors* because clients can give them to the servers to allow the servers to locate documents containing the corresponding words. Thus, given a keyed hash H_k, the trapdoor for word w is simply $t_w = H_k(w)$.

Bloom filters also use hashing internally to determine determine which bits in the filter might correspond to a particular member of the set. Our construction is standard, with two exceptions: (1) we require a cryptographic hash and (2) the document identifier is also included in the hash so that Bloom filters for different documents cannot be correlated. Given a trapdoor t_w from the client, the server computes $H(d, t_w, i)$ for each document d and each Bloom filter row i, then tests the bit in the corresponding bitmap.

Bloom filters for larger documents are larger than filters for smaller documents in order to achieve a roughly uniform density and false positive rate. This reveals the size of each document to the search server, but we believe that this leak is acceptable, since file servers already have this information. However, the density of bits set in all the filters must be identical to prevent servers from learning the density of keywords in each document. This is achieved by setting random bits in filters that have few bits set.

3.3 Optimizations

An important goal of this study is to explore the practical performance constraints of a secure search system. To maximize the performance of our search operation we propose several optimizations:

- **Previous search caching** Mafdet caches the results for keywords previously searched for. Cache entries are of the format

  $\langle$*trapdoor*, *docIds* and *ranking info*, *timestamp*$\rangle$

  The timestamp is the arrival time of the query at the front end. If the same keyword is searched for later, only documents that were uploaded later are searched in full. We place no limit on the size of the cache, since loading a cache entry from disk is still cheaper than scanning the Bloom filters for every document. The effectiveness of this cache is discussed in section 5.5.

- **Fresh Bloom filter caching** Our previous search caching scheme generates scenarios where only a small number of Bloom filters might need to be searched. Accordingly, the latency of queries on recently cached keywords can be significantly increased if we keep the small number of Bloom filters that need to be searched resident in memory. We explore techniques to keep a reasonable number of the most recent Bloom filters resident in memory even though the search server may be concurrently conducting multiple searches that read through a large number of disk resident filters (and thus cycle a lot of data through main memory).

- **Dynamic load balancing** Two types of decisions are involved in load balancing: a long-term *replica set selection* policy determines which servers store newly-added documents, while a dynamic *replica selection* policy determines which of the replicas for a given document process an incoming query for that document.[2] The mapping of documents to replicas is soft state, as it can be reconstructed quickly by the replicas themselves and stored on a newly-elected front end should the original front end fail; however, the mapping may also be replicated for high availability.

4 Implementation

The core of our prototype consists of about 3100 lines of Java, plus several hundred lines of supporting scripts and 1500 lines of Java for our testing and evaluation infrastructure. Our protocols are layered on top of Java's Remote Method Invocation (RMI) infrastructure. The prototype implements search servers, a front end with ranking (see Section 4.1) and static load balancing, client infrastructure for creating filters and trapdoors, and command line and CGI user interfaces. The search servers store Bloom filters on the local file system, and they also cache ranking information for each term previously searched for. Clients are multithreaded and can construct and upload multiple Bloom filters in parallel. Servers process each operation in a separate thread, but do not use more than one thread for an individual search or upload operation.

4.1 Ranking Search Results

For a given query, the front end server assigns each search result a relevance score based on information stored in the Bloom filters. The documents are returned to the client in descending order of these scores. Specifically, relevance is calculated on four main criteria: *query word proximity*, *occurrence count*, *similarity*, and *distinct query word count*. Below, we briefly explain each of these metrics and how we store the relevant information in the Bloom filters.

[2]Although this is an important practical consideration, our test environment uses a homogenous set of unreplicated servers, so our prototype uses a static policy.

Query word proximity favors documents that contain multiple query keywords in close proximity. Inspired by a technique often used in Internet search engines, we divide each document into 64 chunks, and then store occurrence information for each chunk and each keyword separately. For example, if the keyword foo occurs in document d in chunks 5 and 20, we store in d's Bloom filter the bits corresponding to $(\mathsf{foo}, 5)$ and $(\mathsf{foo}, 20)$. In Mafdet, each pair of query keywords that appears in the same chunk adds a premium $P_1 = 7$ to the total relevance score.

Keyword occurrence counting favors documents that contain query keywords many times. We found that occurrence counts are more important in single-word queries, when proximity information cannot be used. Therefore, Mafdet assigns a premium $P_2 = 1$ to each occurrence of a query word for single-word queries, and a reduced premium $P_3 = 0.1$ for each keyword occurrence in multi-word queries. To avoid increasing the size of the Bloom filter needlessly, we only count occurrences up to a threshold of 7.

Similarity favors the occurrence of uncommon words over common words (very common words are filtered out altogether, see Section 4.2). Following a well-known approach [9], we score similarity for given query and document with the following formula: $\sum_{i=1}^{Q}(P_4 + \log \frac{N-n_i}{n_i})$, where Q is the number of query keywords in the document, $P_4 = 5$ is a bonus constant, n_i is the number of documents with keyword i, and N is the number of documents returned by the query.

Distinct query word count favors the appearance of multiple distinct query keywords over one keyword appearing multiple times. To implement this metric, Mafdet rewards the presence of each distinct query keyword in a document with a premium of $P_5 = 8$.

In addition to these four criteria, the system also awards a large bonus $P_6 = 100$ to any document that contains *all* keywords in the query. The values of our constants P_1 to P_6 were determined empirically through experimentation with our training queries.

4.2 Common Keyword Removal

Since the set of Bloom filters may be to big to fit in the memory of the search server, they are stored on the local disk. To limit to effect of I/O on search performance, Mafdet allows the client to send a compressed Bloom filter and it also ignores common words while creating the filter. To that end, it uses a list of words that appeared in more than 50% of documents in a randomly selected sample of 200 from our experimental corpus. Commercial search services use similar algorithms to achieve this goal, but unlike our system, they also allow users to search for phrases such as “The Who”. Ignoring common words has also a positive impact on relevance ranking. Intuitively, if a word appears in very many documents in the corpus, its presence in a document should have no or very little effect on the document’s relevance. This is related to the similarity score described in Section 4.1.

5 Evaluation

We evaluate our search system in terms of its *efficiency* and its *accuracy*. The former concerns the search latency, while the latter concerns search quality. A well-designed interactive search service must offer good performance for both of these attributes, even though, in practice, optimizing one is often at the expense of the other.

5.1 Accuracy Metrics

Following the lead of earlier search service evaluation studies (e.g., [8, 13, 14, 15]), we measure accuracy using the following three metrics: *Recall*, *Precision*, and *Average Precision*. Recall measures the percent of relevant documents returned by a search, precision measures the percent of returned documents that are relevant, and average precision measures how effectively the system clustered the relevant results at the top of the total list of returned results. Arguably, average precision is the most interesting of the three as users might not mind a large result set as long as the most relevant hits are clustered at the top. See [15] for a more detailed explanation on how these metrics are calculated.

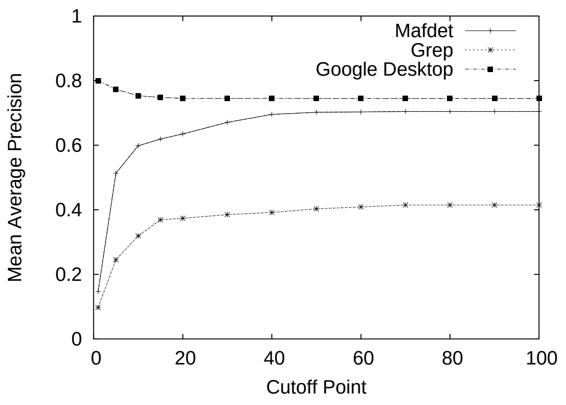

Figure 2: Mean Average Precision over test queries, calculated for Mafdet, Google Desktop, and Grep at fixed cutoff points.

5.2 Accuracy Experiments

We compare our accuracy scores against conjunctive Grep (return documents with all keywords in query)[3] and Google Desktop 20050325 [12]. Google Desktop uses many state-of-the-art information retrieval techniques, and therefore provides an appropriate optimized ceiling for our accuracy evaluation. Similarly, Grep’s simple strategy provides a baseline for our results. Our goal was to achieve accuracy comparable to Google and significantly better than Grep.

For the accuracy experiments, our test corpus consisted of 2000 text excerpts drawn at random from Project Gutenberg [16]. The size of the files were altered to fit a normal distribution with a mean of 500 lines ($\sim$20 kB), and a standard deviation of 100 lines. The number and size of the files was chosen to roughly approximate the text contents of a typical user’s personal computer. Following known IR evaluation methods, we constructed two sets of ten queries each, using the first set to train the system, and the second set to obtain the actual evaluation results presented in this section.

5.3 Accuracy Results

To evaluate the effectiveness of our ranking algorithm, we measured the mean average precision of Mafdet, Grep and Google Desktop at various cutoff points (at cutoff point n, we consider only the first

[3]For the accuracy experiments, we used an implementation of Grep’s functionality within Mafdet.

	Avg. Prec.	Recall	Prec.
Mafdet	0.702	0.876	0.280
Google	0.746	0.438	0.528
Grep	0.401	0.662	0.485

Figure 3: Mean Accuracy Results at cutoff 50.

n results returned by each system). The results are shown in Figure 2. Google Desktop's web legacy is clear. Its ranking algorithms places strong emphasis on presenting only the best quality results near the top ("I'm Feeling Lucky"). In our experiments, Google gets the first result right in 80% of queries, compared to 14% for Mafdet and only 9% for Grep. While Mafdet might not always get the first result right, overall it still does a good job of clustering relevant results toward the top. By a cutoff point of 20, Mafdet's mean average precision is already a factor of 1.70 higher than Grep's and only a factor of 1.17 worse than Google. By a cutoff of 50, at which point the results stabilize for all three systems,Mafdet is only a factor of 1.06 worse than Google and a full factor of 1.75 better than Grep.

Google's aggressive emphasis on returning only very high quality results is obtained at the expense of recall. As shown in Figure 3, Mafdet returns, on average, twice as many relevant documents as Google. Our belief is that for a desktop search scenario, a typical user will be happy to exchange a few more irrelevant documents showing up at the top of the results for a much better recall over all relevant documents. We also notice in Figure 3 that Google boasts a stronger precision value than Mafdet. This too is an expected effect of Google's aggressive filtering. As mentioned, however, Mafdet has a comparable average precision value. This indicates that our system is able to effectively move most of the irrelevant results to the bottom of the ranking, thus mitigating much of the potential negative effect of our lower precision.

Figure 4 presents the average precision results for the three systems for each of the ten test queries, running at cutoff 50. Notice that Mafdet performs close to Google Desktop. Our system matches or beats Google on 50% of the test queries. By contrast, Grep performs worse than Mafdet on 80% of the queries. Queries Q4 and Q5 are of particular interest as Google returns an average precision score

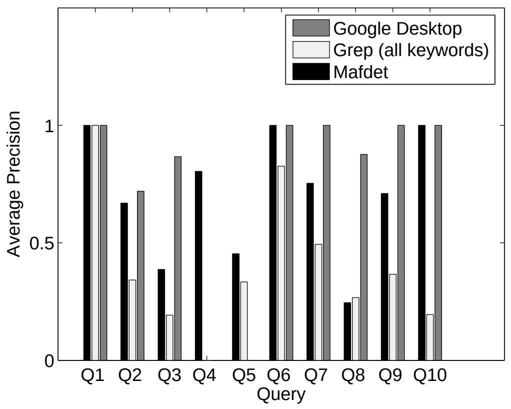

Figure 4: Average Precision results over 10 test queries, calculated for Mafdet, Google Desktop, and Grep at cutoff 50.

of 0 for both. A closer examination shows this result is another casualty of Google's aggressive filtering.

5.4 Tuning Bloom Filters

As described in Section 3.2, the sizes of Bloom filters in Mafdet are proportional to the sizes of the corresponding documents. The factor that relates the two is an important system parameter.[4] If the filters are too dense, users will see an unacceptable number of false positives, and if they are too sparse, then searches will take longer. Furthermore, we want to optimize the number of hash functions used by the filter. It is well known that for a filter with n elements represented with m bits, the minimum false positive rate is achieved with $\frac{m}{n} \ln 2$ hashes. However, hash computations are a performance bottleneck in Mafdet, so we would like to use as few as possible without significantly impacting accuracy.

We evaluated the affect of filter size on average precision using the same test queries as in Section 5.3. The results are presented in Figure 5. Since our ranking metadata occupies a variable number of bits per word depending on word distribution, the absolute value of the parameter on the x-axis has no meaningful interpretation. In our final system we set the density parameter to 16 and use 4 hash functions

[4]Ideally, the fi lter size would correspond to the actual number of elements to be stored, but we use document length instead to prevent the server from inferring anything about average word length or proximity data from the fi lter size.

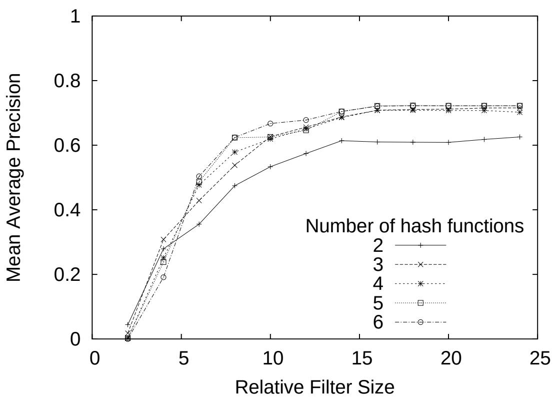

Figure 5: Affect of Bloom filter parameters on mean average precision at cutoff 50 with 6,802 documents.

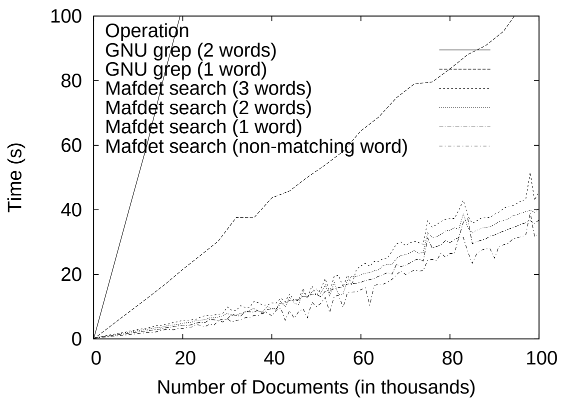

Figure 6: Operation Latencies with respect to number of documents.

based on this graph; this results in filter bitmaps that are 8–12% full, for an aggregate false positive rate on the order of 10^{-4} and virtually no degradation in relative accuracy.

A theoretical analysis in [10] suggests that much smaller filters are possible, but we disagree based on our empirical results. In particular, even seemingly modest false positive rates such as 1% are problematic for queries with high selectivity. For instance, if 5 documents out of a set of 20,000 contain a particular keyword, then a false positive rate of 1% gives 200 false positives; the user gets 205 results, most of them bogus. We also note that the exact impact of false positives on our system can only be measured empirically, since interactions between false positives in the Bloom filters and our ranking algorithm are complex.

5.5 Scalability

To determine the environments in which our system is practical, we examined the query latency of our prototype with respect to the size of the document set. Figure 6 shows the search latency of 4 search servers running on identical machines with dual-2.8 GHz Pentium 4 Xeon processors, 2 GB of RAM, and two 15k RPM SCSI disks in a RAID 0 configuration. We ran the experiments using the HotSpot Java virtual machine build 1.5.0_02-b09 under Linux kernel version 2.4.20-30.9. The VM was restricted to 64 MB of heap space, but we did not limit the size of the kernel's buffer cache. For comparison, we also partitioned the data set into four equal-size chunks and ran 1- and 2-word searches using GNU `grep` 2.5.1 and Google Desktop.

The documents were English text files from Project Gutenberg, truncated to lengths ranging from 8 kB to 56 kB, and with an average length of 27 kB (fitting the same size distribution used in Section 5.3). We formulated 1-, 2-, and 3-word queries from 6 words, each word matching about 1/5th of the document set. The words occurred approximately independently, so nearly twice as many documents matched the two keyword query as compared to the one keyword query. We also tried a single-word query that matched no documents, which captures the search overhead independent of ranking and aggregation.

Figure 6 shows that even for 100,000 documents, searches are processed in well under a minute, and search time scales linearly in the number of documents. Moreover, even as the size of the result set scales with the number of documents in the 1-, 2-, and 3-word tests, the ranking and aggregation overhead at the front end is low. The figure also shows that we significantly outpace `grep`; `grep` must scan the entirety of each document, but Mafdet only needs to look at 4 bits in each Bloom filter. All searches using Google Desktop took well under a second and are not shown. Jitter in the Mafdet results is due to garbage collection on the search servers; the spikes are regularly spaced because data points for the graph were collected in a single left-to-right pass.

Since we envision the storage service being a shared resource, query throughput is also an important metric. At 100,000 documents, the null query time is about 32.81 seconds, so our throughput is 0.03 queries per second. However, our search servers execute each query in a separate thread, so by running 16 queries in parallel we achieved a throughput of 0.06 qps using two CPUs. Profiling data indicates that the main sources of search server overhead in our prototype are unmarshalling Bloom filters read from disk (using Java's serialization framework) and computing cryptographic hashes. Therefore, heavier optimization and cryptographic acceleration hardware are likely to yield significant scalability improvements.

Our evaluation above is actually relatively pessimistic in that they do not take our caching scheme described in section 3.3 into account. Caching is not particularly important for search latency, since the system needs to be acceptably fast in the uncached case to be usable anyway; however, caching has a significant impact on scalability. If the throughput without caching is q_{uc}, the cache hit rate is h, and a cache hit is s times faster than a cache miss, then the throughput with caching, q_c, is given by

$$q_c = q_{uc} \frac{s}{h + s - hs}$$

For a single-word search on sets of 3130 and 6260 documents, a cached Mafdet search was a full 6.3 times as fast as the uncached search, so suppose $s = 6.3$. An analysis of a 43-day AltaVista query log [17] concludes that each search occurs in the log an average of 3.97 times. This gives a conservative lower bound of $2.97/3.97 = 0.748$ on the hit rate h, since [17] considers entire searches whereas we cache results for individual search terms. The result is an improvement in throughput by a factor of 2.7. Given the results for our set of 100,000 documents as presented above, caching should allow us to achieve 0.16 queries per second with 4 search servers.

We also measured search and upload time relative to the number of search servers. Our results in Figure 7 show that search time is proportional to the number of search servers. When search time levels off at about 6 servers, the non-matching search line corresponds to fixed sources of overhead such as wire time and starting a Java VM on the client. The difference between this line and the single word search line is the time spent by the front end performing ranking and aggregation. Upload time is constant, since most of the time is spent hashing each keyword in each document at the client.

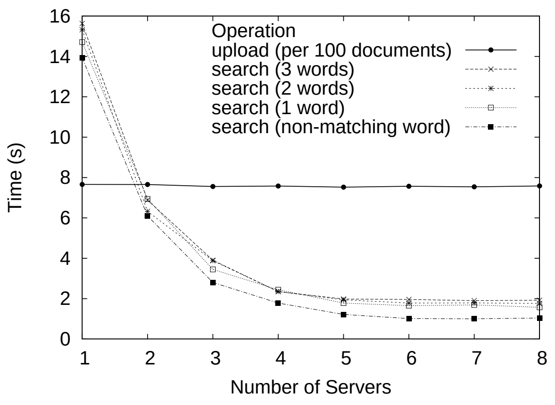

Figure 7: Operation Latencies for a set of 13,924 documents, with respect to number of search servers.

6 Related Work

With the increasing popularity of data-storage outsourcing, there has been, in recent years, a growing interest in techniques related to searching on encrypted data. Most of the related work has been theoretical and, to our knowledge, no practical system is available, in which the techniques are implemented and tested under realistic conditions.

The problem of searching on encrypted data was first addressed by Song *et al.* [18]. They present a symmetric-key scheme that allows users create a trapdoor for a keyword and test whether a given document contains that keyword. The algorithm works by sequentially scanning the document and has $\mathcal{O}(n)$ performance for a document of size n. Boneh *et al.* [3] propose a public-key equivalent of this scheme. Neither paper provides experimental data to verify the usefulness of their schemes in a practical system.

Two schemes for searching on remote encrypted data, developed by Chang and Mitzenmacher [5], require no scan of the documents. They rely, however, on the assumption that an index of all keywords that can appear in the document set is known *a priori*. Uploading new documents in their scheme is possible only under this assumption. Song *et al.* [18] also suggest a similar system.

The scheme proposed by Goh [10] is the most closely related to our work. It employs Bloom filters as an implementation of a *secure index* that can perform the membership testing in $\mathcal{O}(1)$ time. His scheme does not require pre-building the list of all keywords and can be modified to support occurrence search. Goh reports no experience with performance of this scheme in a real system.

Private Information Retrieval protocols (e.g., [6, 7]) allow users to access a database without revealing the retrieved data to the administrator. The theoretic security bounds are very strong but the algorithms require substantial computational and communication overhead.

References

[1] R. Baeza-Yates. *Web Mining: Applications and Techniques*, chapter 14. Idea Publishing Group, 2005.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proc. EuroCRYPT 2004*, pages 506–522, 2004.

[4] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[5] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive, Report 2004/051, 2004. `http://eprint.iacr.org/`.

[6] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, 1998. `http://eprint.iacr.org/`.

[7] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[8] H. Chu and M. Rosenthal. Search engines for the world wide web: A comparative study and evaluation methodology. In *Proc. ASIS '96*, 1996.

[9] W. B. Frakes and R. A. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[10] E. Goh. Secure indexes. Internet, May 2004. `http://crypto.stanford.edu/~eujin/papers/secureindex/`.

[11] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In M. Jakobsson, M. Yung, and J. Zhou, editors, *Proc. of the 2004 Applied Cryptography and Network Security Conference*, pages 31–45. LNCS 3089, 2004.

[12] Google Desktop. `http://desktop.google.com`.

[13] D. Hawking, N. Craswell, P. Bailey, and K. Griffihs. Measuring search engine quality. *Inf. Retr.*, 4(1):33–59, 2001.

[14] H. V. Leighton and J. Srivastava. First 20 precision among world wide web search services (search engines). *J. Am. Soc. Inf. Sci.*, 50(10):870–881, 1999.

[15] K. Mahesh. Text retrieval quality: A primer. `http://www.oracle.com/technology/prodcuts/text/htdocs/imt_quality.htm`.

[16] Project Gutenberg. `http://www.gutenberg.org`.

[17] C. Silverstein, M. Henzinger, H. Marias, and M. Moricz. Analysis of a very large altavista query log. Technical Report 014, Digital Equipment Corporation Systems Research Center, 1998.

[18] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.