INF-3996
Master thesis in
Telemedicine and E-health

Preventing Information Leakage in the Search Engine

by

Joseph Hurley

January 23, 2009

Faculty of Science
Department of Computer Science
University of Tromsø

Abstract

This thesis covers the design, implementation, and evaluation of a search engine which can give each user a customized index based on the documents they are authorized to view. A common solution available today for this situation is to filter the results of a query based on the list of documents a user has access to. In this scenario, it is possible for information to leak from the search engine because the filtering takes place after the results are ranked. Ranking algorithms are usually based on information which considers characteristics of the entire corpus when calculating the score a document will receive. This type of information in the index must be cleaned before it is used to judge the relevant documents for a user query, otherwise data leakage is possible. Cleaning this information at query time might have a dramatic effect on query performance which would discourage use. The work presented here takes this sensitive information and calculates it for every user authorized to view the documents at index time. At query time, the search engine uses a filtered global index for selecting relevant results, but ranks the results using the information stored for the individual user's authorized view of the index. Different designs are compared, but the same concept is present in all implementations. An open source index, Apache's Lucene, was used as a starting point for this work. All modifications were made to Lucene and then compared to an unmodified Lucene for performance evaluations. The findings are that it is feasible to include access control in a basic search engine without incurring dramatic loss in performance.

Acknowledgements

I would like to thank my advisors Åge Kvalnes, and Dag Johansen for their encouragement, support and productive discussions. In addition, Håvard Johansen was frequently involved as well. His interest and suggestions were quite helpful and encouraging.

The rest of the iAD group also deserve my thanks. Their support and encouragement was invaluable.

Finally, I would like to thank my girlfriend Conny Tümmler for her support through this time.

Contents

Chapter 1

Introduction

1.1 Background

Search has become a ubiquitous tool in today's world. People are comfortable inputting 2-3 terms [15] which should distinguish a document they are after. Typically, they are given a list of ranked results, and from this list, they can access the document they are searching. If the query was not successful enough, the list of results might help the user to reformulate it, and try again until they find the document in question. This technique is great for cases where the user knows or suspects a document's existence, but is not sure where to find it [14]. In some cases, the user may know where to find it, but chooses the search interface as a quicker means of accessing the document. In other cases, the user may not even know what they are searching for, but interaction with the search engine can help lead them down the path to satisfying their needs [2].

Search became familiar to most people with the recent boom of Internet connectivity. Internet search engines are typical entry points for many users accessing the web. But search can be used in many other contexts. Enterprise search engines are designed to hold a smaller set of documents, usually specialized to the nature of a particular organization. The content is quite different from those indexed on the web [12].

There are many contexts where a search engine would be an ideal means for users to locate documents. The enterprise falls into this category as resources are heterogeneous and often scattered across many different servers at various locations [7]. Some of these resources may have limited visibility to the members of their organization as defined by an access control policy. In these cases the search engine as is cannot be used to index all documents because some content or even a document's existence must be hidden from unauthorized users or the access control policy may be violated [4].

The easiest solution to use search in this situation is to add a filter to the search engine which obeys an access control list and removes any sensitive documents from the results returned to a user. This approach leaves much room for improvement as it is possible to leak information from this type of index [4]. This is the starting point for the work presented in this thesis.

There is a strong theoretical base in data security, particularly regarding the reverse engineering of information sources to extract information which would be otherwise hidden. [6] presents findings that hidden information can be extracted from databases with clever queries. This and other research in this field are the theoretical inspiration for this work.

1.2 Problem Definition

This thesis shall cover the design and evaluation of a search engine that is resistant to data mining techniques, specifically the reverse engineering of the ranking algorithm. This particular problem arises in the context where the indexed documents are protected by a security model. The search index is optimized for speedy lookups and a near instantaneous response to user queries. The proposed solution's performance must be evaluated to verify that it is not violating these characteristics of the search engine.

1.2.1 A Brief Experiment

To demonstrate this problem, a short experiment was conducted. There has been much research in the recent years about search. A standard metric for measuring different aspects of search performance was necessary. Since document-query relevance is mostly subjective, it is difficult to say that some search techniques are better than others at retrieving the relevant documents. To address this lack of consistency, different researchers gathered small document collections, created a set of queries and asked experts in the fields of the document contents to judge which documents should be retrieved by each query. To compare search techniques between researchers, the different collections were exchanged. Today it is possible to evaluate a search engine using many of the so-called TREC[1]collections on metrics such as precision and recall.

For this experiment, 7 different TREC collections were used (ADI, CACM, CISI, CRAN, MED, NPL and TIME). These range from 82 documents to 11429 documents. Each collection represents a different user or group level access. All possible combinations of the collections were combined and indexed in separate indexes. The provided queries for each collection were then submitted to the indexes which contained the collection in

[1]http://trec.nist.gov/

question. Any documents in the result set which did not belong to this collection were filtered out of the results. The response of each index can then be evaluated to see if the hidden documents had an effect on the visible documents. The ranking algorithm used is based on TFxIDF, see Chapter 2.

The aim is to determine the effect the hidden documents have on search quality, and to verify that simply filtering these documents away is not completely hiding their presence in the index. The trec-eval tool is used to analyze the results of the queries to TREC collections. The tool reports on many different metrics. The metric used in this experiment is mean average precision (MAP). This accounts for document relevance as well as document order. In the ideal result ordering, MAP would be 1. All possible collection combinations were used in this experiment. This resulted in 127 different indexes. 7 of these indexes contained only a single collection. These indexes, once evaluated, produced the baseline MAP value. The data presented in Figure 1.1 is the difference between the baseline MAP value and the MAP values for the same collection in each of the indexes where it is indexed. Each collection exists in 64 indexes (63 in addition to the baseline). So there are 441 data points plotted below. Each point indicates the change in MAP from the baseline case and the percentage of the index which is hidden.

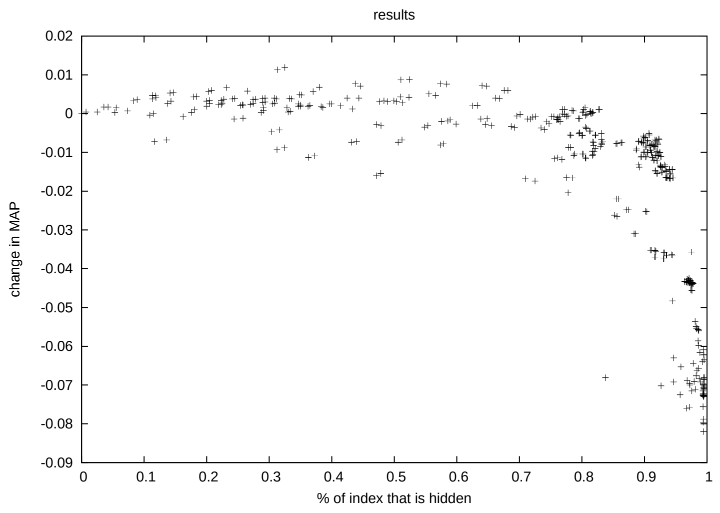


Figure 1.1: Hidden Documents affecting Search Quality.

One thing is certain from this graph: hidden documents do affect the order of results. This is indicated by a change in MAP value calculated by trec-eval. In some cases, the MAP actually increased due to the hidden documents. This was unexpected. However, in most cases, the effect was negative. The only conclusion that can be drawn from this experiment is that the ordering of results can be altered by indexing extra documents and then filtering them out of the result set. [4] presents some techniques which can be used to exploit filtered search results to extract information about the hidden documents. In addition to this conclusion it seems that the trend is that search quality is more negatively affected when the visible documents are composing a smaller proportion of the index. This needs to be explored further with more TREC collections before it can be generalized, but it is nonetheless an interesting observation. One likely explanation is that the effect is dependent on the amount of similar terms, and the query terms, which contained in the collection documents sharing an index.

1.3 Method and Approach

The method used here is following the design paradigm [5]. A system is designed to address a problem, or satisfy a set of requirements which are written to address a problem. It is implemented, and then evaluated based on the design requirements. In this case, a few different implementations are presented. This was possible because of the use of open source software. The use of existing, well established code as a starting point greatly reduces the implementation time, and allows for concentration on the key concerns. The necessary modifications are applied to the existing software. The unmodified version of the software serves as a nice benchmark to compare with to see the cost of the modifications during the evaluation phase.

Chapter 2

Related Work

There have been some concentrated efforts to address this problem. There are certainly many approaches. Some search basics are reviewed here first, before looking deeper to the existing work related to this problem.

2.1 Inverted Indexing

Text-based search centers around the idea of isolating a set of documents from a larger collection based on an input query. The query can contain a number of terms. Each term is used by the search engine to decide which indexed documents should compose the result set. The inverted index is the structure typically used to achieve this functionality [18]. Very quick responses to queries over very large indexes are possible with this structure. All the way from the smallest search appliance to Google[1]and other Internet search engines, there is an inverted index playing a key role in almost any kind of search.

The inverted index is named so because it is mapping terms to information about the documents which contain the term. Thus, unique terms can be parsed out of an input query string and used to select all indexed documents which contain each term with little processing time, and few lookups in the index. This allows for an efficient response to most queries even in very large indexes.

[1]http://www.google.com/

2.2 TFxIDF Ranking

The inverted index, while crucial to text-based search, is not the only component. It allows for selecting documents which are potentially relevant to the query, but it does not provide a means for determining the order which the documents should be presented to the user. The most relevant document should be presented first. There are a few different approaches for ranking. One of the more traditional approaches is known as TFxIDF [17]. TF is term frequency, and IDF is inverse document frequency. The TF component to the function gives some importance to documents which have many occurrences of the term in question, assuming that this is an indicator for relevance. The IDF component gives a higher weight to documents containing a term which is very specific to that document.

2.3 Other Ranking Techniques

In addition to TFxIDF ranking, other techniques can be used in place or in addition to TFxIDF. A technique known as normalization attempts to diminish the affect of TF components of ranking functions. That is to not give preference to documents which happen to be longer, and probably contain more repeating terms simply due to their length, rather than their relevance. Term boosting is another technique which can give a special weight to specific terms. This is usually applied in specific search applications.

In addition to the various ways of determining a document's rank based on the terms, there is also the technique of determining its value based on the other resources which link to it. PageRank [13] is the most famous of these techniques. This is mostly an Internet specific technique.

The ranking function is applied after the relevant documents are selected out of the inverted index. The ranking results in a specific ordering of indexed documents. This ordering can be reverse engineered to determine characteristics about the indexed documents. One solution is to add a boost factor to the ranking function based on user credentials. This could effectively wash the results to avoid data leakage.

2.4 Distributed Search

Distributed search is a concept related to combining multiple indexes under a single search interface. Results from a query would be executed on each index simultaneously and then be aggregated before being presented to the user. The collective index consists of many pieces or fragments. This fragmentation could be configured to reflect the different users and groups of an access control scheme [10]. For each unique set of documents, one index fragment would be required. Then a user's access rights would determine which set of index fragments they would actually search through. This is a legitimate way to hide data leakage through ranking schemes because the hidden documents would not be considered when ordering the visible documents. However, with each index comes a certain amount of overhead. Also query performance may decrease as the number of indexes to query increases. In an environment with many users and groups or those that change somewhat frequently, this solution may not be realistic.

Using a peer 2 peer (p2p) structure for searching through separate indexes based on access control is a plausible solution to this problem. This is promising as an architecture, but is realistic only for a manageable amount of groups. To support a group-object access control, as well as a user-object access control would require a separate peer for every group. This would allow users to search in the subset of peers which represent the groups of documents they are authorized to view, but there is still the case of users being granted access to a specific document which does not hold true for an entire group. Perhaps the cost of setting up a separate peer to handle this case as if it were a new group is too expensive for the gain. Perhaps handling these cases separately from the p2p structure is too expensive at query time. In addition to this shortcoming there is also the issue of search quality. Generating precise results is dependent on communication between the different indexes. To reconstruct a user specific view of the index from a given set of indexes requires that the ranking algorithm is aware of all the index contents. This could be handled outside of the indexes or wherever the ranking takes place, but as the index scales up, this aggregation of data may become too visible at query time [8].

Distributed search is a promising architecture which may be valuable for introducing access control to search. However, it is not explored further here.

2.5 Desktop Search

Desktop search is a very relevant topic when discussing access controlled search because it is operating in an environment which should already have some form of ownership and document visibility defined for its corpus and the users of the system. Different techniques for respecting this access control exist in this realm. The most simple solution is to maintain a separate index for each user of the system. This is a viable solution for systems with few users, but it is wasteful of disk resources when it indexes the public or shared documents in every user index. Often in this setting, disk resources are of the highest concern. For that reason, this concept has been given some attention, but the focus on minimizing the index size is not necessarily the most important concern in other environments, such as enterprise search.

The Wumpus[2]desktop search engine is a good example of this case. It attempts to support user level access control to documents on a single machine by keeping information about the corpus in memory at all times. At query time, the information is included in the ranking calculation only if the user has access to the document it relates to. This allows the mechanism to scale only with the number of documents, it does not matter how many users use the system. A problem might exist when the index grows large enough so the document term information can no longer be held in memory, waiting to be combined in the exact way which represents the user's index. It hides secrets between users. The computation time is acceptable for situations where there are a manageable amount of documents. It will not scale to large amounts of documents, but it will still give acceptable query performance

2.6 Enterprise Search

There have been some efforts made to solve this problem in the enterprise domain as well. Aside from efforts from various commercial products, [1] presents an attempted solution to the problem. The solution is to check the user's access to each document before presenting the results to the user. The evaluation of this solution shows that query response time becomes unacceptable as the unfiltered result set increases. Then it is shown that collection level security is possible for this solution. This reduces the number of lookups to the access control list. However, it imposes a restriction on the organization as to how they should define their access policies. It also may be susceptible to the security issues presented in [4].

2.7 Summary

There are many different options when search is the goal. Some characteristics are almost always the same. Users want their results now. They do not want to wait. Trying to incorporate access control checks at query time may be realistic for some situations, but it most certainly affects query performance.

[2]http://www.wumpus-search.org/

Chapter 3

Requirements

3.1 System Overview

In this thesis, the classic text retrieval search engine is modified to abide by access control lists presenting each user with a personalized index. The search engine for text retrieval is illustrated in Figure 3.1. Users can query the index through an interface. The search engine maintains the index for each user based on the access control lists presented by the security authority. When a user queries the index, the results will be presented to the user as if the user had a completely personalized index consisting of only those documents which they are authorized to view. The result order is the same order they would be returned in from a personal search engine.

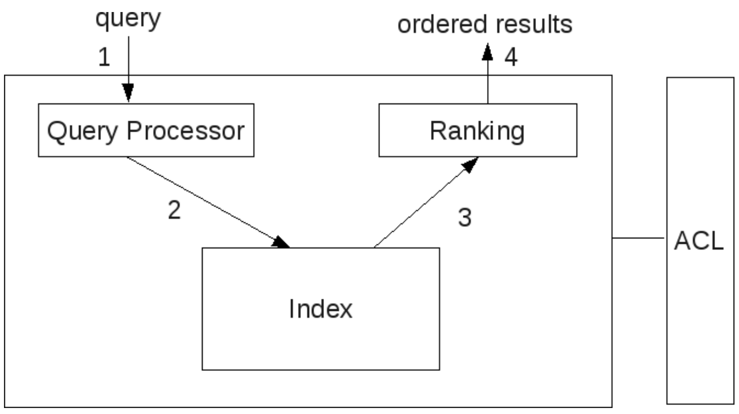


Figure 3.1: System Overview

A user inputs a query string to the search engine (1). This query is processed so it can be used to select documents from the inverted index (2). After this step, a ranking algorithm is applied to the documents selected from the index (3). The ranking function produces a score for each document based on the query string and the information extracted from the index. This score is used to order the results with the intention of putting the most relevant documents first.

3.2 Functional Requirements

3.2.1 Client Application

The client application is a familiar interface for users. A textbox and a search button allow users to issue a query to the search engine. The client application also displays the query response to the user.

> **Query the index**
> The client application shall provide an interface which enables users to query the index.

3.2.2 Search Engine

The search engine is the part of the system that consults the index to select documents which are relevant to a query input. The selected documents are ranked based on a similarity algorithm comparing the query terms to the terms contained in a document.

> **Respond to queries**
> The search engine must respond to queries, considering only the documents the querying user is authorized to view.

3.2.3 Security Authority

The security authority maintains the access control lists for users and groups on the corpus.

> **ACL version control**
> The security authority needs to keep track of changes to both user and group ACLs. Whenever a change is made to an ACL, the corresponding version trackers need to be incremented.

3.3 Non-functional Requirements

No change in Search Quality
Searching in this index should produce the same results in the same order as searching in a personal index.

Acceptable query performance
Query performance should be comparable to search engines without access control. Response time should be under one second, better if it is faster.

Acceptable Indexing performance
The process of indexing documents can slow down, but not to a point which would render the search engine unusable.

Security
Users cannot extract information about a document's existence or content which is not on their ACL.

Chapter 4

Design and Implementation

Many components have to function together to meet the requirements of this system. Each of the major components are described below. Interesting functionality of the components are also included here.

4.1 The Inverted Index

The inverted index used in this solution is borrowed from the open source Lucene[1]project from the Apache Software Foundation[2]. It uses an inverted structure to locate documents which contain specific terms. The key for the index is the term itself, and the value is information about the documents which contain the key. The information stored as the value of this structure is used in the ranking algorithm. Namely, this is the document ids coupled with the corresponding term frequency (TF), or the amount of times this term occurs in this document. Also stored in this structure is the document frequency (used to calculate IDF) for each term. This value counts the number of documents in the corpus which contain this term. From these two pieces of information alone, a ranking scheme can be used to determine how similar a document in the index is to a given query. There are other factors which can yield improved results but these two pieces of information are at the heart of TFxIDF ranking, and they are retrieved from this piece of the index.

Lucene implements the inverted index in an efficient way. There is a term dictionary loaded into memory when the index is opened. Each element in this structure contains data pertaining to a particular indexed term. The document frequency is stored in this structure, as well as pointers to the location of this term in the term position and

[1]http://lucene.apache.org/
[2]http://www.apache.org/

term frequency files, see Figure 4.1. Term frequencies and term position information is necessary for ranking. The pointers must be followed, and disk must be read to find the appropriate data. The term frequencies are written in sorted order based on the internal document id. This was the inspiration for the implementation of the sensitive index.

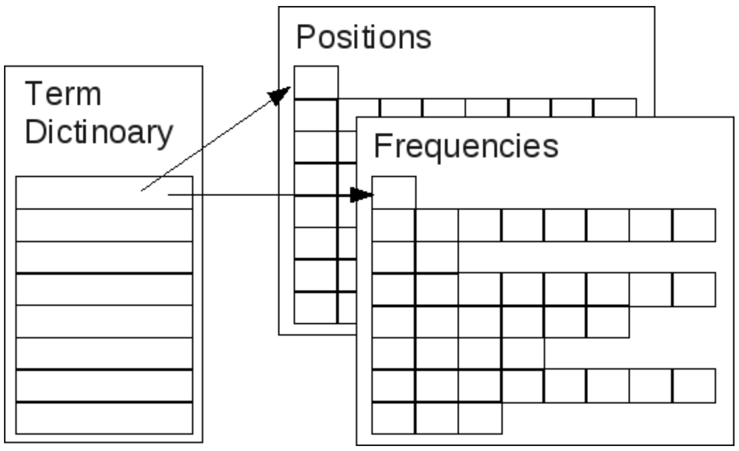


Figure 4.1: Inverted Index

4.2 The Sensitive Index

The sensitive index is the component of the system which ensures that secrets are kept secret. It is a replication of the parts of the inverted index which are potentially revealing secrets. For the ranking algorithm used in this implementation, the IDF value for each term needs to be available for scoring the documents and producing a ranking which does not have the potential to reveal information from the hidden documents. When a set of selected documents are ready to be ranked, the system will read the document specific information (TF values) from the shared index described in the previous section, and read the user specific information (IDF values) from the sensitive index.

Determining how to calculate the values stored in the sensitive index is an implementation detail. The number of instances of this value has an effect on disk size. Determining which information should be included in each value also has implications for indexing, maintaining the index and query time. The goal is to minimize the effect on all of these.

The sensitive index has the potential for increasing the size of the index. To minimize this effect, the stored IDF values do not relate to specific users or groups. Rather they relate to a unique document set from the corpus. If all document viewing rights are mapped down onto the corpus, any documents which end up in the same group can be stored in the same piece of the sensitive index. Each user then has their own sensitive index consisting of a selection of these pieces. This imposes more processing at query time, but that is a trade-off for minimizing this system's index size.

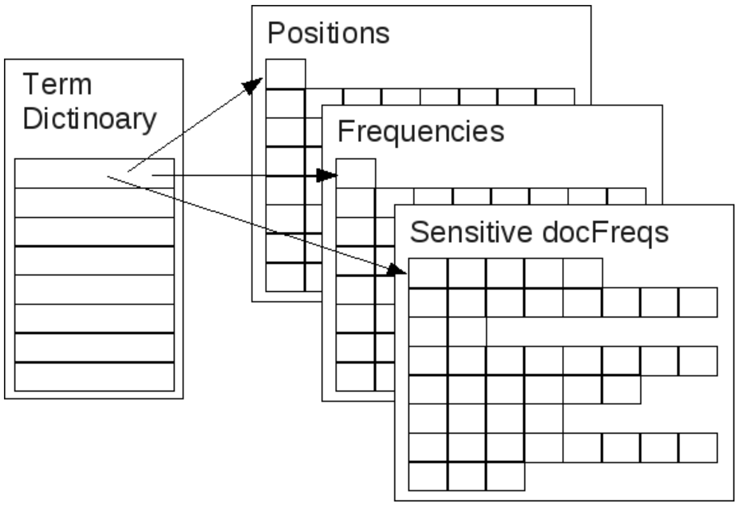


Figure 4.2: Sensitive Index

The implementation of the sensitive index follows the Lucene lookup mechanism for term frequencies and term position information. The in-memory term dictionary was modified to include a pointer to a new file: the sensitive index, as well as information about how many values are stored at the referenced location, see Figure 4.2. This structure is storing sensitized document frequencies for a subset of the corpus. It stores them in the same way as the term frequencies are stored. The component of the solution which talks to the ACL is calculating the necessary document sets for the corpus, and giving them an integer id. This id is written followed by the count of how many documents from this set are containing at least one instance of this term. before the data is flushed to disk, it is sorted to facilitate quick lookups at query time. The document frequency loaded into memory is ignored when a sensitive search is performed, but it was left in to support accurate global search. The superuser's search experience will be very similar to the unmodified version of the index as the sensitive index will never be accessed.

4.2.1 Ranking Search Results

The ranking scheme used in this system is based around TFxIDF. This considers two things when determining how similar a document is to a particular query. It considers how many times the term occurs in a document (term frequency or TF) and how specific a term is to a document (inverse document frequency or IDF). The TF value identifies documents containing terms, and assumes that documents with the most occurrences of a specific term are the most relevant for that term. It does not consider that some documents are longer than others, and maybe the term is very common among all documents indexed. To address these shortcomings of the TF value, IDF is used as well. This is a count of how many indexed documents contain the term in question. This helps the ranking algorithm to determine how specific a term is to a particular document. The average query is two or three terms in length, so there must be some discrimination. With this formula, a score can be given to documents which are suspected to be relevant, and used to order them when presented to the user. There are other scoring techniques in use here, but the IDF value is the only bit of information which is aware of something external to the document in question. Anything which says something pertaining only to one document, such as the TF value, does not need to be sensitized because authorized users are authorized to have this information. Information of this nature can be shared between all users of the system, as the hidden documents will be filtered away. Advanced ranking techniques, such as query expansion based on global data [16] or document boosting based on global inputs are not supported. As a rule, anything considered in the ranking must be either pertaining only to the document in question, or it must be sensitized to the querying user's view of the corpus. In this implementation only IDF is sensitized. This could of course be expanded to include other inputs to the ranking function, allowing for more complex ranking.

4.3 Access Control List

The access control list should be an arbitrary component. It should not matter which system is used to maintain access to documents. In this implementation, A system called Kasai[3]was employed because it provides almost all of the necessary functionality, and it has a Java[4]API which allowed it to be easily integrated with Lucene. Kasai is written in Java and based on a MySQL[5]backend for storing user, group and object policies. Minor modifications where made to allow for easy integration with the other components of the system.

A cached copy of all access control lists are stored close to the index. A version control mechanism is used to detect when a change occurs. Other techniques could have been used, but this was added to Kasai to enable a simpler implementation of the rest of the system. When a user is created, their version value is 0. With each update to their ACL including when a group they belong to gains or loses access to a document, the version number is incremented. This way it is very easy to know if something has changed about a particular user in the ACL. It only requires that the cached copy remembers which version they are storing for each user.

[3]http://kasai.manentiasoftware.com/

[4]http://java.sun.com/

[5]http://www.mysql.com/

The caching mechanism is not only providing fast access to the ACL for the search engine. It is also managing what documents are to be included in which sets. The index uses this information when building and searching in the sensitive index so it needs to be readily available.

4.4 Handling Queries

Queries to the system need to be processed in an acceptable amount of time without revealing the existence of any hidden documents to the users. To ensure the cached copy of the access control list is consistent with the real one, a quick check to Kasai is executed before each query is processed. If the user's access rights have changed, the change would have to be reconciled in the sensitive index before the query is processed. In this case, the query time is affected, but no secrets are revealed. To avoid this effect, the index could be adjusted after the ACL is updated, rather than waiting for the user to query the system.

4.5 Inserting Documents

When a document is inserted to the index, Kasai is contacted to check the status of the cached access control list. Then the cached version is used to populate the sensitive index with the correct term-document information depending on which document sets it needs to be included in at that point in time.

4.6 Two Different Implementations

Two different implementations of this design were built. They use a slightly different approach to interpreting the ACL.

4.6.1 PrivaSearch1

This implementation follows the design explained above. The complete ACL for the corpus is used to calculate unique document groups based on all policies stored in the ACL. The document frequencies are then maintained for each of the resulting pieces. A user is then authorized to view a certain set of these pieces to compose their custom view of the corpus. This new value is substituted into the ranking function in place of the global value. The result order is not based on information about any hidden documents. Therefore, no secrets can be learned from the result set.

4.6.2 PrivaSearch2

This is very similar to privaSearch1 except the document groups are calculated differently. In privaSearch2, each unique user view is assigned a piece of the sensitive index. If users belong to a group and all users in this group have the same authorization, they will all be using the same single piece of the sensitive index. Each unique view of the index is tracked separately. So the same document could be tracked in multiple sensitive index pieces. Most users will then only need to read one value from the sensitive index at query time to get the correct ranking.

4.7 Separate Indexes

The requirements for this design demand that each user be given a personalized view of the corpus based on an ACL. It is only natural then to consider providing a separate index for each user. This was also implemented and used for comparisons with the other implementations. It does not need any special mechanisms other than a multiplexer for updating the correct indexes. When a user queries the index, he also needs to be routed to the correct index. With these two features in place, an ACL aware search solution is possible.

Chapter 5

Evaluation

Evaluation of the system was quite straightforward. Since all of the work on this solution was completed as modifications to existing software, there is an obvious way to evaluate the capability of the solution. In all tests a comparison is made. An unmodified version of Lucene is the baseline for comparisons in every test. Making Lucene aware of access rights does impact the performance of the software. The evaluation is intended to investigate the effects of adding these modifications. The effects are discussed briefly here, but also again in the following chapter.

5.1 Test Collections

The documents used in the evaluation are the same documents used in the experiment described in Chapter 1. There are seven separate collections from TREC which are indexed together in the index to be evaluated. User level ACLs are decided on differently for the different test cases, but common to all cases is the existence of one user per collection who is authorized to view the documents of only their collection. To simulate different situations, alternate ACL configurations are described below for each case.

5.2 Index Characteristics

In this thesis an extra level of complexity was added to the index to handle data access concerns. This additional feature, like the inverted index, will perform differently for different situations. Before discussing an evaluation of the solution, some characteristics about different possible cases need to be discussed. Then the evaluation can be presented in the terms of these cases. This will help to justify some of the design decisions as well as show potential weaknesses of the solution.

5.2.1 Disjoint document distribution

In this case no documents are shared amongst different user groups. This case is illustrated in Figure 5.1. The union of the three groups is equal to the corpus, and the intersection of any combination of the groups is the null set.

If this were the case, and it were a manageable number of groups, it might make sense to keep a separate index for each group. However, there is always the person at the top who might benefit from the ability to search in all indexes transparently. Whether the access controlled solutions apply to this case or not, it is still useful to explore merely to show the performance of an edge case, and to contrast with the other cases.

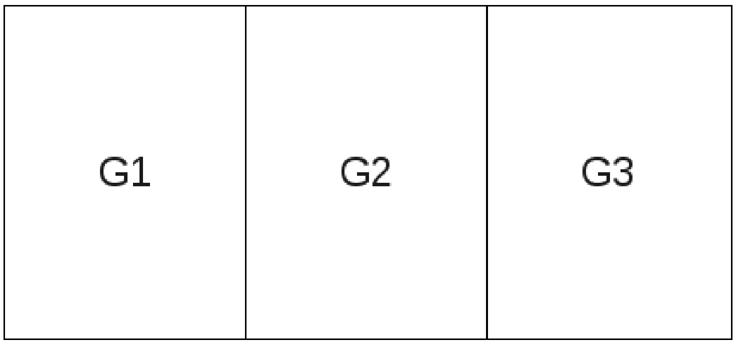


Figure 5.1: Disjoint Corpus Access Rights.

5.2.2 Overlapping Distribution

The main characteristic of the other case is when the documents appear in different user access control lists where those lists are not identical. In Figure 5.2 the same three groups are shown, only this time they have many overlapping regions. In this case, $G1 \cup G2 \cup G3 = C$, the complete corpus. However, all of the possible intersections of the different sets are not the null set. In this case, indexing each group in a separate index is possibly a more appropriate solution. However, the documents which constitute any of the intersections will be indexed more than once. Also labeled in Figure 5.2 are the individual unique sections of the corpus which emerge after projecting all access control policies onto the corpus. This demonstrates how the document sets are calculated for PrivaSearch1.

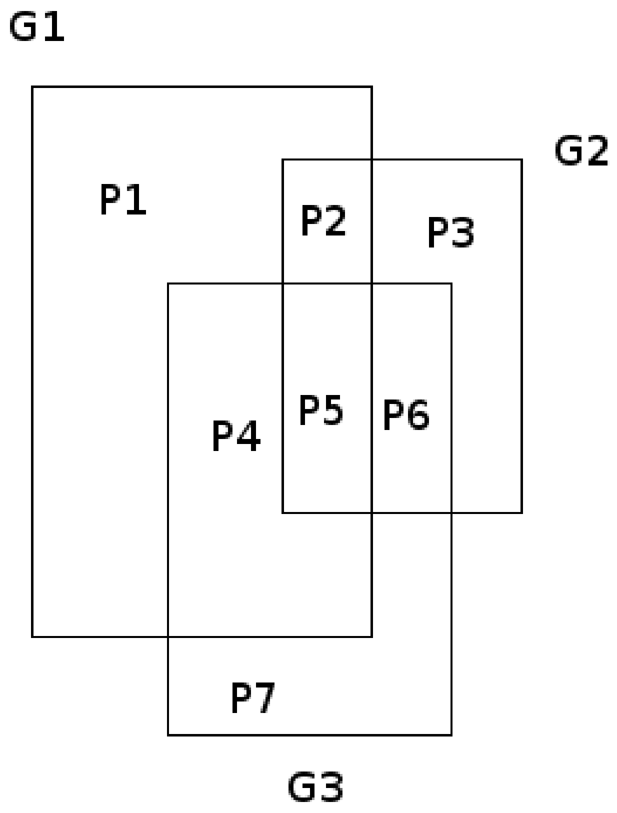


Figure 5.2: Overlapping Corpus Access Rights.

For the PrivaSearch1 implementation, the index must store corpus-wide information about each piece (P1 - P7) rather than each group (G1 - G3). When a particular view of the corpus is requested (G1 - G3) the appropriate pieces are combined to present a customized view of the corpus. For instance, for a user with access rights to G3, the query would be processed only considering the sum of the data contain in P4, P5, P6 and P7. In this manner, all necessary views on the index can be handled without indexing the same documents more than once. If the number of pieces grows, this may become problematic both with index size and with query processing time. This is highly determined by the number of users and groups and the type of access rights they have to the corpus. Exploring this characteristic of the access controlled corpus is the goal of the next cases.

5.2.2.1 Overlapping with few unique pieces

This is the case where a small percentage of the users may require access to a set of documents existing inside a separate existing document group. The result is very similar to Case 1, however the intersection of user views is no longer the null set.

5.2.2.2 Overlapping with many unique pieces

There is a worst case when considering the amount of partitioning the ACL projections create over the corpus. Case 4 represents the worst case and those configurations which are close to the worst case. The worst case can be calculated by looking at the number of unique user views on the corpus, n. A user view is considered unique if it contains at least one document which is unique to this view. Any number of users can share a unique user view. If the views are disjoint then the index can consist of n pieces. When overlapping occurs between views, extra pieces become necessary. The greatest possible number of unique views to the index is $P(n)$ where

$$P(n) = 1 + \sum_{i=2}^{n} (i-1)2$$

This is the worst case for PrivaSearch1, but it seems like a highly unlikely situation. It should be noted that the worst case for PrivaSearch2 is n, the number of unique user views.

5.2.3 Hierarchical organization

This is the final index characteristic explored in this chapter. Typically organizations have some type of hierarchy in place. This hierarchy may determine what resources a given member has access to. To consider the index organized in this structure, it may help to view the hierarchy as a tree, where each node represents a unique user view. There is at least one user who only has access to the documents represented by a node. The rights of all child nodes are inherited up the tree. There can be hierarchical organization with little to no overlap, and there can be hierarchical organization with much overlap. When there is no overlap, the case will resemble case 1, the disjoint distribution. That is the low level users will be searching using only a single piece of the sensitive index, but the higher level users will search using multiple pieces. They will be combined to compose the larger view of the corpus. With PrivaSeach2, each node of the tree would have it's own piece.

5.3 No Change in Search Quality

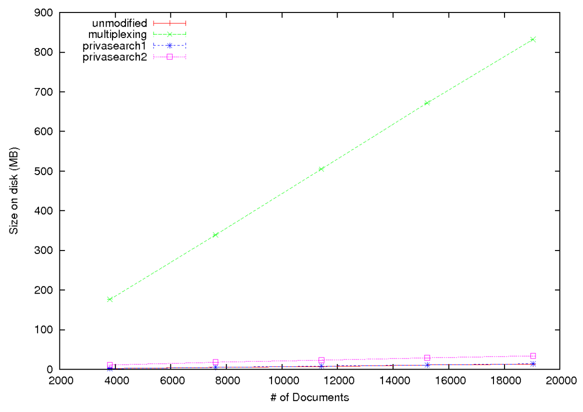
To verify this requirement is satisfied, a TREC evaluation is made on two indexes. The first index is the unmodified Lucene index containing all documents of a single collection. The second index is the access controlled version containing the same TREC collection along with additional TREC collections. The other collections are hidden to the evaluation by the access control mechanism. Satisfying this test means that the TREC output will be identical for both indexes. Further inspection of the individual results for each query was made to be certain there was no data leakage. The search quality and results are identical to the baseline in both implementations.

5.4 Index Size

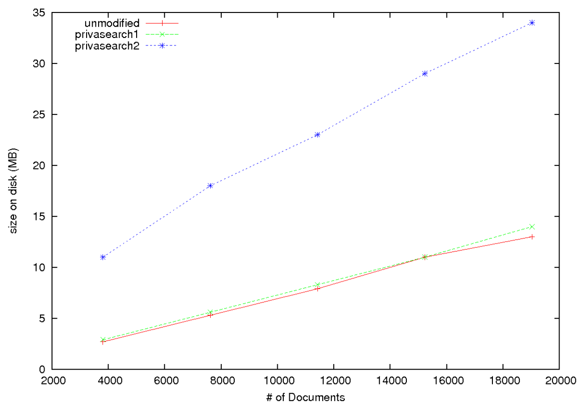
The index must scale with respect to disk size. Since PrivaSearch adds an extra dimension to the index, the size of the index will increase. The following experiments show how the index size is affected in the different circumstances explained earlier.

5.4.1 Case 1: Disjoint distribution with worst case hierarchy

Figure 5.3(a) shows how each solution performs in the case where there is no overlap between groups, but there is at least one user for every possible combination of the groups. PrivaSearch2 is not performing well in this case because it tracks every unique user separately. The change in disk requirements between both PrivaSearch implementations is more visible in Figure 5.3(b), a zooming of Figure 5.3(a). So there are g! document sets, where there is actually only g unique sets of documents. PrivaSearch1 is only tracking information about the unique sets. Real world scenarios would have some hierarchy in the organization, but this is the extreme case. Nevertheless, it helps to show a benefit of PrivaSearch1, and a weakness of PrivaSearch2. The multiplexing solution is also not well suited for this case. Like PrivaSearch2, it treats every unique user view of the corpus independently from the others, only it is creating a complete separate index for each view. This implementation is very costly in terms of disk size and not realistic for scaling in a case like this, or even one simply meant to serve many users. Privasearch1 seems to follow the trend of the unmodified index, with the added overhead of storing extra document frequencies for each of the unique index pieces. While they will all scale to some extent regarding disk size, PrivaSearch1 is the best approach for this type of environment. It is able to grow to hold more documents than the other approaches.



(a) Case 1



(b) Case 1 (Zoomed)

Figure 5.3: Index Size: Case 1

5.4.2 Case 2: Disjoint distribution with no hierarchy

In 5.4 both PrivaSearch solutions are nearly equal regarding disk size. This is due to the fact that there are the same amount of unique users as there are document groupings. The two implementations are storing roughly the same amount of data to provide the secured index to the users. It is also noteworthy that the effect of the multiplexing solution is greatly reduced here. In situations where data security is of the highest concern, and the user rights are organized in this way, in moderate amounts, this should at least be considered as an option.

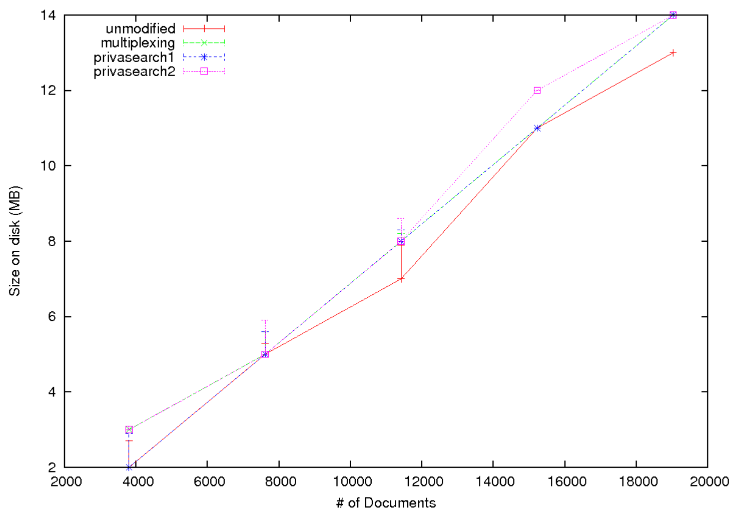
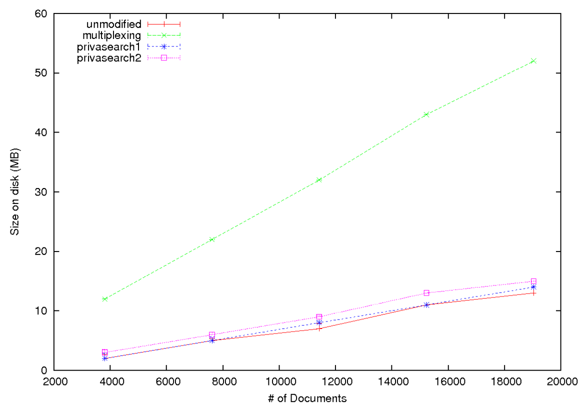


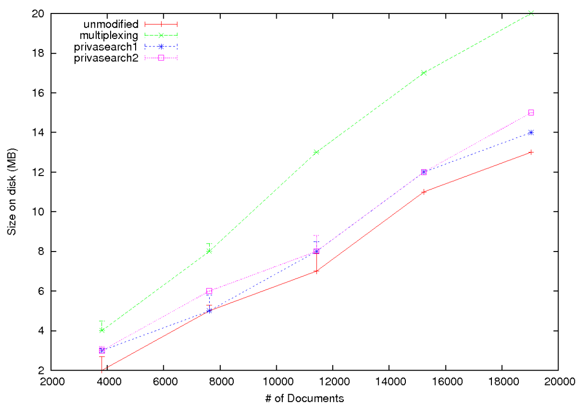
Figure 5.4: Index Size for Case 2

5.4.3 Case 3: Overlapping distribution with few unique pieces

The following graph shows how the different solutions perform in this case. The number of unique document groups and unique user views of the corpus are equal in this configuration. Whenever this is the case, both PrivaSearch solutions will have roughly the same disk space requirements.



(a) Case 3



(b) Case 4

Figure 5.5: Index Size

5.4.4 Case 4: Overlapping distribution with many unique pieces

Here there are more unique index pieces than user views of the corpus. When this is the configuration of the access control lists, PrivaSearch2 is appropriate if disk space is the main concern.

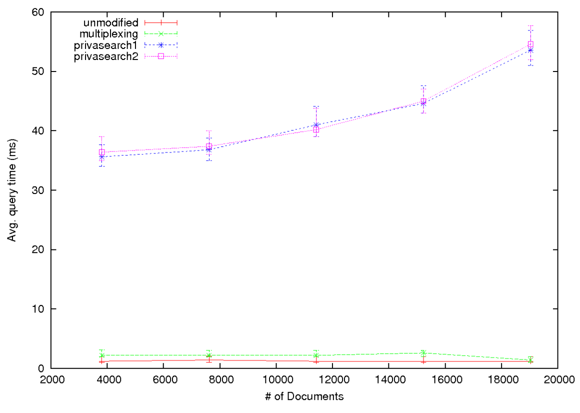
There is going to be at least a little bit of hierarchy in an organization. Even if there is a fair amount of overlap, the effects of the hierarchical organization will offset the benefits PrivaSearch2 has with overlapping access rights. Both PrivaSearch solutions are roughly equivalent regarding their disk requirement in a realistic case. The worst cases were explored in this section to show exactly which characteristics would cause the indexes to require more disk.

5.5 Acceptable query performance

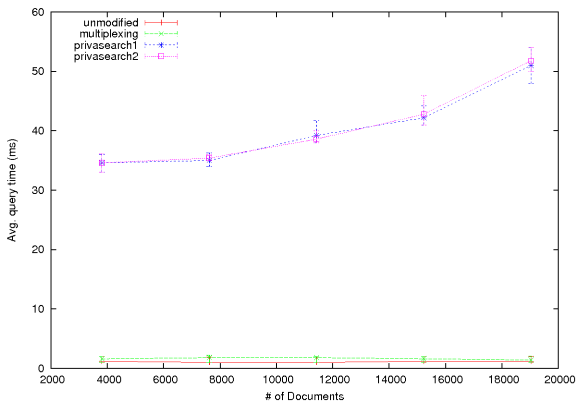
In order for a search engine to be useful to its users, it must respond to queries in a reasonable amount of time. This is usually on the order of hundreds of milliseconds with modern search technology and hardware. There are many approaches to add access control to the search engine. The approaches which may have a dramatic effect on query performance are not acceptable. This graph shows a baseline test issued to indexes with the same amount of documents. One of the indexes is unmodified and the others are supporting access control. The returned results for the queries are not identical between the unmodified version and the others. Between the ACL supporting indexes, the query results returned in the test were identical. This compares query time between the different implementations operating on the same cases as the previous section's tests.

In all of the cases, both of the PrivaSearch solutions experience an increase in query processing time as the number of documents increase. This is a bit discouraging because both the unmodified and multiplexing solution are constant for the index sizes tested.

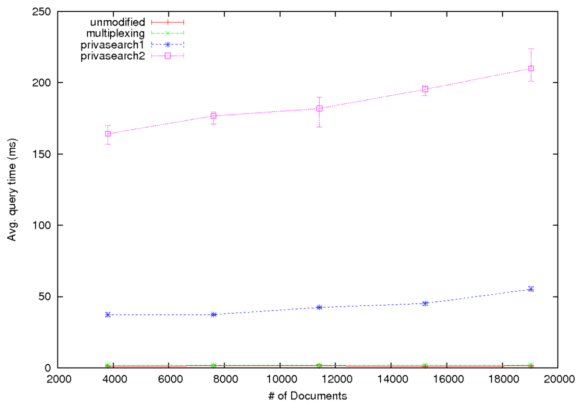
In Figure 5.6(c) The average query time for PrivaSearch2 was much higher than for PrivaSearch1. This can be attributed to the characteristics of the case.



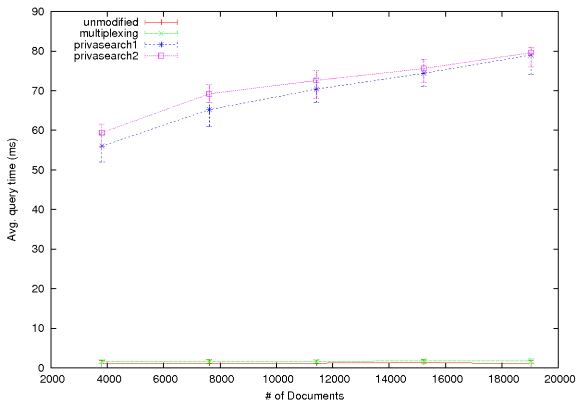
(a) Case 1



(b) Case 2



(c) Case 3



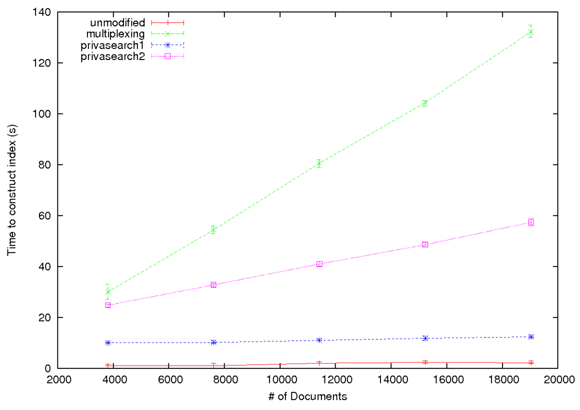
(d) Case 4

Figure 5.6: Query Performance

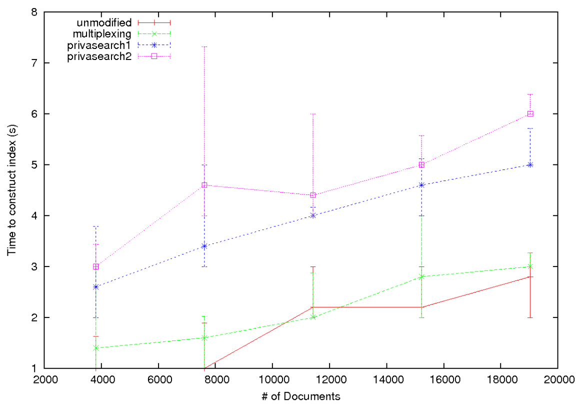
At some point, the query time will become unacceptable, and the PrivaSearch solutions will have met their limit for scaling. This is determined both by the number of documents, the number of unique users and the type of access they have to the corpus, as shown here. Once this point is met, there are many techniques for distributing an index [3, 10]. Especially in this case, the PrivaSearch query overhead could be reduced dramatically by splitting the index into pieces based on the document groups stored in the sensitive index.

5.6 Indexing performance

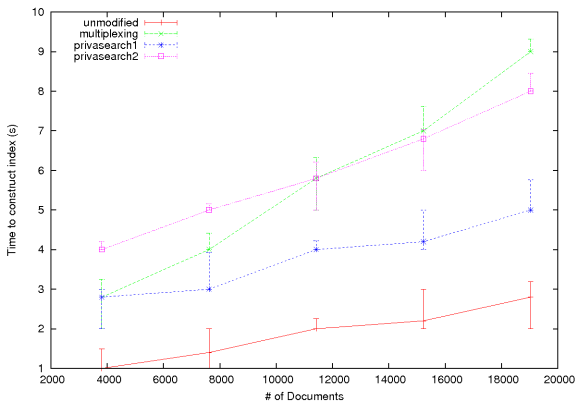
Indexing performance is less important than query performance, but it is still a concern. The tests presented in Figure 5.7 shows that indexing performance degrades slightly, but at an acceptable amount.



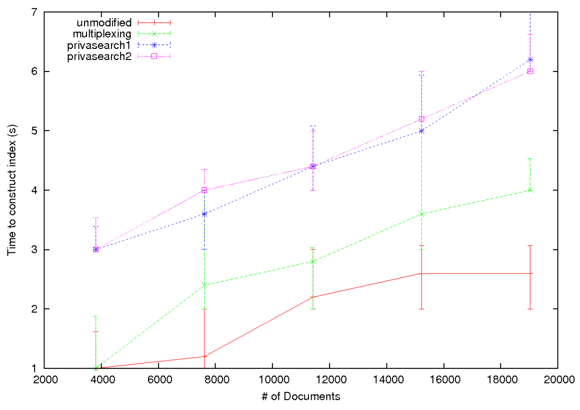
(a) Case 1



(b) Case 2



(c) Case 3



(d) Case 4

Figure 5.7: Indexing Performance

5.7 Evaluation Environment

The experiments in this chapter were run on an HP nc8430 laptop with the following specifications:

CPU: Intel Centrino Duo T7200, 2.00 GHz
RAM: 2 GB 533 MHz DDR2 SDRAM
HDD: 80GB 5400RPM 8MB SATA/150 2.5” 9.5mm HDD
OS: Ubuntu 8.10

experiments were repeated 5 times each and the range of values are plotted on the graphs along with the average for each measured index.

5.8 Summary

These experiments show that the solutions explored are mostly feasible for the cases tested. Multiplexing is only realistic for a small number of users, or in cases where the necessary hardware is abundant, but it is clearly very wasteful. More experiments with larger indexes and different user/group arrangements need to be explored before it can be determined to be a scalable solution.

Chapter 6

Discussion

6.1 Functional Requirements

Both PrivaSearch implementations satisfy the functional requirements. That is that they can operate as a normal search engine with the added awareness of which user is actually searching.

6.2 Non-Functional Requirements

6.2.1 No Change in Search Quality

As demonstrated in Chapter 1, search quality can be affected by the existence of extra documents in the index. Information about these hidden documents make it into the ranking algorithm, and this can have negative, neutral or sometimes positive effect on search quality. The existence of this effect is what can be exploited by a clever user. Removing this effect is a defense against this type of attack.

6.2.2 Acceptable Query Performance

Of the cases evaluated, the query time remained very close to the unmodified index. This was due to the design strategy of moving the computationally intense operation, calculating the correct document score for a given user, mostly to the indexing phase. This minimized the effect on the query time critical path, which is arguably the most important

6.2.3 Acceptable Indexing performance

As discussed in the previous section, the solution is optimized for query time, not indexing. While there may be some environments where indexing speed is the main concern, the overhead introduced to indexing by PrivaSearch is largely transparent to the user. The evaluation shows that even in the worst cases, the overhead is reasonable.

6.2.4 Security

The danger of leaking information through the ranking function was sealed with PrivaSearch, but there may be other vulnerabilities which were not addressed here. To say that it is completely secure of data leakage may be to overlook something.

6.3 PrivaSearch1 vs. PrivaSearch2

PrivaSearch2 was designed with the intention of storing the least possible amount of information while still being able to respond to any user's query in a secure way. PrivaSearch2 was implemented as an afterthought. It is a relatively simple implementation after PrivaSearch1 was complete. It seemed that PrivaSearch1 would give much better indexing performance, at the cost of a slight drop in query performance compared to PrivaSearch2's approach. The evaluation doesn't show that there is a great benefit of PrivaSearch1, except in the worst case for PrivaSearch2.

PrivaSearch2 does track the same documents in different places in the index. This means that updates would be more complex than with PrivaSearch1, which only has one document set containing a given document. PrivaSearch1 is perhaps superior in that respect. However, this is not implemented fully, and was therefore not evaluated. Perhaps the effect is something similar to the indexing performance. In that case, the more simplistic PrivaSearch2 is superior. But it also depends on the structure of the organization as the different cases in the evaluation have revealed.

6.4 Additional Uses

Securing information leakage is a key benefit to PrivaSearch. However, there may be other side effects that such an index provides. Consider how valuable enterprise search is to a given organization. Its effects have been determined to greatly boost productivity, among other things. With the emergence of virtual organizations and the temporary interoganizational relationships that come with this trend [11], there comes a time for cooperation between organizations. With such an index in place all of the internal documents relevant to the cooperation could be shared through the search interface simply by creating a temporary access policy for the partner organization. This minimizes the overhead for setting up special arrangements for the partnership. The idea of the search engine as a valuable shared resource is presented from a distributed search perspective in [10].

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The user experience is arguably the most important determinant of a search engine's success. In this thesis, a solution was presented to defend a basic search engine against the extraction of hidden information. With most implementations, this means a significant drop in query performance. The design evaluated here was able to maintain reasonable performance measures with a small index. The structures behind PrivaSearch are designed to scale. However, every extra lookup and computation at query time means a drop in the user experience. It is a promising design, nonetheless.

Filtering search results is not a sufficient solution for providing access control to the search index. In order to respect document access policies, the results from a search on the complete corpus must be adjusted so they match exactly the results of the same search on an index containing only the visible documents. Washing the results of a shared index after the search completes is computationally expensive and perhaps difficult to manage. To maximize query performance while respecting document access policies, a separate sensitive index is maintained alongside the normal, global index. The contents of this index are used to accurately score and rank the documents selected from the global index. This introduces extra disk reads which slightly decrease query performance compared to the same index without these modifications. It also increases the disk requirements of the index. These are acceptable trade-offs for an access controlled search engine which does not reveal secrets contained in hidden documents. The solution presented in this thesis will scale to handle more indexed documents. How well it will scale is highly dependent on the configuration of the user access rights,

7.2 Future Work

The design chosen for this system were made so it can operate in an enterprise environment, as opposed to desktop search, or Internet search. Some more work could be done by looking into common enterprise access control mechanisms to see how they could fit into the scheme presented here. This would make it a more realistic option for deployment in such a setting.

Exploring the possibilities of applying this solution to the desktop environment might also be worthwhile, as the evaluation showed the disk overhead to be quite reasonable except for the worst cases. Perhaps the worst cases are less likely to occur in a desktop environment.

The first experiment presented in this thesis exposed some potential trends. Exploring this thread further may lead to some interesting results.

There are many different approaches to ranking search results. PageRank [13] or SALSA [9] consider links between documents as an indicator of a particular document's value, for instance. It would be exciting to investigate how these schemes might fit in with the PrivaSearch approach.

Document updates are an important capability of search engines, particularly in the enterprise environment. This needs to be implemented and evaluated, as it is suspected that the PrivaSearch sensitive index could allow for efficient updates to not only documents, but user and group access rights as well. This would be great if it could happen without re-indexing the document(s) in question, or rebuilding the index.

References

[1] Bailey, P., Hawking, D., and Matson, B. Secure search in enterprise webs: tradeoffs in efficient implementation for document level security. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management* (New York, NY, USA, 2006), ACM, pp. 493–502.

[2] Belkin, N. J. Helping people find what they don't know. *Commun. ACM 43*, 8 (2000), 58–61.

[3] Butler, M. H., and Rutherford, J. Distributed lucene : A distributed free text index for hadoop. Tech. rep., HP Labs, 2008.

[4] Büttcher, S., and Clarke, C. L. A. A security model for full-text file system search in multi-user environments. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2005), USENIX Association, pp. 13–13.

[5] Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. Computing as a discipline. *Commun. ACM 32*, 1 (1989), 9–23.

[6] Denning, D. E., and Schlörer, J. A fast procedure for finding a tracker in a statistical database. *ACM Trans. Database Syst. 5*, 1 (1980), 88–102.

[7] Hawking, D. Challenges in enterprise search. In *ADC '04: Proceedings of the 15th Australasian database conference* (Darlinghurst, Australia, Australia, 2004), Australian Computer Society, Inc., pp. 15–24.

[8] Klampanos, I. A., and Jose, J. M. An architecture for peer-to-peer information retrieval. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval* (New York, NY, USA, 2003), ACM, pp. 401–402.

[9] Lempel, R., and Moran, S. Salsa: the stochastic approach for link-structure analysis. *ACM Trans. Inf. Syst. 19*, 2 (2001), 131–160.

[10] Meij, E., and Rijke, M. D. Deploying lucene on the grid, 2006.

[11] MOWSHOWITZ, A. Virtual organization. *Commun. ACM 40*, 9 (1997), 30–37.

[12] MUKHERJEE, R., AND MAO, J. Enterprise search: Tough stuff. *Queue 2*, 2 (2004), 36–46.

[13] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[14] ROSE, D. E., AND LEVINSON, D. Understanding user goals in web search. In *WWW '04: Proceedings of the 13th international conference on World Wide Web* (New York, NY, USA, 2004), ACM, pp. 13–19.

[15] SILVERSTEIN, C., MARAIS, H., HENZINGER, M., AND MORICZ, M. Analysis of a very large web search engine query log. *SIGIR Forum 33*, 1 (1999), 6–12.

[16] XU, J., AND CROFT, W. B. Query expansion using local and global document analysis. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval* (New York, NY, USA, 1996), ACM, pp. 4–11.

[17] YUWONO, B., AND LEE, D. L. Search and ranking algorithms for locating resources on the world wide web. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 164–171.

[18] ZOBEL, J., AND MOFFAT, A. Inverted files for text search engines. *ACM Comput. Surv. 38*, 2 (2006), 6.

List of Figures

Appendix A

CD-ROM

All source code and experiment scripts can be found on the included CD-ROM. There is also a readme file with more detailed information about the contents.