ΑΠΟΔΟΤΙΚΗ ΠΡΟΣΤΑΣΙΑ ΙΔΙΩΤΙΚΟΤΗΤΑΣ ΣΕ ΜΗΧΑΝΕΣ ΑΝΑΖΗΤΗΣΗΣ ΚΕΙΜΕΝΟΥ

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Μηχανικών Η/Υ και Πληροφορικής
Εξεταστική Επιτροπή

από την

Μιχέλη Ειρήνη

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Οκτώβριος 2013

Dedication

To the person who taught me well and believed in me the most...,

my grandmother Efthimia

ACKNOWLEDGEMENTS

I would never have been able to finish this thesis on my own. That is why I feel the deep need to express my gratitude to all those who helped me, even if I haven't noticed it.

First, I am sincerely grateful to Professor Stergios Anastasiadis for his constant guidance, patience, and encouragement. Under his supervision, he soon made me realize his deep knowledge in the field of computer science. With his insight, he managed to turn every hesitation of mine into inspiration and creativity, which was a core point to go the full distance.

Completing this thesis would have been more difficult if I didn't have the support and friendship provided by the other members of the System Research Group. I am mostly grateful to George Kappes, who was there whenever I was in need. He also helped me to continue my studies when I stayed away from Ioannina. His offer was valuable. Special thanks should be given to George Margaritis for introducing me to the topic as well for the support on the way. Also, I am thankful to Andromachi Hatzieleftheriou for answering to my questions and advising me every time I was in need.

No words to express my gratitude to my family for their support and immense love, which was the one that gave me the courage to keep on. My sister Anna was the person who listened to my thoughts and aspirations, even though she lives miles away. Furthermore, I am grateful to my beloved uncle Ioannis and his wife Maria for their support since the early stages of my high school years.

I am also thankful to my childhood friends Ilianna and George Chiotis, as well as my cousins Anastasia and Kyriakos Savvidis. The summer night conversations, the books borrowed from them and the radio broadcast in which George participated in, were some of the activities that helped me to deal with stress.

At last but not least, I would like to thank Professors Aristidis Likas and Panayiotis Tsaparas for their precious remarks and review of this thesis.

Table of Contents

List of Figures

List of Tables

Abstract

Eirini C. Micheli, MSc, Computer Science and Engineering Department, University of Ioannina, Greece. October, 2013. Efficient Privacy Protection in Full-Text Search Engines.

Thesis Supervisor: Stergios V. Anastasiadis

Personal and enterprise environments store and manage an increasing volume of data as their storage capacity improves exponentially. This trend drives the demand for full-text search engines that help to automatically locate relevant information. Privacy refers to the ability or right of individuals to control what information is collected about them, who uses it, and for what purpose. In multi-user environments, access control has to be enforced to protect privacy during search. Accordingly, search engines often build one index per user, or they create a system-wide index and filter the results by access rights. In order to protect privacy at improved efficiency, the search engine creates one index for each set of documents accessed by the same set of users. However, this approach lacks tunable parameters to meet different performance needs.

In this thesis, we investigate how to provide a tunable solution for privacy protection in search engines over multi-user environments. Thus, we introduce a novel strategy to organize user documents into indices. The key insight is to cluster documents based on the similarity of their access permissions using a Similarity parameter, and then map the documents and users into indices using a Threshold parameter. The experimental study of our solution shows that a trade-off arises between the query performance and the maintenance cost across different similarity and threshold values. Over a prototype implementation, we experimentally identify those parameter values that achieve a substantial reduction of query response time, while slightly raising the maintenance cost.

Εκτεταμενη Περιληψη στα Ελληνικα

Ειρήνη Μιχέλη του Χρήστου και της Ευσταθίας. MSc, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Οκτώβριος 2013. Αποδοτική Προστασία Ιδιωτικότητας σε Μηχανές Αναζήτησης Κειμένου.

Επιβλέπων: Στέργιος Β. Αναστασιάδης

Συστήματα αναζήτησης τα οποία διατηρούν αρχεία πολλαπλών χρηστών έρχονται συχνά αντιμέτωπα με το πρόβλημα της προστασίας της ιδιωτικότητας του περιεχομένου των αρχείων. Οι προσωπικοί και εταιρικοί υπολογιστές καθώς και τα περιβάλλοντα κοινωνικής δικτύωσης, υποστηρίζουν τη συνύπαρξη πολλαπλών χρηστών, ενώ ταυτόχρονα τους παρέχουν μηχανισμούς ελέγχου πρόσβασης. Οι μηχανισμοί πρόσβασης χρησιμοποιούνται προκειμένου να ορίσει κάποιος τις επιτρεπόμενες ενέργειες των υπόλοιπων χρηστών πάνω στα αρχεία που του ανήκουν. Επομένως, το σύστημα αναζήτησης, εφόσον διαχειρίζεται τα αρχεία κειμένου όλων των χρηστών, πρέπει με κάποιο τρόπο να διατηρεί τις πληροφορίες ελέγχου πρόσβασης προστατεύοντας την ιδιωτικότητα. Έτσι, μπορεί να εξασφαλίσει ότι η απάντηση στο ερώτημα κάθε χρήστη περιλαμβάνει μόνο κείμενα για τα οποία έχει το δικαίωμα ανάγνωσης.

Πολλές ερευνητικές εργασίες έχουν γίνει πάνω στην προστασία της ιδιωτικότητας σε μηχανές αναζήτησης κειμένου όταν διατηρούν αρχεία πολλαπλών χρηστών. Κάποιες από αυτές προτείνουν τη δημιουργία ενός ξεχωριστού ευρετηρίου για κάθε χρήστη, εισάγοντας κάθε αρχείο στα ευρετήρια όλων των χρηστών που έχουν τη δυνατότητα να δουν το περιεχόμενό του. Η λύση αυτή προστατεύει την ιδιωτικότητα και παρέχει γρήγορη απάντηση των ερωτημάτων, αφού η αναζήτηση κάθε χρήστη περιορίζεται μόνο στο ιδιωτικό του ευρετήριο. Το κόστος όμως μιας τέτοιας προσέγγισης είναι απαγορευτικό καθώς κάθε αρχείο εισάγεται σε πολλαπλά ευρετήρια, με αποτέλεσμα να δαπανάται μεγάλος αποθηκευτικός χώρος. Επιπλέον, η ενημέρωση κάθε αρχείου πυροδοτεί μια σειρά ενημερώσεων σε πολλαπλά ευρετήρια.

Μια άλλη προσέγγιση που ακολουθείται είναι η χρήση ενός ενιαίου ευρετηρίου, το οποίο περιλαμβάνει όλα τα αρχεία των διαφορετικών χρηστών. Έτσι, αποφεύγεται η εισαγωγή ενός αρχείου σε πολλαπλά ευρετήρια, με την προϋπόθεση όμως ότι κατά την εκτέλεση ενός ερωτήματος τα αποτελέσματα φιλτράρονται, ώστε τελικά ο χρήστης να λαμβάνει μόνο κείμενα των οποίων το περιεχόμενο είναι εξουσιοδοτημένος να δει. Παρ' όλο που η λύση αυτή φαίνεται ότι εξασφαλίζει την ιδιωτικότητα, το τρωτό της σημείο βρίσκεται στο γεγονός ότι, οποιοσδήποτε χρήστης μπορεί έμμεσα να εξάγει συμπεράσματα για το πλήθος των κειμένων που περιέχουν έναν συγκεκριμένο όρο, αλλά και για το περιεχόμενο των κειμένων για τα οποία δεν έχει τα κατάλληλα δικαιώματα πρόσβασης.

Οι παραπάνω προσεγγίσεις αποτελούν δύο ακραίες λύσεις, κάθε μια από τις οποίες θυσιάζει είτε την αποδοτικότητα, είτε την προστασία της ιδιωτικότητας του συστήματος αναζήτησης. Υπάρχουν νέες λύσεις που ακολουθούν διαφορετική προσέγγιση προστατεύοντας την ιδιωτικότητα των κειμένων και βελτιώνοντας την αποδοτικότητα του συστήματος αναζήτησης. Όμως, η αποδοτικότητά τους είναι μη παραμετροποιήσιμη.

Στην παρούσα εργασία, εξετάζουμε τον τρόπο με τον οποίο μπορούμε να επιτύχουμε αποδοτική προστασία της ιδιωτικότητας σε συστήματα αναζήτησης με πολλαπλούς χρήστες. Βασικός μας στόχος είναι να δώσουμε μια λύση, η οποία επιτρέπει τη ρύθμιση του χρόνου εκτέλεσης των ερωτημάτων και του κόστος διατήρησης των ευρετηρίων με τη χρήση ορισμένων παραμέτρων, ενώ ταυτόχρονα εξασφαλίζει την ιδιωτικότητα. Προς την εκπλήρωση του στόχου μας εισάγουμε μια νέα στρατηγική οργάνωσης των αρχείων των χρηστών σε ευρετήρια που αξιοποιεί το βαθμό ομοιότητας, ο οποίος εισάγεται ως παράμετρος, μεταξύ των λιστών ελέγχου πρόσβασης των αρχείων και τα ομαδοποιεί. Έπειτα, διαχωρίζουμε τα αρχεία κάθε ομάδας και καθορίζουμε το πλήθος και το περιεχόμενο των ευρετηρίων που δημιουργούμε βάση μιας επιπλέον παραμέτρου. Οι πειραματικές μετρήσεις φανερώνουν ότι επιτυγχάνεται αύξηση ή μείωση του χρόνου εκτέλεσης των ερωτημάτων και του κόστους διατήρησης των ευρετηρίων, ανάλογα με τις τιμές των παραμέτρων που εισάγουμε. Τελικά, με συγκεκριμένη παραμετροποίηση επιτυγχάνουμε την μείωση του χρόνου εκτέλεσης των ερωτημάτων με μικρή αύξηση του κόστους διατήρησης των ευρετηρίων, ενώ ταυτόχρονα παρέχουμε παρέχουμε προστασία της ιδιωτικότητας.

Chapter 1

Introduction

1.1 Thesis Scope

Over the last years, the improvements in storage capacity enable users to store and manage a large amount of data. File systems organize files into a hierarchical namespace and a file access requires explicit knowledge of the file's name and location. Even though hierarchical namespace is an appropriate way for users to organize their files, its limitations become obvious when the number of files within a system significantly grows. The increasing amount of data in desktop and enterprise environments complicates the management of files as it is not easy to remember where each file is stored.

In order to effectively find and manage text files (documents), a *full-text search engine* (or simply *search engine*) builds indices on a collection of documents and enables users to search for information within the documents' content. Information search is expressed by submitting search queries with terms to the search engine. The search engine evaluates a query and returns a list of the documents whose content is relevant to the query terms. Each document in this list is associated with a relevance score which indicates how relevant the document is to the query. The highest the relevance score the more relevant the document is. Then, the search engine ranks the document list based on the relevance scores of the documents and finally returns it to the user who submitted the query.

Desktop and enterprise environments support multiple users. In multi-user environments, users usually define through an access control mechanism who can access their files and how. A type of access control is the Access Control List (ACL). An ACL contains users and/or groups along with their respective rights on a particular file. Each file or folder has its own ACL. Hence, a search engine that operates in such an environment needs to index the documents respecting their access rights, protecting privacy, and ensuring that each user only obtains information concerning documents that he is allowed to read.

In order to protect privacy, many search engines create one index per user. The index of each user contains all documents that he is allowed to read. This approach offers high query performance because each user has his private index and the search engine accesses only that index to answer his queries. While privacy is provided, it is implied that each document resides in the indices of all users that can read it. Hence, this approach is too costly due to the large disk space consumption and the high index maintenance cost.

Instead, search engines can use a system-wide index which contains the documents of all users. The search engine can then filter each search result to only include documents readable by the user who issued the query. This approach eliminates the multiple document insertions in indices and limits the disk space consumption, but it does not protect privacy under some circumstances [6]. In particular, it is possible for an arbitrary user to infer the number of documents containing a particular term or the entire content of a document that he is not allowed to read. These privacy threats can be avoided but at the expense of query performance [6].

The above approaches sacrifice either the efficiency or the privacy of the search engine in order to provide full-text search in multi-user environments. A different approach, that provides privacy and improves efficiency, creates one index for each set of documents accessed by the same set of users [37]. Documents accessed by a single user are indexed by a private index, while documents shared between the same set of users are indexed by shared indices. A user's query is answered by combining the results of his private index with the shared indices that he can access. Even though this approach provides privacy and is more efficient than the previous one, it does not provide any tunable parameters to trade query performance for index maintenance cost and meet different performance needs.

Motivated by the privacy threats that arise in full-text search and the lack of a tunable solution that achieves a trade-off between query performance and index maintenance cost, we provide a solution that addresses both of them. Our approach introduces a novel strategy to organize users' documents into indices. We group documents into clusters, each containing documents with similar ACLs. The similarity between the ACLs of documents within a cluster is determined by a Similarity parameter. Then, we map documents and users to indices based on the common users of the ACLs within a cluster. In addition, we use a Threshold parameter which determines in which indices the documents and the users are mapped. We perform several measurements for different Similarity and Threshold values and show that our solution introduces a trade-off between query performance and index maintenance cost. By choosing the appropriate Similarity and Threshold values, we substantially reduce the query response time in comparison to an approach that creates one index for each set of documents with the same ACL, while slightly increase the index maintenance cost. Overall, our approach provides privacy and offers a tunable solution that trades maintenance cost for query performance and vice versa, depending on the performance needs.

1.2 Outline

The subsequent chapters are organized as follows:

In chapter 2 we present essential background knowledge about text indexing, text search, and clustering. Then, we describe the basic access control models and discuss privacy.

In chapter 3 we focus on how an arbitrary user can compromise the search engine results to obtain information that he is not allowed to access.

In chapter 4 we introduce our design goals. Then, we give an overview of our indexing workflow scheme and analyze its individual components. Furthermore, we explain the important decisions made before implementation.

In chapter 5 we provide the details of our implementation.

In chapter 6 we define our experimental environment and methodology, and present the experimental results. We present results from both our indexing workflow scheme and a full-text search engine.

In chapter 7 we review prior related research that focuses on full-text search and privacy protection in multi-user desktop and enterprise environments as well as in social networks. In addition, we review approaches that provide secure data storage.

In chapter 8 we present the conclusions regarding this thesis and discuss possible future research directions.

Chapter 2

Background

In this chapter, we briefly present essential background knowledge on full-text indexing, full-text search, and clustering. Then, we describe the basic access control model and discuss privacy in full-text search engines.

2.1 Text Indexing

When dealing with a small number of documents, it is possible for the search engine to directly scan the content of the documents in order to satisfy a search query. However, when the number of documents is large, the best solution is to divide the search process into two steps: indexing and search. In the indexing step, the content of all documents is parsed and one or more indices are built. In the search step, users submit queries consisting of terms and the search engine returns the documents that are relevant to the query using the indices built in the previous step.

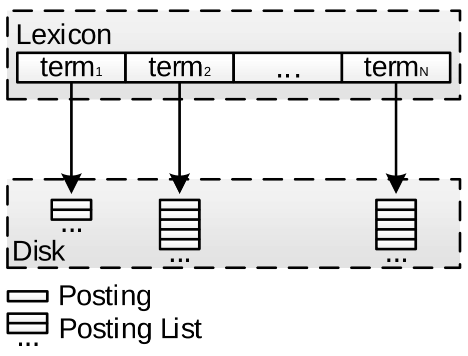


Figure 2.1: The lexicon maps each distinct term to the position of its corresponding posting list on the disk.

2.1.1 Index Structure

In general, several data structures have been proposed for the construction of a full-text search index, such as signature files [13], bitmaps [42], and suffix-arrays [29]. However, the most effective and widely used data structure is the *inverted index* [17], which consists of an *inverted file* and a *lexicon*.

The inverted file stores for each term t a list of pointers to all documents that contain the term. Each pointer in this list is called *posting* and specifies the exact position in the document where the term occurs, while each list of pointers is called *posting list*. The lexicon maps each distinct term t that appears in the documents to the position of its corresponding posting list on the disk (Figure 2.1). It is usually implemented as a hash table or a sorted structure for efficient look up.

Although the inverted index is the preferred and most prominent index structure, the actual choice of an appropriate index construction and maintenance method is important to the search engine performance.

2.1.2 Inverted Index Construction and Maintenance

The main reason that makes index construction challenging is the fact that the volume of data involved cannot be held in main memory. The most commonly used index construction algorithm is the *merge-based inversion* [42]. In merge-based inversion, documents are parsed in batches and their postings are accumulated in memory, constructing the corresponding posting lists. When memory is full, the posting lists are flushed to disk creating a sub-index and then are deleted from memory. Finally, all sub-indices are merged into one on-disk index.

While the previous index construction method is useful for *static collections*, it is not appropriate for *dynamic collections*. In dynamic collections, existing documents are deleted or modified, and new documents are created. Therefore, the search engine needs to keep the indices in sync with the document collection that constantly changes. This task is referred to as *index maintenance*. When a new document is added in the collection, its postings must be added to the posting lists of the existing index. In case of a document deletion, all postings referring to the deleted document must be removed from the posting lists. Document modifications usually are handled as a deletion and re-insertion of the document, ensuring that the search engine returns the new version of the document in search results.

In principle, inserting a single document into an existing index requires the update of every posting list corresponding to a term in the document. For fast insertion, it is necessary to avoid accessing the corresponding disk-resident posting lists every time a new document is added. Therefore, several index maintenance techniques amortize the update cost over a sequence of document insertions.

The *rebuilding* strategy periodically reconstructs the entire index, including the documents added in the collection since the last rebuild. Although the cost of the rebuilding method is prohibitive for large collections, its use is appropriate in some cases. Indeed, many search services use this model, re-crawling documents every day or week and re-indexing them.

A completely different approach, called *Remerge*, has been proposed by Lester et al. [26]. This approach uses one on-disk and one in-memory index which accumulates the postings of new documents. When there is no available memory, the in-memory index is merged with the existing on-disk index, creating a new on-disk index. The drawback of this approach is that it requires the entire on-disk index to be read and written again to disk every time the system runs out of memory.

On the other hand, the *Nomerge* strategy does not perform any merge operations. When memory is full, postings are written to disk, creating a new on-disk sub-index. The on-disk sub-indices are never merged. Thus, when the posting list for a given term needs to be retrieved, its sub-lists must be fetched from all sub-indices. Nomerge is known for its high indexing performance, as each posting is only written once to disk and never read during indexing. However, this approach requires many disk seeks in order to fetch a term's posting list, as each posting list may be split in many sub-indices. Hence, it is impractical for large document collections because a great number of sub-indices is created degrading the query performance.

The two strategies mentioned above represent two extreme cases: Remerge always merges the in-memory index with the on-disk index, while Nomerge never merges two successive in-memory indices. The *LogarithmicMerge* strategy [25] is a compromise between the previous two. Every time memory is full, the in-memory postings are stored as an on-disk sub-index. When the number of sub-indices with similar sizes reaches a predefined threshold (*mergefactor*), these sub-indices are merged into a larger one. For instance, suppose that the mergefactor is 10 and the buffer size used for the indexing process is 32 MB. When the tenth sub-index is about to be written to disk, all the sub-indices are merged into a single index of 320 MB. In the same context, when the number of 320 MB sized indices reaches ten, they are merged into a 3200 MB index. The advantage of this strategy is that the mergefactor provides a trade-off between indexing and query processing performance.

2.1.3 Search Queries

A search engine allows users to submit search queries to find the information they need using the index. Each query consists of terms that describe the information that a user needs to find. A widely used type of queries is the Boolean query. A conventional Boolean query consists of a list of terms combined using operators, such as *AND, OR,* and *NOT*.

The most commonly used Boolean operator is the AND operator. Supposing the following conjunctive query of r terms:

$term_1$ AND $term_2$ AND ... AND $term_r$,

all terms must occur somewhere in a document in order to be included in the query response. The simplest way to answer such a query is to look up each term in the lexicon and retrieve its corresponding posting lists. Then, the intersection of the posting lists is calculated. The procedure begins by picking the posting list of the least frequent term. This list contains a set of candidate documents that might be answers to the query. All remaining posting lists are processed based on this candidate set, in increasing order of term frequency. If a document in the candidate set is not present in any subsequent posting list, it is discarded. This means that the size of the set of candidate documents does not increase. At the end, all documents that exist in the candidate list are those which contain all query terms, and they are returned to the user.

Another commonly used Boolean operator is the *OR* operator. The documents that are included in the result of such a query are derived from the union of the posting lists of the query terms. Thus, these documents may contain one, two, or all the query terms.

2.1.4 Result Ranking

Merely returning the results of a query is not very useful as a search answer. Some of the documents included in the result are relevant to the query terms, while some others are less relevant or even irrelevant. Therefore, the user is forced to make an extra effort in order to identify the documents that are most relevant to his query. The more documents included in the result list, the more difficult and time-consuming the identification process becomes.

One way to help users to easily find the relevant documents is to rank the documents in the returned list. The documents that have a higher probability to be considered relevant by the user are ranked higher. For this purpose, we need a metric that characterizes each document with a relevance score, and gives a good indication of which documents are more relevant to a given query. Using such a metric, the full-text search engine can only return the top-k ranked documents, and the user can restrict the result inspection only to them.

Usually, web ranking algorithms leverage the links between pages in order to infer the importance of a page. Google's PageRank [32] is an algorithm that uses this approach and assigns a numerical value (referred as PageRank) to each page, with the purpose of measuring its relative importance to the query. The PageRank of a page is defined recursively and depends on the number and PageRank of all pages that link to it. Hence, a page that is linked to by many pages with high PageRank receives a high rank itself.

This approach is not appropriate for a file system ranking algorithm as there are no links between files. In order to apply this approach on file systems some approaches attempt to extract semantic information from files. Connections [40] extracts temporal relationships from files based on file usage patterns and builds a graph. Each node of the graph represents a file and each edge represents a link. Two files are linked if they are opened at the same time window. However, this approach can link two files that are opened in the same time window even though they might not be relevant to each other.

In order to improve the ranking results, most relevance ranking functions use a similarity measure to measure the closeness of each document to the query. The underlying principle is that the higher the similarity score awarded to a document, the greater the estimated likelihood that a human would judge it to be relevant. Most similarity measures use some composition of fundamental statistical values:

- $f_{d,t}$, the frequency of term t in the document d.
- $f_{q,t}$, the frequency of term t in the query.
- f_t , the number of documents containing one or more occurrences of term t.
- F_t, the number of occurrences of term t in the collection.
- N , the number of documents in the collection.
- n, the number of indexed terms in the collection.

These basic values are combined in a way that follows three observations:

1. Less weight is given to terms that appear in many documents.
2. More weight is given to terms that appear many times in a document.
3. Less weight is given to documents that contain many terms.

A typical formulation, which is quite effective in practice, calculates the cosine of the angle in the n-dimensional space between a query term $w_{q,t}$ and a document $w_{d,t}$:

$$w_{q,t} = ln(1 + \frac{N}{f_t}) \qquad w_{d,t} = 1 + ln f_{d,t}$$

$$W_d = \sqrt{\sum_t w_{d,t}^2} \qquad W_q = \sqrt{\sum_t w_{q,t}^2}$$

$$S_{q,d} = \frac{\sum_t w_{d,t} w_{q,t}}{W_d W_q} \qquad (2.1)$$

The similarity between the query q and the document d is expressed by the term $S_{q,d}$. In this equation, the term W_q can be neglected as it is a constant for a given query and does not affect the ordering of documents. The quantity $w_{q,t}$ typically captures the property often described as the inverse document frequency of the term (IDF), while $w_{d,t}$ captures the term frequency (TF). A greater TF value means that a document is more relevant if it contains more occurrences of a query term; a greater IDF value means that a query term is more important if it occurs in fewer documents.

One of the most prominent and most sophisticated TF/IDF scoring functions is Okapi BM25 [35]:

$$w_{q,t} = ln(\frac{N - f_t + 0.5}{f_t + 0.5})\frac{(k_3+1)f_{q,t}}{k_3 + f_{q,t}} \qquad w_{d,t} = \frac{(k_1+1)f_{d,t}}{K_d + f_{d,t}}$$
$$K_d = k_1((1-b) + b\frac{W_d}{W_A}) \qquad S_{q,d} = \sum_{t \in q} w_{q,t} w_{d,t} \qquad (2.2)$$

in which the values k_1 and b are parameters, set to 1.2 and 0.75 respectively; k_3 is a parameter that is set to ∞, so that, the expression $(k_3 + 1)\frac{f_{q,t}}{k_3 + f_{q,t}}$ is assumed to be equivalent to $f_{q,t}$. W_d and W_A are the document length and average document length, counted in words or bytes.

In order to evaluate the accuracy of a ranking method, we need some metrics to calculate the portion of the actual relevant documents included in the result and quality of ranking. Two basic metrics are the *precision* and the *recall*. The precision of a ranking method is the fraction of the top-k ranked documents that are relevant to the query, while the recall of a method is the fraction of the total number of relevant documents included in the top-k documents. More to the point, high recall means that an algorithm returns most of the relevant results, while high precision means that an algorithm returns substantially more relevant results than irrelevant.

2.2 Clustering

Clustering is an important and useful technique used in a wide variety of fields, such as pattern recognition, information retrieval, and data mining. Clustering methods group a large number of objects into a small number of meaningful clusters for further processing

[21]. Objects within a cluster are similar to each other and different from objects in other clusters. Therefore, a good clustering method is the one that achieves the greater similarity within a cluster and the greater difference between clusters.

Usually, every single object x, which is used as input to a clustering method, consists of a vector of d dimensions $x = (x_1, x_2, ...x_d)$. The individual components x_i are called *features* or *attributes* and describe the objects. Attributes are used by clustering methods to group the objects.

The basic steps that a typical clustering activity involves are: a) the representation of the objects; b) the definition of an appropriate proximity measure; and c) the clustering process [22]. Object representation refers to the number of available objects, and the number and type of the attributes available to the clustering algorithm. Object proximity is usually measured by a distance or similarity measure defined on pairs of objects and is stored in a matrix whose rows and columns correspond to objects. The clustering process groups the objects by consulting the proximity matrix. The output can be:

- *exclusive*, where each object belongs to exactly one cluster,
- *overlapping*, where an object can simultaneously belong to more than one cluster, or
- *fuzzy*, where each object has a certain degree of membership in each of the output clusters.

Finally, some objects may be considered as *outliers* or *noise* and may not be part of any formed cluster. Outliers are either objects that have different characteristics from most of the objects in the data set, or values of an attribute that are unusual with respect to the typical values for that attribute. On the other hand, the concept of noise is slightly different as it refers to a random component of a measurement error and may involve the distortion of a value or the addition of spurious objects.

2.2.1 Similarity And Distance Measures

Clustering requires a definition of the "closeness" of two objects. Closeness is defined in terms of the similarity measure between two objects. Similarity or Distance measures map the similarity or distance between two objects into a single numeric value. The similarity expresses how similar are two objects. Similarity is higher for pairs of objects that are more alike and lower for pairs that are less alike. On the other hand, distance expresses how different are two objects. When similarity values range from zero (no similarity) to one (complete similarity), then for a given similarity value s, we can compute the distance $d = 1 - s$ and vice verca.

A standard distance measure which is widely used in clustering problems is the *Euclidean Distance*. The Euclidean Distance d between two objects x and y in a n-dimensional space is given by the following equation:

$$d(x, y) = \sum_{k=1}^{n} (x_k - y_k)^2, \tag{2.3}$$

where n is the number of dimensions and x_k, y_k the k^{th} attributes of x and y respectively.

Jaccard Similarity Coefficient is a measure which is used to compute the similarity of objects with asymmetric binary attributes. In this case only "1" matters. For instance, suppose two objects $x = (1, 0, 0, 0, 0, 0, 0)$ and $y = (0, 0, 0, 0, 0, 1, 1)$ each one represented by a binary vector of attributes. The Jaccard Coefficient is given by the equation:

$$J = \frac{f_{11}}{f_{01} + f_{10} + f_{11}}, \tag{2.4}$$

where:

- f_{11} is the number of attributes in which both x and y are "1",
- f_{01} is the number of attributes in which x is "0" and y is "1", and
- f_{10} is the number of attributes in which x is "1" and y is "0".

The number of attributes in which both x and y are "0" does not contribute to the estimation of the similarity value as the presence of an attribute is more important than its absence.

One of the most popular similarity measures is the *Cosine Similarity*. Assuming two objects x and y represented as vectors of attributes, the cosine similarity is given by the following formula:

$$cos(x, y) = \frac{x \cdot y}{\|x\| \, \|y\|}, \tag{2.5}$$

where the numerator of the fraction indicates the vector dot product

$$\sum_{k=1}^{n} x_k y_k \tag{2.6}$$

and $||x||$ reflects the vector length:

$$\|x\| = \sqrt{\sum_{k=1}^{n} x_k^2} = \sqrt{x \cdot x}. \tag{2.7}$$

Cosine similarity ignores zero matches like the Jaccard Similarity Coefficient, but it can also be used to compute the similarity of objects with non-binary attributes. For instance, it is an appropriate measure for computing the similarity of text documents. In such a case, documents are represented as vectors of attributes and each attribute corresponds to the frequency of a term's occurrence in the document.

Finally, not all similarity or distance measures are suitable for a given situation. In addition, choosing the appropriate measure is crucial for clustering, and hence, it is of high importance to understand the effectiveness of different measures in order to choose the best for each case.

2.2.2 Clustering Methods

Several clustering algorithms exist but none of them is universally applicable and appropriate for each kind of dataset or clustering problem.

Partitional Clustering. The most common type of clustering methods can be characterized as *partitional* clustering, which is an exclusive division of the set of objects into clusters.

One of the simplest and most popular clustering algorithms is K-means [28]. K-means represents a cluster by the mean value of all objects contained in it. Initially, it randomly selects k cluster centers. Then, in each iteration, K-means assigns each object to its closest cluster center based on the similarity function and recomputes the center of each cluster. This process is repeated until a convergence criterion is met, for instance there is no possible reassignment of any object from one cluster to another. K-means is popular because it is easy to implement and its time complexity is $O(n)$, where n is the number of objects. A drawback of this algorithm is its sensitivity to the selection of the initial cluster centers and that the number of clusters needs to be defined in advance.

A density-based clustering algorithm called DBSCAN [11] is also a partitional clustering algorithm. DBSCAN overcomes the shortcoming of the K-means algorithm, as the number of clusters is automatically detected, and locates regions of high density that are separated from each other by regions of low-density. DBSCAN's definition of a cluster is based on the notion of density reachability. Basically, an object q is directly density-reachable from an object p if it is closer than a given distance Eps (hence part of its $Eps - neighborhood$) and if is surrounded by a number of objects such that one may consider p and q to be part of a cluster. The q is called density-reachable from p if there is a sequence of $p_1, ..., p_n$ objects with $p_1 = p$ and $p_n = q$ where each p_{i+1} is directly density-reachable from p_i. There is a case where an object q might lie on the edge of a cluster, having fewer neighbors than a given number to count as dense itself. This would halt the process of finding a path ending at the first non-dense object. By contrast, starting the process with p would lead to q. In this case, the process would halt there and q would be the first non-dense object. Due to this asymmetry, the notion of density-connected is introduced: two objects p and q are density-connected if there is an object o such that both p and q are density-reachable from o. A cluster satisfies two properties: a) all objects within the cluster are mutually density-connected, and b) if an object is density-connected to any object of the cluster, it is part of the cluster as well.

DBSCAN requires two parameters: a) *Eps*, which is the radius that delimits the neighborhood area of an object, and b) $MinObjs$, which is the minimum number of objects required to form a cluster. It starts with an unvisited object and retrieves its Eps-neighborhood. If the size of its Eps-neighborhood is larger than MinObjs, then a cluster is formed. Otherwise, the object is labeled as noise. However, this object might later be part of another cluster. If an object is found to be a dense part of a cluster, then its Eps-neighborhood is also part of that cluster. This process is repeated until the density-connected cluster is completely found and then, a new unvisited object is processed.

Hierarchical Clustering. Another well-known type of clustering method is characterized as hierarchical clustering and includes two basic approaches [31]:

- The *agglomerative* approach starts the clustering process with each object as an

Subject \ Object	$File_1$	$File_2$	...	$File_N$
$User_1$	rwx	r		rwx
$User_2$	r	rw		rwx
...				
$Group_1$	x			rwx

Table 2.1: Access Control Matrix

individual cluster. Each step of this approach merges two clusters that are the most similar. Thus the total number of clusters decreases after each step. This is repeated until the desired number of clusters is obtained or only one cluster remains.

- The *divisive* approach reverses the clustering process and starts with just one cluster that contains all the objects. Afterwards, the single cluster is split into two or more clusters until the number of clusters becomes equal to the number of objects, or equal to a number specified by the user.

Generally, hierarchical clustering is preferred when a hierarchy is required and is displayed using a tree-like diagram. Although the number of clusters does not need to be specified in advance, a termination condition has to be defined. However, the major drawback of hierarchical algorithms is the high computational and storage cost they involve.

2.3 Access Control

In multi-user environments many users are allowed to coexist and interact with each other. Generally, such systems distinguish individual users through authentication at login time and associate an identity with each file or folder. One challenge in multi-user environments stems from the fact that all files are stored in a shared storage space. With no measures taken, it is possible for users to have access to any file. Therefore, there is an imperative need to use an access control mechanism.

Access control matrices can be used to implement access control mechanisms (Table 2.1). These matrices store for each *subject* (user or group) its access rights (Read, Write, eXecute) on distinct *objects*. An object can be a file, folder, or another system resource.

The columns of the matrix refer to objects and the rows refer to subjects. One way to simplify the management of access rights is to store the access control matrix by columns along with the object to which the column refers. This is called an Access Control List (ACL) and contains subjects as well as their access rights on the object to which the ACL refers. Another way to manage the access control matrix is to store it by rows. Each row is called *capability* and refers to the access rights of a subject on each object.

Generally, when a user makes a request to access a file, a check is made to ensure that he has the appropriate access rights, otherwise the access is denied. This mechanism is known as authorization.

2.4 Search Privacy

As full-text search is an indispensable tool for finding information, search engines need to protect privacy when operating in a multi-user environment. Privacy protection in search engines is considered a non trivial problem to solve, especially when there is a need to retain the search engine's performance.

In order to protect privacy in search engines, the primary goal is to ensure that the search engine respects the access control restrictions applied on the documents. This means that whenever a user issues a query, the search engine must only return documents that the user is allowed to read. However, in some cases this is not enough to preserve privacy.

In the case where the search engine uses a system-wide index, a user can infer information about documents that is not allowed to read. When a system-wide index is used, it is necessary to filter the results of a query before returning them to the user. In particular, the search engine computes the list of all documents matching the query, ranks them depending on their relevance scores, and then it filters this list. Filtering removes from the list all documents that the user who issued the query is not allowed to read. This postprocessing approach along with a TF/IDF ranking algorithm permits a user to leverage the ranking results in order to infer information about documents that he is not allowed to read [6].

This is the case where privacy in a search engine is not preserved and the impacts of information leakage could become disastrous if sensitive information is disclosed. Under these considerations, in order to protect privacy, a search engine needs to ensure that a result list returned to a user only contains documents that he is allowed to read and that a user cannot infer any information about documents that he is not allowed to read.

2.5 Summary

Full-text search is separated into two stages: the indexing, and the search stage. In the indexing stage, the search engine parses and indexes the documents in one or more indices. The most popular index structure is the inverted index and possible techniques that can be used to keep the indices in sync with the document collection are: a) Rebuild, b) Remerge, c) Nomerge, and d) LogarithmicMerge. In the search stage, users issue queries to the search engine. The search engine computes and ranks the list of documents matching the query, and then returns the ranked list to the user.

Clustering methods are used to group objects into a small number of meaningful clusters. Clustering is based on the similarity or distance between two objects, and two basic types of clustering methods can be characterized as partitional and hierarchical.

In multi-user environments, users enforce access control over their files defining who can access them and how. Hence, search engines that operate in such environments need to protect privacy ensuring that a result returned to a user only contains documents that he is allowed to read, and there is no possibility for a user to infer any information about documents that he is not allowed to read.

Chapter 3

Privacy Threats in Full-Text Search

When using a single system-wide index, all existing files are indexed regardless of access control privileges. Thus, whenever a search query is issued, the results are filtered in order to exclude documents for which the user may not have the appropriate access privileges. However, the filtering takes place after ranking all matching documents.

In this chapter, we focus on the methods that an arbitrary user can apply to compromise the search results in a multi-user environment. All of the methods we describe are discussed in a previous work by Büttcher and Clarke [6]. The lack of per-user relevance scores in conjunction with the above post-processing approach and a TF/IDF scoring function can be exploited by an arbitrary user to calculate the number of documents that contain a given term T. The calculation includes those documents that the user is not allowed to search. Extracting the exact number of documents or an approximation depends on whether the full-text search engine returns the document relevance scores or simply the ranked list of documents. Also, it is possible to reveal the content of a document if the full-text search engine supports complex queries. However, TF/IDF scoring functions are the most prominent scoring functions and should be used in a search engine while retaining privacy.

3.1 Attacking Privacy Through Relevance Scores

We assume a full-text search engine that uses a system-wide index and a TF/IDF scoring function (Okapi BM25) to perform relevance ranking on documents matching a search query. The execution of the following steps lead to the exact calculation of the number of documents containing a term T.

A possible starting point is the equation that calculates the relevance score of a document d:

$$score(d) = \sum_{(T,q_T)\in Q} \frac{q_T \cdot w_T \cdot d_T \cdot (1+k_1)}{d_T + k_1 \cdot ((1-b) + b \cdot \frac{dl}{avgdl})}, \tag{3.1}$$

where d_T is the number of occurrences of the term T within d, q_T is the weight of T in the query count by the number of its occurrences, and $w_T = log(\frac{|D|}{|D_T|})$ is the *IDF* weight of the query term T. $|D|$ is the total number of documents and $|D_T|$ is the total number of documents containing T. dl is the length of the document d (number of terms), and $avgdl$ is the average document length in the system. Parameters k_1 and b are usually chosen as 1.2 and 0.75 respectively.

For the given term T that an arbitrary user $User_A$ is interested in, he needs to obtain the number of documents that contain it by solving equation (3.1) for $|D_T|$. However, the value of $|D|$ and $avgdl$ is unknown. $User_A$ creates documents containing specified terms and issues customized queries to the search engine. By leveraging the relevance scores of the returned documents and the above equation, he can determine the value of the unknown parameters, and finally the value of $|D_T|$.

Initially, $User_A$ generates two random terms T_2 and T_3 that do not appear in any document in the collection, and then creates three documents D_1, D_2 and D_3 such that

- D_1 contains only the term T_2,
- D_2 consists of two occurrences of the term T_2,

- D_3 contains only the term T_3 .

The next step is to issue two queries, one only containing the term T_2 and another the term T_3. For the first query, D_1 and D_2 are returned as matching documents, while document D_3 is returned for the second query. Their relevance scores are revealed by the search engine using the equation (3.1). Note that only the term T_2 contributes to the relevance score of D_1 and only the term T_3 contributes to the relevance score of D_3. Also, the weight of T_2 in the query and its number of occurrences within D_1 are equal to 1. This also holds for the term T_3 and the document D_3. Hence, the relevance scores of the documents are:

$$score(D_1) = \frac{log(\frac{|D|}{2})(1 + k_1)}{1 + k_1((1 - b) + \frac{b}{avgdl})} \tag{3.2}$$

and

$$score(D_3) = \frac{(1 + k_1)log(\frac{|D|}{1})}{1 + k_1((1 - b) + \frac{b}{avgdl})}. \tag{3.3}$$

Then, dividing equations (3.2), (3.3) results in

$$\frac{score(D_1)}{score(D_3)} = \frac{log(\frac{|D|}{2})}{log(\frac{|D|}{1})}, \tag{3.4}$$

and thus

$$|D| = 2^{(\frac{score(D_3)}{score(D_3) - score(D_1)})}. \tag{3.5}$$

Now, the only unknown value is the average document length ($avgdl$) in the system, which can be obtained by using equation (3.3) and solving for $avgdl$.

Once all parameters of the BM25 scoring function are known, the attacker creates a new document D_4 which contains the term T, and submits a query including only the term T. Consequently, the search engine returns the document D_4 accompanied by $score(D_4)$. Finally, this information is used to construct the equation:

$$score(D_4) = \frac{(1 + k_1)log(\frac{|D|}{|D_T|})}{1 + k_1((1 - b) + \frac{b}{avgdl})}, \tag{3.6}$$

where D_T is the only unknown value. Hence, solving for D_T, the $User_A$ knows the exact number of documents containing the term T.

3.2 Attacking Privacy Through Ranking Results

Returning the relevance scores of the matching documents is an essential prerequisite to achieve the above exploitation. Nonetheless, following a barely different approach, it is possible to compute an approximation of the number of documents containing a term T, even if the relevance scores of matching documents are omitted. Indeed, this can be accomplished by simply leveraging the order in which matching documents are returned by the full-text search engine and the observation that the most interesting terms are infrequent.

Suppose that $User_A$ intends to reveal the number of documents containing the term T ($|D_T|$). $User_A$ creates documents containing specified terms and issues customized queries to the search engine. Then, he leverages the order in which matching documents are returned and the equation (3.1).

Initially, he creates a single file D_0 which contains only the term T. Then, he generates a unique random term T_2, and creates 1000 documents $D_1 ... D_{1000}$, each of which contains this term. Afterwards, by submitting a Boolean *OR* query comprising terms T and T_2, the search engine returns the matching documents ranked by their BM25 relevance score. If D_0 appears before any of the documents $D_1 ... D_{1000}$, $User_A$ can deduce that $score(D_0) \geq score(D_{1000})$ holds. Hence, solving the inequality for $|D_T|$, he knows that $|D_T| \leq 1000$. Instead, if D_0 appears after the documents $D_1 ... D_{1000}$, then he knows that $|D_T| \geq 1000$.

Furthermore, a better approximation of $|D_T|$ can be achieved by using the following strategy. At the beginning, the arbitrary $User_A$ generates a second random term T_3 and creates 1000 documents ($D_1 ... D_{1000}$), each containing the two terms T_2 and T_3. Also, a third random term T_4 is generated and 999 documents ($D_{1001} ... D_{1999}$) are created, each of which contains T_4. Lastly, one more document D_0 which contains the two terms T and T_4 is needed.

After creating all these documents, $User_A$ submits a query consisting of terms T, T_2, T_3, and T_4. The returning relevance scores for the matching documents D_0 and $D_1 ... D_{1000}$ are computed using the equation (3.1):

$$score(D_0) = C(log(\frac{|D|}{|D_T|}) + log(\frac{|D|}{|D_{T_4}|})) \tag{3.7}$$

and

$$score(D_i) = C(log(\frac{|D|}{|D_{T_2}|}) + log(\frac{|D|}{|D_{T_3}|})), \tag{3.8}$$

for $1 \leq i \leq 1000$ respectively. The constant C is the same for all documents and is given by the equation:

$$C = \frac{(1+k_1)}{1 + k_1((1-b) + \frac{2b}{avgdl})}. \tag{3.9}$$

Then, $User_A$ begins deleting the documents $D_{1001}\, to\, D_{1999}$, one at a time. In the early stages, $score(D_0)$ is lower than $score(D_1)$, but as soon as the number of documents containing the term T_4 is reduced, T_4 acquires a greater *IDF* weight and document D_0 acquires a greater score. After d document deletions, $User_A$ knows that:

$$\begin{aligned} score(D_0) \geq score(D_1) &\Rightarrow C(log\frac{|D|}{|D_T|} + log\frac{|D|}{|D_{T_4}|}) \geq C(log\frac{|D|}{|D_{T_2}|} + log\frac{|D|}{|D_{T_3}|}) \\ &\Rightarrow log\frac{|D|}{|D_T|} + log\frac{|D|}{1000-d} \geq 2log\frac{|D|}{1000}, \end{aligned} \tag{3.10}$$

since $|D_{T_2}| = |D_{T_3}| = 1000$ and $|D_{T_4}| = 1000 - d$. Right before this point, say at $d - 1$ deletions, $score(D_0)$ is still lower than $score(D_1)$, and hence the equation

$$log\frac{|D|}{|D_T|} + log\frac{|D|}{1000-(d-1)} \leq 2log\frac{|D|}{1000} \tag{3.11}$$

holds. Ultimately, combining the inequalities (3.10), (3.11) gives:

$$-log(|D_T|) - log(1000-d+1) \leq 2log(1000) \leq -log(|D_T|) - log(1000-d) \tag{3.12}$$

which implies

$$\frac{1000^2}{1000-d+1} \leq |D_T| \leq \frac{1000^2}{1000-d}. \tag{3.13}$$

Hence, the attacker is capable of obtaining a range of possible values, one of which corresponds to the actual number of documents containing the term T. However, if the approximation gained is not good enough, the process can be repeated by adding more than two terms per document.

3.3 Revealing the Content of Documents

The methods described above can be used to obtain the number of documents that contain a particular term. While this is already harmful, it can be much worse if the full-text search engine allows queries of arbitrary length. For instance, it is possible to obtain the entire content of a document by guessing its words. In particular, knowing that a certain document contains the phrase "A B C", an arbitrary user can try all possible terms D and calculate the number of documents which contain "A B C D" until he finds a D that gives a non-zero result. Afterwards, he can continue with the next term E and so on.

3.4 Towards a Secure and Efficient Search System

It is shown that a post-processing approach combined with an TF/IDF scoring function permits to an arbitrary user to infer file contents without actually reading any files. However, TF/IDF scoring functions are the most popular and the most prominent scoring functions. A search engine that uses a TF/IDF scoring function can achieve accurate query results, and hence, we could benefit from using a TF/IDF scoring function in such a way that privacy is retained. Nevertheless, there is a need of approaches that combine privacy and accurate results with search performance and reasonable index maintenance cost, leading to a secure and efficient search engine.

3.5 Summary

Search engines that use a single system-wide index along with the post-processing ranking approach and a TF/IDF scoring function can pose a severe privacy threat. More precisely, an arbitrary user can compromise the search results in a multi-user environment. He can calculate the number of documents that contain a given term T by leveraging either the relevance scores or the position of the documents in the ranking list. Furthermore, it is possible to reveal the content of documents that he is not authorized to access when the search engine supports Boolean queries and phrases. However, TF/IDF scoring functions are widely used as they are the most prominent scoring functions. A search engine needs them in order to provide accurate query results, and hence they should be used in such a way that also retains the privacy.

Chapter 4

Design

This chapter is devoted to the design goals and the overview of our work. Furthermore, we describe its major components in more detail and explain any decisions made before proceeding with the implementation.

4.1 Goals

Much of the research work on preserving privacy in desktop and enterprise multi-user environments follows two basic approaches. These approaches maintain either one index per user or a single system-wide index.

Retaining one index per user ensures privacy and achieves high query performance, as each user only accesses his private index and the query processing is restricted to that index. Unfortunately, these significant benefits come with great disk space consumption and increased maintenance cost because each document is indexed by every index whose owner has access to it. Hence, multiple copies of the same document exist in several indices at any given time. Furthermore, the problem gets worse when many users share many documents.

On the contrary, retaining a system-wide index means that all existing documents are indexed regardless of access control restrictions [6]. This approach provides efficient index updates and low storage usage because each document is indexed only once. However, it poses severe privacy issues which can be eliminated in the expense of query performance due to the need of result filtering in order to ensure that each search result only contains documents that the respective user is allowed to search.

A different approach that improves efficiency while ensuring privacy creates one index for each set of documents with the same ACL [37]. In particular, documents accessed by a single user are indexed by his private index. Instead, shared documents with identical ACLs are indexed by a single index, which is accessed by the users of the specified ACL. Even though this approach offers privacy and lower maintenance cost, it does not provide any parameters to tune the query performance and the maintenance cost.

The main purpose of our solution is to protect privacy in multi-user environments and provide a more flexible solution. We introduce a novel strategy to organize users' documents into indices by leveraging the similarity of their ACLs. We group documents into clusters, each containing documents with similar ACLs to some extent. The similarity between the ACLs of documents within a cluster is determined by a Similarity parameter. Then, we map the documents and the users to indices based on the intersection and differences of the ACLs within a cluster. In addition, we use a Threshold parameter which determines how a difference is treated, and limits or raises the duplicate documents. Therefore, we provide a trade-off between query performance and maintenance cost and ensure that the indices accessed by a user only contain documents that the user is allowed to search.

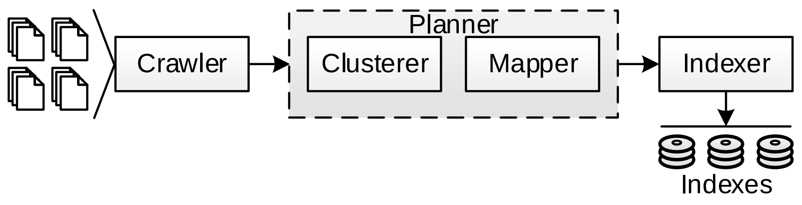


Figure 4.1: Overview of our indexing workflow.

4.2 Overview of Indexing Workflow

Our design is based on a planning scheme that groups users' documents before the indexing process. Figure 4.1 gives an overview of our indexing workflow:

- A *Crawler* gathers the access control information of the documents in the system.
- The *Planner* utilizes this information to separate the documents. The Planner consists of two components: a) the *Clusterer*, which clusters the documents based on the similarity of their ACLs; and b) the *Mapper*, which maps documents and users within a cluster to indices. Note that each document may be indexed by more than one index.
- The *Indexer* indexes the documents in the full-text search engine based on information obtained from the Planner.

The details about the distinct components and the algorithms used to implement our solution are given in the following subsections.

4.3 Crawler

Crawling is an important and essential part of a search engine and typically refers to the process of discovering the content to index. However, its operation is more general as it is used to express the process of collecting information within a system.

In our case, crawling refers to the discovery of new or updated documents, the extraction of the document ACLs, and the extraction of the information about their location on disk. The content of the documents is not used in the next phase, so there is no need to get it in such an early stage: the Planner only needs the ACL and the path of the documents. Therefore, the crawler is used to extract only this information from the documents and fuel the Planner with them.

4.4 Planner

The Planner executes the main bulk of the work, and receives a pair of path and ACL for each document in the system. Then, the Clusterer uses this information to group the documents into clusters, and the Mapper maps the documents and the allowed users to indices.

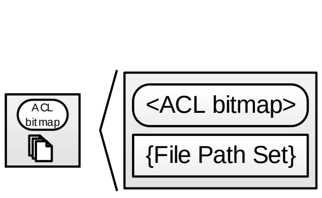
4.4.1 Clusterer

The Clusterer is responsible to cluster the documents based on the similarity between their ACLs. Before starting the clustering process, a number of important decisions should be made in order to cluster the documents in an efficient way.

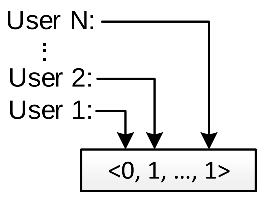
Object Representation. Once the Clusterer obtains the document paths along with their ACL, the clustering process starts by representing the objects. The next step is to construct the similarity matrix by computing the similarity of each pair of objects and storing it in the appropriate matrix slot. Normally, each object would be represented as a pair of a document path and its corresponding ACL. However, in the case where the number N of documents is high, the similarity matrix would considerably grow in size with $N \times N$ slots. For instance, assuming $N = 50000$ is the number of documents and $b = 4\,Bytes$ is the size of each matrix slot, then the total memory consumption becomes $C = N \times N \times b = 9\,GB$. Even when using only half of the similarity matrix, it still occupies a lot of memory.

In order to avoid this cost, we represent the objects in a more suitable way. To achieve this, we add an extra document-grouping step before the actual clustering process. In particular, we gather in the same group all documents with identical ACLs, and this group represents the actual object for the clustering process.

We denote each object as a *Document Family*. Each Document Family consists of a set of document paths (rather than a single document path) and a binary *ACL bitmap* generated by the ACL of the corresponding document set (Figure 4.2(a)). The dimensions



(a) Document Family.



(b) ACL bitmap.

Figure 4.2: Document Families include a set of document paths and their corresponding ACL bitmap. ACL bitmaps are created from the ACLs and represent whether each user is included in a document's ACL or not.

of the ACL bitmap are equal to the number of the users in the system and its components are either one or zero depending on whether the corresponding user appears in the ACL of the Document Family or not (Figure 4.2(b)).

The importance of such an object representation is demonstrated by the dramatic reduction of memory consumption. The size of the similarity matrix depends on the number of different ACLs in the system rather than the total number of documents. Moreover, the number of different ACLs is expected to be small compared to the total number of documents as many documents have common sharing attributes [23].

Choosing the Clustering Algorithm. Several clustering algorithms exist in order to meet different needs as none of them is universally applicable and appropriate for every kind of dataset and clustering problem. However, two of them captured our interest.

One algorithm that we initially considered was K-means. After examining its properties and prerequisites, we realized that it eventually might not be such a good choice. The main reason that made us disregard K-Means is the demand to specify in advance the number of clusters to which the clustering method will end up. In our approach, this is an obstacle because we do not know the number of the final clusters. Moreover, the distance measure used by default to compute the distance from a data object to each cluster center is the Euclidean distance. As highlighted previously [10], Euclidean distance does not work well in high dimensions and its performance may not be optimal when dealing with binary data.

Taking into account these considerations, we decided to use another algorithm: the

DBSCAN algorithm. Unlike K-means, DBSCAN does not require to predefine the number of the clusters. Instead, the final number of clusters is revealed after the algorithm has been executed. Furthermore, one can choose an arbitrary distance function rather than the Euclidean distance.

Choosing the Similarity Measure. Given the number and diversity of similarity and distance measures that are available, choosing one is also a challenging process. Taking into account that each Document Family is represented by an ACL bitmap and that the presence of an attribute is more important than its absence, we decided to use the Jaccard Coefficient. Although DBSCAN works with distance functions, we prefer to use the notion of similarity because we want to focus on the similarity between the ACLs rather than their dissimilarity.

Choosing the Algorithm Parameters. As already discussed in subsection 2.2.1, the DBSCAN algorithm requires two parameters: a) the *Eps* radius, which defines the maximum distance between two objects in order to be considered as neighbors and is computed by the *Similarity* parameter; and b) the *MinObjs* value, which defines the minimum size of the neighborhood that a particular object must have in order to be included in a cluster.

Our goal is to cluster all the Document Families and avoid characterizing any of them as noise. Thus, we permit the creation of clusters that contain a single Document Family. For this reason, we set the MinObjs value to one. We do not set the radius to a fixed value as our target is to monitor the behavior of our solution under different values of similarity. Hence, the tunable Similarity parameter defines its value.

The Clustering Process. So far, we explained several important decisions concerning the algorithms of our solution; now we move to the description of the actual clustering process.

The algorithm starts with an arbitrary Document Family F which is not yet member of any cluster. It marks F as visited and retrieves its *Neighborhood* which is the Document Families whose similarity to F is equal or greater than the Similarity value. The Document Families belonging to the Neighborhood are added in the *Neighbors List.* If the size of this list is equal or greater than one, a new cluster is formed containing F. Then, the algorithm examines every Document Family F' in this list. If F' is not visited, it is first marked as visited and its Neighborhood is retrieved. If the Neighborhood size is ≥ 1,

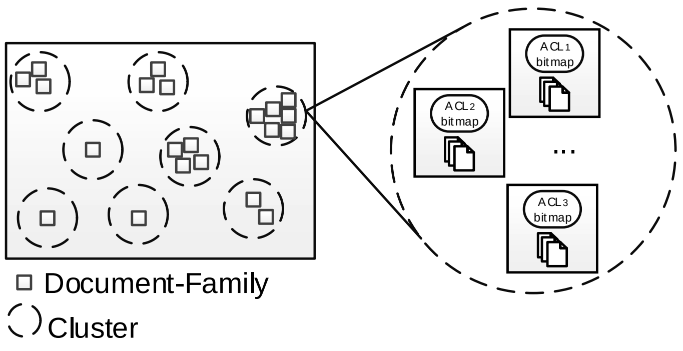


Figure 4.3: We illustrate an example of Document Family clustering. We assume a similarity value higher than 0 and smaller than 100. The formated clusters contain either one or more Document Families. In particular, a cluster with more than one Document Families includes multiple sets of document paths, and each set has a single ACL bitmap.

then the neighbor Document Families are added in a temporary list which is joined with the Neighbors List. This process is repeated until no more Document Families are left in the Neighbors List. Subsequently, another Document Family of the dataset is visited.

After all Document Families have been visited and assigned to a cluster, each cluster ends up with one or more Document Families. The number of documents included in clusters with more than one Document Families is at least equal to the number of different Document Families in the cluster. Some of these documents have different ACLs but similar to some extent. This similarity refers to the common users between their ACLs and depends on the Similarity value. Note that even clusters with a single Document Family contain more than one documents, but these documents have exactly the same ACL.

Figure 4.3 depicts an example of clustering output. Each cluster contains either one or more Document Families, and each Document Family represents a set of documents with identical ACL. Clusters with more than one Document Families include multiple sets of documents each of which has its own ACL bitmap. The ACL bitmaps of the Document Families included in the same cluster are similar to each other.

At this point, the Clusterer job is done and the generated clusters are given as input to the Mapper.

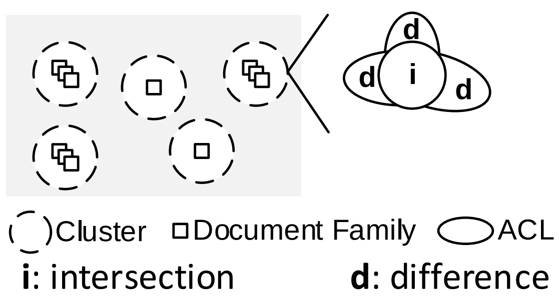


Figure 4.4: Clusters with more than one Document Families contain documents with different ACLs. For these clusters, we find the intersection and the differences between the ACLs they contain. The intersection is the set of users included in every Document Family in the cluster, while each difference is the set of users of each ACL in the cluster that is not included in the intersection.

4.4.2 Mapper

Mapping is a stage of high importance because it determines how the indices are formed. The output of this step is the number of indices and a description for each index. The information that describes each index is: a) its name; b) the set of users that have access to it; and c) the paths of documents that are going to be indexed by it. This information will be later used by the Indexer, which performs the actual indexing process.

The underlying idea of our Mapper is based on the observation that a cluster may contain documents with similar ACLs to some extent. For each cluster, the Mapper creates an intersection and multiple difference ACL parts (Figure 4.4). The intersection ACL part is formed by the intersection of the ACLs (*ACL intersection* or simply intersection) in a cluster. An intersection contains the set of users that are included in the ACL of every Document Family in a cluster. Each difference ACL part is formed by the remaining portion of each ACL (*ACL difference* or simply difference) in a cluster and contains the set of users of this ACL that are not included in the intersection. The users and the documents that correspond to these ACL parts are then mapped to indices.

Algorithm 1 provides a high-level description of how the Mapper works. For each cluster, it performs three major tasks: 1) it finds the intersection between its ACLs; 2) for the ACL of each Document Family, it computes the set of users that do not belong to the intersection; and 3) decides whether a difference part maps to one or more indices according to a threshold. The inputs to the algorithm are the created clusters and the

Algorithm 1: Index Mapping

```
   Input: Clusters of Document Families and Threshold value
   Output: Information about each index (name, user IDs, and documents IDs)
1  for each $c \in Clusters$ do
2      // Compute the intersection in cluster c
3      findIntersetion(c)
4      if (!intersection.empty() AND
       $\exists DocumentFamily1, DocumentFamily2 \in c : DocumentFamily1 \neq DocumentFamily2$) then
5          // Map one index to the users of the intersection and all documents of cluster c
6          mapIndex(intersection[c].users, c.Documents)
7          // Compute the differences and map to indices based on the Threashold
8          for each DocumentFamily $\in c$ do
9              findDifference(DocumentFamily.users, intersection[c])
10             indexWithTreshold(difference.users, DocumentFamily.Documents)
11         end
12     else
13         // Map the indices based on the Threshold value
14         for each DocumentFamily $\in c$ do
15             indexWithTreshold(DocumentFamily.users, DocumentFamily.Documents)
16         end
17     end
18 end
```

value of the threshold.

Initially, for each cluster c, we check whether a non-empty intersection exists. If a non-empty intersection exists, the Mapper maps a single index to the users of the intersection and indexes all documents of the cluster in that index (line 6). Then, we compute the difference with the intersection for each individual Document Family in the cluster c and map one or more indices depending on the threshold value (lines $8-11$). When intra-cluster intersection is empty, it means that either the cluster has a single Document Family or the cluster has more than one Document Families but there are no common users between their ACLs. In both cases, each Document Family is mapped to indices based on the threshold value (lines $14-16$). In particular, if the product of the number of users that belong to a difference part with the number of its corresponding documents is higher than the threshold value, then one index is mapped to these users and documents. Otherwise, the documents are duplicated to the private index of each user in the difference part (Algorithm 2).

Intra-Cluster Intersection and Differences. The fact that the ACLs within a cluster may share common users is leveraged to create an index. This index contains

Algorithm 2: Procedure indexWithTreshold(users, documents)

```
1 if (users.size() × documents ≥ Threshold) then
2     mapIndex(users, documents)
3 else
4     cpInPrivateIndexes(users, documents)
5 end
```

all documents included in the cluster (regardless of their ACL) and is accessed only by users that belong to the intra-cluster intersection. Non-empty intersections only exist in clusters that have more than one Document Families. In order to find an intersection, we retrieve the ACLs by using the corresponding ACL bitmaps. We are certain that each intersection is mapped to a single index either by using an existing index with the same set of users accessing it, or by creating a new one. However, if an empty intersection exists, then each ACL is treated as a difference.

Except for the intersections, we also have to take care of the differences. The default case maps each difference and its corresponding documents to a single index. This index is only accessed by the users that belong to the difference. The index of a difference only includes documents that correspond to the specified ACL rather than the total documents held by the cluster. Unavoidably, this leads to document duplication as these documents are indexed both in the index mapped to the intersection of the cluster and in the index mapped to the difference. In addition, we observe that there might be common users between two or more differences, but we do not consider them for the moment because the solution would become more complicated.

Privacy, our most important goal, is achieved as each mapped index is only accessed by users that are allowed to search the indexed documents. While this is a good solution and keeps the number of document duplicates at low levels, it does not bring the best search performance. Even though the number of indices that a user needs to search is reduced, especially for the users that belong to intersections, a more intuitive mapping can further improve the query performance.

Map to One or Multiple Indices. To further improve the query performance while still meeting the privacy, we introduce a threshold denoted as *Threshold.* Threshold limits or raises the document duplicates and this translates to the increase or the decrease of the indices that each user has access to.

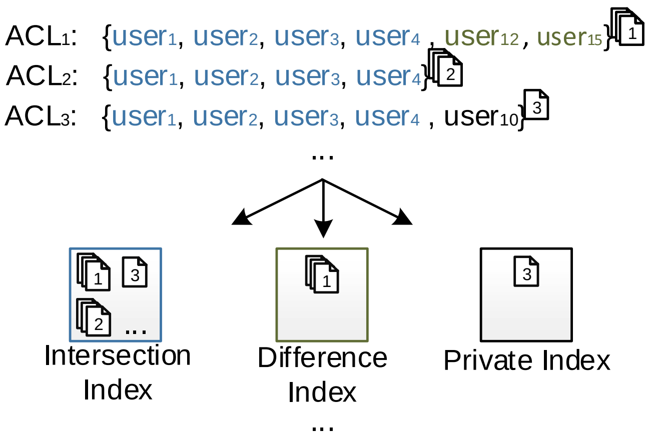

ACL1: {user1, user2, user3, user4 , user12, user15}
ACL2: {user1, user2, user3, user4}
ACL3: {user1, user2, user3, user4 , user10}
...
Intersection Index
Difference Index
Private Index
...

Figure 4.5: Users $U_1 ... U_4$ belong to the ACL intersection, while users U_{12}, U_{15} and U_{10} belong to the ACL differences. The Mapper maps three indices. The first index corresponds to the users of the ACL intersection and includes all documents in the cluster. Each of the remaining indices corresponds to the users of each ACL difference. Also, the set of documents that correspond to each ACL difference is duplicated in each mapped index (difference and private index).

Using this threshold, the Mapper decides whether to map a difference to a single or multiple indices. Therefore, every time a difference is computed, the algorithm checks whether the number of the corresponding documents multiplied with the number of users in the difference is lower than the predefined Threshold or not. If this product is lower than Threshold, then each of these documents is indexed by the private index of each user that is included in the difference. Otherwise, we treat the difference as in the default case by mapping it to a single index.

Figure 4.5 illustrates an example of the mapping process. Considering the cluster of Figure 4.3, we show the ACLs and the documents corresponding to each ACL bitmap. The Mapper computes the intersection and maps it to an *intersection index*. The intersection index contains all the documents included in the cluster. Moreover, only users $U_1, ..., U_4$ that belong to the intersection have access to it. Also, the Mapper maps each difference to a *difference* index where the corresponding documents are duplicated. Thus, a difference index is mapped for users U_{12} and U_{15}, while a private index is mapped for user U_{10} as he is the only user that belongs to the third difference.

Mapping the users and documents of each difference to a single index is not always appropriate. In the case where differences include few users and documents, we need to maintain many indices that include a small number of documents and each is accessed by a small number of users. Moreover, if a user belongs to many such differences, then the number of indices that he accesses is large. On the other hand, mapping the documents of each difference in the private index of each user would lead to many document duplicates in the case where the number of users and/or documents in the difference is high. Thus, we decide whether to map a difference to a single or multiple indices by checking the product of the number of documents and users in the difference. This product shows the number of document duplicates that are going to be created per ACL. We can limit the number of document duplicates by creating a single index for a difference if the above product is higher than the predefined Threshold, or raise it by duplicating the corresponding documents in the private index of each user in the difference if the product is lower than the Threshold.

Indeed, this approach tends to further reduce the number of indices in which a user has to search when some of the shared documents are duplicated in his private index. However, the higher the Threshold value, the fewer the indices that a user has access to and the more documents are duplicated into multiple indices.

4.5 Indexer

The last remaining phase is to index the documents by leveraging the information generated by the Mapper. To that end, the Indexer gets one by one the index names and their corresponding document paths. The documents are indexed by the specified index in bulks of 500 documents or less if not enough. Through bulk indexing the time spent in indexing phase can be substantially reduced.

The Indexer gets the document paths that belong to each index rather than the indices in which each document is indexed. We choose this approach because it incurs less overhead compared to the second one. This happens due to the fact that in the first case, the writes included in the bulk index request occur in the same file (corresponding to a single index). On the contrary, in the second case, each bulk consists of requests, each of which involves the same document but corresponds to different index. Therefore, this translates into many small writes in multiple indices (each corresponding to at least one on-disk file) and degrades the indexing performance.

4.6 Incremental Indexing

The procedure described above deals with the construction of the indices. However, in general we want the indices to handle incoming updates and queries in such a way that the privacy level already achieved remains intact.

4.6.1 Updates

When new documents are added into a document collection, the search engine needs to update the existing index data-structure. Existing index-maintenance strategies accumulate postings from incoming documents in main memory and add them to the existing on-disk inverted lists when a pre-defined memory utilization threshold is exceeded.

In our solution, things are slightly different as we maintain more than one indices. Hence, we have to find the appropriate index to insert a new document with respect to its corresponding ACL. Therefore, each new document is indexed by the index which is accessed by the same set of users that are included in the document's ACL. In the case where such an index does not exist, we create a new one.

This approach raises two issues: a) the number of the total indices might be increased if documents, whose ACLs does not match to any of the existing indices, appear very often; and b) the new indices may retain a small number of documents. However, a possible solution is to periodically re-cluster the documents and re-create the indices.

Apart from the forthcoming documents, one such system needs to deal with changes made to the ACLs of already indexed documents. This implies that these documents are deleted and reinserted in accordance with the preceding procedure.

4.6.2 Search

A user should be informed of the indices in which he has access to before he starts submitting search queries. Thus, we assume that each user is authenticated to an authentication server in order to ensure that a user is the person he claims to be. Then, he receives the corresponding list of the indices and he is able to search the indexed document collection. Consequently, his queries are only directed to the indices included in that list.

In order to achieve this, the authentication server needs to maintain information regarding the users and the indices that each user has access to. In addition, the authentication server needs to be aware of any changes concerning this information and constantly being kept up with them.

4.7 Summary

Much of the previous research on full-text search in multi-user environments presents solutions that offer either high query performance but increased maintenance cost, or low maintenance cost at the expense of slow queries and privacy issues. New approaches protect privacy and improve efficiency, but lack a tunable solution that trades the cost of index maintenance and query performance while ensuring privacy.

With this in mind, we propose an indexing workflow scheme that organizes documents into indices by leveraging the similarity of their ACLs. Our main idea is to create clusters of documents with similar ACLs to some extent and then create indices based on the intersections and differences of the ACLs of each cluster. We ensure privacy in multi-user environments while introducing a trade-off between index maintenance cost and query performance.

Chapter 5

Implementation

In this chapter we provide details of our implementation, which involves the Indexer and the Planner with its two main components: a) the Clusterer; and b) the Mapper. The Planner implementation involves the C/C++ programming language and the STL library as well. The Indexer is implemented in Perl v5.10.1. Additionally, we present a brief discussion of why we use document IDs and not document paths in each Document Family.

5.1 Planner

The Planner uses two parameters, the Similarity and the Threshold used by the Clusterer and the Mapper respectively (Table 5.1). The Clusterer uses the Similarity parameter to cluster documents with similar ACLs to some extent and defines how similar are the

Planner Parameter	Description
Similarity	Defines how similar are the ACLs of the Document Families within a cluster. It is used by the Clusterer.
Threshold	Defines how each Document Family difference within a cluster is treated. It is used by the Mapper.

Table 5.1: Planner Parameters.

ACLs within a cluster. The Mapper computes the intersection and the differences of the ACLs within a cluster and then uses the Threshold parameter to map the users and the documents of each cluster to one or more indices. The Threshold defines how each difference is treated.

5.1.1 Clusterer

Figure 5.1 illustrates the Clusterer operation. The Clusterer operates in two steps: the document-grouping and the clustering step. It receives pairs of document IDs and ACLs, which are inserted in a hash table creating the Document Families. Then, it groups the Document Families into clusters and gives them as input to the Mapper.

Document-Grouping Step. Before proceeding to the clustering process, we create the Document Families. As each Document Family contains a set of documents with identical ACLs, is described by: 1) a set of the unique identification number (ID) of each document that belongs to the specified Document Family; and 2) a binary ACL bitmap.

In order to build the Document Families, we use a chained hash table. The hash table entries consist of three fields, each of which is used to store: 1) a set of document IDs, 2) the documents' ACL, which consists of a set of user IDs, and 3) the pointer to the next entry. The ACL of a document is used as key to the hash function. Every time a new document is encountered, we check whether a document with an identical ACL has already been inserted in the hash table or not. If yes, then we simply add the ID of the new document in the document IDs set of the corresponding entry. Otherwise the ID and its ACL are inserted in a new entry according to the hash function. Therefore, each hash

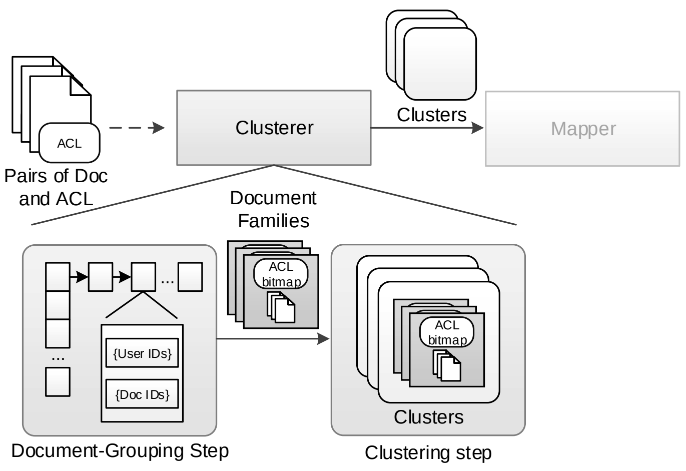


Figure 5.1: We illustrate how the Clusterer operates. It receives pairs of document IDs and ACLs and then creates the Document Families using a hash table (document-grouping step). Then, it clusters the Document Families (clustering step) and the created clusters are given as input to the Mapper.

table entry contains all document IDs with the same ACL.

Once all documents with identical ACLs are in the same entry, a further step is needed to obtain the Document Families: the construction of each ACL bitmap. Each ACL bitmap is constructed by using the ACL of the corresponding hash table entry. Finally, each set of document IDs along with their ACL bitmap form a Document Family.

Clustering Step. As the clustering process proceeds, it forms clusters of Document Families. Each cluster is represented as a vector of Document Family IDs and is stored in the *Cluster Vector*. Thus, at the end of the clustering process, the Cluster Vector contains all the formatted clusters.

Finally, after the clustering process finishes, the Cluster Vector is given as input to the Mapper.

5.1.2 Mapper

The Mapper receives the created clusters from the Clusterer and maps their Document Families to indices (Figure 5.2). For each index the Mapper creates a description, which is finally stored in a file. This file is then used by the Indexer for the indexing process.

During the mapping phase, the ACLs of the Document Families included in each cluster are split in ACL parts. The basic ACL part consists of the intra cluster intersections,

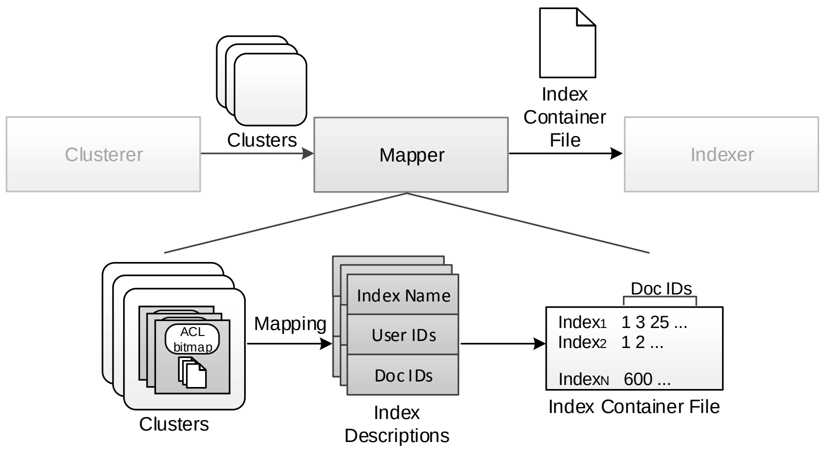


Figure 5.2: We depict how the Mapper operates. It receives the clusters from the Clusterer and then maps the Document Families within each cluster to one or more indices. Also, the Mapper creates a description for each index and stores it in the index container file. This file is then given as input to the Indexer.

while the other parts arise from each difference. As the ACLs are represented as sets of user IDs, both the intersections and the differences are found by using the corresponding functions provided by the STL library.

Once the intersection of each cluster is found, the Mapper maps the users that belong to the intersection along with all documents in the cluster to a single intersection index. On the contrary, each difference along with its corresponding documents is mapped to a single difference index or many private indices depending on the Threshold value. All index descriptions are stored in an *Index Vector* and each of them is described by: 1) the index name; 2) the set of users that have access to it; and 3) the document IDs that are included in it.

Finally, two files are created that store information about the indices. The first file, denoted as *access control file*, contains one entry for each index. This entry includes the set of the users that have access to the specified index. It is used before a user starts submitting queries in order to acquire the list of indices in which he can search. The second file, denoted as *index container file*, also contains one entry for each index, and each entry includes the document IDs that are going to be indexed by that index.

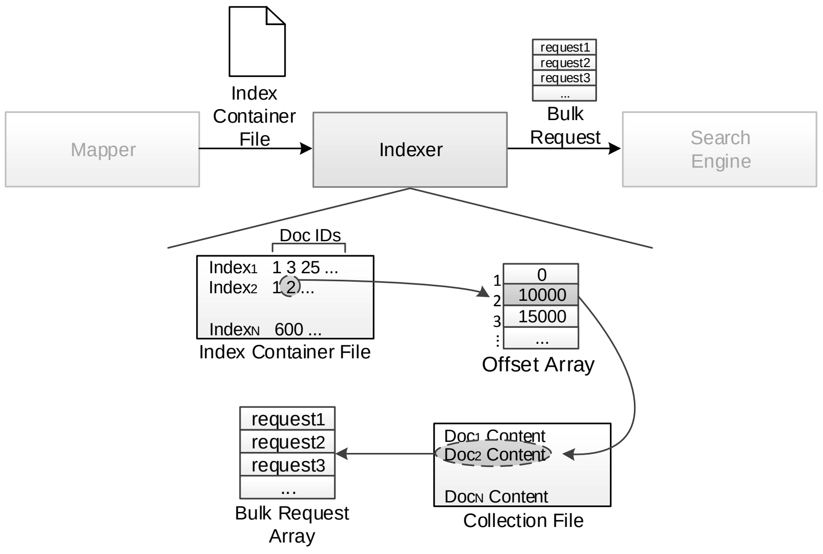


Figure 5.3: We depict how the Indexer operates. It parses each line of the index container file and gets the content of each document from the collection file by using the offset array. For each document, the Indexer creates one index request and stores it in the bulk request array. When a predefined number of request are accumulated, the Indexer sends a bulk index request to the search engine.

5.2 Indexer

The Indexer leverages the information stored in the index container file and creates the indices by indexing its corresponding documents.

Figure 5.3 depicts how the Indexer combines the information generated by the Mapper with the real documents, each stored in a single line of the *collection file*. The Indexer parses each line of the index container file and obtains an index name along with its corresponding document IDs. Each document ID denotes the line at which each real document is stored in the collection file. Then, for each document ID, the Indexer gets the corresponding real document using an *offset array*.

The offset array contains, for each document ID, the offset at which the corresponding real document begins. Consequently, the Indexer creates an index request that includes the index name along with the document content and stores it in the *bulk request array*.

Every time either this array contains $MaxBulk$ requests (MaxBulk = 500) or there are no more documents left for a specified index, the Indexer sends a bulk request containing these documents to the search engine. Then, the Indexer processes the next line of the

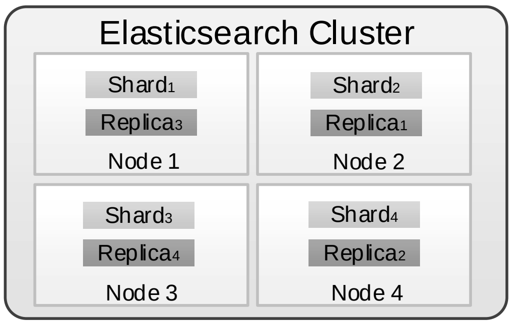


Figure 5.4: We depict an example of a four node Elasticsearch cluster. One index with four primary shards and one replica per shard is stored across the multiple nodes.

index container file. Once all index lines have been processed, the indexing process finishes and the search engine is ready to handle incoming queries.

5.3 Search Engine

The search engine we use is the Elasticsearch [16]. Elasticsearch is a distributed, free/open source search server written in java and based on the Apache Lucene library [14]. It runs on a single search server or on multiple cooperating servers when dealing with large data sets or needing fault tolerance. These multiple servers are called *cluster* and each of them is called *node*.

The nodes are used to store the indexes and serve the incoming queries. When the indexes contain a large amount of documents, each index may be split into smaller individual parts called *shards*. Each shard is a separate index and can be placed on a different node in order to achieve better performance. When a query is addressed to an index that is built from multiple shards, Elasticsearch sends the query to each relevant shard and merges the individual results.

In order to achieve higher query performance and availability, each shard (*primary shard*) may have one or more replicas. The primary shard is the place where the index update operations are initially applied. The primary shard, as well as the replicas, are used to answer the queries. When the primary shard is lost, the Elasticsearch cluster chooses a replica to be the new primary shard.

Figure 5.4 depicts an example of a four node Elasticsearch cluster. One index with four shards and one replica per shard is stored in the cluster. Primary shards 1 to 4 are each stored in Nodes 1 to 4 respectively. Each replica is stored in a different node from the one that holds its corresponding primary shard.

5.4 Discussion

The implementation details refer to document IDs rather than document paths as described in the design description. In a real system, documents can be identified by their paths. However, in our case, each document exists through a document ID and it is not mapped to a real document until the indexing phase. This is a conscious choice considering that we do not use documents that exist on a real system but we build our own synthetic system (including users, groups and document ACL) based on observations of a real system. Moreover, this option gives us the ability to capture the behavior of our solution under different scenarios.

5.5 Summary

The implementation of our solution includes the Indexer and the two components of the Planner: the Clusterer and the Mapper. In order to build the Document Families before the clustering process, we use a chained hash table whose entries contain: a set of document IDs and the set of users that have access to them. We use vectors to store the Document Families of the created clusters. Also, the Mapper uses a vector to store the mapped indexes. Each index contains its name, the IDs of the users that have access to the index, and the IDs of the documents to be indexed. The Indexer then leverages the information about each index, gets the content of the real documents, creates index requests, and sends them in bulk to the search engine.

Chapter 6

Experimental Results

In this chapter we present an extensive study of the behavior of our solution under different scenarios. Initially, we describe the methodology of our evaluation and the experimental setup. Next, we focus on the results retrieved by the Planner and a search engine.

6.1 Experimental Methodology

Due to the lack of a real-world document ACL dataset, we implemented an *ACL Generator*. The ACL generator creates a synthetic ACL dataset based on observations from a real one. We verified that the generated data comply with the statistics of the real-world dataset and performed several measurements to study and evaluate the behavior of our solution.

ACL Generator. In order to implement the ACL Generator, we use the observations on the access control usage presented in [38]. The authors collected a snapshot of available documents, along with their access control lists, from a corporation 's Docushare server [20]. The documents were either publicly available or an ACL was specified for them. For the later case, we use the distribution of the sizes of user groups, as well as the distribution of the ACL entries (number of users and groups that are included in each ACL). However, we have no information about which users and groups belong to a specified ACL, or how many documents share a common ACL.

Based on the above observations, the ACL Generator works in two steps:

- The first step refers to the creation of the user groups. We assume a predefined number of users and groups, each of which is represented by an ID. For each group, we pick the number of its members (maximum 50) following the distribution acquired from the above study. Also, we uniformly choose a set of user IDs as members of the specified group.
- The second step refers to the creation of the ACLs and their mapping to document IDs. We assign a number of individual users and groups to each ACL according to the corresponding distribution of the survey. Since users organize and store their documents under directories, documents under the same directory tend to inherit the same ACL [38]. Thus, we map each ACL to a random number of documents (maximum 50) rather than to a single document.

After completing the steps above, each document ID has been associated with an ACL. Although the entries of each Generated ACL are both individual user and group IDs, we end up with each ACL being a set of user IDs because we replace each group ID with its members' IDs. Moreover, we can change the group size distribution in order to study the behavior of our solution on different ACL datasets beyond those that are found in the above study.

ACL Generator Configuration. The basic parameters of the ACL Generator are the number of users, the number of groups, and the number of documents in the system

Parameter	Default Value
Users	200
Groups	131
Documents	50000

Table 6.1: Basic configuration parameters of the ACL Generator. We create 200 user IDs, 131 group IDs, and 50000 document IDs.

(Table 6.1). Moreover, the documents are separated in three categories: a) private, when only accessed by their owner; b) shared, when a set of users and groups have access to them; and c) public, when all the users of the system can access them. In particular, 22.7% of the documents are public, 0.56% are private, and 76.74% are shared. These values were selected based on the observations from the previous study.

Examined Cases. We use three different ACL Generator cases to study the behavior of our solution. In each case, we keep the basic configuration parameters as shown in Table 6.1 and use different distributions for the group sizes. In particular, we examine: a) the *Doc Server case*, where we use the distribution of the survey; b) the *Zipfian case*, where we use the zipfian distribution with a parameter set to 0.7, 1.7, and 2.2; and c) the *Uniform case*, where the uniform distribution is used.

Experiment Parameters. First, we study the results retrieved by the Planner. The MinObjs parameter of the clustering algorithm is fixed at 1 during all experiments, while Similarity and Threshold parameters are tunable. Therefore, we examine our solution for varying values of Similarity and Threshold in order to capture their effect on performance. Then, we study the behavior of our solution in a search engine.

6.2 Experimental Setup

For the overall evaluation of our solution we use a search engine, three nodes of a cluster of servers, and a standard collection of documents.

Search engine. Our search engine consists of a two-node Elasticsearch (ES) cluster. Moreover, we used a separate node to issue the search queries to the ES servers using up to 8 ES clients. One node is enough to accommodate 8 clients because each client just issues queries sequentially and waits for each query response before sending the next query.

Node Configuration. The experiments are conducted on three nodes of a cluster running Debian GNU-Linux $v6.0$ squeeze, with the Linux kernel 2.6.32. Two of them are used as ES servers; Each server is equipped with two quad-core 2.33 GHz Intel Xeon $E5345$ processors, 4 GB RAM, an active gigabit Ethernet port, and two 7200 RPM SATA disks (one 500 GB and the other 1 TB). In each ES server node, the 2 GB of RAM are pinned to the ES server, while the other 2 GB are left for the system. Also, each ES server uses the 1 TB disk to store the indices. The third node includes one quad-core 2.33 GHz Intel Xeon $E5345$ processor, 2 GB RAM, an active gigabit Ethernet port, and two 250 GB 7200 RPM SATA disks. This node is initially used to index the documents and then acts as the ES client.

Dataset. In order to evaluate our solution in a real search engine we use the GOV2 dataset from the TREC Terabyte track [19], but we only index a part of it. Specifically, we use the first 50000 documents which are approximately 820 MB. We choose to index 50000 documents because we want our dataset to be in compliance with the observations of the study. Our query set consists of 5000 standard queries [19] and the average number of terms per query is 2.8.

6.3 Planner Results

The main goal of this section is to find the range for Similarity and Threshold parameters that strike a good balance between query performance and update cost. We also analyze and explain the behavior of our solution across different Similarity and Threshold values. In particular, we focus on the following different combinations of Similarity and Threshold values (Table 6.2):

- Similarity at 100% and Threshold set to 0, means that a large number of clusters is created each containing a single Document Family. For each Document Family, we create a single index.

Similarity	**Threshold**	**Effect**
100%	0	each cluster contains one Document Family ⇒ 1 index per Document Family
60%	0	some clusters contain multiple Document Families ⇒ 1 index for non-empty intersections ⇒ difference documents in a single index
60%	> 0	some clusters contain multiple Document Families ⇒ 1 index for non-empty intersections ⇒ difference documents in a single index or private indices
0%	∞	1 cluster includes all Document Families ⇒ difference documents in private indices

Table 6.2: Effect of different combinations of Similarity and Threshold values.

- Similarity at 60% and Threshold set to 0, means that fewer clusters are created and some of them include multiple Document Families. These clusters may have a non-empty intersection. Hence, we create one intersection index for all Document Families in the cluster and one difference index for each Document Family difference.

- Similarity at 60% and non-zero Threshold, means that clusters may have a non-empty intersection as in the previous case. However, the documents of each Document Family difference may be duplicated to one or more indices.

- Similarity at 0% and infinite Threshold, means that a single cluster is created including all Document Families. The documents of each Document Family are indexed by the private indices of each user that belongs to the particular Document Family.

The basic information obtained from the Mapper refers to the average number of indices that a user has access to, the average number of indices in which a document is indexed, and the total number of indices. In order to explain these results, we leverage the information about the number and the size (measured by count of documents) of the clusters.

The results of the Clusterer are affected by the chosen Similarity value, while the results of the Mapper are affected by both the Similarity and the Threshold value. Also,

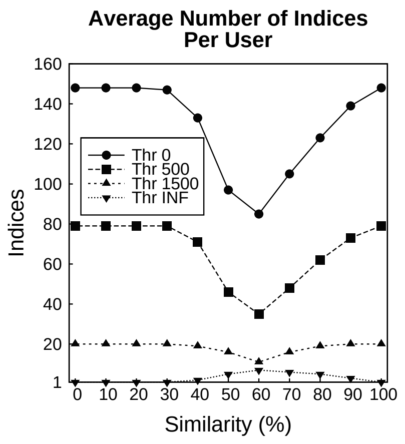


Figure 6.1: Average number of indices that a user needs to search every time he issues a query for varying values of Threshold and Similarity.

three different types of indices are created: a) private, which is only accessed by one user; b) shared, which is accessed by a specific set of users; and c) public, which is accessed by all users.

6.3.1 Indices Per User

First, we examine the average number of indices that a user needs to access when he issues a query. The average number of indices per user affects the query response time and gives an explanation for the results obtained from a search engine in the following section. In general, the lower the number of indices accessed, the better query response time we expect on the search engine.

In Figure 6.1, we depict how this number changes across different values of Threshold and Similarity. The first observation is that for non-zero Threshold the average number of indices accessed per user drastically decreases. In particular, increasing Threshold from 0 to 500 almost halves the average number of indices accessed per user. This is because higher values of Threshold lead to the duplication of more and more documents in the private index of each user. Hence, each user tends to only access his private index as Threshold increases.

Another observation is that the average number of indices accessed per user is minimized for the same Similarity value across several Threshold values (0, 500, and 1500). The question that arises is why we observe this decrease for Similarity values close to 60%, and why the average number of indices accessed per user increases again when moving to lower Similarity values.

To answer the initial question, it is necessary to examine how the Clusterer and the Mapper work under different Similarity values (focusing on 100% and 0%). Figure 6.2 depicts the Clusterer results across different Similarity values: a) the number of created clusters, and b) the average number of documents included in each cluster. As Similarity decreases from 100% to 0%, the Clusterer creates fewer and fewer clusters, while the average number of documents included in each cluster increases. Similarity at 100% means that all documents in a cluster have the same ACL. Hence, each cluster only contains one Document Family and has an empty intersection. On the other hand, a Similarity value close to 0% means that documents with dissimilar ACLs can be part of the same cluster, and hence, only one cluster containing all the documents is created. Thus, with a high probability, this cluster has an empty intersection.

The absence of intra-cluster intersection for 0% and 100% Similarity leads the Mapper to treat each Document Family as a difference. This means that we index the documents of each Document Family based on the Threshold value. Therefore, for Threshold set to 0, we create a single index for the documents of each Document Family in a cluster. For infinite Threshold, the same documents of each Document Family in a cluster are indexed by the private index of each user in the ACL of the Document Family. For intermediate Threshold values, the Mapper indexes the same documents of each Document Family either in a single index, or in the private indices of each user in the ACL of the Document Family. This depends on whether the number of documents in a Document Family multiplied with the number of users in its ACL is higher than the Threshold value or not.

Despite the different clustering output, the Mapper gives the same results for the two extreme Similarity values (0% and 100%) across different Threshold values. This is because each Document Family is treated as difference for both Similarity values and produces the same index mapping results. However, both the Clusterer and the Mapper results differ for Similarity values between 0% and 100% leading to the reduction of the average number of indices per user.

Consequently, in order to answer the first part of our initial question, which is why we observe the average decrease for Similarity close to 60%, we compare the content of the clusters between the 100% and 60% Similarity. When Similarity is 100% and Threshold

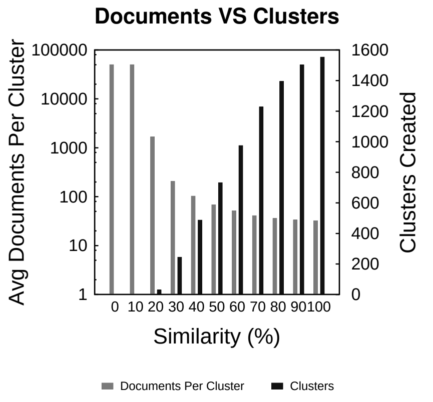


Figure 6.2: We examine the total number of clusters and the number of documents per cluster across different Similarity values. The total number of clusters decreases and the total number of documents per cluster increases as the Similarity drops from 100% to 0%. The results are the same across different Threshold values as Threshold only affects the partitioning and not the clustering phase.

is 0, the number of clusters is large and each cluster contains a single Document Family. As we create one index for each Document Family, the number of indices is equal to the number of clusters. For this reason, each user needs to access a large number of indices as the documents are spread in many indices. On the contrary, as Similarity gets values lower than 100%, the Clusterer creates fewer and fewer clusters, while more and more Document Families (and documents) are included in each cluster (Figure 6.2). Also, many clusters contain Document Families whose ACLs have a non-empty intersection. As the Mapper creates a single index for each intersection, all the documents of a cluster are indexed by that index. Hence, the users that belong to intersections only access one index in order to find the documents included in a cluster and the average number of indices per user decreases. Also, the intersections are responsible for the decrease of the average number of indices per user at 60% Similarity for Threshold at 500 and 1500.

However, the average decrease at 60% Similarity does not hold for Threshold values close to infinity. For these values, the average number of indices per user is higher for 60% Similarity than for 100%. An 100% Similarity value combined with an infinite Threshold means that each each cluster contains a single Document Family. Due to the infinite Threshold value, each document is duplicated to the private index of each user in the ACL of the Document Family. Therefore, the total number of indices is limited to the number of the users in the system and each user only accesses his private index (Figure 6.5(d)). On the contrary, when the Similarity value is 60%, some of the clusters have a non-empty intersection. Moreover, each document of each difference is indexed by the private index of each user in that difference. Hence, the total number of indices is the sum of the private indices plus the indices that are created due to the intersections, and each user accesses not only his private index but also some of the intersection indices.

The second part of our question, which is why the average number of indices per user increases for Similarity values lower than 60%, is answered if we understand how the Similarity value affects the homogeneity of the ACLs within each cluster. Low Similarity values mean that the likelihood of a non-empty intersection in a cluster is small. Hence, the users cannot benefit from the indices of intersections and the average number of indices per user increases again.

In general, as we decrease Similarity from 100% to 60%, the average number of indices per user decreases for Threshold in range 0-1500. This holds because documents of different Document Families are indexed by a single index due to the intra-cluster intersections. Hence, the users that belong to the intersections access fewer indices than in the case where Similarity is 100% and one index is created for the documents of each Document Family. Moreover, when the Threshold value increases, the users that belong to the differences or to Document Families that are treated as differences access fewer indices because many documents are indexed by the private index of each user. Thus, for Threshold in range 0-1500, the average number of indices per user decreases as Similarity drops from 100% to 60%.

6.3.2 Indices Per Document

In Figure 6.3, we depict how the various Threshold values affect the average number of indices in which a document is indexed (document duplication). We observe that when the Threshold is set to 0, the document duplication is limited as each document is indexed by at most two indices. For a cluster with a non-empty intersection, each document is indexed by the intersection index and in the index of the difference. For a cluster with an empty intersection, each document of the cluster is indexed by a single index. On the contrary, for infinite Threshold value, the document duplication significantly increases as

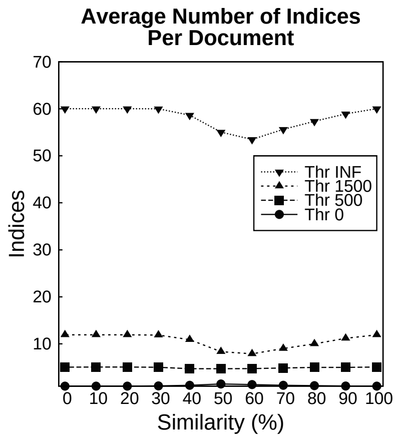


Figure 6.3: Average number of indices that each document is indexed for varying values of Threshold and Similarity.

the documents of each Document Family are indexed by the private index of each user in its corresponding ACL. However, for low Threshold values, the average number of indices containing a document does not exceed the 15 indices.

Another observation is that the document duplication decreases as Similarity decreases from 100% to 60%, where it gets its minimum value. Then, the average starts increasing again for Similarity values lower than 60%. For high Threshold values (1500 and infinite) and Similarity value at 60%, the majority of the clusters have a non-empty intersection. The number of users in each intersection is large and the number of users in each difference is small. Thus, each document appears in a small number of private indices, each corresponding to a user of the difference. This means that the number of duplicates is limited. On the contrary, the number of indices in which each document is indexed is higher when the Similarity value is different from 60%. In these cases, either all or the most of the documents of a cluster are indexed by private indices. Hence, the document duplication decreases for Similarity values close to 60%.

For low Threshold values (0 and 500), the results are different from those described above. More precisely, the average number of indices per document slightly increases as Similarity drops from 100 down to 60%. In case where Similarity is 100% and Threshold is set to 0, the documents of each Document Family are indexed by a single index. Thus, we have no duplicates. On the contrary, the number of duplicates raises when Similarity reaches 60%, as each document of each Document Family is indexed by at least two indices. In particular, it is indexed by the index of the intersection and by the index of the difference. In the case where Threshold is set to 500 the average number of indices per document does not significantly change across different Similarity values. Nevertheless, the increase of the duplicate documents is insignificant for these low Threshold values.

In general, the average number of indices per document increases as the Threshold value increases. For infinite Threshold, we get the highest average number of indices per document because each document is indexed by the private index of each user that is allowed to read it. On the contrary, for Threshold set to 0, we get the lowest average number of indices per document because each document is indexed by at most two indices (difference index and/or intersection index). For the other Threshold values, the average number of indices per document increases insignificantly in comparison to the latter case. Also, as Similarity decreases from 100% to 60% the average number of indices per document decreases for Threshold values over 500, while for lower Threshold values it insignificantly increases.

6.3.3 Query/Update Trade-off

So far, we studied the average number of indices per user and per document, each of which is related to the query performance and the update cost respectively.

The previous figures indicate that a 60% Similarity value approximately gives the best results regarding the average number of indices per user and per document. Whenever this is not the case, the increase in the average number of indices per user and the average number of indices per document is insignificant. Hence, we keep the Similarity fixed at 60% and change the Threshold.

Figure 6.4 depicts the trade-off between query performance and update cost across different Threshold values. While the Threshold value grows, we observe that the average number of indices per user decreases and the average number of indices in which each document is indexed increases. For Threshold set to 0, we get the highest average number of indices per user, as each user has access in many indices, and the lowest average number of indices per document, as each document is indexed by a single index. On the contrary, for Threshold set to ∞, we get the lowest average number of indices per user because each user only accesses his private index. However, in this case we get the highest average number of indices per document because each document is indexed by the private index

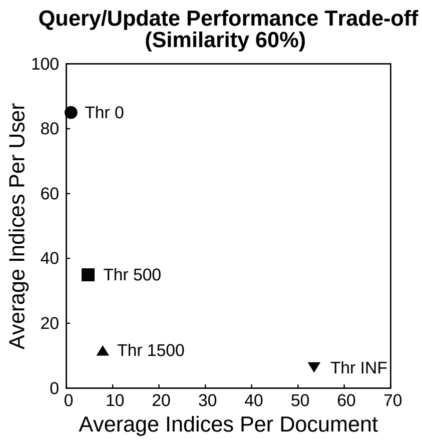


Figure 6.4: We depict the trade-off between the query performance and the update cost for a given Similarity value and across different Threshold values.

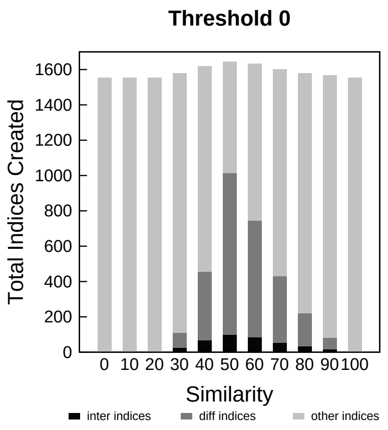
of each user that has access to it. Hence, in the first case (Threshold is 0), we expect to get the worst search performance and the lowest maintenance cost, while in the second case (infinite Threshold), we expect to get the best search performance and the highest maintenance cost.

It is worthwhile to note that there is a point between 1500 and infinite Threshold value where even though we increase the Threshold value, the average number of indices per user and per document does not change. At this point, every shared document (except the public ones) is indexed by the private index of each user that can read it. Hence, each user only accesses his private and the public index. Public documents are only indexed by the private index of each user after the Threshold value becomes higher than the product $users \times number\ of\ public\ documents$.

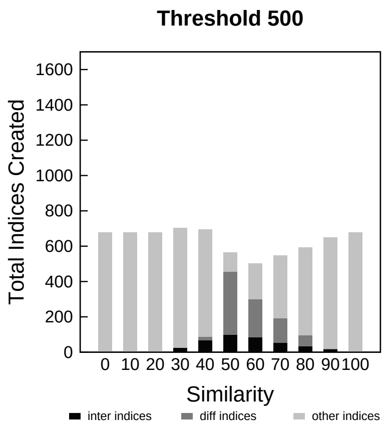
Overall, Threshold values in the range between 500 and 1500 seem to strike a good balance between the number of indices per user and the number of indices per document.

6.3.4 Total Number of Indices

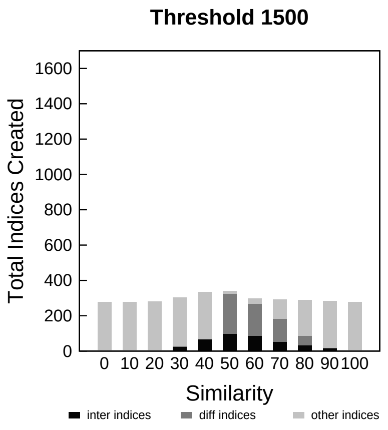
Figure 6.5 illustrates the total number of indices, which is affected by the way we treat each difference, for various Threshold values. As Threshold increases, the total number of indices decreases because the documents of the most differences are indexed by private indices, and hence, fewer and fewer indices are created due to the differences indexed by a separate single index. Moreover, we observe that the number of indices due to the intra-cluster intersections increases for a given Threshold and Similarity values around



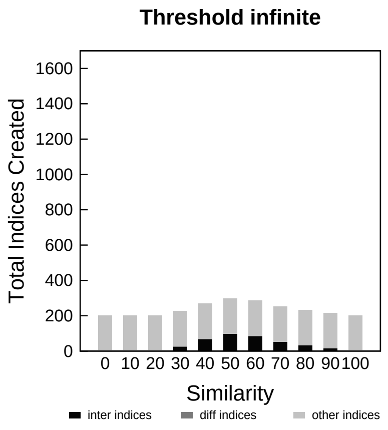
(a)



(b)



(c)



(d)

Figure 6.5: We examine the total number of indices across different Similarity and Threshold values. The total number of indices created decreases as Threshold value increases. The total indices is the sum of the indices due to the intra-cluster ACLs intersections (inter indices), the single indices that are created from each difference of each intra-cluster intersection (diff indices), and the private indices plus the indices which are created from Document Families that are treated as differences (other indices).

60% because many clusters contain Document Families with similar ACLs.

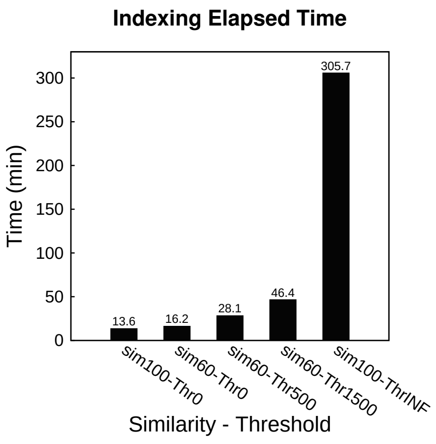


Figure 6.6: We measure the indexing time for different Similarity and Threshold values. As Threshold increases, each document is indexed to more and more indices, and hence, the total indexing time increases.

6.4 Search Engine Results

In this section, we compare the overall search engine performance for different Planner configurations from three different perspectives: a) indexing time; b) disk space overhead; and c) query response time. Our measurements show that different Planner configurations lead the search engine to different levels of query performance and update cost. Therefore, different performance needs are met by using the appropriate Similarity and Threshold values.

6.4.1 Indexing Time

Indexing time refers to the total delay to index the documents. Although our document set is composed of a standard number of documents (50000), the total indexing time differs across the various Planner configurations. Different Planner configurations indicate that some documents are indexed by more than one index, depending on the Similarity and the Threshold value. Thus, the more documents indexed multiple times, the more time is needed to complete the indexing process.

In Figure 6.6, we depict how the various Planner configurations affect indexing time. As we move towards higher Threshold values, the indexing time increases. However, for Threshold set to 0, the indexing time is comparable between 100% and 60% Similarity due to the limited document duplication across different indices (13.36 min and 16.2 min

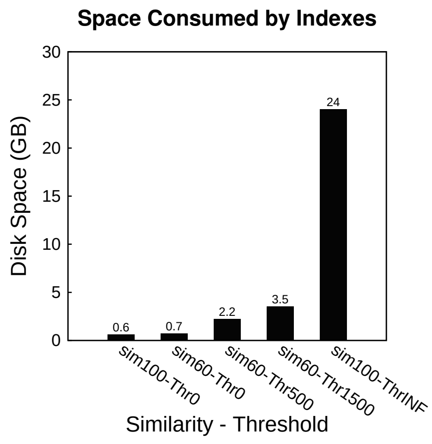


Figure 6.7: We compare the disk space overhead across different Similarity and Threshold values.

respectively). On the contrary, for infinite Threshold, we observe a remarkable increase in the indexing time at 305.75 min. In this case, each document is indexed by the private index of each user that has access to it, and hence, the indexing time significantly increases due to the large number of duplicate documents.

Consequently, the value of Threshold is an important factor that affects the indexing time: higher values mean more duplicates across multiple indices, and hence higher indexing time. However, the Threshold value not only affects the indexing time but the disk space overhead as well.

6.4.2 Disk Space Overhead

In general, it is important to keep the disk space consumption as low as possible. The disk space consumption is related to the number of indices in which each document is indexed, and hence, it grows as the Threshold value increases.

In Figure 6.7, we present the disk space consumption across different Planner configurations. As expected, for higher Threshold values the disk consumption grows, as the same document is indexed by more than one index. The infinite Threshold value leads to the highest disk space consumption, as each document is indexed by the private index of each user that can read it. However, setting the Threshold to a non-infinite value does not considerably affect the disk space consumption in comparison to a value of 0. In particular, when the Threshold is set to 0, the disk space consumption is comparable between the cases where Similarity is 100% and 60%. It is close to the size of the original size of documents, which is 820 MB. When Threshold is 500 and 1500, the disk space consumption grows by a factor of 2.5 and 4.2 respectively. Finally, for infinite Threshold, the disk space consumption increases by a factor of 29.2.

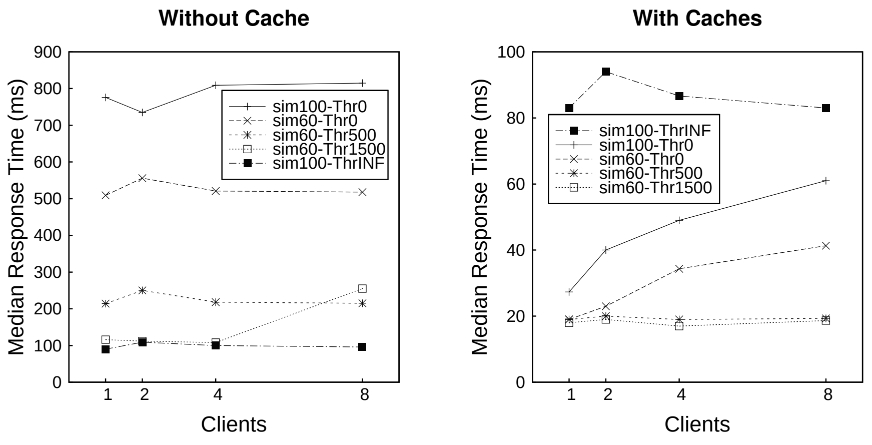
The disk space consumption can also affect query performance, as the size of the indices is closely related to the resident set during query execution. For instance, if the total size of indices is small enough to fit in memory, then the search engine serves queries without performing costly disk accesses. On the contrary, if the total size of indices is larger than available memory, then only a part of it is kept in memory. Hence, the search engine will access the disk with high probability in order to retrieve all the indices that are needed to serve a query.

6.4.3 Search Performance

After examining the indexing time and the disk space consumption, we investigate the search performance across different Planner configurations and number of clients. The search performance is evaluated by measuring the query latency at the search engine, from the time it receives a query to the time it sends the answer to the client.

Initially, we present the median of query response times. We prefer the median and not the average because the median more accurately reflects the most representative value of a set of observations (query response times in our case). In general, the average is computed by adding all the observations and dividing by the number of the observations. On the contrary, the median is computed by arranging all the observations from lowest to highest value and picking the middle one. This means that the median is the value for which 50% of the observation are higher, and 50% smaller than this value. Hence, in cases where the set of observations contains an extreme value that differs greatly from the other values, the median is a better indicator of the most representative value of the set of observations.

When evaluating the search performance, we also present the $90th$ percentile of query response times, which reflects the value for which 90% of the observation are smaller and 10% are higher than this value. Moreover, we present both the median and the $90th$ percentile response times across different number of clients for two different cases. In the first case, ES servers use the cache when serving the incoming queries, while in the second case, they do not use the cache.



(a) (b)

Figure 6.8: We depict the median query response times across different Planner configurations and number of clients.

The rationale of this strategy becomes clear by examining the size of the working set (Figure 6.7). In most Planner configurations, the largest part of the working set fits in memory. Thus, we experimented with both cases (with and without cache) in order to give a clearer view of the search performance. In addition, when the indexing process finishes and before the search engine starts serving queries, we optimize the indices in such a way that each index is stored in a single file on disk (Lucene uses Logarithmic-Merge [25] and a single index may be more than one file on the disk).

The first observation from the experiments is that the median query response time, across different Planner configuration parameters, is as one might expected: the query response time increases in the cases where a large number of indices is accessed per user, and hence per query (Figure 6.8(a)). For instance, for 100% Similarity and infinite Threshold, we get the lowest query response time because only one index is accessed per user. According to Planner results, as Threshold decreases each user accesses more indices. Thus, we observe that query response time increases for low Threshold values.

Not using the cache implies that a query is served from the disk. After the execution of each query, we flush the cache. Hence, the query performance degrades due to the large number of disk seeks that is needed in order to fetch the corresponding indices. For Threshold set to 0 and 100% Similarity, the number of indices per user is high enough to substantially increase the query response time over 700 ms. For higher Threshold values and 60% Similarity, the query response time is reduced because the number of indices per user decreases. In particular, the query response time is about 200 ms for 60% Similarity and Threshold 500. However, the best search performance is achieved for infinite Threshold as each user needs only to access his private index.

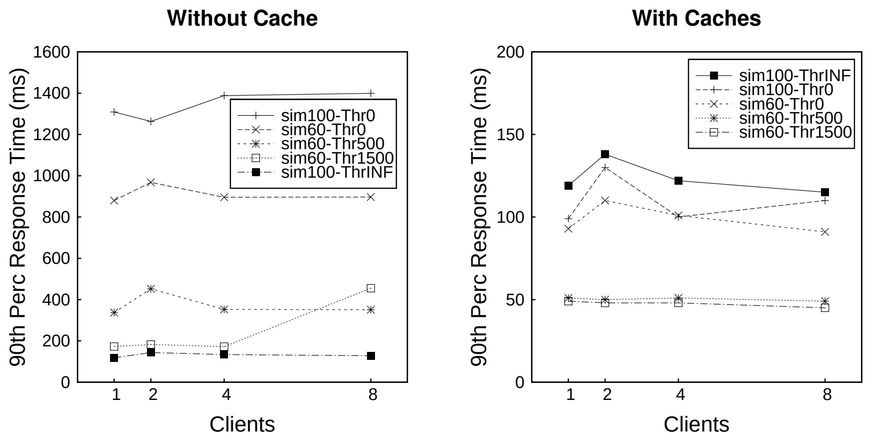
On the other side, using the cache implies that there is a possibility to serve a query from the cache rather than the disk. This is the case where the indices required to serve the query are already in memory because the search engine used them to serve a previous query. However, this possibility depends on the size of the working set. In Figure 6.8(b), we present the corresponding response times in the case we use the cache. As expected, the search engine performs better than the case without the cache. In particular, the response time decreases from hundreds to tens of milliseconds. However, for non infinite Threshold, the trend between different Planner configuration remains almost the same as in the case without the cache.

Contrary to what one might expect, we observe that the median response time for infinite Threshold value is higher than that of lower Threshold values when we use the cache. A reasonable explanation is that the working set does not fit in memory and the search engine satisfies the majority of the queries from the disk. Even though the number of indices is limited to the number of users and each query involves a single index, the working set size is significantly higher than that of lower Threshold values. Hence, the query response times are similar to those without the cache (Figure 6.8(a), 6.8(b)).

Figure 6.8 also visualizes the sensitivity of various number of clients to the median query response time. The median response time is comparable across different number of clients (1 to 8). When we use the cache, the median response time ranges between 18 ms and 82 ms and without cache it ranges between 95 ms and 800 ms.

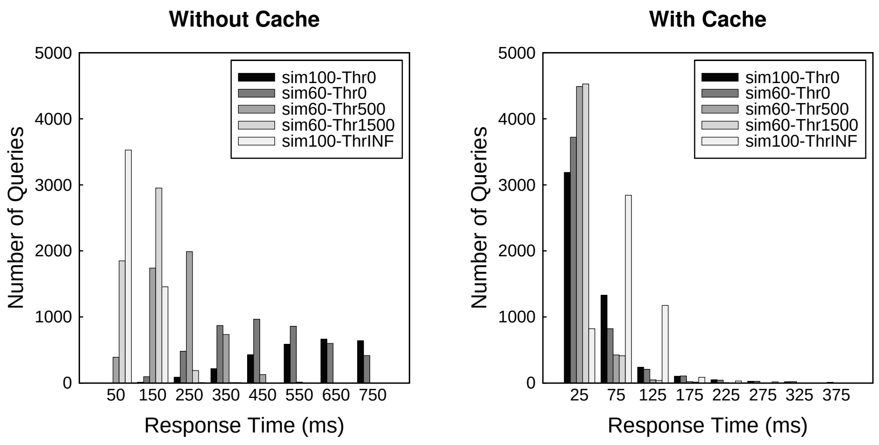
The $90th$ percentile of query response times are also comparable across different number of clients (Figure 6.9(a), 6.9(b)). In particular, when we activate cache, the $90th$ percentile ranges between 50 ms and 120 ms while without the cache it ranges between 100 ms and 1400 ms.

Figure 6.10(a) illustrates the histogram of the query response times without the cache. More precisely, we observe that Similarity values set to 60% or 100% combined with Threshold set to 0, the response times for the majority of the queries exceed 300 ms.



(a) (b)

Figure 6.9: We depict the 90th percentile query response times across different Planner configurations and number of clients.

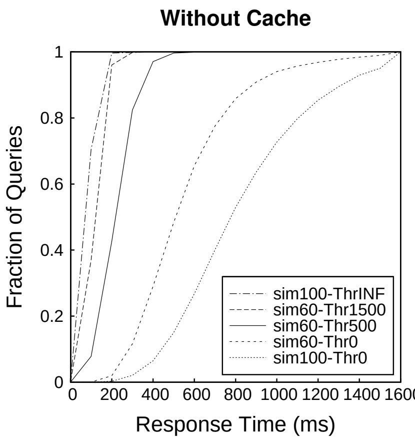


(a) (b)

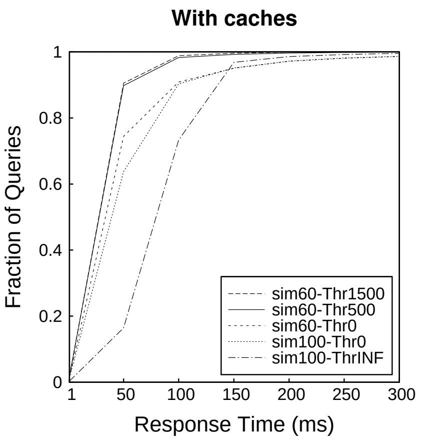
Figure 6.10: Histogram of query response times across different Planner configurations.

However, for higher Threshold values, the response time for the majority of the queries ranges between 50 ms and 300 ms. When we use the cache, the results show that most queries have a response time around 25 ms (Figure 6.10(b)). However, for infinite Threshold, the response time for the majority of the queries ranges between 75 ms and 125 ms.

An alternative view of the above information is depicted in Figure 6.11. A cumulative distribution function of query response times gives an estimation of the fraction of queries



(a)



(b)

Figure 6.11: CDF of query response times.

executed in less than a certain amount of time. If we use the cache, the 89% of the queries take less than 50 ms when the Similarity and the Threshold value is set to 60% and 500 (or 1500) respectively (Figure 6.11(b)). For Threshold set to 0 and 60% Similarity, we observe that 74% of queries take less than 50 ms. For Threshold set to 0 and 100% Similarity, the 63% of queries take less than 50 ms. On the contrary, for infinite Threshold, this fraction hardly reaches the 16% as most of the queries are satisfied from the disk rather than the cache.

Figure 6.11(a), illustrates the fraction of queries executed in less than a certain amount of time without the cache. The fraction of the queries that take less than 200 ms is 42.6% when the Threshold is 500 and the Similarity is 60%. For higher Threshold values, the 96% of queries is completed in less than 200 ms, while for Threshold set to 0 combined with 60% or 100% Similarity, the fraction of queries that take less than 200 ms is barely 1.96% and 0.3% respectively. These queries take longer because the average number of indices per user is high, and hence more indices are fetched from the disk.

Finally, we examine the search engine throughput for different Planner configurations. Rather than measuring throughput for both cases (with and without cache), we focus on the case with cache. We assume that in general, the search engine makes extensive use of the cache and uses the disk when needed.

Figure 6.12 shows the effect on throughput of various Similarity and Threshold values

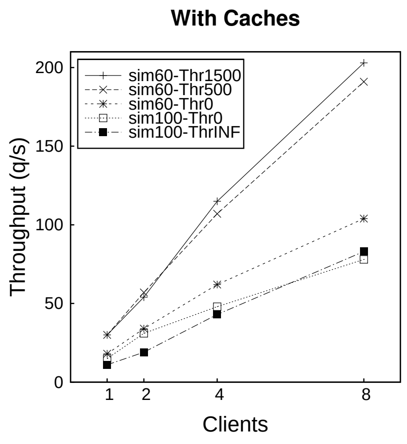


Figure 6.12: We compare the search throughput across different Planner configurations and number of clients.

across different number of clients. In each case, we observe that increasing the number of clients from 1 to 8 leads to higher throughput. For infinite Threshold, the sequential throughput is 11 q/s but it increases by a factor of 7.5 for up to 8 clients. When Threshold is either 500 or 1500, the throughput is 30 q/s and increases by about a factor of 6, as we increase the number of clients from 1 to 8. When the Threshold is set to 0 and Similarity is 100%, the throughput is 15 q/s and increases by about a factor of 5.5. At last, when the Threshold is set to 0 and Similarity is 60%, the throughput is 18 q/s and increases by about a factor of 5.7.

In general, the throughput increases linearly as the number of clients increases. However, the highest throughput is achieved for Threshold values around 500 and 1500 and the lowest for infinite Threshold.

6.5 Exploring Different ACL Synthetic Datasets

The ACL Generator uses the observations on access control usage presented in a study to create the synthetic ACLs. Given a number of users, groups and document IDs, it creates user groups by picking the number of their members (maximum 50) following the distribution acquired from the study (Doc Server distribution). Then, the Generator uniformly chooses a set of user IDs as members of a specified group. Finally, it assigns a number of individual user and group IDs to each ACL according to the corresponding

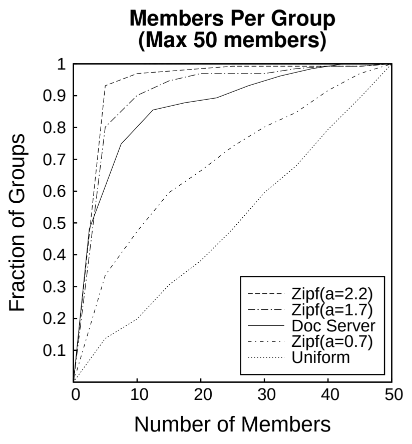


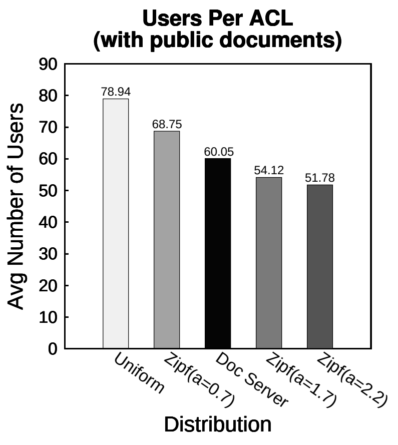
Figure 6.13: We depict the distribution of members per group when using different probalitity distributions for the size of each group. When using the Uniform distribution, the groups tend to have many members. For the Zipfian distribution, only a small fraction of the groups has a large number of members, and this fraction decreases as the value of parameter a increases.

distribution of the survey.

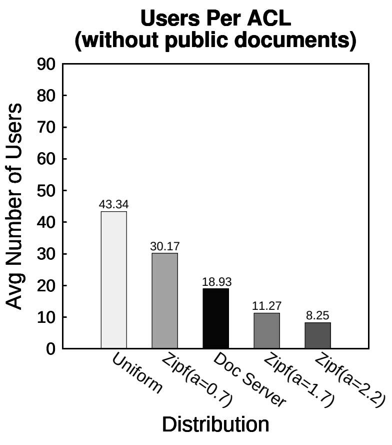
To experiment with different synthetic ACL datasets, we repeat the same procedure, but this time we use different probability distributions for the size of each group. In particular, we use the Zipfian distribution for different values of parameter a (0.7, 1.7, and 2.2) as well as the Uniform distribution. The maximum size of a group is 50 members, and the basic ACL Generator parameters are 200 users, 131 groups, and 50000 documents. Both the maximum size of a group and the basic ACL Generator parameters are the same during the experiments with different probability distributions.

Group Membership and ACLs. Figure 6.13 depicts the distribution of members per group across the usage of different probability distributions for the size of each group. When using the Uniform distribution, groups contain more members than in the case we use the Zipfian or the distribution of the study (Doc Server). For the Zipfian distribution as the value of parameter a increases, groups tend to contain fewer members and only a small fraction of groups contains a large number of users. In comparison to the Doc Server case, the Zipfian distribution with parameter a set to 1.7 and 2.2 decreases group sizes, while group sizes increase with parameter a set to 0.7.

Using different probability distributions for the group sizes, we indirectly change the size of the ACLs (count by number of users included). Figure 6.14 depicts how the average



(a)

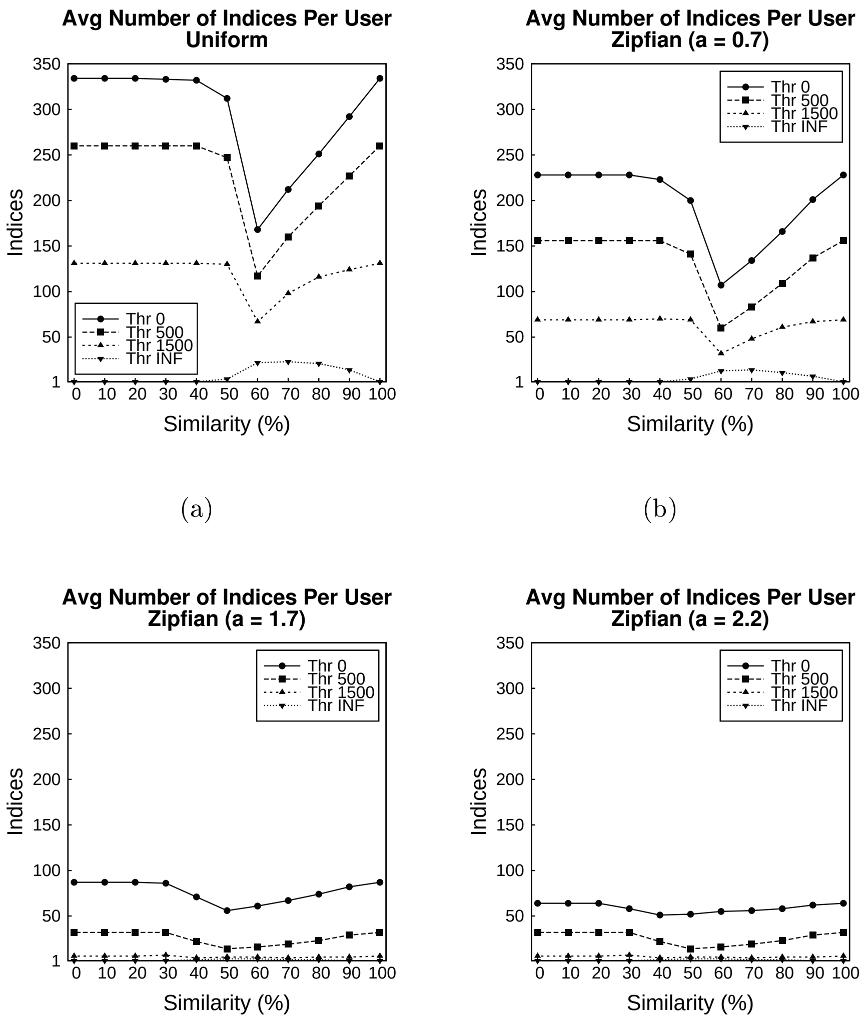


(b)

Figure 6.14: We depict how the size of the ACLs is affected by the different probalitity distributions used for the size of each user group in two different cases. The ACLs of the public documents are not affected by the group sizes. Thus, we present the results for two different cases to give a clearer view of how the ACL sizes change. In the first case, we take into account the ACLs of the public documents (a), while in the second we omit them.

number of users included in ACLs changes across different probability distributions in two different cases. In the first case, we take into account the ACLs of the public documents even though they are not affected by the group sizes (Figure 6.14(a)). The size of the ACLs of the public documents is fixed to 200 users, which is the number of all users. In the second case, we exclude the ACLs of the public documents and only keep those that are affected by the group size in order to give a clearer estimation of how the ACL sizes change (Figure 6.14(b)). The results in both cases follow the same trend: a) Uniform distribution creates ACLs that include many users because in this case groups included in ACLs also have many members, b) when using the Zipfian distribution, the ACLs include fewer users as parameter a increases from 0.7 to 2.2, and c) in comparison to the Doc Server distribution, the Zipfian distribution with parameter a set to 1.7 and 2.2 decreases the average number of users in the ACLs, while the Zipfian distribution with parameter a set to 0.7 increases the average.

Average Number of Indices Per User. As already explained, the different proba-



(a) (b)

(c) (d)

Figure 6.15: We depict how the average number of indices per user is affected by the different probalitity distributions used for the group sizes. The average number of indices per user is higher for the Uniform distribution. As the value of parameter a of the Zipfian distribution increases, the average number of indices per user decreases.

bility distributions affect the group sizes. Group sizes in their turn affect the probability for a user to belong in multiple ACLs, and hence the average number of indices that a user needs to access when he issues a query.

In Figure 6.15, we examine the average number of indices per user across different values of Similarity and Threshold when using different probability distributions for the group sizes. The first observation is that the results follow the same trend as in the case we use the distribution of the survey for the group sizes (subsection 6.3.1):

- Similarity 0% and 100% give the same results across different Threshold values despite the different clustering output. Similarity 100% creates many clusters each of which contains a single Document Family, while Similarity 0% creates a single cluster which contains all Document Families. However, in both cases each Document Family is treated with the same way (as difference) and the Mapper produces the same results.

- The average number of indices per user is minimized for a Similarity value lower than 100% across several Threshold values (0, 500, and 1500). For Similarity values lower than 100%, clusters may contain Document Families whose ACLs have a non-empty intersection. As the Mapper creates a single index for each intersection, all the documents of a cluster are indexed by that index. Hence, the users that belong to intersections only access one index in order to find the documents included in a cluster and the average number of indices per user decreases.

- As Threshold value increases, the average number of indices per user decreases because many documents are indexed by the private index of each user. Hence the users that belong to the differences or to Document Families that are treated as differences access their private index and fewer shared indices.

However, the average number of indices per user is higher for the Uniform and the Zipfian distribution with parameter a at 0.7 than for the distribution of the survey. On the contrary, for the Zipfian distribution with parameter a at 1.7 and 2.2, the average number of indices per user is lower than that of the distribution of the survey, which reaches the 148 indices for 100% Similarity and zero Threshold.

We observe that the Uniform distribution achieves the highest average number of indices per user, which reaches the 334 indices for 100% Similarity and zero Threshold, while the Zipfian with a parameter set to 2.2 achieves the lowest, which reaches the 34 indices. When we increase the value of parameter a, the average number of indices per user decreases because the probability of a user to belong in many groups, and hence in many ACLs, is lower . This is because most groups have a small number of members and the members are chosen uniformly.

Another important observation is that the Similarity value which minimizes the average number of indices per user decreases from 60% to 40% as the value of parameter a increases. Moreover, the higher the value of parameter a, the lower the contribution of the Similarity parameter to the decrease of the average number of indices per user, for a given Threshold value.

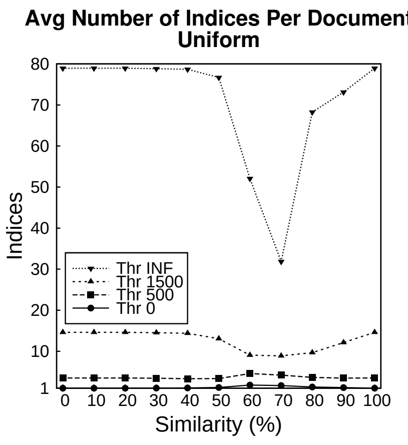
Average Number of Indices Per Document. In Figure 6.16, we examine the average number of indices per document. The results follow the same trend as in the case we use the distribution of the survey for the group sizes (subsection 6.3.2):

- The average number of indices per document increases as the Threshold value increases. For infinite Threshold, we get the highest average number of indices per document because each document is indexed by the private index of each user that he is allowed to read it. On the contrary, for Threshold set to 0, we get the lowest average number of indices per document because each document is indexed by at most two indices (difference index and/or intersection index). For the other Threshold values, the average number of indices per document increases insignificantly in comparison to the latter case.

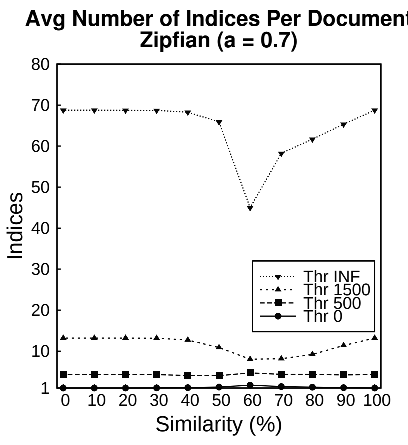
- At the Similarity value which minimizes the average number of indices per user, the average number of indices per document decreases for Threshold values over 500, while for lower Threshold values it insignificantly increases.

However, we observe that the average number of indices per document is higher for the Uniform and the Zipfian distribution with parameter a at 0.7 than for the distribution of the survey. This is because each document can be read by large number of users (as we have large ACLs), and hence for a high Threshold value each document is indexed by a large number of private indices. On the contrary, for the Zipfian distribution with parameter a at 1.7 and 2.2, the average number of indices per document is lower than that of the distribution of the survey, which reaches the 60.04 indices for 100% Similarity and infinite Threshold. In this case, the ACLs are smaller and fewer document duplicates are created.

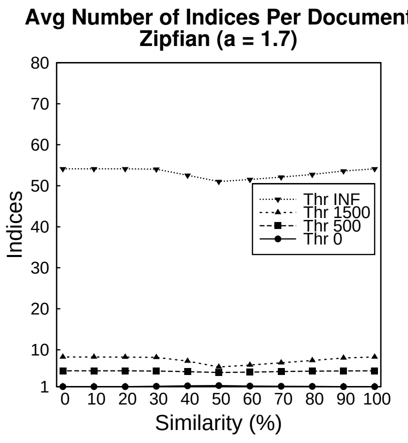
We observe that the Uniform distribution achieves the highest average number of indices per document, which reaches the 78.94 indices for 100% Similarity and infinite Threshold, while the Zipfian with a parameter set to 2.2 achieves the lowest, which reaches



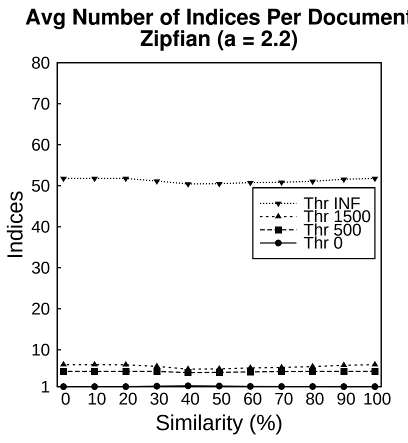
(a)



(b)



(c)



(d)

Figure 6.16: We depict how the average number of indices per document is affected by the different probalitity distributions used for the group sizes. The average number of indices per document is higher for the Uniform distribution. As the value of parameter a of the Zipfian distribution increases, the average number of indices per document decreases.

the 51.78 indices. Also, as the value of parameter a increases, the average number of indices per document decreases for high Threshold values because the ACLs include fewer number of users, and hence each document is indexed by fewer private indices.

Another important observation is that the reduction of the average number of indices per document is more acute for the Uniform and Zipfian distribution with low values of parameter a. This holds for the Similarity value that minimizes the average number of indices per user and infinite Threshold. A reasonable explanation is that at this Similarity value, the differences in each cluster contain few users, and hence the documents corresponding to differences are indexed by few private indices. On the contrary, for Similarity 100% and infinite Threshold, each document is indexed by a large number of private indices as each ACL in the cluster includes many users.

6.6 Summary

In order to evaluate our solution, we implemented an ACL Generator that creates synthetic ACLs taking into account the observations on the access control usage presented in a previous study. The experimental results show the trade-off arising between the query performance and the maintenance cost across different Similarity and Threshold values.

For Threshold set to 0 and 100% Similarity, we get the highest query response time because in this case the number of indices accessed per user is high. However, the indexing time and the disk space consumption are the lowest of all other cases. When Similarity is 60% and Threshold is set to 0 or higher, we get better query response time because the number of indices per user decreases due to the intersection of the ACLs within the clusters. On the contrary, the indexing time and the disk space consumption slightly increase because documents are duplicated across a small number of indices. For infinite Threshold, one might expect to get the lowest query response time, as each user only accesses one index. However, the query response time is higher than in the other cases when we leverage the cache. In this case, the working set is larger than in the other cases and does not fit in memory. Hence, the search engine satisfies the majority of the queries from the disk. Moreover, for infinite Threshold we get the highest indexing time and disk space consumption due to the high number of duplicates across multiple indexes.

To examine more synthetic ACL datasets, we conducted additional measurements using different distributions for the size of user groups in the ACL Generator. The Planner results follow the same trend as in the case we use the distribution from the previous study. However, the average number of indices per user and per document is higher or lower than that of the distribution from the study for the different examined distributions.

Taking into account the presented results, this chapter gives many insights on how to tune such an indexing workflow scheme depending on the performance needs and keep a balance between maintenance cost and query performance.

CHAPTER 7

RELATED WORK

Full-text search has been a topic of great interest for the last few years. However, researchers turned their attention to the privacy issues arising when the full-text search engine operates in a multi-user environment. Therefore, a notable amount of research has been devoted also in this area.

This chapter revisits prior research focused primarily on desktop and enterprise search engines by capturing their benefits as well as their limitations. Furthermore, we present existing research on secure full-text search in social networks as they support multiple users and the privacy issues referred in the previous chapter still hold. Finally, we review approaches that provide secure data storage.

7.1 Desktop and Enterprise Search

Much of the focus of recent research at the concern of full-text search has been on providing privacy, keeping low the maintenance cost and increasing the query performance. While privacy protection is necessary, the performance of a full-text search engine is also very important. However, these are two opposing principles, and it is difficult to achieve both.

7.1.1 Per User Indices

Google 's Desktop Search tool [1] is one of the most popular desktop search engines. It was designed in order to bridge the gap between the increasing amount of stored data and real-time searching on a single-user machine. When used in multi-user environments it creates either one index per user or a system-wide index. In the first case, each index includes all the documents that a particular user is allowed to read. Hence, when a user submits a query, it is addressed to his private index. In the later case, all users' documents are indexed by a system-wide index taking no account of their owner. However, this index needs to be created and accessed only by users with administrative rights, posing severe privacy threats. Therefore, the above limitation makes this case unsuitable for multi-user environments [41].

Soon after Google Desktop Search, other desktop search tools appeared as well, such as Yahoo! Desktop Search [9] and Copernic [8]. Both of them bear a strong resemblance to Google Desktop Search in the way they operate in multi-user environments. More particularly, they integrate access control during indexing time by ensuring that each user has a distinct index. Thus, in each user's index are only indexed documents that he is allowed to read.

Creating one index per user implies that we have a completely safe way of organizing the user's documents into indices. Since each user only searches among documents that he is allowed to read, each query result is restricted to those documents. Hence, users cannot infer the content of documents that they are not allowed to read. In this approach, the query performance is high as only one index has to be searched per query and there is no additional cost to apply any access control.

Despite the privacy protection provided by the above approach, some performance issues arise. The fact that a single document can be readable by multiple users and all documents readable by a user are indexed by his private index leads to document duplication across two or more indices. Document duplication makes the full-text search engine suffer not only from disk consumption, but also from costly maintenance. The number of indices for a document is equal to the number of users that are allowed to read it. This also holds for updates. Whenever a document is updated, the changes are also applied to all indices in which this document has been indexed. Furthermore, this issue becomes more acute as the rate of documents that are readable by many or all users increases.

7.1.2 Shared Index

The need for overcoming the limitations derived from the one index per user approach led to the use of a single system-wide index. In this approach, all documents are indexed by the same index and the access control information is taken into account before handing in the results of a query. Hence, the extra indexing and updating cost are avoided since each document is indexed only once.

Apple's Spotlight [2, 39], the Mac OS indexing and retrieval facility, uses such an approach. It provides full-text search for separate user accounts by extracting and indexing metadata in a single system-wide index, while respecting the ownership of the documents. Whenever a search query arrives, Spotlight computes the list of documents matching the query and then it filters this list. Filtering is performed by checking the document permissions and removing from the result any documents that the user is not allowed to read. A possible drawback of Spotlight is that it returns results in lexicographic order and does not employ any relevance ranking algorithm. This, eliminates privacy threats based on TF/IDF ranking algorithms, but also hampers users to find fast the most relevant matching documents. Indeed, a non ranking approach is not always appropriate, especially when the number of documents is large [6].

A similar approach is implemented by Microsoft [7]. In particular, the search engine identifies users and groups that are granted or denied access to each document by adding authorization information (Access Control List) to each document. Thus, whenever a user submits a query, the list of matching documents is computed and filtered by taking into account the documents' Access Control Lists. Accordingly, the query results are restricted to the documents that a user is allowed to read. In addition, regarding its ranking model, the Microsoft's search engine offers several ranking models. However, the default model is based on a TF/IDF scoring function.

Google supports enterprise search with the Google Search Appliance [12]. Its main purpose is to provide fast, relevant search results. Google Search Appliance creates access credentials provided by the system administrator in order to index users' documents. At query execution time the index is searched and a list of all matching results is retrieved. Prior to returning the final result list to the user it removes the documents that do not comply with the corresponding credentials. The result ranking is based on a TF/IDF style ranking algorithm [30]. However, it can be influenced by some features provided, such as self-learning scorer[1].

While a single system-wide index reduces the disk space consumption and the update cost, it adds an extra cost at query execution in order to satisfy the access control restrictions. In addition, it can pose serious privacy threats; Ranking algorithms, based on TF/IDF scoring function in conjunction with result filtering after computing the relevance score of matching documents, permit to a user to infer information about documents that he is not allowed to read [6]. However, following different approaches is possible to eliminate privacy threats.

7.1.3 Secure Approaches

In an effort to eliminate privacy threats in full-text search engines when operating in a multi-user environment, Büttcher and Clarke [6] designed the Wumpus search engine. Although Wumpus uses a system-wide index, the result ranking is only performed on documents that a user is allowed to read. Whenever a query is received, the posting lists of its terms are computed. Then, the access control restrictions are applied by removing any occurrences of the query term within documents that are not readable by the user who submitted the query. Finally, the ranking step follows based on the final form of the posting lists. Since the ranking step is only applied on documents readable by a user, the user cannot infer any information about documents that he is not allowed to read.

[1]This feature automatically analyzes user behavior and the specific links that users click on for specific queries in order to fine tune relevance and scoring

However, a possible drawback of this approach is that it requires the information (e.g. owner, permissions) about every indexed i-node to be kept in memory. This could become a problem when the indexed documents reach the limit of a few million.

Singh et al. [37] proposed a distributed approach that couples search and access-control into a unified framework, while protecting privacy in multi-user environments. The main idea is to build indices, each of which maintains documents that have exactly the same access control privileges. Towards that direction, they build a graph whose edges reflect the documents that a user or a user group has access to and then they divide documents into independent access-privileges based chunks, which they call access-control barrels (ACB). However, documents readable by a user may be spread in many ACBs. Thus, in a user's subsequent search the results are derived from all the ACBs which contain documents that the user is allowed to read. Unfortunately, there is no upper bound for the number of different ACBs that potentially can be created and for the number of ACBs that a user has access to. Moreover, it is shown that it is impossible to reduce the number of ACBs without either duplicating documents in barrels or violating the security restrictions. Hence, in order to reduce the number of ACBs, all ACBs in which few users have access are removed and the documents contained in them are moved to each user's private ACB.

LI et al. [15] present a different approach to protect privacy in search engines when operating in multi-user environments. The core idea is to assign multiple IDF values (one for each user) to each term. In addition, these term IDF values are computed by only taking into account documents that the specified user is allowed to read. In order to do this, a personalized index is build for each user in an early stage and then these indices are merged into a global index. Thus, when a user submits a query, the relevance scores of the matching documents are calculated by the user's personalized information of the previous step. Hence, a user cannot infer any information about documents that he is not allowed to read. However, while this approach protects privacy, nothing is said about the index maintenance cost and how efficiently the privacy protection lines with incoming document updates.

7.2 Metadata Search

Beyond the research on full-text search, some remarkable approaches that deal with metadata search have been presented. Leung et al. [27] designed Spyglass, a metadata search engine which is focused on how to exploit metadata properties in order to improve search performance and scalability in large-scale storage systems. This is possible through hierarchical partitioning which partitions the file system based on the namespace. Each partition corresponds to a separate index. Hence, each index contains documents that belong to a unique partition of the namespace. In addition, each partition is stored sequentially on disk; Bloom Filters [5] are used to restrict the search only to partitions that may contain documents relative to a query. However, this solution only refers to metadata search and without reference to privacy protection in multi-user environments.

Parker-Wood et al. [33] introduced a security aware index partitioning algorithm and a series of metrics which can be used to evaluate the expected performance of different partitioning algorithms. Security-aware partitioning partitions the file system according to group and user security permissions while walking over it in a breadth first search. The access permissions of a document or directory are determined by examining all permissions in the directories above. If the permissions on the current document or directory are more restrictive than that of the current partition, then a new partition is created. Then, all documents in each partition are accessed by the same set of users, and each user can only search in partitions that include documents that he is allowed to read. Even though security is ensured, this solution generates many small partitions. However, one possible way to reduce the number of created partitions, is to merge those that are accessed by the same set of users.

7.3 Full-Text Search in Social Networks

Social networks, such as Facebook[2], Twitter[3], and Google+[4], are popular online communities that provide interaction, communication and information sharing between users by

[2]https://www.facebook.com/
[3]https://twitter.com/
[4]https://plus.google.com/

using the notion of friendship. As the functionality of social networks is primarily based on data generated by users, data handling and privacy are important issues to them. Thus, in order to keep data away from undesirable viewers, social networks introduce some access control mechanisms that enable users to restrict their data visibility to a desirable subset of users.

Although these data may be of different types, we focus on the text content exchanged between users. Usually, this text context is referred to as *post*. As the social network population grows, the amount of shared data among users also grows. This mandates the use of full-text search engines in order to help users to easily find the content they are looking for. However, while not all data is accessible to everyone, the search engine must adhere to the privacy settings enforced on each users' content.

As a matter of fact, the problem of enforcing access control at desktop and enterprise search is also inherited in social networks. Bjørklund et al. [3], first integrated access control of social network content in a full-text search engine. More particular, they investigated several ways of index designs, but they concluded to the use of a single index containing all users' documents along with user or friend lists. In the case of user lists, each user has his author list which contains the document IDs posted by him. In the case of friend lists , each user also has an author list, but this list contains all documents posted by him and all of his friends as well. Therefore, in order to enforce access control, the results from the index are intersected with the set of author lists that correspond to the user and all his friends. In the case of friend lists, the set of the author lists is calculated by the users' single author list, as it contains the document IDs posted by him and his friends. In the case of user lists, the set of the author lists is calculated by the union of the author lists for each individual friend of the particular user. However, the friend lists approach introduces an update cost as each document posted by a user is inserted in the author lists of all of his friends, while the user lists approach degrades the search performance as multiple author lists must be processed to answer a single query.

In [4], Bjørklund et al. extended their previous work by introducing a new hybrid approach. Initially, each user has one author list that contains the document IDs only posted by him. Also, each user has an additional author list that contains all documents authored by a selected set of users L_u which is a subset of the corresponding user's friends. Thus, there is no need to access the specific author lists for users in L_u whenever a user

u issues a query. The search engine computes the intersection of a query term posting list with the union of author lists; then it returns the query result. As more and more users are represented in L_u, queries become more efficient at the expense of update cost. Hence, the workload characteristics and the use of cost models in optimization algorithms contribute to the selection of an appropriate content for each L_u.

Finally, Facebook offers Inbox Search which is a feature that enables users to search through their Facebook Inbox. Inbox messages also have restrictive visibility, and thus their access control restrains must be retained through the search process. For this purpose, Facebook maintains a per-user index of all messages that have been exchanged between the sender and the recipients of the message. Also, it uses Cassandra as its backend storage system [24]. When messages are exchanged between a small set of users, the per-user index is an affordable solution. However, when talking about posts, which are visible from an extensively larger user set, issues arise from the content redundancy. This is caused due to the duplication of a single *post* to the index of each user that is allowed to read it. Furthermore, this can be worse, as generally in social networks, the number of a users' posts tends to increases as the number of his friends grows [18].

7.4 Secure Data Storage

Data handling to protect privacy is a more general problem and also concerns many online applications and storage systems. Online applications are vulnerable to the disclosure of private information due to software bugs that permit arbitrary users to gain access to private data.

Popa et al. [34] introduced a new system called CryptDB for securing database-backed applications. They address two basic threats: a) a user that gains complete control of application and database management system (DBMS) server including the CryptDB proxy server, and b) a database administrator (DBA) that has the ability to capture and leak private data by snooping the DBMS sever. The main idea is to encrypt all data stored in the database and execute queries over the encrypted data. CryptDB works as a middle layer, that receives all queries (including search on encrypted text), encrypts data and sends it to the DBMS server. Then, it receives the encrypted data from the database, decrypts and sends it to the authorized user. Different keys are used to encrypt different columns and users' data. Also, data may be encrypted in one or more *onions of encryption* (different encryption types) depending on the queries applied over them. All keys are stored in the CryptDB proxy server and keys that decrypt the data accessible to a single user are chained to his password. CryptDB allows only authorized users to gain access to encrypted data and minimizes the amount of revealed private data. In particular, it restricts the leakage to the data of currently active users for the duration of the compromise. In addition, the DBMS server never receives decryption keys needed to decrypt data, ensuring that a DBA cannot gain access to private data.

Schultz et al. [36] secure databases that handle data of multiple users through a decentralized information flow control system called IFDB. The system tracks information as it flows in the database and controls what can be revealed. To achieve this, it is based on three basic concepts: *principals*, *tags*, and *labels*. Principals are entities in the system such as users that are interested in controlling the sensitivity of their data. Tags are identifiers attached to data to denote their sensitivity, and labels are sets of tags summarizing the sensitivity of all data contained in a data object. Each process that runs with the authority of a particular principal has a label, which reflects the tags of all the data this process reads. The basic rule is that information can flow from a source S to a destination D if the labels of S are a subset of the labels of D. However, in some cases tags can be removed from labels in order to send sensitive information to an authorized user. Overall, IFDB controls the information flow and enforces a security policy preventing sensitive information leakage.

Cryptographic storage systems that store and manage files of multiple users also provide mechanisms that enable file sharing and encryption of the stored files. In order to share encrypted files in such systems one has to manage and share keys among users sharing a file with an efficient and scalable manner.

Plutus [23] is a cryptographic storage system that provides secure file sharing over an untrusted file server by encrypting files. Its main idea is to group all files with identical sharing attributes into the same group called *file group* and protect them with the same key. This reduces the number of keys that users need to manage and exchange because the number of keys is detached from the growth of the number of files and is restricted to the number of groups of files with different sharing attributes. Every file group is associated with a symmetric key called *file-lockbox* key and is the same for all files within that file group. Hence, whenever a user wants to share a number of files with other users, he creates a file group and generates a file-lockbox key. Then, he distributes the file-lockbox key to the users with whom he shares the files of the particular file group, enabling them to access these files. The way that Plutus operates makes it a secure storage system that protects and shares data over an untrusted server, while enables individual data owners to control who gets access to their files.

7.5 Summary

In this chapter, we presented prior research in the area of full-text search in multi-user environments. Initially, we presented the two basic approaches. The first approach builds one index per user in order to protect privacy and high query performance. However, it is characterized by great disk space consumption and update cost. The second approach indexes all users' documents in a single system-wide index, and uses filtering algorithms before returning the query results. Despite the low disk space consumption and update cost offered by this approach, the result filtering impacts the query performance and poses privacy threats in some cases. Then, we presented some different solutions that eliminate the privacy threats, and outlined some remarkable approaches on metadata search and social network full-text search. Finally, we presented approaches that provide secure data storage.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Privacy protection in full-text search engines over multi-user environments is an important issue. In order to protect privacy, existing solutions utilize different approaches to organize users' documents into indices. A simple approach is to create one index per user. This approach offers high query performance, but at the cost of great space consumption as each document is indexed by the private index of each user that he is allowed to read it. On the contrary, when retaining a system-wide index, all existing documents are indexed regardless of access control restrictions. This approach provides efficient index updates and low storage usage because each document is indexed only once. Moreover, this approach poses severe privacy issues, which can be eliminated in the expense of query performance. A different approach that improves efficiency while protecting privacy creates one index for each set of documents that have the same ACL. Even though this approach protects privacy and has lower maintenance cost than the previous one, it does not provide any parameters to tune the query performance and the maintenance cost.

Motivated from the need to protect privacy and the lack of a tunable solution that trades query performance and maintenance cost, we introduced a novel strategy to organize users' documents in indices. We group documents into clusters based on the similarity of their ACLs. The similarity between the ACL of documents within a cluster is determined by a Similarity parameter. Then, we map documents and users to indices based on the intersection and differences of the ACLs within a cluster. In addition, we use a Threshold parameter which determines in which indices the documents and the users are mapped. Performing several measurements across different Similarity and Threshold values, we show that our strategy introduces a trade-off between the query performance and the maintenance cost. By choosing the appropriate Similarity and Threshold values, we substantially reduce the query response time, while slightly raising the maintenance cost. For a given threshold value, the query response time decreases when Similarity is 60%. Moreover, high Threshold values can further achieve better query performance. Overall, our strategy protects privacy and provides a tunable solution that trades maintenance cost and query performance depending on the needs.

8.2 Future Work

The main direction of our future work is to further investigate the behavior of our solution in the context of a real ACL dataset. Even though the evaluation of our solution is based on observations retrieved from a real ACL dataset, it is of primary importance to experiment with a real ACL dataset in order to further validate the benefits of our strategy.

Moreover, we target to support full-text search over content generated in social networks. Privacy protection is of major importance in such environments, as users enforce access control in the generated content. Hence, we need to examine how our approach performs in a social network dataset and validate its applicability in social networks.

Further exploration of other types of clustering algorithms and similarity measures is also worthwhile.

Another interesting direction for future work is to integrate our solution into a full-text search engine. Furthermore, we intend to investigate the potential of tuning the Similarity and Threshold parameters by inspecting the ACLs as well as the number of documents associated with each of them.

Bibliography

[1] http://googledesktop.blogspot.gr/.

[2] Spotlight overview. http://developer.apple.com/library/mac/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.pdf, Apple Inc. 2004, 2007.

[3] Truls A. Bjørklund, Michaela Götz, and Johannes Gehrke. Search in social networks with access control. In *Proceedings of the 2nd International Workshop on Keyword Search on Structured Data*, KEYS '10, pages 4:1–4:6, New York, NY, USA, 2010. ACM.

[4] Truls A. Bjørklund, Michaela Götz, Johannes Gehrke, and Nils Grimsmo. Workload-aware indexing for keyword search in social networks. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 535–544, New York, NY, USA, 2011. ACM.

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

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

[7] http://technet.microsoft.com/en-us/library/jj219738.aspx.

[8] http://www.copernic.com/.

[9] http://www.x1.com/.

[10] Levent Ertoz, Michael Steinbach, and Vipin Kumar. A new shared nearest neighbor clustering algorithm and its applications. In *Workshop on Clustering High Dimensional Data and its Applications at 2nd SIAM International Conference on Data Mining*, 2002.

[11] Martin Ester, Hans-Peter Kriegel, J?rg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad, editors, *KDD*, pages 226–231. AAAI Press, 1996.

[12] `http://www.google.com/enterprise/search/`.

[13] Christos Faloutsos. Signature-based text retrieval methods: A survey. *IEEE Data Eng. Bull.*, 13(1):25–32, 1990.

[14] `http://lucene.apache.org/`.

[15] LI Guilin, GAO Xing, HUANG Huiying, and Minghong LIAO. A bloom filter based security index for enterprise search engines. *Computational Information Systems*, 8(12):4931–4938, 15 June 2012.

[16] `http://www.elasticsearch.org/`.

[17] D. Harman, E. Fox, R. Baeza-Yates, and W. LEE. Inverted files. In W.B. Frakes and R. Baeza-Yates, editors, *IR*, chapter 3, pages 28–43. Prentice-Hall, 1992.

[18] Bernardo A. Huberman, Daniel M. Romero, and Fang Wu. Social networks that matter: Twitter under the microscope, 2008. cite arxiv:0812.1045.

[19] `http://trec.nist.gov/data/terabyte.html`.

[20] `http://docushare.xerox.com/`.

[21] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, September 1999.

[22] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[23] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

[24] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[25] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, CIKM '05, pages 776–783, New York, NY, USA, 2005. ACM.

[26] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *Proceedings of the 27th Australasian conference on Computer science - Volume 26*, ACSC '04, pages 15–23, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[27] Andrew W. Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. Spyglass: fast, scalable metadata search for large-scale storage systems. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 153–166, Berkeley, CA, USA, 2009. USENIX Association.

[28] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability - Vol. 1*, pages 281–297. University of California Press, Berkeley, CA, USA, 1967.

[29] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[30] M. Eng (Huy Le) Nguyen, Huy. Improving search quality of the google search appliance. Master's thesis, Dept. of Electrical Engineering and Computer Science, 2009.

[31] N.Rajalingam and K.Ranjini. Article: Hierarchical clustering algorithm - a comparative study. *International Journal of Computer Applications*, 19(3):42–46, April 2011. Published by Foundation of Computer Science.

[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[33] Aleatha Parker-Wood, Christina Strong, Ethan L. Miller, and Darrell D. E. Long. Security aware partitioning for efficient file system search. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–14, Washington, DC, USA, 2010. IEEE Computer Society.

[34] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[35] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. Okapi at trec-7: Automatic ad hoc, filtering, vlc and interactive. In *TREC*, pages 199–210, 1998.

[36] David Schultz and Barbara Liskov. IFDB: Decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems*, Prague, Czech Republic, April 2013.

[37] Srivatsa Mudhakar Singh Aameek and Liu Ling. Efficient and secure search of enterprise file systems. In *ICWS*, pages 18–25. IEEE Computer Society, 2007.

[38] D. K. Smetters and Nathan Good. How users use access control. In *Proceedings of the 5th Symposium on Usable Privacy and Security*, SOUPS '09, pages 15:1–15:12, New York, NY, USA, 2009. ACM.

[39] Carol Smith. Spotlight on spotlight. *Info 624 information retrieval systems*, Summer 2005.

[40] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 119–132, New York, NY, USA, 2005. ACM.

[41] Benjamin Turnbull, Detective Sergeant, and Barry Blundell. Abstract googling forensics: An analysis of the google desktop search. *International Journal of Digital Evidence*, 5(1), Fall 2006.

[42] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

Short Vita

Micheli Eirini was born in 1986 in Arta. She was accepted at the Department of Computer Science of the University of Ioannina in 2005. She received her B.Sc. degree in 2011. Then, she continued her studies as a M.Sc. student at the same department under the supervision of Professor Stergios Anastasiadis. Currently, she is a postgraduate student and a member of the Systems Research Group (SRG). Her research interests lie in the fields of full-text search privacy as well as file and storage systems.