Permission-based Index Clustering for Secure Multi-User Search

Eirini C. Micheli, Giorgos Margaritis, Stergios V. Anastasiadis
Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece
E-mail: {emicheli,gmargari,stergios}@cs.uoi.gr

Abstract. Secure keyword search in shared infrastructures prevents stored documents from leaking sensitive information to unauthorized users. A shared index provides confidentiality if it is exclusively used by users authorized to search all the indexed documents. We introduce the Lethe indexing workflow to improve query and update efficiency in secure keyword search. The Lethe workflow clusters together documents with similar sets of authorized users, and creates shared indices for configurable document subsets accessible by the same users. We examine different datasets based on the empirical statistics of a document sharing system and alternative theoretical distributions. We apply Lethe to generate indexing organizations of different tradeoffs between the search and update cost. We show the robustness of our method by obtaining configurations of similar low costs from application of prototype-based and density-based clustering algorithms. With measurements over an open-source distributed search engine, we experimentally confirm the improved search and update performance of the particular indexing configurations that we introduce.

Keywords. confidentiality, privacy, access control, data storage, outsourced services, document sharing, search engines, clustering, full-text indexing

1 Introduction

Keyword search is an indispensable service for the automated text retrieval of diverse storage environments, such as personal content archives, online social networks, and scalable cloud facilities. As the accumulated data is becoming predominantly unstructured and heterogeneous, the role of text processing remains crucial in big-data analytics [15]. Storage consolidation increasingly moves sensitive data to public infrastructures and makes insufficient the confidentiality achieved by storage access control alone. For instance, the aggregation of personal data from seemingly unrelated sources is currently recognized as severe threat to the privacy of individuals [19].

An inverted index is the typical indexing structure of keyword search. The stored documents are preprocessed into a posting list per keyword (or term) with the occurrences (or postings) of the term across all the documents. A single index shared among multiple users offers search and storage efficiency. However, it can also leak confidential information about documents with access permissions limited to a subset of the users [7, 9, 29, 35]. The problem persists even if a query is initially evaluated over the shared index, and the inaccessible documents are later filtered out before the final result is returned to the user [9].

A known secure solution serves queries from a shared index by restricting access to the postings of searchable documents, and filtering out early the postings of documents that are inaccessible to the user [9]. In online social networks, recent research applies advanced list-processing operators and cost models to improve secure search efficiency [7]. First, it organizes the friends of each user into groups based on characteristics of the search workload. During query handling, it intersects the list of documents that contain a term against the list of documents authored by the querying user or members of her friend groups.

A different secure solution partitions the document collection by search permissions, and maintains a separate index for each partition [35]. The indices are systematically encrypted by the client before they are remotely hosted by a service provider. The document collection is indexed by a limited number of indices and query handling is restricted to documents searchable by the querying user. Variations in the search permissions of different documents can increase the number of indices. Smaller indices can be completely eliminated by replication of their contents to private per-user indices, but at increased document duplication across the indices and corresponding update cost.

In the present study, we aim to achieve low search latency and update resource requirements by limiting both the number of indices per user and the document duplication across the indices. We group by search permissions the documents into families, and cluster together the families with similar permissions. We maintain a single index for the documents searchable by a maximal common subset of users in a cluster. Cluster documents whose users lie outside the above subset are inserted into either per-user private indices or additional multi-user indices.

Our indexing organization for secure keyword search is innovative because we: (i) Use prebuilt secure indices to skip unnecessary I/O and list filtering during query time and avoid the consumption of extra resources. (ii) Effectively balance the number of searched and maintained indices through configurable merging of indices for documents with common authorized users. (iii) Experimentally demonstrate achieving low search and indexing time for workloads with different empirical and theoretical characteristics. To the best of our knowledge, Lethe represents the first application of index clustering to achieve efficiency in secure keyword search.

In Sections 2 and 3 we present the Lethe indexing workflow and our prototype implementation. In Sections 4 and 5 we describe our experimentation environment and analyze our experimental results. In Section 6 we examine previous related research, and in Section 7 we summarize our conclusions and plans for future work.

2 Indexing Organization

We next provide the basic assumptions and goals of our work, and describe the stages of the Lethe indexing workflow that we propose. We summarize the definition of the symbols involved in the Lethe workflow at Table 1.

2.1 Assumptions and Goals

We target collections of text documents in shared storage environments accessible by multiple users. The system applies access control to protect the confidentiality and integrity of the stored documents from actions of unauthorized users. We designate as *owner* the user who creates a document, and as document *searchers* the users who are allowed to search for the document by keywords and read its contents. The system preprocesses the documents content into the necessary indexing structure to enable interactive search through keyword criteria set by the searchers. In our indexing organization we set the following goals:

Definition of Symbols

Symbol	*Description*	*Properties*
$A_\mathcal{T}$	relation of search permissions	$\subseteq D_\mathcal{T} \times S_\mathcal{T}$
c	cluster	
$C_\mathcal{T}$	set of family clusters from dataset $\mathcal{T}$	
D_c	document set of cluster c	$= \bigcup_{f \in F_c} D_f$
D^i_c	document set of cluster intersection P_c	$= D_c$
D_f	document set of family f	$\subseteq D_\mathcal{T}$
D^d_f	document set of family difference P_f	$= D_f$
$D_\mathcal{T}$	document set of dataset $\mathcal{T}$	
D^e_u	document set of private collection P_u	$\subseteq D_\mathcal{T}$
f	family	$= (D_f, S_f)$
$F_\mathcal{T}$	set of document families in dataset $\mathcal{T}$	
F_c	family set of cluster c	$\lvert F_c \rvert \leq \lvert D_c \rvert$
$I_\mathcal{T}$	set of indices for dataset $\mathcal{T}$	
I_c	shared index of cluster intersection P_c	$\in I_\mathcal{T}$
I_f	shared index of family difference P_f	$\in I_\mathcal{T}$
I_u	private index of searcher u	$\in I_\mathcal{T}$
L_s	searcher similarity	$\in [0, 1]$
M_f	access bitmap of family f	
N^c_c	number of created clusters	$\leq N^r_c$
N^r_c	number of requested clusters	
P_u	private collection of searcher u	$= (D^e_u, \{u\})$
P_c	intersection of cluster c	$= (D^i_c, S^i_c)$
P_f	difference of family f	$= (D^d_f, S^d_f)$
R^d_f	duplication product	$= \lvert D^d_f \rvert \cdot \lvert S^d_f \rvert$
S^i_c	searcher set of cluster intersection P_c	$= \bigcap_{f \in F_c} S_f$
S_f	searcher set of family f	$\subseteq S_\mathcal{T}$
S^d_f	searcher set of family difference P_f	$= S_f - S^i_c$
$S_\mathcal{T}$	searcher set of dataset $\mathcal{T}$	
$\mathcal{T}$	dataset	$= (D_\mathcal{T}, S_\mathcal{T}, A_\mathcal{T})$
T_d	duplication threshold	$\in [0, \infty)$
u	searcher user	

Table 1: We summarize the definition of symbols used in the description of Lethe.

- **Security** Ensure that the indexing structure provides confidentiality of the searched documents with respect to the document contents and their statistical characteristics (e.g., number of documents, term frequency).
- **Search efficiency** Minimize appropriate metrics (e.g., average, percentile) of the search latency per query.
- **Indexing cost** Minimize the document insertion I/O activity and indexing storage space required for the entire collection.

We require that users be authenticated by the system and receive authorization to only search for documents with the necessary access permissions. Accordingly, we build a sep-

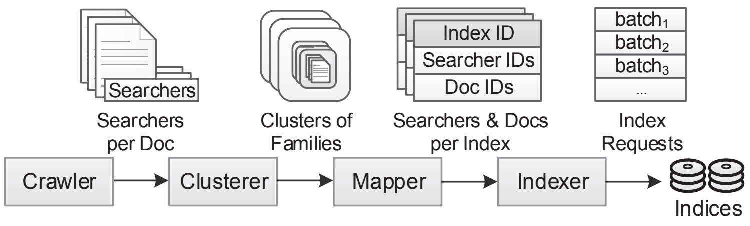


Figure 1: The four stages of the Lethe workflow.

Algorithm 1: Crawler

```
Input: $O$: folder location
Output: $\mathcal{T}$: dataset
1 // Crawl documents and searchers of dataset
2 $D_{\mathcal{T}} \leftarrow$ get_doc_ids($O$)
3 $S_{\mathcal{T}} \leftarrow$ get_searcher_ids($O$)
4 $A_{\mathcal{T}} \leftarrow$ get_search_perms($O$)
5 $\mathcal{T} \leftarrow (D_{\mathcal{T}}, S_{\mathcal{T}}, A_{\mathcal{T}})$
```

arate index for each document subset with common access permissions. We assume that search with encrypted keywords over encrypted documents does not necessarily hide the search activity and stored content from the storage provider or co-located unauthorized searchers [35, 37, 41]. Therefore, we examine secure multi-user search orthogonally to storage encryption.

2.2 Lethe workflow

In order to realize our goals we need to build an indexing organization based on the search permissions of the documents. Thus we introduce the Lethe workflow consisting of four basic stages to crawl the document metadata, cluster together the documents with similar searchers, map the documents to indices, and build the indices in the search engine (Fig. 1).

Crawler For a text dataset $\mathcal{T} = (D_{\mathcal{T}}, S_{\mathcal{T}}, A_{\mathcal{T}})$, we define as $D_{\mathcal{T}}$ the set of documents, $S_{\mathcal{T}}$ the set of searchers, and $A_{\mathcal{T}} \subseteq D_{\mathcal{T}} \times S_{\mathcal{T}}$ the document-to-searcher relation that identifies the users who are allowed to search each document. At the first stage of Lethe, we crawl the names (e.g., paths) and permissions (e.g., allowed searchers) of the documents in the dataset. Thus we assign unique identifiers to the documents and searchers of $\mathcal{T}$ generating the sets $D_{\mathcal{T}}$, $S_{\mathcal{T}}$, and $A_{\mathcal{T}}$ (Algorithm 1).

Clusterer We group into a separate *family* $f = (D_f, S_f)$ each set of documents $D_f \subseteq D_{\mathcal{T}}$ with identical set of searchers $S_f \subseteq S_{\mathcal{T}}$ generating the set of families $F_{\mathcal{T}}$ (as further explained in Section 3). In order to maintain a single index for the searchers who are common among similar families, we need to identify those families with substantial overlap in their searcher sets. We address this issue as a universal clustering problem over the searchers of the families in $F_{\mathcal{T}}$. We parameterize the clustering method appropriately to assign every family to exactly one cluster (i.e., without omitting any family as noise).

Let *searcher similarity* $L_s \in [0, 1]$ be a configurable parameter to adjust the number of common searchers across the families of each created cluster. We generate a set $C_{\mathcal{T}}$ of family clusters, where each cluster $c \in C_{\mathcal{T}}$ contains a set F_c of families and each family $f \in F_c$ contains the document set $D_f \subseteq D_{\mathcal{T}}$ (Algorithm 2). The document set D_c of cluster c is derived

Algorithm 2: Clusterer

```
Input: $\mathcal{T}$: dataset, $L_s$: searcher similarity
Output: $F_\mathcal{T}$: family set, $C_\mathcal{T}$: set of family clusters
1  // Create document families
2  Table Families
3  for each document $d$ in $D_\mathcal{T}$ do
4      $key \leftarrow$ get_searcher_ids($d$, $\mathcal{T}$)
5      Families(key).add_doc($d$)
6  end
7  for each family $f$ in Families do
8      $D_f \leftarrow get\_doc\_ids(f)$
9      $S_f \leftarrow get\_searcher\_ids(f)$
10     $F_\mathcal{T} \leftarrow F_\mathcal{T} \cup \{(D_f, S_f)\}$
11 end
12 // Cluster families by $L_s$
13 $C_\mathcal{T} \leftarrow$ family_clustering($F_\mathcal{T}$, $L_s$)
```

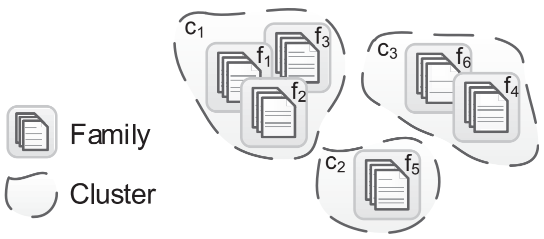


Figure 2: Clustering by searcher similarity L_s. We show an example of six document families $f_1, f_2, f_3, f_4, f_5, f_6$ grouped into three clusters c_1, c_2, c_3.

from the union of the documents contained across all the families of c, i.e., $D_c = \bigcup_{f \in F_c} D_f$.

As a clustering example, in Fig. 2 we show the document families $f_1, f_2, f_3, f_4, f_5, f_6$ partitioned into the three clusters $c_1 = \{f_1, f_2, f_3\}$, $c_2 = \{f_5\}$ and $c_3 = \{f_4, f_6\}$ resulting from the choice of L_s.

Mapper We define as $I_\mathcal{T}$ the set of indices needed to securely index the dataset $\mathcal{T}$. We strive to map each family f to the minimum number of indices required to securely handle keyword queries over D_f, but also minimize the total number $|I_\mathcal{T}|$ of indices maintained for the dataset. First, we dedicate to every searcher $u \in S_\mathcal{T}$ the private collection $P_u = (D_u^e, \{u\})$, where $D_u^e \subseteq D_\mathcal{T}$ is the set of documents exclusively searchable by user u. We assign to P_u a *private* index $I_u \in I_\mathcal{T}$ containing all the documents of D_u^e.

Let the *cluster intersection* of cluster $c \in C_\mathcal{T}$ be a pair $P_c = (D_c^i, S_c^i)$, with $D_c^i = D_c$ the contained documents and $S_c^i = \bigcap_{f \in F_c} S_f$ the intersection of searchers in the families of F_c. From the family definition, the documents in D_c^i are searchable by all the searchers in S_c^i. If $S_c^i \neq \emptyset$, we dedicate a separate index $I_c \in I_\mathcal{T}$ to the intersection P_c.

For every family $f \in F_c$, we define a *family difference* $P_f = (D_f^d, S_f^d)$, where $D_f^d = D_f$ and $S_f^d = S_f - S_c^i$. The set S_f^d contains the searchers of f not included in the S_c^i of intersection P_c. If $S_f^d \neq \emptyset$, the users of S_f^d should be allowed to securely search for documents in D_f^d.

In order to support search over each family difference P_f, an extreme approach is to insert all documents $d \in D_f^d$ to all the private indices $I_u, u \in S_f^d$. However, a difference P_f may

Algorithm 3: Mapper

```
Input: $\mathcal{T}$: dataset, $C_{\mathcal{T}}$: set of family clusters, $T_d$: duplication threshold
Output: $I_{\mathcal{T}}$: set of specified indices
// Create private indices
for each user $u \in S_{\mathcal{T}}$ do
    $D^e_u \leftarrow$ get_private_docs($u$, $\mathcal{T}$)
    $I_u \leftarrow$ create_index($D^e_u$, $\{u\}$)
    $I_{\mathcal{T}} \leftarrow I_{\mathcal{T}} \cup \{I_u\}$
end
for each cluster c in $C_{\mathcal{T}}$ do
    $F_c \leftarrow$ get_families(c)
    // Find cluster intersection
    $S^i_c \leftarrow \bigcap_{f \in F_c} S_f$
    $D^i_c \leftarrow \bigcup_{f \in F_c} D_f$
    // Dedicate index to cluster intersection
    if ($S^i_c \neq \emptyset$) then
        $I_c \leftarrow$ create_index($D^i_c$, $S^i_c$)
        $I_{\mathcal{T}} \leftarrow I_{\mathcal{T}} \cup \{I_c\}$
    end
    // Identify family differences
    for each family f in $F_c$ do
        $S^d_f \leftarrow S_f - S^i_c$
        // Dedicate index to family difference
        if ($S^d_f \neq \emptyset$) then
            $D^d_f \leftarrow D_f$
            // Approximate document duplication
            $R^d_f \leftarrow |D^d_f| \cdot |S^d_f|$
            if ($R^d_f \geq T_d$) then
                $I_f \leftarrow$ create_index($D^d_f$, $S^d_f$)
                $I_{\mathcal{T}} \leftarrow I_{\mathcal{T}} \cup \{I_f\}$
            else
                // Add docs to private indices
                for each user $u \in S^d_f$ do
                    add_docs($I_u$, $D^d_f$)
                end
            end
        end
    end
end
```

contain a relatively large number $|D^d_f|$ of documents searchable by a considerable number $|S^d_f|$ of users. Hence, we could end up with a large number of documents duplicated across the private indices of many users. At the other extreme, we could dedicate a separate index $I_f \in I_{\mathcal{T}}$ to every difference P_f with $S^d_f \neq \emptyset$. However, this approach runs the risk of generating a large number of indices, each serving a small number of documents and searchers.

We introduce the *duplication product* $R^d_f = |D^d_f| \cdot |S^d_f|$ to approximate the potential document duplication resulting from indexing a family difference. Although not examined

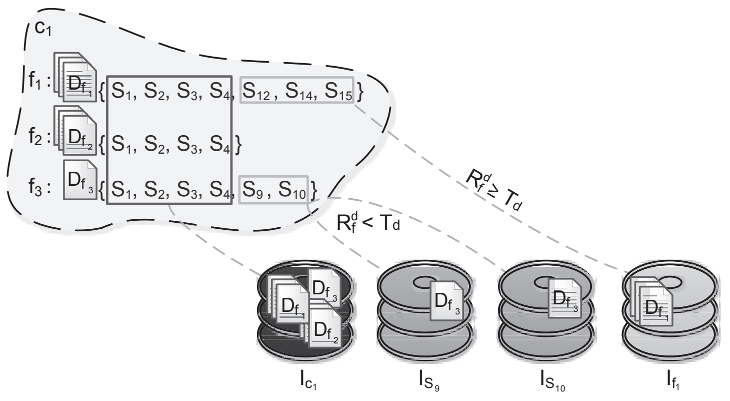

c1
f1 : Df1 {S1, S2, S3, S4, S12, S14, S15}
f2 : Df2 {S1, S2, S3, S4}
f3 : Df3 {S1, S2, S3, S4, S9, S10}
Rfd ≥ Td
Rfd < Td
Ic1
IS9
IS10
If1

Figure 3: Cluster mapping to intersection, difference and private indices based on comparison of R^d_f to the duplication threshold T_d. We show an example with four indices based on cluster c_1 of three families f_1, f_2, f_3 from Fig. 2.

further, for increased accuracy of R^d_f over diverse document sizes, an alternative approach could be to replace $|D^d_f|$ with the total number of postings contained in all documents $d \in D^d_f$.

Subsequently, the decision of whether we should create a dedicated index I_f depends on how R^d_f compares to the configurable *duplication threshold* T_d. We assume that $R^d_f < T_d$ implies an affordable cost of inserting the documents $d \in D^d_f$ to private indices $I_u, u \in S^d_f$. Instead, $R^d_f \geq T_d$ suggests that devoting a separate index I_f to the difference P_f is preferable. We use pseudocode to summarize the above steps in Algorithm 3.

An optimization that we also examine is to pursue additional duplication reduction by intersecting the searchers of the differences $P_f, f \in F'_c$, for appropriate family subsets $F'_c \subset F_c$ of all the clusters $c \in C_{\mathcal{T}}$. Thus, we strive to take advantage of possible searcher similarity across all the family differences of the entire dataset (further explained in Section 5.2).

Continuing the example of Fig. 2, in Fig. 3 we show the cluster c_1 consisting of three families with common searchers S_1, S_2, S_3, S_4. We create an intersection index I_{c_1} that contains the documents $D_{f_1} \cup D_{f_2} \cup D_{f_3}$ of the three families and is accessible by the searchers S_1, S_2, S_3, S_4. We create a difference index I_{f_1} for the searchers S_{12}, S_{14}, S_{15} and the documents D_{f_1}. Instead, we copy the documents of D_{f_3} to the private indices I_{S_9} and $I_{S_{10}}$ of the respective searchers S_9 and S_{10}.

Indexer We insert each document $d \in D_{\mathcal{T}}$ to the appropriate indices of $I_{\mathcal{T}}$ specified by the above mapping stage. In order to keep low the I/O activity required by the search engine for index building, we generate each index separately through a specification of the allowed searchers and contained documents (Algorithm 4 further explained in Section 3). We have experimentally validated that the alternative approach of specifying to the search engine the list of indices containing each document leads to lower storage locality and higher I/O activity during index creation.

As new documents are added to the collection, we seek to use existing indices to securely serve all the searchers of each document. Periodically, we repeat the previous clustering and mapping stages to optimize the search over the accumulated document collection. Deletions or modifications of inserted documents are handled with the necessary changes of the index contents and potential reorganization of their mapping to documents.

Algorithm 4: Indexer

```
Input: $\mathcal{T}$: dataset, $I_{\mathcal{T}}$: set of indices
Output: Search engine with prebuilt document indices for authorized searchers
// Build indices by document batches
Batch batch
for each index i in $I_{\mathcal{T}}$ do
    batch.set_searchers($i$, $S_i$)
    for each document d in $D_i$ do
        batch.load_text($i$, $d$, $\mathcal{T}$)
        if (batch.size() $\geq$ MAX_BATCH_SIZE $\vee$ no more documents) then
            search_engine.build_index(batch)
            batch.clear_docs()
        end
    end
end
```

3 Lethe Prototype

Based on the above workflow design, our software prototype consists of four components: (i) *crawler*, (ii) *clusterer*, (iii) *mapper*, and (iv) *indexer*.

The crawler specifies a unique identifier for each document and gathers information about the permitted document searchers. The clusterer organizes the documents into families according to their searchers, and then clusters the families based on their relative searcher similarity L_s. We insert each document identifier into a hash table by using as key the respective list of authorized searchers (Algorithm 2). The documents with identical searchers belong to the same family and have their identifiers stored at the same entry of the table.

The searchers of family f are concisely represented through the *access bitmap* M_f. The bitmap length is equal to the total number of users $|S_{\mathcal{T}}|$ in the dataset. The bit value of M_f is set to 1 at the positions specified by the identifiers of the searchers $u \in S_f$. Families with similar access bitmaps are grouped into the same cluster, which is represented as a vector of family identifiers. The clustering algorithm that we use either receives the number of clusters as input (e.g., K-medoids), or determines it automatically based on the provided value of searcher similarity L_s (e.g., DBSCAN) [38].

Within each cluster, the mapper identifies the cluster intersections and family differences and specifies their contained documents and authorized searchers, respectively. We assign a dedicated index to each cluster intersection. However, we use a dedicated index, or the private indices of the respective searchers, for each family difference according to the duplication product R_f^d and the duplication threshold T_d. The indexer receives the index specifications from the mapper and splits each index into document batches (Algorithm 4). Then it communicates with the search engine to insert the documents of each batch to the respective index after the necessary initialization.

The search engine serves queries by using the permitted indices of each authorized searcher. In our prototype implementation, we use the Elasticsearch distributed search engine [16]. Elasticsearch is free, open-source software written in Java and based on the Apache Lucene library. It uses one or multiple servers (nodes) to store the indices and serve incoming queries. If an index contains a large amount of documents, it is split into smaller parts, called shards. Each shard can be placed on a separate node or replicated on multiple nodes for improved performance and availability.

4 Experimentation Environment

Starting from the published statistics of a real dataset, we generate a synthetic workload and several variations derived from theoretical distributions of different permission parameters. We obtain our experimentation results with a prototype implementation of the Lethe workflow, which includes indexing and query handling over the Elasticsearch search engine.

4.1 Document Datasets

We generate synthetic document collections with searcher lists based on published measurements of an existing dataset (DocuShare [36]), or controlled variations of permission parameters according to theoretical distributions. From DocuShare, we set the number of users to 200, user groups to 131, documents to 50000, and the maximum group size to 50. Accordingly, we also configure the maximum number of users per document equal to 29 and the maximum number of groups per document equal to 7.

We uniformly assign users as group members, but determine the size of each group based on either the DocuShare statistics, or the Zipfian distribution with α = 0, 0.7, or 2.2. We also uniformly assign users and groups to each document, but specify the number of users and groups allowed to search each document either according to the DocuShare statistics, or based on the Zipfian distribution with α = 0, 0.7, or 2.2.

We evaluate our solution over the first 50000 documents (820MB) of the GOV2 dataset from the TREC 2006 Terabyte track [39]. Our query set consists of the 50000 standard queries from the Efficiency Topics in the TREC 2006 Terabyte track [39]. The average number of terms per query is equal to 2.8.

4.2 Clustering

We use a partitioning clustering algorithm to exclusively assign each family to a single cluster. Each family is represented by an access bitmap consisting of asymmetric binary attributes to track the document accessibility by individual searchers. We measure object closeness with the Jaccard similarity coefficient, which is commonly used for objects with asymmetric binary attributes [38].

We primarily use the DBSCAN (density-based spatial clustering of applications with noise) algorithm because it automatically determines the number of clusters by locating regions of high density separated by regions of low-density [17]. For n points, the computational complexity of DBSCAN is $O(n^2)$ in the worst case, and $O(n \log n)$ if efficient spatial data structures are used for neighborhood search; the space complexity of DBSCAN is $O(n)$ [38]. The algorithm uses the parameter Eps, as the radius delimiting the neighborhood area of a point, and the parameter MinPts, as the minimum number of points in a neighborhood required to form a cluster. In our experiments, we set MinPts = 1 and Eps = L_s.

Alternatively we use the K-medoids algorithm, a prominent partitioning clustering method based on K-means [21]. As medoid is chosen the most representative point of a group of points. The method can be widely applied to different types of data because it only requires a proximity measure for each pair of objects. The algorithm chooses a number of initial medoids according to the user-specified parameter K, and then repeatedly attempts to minimize the distance between the points of each cluster and the respective medoid. The computational complexity of K-medoids with k clusters and n points is $O(k(n-k)^2)$ per iteration [27].

In our experiments we vary the parameter K to examine the sensitivity of the obtained results. For clarity, we refer to the number K of requested clusters with the symbol N_c^r, and the number of clusters actually created by the algorithm with the symbol N_c^c.

A proper initialization method is crucial for the convergence to a solution that is close to optimal. For improved clustering accuracy, the K-means++ initialization chooses uniformly the first medoid, and then picks each subsequent medoid with probability proportional to the squared distance of the point from its closest medoid [2]. In our experiments, we combine K-medoids with K-means++ [34].

4.3 System Implementation

We implemented the crawler, clusterer and mapper in C/C++ with STL, and the indexer in Perl (v5.10.1). In our experiments, we use Elasticsearch v0.90.1. We execute the Lethe workflow and the Elasticsearch client on a machine with one quad-core x86 2.33GHz processor, 4GB RAM, 1 GbE link, and two 250GB 7.2KRPM SATA disks. The clustering algorithm in each experiment takes limited amount of time in the order of tens of seconds. We run Elasticsearch on two servers, each with two quad-core x86 2.33GHz processors, 4GB RAM, 1 GbE link, and two 7.2KRPM SATA disks of capacity 500GB and 1TB. Each server devotes 2GB of RAM to the execution of Elasticsearch. All the machines use Debian v6.0 squeeze with Linux kernel v2.6.32. Next we experimentally evaluate and analyze several important properties of our workflow.

5 Measurement Results

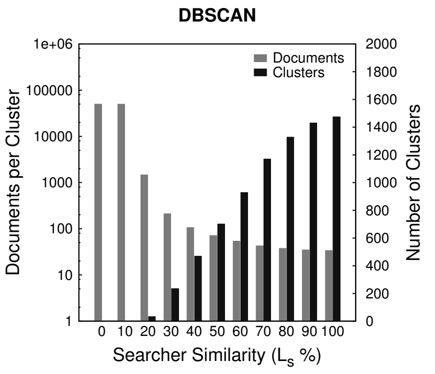
We first examine in detail the indexing characteristics of alternative clustering algorithms and configurations. Then we measure the performance of query handling and requirements of index building for the DocuShare dataset over the Elasticsearch distributed search engine. We also examine the sensitivity of indexing costs to different parameters and group distributions. Finally, we analyze the security and efficiency characteristics of our approach.

5.1 Cluster-based Indexing

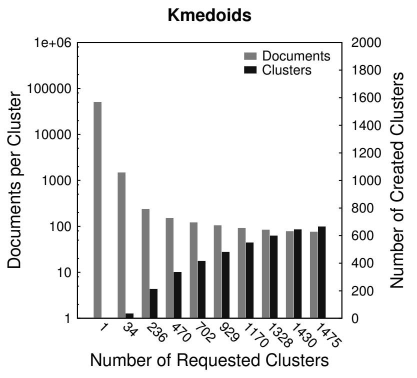
We applied the Lethe workflow to organize into clusters the document families of DocuShare. In Figure 4, we show the number of created clusters and documents per cluster for the DBSCAN and K-medoids algorithms. The DBSCAN algorithm results into two extreme cases: L_s = 0% with 1 cluster of 1475 families and 50000 documents, and L_s = 100% with 1475 single-family clusters of 33.9 average documents per cluster (Fig. 4a). Instead, a moderate L_s = 60% creates 929 clusters with 53.8 documents per cluster.

Using the K-medoids algorithm, we experimented with different numbers N_c^r of explicitly requested clusters (x axis) in the range 1-1475. In Fig. 4b, the number N_c^c of created clusters on the right y axis lies in the range 1-664. Indicatively, the algorithm leaves 811 empty clusters when 1475 are requested. The average number of documents per cluster varies from 50000 in 1 cluster to 75.3 in 664 clusters. In particular, when we set N_c^r = 929, the algorithm creates N_c^c = 480 clusters with 104.2 documents per cluster.

In Fig. 5, we examine in detail the number of intersection (I_c), difference (I_f), and private (I_u) indices resulting from different L_s and T_d values. As we vary T_d from 0 to ∞ with DBSCAN in Figures 5a-d, the total number of indices drops considerably. Accordingly, the



(a)



(b)

Figure 4: For the synthetic dataset based on DocuShare[36], we examine the number of created clusters and the number of documents per cluster across different L_s and N_r^c values.

maximum number of indices decreases from 1561 at $T_d = 0$ ($L_s = 50\%$), to 703 at $T_d = 500$ (30%), 337 at $T_d = 1500$ (40%), and 292 at $T_d \rightarrow \infty$ (50%). In our measurements, most of the difference indices at $T_d \leq 1500$ correspond to families of clusters with empty intersection. Higher T_d values suppress the creation of difference indices and replicate more documents across the private indices. Thus, setting $T_d \rightarrow \infty$ minimizes the total number of indices.

In Figures 6a-d, we show the detailed breakdown into different types of indices created by the K-medoids algorithm. The maximum number of indices varies from 1659 at $T_d = 0$, 681 at $T_d = 500$, 391 at $T_d = 1500$ to 364 at $T_d \rightarrow \infty$. In comparison to DBSCAN, K-medoids creates more indices in several cases except for $T_d = 500$. A notable difference between the two algorithms is that the number of intersection indices created by K-medoids lies in the range 91-164, unlike the respective count of DBSCAN that lies significantly lower in the range 1-92. This is an important observation because intersection indices take advantage of the searcher similarity among the documents for improved index sharing.

Arguably, we need to balance the indices per searcher against the indices per document in order to achieve high performance in both search and update operations. For instance, setting $L_s = 60\%$ and $T_d = 1500$ to DBSCAN lets the mapper specify a moderate number of 298 indices: 84 shared indices for cluster intersections, 14 indices for specific family differences, and 200 private indices for individual searchers (Fig. 5c). Next we examine in detail how different indexing configurations affect the estimated search and build cost.

5.2 Search and Update Cost with DBSCAN and K-medoids

The search cost is determined by the number of document lists merged for the search result and depends on the number of indices per searcher. In Fig. 7a, we examine different values of L_s and T_d over DocuShare with DBSCAN. The indices per searcher vary in the range 35.8 - 79.6 at $T_d = 500$, and the range 11.6 - 22.5 at $T_d = 1500$. Setting $L_s = 0\%$ or 100% tends to maximize the indices per searcher, because it only permits a limited amount of index sharing within each cluster (of a single family or diverse families). On the contrary, setting L_s = 60% usually minimizes the indices per searcher due to index sharing within both cluster intersections and family differences. However, setting $T_d \rightarrow \infty$ suppresses the index sharing of family differences and inverts the effect of L_s; then L_s = 0% or 100% minimizes the number of indices per searcher.

The cost of index building depends on the average number of indices that handle a document and have to be updated during document insertion. In Fig. 7b, we examine the sensitivity of indices per document to L_s and T_d with DocuShare. For example, setting T_d = 1500 or $T_d \rightarrow \infty$ varies the number of indices per document in the ranges 7.8 - 11.7 and 53.5 - 60.1, respectively. As before, higher T_d values discourage the creation of shared indices, dedicated to family differences, and replicate more documents across the private indices. At T_d = 0 or 500, the curves remain almost flat and the number of indices drops to about 1 or 5.

From our observations on the search and update cost from Figures 7a,b we notice diverse effects across the L_s and T_d parameters. In particular, L_s leads the two costs to low or peak values depending on T_d. On the contrary, a higher T_d consistently decreases the search cost but increases the update cost. We found as one reasonable choice setting L_s = 60% and T_d = 1500 to combine 11.6 indices per searcher with 7.8 indices per document. If we reduce T_d to 500 in DocuShare, the search cost increases by a factor of 3 (from 11.6 to 35.8), and the corresponding update cost only drops by 41% (7.8 to 4.6).

In Figures 7c,d we also examine the indices per searcher and per document across different values of N_c^c and T_d with K-medoids. Setting T_d = 1500 lowers the indices per searcher in the range 10.9 - 22.3 and leads to indices per document in the range 7.1 - 11.7. Subsequently, with N_c^c = 415 clusters, we get 11.6 indices per searcher and 7.1 indices per document. This outcome is remarkably close to the respective search cost of 11.6 and update cost of 7.8 obtained above with the suggested configuration of DBSCAN.

We additionally investigated the possibility of further clustering together the searchers of the family differences. We did that with the expectation of further reducing the indices per searcher and per document through aggressive index sharing. Although not included in our shown figures, with DBSCAN, we end up with 12.2 indices per searcher and 7.6 indices per document for L_s = 60% and T_d = 1500. With K-medoids, the configuration with N_c^c = 334 and T_d = 1500 leads to 11.1 indices per searcher and 6.9 indices per document. Both these configurations of DBSCAN and K-medoids have costs comparable to the results taken above without difference clustering. Due to its limited benefit, we decided to skip this optimization for the rest of our experiments.

More generally, appropriate values of T_d and L_s or N_c^c can flexibly balance the indices per searcher and per document according to the operation characteristics of the workload and the optimization objectives of the service provider (e.g., optimize the system for handling search or update operations more frequently). In the rest of our experiments, we only use DBSCAN, because we found the choice of L_s more intuitive in comparison to N_c^c, which is indirectly generated through the user-specified N_c^r input parameter of the K-medoids.

5.3 Elasticsearch

In Fig. 8, we experimentally consider indexing the DocuShare dataset using five configurations with L_s = 60% or 100%, and T_d = 0, 500, 1500 or $T_d \rightarrow \infty$. We examine a number of clients concurrently submitting back-to-back queries against the Elasticsearch engine. Each client independently submits 6000 queries uniformly chosen from a standard pool of 50000 queries. We use the first 1000 queries for cache warmup in the Elasticsearch nodes, and only report the response time from the last 5000 queries per client.

In Fig. 8a, we show the average response time measured over three experiment runs with coefficient of variation CV<0.07. The response time lies in the range 17.7 - 73.5ms with 1 client, and the range 21.8 - 774.2ms with 16 clients. It is remarkable that L_s = 60% with T_d = 1500 leads to response time in the range 17.8 - 21.8ms, and T_d = 500 expands the range to 17.7 - 52.0ms. Essentially, family similarity in clusters allows multiple documents to be served by shared intersection or difference indices (Figures 5b,c) and leads to low query response time. On the contrary, T_d = 0 was found in Fig. 5a to create a large number of indices; as a result, some queries take up to several seconds and increase the average measured time accordingly.

In Fig 8b, we illustrate the cumulative fraction of queries across the measured values of response time. On the left, the curves with L_s = 60% and T_d = 500 or 1500 almost *overlap* with at least 96% of the queries responded within 50ms. At the right side, with L_s = 100% and $T_d \to \infty$, the respective percentage drops below 19%. We attribute this poor performance of the last case (100%,∞) to excessive document duplication and required storage space, which does not allow the indices to simultaneously fit in memory (further demonstrated by Fig 8d explained below).

We additionally measure the query throughput for 1 to 16 clients in Fig. 8c. We notice that throughput rises by a factor of 9.2, from 37.0q/s to 339.8q/s, in case (60%, 1500), but only reaches 201.1q/s in curve (60%, 500). In the remaining cases, throughput stays below 39.7q/s (e.g., L_s = 60 and T_d = 0 at 2 clients) as a result of the number of indices accessed per query, or the total amount of indexing information involved in query handling.

We illustrate the resource requirements of index building in Fig 8d. The elapsed time and storage space vary in the ranges 12.9 - 307.2min and 0.5 - 22.7GB, respectively. Setting L_s = 100% with T_d = 0 or $T_d \to \infty$ leads the requirements at the low or high end of the above ranges. Instead, L_s = 60% keeps the measured values in-between, but close to the low end, e.g., 46.5min and 3.4GB for T_d = 1500. This outcome makes sense, because L_s = 60% creates a moderate number of indices as a result of clustering, unlike L_s = 100% that creates a minimum or maximum number of indices (Fig. 5).

5.4 Size Distribution of Searcher Groups and Permission Lists

We also explore the sensitivity of the search and update cost to variations in the distribution of group size or number of users and groups per document.

As an intermediate step, in Fig. 9a we illustrate the cumulative fraction of groups corresponding to different numbers of searchers. In three curves the group size follows Zipfian distribution with α set to 0 (uniform), 0.7, or 2.2, but it follows the DocuShare empirical statistics in the fourth curve. Fig. 9a illustrates that a low α equally distributes the sizes across the different searcher groups, unlike higher α values that narrow the larger sizes to fewer groups. From Fig. 9b it follows that the closer to uniform (α = 0) the distribution gets, the higher the number of searchers per document becomes. Thus, lower α values are likely to require more indices per searcher and per document for secure query handling.

In Fig. 9c, we set the number of users and groups per document to the statistics of DocuShare or Zipfian distribution. Thus we specify directly the number of entries in the permission lists rather than indirectly through the size of the searcher groups. Although somewhat higher, the number of searchers per document follows similar trends to those of Fig. 9b. It is interesting that the DocuShare statistics have average number of searchers per document roughly corresponding to Zipfian group size with α = 1.2 (not shown), but Zipfian number of users and groups per document with α = 2.2. Starting with these basic observations, we next examine in detail the sensitivity of the search and update cost to the Zipfian distribution of the searcher groups and the permission lists.

In Fig. 10, we examine the search cost for Zipfian group sizes of α = 0, 0.7, or 2.2. With higher α, the number of indices per searcher decreases because more groups have smaller size (Fig. 9a), a searcher belongs to fewer groups, and a smaller number of indices handle the documents of the searcher. For different L_s and T_d values, we notice the same trends already observed in the original DocuShare dataset in Fig. 7 (e.g., L_s = 60% reduces the indices per searcher for $T_d \not\to \infty$). The impact of L_s diminishes as α increases, because the lower similarity among the searchers reduces the opportunity for index sharing.

In Fig. 11, we show the search cost for Zipfian distribution of the number of users and groups per document. In comparison to Fig. 10, the indices per searcher is higher across the different values of α. Also, in Fig. 11 the reduction of indices per searcher at moderate L_s remains about the same as α increases, unlike a corresponding drop in Fig. 10. Additionally, we found the Zipfian number of users and groups for α = 0, 0.7 and 2.2 resulting into up to 62, 83 and 124 intersection indices (not shown), respectively, unlike 133, 100 and 93 obtained from variation of the group size. Our results suggest that the more intersection indices of the group size translate into lower search cost due to improved index sharing among the searchers.

In Fig. 12 we consider the sensitivity of the indices per document (update cost) to the Zipfian distribution of group size. By setting α = 0, 0.7 and 2.2, the maximum number of indices per document drops from 79.0 down to 69.0 and 51.8, respectively. This behavior follows from the decreasing group sizes at higher α and the lower document duplication across the private indices. Within each plot, there is a low point reached at L_s = 60% for T_d = 1500 or $\to \infty$, and a respective peak at T_d = 0 or 500. As with the search cost above, the number of indices per document is substantially less sensitive to L_s at increasing α.

We also examine the effect from Zipfian number of users and groups per document in Fig. 13. Setting α = 0, 0.7 or 2.2 reduces the maximum number of indices per document from 83.0 to 74.7 and 59.5, respectively. Setting $T_d \to \infty$ to suppress the difference indices, L_s = 60% reduces the update cost for decreasing α in Fig. 13 rather than increasing α value in Fig. 12. Essentially, the Zipfian users and groups with higher α lead to increased number of indices per document in comparison to Zipfian group size, and provide more opportunity for index sharing with the appropriate similarity settings of DBSCAN.

Overall, the Lethe method leads to different combinations of search and update costs, based on the available similarity of authorized users across variations of the group size and number of users and groups per document.

5.5 Further Analysis and Discussion

Our experiments provide strong evidence for an improved method to achieve secure and efficient keyword indexing. An insecure shared index typically implements document ranking based on sensitive statistics of the entire indexed dataset, such as the number of documents that contain a queried term and the total number of documents. For increased efficiency during query processing, search engines often precompute approximate document scores and sort the postings accordingly before they store them compressed on disk [1, 41, 20].

Secure filtering of inaccessible documents requires that the search engine selectively retrieve the postings of accessible documents and calculate their ranking scores for the querying searcher [9]. Hence, search I/O time can be wasted for unnecessarily retrieving postings that will be discarded through filtering, or for seeking over them. Also, the computation of document scoring relies on user-specific statistics (e.g., total number of accessible documents) that have to be calculated during query processing instead of being precomputed earlier during index building. Moreover, efficient user-specific filtering requires a substantial amount of main memory in the servers of the search engine to keep the access permissions of individual users.

Our method is secure because query processing only involves indices of documents that the searcher is fully permitted to access. At the same time, we avoid the I/O cost of selective posting retrieval per index, the extra computational cost for user-specific document scoring and list processing, and the main memory dedicated to access permissions [7]. For ranking purposes, we only merge the result lists obtained from multiple indices, as typically already done by parallel and distributed search engines [20].

The clustering of document families allows serving the common searchers in the cluster intersection with a single index. The resulting reduced number of indices per searcher translates into smaller number of result lists to be generated and merged during query handling. Our method is robust with respect to two distinct clustering methods that we examined, as they both end up to indexing configurations with similar search and update costs.

In terms of space requirements, existing secure solutions either fully replicate each document to the private index of every authorized searcher, or duplicate the documents to the private indices of authorized users that remain outside the shared indices, or fully avoid duplication by using a single index at extra I/O and computational cost of query processing [35]. Our method flexibly controls indexing duplication through threshold T_d for preventing the insertion of the same document to an excessive number of private indices. Thus, we create extra shared indices if the number of documents and their common searchers justify the additional indexing cost for a family difference.

The seemingly similar problem of role mining applies data mining techniques to automatically discover roles from existing system configuration data. Roles are intermediate *hierarchical* constructs that facilitate the assignment of permissions to users by system administrators. Role mining aims to hierarchically organize *overlapping* permission sets into roles to efficiently assign the authorized permissions to each user [32, 40]. Role mining algorithms generate a prioritized sequence of roles or a complete state of role-based access control. To this end, they minimize a cost measure, such as the number of roles, or the number of user assignments and permission assignments [25].

We claim that index clustering is a different problem because it identifies similar searcher sets of documents to build a shared index for those documents that are accessible by the same searchers. Subsequently, it assigns each user to a disjoint set of indices that cover all the documents that the user can access. Both the flat organization of the document clusters and the disjointness of the indices that serve each user make index clustering distinct. Our optimization objective is to balance the number of indices per user against the amount of document duplication across different indices. We leave for future work the possible alternative approach of examining index clustering as a problem of cost optimization similar to the way that role mining was formulated in previous research.

6 Related Work

We compare our work with related research results previously developed for secure text indexing, remote storage of encrypted documents, and online social networks.

In preliminary prior work, we introduced the Lethe workflow to create secure configurations of shared inverted indices [23]. We subsequently did extensive experimental study of the characteristics of different indexing configuration across a range of empirical and theoretical datasets, and measured the search and update performance over an open-source distributed search engine [24]. This manuscript unifies our above prior work and expands it in several ways, most notably by including consideration of the K-medoids clustering algorithm in comparison to DBSCAN, examination of Zipfian number of users and groups per document, and investigation of the benefit from aggressively clustering together the family differences for further index sharing.

Secure Indexing Büttcher and Clarke examined relevance-ranking search on the vector space model [9]. Secure search must only deliver query results of files searchable by the querying user. A system-wide index is insecure because it can leak sensitive information about file and term statistics. A solution integrates security restrictions into query processing so that the index manager only returns the parts of posting lists that are accessible to the user. However, this approach leads to performance slowdown in some cases and occupies substantial amount of main memory to keep the access permissions.

Singh et al. logically organize the filesystem into access-control barrels, which are sets of files with identical access privileges [35]. The system constructs a separate index per barrel, and restricts query handling to permitted barrels. An access credentials graph is constructed with the access credentials of users, groups and barrels as nodes, and edges that minimally connect users to their groups and barrels. The maintained indices are safely reduced by eliminating barrels of size below a configured threshold. Instead, we apply a clustering method to control the document duplication across different indices.

Li et al. maintain a list of permitted users with each term. At configurable loss of ranking and security accuracy, they combine list nodes of similar inverse document frequency and use a Bloom filter to store the corresponding users [22]. Zerr et al. store appropriately transformed relevance scores in an untrusted server to avoid disclosing information about the indexed data [41]. Pang et al. safeguard the content and result of user queries over the vector space model [28]. The document server uses a partially-encrypted suppressed index to handle queries, but the system allows only authorized users to prune false positives and retrieve the matching documents.

Bawa et al. organize the providers of documents into privacy groups and build bit vectors to summarize the terms of the documents in each privacy group [4]. A designated host constructs a privacy-preserving index from the bit vectors to identify the privacy groups that match a query. In relational databases, information flow control has been applied to serve large amounts of confidential information to many users [33]. A different study aims to improve metadata search efficiency by hierarchically partitioning the filesystem by access permissions but leaves for future study the full merging of partitions with identical permissions, which we do in document families [29].

Encrypted Storage Song et al. describe techniques to securely search remote documents maintained in encrypted form [37]. The client queries the server through a key and a plaintext or encrypted keyword. The server identifies keyword locations through linear scan of the encrypted documents. Chang and Mitzenmacher use an encrypted bitmap to encode the presence of particular keywords in a document [10]. The user submits a permuted keyword identifier along with a key to search for specific encrypted documents. Similarly, the Mafdet system inserts keyed hashes of document keywords into a Bloom filter at the server [3]. Thus, a client only submits keyword hashes to search for documents.

Damiani et al. enable access control for database-as-a-service systems [13]. They group users by access privileges and create a user hierarchy from the possible user sets and subsets. In order to reduce the number of keys per user, they exploit key derivation methods to derive the keys of lower-level nodes from keys of predecessors. For simpler key management, they also transform the directed acyclic graph of the user hierarchy into an equivalent tree structure. Other related research applies data possession proofs to prevent file hiding into the cloud by unauthorized clients [26].

CryptDB supports keyword search over individually encrypted words of a text column in a relational database [31]. Pervez et al. assume that both files and inverted indices are stored in encrypted form at the cloud [30]. Authorized users submit encrypted search criteria to a third party, which homomorphically encrypts them before their transmission to the cloud server. The cloud server uses a user-specific key to re-encrypt the index for query evaluation.

The Sedic system partitions data according to security levels, and only replicates sanitized data to the public cloud for MapReduce processing [42]. PRISM transforms the problem of keyword search over encrypted files into privacy-preserving map and reduce tasks [8]. Secure search of big data returns the records matching a query without revealing the query or its results to the provider [14]. Approximate string matching between two parties is securely implemented without third-party involvement by estimating whether the distance of two encrypted Bloom filters lies below a threshold [5]. Overall, the above research focuses on indexing or processing of encrypted data in cloud storage.

Online Social Networks Keyword search in social networks is possible through a set of inverted indices with each index containing posting lists of documents from particular users. Access control is enforced through intersection of the search result with the identifiers (author list) of documents authored by a particular set of users [6]. Alternative cost models are examined to optimally include specific friends in the author list of each user. Furthermore, the HeapUnion operator is introduced to efficiently process multiple lists of document identifiers [7].

Hummingbird is a microblogging system that cryptographically hides from a user the topics on which other users follow her, and from third parties the fact that a user follows another user on a specific topic [12]. Cheng et al. enable fine-grain specification of access-control policies in user-to-user, user-to-resource and resource-to-resource relationships over social networks [11]. Hails provides data-flow confinement at the client and server so that mutually-untrusted web applications can interact safely [18]. These are general issues of access control in social networks beyond our study scope.

7 Conclusions and Future Work

We use clustering to identify documents with similar sets of authorized searchers. Accordingly, we generate shared indices for documents with common authorized searchers of sufficient volume. We experimentally use tunable parameters to combine low numbers of indices per user and per document. Our method is relatively insensitive to alternative clustering algorithms and parameter variations that we experimented with. Our measurements over a distributed search engine confirm that our indexing organization achieves higher search and update performance. As future work, it is interesting to analytically model the operation costs and study index clustering as a cost optimization problem. We additionally plan to experimentally explore different types of access permission clustering over a broader dataset collection from collaborative environments, cloud storage and social networks.

Acknowledgements

We are thankful to the anonymous reviewers for their constructive comments. This research was supported by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund (Project "Cloud9").

References

[1] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *ACM SIGIR Conference on Research and Development on Information Retrieval*, pages 372–379, Seattle, WA, Aug. 2006.

[2] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1027–1035, New Orleans, LA, Jan. 2007.

[3] S. Artzi, A. Kieżun, C. Newport, and D. Schultz. Encrypted keyword search in a distributed storage system. Technical Report MIT-CSAIL-TR-2006-10, CSAIL, MIT, Feb. 2006.

[4] M. Bawa, R. J. Bayardo, R. Agrawal, and J. Vaidya. Privacy-preserving indexing of documents on the network. *The VLDB Journal*, 18(4):837–856, Aug. 2009.

[5] M. Beck and F. Kerschbaum. Approximate two-party privacy-preserving string matching with linear complexity. In *IEEE International Congress on Big Data*, pages 31–37, Santa Clara, CA, June 2013.

[6] T. A. Bjørklund, M. Götz, and J. Gehrke. Search in social networks with access control. In *International Workshop on Keyword Search on Structured Data*, pages 4:1–4:6, Indianapolis, IN, June 2010.

[7] T. A. Bjørklund, M. Götz, J. Gehrke, and N. Grimsmo. Workload-aware indexing for keyword search in social networks. In *ACM International Conference on Information and Knowledge Management*, pages 535–544, Glasgow, UK, Oct. 2011.

[8] E.-O. Blass, R. D. Pietro, R. Molva, and M. Önen. PRISM - privacy-preserving search in MapReduce. In *Privacy Enhancing Technologies Symposium*, pages 180–200, Vigo, Spain, July 2012.

[9] S. Büttcher and C. L. A. Clarke. A security model for full-text file system search in multi-user environments. In *USENIX Conference on File and Storage Technologies*, pages 169–182, San Francisco, CA, Dec. 2005.

[10] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *International Conference on Applied Cryptography and Network Security*, pages 442–455, New York, NY, June 2005.

[11] Y. Cheng, J. Park, and R. Sandhu. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *IEEE International Conference on Social Computing/IEEE International Conference on Privacy, Security, Risk and Trust (SocialCom/PASSAT)*, pages 646–655, Amsterdam, Netherlands, Sept. 2012.

[12] E. D. Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: Privacy at the time of Twitter. In *IEEE Symposium on Security and Privacy*, pages 285–299, San Francisco, CA, May 2012.

[13] E. Damiani, S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Key management for multi-user encrypted databases. In *ACM Workshop on Storage Security and Survivability*, pages 74–83, Fairfax, VA, Nov. 2005.

[14] K. E. Defrawy and S. Faber. Blindfolded data search via secure pattern matching. *Computer*, 46(12):68–75, Dec. 2013.

[15] V. Dhar. Data science and prediction. *Communications of the ACM*, 56(12):64–73, 12 2013.

[16] An end-to-end search and analytics platform. `www.elasticsearch.org`.

[17] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *AAAI International Conference on Knowledge Discovery and Data Mining*, pages 226–231, Portland, OR, Aug. 1996.

[18] D. G. Giffin, A. Levy, D. Stefan, D. Terei, D. Maziéres, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 47–60, Hollywood, CA, Oct. 2012.

[19] M. Jensen. Challenges of privacy protection in big data analytics. In *IEEE International Congress on Big Data*, pages 235–238, Santa Clara, CA, June 2013.

[20] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Adaptive parallelism for web search. In *ACM European Conference on Computer Systems*, pages 155–168, Prague, Czech Republic, Apr. 2013.

[21] L. Kaufman and P. J. Rousseeuw. *Partitioning Around Medoids (Program PAM)*, chapter 2. John Wiley & Sons, Inc, Hoboken, NJ, 1990.

[22] G. Li, X. Gao, H. Huang, and M. Liao. A Bloom filter based security index for enterprise search engines. *Journal of Computational Information Systems*, 8(12):4931–4938, June 2012.

[23] E. C. Micheli, G. Margaritis, and S. V. Anastasiadis. Efficient multi-user indexing for secure keyword search. In *International Workshop on Privacy and Anonymity in the Information Society (PAIS)*, pages 390–395, Athens, Greece, Mar. 2014.

[24] E. C. Micheli, G. Margaritis, and S. V. Anastasiadis. Lethe: Cluster-based indexing for secure multi-user search. In *IEEE International Congress on Big Data*, pages 323–330, Anchorage, AK, June 2014.

[25] I. Molloy, N. Li, T. Li, Z. Mao, Q. Wang, and J. Lobo. Evaluating role mining algorithms. In *ACM Symposium on Access Control Models and Technologies*, pages 95–104, Stresa, Italy, June 2009.

[26] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: using cloud storage as attack vector and online slack space. In *USENIX Security Symposium*, pages 65–76, San Fransisco, CA, Aug. 2011.

[27] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *International Conference on Very Large Data Bases*, pages 144–155, Santiago, Chile, Sept. 1994.

[28] H. Pang, J. Shen, and R. Krishnan. Privacy-preserving similarity-based text retrieval. *ACM Transactions on Internet Technology*, pages 4:1–4:39, Feb. 2010.

[29] A. Parker-Wood, C. Strong, E. L. Miller, and D. D. Long. Security aware partitioning for efficient file system search. In *IEEE International Conference on Massive Storage Systems and Technology*, pages 1–14, Incline Village, NV, May 2010.

[30] Z. Pervez, A. A. Awan, A. M. Khattak, S. Lee, and E.-N. Huh. Privacy-aware searching with oblivious term matching for cloud storage. *Journal of Supercomputing*, 63(2):538–560, Feb. 2013.

[31] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *ACM Symposium on Operating Systems Principles*, pages 85–100, Cascais, Portugal, Oct. 2011.

[32] J. Schlegelmilch and U. Steffens. Role mining with ORCA. In *ACM Symposium on Access Control Models and Technologies*, pages 168–176, Stockholm, Sweden, June 2005.

[33] D. Schultz and B. Liskov. IFDB: decentralized information flow control for databases. In *ACM European Conference on Computer Systems*, pages 43–56, Prague, Czech Republic, Apr. 2013.

[34] E. Silva, T. Teixeira, G. Teodoro, and E. Valle. Large-scale distributed locality-sensitive hashing for general metric data. In *International Conference on Similarity Search and Applications*, Los Cabos, Mexico, Oct. 2014.

[35] A. Singh, M. Srivatsa, and L. Liu. Search-as-a-service: Outsourced search over outsourced stor-

age. *ACM Transactions on the Web*, 3(4):13:1–13:33, Sept. 2009.

[36] D. K. Smetters and N. Good. How users use access control. In *Symposium On Usable Privacy and Security*, Mountain View, CA, July 2009.

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

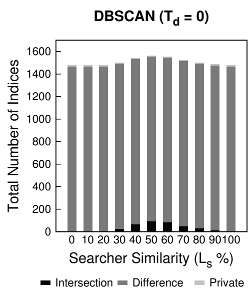
[38] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*, chapter 8. Addison-Wesley, Boston, MA, May 2005.

[39] TREC terabyte track. `http://trec.nist.gov/data/terabyte.html`, 2006. National Institute of Standards and Technology, Gaithersburg, MD.

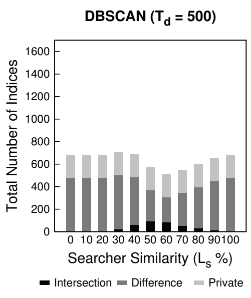
[40] J. Vaidya, V. Atluri, and J. Warner. RoleMiner: mining roles using subset enumeration. In *ACM Conference on Computer and Communications Security*, pages 144–153, Alexandria, VA, Oct. 2006.

[41] S. Zerr, D. Olmedilla, W. Nejdl, and W. Siberski. Zerber^{+R}: top-k retrieval from a confidential index. In *International Conference on Extending Database Technology*, pages 439–449, Saint Petersburg, Russia, Mar. 2009.

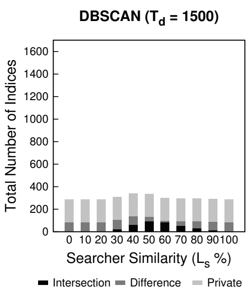
[42] K. Zhang, Z. Zhou, Y. Chen, X. Wang, and Y. Ruan. Sedic: Privacy-aware data intensive computing on hybrid clouds. In *ACM Conference on Computer and Communications Security*, pages 515–525, Chicago, IL, Oct. 2011.



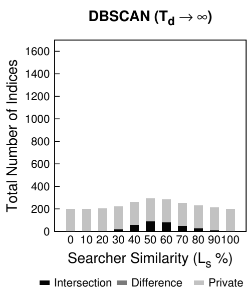
(a)



(b)

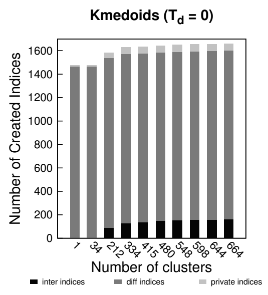


(c)

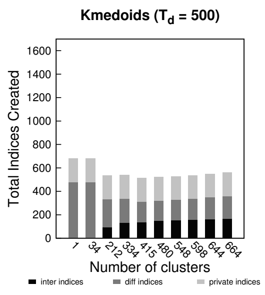


(d)

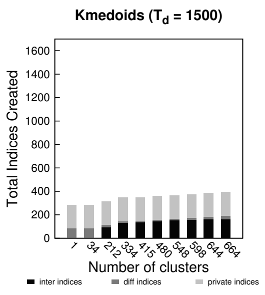
Figure 5: The types of indices created across different T_d and L_s values over DocuShare with DBSCAN. Intersection indices refer to cluster intersections, private indices are dedicated to specific searchers, and difference indices correspond to document families in clusters with empty or non-empty intersection.



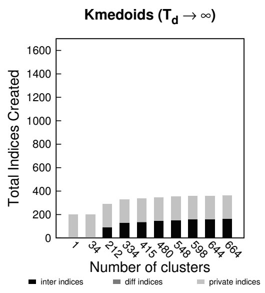
(a)



(b)



(c)



(d)

Figure 6: The types of indices created across different T_d and L_s values over DocuShare with the K-medoids algorithm.

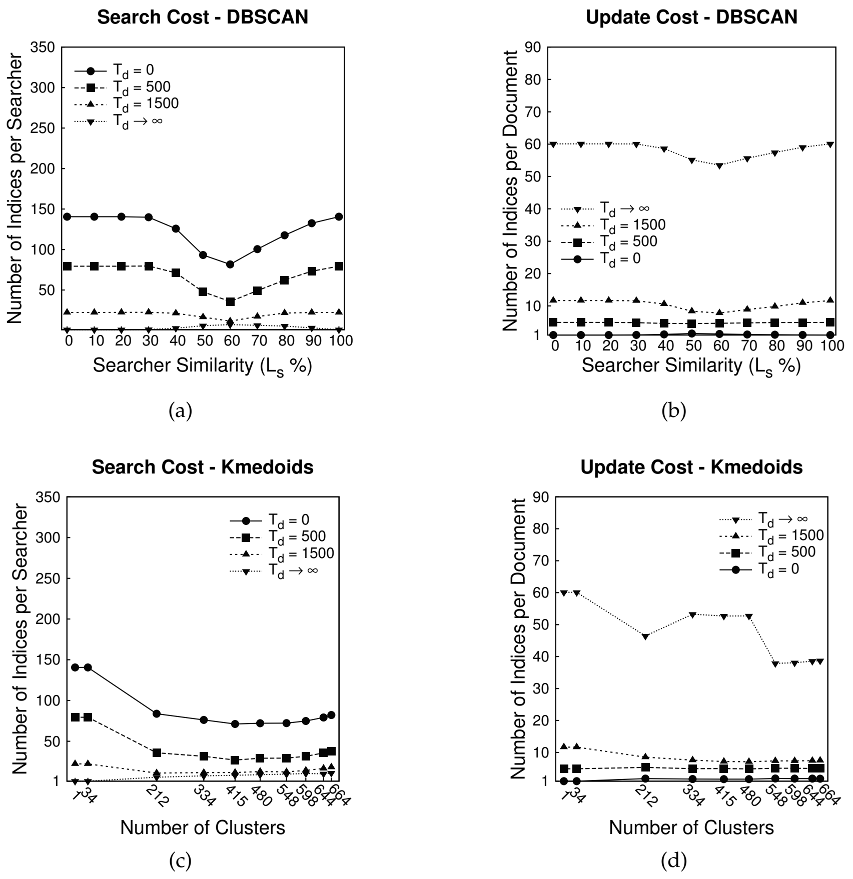
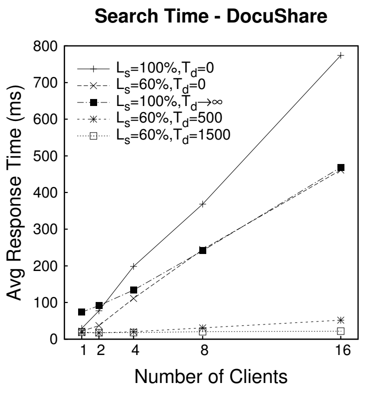
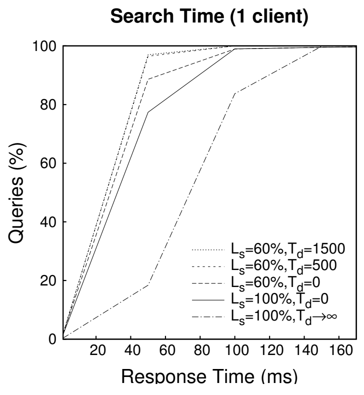


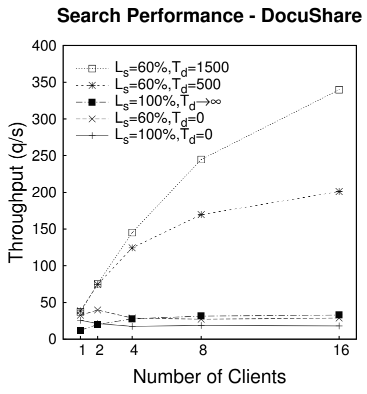
Figure 7: The number of indices per searcher (search cost) and per document (update cost) of the DocuShare dataset for DBSCAN (a,b) and K-medoids (c,d).



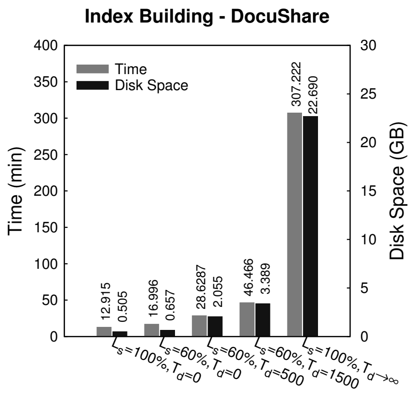
(a)



(b)



(c)



(d)

Figure 8: We show (a) the average response time and (b) the cumulative fraction of queries with different response time along with (c) the query throughput of serving DocuShare over Elasticsearch. (d) We compare the time and storage space of index building by Elasticsearch across different similarity and duplication parameter values.

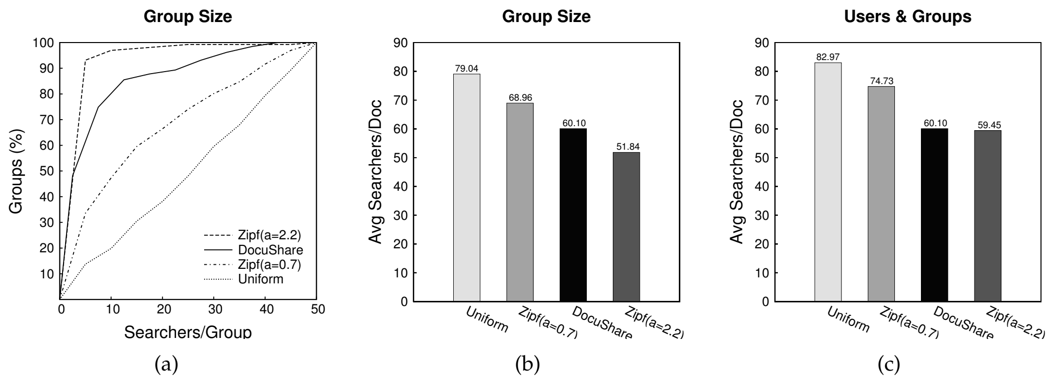


Figure 9: For different distributions of the group size, we illustrate (a) the cumulative fraction of groups containing different numbers of searchers, and (b) the average number of authorized searchers per document. (c) For different distributions of the number of users and groups per document, we show the average number of searchers per document.

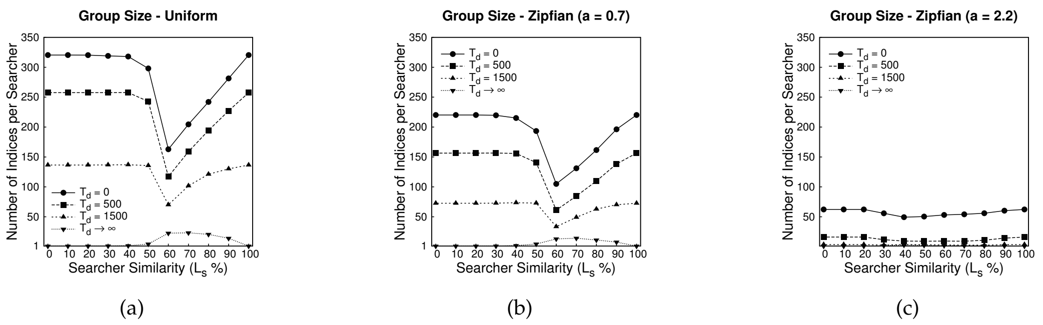


Figure 10: For Zipfian distributions of group size, increasing α reduces the indices per searcher.

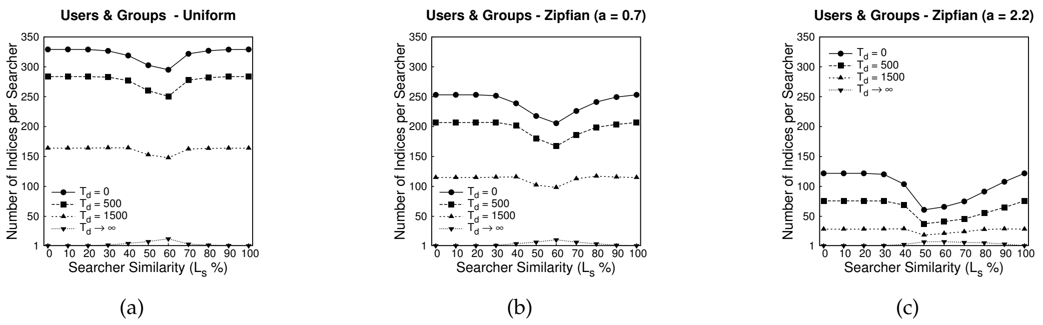
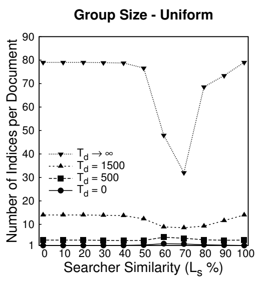
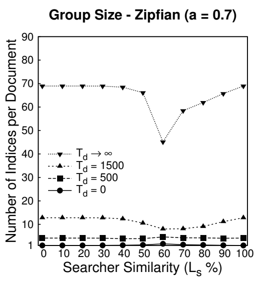


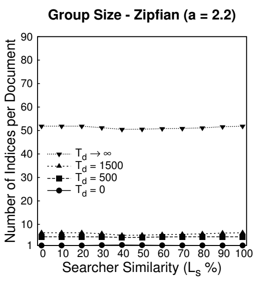
Figure 11: The number of indices per searcher for number of users and groups per document following the Zipfian distribution with α = 0, 0.7 and 2.2.



(a)

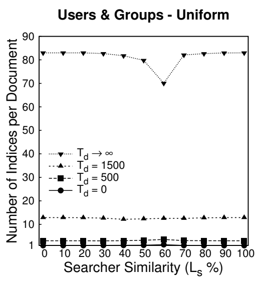


(b)

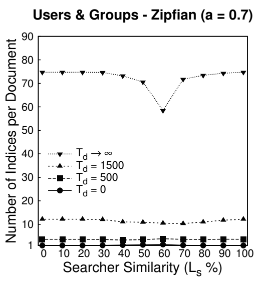


(c)

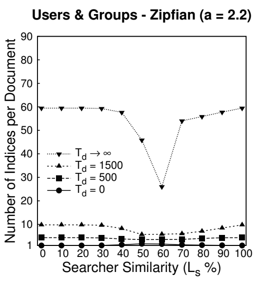
Figure 12: We illustrate the number of indices per document for Zipfian distribution of group size. A higher α value of the distribution reduces the indices per document.



(a)



(b)



(c)

Figure 13: The number of indices per document for Zipfian distribution of the number of users and groups in the permission list of each document, with α = 0, 0.7 and 2.2.