Lethe: Cluster-based Indexing for Secure Multi-User Search

Eirini C. Micheli, Giorgos Margaritis, Stergios V. Anastasiadis
Department of Computer Science and Engineering
University of Ioannina, Greece
{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. With measurements over an open-source distributed search engine, we experimentally confirm the improved search and update performance of particular configurations that we introduce.**

***Keywords*-confidentiality, privacy, access control, data storage, outsourced services, document sharing, search engines**

I. 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 [12]. Storage consolidation increasingly moves sensitive data to public infrastructures, which 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 [15].

An inverted index is the typical indexing structure of keyword search. The stored documents are preprocessed into a posting list per keyword (or term), which provides 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], [22], [19], [5]. 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 [7].

A known secure solution relies on a shared index to serve queries by restricting access to the postings of searchable documents, and filtering out the postings of documents that are inaccessible to the user [7]. In online social networks, recent research applies advanced list-processing operators and cost models to improve secure search efficiency [5]. 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 [22]. The collection ends up indexed by a limited number of indices, and query handling is limited to indices of documents searchable by the querying user. However, the number of indices increases by minor variations in the search permissions of different documents. Although smaller indices can be completely eliminated by replicating their contents to private per-user indices, this approach increases document duplication across the indices and the respective update cost.

In this 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 one 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) skip query-time list filtering by using prebuilt secure indices, and (ii) effectively reduce the number of searched or maintained indices through configurable partial merging of indices for documents with common authorized users. In Sections II and III we present the Lethe indexing workflow and our prototype implementation. In Sections IV and V we show our experimental results and examine previous related work, while in Section VI we summarize our conclusions and plans for future work.

II. 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.

978-1-4799-5057-7/14 $31.00 © 2014 IEEE
DOI 10.1109/BigData.Congress.2014.54

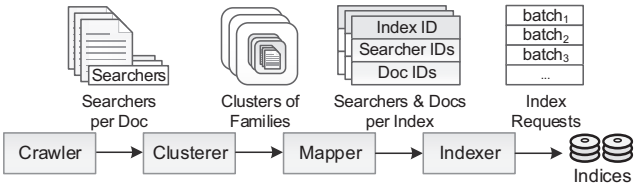

Fig. 1. The four stages of the Lethe workflow.

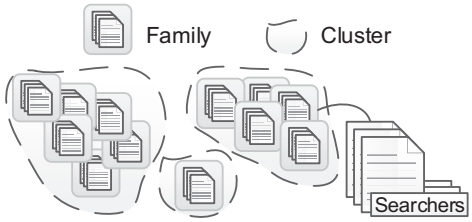

Fig. 2. Document families clustered by searcher similarity L_s.

A. 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 *searchers* of a document the users who are authorized 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:

- **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 the search latency per query as measured through an appropriate metric (e.g., average, high percentile).
- **Indexing Cost** Minimize the document insertion I/O activity and indexing storage space required for the entire collection.

We require that users are authenticated by the system and receive authorization to only search for documents with the necessary access permissions. Accordingly, we build a separate index for each document subset with common access permissions. We examine secure search in multi-user environments without special consideration of encrypted storage. On its own, search with encrypted keywords over encrypted documents does not necessarily hide the search activity and stored content from the storage provider or unauthorized searchers [22], [27].

B. The Lethe workflow

We introduce the Lethe workflow consisting of four basic stages for crawling the document metadata, clustering together the documents with similar searchers, mapping the documents to indices, and generating the indices (Fig. 1).

Crawler In order to realize our goals, we build an appropriate indexing organization based on the document search permissions. Let a text dataset $\mathcal{T} = (D_\mathcal{T}, S_\mathcal{T})$, where $D_\mathcal{T}$ is the set of documents, and $S_\mathcal{T}$ the set of users with search permissions over one or more documents of $D_\mathcal{T}$. First we crawl the names (e.g., paths) and permissions (e.g., allowed searchers) of documents in $\mathcal{T}$, and assign unique identifiers to the members of $D_\mathcal{T}$ and $S_\mathcal{T}$. Then 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 $S_f \subseteq S_\mathcal{T}$. We define as $I_\mathcal{T}$ the set of indices that we build to securely index the $D_\mathcal{T}$.

Clusterer We aim to maintain a single index for the searchers who are common among similar families. Accordingly, we need to identify those families with substantial overlap in their searcher sets. We address this issue as a universal clustering problem over the searcher sets of the families in the entire dataset $\mathcal{T}$ (Fig. 2). We parameterize the clustering method as necessary to assign every family to exactly one cluster, i.e., without omitting any family as noise.

Let the *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 $D_f \subseteq D_\mathcal{T}$. The document set D_c of cluster c is derived from the union of the documents contained across all the families of c, i.e., $D_c = \bigcup_{f \in F_c} D_f$.

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

Let the *cluster intersection* of cluster $c \in C_\mathcal{T}$ be a pair $P_c = (D^i_c, S^i_c)$, with $D^i_c = D_c$, and $S^i_c = \bigcap_{f \in F_c} S_f$ the intersection of searchers in the families of F_c. By family definition, the documents in D^i_c are searchable by all the searchers in S^i_c. If $S^i_c \neq \emptyset$, we dedicate a separate index $I_c \in I_\mathcal{T}$ to the intersection P_c. For every family $f \in F_c$, we also define a *family difference* $P_f = (D^d_f, S^d_f)$, where $D^d_f = D_f$ and $S^d_f = S_f - S^i_c$. Hence, S^d_f corresponds to the searchers of family f not contained in S^i_c of P_c. If $S^d_f \neq \emptyset$, we have to allow the users $u \in S^d_f$ to securely search for documents $d \in D^d_f$.

An extreme approach to address the above P_f search problem is to insert all documents d, $\forall d \in D^d_f$, to all private indices $I_u, \forall\, u \in S^d_f$. However, a difference P_f may contain a relatively large number $|D^d_f|$ of documents searchable by a considerable number $|S^d_f|$ of users. Hence, the above approach

Summary of Symbols

Symbol	*Description*	*Properties*
$C_{\mathcal{T}}$	set of family clusters from dataset $\mathcal{T}$	
D_c	document set of cluster c	$= \bigcup_{f \in F_c} D_f$
D_c^i	document set of cluster intersection P_c	$= D_c$
D_f	document set of family f	$\subseteq D_{\mathcal{T}}$
D_f^d	document set of family difference P_f	$= D_f$
$D_{\mathcal{T}}$	document set of dataset $\mathcal{T}$	
D_u^e	document set of private collection P_u	$\subseteq D_{\mathcal{T}}$
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	
P_u	private collection of searcher u	$= (D_u^e, \{u\})$
P_c	intersection of cluster c	$= (D_c^i, S_c^i)$
P_f	difference of family f	$= (D_f^d, S_f^d)$
R_f^d	duplication product	$= \lvert D_f^d \rvert \cdot \lvert S_f^d \rvert$
S_c^i	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_f^d	searcher set of family difference P_f	$= S_f - S_c^i$
$S_{\mathcal{T}}$	searcher set of dataset $\mathcal{T}$	
T_d	duplication threshold	$\in [0, \infty)$

TABLE I
WE SUMMARIZE THE DESCRIPTION OF SYMBOLS USED IN LETHE.

could end up to 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 $S_f^d \neq \emptyset$. However, this approach runs the risk of generating in the system a large number of indices, each serving a small number of documents and searchers.

We introduce the *duplication product* $R_f^d = |D_f^d| \cdot |S_f^d|$ to approximate[1] the potential document duplication resulting from indexing a family difference P_f. Subsequently, the decision of whether we should create a dedicated index I_f depends on how R_f^d compares to the configurable *duplication threshold* T_d. We assume that $R_f^d < T_d$ implies an affordable cost of inserting the documents $d, \forall d \in D_f^d$, to private indices $I_u, \forall u \in S_f^d$. Instead, $R_f^d \geq T_d$ suggests that devoting a separate index I_f to the difference P_f is preferable.

An optimization that we do not examine further, due to its complexity, is to pursue additional duplication reduction by intersecting the searchers of the differences $P_f, \forall f \in F_c'$, for appropriate $F_c' \subset F_c$ corresponding to cluster c.

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 necessary I/O activity, we separately generate each index through a specification of the documents contained in the index. We have experimentally validated that the alternative approach of specifying the list of indices containing each document leads to higher I/O activity during index creation due to lower storage locality.

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. We summarize the description of the symbols involved in the Lethe workflow at Table I.

III. PROTOTYPE IMPLEMENTATION

Based on the above design, our prototype implementation 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. 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.

Since we do not know in advance the appropriate number of clusters, we use a clustering algorithm that generates this number as output (e.g., DBSCAN) rather than requiring it as input (e.g., K-means) [25]. Within each cluster, the mapper identifies the cluster intersections and family differences. Each intersection or difference is specified through the contained documents and authorized searchers. We assign a dedicated index to each cluster intersection; we also use a dedicated index, or the private indices of the respective searchers, for each family difference according to the duplication threshold T_d. The indexer receives the index specifications from the mapper, and splits each index into document batches. Then it communicates with the search engine to insert the documents of each batch to the respective index after the necessary initialization. Finally, 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 [13]. 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.

[1]For increased accuracy of R_f^d over diverse document sizes, we could replace $|D_f^d|$ with the total number of postings contained in all documents $d \in D_f^d$.

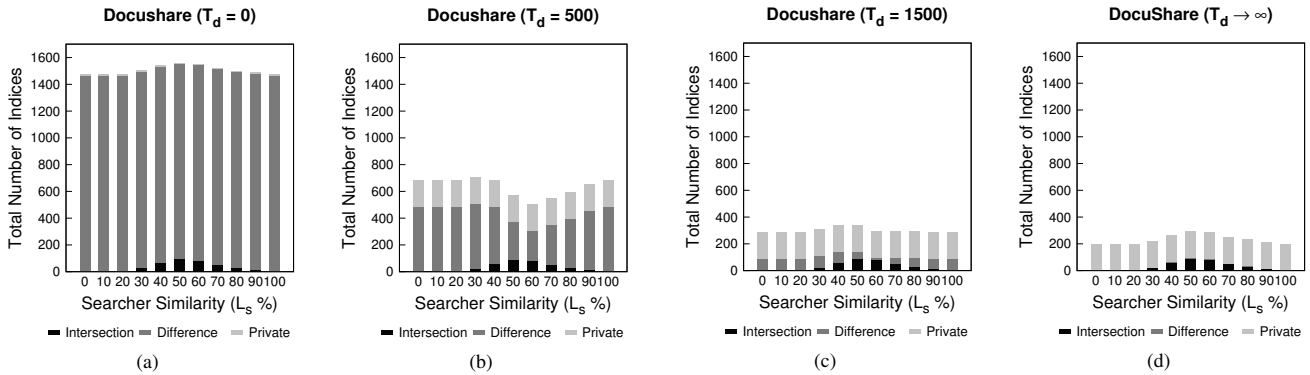

Fig. 3. We consider the types of indices emerging across different T_d and L_s values over DocuShare. Intersection indices refer to cluster intersections, private indices are dedicated to specific searchers, and difference indices correspond to document families in clusters with empty or non-empty intersection.

IV. Experimental Evaluation

We use the published statistics of a real dataset to generate a synthetic workload, and apply a prototype implementation of the Lethe workflow that we developed. Then we measure the indexing performance over a search engine, and examine the search and update cost across different parameters. Finally, we analyze the security and efficiency characteristics of our approach.

A. Document Datasets

We generate a synthetic document collection with searcher lists based on published measurements of an existing dataset (DocuShare [23]). Accordingly, we set the number of users to 200, user groups to 131, documents to 50000, and maximum group size to 50. We specify the sizes of individual groups from the DocuShare statistics, and uniformly pick users as group members.

Based on the DocuShare statistics, we specify the number of users and groups allowed to search each document; then we uniformly assign users and groups to each document. Alternatively, we determine the sizes of user groups based on the Zipfian distribution with α set equal to 0, 0.7, or 2.2, but keep the remaining empirical properties of DocuShare. We evaluate our solution over the first 50000 documents (820MB) of the GOV2 dataset from the TREC Terabyte track [26]. Our query set consists of 50000 standard queries with average number of terms per query equal to 2.8.

We implemented the crawler, clusterer and mapper in C/C++ with STL, and the indexer in Perl (v5.10.1). For clustering we applied the DBSCAN algorithm with MinObjs = 1 and Eps = L_s [25]. 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. 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 respectively. 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.

B. Measurement Results

We first examine in detail the indexing characteristics of alternative clustering configurations. Then we measure the performance of query handling and requirements of index building for the DocuShare dataset over Elasticsearch. Finally, we examine the sensitivity of indexing costs to different parameters and group distributions.

1) Cluster-based Indexing: We applied the Lethe workflow to organize into clusters the document families of DocuShare. From measurements of the created clusters (not shown), we found two extreme cases: one with L_s = 0% leading to 1 cluster of 1475 families and 50000 documents, and another with L_s = 100% leading to 1475 single-family clusters of 33.9 documents on average each. Instead, a moderate choice of L_s = 60% generates 929 clusters with 53.8 documents per cluster.

In Fig. 3, we show 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 ∞ in Figures 3a-d, the total number of indices drops considerably. Indeed, the maximum number of indices decreases from 1561 at T_d = 0 (L_s = 50%), to 703 at T_d = 500 (L_s = 30%), 337 at T_d = 1500 (L_s = 40%), and 292 at $T_d \to \infty$ (L_s = 50%). From additional measurements (not shown), we found that most of the difference indices at $T_d \leq 1500$ correspond to families contained in clusters with empty intersection. Higher values of T_d suppress the creation of difference indices, and let more documents be replicated across the private indices. Therefore, setting $T_d \to \infty$ minimizes the total number of indices through document replication.

However, ideally 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 lets the mapper specify a moderate number of 298 indices: 84 shared indices for cluster intersections, 14 indices for specific families, and 200 private indices for individual searchers. We next examine the effect of different indexing configurations to the actual search and build performance of the search engine.

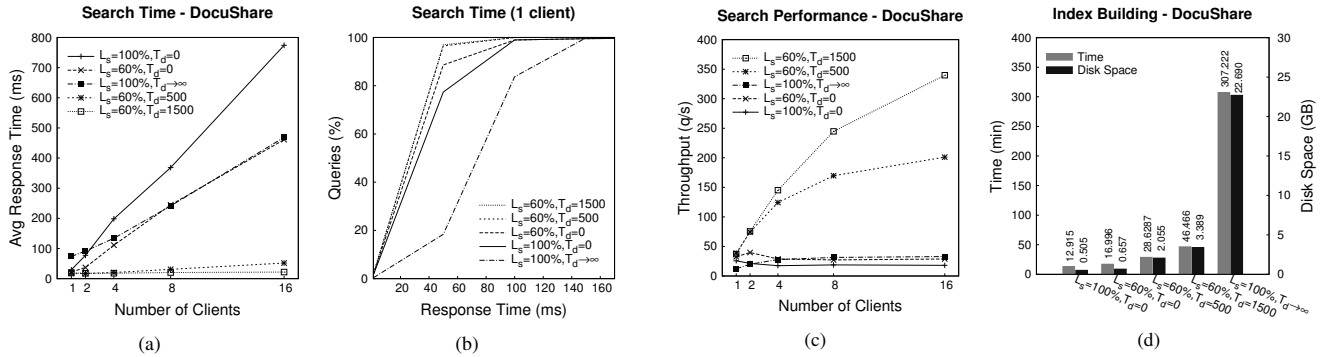

Fig. 4. We show (a) the average response time and (b) the cumulative fraction of queries with different response time, (c) the query throughput, and (d) the time and storage space required for index building of DocuShare over Elasticsearch.

2) Elasticsearch: In Fig. 4, we experimentally consider indexing the DocuShare dataset using five configurations with L_s = 60% or 100%, and T_d = 0, 500, 1500 or $T_d \to \infty$. 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. 4a, 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 slightly 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 3b,c), leading to low query response time. On the contrary, T_d = 0 creates a large number of indices (Fig. 3a); as a result, it causes some queries to take up to several seconds, and increases the measured average time.

In Fig 4b, 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 responding 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 case (100%, ∞) to excessive document duplication and the resulting required storage space, which does not allow the indices to simultaneously fit in memory (further demonstrated by Fig 4d explained below).

We additionally measure the query throughput for 1 to 16 clients in Fig. 4c. We notice that throughput rises by a factor of 9.2, from 37.0q/s to 339.8q/s, in case (60%, 1500), and it reaches 201.1q/s in curve (60%, 500). However, 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.

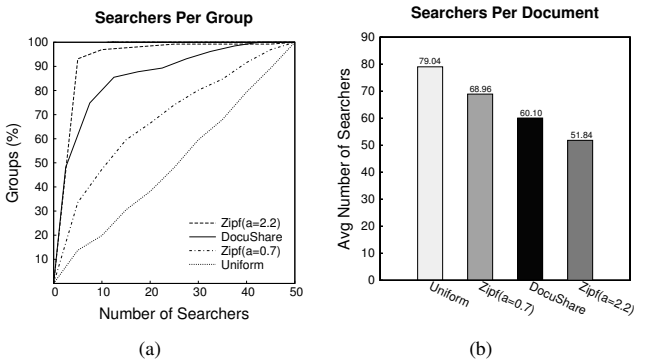

Fig. 5. For different distributions of group size, we illustrate (a) the cumulative fraction of groups containing different numbers of searchers, and (b) the average number of authorized searchers per document.

We illustrate the resource requirements of index building in Fig 4d. 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. 3).

3) Distribution of Group Size: We explore the sensitivity of index clustering to group membership by considering different distributions of group size. In Fig. 5a, we illustrate the cumulative fraction of groups corresponding to different numbers of searchers. Three curves refer to variations of the Zipfian distribution with α set to 0 (uniform), 0.7, or 2.2, and another curve refers to the DocuShare empirical statistics. The plot illustrates that low α values equally distribute the sizes across the groups, and higher α values narrow down the larger sizes to fewer groups.

This is further demonstrated in Fig. 5b, from which it follows that the closer to uniform (α = 0) the distribution gets, the higher the number of searchers per document becomes. As

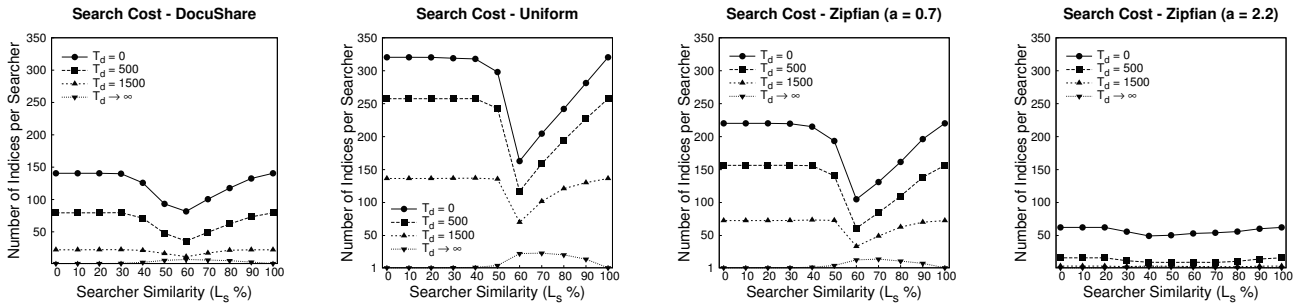

Fig. 6. We examine the number of indices per searcher in the DocuShare dataset, and three variations of it with theoretical distributions of group size. Increasing α reduces the indices per searcher. The DocuShare empirical distribution effectively lies between α=0.7 and 2.2.

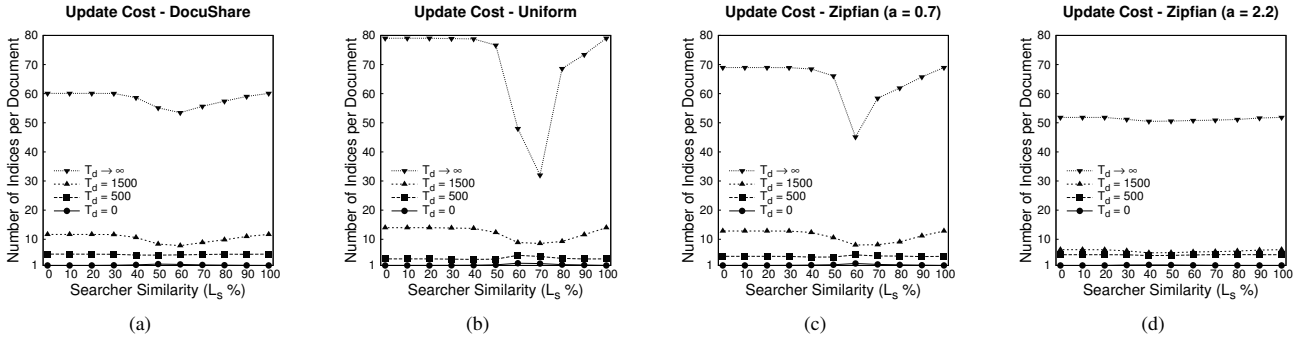

Fig. 7. We illustrate the number of indices per document for the DocuShare dataset and three theoretical distribution of group size. A higher α value in the Zipfian distribution tends to reduce the indices per document. The empirical distribution of DocuShare effectively lies between α = 0.7 and 2.2.

a result, we anticipate that lower α values require more indices per searcher or per document to securely handle the queries.

4) Search Cost: The search cost depends on the number of indices per searcher, which determines the number of document lists merged for the search result. In Fig. 6a, we examine different values of L_s and T_d over DocuShare. The indices per searcher vary in the range 35.8 - 79.6 at T_d = 500, and 11.6 - 22.5 at T_d = 1500.

Setting L_s = 0% or 100% usually maximizes the indices per searcher, because it only permits a limited amount of index sharing within a cluster of multiple diverse families, or a single family. On the contrary, setting L_s = 60% usually minimizes the indices per searcher due to index sharing within cluster intersections and family differences. However, setting $T_d \rightarrow \infty$ suppresses the index sharing of family differences, and inverses the effect of L_s: setting L_s = 0% or 100% actually minimizes the number of indices per searcher.

In Figures 6b-d we further examine the indices per searcher for Zipfian group sizes of α = 0, 0.7, or 2.2. At increasing α, the indices per searcher decreases because more groups have smaller size (Fig. 5). Therefore, a searcher belongs to fewer groups, and a smaller number of indices can index the documents of each searcher. For different L_s and T_d values, we notice the same trends already observed in the original DocuShare dataset, i.e., L_s = 60% reduces the indices per searcher for $T_d \neq \infty$. Overall, the impact of L_s diminishes as α increases, because there is less similarity among the document searchers, and less opportunity for index sharing.

5) Update Cost: 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. 7a, we examine the sensitivity of the update cost to L_s and T_d over DocuShare. At L_s = 60%, setting T_d = 1500 or $T_d \rightarrow \infty$ minimizes the number of indices per document to 7.9 and 53.5, respectively. Essentially, higher T_d values discourage the creation of shared indices dedicated to family differences, and replicate more documents across the private indices. Instead, the curves remain almost flat when T_d = 0 or 500, and the number of indices drops to about 1 or 5, respectively.

In Figures 7b-d we further consider the sensitivity of the indices per document to the distribution of the group size. Increasing α reduces the maximum number of indices per document from 79.0 (α = 0), to 69.0 (α = 0.7) and 51.8 (α = 2.2). This behavior follows from the decreasing group sizes appearing at higher α, and the resulting lower document duplication across the private indices. Within each plot, there is a low point reached around L_s = 60% for T_d = 1500 or $\rightarrow \infty$, and a respective peak at T_d = 0 or 500. As with the search cost, the number of indices per document is less sensitive to L_s in Zipfian distributions of higher α.

If we combine the observations on the search and update cost, we notice diverse effects across the L_s and T_d param-

eters. In particular, L_s leads the two costs to low or peak values depending on T_d. Instead, as T_d grows, it consistently decreases the search cost, but makes more costly the update. One reasonable choice is to set L_s = 60% and T_d = 1500, because we combine 7.9 indices per searcher with 11.6 indices per document in DocuShare. Instead, if we reduce T_d to 500 in DocuShare, the search cost increases by a factor of 3 (from 11.7 to 35.8), and the corresponding update cost only drops by 41% (7.9 to 4.7).

More generally, the values of L_s and T_d should lead to the appropriate balance of indices per searcher and per document depending on the operation characteristics of the workload and the optimization objectives of the service provider.

C. Analysis of Results

Our experiments provide strong evidence for an improved method to achieve efficient and secure keyword indexing. The method is secure because a query can only use indices of documents that the searcher is permitted to access [7]. The method is also efficient for several reasons.

First, we guarantee that the result returned by an index does not require any filtering to remove documents inaccessible to the searcher. We only require to merge the results from multiple indices for ranking purposes, as is typically already done by parallel or distributed search engines. Thus, we avoid the extra query-time overhead for list processing required by previous secure methods [5].

Second, the clustering of document families allows the service of common searchers in the cluster intersection with a single index. Thus, we reduce the average number of indices per searcher, which translates into smaller number of result lists to be generated and merged during query handling. To the best of our knowledge, Lethe represents the first application of index clustering to achieve efficiency in secure keyword search.

Third, the control of indexing duplication through the threshold T_d prevents the insertion of the same document to an excessive number of multiple private indices, which was previously required [22]. Instead, we create extra shared indices whenever the number of documents and their common searchers justify the additional indexing cost.

V. 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.

Secure Indexing Büttcher and Clarke examined relevance-ranking search on the vector space model [7]. 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 solution leads to performance slowdown in some cases.

Singh et al. logically organize the filesystem into access-control barrels, which are sets of files with identical access privileges [22]. 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 reduce the document duplication across different indices.

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 [2]. A designated host constructs a privacy-preserving index from the bit vectors to identify the privacy groups that match a query. Pang et al. safeguard the content and result of user queries over the vector space model [18]. 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. Zerr et al. allow the inclusion of transformed relevance scores in an untrusted server without disclosing information about the indexed data [27].

In a preliminary work, we previously introduced the Lethe workflow to create secure configurations of shared inverted indices [16]. The present paper substantially expands our prior work most notably (i) through extensive experimental study of the characteristics of different indexing configuration across a range of empirical and theoretical datasets, and (ii) with search and update performance measurements over an open-source distributed search engine.

Secure search of big data returns the records matching a query without revealing the query or its results to the provider [11]. 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 [3].

Encrypted Storage Song et al. describe techniques to securely search remote documents maintained in encrypted form [24]. 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 [8]. 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 [1]. Thus, a client only submits keyword hashes to search for documents.

CryptDB supports keyword search over individually encrypted words of a text column in a relational database [21]. Pervez et al. assume that both files and inverted indices are stored in encrypted form at the cloud [20]. 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 [28]. PRISM transforms the problem of keyword search over encrypted files into privacy-preserving map and reduce tasks [6]. Other related research applies data possession proofs to prevent file hiding into the cloud by unauthorized clients [17]. Therefore, the above research focuses on indexing or processing of encrypted 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 [4]. Alternative cost models are examined to optimally include specific friends in the author list of each user, and the HeapUnion operator is introduced to efficiently process multiple lists of document identifiers [5].

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 [10]. 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 [9]. Hails provides data-flow confinement at the client and server so that mutually-untrusted web applications can interact safely [14]. These are general issues of access control in social networks beyond our study scope.

VI. 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 measurements over a distributed search engine confirm that our indexing organization achieves higher search and update performance. In our future work, we plan to analytically model the operation costs, and experimentally explore alternative clustering methods over a broader dataset collection from collaborative environments, cloud storage and social networks.

VII. Acknowledgment

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] 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.

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

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

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

[5] T. A. Bjørklund, M. Götz, J. Gehrke, and N. Grimsmo. Workload-aware indexing for keyword search in social networks. In *ACM Intl. Conf. Information and Knowledge Management (CIKM)*, pages 535–544, Glasgow, UK, Oct. 2011.

[6] 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.

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

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

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

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

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

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

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

[14] 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 Symp. Operating Systems Design and Implementation (OSDI)*, pages 47–60, Hollywood, CA, Oct. 2012.

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

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

[17] 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 Symp.*, pages 65–76, San Fransisco, CA, Aug. 2011.

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

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

[20] 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.

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

[22] A. Singh, M. Srivatsa, and L. Liu. Search-as-a-service: Outsourced search over outsourced storage. *ACM Trans. Web*, 3(4):13:1–13:33, Sept. 2009.

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

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

[25] P.-N. Tan, M. Steinbach, and V. Kumar. *Data Mining*, chapter 8. Addison-Wesley, May 2005.

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

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

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