University of California
Santa Barbara

Efficient Cloud Backup and Private Search

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Daniel Michael Agun

Committee in charge:

Professor Tao Yang, Co-Chair
Professor Stefano Tessaro, Co-Chair
Professor Rich Wolski

June 2019

The Dissertation of Daniel Michael Agun is approved.

Professor Rich Wolski

Professor Stefano Tessaro, Committee Co-Chair

Professor Tao Yang, Committee Co-Chair

March 2019

Efficient Cloud Backup and Private Search

Acknowledgements

First of all I want to thank my Comittee Tao Yang, Stefano Tessaro, and Rich Wolski, especially Tao for his consistent patenience and mentorship over my years here.

Thank you, to my fellow lab members Jinjin Shao, Shiyu Ji, and Xin Jin for their valuable collaboration and help with ideas, implementations, and evaluations. Also, I want to thank to Olaoluwa Osuntokun for help with prototype implementations, and others who have helped in my research.

Finally I to thank my amazing wife Christina Agun, who is always there to help and support me, as well as my parents for their love and support.

This work is supported in part by NSF IIS-1528041 and IIS-1118106. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Curriculum Vitæ

Daniel Michael Agun

EDUCATION

2019	*PhD,* Computer Science (expected), University of California, Santa Barbara
2012	*Bachelor of Science,* Computer Science, Gonzaga University

PUBLICATIONS

Shiyu Ji, Daniel Agun, Jinjin Shao, Tao Yang, "Privacy and Efficiency Tradeoffs for Multiword Top K Search with Linear Additive Rank Scoring". *The Web Conference (TWC) 2018.*

Shiyu Ji, Daniel Agun, Jinjin Shao, Tao Yang, "Privacy-aware Ranking with Tree Ensembles on the Cloud". *SIGIR 2018.*

Daniel M. Agun,Wei Zhang, Tao Yang, "Low-Profile Source-side Deduplication for Virtual Machine Backup". *HotCloud 2016.*

Wei Zhang, Daniel M Agun, Tao Yang, Rich Wolski, Hong Tang, "VM-Centric Snapshot Deduplication for Cloud Data Backup". *Mass Storage Systems and Technologies (MSST) 2015.*

Daniel Agun & Hiranya Jayathilaka. "Extending Modern PaaS Clouds with BSP to Execute Legacy MPI Applications". *Poster at SOCC 2013.*

Agun, M., & Bowers, S. (2011). "Approaches for Implementing Persistent Queues within Data-Intensive Scientific Workflows". *Services (SERVICES), 2011 IEEE World Congress on* (Vol. 1, pp. 200207).

P. Missier, B. Ludscher, S. Dey, M. Wang, T. McPhillips, S. Bowers, M. Agun, I. Altintas (2012). "Golden Trail: Retrieving the Data History that Matters from a Comprehensive Provenance Repository". *International Journal of Digital Curation* (Vol. 1, pp. 139-150).

Abstract

Efficient Cloud Backup and Private Search

by

Daniel Michael Agun

As organizations and companies are increasingly offloading data and computation to the cloud to reduce infrastructure administration, data volume keeps growing and new services and algorithms are needed to meet increasing demands for both storage capacity and privacy.

The first part of my thesis will address cloud data backup. Organizations and companies often backup and archive high volumes of binary and text datasets for fault tolerance, internal investigation, and electronic discovery. Source-side deduplication has an advantage to avoid or minimize duplicated data transmitted over the network, however it demands more computing resource to perform extensive fingerprint comparison which would otherwise be available for primary services at the source. For data stored in the cloud, users need efficient, scalable services for searching these files. In the first part of this thesis, I will cover the key components of existing solutions for large-scale backup storage in the cloud. I will go into detail on how deduplication is important to large scale backup systems, and review some ongoing work. I will also detail my contributions in this area towards low-profile source-side deduplication.

The second part of my thesis addresses an open problem for efficient private document search on data hosted on the cloud. As sensitive information is increasingly centralized into the cloud, for the protection of data privacy, such data is often encrypted, which makes effective data indexing and search a very challenging task. To overcome the challenges of querying encrypted datasets, searchable encryption schemes allow users

to securely search over encrypted data through keywords. No existing solutions for efficient ranking which involves complex arithmetic computation in feature composition and scoring currently exist, and without relevant ranking of search results queries over very large datasets which may return many results can be impractical. In the second part of my thesis I will review existing work on private search and introduce our ongoing and published work for this open problem, focusing on how to make private search practical and scalable for large datasets.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement and Motivation

Companies are increasingly moving storage resources to cloud-hosted datacenters. As companies reduce their self-hosted infrastructure and server resources, new solutions are needed to typical big data problems to facilitate this model. At the cloud host, techniques are needed to support VM snapshots for huge numbers of VMs at high scale. Deduplication helps reduce this storage cost, but converged-cloud suitable techniques are needed to efficiently eliminate duplicate data. For the cloud-hosted company, new solutions are needed for sensitive data to be moved to the cloud so traditional features like rich text search can be supported. Privacy-preserving search addresses this issue, but existing techniques either have large privacy leakage or are too slow to support larger-scale datasets of millions of documents. This thesis research is focused on addressing some of the challenges with moving to the cloud.

Existing deduplication approaches follow a scale-up approach, and even when scaling out tend to keep machine numbers small and use high-end machines. In a converged cloud architecture, scale-out deduplication techniques suitable for commodity hardware

is needed. Virtual disks may also be continuously backed up to reduce backup needs to handle hardware failures, but regular snapshots is a complimentary approach which can help recover from hardware failures, but also software failures that continuous backups don't address.

The first part of this work is a low-profile scale-out solution to virtual disk snapshot deduplication suitable for a converged datacenter wanting to snapshot every single VM every day. We propose an asynchronous source-side deduplication scheme which distributes the global index in small partitions and orchestrates multi-round duplicate detection in batches to keep resource requirements low while achieving reasonable performance and high deduplication rates. We also use an approximate deletion scheme to remove most unreferenced data after deletions in under 1 minute. Our evaluation shows that this approach scales well even with highly skewed workload distributions and validates the approximate deletion algorithm to make full garbage collection runs infrequent.

As companies are moving storage resources to the cloud, they still want to have the features they can support with local datacenters. For sensitive data, there are now extra considerations since the user of the cloud does not have physical control over the machine, and so has to trust the cloud provider not to abuse their control over the machine. For most purposes this may be fine and can be managed with legal agreements. However, for highly sensitive data such as legal documents or corporate secrets legal agreements alone may not be enough and there need to be additional safeguards in place to protect company/user information from a cloud provider that has suffered a data breach (or a malicious one).

To store documents, we can use traditional approaches like encryption to protect data at rest, but if the cloud-VM decrypts the data, then a dishonest physical server could read the data from RAM (or if keys are stored in the cloud-VM it could directly decrypt), so in the case where we have private information stored in a cloud-VM, it needs

to be transferred out of the cloud and decrypted using keys not directly stored in the cloud to be able to guarantee privacy. Other work investigates how to store and retrieve documents securely in the cloud [1].

For large scale systems containing millions of documents, search features are often needed to find documents of interest. The search index itself now contains much information about the contents of the documents, up to containing the entire document text in inverted-index form. Using these traditional IR techniques a dishonest server could learn about the structure and contents of the encrypted private documents without ever seeing the documents themselves.

To solve this problem, we need private search techniques such as searchable encryption[2, 3].

The second part of the thesis work addresses how to support queries over sensitive user-owned data using cloud-hosted servers. The key difference compared to previous work is efficiently supporting rich boolean queries and supporting document ranking with server-side partial filtering. I propose a new scheme which supports privacy-preserving ranking of documents against private boolean queries, which can support two broad levels of privacy: server scored and client ranked with no privacy leakage due to ranking, and server scored with partial server-side filtering and client ranking for the final top K. Most previous work on ranking has high leakage or is limited to hundreds to thousands of documents, where I want to support millions of documents. Our evaluation validates the efficiency of this approach for a modest-sized cloud dataset and shows that the higher level of privacy is the same as previous private search research.

1.2 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 gives the background for the thesis problems. Chapter 3 presents a low-profile distributed approach for batched deduplication in large cloud datacenters. Chapter 4 proposes a novel private search scheme with ranking for searching over cloud hosted data. Chapter 5 concludes this dissertation and discusses some potential future work.

Chapter 2

Background and Related Work

2.1 Cloud Backup

In a virtualized cloud environment such as ones provided by Amazon EC2[4] and Alibaba Aliyun[5], each instance of a guest operating system runs on a virtual machine, accessing virtual hard disks represented as virtual disk image files in the host operating system. Because these image files are stored as regular files from the external point of view, backing up VM's data is mainly done by taking snapshots of virtual disk images. A snapshot preserves the data of a VM's file system at a specific point in time.

Periodic backup of virtual machine (VM) snapshots is important for data retention and fault recovery, but that demands high storage cost and network traffic. Backup requests may be initiated by users periodically while some vendors even provide automatic backup. The challenge is the sheer size of backup data. Since backup data is highly redundant, source-side deduplication has an advantage to avoid or minimize duplicated data transmitted over the network. This can be accomplished using a dirty-bit version change detection [6, 7]. Additional target-side deduplication can be deployed with fingerprint-based techniques [8, 9] using backup products such as ones from NetApp [10]

or EMC [11].

Even with dirty-bit version change tracking, there is still a large amount of undetected duplicates transmitted from the primary cloud storage to a backup storage. Our work is targeted at architectures where a backup service can be collocated with other cloud services in order to perform complete source-side deduplication. Collocated backup is also feasible in a *converged* architecture that supports the functionality of infrastructure-as-a-service. In such an architecture, commodity datacenter components are deployed and processing and storage for multiple services are shared. Examples of such an architecture can be seen at cloud providers at Google and Alibaba, and at Nutanix [12].

2.2 Private Search

As sensitive information is increasingly stored on the cloud, preserving privacy is a critical factor for users to adopt cloud-based information services including keyword search. A cloud server is often considered as *honest-but-curious*: namely such a server honestly executes protocol specification and hosted programs, but it may observe and infer the private information of a client during execution or by inspecting hosted data. To deal with such a server, searchable encryption [2, 3] can be used to conduct privacy-preserving server-side query matching with single keyword [13, 14], conjunctive multi-words using the OXT protocol [15, 16, 17], or disjunctive multiwords [18]. The studies including OXT do not support ranking, and efficient and secure top K ranking is an open research problem because of the challenge in achieving both.

The main challenge to perform server-side privacy-preserving top K ranking is that advanced ranking involves arithmetic computation based on raw features (defined in Section 4.2.1) and hiding feature information through encryption prevents the server from performing effective scoring and result comparison. On the other hand, unencrypted

feature values can lend themselves to privacy attacks [19, 20]. Homomorphic encryption [21, 22] is one idea offered to secure data while letting the server perform arithmetic calculations without decrypting the underlying data. The server still cannot rank encrypted scores however and such a scheme is not computationally feasible when many numbers are involved, because each addition or multiplication is extremely slow.

Another challenge is that advanced ranking considers a variety of features (e.g. [23, 24, 25, 26, 27, 28]) and feature vectors are often sparse with many zero values. Space usage can grow explosively if zeros are explicitly stored. Using a compact data structure representation without a proper defense can leak statistic information about index, which may lead to leakage-abuse attacks [19, 20, 29].

Previous work on similarity-based secure ranking [30, 31, 32] can support standard TFIDF ranking. Those works do not address the efficiency challenge however, and are limited to searching hundreds or low thousands of documents. It appears implausible to develop a completely secure ranking scheme without resorting to heavyweight cryptography such as functional encryption [33, 34, 21]. With a practical restriction towards fast response time, a server-hosted algorithm must deploy a compromised lightweight scheme to compute ranking scores and select results. Later in Chapter 4, I will address these challenges and propose a system which supports private intersection and ranking while searching millions of documents.

Chapter 3

Low-Profile Deduplication

Source-side deduplication [7, 35, 36, 37] eliminates duplicates before backup data is transmitted to a secondary storage, which saves network bandwidth significantly; however its resource usage can impact other co-located cloud services. It is memory-intensive to compare a large number of fingerprints and identify duplicates, even with optimization or approximation techniques developed [38, 39, 40, 41]. When deleting unused snapshots, a grouped mark-and-sweep approach [38] is effective while it still carries a significant cost, especially in a distributed setting. Since backup is a secondary service, cloud providers normally do not want it to contend for resources with other collocated primary services. The focus of this work is to develop a low-profile approximated scheme for aggressive source-side deduplication while using a minimal amount of resource.

Most previous work on deduplication uses an inline approach which optimizes the performance of each individual chunk backup operation. Our work considers a backup service with source-side deduplication for converged cloud architectures that is co-located with user VMs, sharing the cloud compute, network, and storage resource with them.

The main contribution of this work is a design and implementation of a backup service to integrate the VM-centric techniques while minimizing the resource impact on

other collocated cloud services and a low-profile approach for collaborative source-side deduplication with *asynchronous* duplicate detection and message exchange. It extends our preliminary simulation study [42] to restrict the scope of cross-VM deduplication and simplify the deduplication process by separating popular data chunks. We term this approach *VM-centric* because the deduplication algorithm considers VM boundaries in making its decisions as opposed to treating all blocks as being equivalent within the storage system. To conduct inter-VM deduplication, we let all machines collaborate in an asynchronous model. Each machine is responsible of duplicate detection for a range of fingerprints and it collects duplicate detection requests, and processes them partition by partition periodically and asynchronously.

This work is presented as follows. Section 3.1 discusses related approaches and motivates our design. Section 3.1 presents a VM-centric approach. Section 3.2 is a collaborative deduplication scheme for inter-VM data processing. Section 3.4 discusses the fast approximate deletion scheme and leakage repair. Section 3.5 presents the evaluation results, and finally, Section 3.6 overviews the contributions of this work.

3.1 Problem Definition

Figure 3.1 illustrates a cloud cluster architecture targeted by our source-side deduplication scheme. Each commodity server in the cluster hosts a number of virtual machines. A backup service, co-located with other cloud services on these servers, performs source-side duplicate detection collaboratively, manages metadata, and backs up data. The backup data can be transmitted to a separate secondary storage or to a distributed storage system in this cluster. A dirty-bit method can be used to detect what has been been changed from the previous version [6, 7]. While VM data is stored in a distributed filesystem, each machine typically caches actively used virtual images; the snapshotting

of the backup service can exploit the cached copy so that reading modified backup data can be conducted locally.

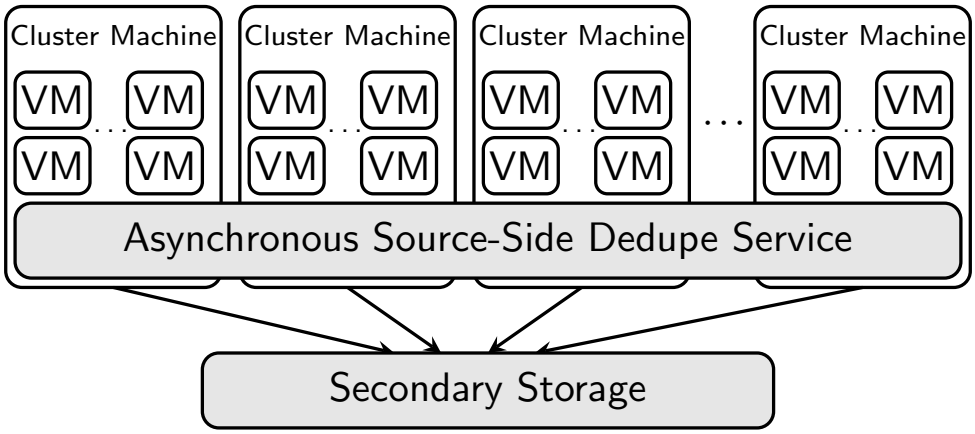


Figure 3.1: Collaborative source-side data backup running on a cloud cluster collocated with other services.

File backup systems have been developed to use fingerprints generated for data "chunks" to identify duplicate content [8, 43]. Source-side deduplication is studied at a chunk or file level [7, 35] for client data backup and can be integrated with global deduplication among multiple clients [36]. In this context, the client side is the source-side and the client-side computing resource restriction is not a primary concern. Our work considers the source-side deduplication in a cloud cluster environment before data is transferred to a backup storage and the computing resource constraint at the cluster side is a major concern.

It is expensive to compare a large number of fingerprints so a number of techniques have been proposed to improve duplicate identification. For example, the Data Domain method [9] uses an in-memory Bloom filter to identify non-duplicates and a prefetching cache for potential duplicates that might hit in the future. Additional inline deduplication and approximation techniques are studied in the previous work [40, 44, 45, 42].

Even with these optimization and approximation techniques, resource usage such as memory for deduplication is still extensive for a shared cloud cluster. For example, in

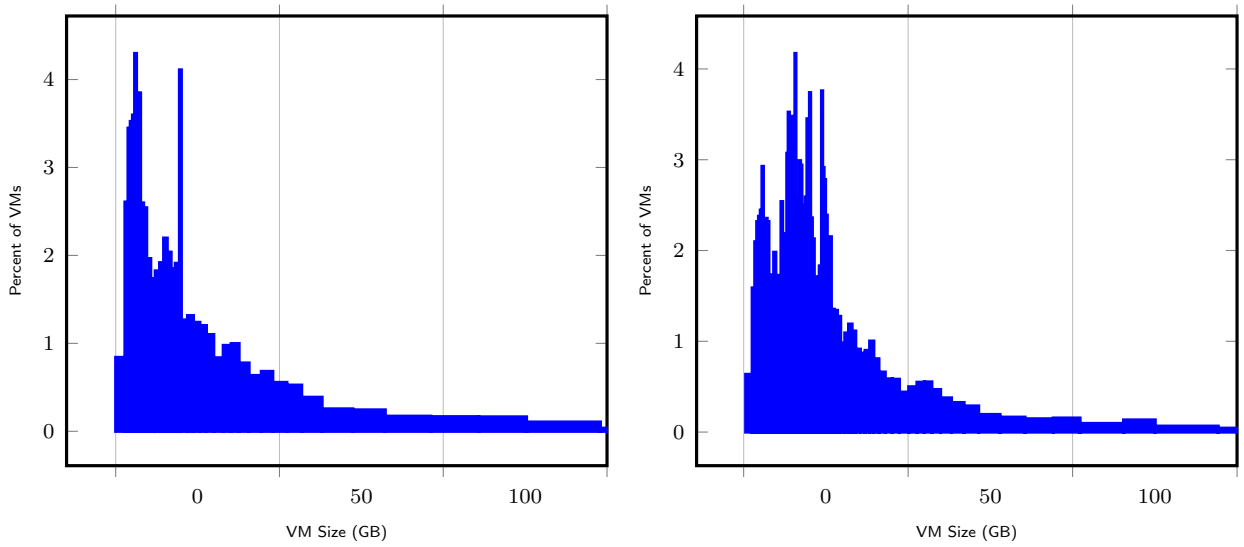
the experiments discussed in Section 3.5, the raw snapshot data has a size of 1,000TB on 100 physical machines that host VMs, cross-machine fingerprint comparison using Bloom filter and approximated routing [9, 39] still needs several gigabytes of memory per machine. This can impact other primary cloud services sharing the same computing resource. Our objective is to have an approximation scheme which uses no more than a few hundred megabytes of memory during normal operation. Thus the desired ratio of raw data to memory ratio is from 100K:1 to 30K:1. Our proposed scheme achieves 85K:1 and this ratio is about the same as the number of machines increases.

Index sampling with a prefetch cache [38] is proposed for efficient single-machine deduplication. This scheme uses 25GB memory per 500TB of raw data and thus the raw data to memory ratio is 20K:1. The scheme proposed in this work uses three times less memory. We have not incorporated this index sampling technique because it is difficult to extend for a distributed architecture. To use a distributed memory version of the sampled index, every deduplication request may access a remote machine for index lookup and the overall overhead of access latency for all requests is significant.

Our previous work has proposed a low-cost parallel deduplication that processes backup requests in multiple synchronized stages [46]. This work performs duplicate detection before actual data backup and all physical machines go through the following synchronized processing stages. 1) Dirty segment scanning 2) Duplicate detection message exchange. 3) Distributed fingerprint comparison. 4) Output of duplicate detection results. 5) Non-duplicate backup. The deduplication is conducted in the data chunk level. Once a data chuck is written to the backup storage, a storage reference is returned. For another chunk with the same fingerprint, this reference can be used so that metadata of duplicates points to the same backup storage location.

The key advantage is that the above parallel scheme is simple with very low resource usage. The primary disadvantage is that the multi-stage synchronized process among

machines is vulnerable when some participating machine is abnormally slow. The slowness can be caused by some internal failure, load imbalance, or uneven VM data size distribution. For example, based on data from cloud provider Alibaba.com, we have found that VM size distribution of their production clusters is highly skewed. Figure 3.2 shows two such clusters, with the left and right clusters having 4200 and 8000 VMs respectively, and skews (Max/Average VM size) of 20 and 45. Then small VMs must wait a very long time for the big VMs to finish. To address the slow data processing with a synchronous approach, we propose an asynchronous solution in next section.



(a) (b)

Figure 3.2: Skewed distribution of VM sizes in two representative production clusters

3.1.1 Where to Deduplicate

The first question to be answered is where to perform deduplication. There are only 3 options, so lets consider each of them.

1. Target-side

2. Source-side

3. somewhere between

Target-side deduplication performs deduplication at the backup storage. This has the advantage of not requiring any modifications to the primary storage system. Data to be backed up can be sent to the backup system without regard for when/how or even if deduplication occurs. Another advantage is that we have a global view of all backup data, so cross-source backups may be easier. The downside to this approach is that we must send all snapshot data over the network, even though we already know most of it will be duplicate data.

Source-side deduplication is done at/near the primary storage. There is greater opportunity for network savings here as we can avoid sending duplicate data over the network. A major challenge to this approach however is how to avoid impacting primary services. Backup is typically a secondary service, so any backup solution involving source-side deduplication must be low-profile enough that it will not impact primary services.

Deduplication can also be performed in a third location. In this case the deduplication machine likely is just responsible for deduplication indices, and either the source or target are responsible for driving the deduplication (sending detection requests, accumulating responses, and eliminating duplicate data).;

In this work I propose a solution for source-side deduplication based on my work in [47].

3.1.2 When to Deduplicate

There are two basic options for when to perform deduplications: inline or offline. Inline deduplication detects and eliminates duplicates before writing to disk, while offline

deduplication writes all backup data to disk at the target, then later performs duplicate detection and elimination.

For inline deduplication, the main bottleneck is detection latency. For high throughput systems, there is a limit to how much RAM is available for buffering writes. This means that we must either detect the duplicate or assume that the data is unique fast enough to keep up with the incoming requests in order to avoid running out of RAM. We can spill buffers to RAM, but for a pure-inline approach at some point we need to start considering offline deduplication (or else we will be receiving request, writing to disk, reading from disk, then detecting and responding instead of just receive,detect,respond). The key advantage to an inline solution is that it requires less temporary disk space, and, in situations where most data is duplicate (e.g. daily VM snapshots), this is usually the case.

For offline deduplication, we need to keep resource utilization low enough that it does not impact new incoming backup data, and to have average daily throughput at least as high as the amount of data added each day. The advantage to offline deduplication is that saving a snapshot is straightforward. Offline deduplication keeps the duplicate detection out of the critical path so the bottleneck becomes simply the throughput instead of both throughput and latency. The downside to a pure offline approach is the amount of temporary storage space it requires (equal to the size of primary storage). For the case of VM virtual disk snapshots, the ratio of duplicate to unique data may be 50-100:1 (in the datasets I have been working with 1-2% change per day is typical).

There are major problems for both pure inline and pure offline deduplication, so I propose the best solution will most likely incorporate both. The solution I propose in my work[47] performs low-latency, cheap deduplication inline (e.g. comparisons against previous snapshot), and then performs batches of duplicate detection in a semi-offline manner. In this way we save most of the temporary storage space (X% instead of 100%

of the data size) and can reduce the pressure to decrease latency in duplicate detection for the rest. By performing most deduplication up front, that also reduces the scope of the more expensive detection, which may also save time.

3.1.3 Local vs Global Deduplication

Full deduplication (ensuring all data chunks stored in backup storage are unique) requires considering duplicates both within data from a single source (local deduplication) and duplicates across primary storage systems (global deduplication).

Local deduplication has much smaller scope, so the cost is generally lower (detection involves lookups into a much smaller local index compared to a global index). For repeated snapshots of a single VM disk this can still achieve very high deduplication ratios, since only a small percent of the entire disk is rewritten per day on average. The problem with pure-local deduplication is that it cannot detect duplicates across objects such as operating system files, and even for repeated snapshots pure-local deduplication cannot detect duplicate chunks that show up in many storage objects around the same time such as new software version.

Global deduplication provides the opportunity for full deduplication. This means the opportunities for storage savings are higher. However, the scope of duplicate detection is now much larger (all backup storage), so the cost is higher for detection, indexing, and garbage collection.

In section 3.2.1 I discuss our solution involving hybrid deduplication, where we perform some deduplication locally, and then perform global deduplication over the most popular data.

3.2 Strategies and Architecture

Figure 3.1 illustrates a cloud cluster platform targeted by our scheme. Each server hosts multiple VMs and the co-located backup service collects and fulfills snapshot backup requests for VMs every day. The backup data can be transmitted to a separate secondary storage or to a distributed storage system in this cluster. While VM data is stored in a distributed file system, each machine typically caches actively used virtual images; the backup service can exploit the cached copy so that reading modified backup data can be conducted locally.

3.2.1 Strategies

For inner-VM deduplication, each machine can conduct local deduplication using the dirty-bit method. For the remaining duplicates, cross-VM deduplication is necessary and we adopt the VM-centric strategy [48] which simplifies the cross-VM deduplication by focusing on top-k popular duplicates. This is necessary because of the substantial resources global deduplication requires when detecting the appearance of a chunk in any VM for cross-machine fingerprint comparison. Our experimental results show that choosing the top 2-4% most popular items (called PDS) for cross-VM deduplication can accomplish close to 98% of deduplication efficiency compared to full deduplication. Chunks which are not deduplicated locally or by the popular data set are backed up to a data store for each VM, which allows us to simplify deletion. The asynchronous cross-VM deduplication works full or popular chunk deduplication while adopting popular chunk cross-VM deduplication can greatly simplify our deletion design as we discuss later.

We propose two new strategies to accomplish aggressive source-side deduplication with minimal resource usage.

1. Distribute the signature index, such as PDS index, to a cluster of cloud ma-

chines and compare the signatures of candidate chunks with distributed index asynchronously in a multi-round collaborative manner. The asynchronous elimination is necessary to better tolerate load imbalance and straggling tasks. Such a scheme requires a significant amount of buffer space and also some stragglers slow down the entire process and increase the average VM backup time. We play a trade-off through multi-round batch scheduling to limit the size of buffer memory usage and detect the stragglers more aggressively while still allowing a good load balancing.

2. Snapshot deletions can occur frequently since old snapshots become less useful, and both space accounting and reclamation are resource consuming. Filtering out unreferenced data from shared duplicates would require either maintaining expensive live schemas or conducting global reference counting [38, 49]. We propose an approximation strategy and take advantages of separating popular data chunks from unpopular ones to minimize resource utilization. We utilize a bloom-filter [50] for chunks used by different snapshots within the same VM for approximated reference counting with periodic repair. We delay reference counting as much as possible for chunks that are shared among VMs as popular items, assuming other VMs still use such chunks.

3.2.2 Software Architecture

Fingerprint index for popular chunks is partitioned and distributed among the cloud machines that participate in collaborative deduplication. Each physical machine that hosts a VM reads dirty chunks, performs inner-VM duplicate detection, and then sends the signatures of the remaining dirty data to other machines that host popular signature partitions. Thus each machine that hosts a VM and a deduplication service runs the

two agents asynchronously. The **backup agent** at each machine leverages the dirty-bit change tracking and inner VM deduplication, and then runs three concurrent threads. The request thread schedules the backup in batches and initiates duplicate detection requests for each scheduled VM. It reads the dirty documents, divides them into chunks, and computes chunk fingerprints [51]. Then it sends a duplicate detection request for each chunk to the proper machine. The second thread accumulates responses from duplicate detection agents. The third thread performs the real backup of non-duplicate chunks. The **duplicate detection agent** manages three threads to accumulate detection requests and compares them to local index data periodically. It also updates the index when new fingerprints are added the system. The main thread performs multiple rounds of fingerprint comparisons. Data processing of the entire job is divided into multiple rounds where each of these rounds performs a comparison of part of the entire data with the existing fingerprint partitions at each machine. The multi-round setting provides flexibility to handle the skewed workload so that small VM data backup can be handled as early as possible rather than the large VMs being a bottleneck for the whole deduplication process.

When some machines fail or respond slowly, a backup agent sets up a timeout and activates a detection task in another machine or temporarily considers these unprocessed requests as non-duplicates. The system conducts global cleanup and removes undetected duplicates periodically (e.g. every few months).

It is a challenge to control the resource usage at each round of processing. For memory usage, we divide the entire fingerprint index evenly and distribute to multiple machines. For each machine, we further partition the local index so we only need one index partition at a time for duplicate detection. Thus memory usage for each round is controllable and can be be very small.

For disk I/O bandwidth usage, we set a bandwidth limit with I/O throttling so that

other cloud services are minimally impacted. For example, in our current testbed, the limit is 50MB/s, which is about 16.7% of the peak local storage bandwidth. Also we set a limit for the total I/O data size per machine. For local disk usage, the cost for buffering messages and storing distributed fingerprint index is relatively small.

When one of machines fails or slowly responds, archive agents may not be able to make progress to output non-duplicate data and can create a bottleneck. We use two strategies to address this.

1. Replicate the fingerprint partitions of a machine to another machine in the cluster. When a machine does not respond for a period of time, we will activate its secondary agent to perform detection.

2. Set a timeout for VMs duplicate detection. If there is no response for an outstanding request for for a period of time, we will consider the corresponding fragment to be new. Thus the output of VM archive may be triggered without receiving all of its duplicate detection requests. Periodically we conduct global cleanup and remove undetected duplicates.

The overall job time of the above scheme is modeled as

$$J_a \approx O(\sum_{r=1}^{R} \max_{i=1}^{p}(C(\sum_{j \in U_{i,r}} D_{i,j}),$$

$$L_i + f(\sum_{i=1}^{p} \sum_{j \in U_{i,r}} D_{i,j}/p)))$$

where R is the number of schedule rounds and $U_{i,r}$ is the set of VMs scheduled for deduplication at round r of machine i. The average processing time per VM is about J_a/R.

Our backup service runs two agents at each physical machine, discussed below. We

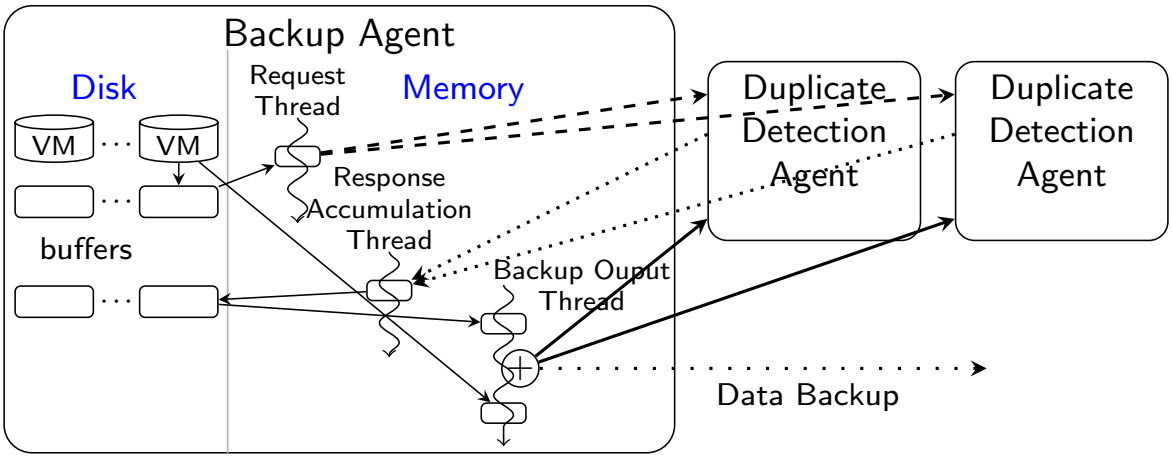


Figure 3.3: Workflow for an asynchronous backup agent.

use the following three symbols in the rest of this work. p is the number of physical machines in the cluster. v is the average number of VMs hosted per machine. Each machine hosts q partitions of global fingerprint index for duplicate detection.

3.2.3 Backup Agents

The backup agent at each machine leverages the dirty-bit version change tracking to identify modified data segments, and then runs three concurrent threads as illustrated in Figure 3.3.

1. *Duplicate detection request thread.* This thread first initiates duplicate detection requests one VM at a time. The backup agent reads the dirty segments, divides each segment into chunks, and computes chunk fingerprints. After fingerprinting, the agent compares fingerprints against the previous snapshot to perform local deduplication. For each duplicate chunk, the backup agent sends the chunk fp to the proper detection agent for reference counting. For each new chunk, it sends a duplicate detection request to a proper machine. We uniformly divide and map the chunks into a set of detection machines using a simple hash partitioning based on fingerprint, and each machine is assigned part

of the global fingerprint index. When request sending for one VM completes, this thread notifies duplicate detection agents in all machines. We use a disk buffer to keep a copy of all outgoing requests to each machine. If the destination machine fails or is too slow to respond, we may request another machine to perform detection.

2. *Response accumulation thread*. This thread accumulates responses from duplicate detection agents and divides them by VM, Thus there are v local buffers to store responses. Each response is a duplicate summary that includes a fingerprint value, storage reference if it was found in the index, and the VM id. Once all responses for a VM have been received, this thread notifies the backup data output thread that the backup data of this VM is ready to be transmitted.

3. *Backup data output thread*. This thread performs the real backup of non-duplicate chunks. The first step is to fetch the on-disk duplicate detection responses for a VM. Next we re-read the dirty segments of the VM and process each chunk using the duplicate detection responses. For duplicate chunks, this thread saves chunk storage references to the metadata structure (i.e. snapshot recipe). For non-duplicate chunks, it sends chunks to backup storage, and the new chunk references are saved in the metadata. For chunks that are new to the index, this thread notifies the duplicate detection agents for index update with the new chunk reference.

3.2.4 Duplicate Detection Agent

This agent accumulates detection requests and compares them to local index data periodically. It also updates the index when new fingerprints are added the system. As shown in Figure 3.4, it accomplishes the above tasks with three threads.

1. *Request accumulation thread*. This thread accumulates requests from other machines, divides them into q request partitions based on their fingerprint values, and buffers

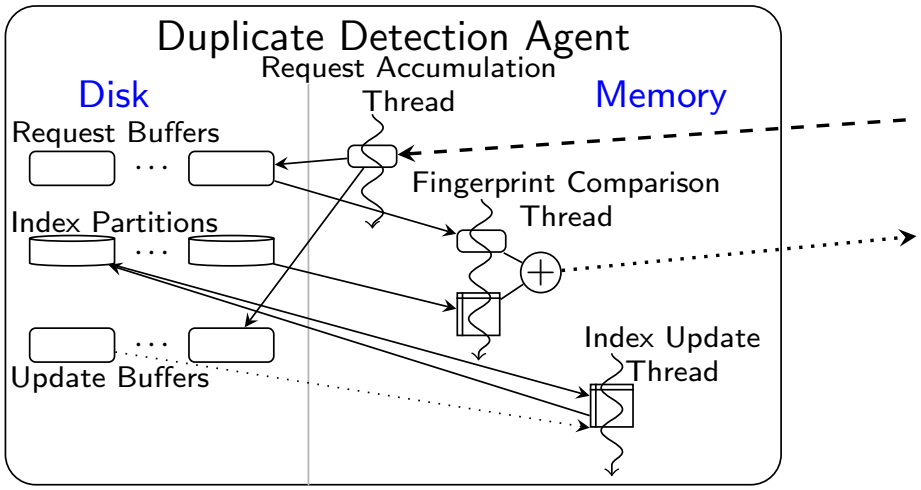


Figure 3.4: Asynchronous Detection Agent Workflow

partitioned requests in the local storage.

2. *Fingerprint comparison thread.* This thread performs multiple rounds of fingerprint comparisons. Each round performs a comparison based on one fingerprint partition at a time: it loads one request partition and the corresponding fingerprint index partition from the local storage to memory, and then compares them to identify duplicates. References are counted for each duplicate, to determine popular chunks (the actual PDS is only updated monthly though). For unpopular chunks they are marked as such, while for popular chunks a reference is returned (or a message that the chunk is new to the PDS and requires backup). The comparison results are summarized and sent to the backup agents which initiated corresponding requests. Since summary records are re-distributed to all p machines, we allocate p send buffers.

3. *Index update thread.* Once new and unpopular chunks are created for non-duplicates in backup storage, the backup agents send index update messages for all new chunks. The above request accumulation thread collects such requests. This index update thread adds new entries to the partition indices, one partition at a time. Although we conduct index update in parallel with duplicate detection, we have the

threads processing different partitions at the same time to eliminate the need for locking.

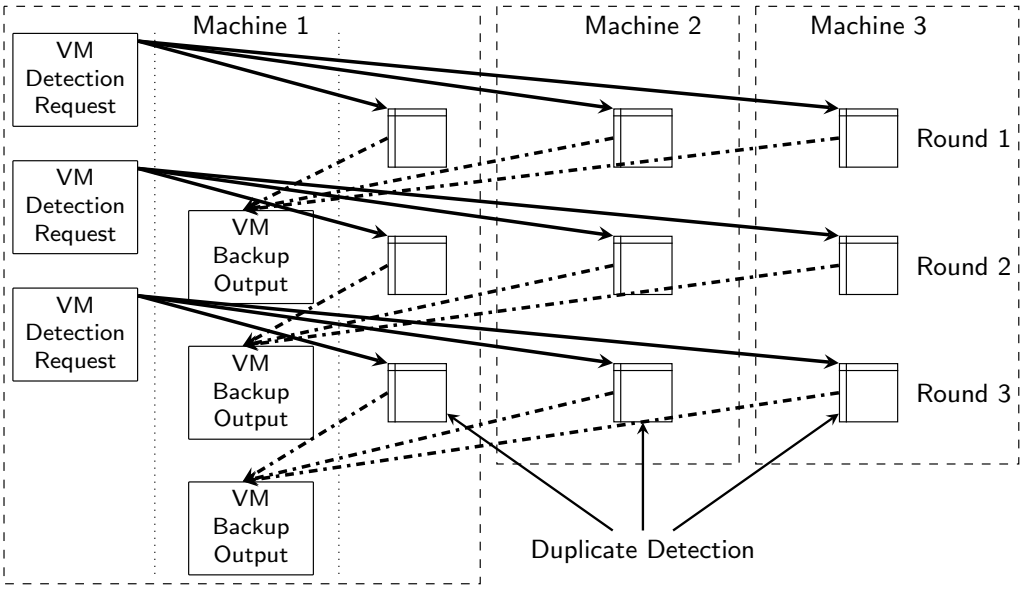


Figure 3.5: 3-rounds of comparison batches. Backups from Machine 1 are shown.

3.2.5 Resource Management, Round Scheduling, and Exception Handling

There are three issues we discuss with asynchronous deduplication. The first issue is how we can effectively control the resource usage in terms of memory and I/O bandwidth. The second issue is to determine the number of rounds for duplicate detection; since each round reloads the partitions of fingerprint index one by one, multiple rounds increases the I/O bandwidth usage. The third issue is how to handle exceptional cases when some of machines fail or are extremely slow.

3.3 Scheduling and Resource Control

Each backup agent conducts k rounds of backup batches and it selects the request initiation of a VM with a smaller data size first. The objective is to complete the backup of small VMs first and to shorten the average backup time per VM. Figure 3.5 shows an example of scheduling in which 3 rounds of fingerprint comparison are triggered at each machine. Through the rest of this work, we use parameter values $v = 25$, p=100, $D = 8.8$GB, $\mu = 22.8\%$, $b = 10$, r=136, and $q = 400$ as an example based on a test dataset discussed in Section 4.4.

To control memory usage, we divide the entire fingerprint index evenly and distribute to p machines. We assume each machine hosts VMs and also participates in collaborative deduplication, though other schemes could be investigated (such as a small number of dedicated index machines). Each machine further divides the local index to q partitions so that the duplicate detection agent loads one index partition at a time during comparison with a small memory need. The main memory usage is buffering for communication among machines. 1) Each backup agent uses p request send buffers and $\frac{V}{p*k}$ response receive buffers per machine where V is the total number of VMs hosted in a cluster; 2) Each duplicate detection agent uses p request receive buffers and p send buffers. The fingerprint comparison thread needs memory to load one of q on-disk index partitions and the requests for that partition. The total memory usage is about $\frac{D*V}{r*p}[\frac{1}{k*q} + \frac{b\mu}{q} + \frac{1}{k}]$ where D is the amount of modified data per VM that needs backup after inner VM deduplication; μ is the percentage of unique chunk entries among dirty data accumulated in all snapshots; b is the average number of snapshot versions maintained for each VM; r is the ratio of average chunk size over the index entry size. When the overall index size increases, the memory cost can still be well controlled by increasing q value. This is a key advantage over [48].

$\frac{D*v(b\mu+1)}{rq}$. The sum of the above numbers is the total memory space required for both backup and detection agents. A large k value reduces the overall space significantly, but it increases the chance of load imbalance, the cost of synchronization, and also the cost of disk I/O in repeated reading of signature index data for k-round comparisons. Our rule is to allocate less than 100MB of memory usage per machine. The above formula then leads to the estimation of k. For our test data with k=12 which means 9% of VM is handled at a time for each machine. The total memory usage for such a setting is about 72MB per machine, excluding some miscellaneous costs such as thread space.

When a VM is extremely large relative to others in its cluster, special handling is needed to control memory usage. The backup agent divides this VM into a number of sub-VMs and the size of each sub-VM is the same as the average VM size in the cluster. The response accumulation thread buffers the detection response for a chunk based on its corresponding sub-VM ID. The backup data output thread reads one sub-VM at a time for this VM, and checks duplicate status from the corresponding sub-VM response buffer. The metadata for the VM can then be reconstructed by appending each sub-VM's metadata.

This multi-round processing allows the backup output thread to process VMs as early as possible. How often should fingerprint comparison batch operations be scheduled at the duplicate detection agent? One strategy is to start a new round as soon as there is a VM that completes its detection request process, namely the number of rounds $k = p \times v$. This number is too large, consuming a large mount of I/O since each round needs to reload the entire local fingerprint index to memory.

To identify a reasonable value for k, we use upper bound namely $(2 + f)Dv$ aforementioned for local I/O size. Cost of local I/O of each machine involves:

1. reading dirty VM data twice costs $2Dv$;

2. performing k rounds of fingerprint comparisons costs $k\frac{Dvb\mu}{r}$.

The cost for local disk storage usage is relatively small for buffering messages and storing distributed fingerprint index; in our tested dataset, the total disk overhead is under 10GB per machine. The CPU usage is also small, less than 15% of one core during our experiments. The main resource usage in our asynchronous scheme that may create a resource contention is memory and disk I/O bandwidth. For disk I/O bandwidth usage, we set a bandwidth limit with I/O throttling so that other cloud services are minimally impacted; in our experiment, the limit is 60MB/s, which is about 20% of the peak local storage bandwidth.

Handling Slow or Failed Machines. Every machine in the cluster is responsible for part of the fingerprint index and if one of them fails or slowly responds, backup agents may not be able to make progress to output VM data. We use two strategies to address this.

1. Replicate the fingerprint partitions of a machine to another machine in the cluster. When a machine does not respond for a period of time, we will activate its secondary agent to perform detection.

2. Set a timeout for VM's duplicate detection. If there is no response for an outstanding request for for a period of time (e.g. 3 times the expected response time), we will consider the corresponding chunk to be new. Thus the output of VM backup may be triggered without receiving all of its duplicate detection requests. Periodically we re-conduct global cleanup and remove undetected duplicates.

The other key idea of our deduplication solution is batch processing of requests. Between processing steps, we queue requests and responses on disk to reduce the memory requirements of our system. The additional improvement to the deduplication model

here compared to our previous work is that we reduce the number of message exchanges to 3 (excluding data backup), compared to the previous approach which required 4 synchronized all-to-all exchanges.

For comparison, consider a system which performs all processing in memory. We will evaluate resource requirements in a 100 machine cluster, with 25x40GB VMs per machine, where all VMs are backed up every night. Our testing on production data has shown approximately 78% of data to be eliminated using dirty segment detection with 2MB segments, which corresponds to 8.8GB of dirty data per VM, or 220GB of dirty data per machine. If we use chunking with an average size of 4KB, and assume 30B requests (20 bytes for SHA-1 hash, plus 10 for VM identifiers and status flags), memory requirements are as much as 1.6GB just for receive buffers, and if we are trying to have minimal impact on any primary services this is not desirable. If we instead transmit the complete set of dirty data (without additional source deduplication), all 220GB per machine must be transmitted from the backup agent.

Our solution to this problem is on-disk queuing of incoming and outgoing messages, followed by batch processing of those queues. This allows for extremely low memory requirements - we restrict memory to 35MB, most of which is for receiving network buffers or for partition index, depending on the current stage of processing. With on-disk queuing and batching of requests, The only structure which needs to fit completely in memory is a single partition index, and by adjusting the number of partitions different memory requirements can be met.

The duplicate detection scheme we use consists of three steps in the duplicate detection agent, as described in Section 3.1.

1. Read one partition index into memory.

2. Switch the update and duplicate detection request queues to new files (so we can

process the current batch).

3. Process all the updates in the update request file. For any duplicate updates, queue a reference update for that backup agent.

4. Process all the requests in the detection request file. Responses are queued to the same files as the reference updates.

5. Send all the updates back to backup agents from the queue files (batching responses as much as possible).

With our batch processing scheme, only 3 sequential reads are required per partition per batch, and there is no seeking required to locate index entries on disk, as with many inline approaches. This allows us to minimize and control the impact of the deduplication service on disk bandwidth and memory usage.

The tradeoffs made by our batch processing algorithm are that we significantly reduce memory requirements, but increase duplicate detection request latency and slightly increase storage space. The increase in storage space is small, and is no more than would have to be stored in memory otherwise (so 1.6GB as described above). The increase in request latency is more important, so we consider that more closely in the following section.

3.4 Approximated Lazy Snapshot Deletion

We do not need to remove chunks that are shared among other VMs as popular items if other VMs still use the chunk. The system periodically gathers usage information in each index partition, recomputes the top popular items, and adjusts the deletion decisions since each machine maintains the usage information of each chunk per partition. For

chunks which are not shared by other VMs, we need to quickly identify if they are used by other snapshots of the same VM. Since the VM size is highly skewed in practice, a large VM may still require a substantial amount of memory for the mark-and-sweep process of data chunks used by all snapshots of this VM.

We use a Bloom filter per snapshot to quickly check if the chunks are still referenced by living snapshots. The tradeoff is that there is a small false positive probability where it identifies unused data as in use, i.e., storage leakage.

3.4.1 Approximate Deletion

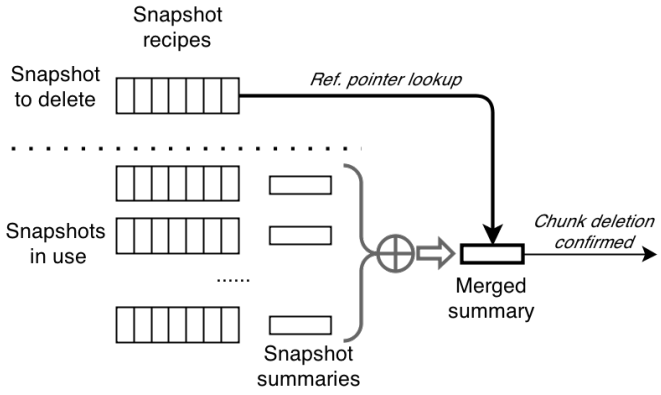


Figure 3.6: Approximate deletion

The approximate deletion algorithm contains three aspects.

3.4.2 Computation for Snapshot Reference Summary

Every time there is a new snapshot created, we compute a Bloom-filter with z bits as the reference summary vector for all non-popular chunks used in this snapshot. The items we put into the summary vector are all the references appearing in the metadata of the snapshot. For each VM we preset the vector size according to estimated VM image size; given h snapshots stored for a VM, there are h summary vectors maintained. We adjust

the summary vector size and recompute the vectors if the VM size changes substantially over time. This can be done during periodic leakage repair described below.

3.4.3 Fast Approximate Deletion with Summary Comparison

To approximately identify if chunks are still used by other undeleted snapshots, we compare the reference of deleted chunks with the merged reference summary vectors of other live snapshots. The merging of live snapshot Bloom-filter vectors uses the bit-wise OR operator. Since the number of live snapshots h is limited for each VM, the time and memory cost of this comparison is small, linear to the number of chunks to be deleted. If a chunk's reference is not found in the merged summary vector, this chunk is not used by any live snapshots. Thus it can be deleted safely. However, among all the chunks to be deleted, there are a small percentage of unused chunks which are misjudged as being in use due to Bloom filter false positives, resulting in storage leakage.

One advantage of the above fast method is that it can finish and free storage usage immediately, while other off-line methods (e.g. [38, 52]) cannot. That is important for storage accounting as users pay for used storage and delayed deletion affects the accounting.

3.4.4 Periodic Repair of Leakage

Leakage repair is conducted periodically to fix the above approximation error by comparing the live chunks for each VM with what are truly used in snapshot recipes. Since it is a VM-specific procedure, the space cost is proportional to the number of chunks within each VM. This is much less expensive than the VM-oblivious mark-and-sweep which scans snapshot chunks from all VMs, even with optimization [38]. Given n VMs in a cluster, the space cost is n times larger. For example, consider each reference

consumes 8 bytes plus 1 mark bit. A VM that has 40GB backup data with about 10 million chunks will need less than 85MB of memory to complete a VM-specific mark-and-sweep process in less than half an hour, assuming 50MB/s disk bandwidth is allocated. We have conducted an analysis to estimate on how often leak repair should be conducted. Assume that a VM keeps h snapshots in backup storage, creates and deletes one snapshot every day. Let u be the number of chunks brought by initial backup for a VM, Δu be the average number of additional chunks added from one version to next snapshot version. ϵ is the misjudgment rate of being in use caused by the merged Bloom filter. Each Bloom filter vector has z bits for each snapshot and let j be the number of hash functions used by the Bloom filter. Then $\epsilon = (1-(1-\frac{1}{z})^{jU})^j$ where $U = u + (h-1)\Delta u$. Then the total number of chunks in a VM's snapshot store is about: $U = u + (h-1)\Delta u$.

Each Bloom filter vector has z bits for each snapshot and let j be the number of hash functions used by the Bloom filter. Notice that a chunk may appear multiple times in these summary vectors; however, this should not increase the probability of being a 0 bit in all h summary vectors. Thus the probability that a particular bit is 0 in all h summary vectors is

$$(1-\frac{1}{z})^{jU}.$$

Then the misjudgment rate of being in use is:

$$\epsilon = (1-(1-\frac{1}{z})^{jU})^j. \tag{3.1}$$

For each snapshot deletion, the number of chunks to be deleted is nearly identical to the number of newly added chunks Δu. Let R be the total number of runs of approximate deletion between two consecutive repairs. We estimate the total leakage L after R runs

as:

$$L = R\epsilon\Delta u.$$

When leakage ratio L/U exceeds a pre-defined threshold τ, we trigger a leak repair. Namely,

$$\frac{L}{U} = \frac{R\Delta u\epsilon}{u + (h-1)\Delta u} > \tau \Longrightarrow R > \frac{\tau}{\epsilon} \times \frac{u + (h-1)\Delta u}{\Delta u}. \tag{3.2}$$

For example for our test data in Section 4.4, $h = 10$ and each snapshot adds about 0.1-5% of new data. Thus $\Delta u/u \approx 0.025$. For a 40GB snapshot, $u \approx 10$ million. Then $U = 12.25$ million. We choose $\epsilon = 0.01$ and $\tau = 0.05$. From Equation 3.1, each summary vector requires $z = 10U = 122.5$ million bits or 15MB. From Equation 3.2, leak repair should be triggered once for every R=245 runs of approximate deletion. When one machine hosts 25 VMs and there is one snapshot deletion per day per VM, there would be only one full leak repair for one physical machine scheduled for every 9.8 days. If $\tau = 0.1$ then leakage repair would occur every 19.6 days, so we use this value to have only 1-2 repairs per month.

3.5 Evaluation

We have evaluated our prototype implementation on a Linux cluster with 8-core 3.1GHz AMD FX-8120 using a production dataset from Alibaba Aliyun's cloud platform [5]. There are 2500 VMs running on 100 physical machines, each machine hosts up to 25 VMs, and each VM keeps 10 snapshots in backup storage. Each VM has about 40GB of storage data on average. The fingerprint for variable-sized chunks is computed using their SHA-1 hash. We have also included some synthetic traces based on VM size distributions from larger Alibaba clusters (see Figure 3.2). We compare three source-side

deduplication schemes.

1. Pure dirty-bit detection. All data are divided into 2MB fix-sized segments and only dirty segments are sent to backup storage.

2. Synchronous multi-stage scheme [46].

3. Asynchronous scheme.

3.5.1 Resource Usage

Table 3.1 compares resource consumption. Column 2 shows the memory required during deduplication and backup per physical machine. Fingerprint comparison does need more memory than the pure dirty-bit method. The multi-round collaborative scheme uses concurrent thread processing and thus requires more memory than the synchronous scheme. However we limit its memory usage to 90MB, which is a small fraction of the available memory on a server. Column 3 of Table 3.1 shows the total size of local disk IO required. Our scheme reads backup data twice, thus doubling the I/O size. The collaborative scheme does incur slightly more I/O than the synchronous one because of k rounds of fingerprint comparison. Column 4 lists the final size of output data sent to the backup storage for 2500 VMs with $p = 100$. The dirty-bit method reduces the data size by 75.86%. The final data after our collaborative deduplication is 4.55x smaller. Column 5 shows the network communication size including backup data transmission, and inter-machine deduplication message exchange. The pure dirty-bit approach does not have inter-machine deduplication, but communicates 4x more data because of more undetected duplicates.

Our approach has much lower network IO requirements than the dirty-bit approach because we apply aggressive source-side deduplication. Without source-side dedup, all

Algorithm	Mem (MB)	Local IO (GB)	Storage (GB)	Network (GB)
Dirty Bit	<10	220	22000	22000
Synchronous	40	453	4840	5500
Collaborative	90	491	4840	5500

Table 3.1: Resource usage comparison per snapshot. Local disk IO and memory costs are per machine. Storage and network cost are for 100 physical machines after deduplication.

Hours	Job span	Backup time per VM (even)	Backup time per VM (skew)
Dirty Bit	1.25	0.05	0.05
Synchronous	50.40	2.75	50.40
Collaborative	2.36	0.23	0.23

Table 3.2: Job span and average per-VM backup time

dirty segments must be sent to the backup system, so this is one of the main advantages of our approach. After including the overhead of our collaborative deduplication service, the dirty-bit approach must send 3.5 times more data over the network because the dirty 2MB segments still consist of mostly clean blocks (if our testing only 22% of chunks in dirty segments are unique).

3.5.2 Processing Time

Table 3.2 shows the total job time (job span in hours) in Column 2 for a synthetic 2500 VM dataset with a skewed VM size distribution (max/average=20) following Figure 3.2. Columns 3 is the average per-VM backup time for this dataset with a relatively even size distribution. Column 4 shows the per-VM time when VM size is skewed. Our scheme reads VM data twice, and thus doubles the job span. Collaborative processing uses 12 rounds and is much faster than the synchronous scheme. In the skewed case, backup time per VM is very high for the synchronous scheme because all VMs must wait for the completion of large VMs at each synchronized stage.

3.5.3 Impact of k rounds

Figure 3.7 shows the average backup time per VM and job span in the asynchronous scheme. As k increases, the average backup reduces because a large k value provides more opportunities for earlier VM output and the job span increases slightly because there is more multi-round processing overhead.

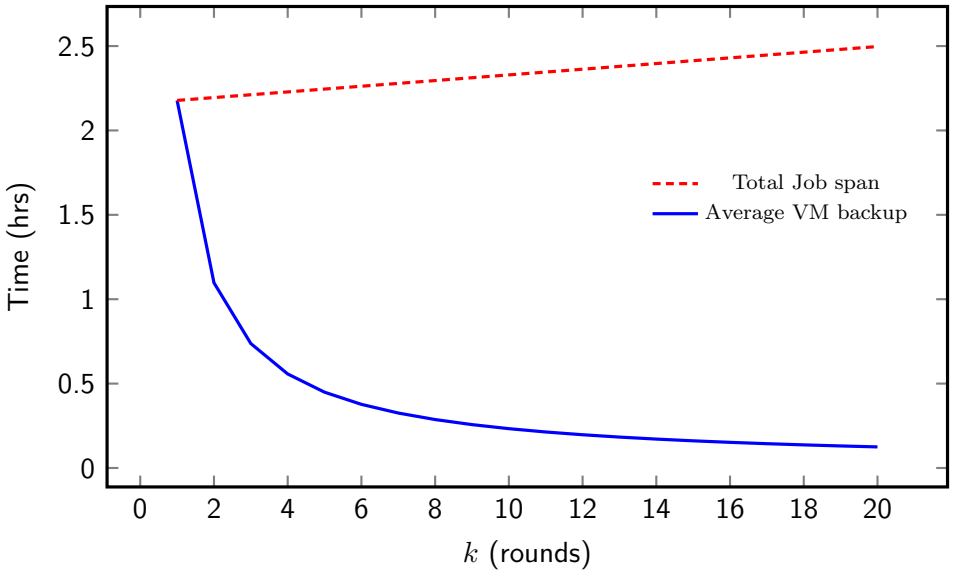


Figure 3.7: Job span and average per-VM backup time

	Time p=50 (hours)	Time p=100 (hours)	Memory (GB)
Mark&sweep	35.9	84.3	1.2–3
Grouped mark&sweep	18.6	43.6	1.2–3
Local w/o sum.	0.7	0.82	0.05 – 1.96
Approx. local	0.012	0.014	0.015
Leak repair	0.7	0.82	0.05 – 1.96

Table 3.3: Processing time and per-machine memory usage of four deletion methods

3.5.4 Effectiveness of Approximate Deletion

Table 3.3 lists a comparison of processing time and memory usage using the four deletion methods when the number of physical machines $p = 100$ and $p = 50$. These four

methods are 1) the standard mark-and-sweep method. 2) Grouped mark-and-sweep [38]. 3) local without using summary vectors (Row 4). 4) approximate local with summary vectors. The last row of Table 3.3 is the cost of the leakage repair for local with summary vectors. The mark-and-sweep process requires all machines read snapshot metadata for usage comparison. The I/O read speed for the back-end distributed file system is about 50MB/second and there is some throughput contention when all machines read data simultaneously: the speed drops to about 30MB/second when $p = 50$ and 25MB/s with $p = 100$. We explain the results for $p = 100$ below. The explanation for $p = 50$ is similar and the result difference for $p = 100$ and $p = 50$ shows our deletion method scales well when p increases.

For the mark-and-sweep method (Row 2 of Table 3.3) on $p = 100$, we conduct p phases of the mark-and-sweep process. At each phase, a physical machine reads $1/p$ of the non-deduplicated chunk metadata and keeps a reference table in the memory. Then all machines read the metadata of snapshots in parallel and mark the used chunks in the above reference table. The above phase is repeated 100 times (one for for each physical machine). The memory allocated at each physical machine is for the chunk reference table at each phase. The average size is 1.2GB and the maximum is 3GB due to data skew. There is a trade-off between memory usage of a reference table in terms of the size and the total processing time. If we reduce the size of the reference table at each phase, then there are more phases to mark all data and the whole process will take more time. For the grouped mark-and-sweep (Row 3), about 50% of snapshot metadata reading can be avoided by actively tracking the reference usage of non-duplicate chunks. Thus it takes 50% less time, which is about 43 hours, but the memory requirement does not decrease. Notice that in our setting, because snapshot deletion occurs frequently, the grouped mark-and-sweep approach becomes less effective in reducing metadata I/O.

For the local deletion without summary vectors (Row 4 of Table 3.3), all physical

Deletions	1	3	5	7	9
Estimated	.02%	.06%	.10%	.14%	.18%
Measured	.01%	.055%	.09%	.12%	.15%

Table 3.4: Accumulated storage leakage by approximate snapshot deletions ($\Delta u/u = 0.025$)

machines conduct the mark-and-sweep process in parallel, but each machine only handles one VM at time and the scope of meta data comparison is controlled within the single VM. Popular chunks are excluded. The average memory usage is the index size of non-deduplicated VM chunks, which is about 50MB on average and the largest size is 1.96GB. For approximate deletion with summary vectors (Row 5), each physical machine loads the VM snapshots and only needs to compare with the summary vectors. The memory usage is controlled around 15MB for hosting the summary vectors and small buffers. The deletion time is reduced to less than 1 minute. The periodic leakage repair (Row 6) still takes about 0.83 hours while using an average of 50MB memory. For few big VMs due to data skew, their repair uses up to 1.9GB memory and lasts about 1 minute. Such a repair does not occur often (e.g. every 19.6 days).

Table 3.4 shows percentage of unused storage space per VM misjudged as "in use" due to approximate deletion. In this experiment, we select 105 VMs and let all the VMs accumulate 10 snapshot versions, then start to delete those snapshots one by one in reverse order. As we know the actual storage needs after each snapshot creation, the storage leakage can be detected by comparing the size of remaining data in use after deletion to the correct number. Row 1 in Table 3.4 is the number of snapshot versions deleted. Entry value 3 in this row means that snapshot versions of all VMs from 1 to 3 are deleted. Row 2 is based on a predicted leakage analysis briefly discussed in Section 3.4.4 given $\Delta u/u = 0.025$. Row 3 lists the actual average leakage measured during the experiment for all the VMs. The Bloom filter setting is based on $\Delta u/u = 0.025$. After 9 snapshot deletions, the actual leakage ratio reaches 0.0015 and this means that there

is only 1.5MB space leaked for every 1GB of stored data. The actual leakage can reach 4.1% after 245 snapshot deletions for all VMs. This experiment shows that the leakage of our approximate snapshot deletion is very small, below the estimated number.

3.6 Concluding Remarks

The contribution of this work is a scalable solution with multi-round source-side deduplication and approximated deletion for frequent VM snapshot backup. We minimize resource requirements to reduce impact on other primary services by separating duplicate detection from data backup. For the tested dataset, the network cost is reduced by 4x and storage cost is reduced by 4.55x compared to a pure dirty-bit-based method. The multi-round deduplication is an order of magnitude faster than a synchronous scheme, when some machines are very slow or have a skewed load. Approximate snapshot deletion only requires 15MB per machine within 1 minute in the tested cases, which is over 3114x faster than the grouped mark-and-sweep method. Leakage repair is 53x faster with 35% to 96% less memory usage. If we were handling the case mentioned in Section 3.1 with 100,000 VMs using 4000 machines, the size of all VM snapshots sent could be reduced from 0.96 petabyte with a dirty-bit method to 196 terabytes while each physical machine uses about 90MB memory, 10GB disk space, and less than 1% of CPU and sends about 6.6GB metadata for low-profile duplication in less than 3 hours.

Chapter 4

Multiword Top K Private Search

4.1 Private Search Introduction

As companies are moving storage resources to the cloud, they still want to have the features they can support with local datacenters. For sensitive data, there are now extra considerations since the user of the cloud does not have physical control over the machine, and so has to trust the cloud provider not to abuse their control over the machine. For most purposes this may be fine and can be managed with legal agreements. However, for highly sensitive data such as legal documents or corporate secrets legal agreements alone may not be enough and there need to be additional safeguards in place to protect company/user information from a cloud provider that has suffered a data breach (or a malicious one).

To store documents, we can use traditional approaches like encryption to protect data at rest, but if the cloud-VM decrypts the data, then a dishonest physical server could read the data from RAM (or if keys are stored in the cloud-VM it could directly decrypt), so in the case where we have private information stored in a cloud-VM, it needs to be transferred out of the cloud and decrypted using keys not directly stored in the

cloud to be able to guarantee privacy. Other work investigates how to store and retrieve documents securely in the cloud [1].

For large scale systems containing millions of documents, search features are often needed to find documents of interest. The search index itself now contains much information about the contents of the documents, up to containing the entire document text in inverted-index form. Using these traditional IR techniques a dishonest server could learn about the structure and contents of the encrypted private documents without ever seeing the documents themselves.

For example, given feature values f_1 and f_2, the server that uses a homomorphic encryption $E()$ can compute $E(f_1 + f_2)$ using $E(f_1)$ and $E(f_2)$ without knowing f_1 and f_2. But such a scheme is still not computationally feasible when many numbers are involved, because each addition or multiplication is extremely slow, not mentioning the ability of comparing two results using scores computed with homomorphic encryption. For example, the order of two sum values $f_1 + f_2$ and $f'_1 + f'_2$ is unknown to the server even it can securely compute $E(f_1 + f_2)$ and $E(f'_1 + f'_2)$. Order-preserving encryption (e.g. [53, 54, 55, 56]) allows a sever to compare two encrypted numbers without knowing the actual numbers but it does not support additions or multiplication of encrypted numbers.

The sparsity of feature vectors when we have a rich set of features (e.g. [23, 24, 25, 26, 27, 28]) is another challenge. Explicit storage of all these features is not practical due to the explosion in storage space. With no protection of the features we can use standard index compression techniques to save space, however prior work has shown term frequency and other information used for ranking may be susceptible leakage abuse-attacks [19, 20, 29].

The previous work on similarity-based secure ranking [30, 31, 32] converts each TFIDF-based feature vector into two random vectors. Index construction builds forward

index after multiplying feature vectors with secret matrices. Online query processing transforms the given query vector with matrix multiplication also, and derives dot similarity of a transformed query vector with all encrypted document vectors, which results in time complexity as $O(T^2 + DT)$ where D is the number of documents and T is the dictionary size. To extend this model to consider proximity features such as word pairs, parameter T becomes very large. Inverted indexing is not feasible because offline and online matrix transformation with randomization yields dense feature and query vectors, and the space cost of the index becomes $O(DT)$. Recent work in [57, 58] follows the above matrix transformation while considering multi-user data ownership and dynamic document update. The datasets tested in [30, 31, 32] are small with only thousands of documents or terms and high search cost with no inverted index support prohibits such an approach from handling a slightly larger dataset.

Our strategy to address this private ranking open problem is to leverage the previous searchable encryption research and strike various tradeoffs in developing an efficient scheme with limited information leakage to a server. We adopt a client-server collaborative approach in which the server conducts efficient matching, additive scoring, and partial ranking while the client does query preprocessing and the final top result selection. Our design only uses one round of client-server communication since multi-round active communication between the server and client (e.g. [1, 59]) incurs a much higher communication cost and response latency.

We assume that a client owns a dataset and places the corresponding index on the cloud to be searchable by the client. This work does not consider the multi-ownership of documents [57] and it assumes the index on the cloud can be periodically refreshed, but without considering dynamic index update [13, 58]. The presentation assumes that a query contains at least two words because it is relatively easy to handle single-word queries by storing encrypted pre-ranked document IDs for each word on the server.

The **contribution** of this work is an indexing and online search scheme with linear additive scoring for this open privacy-preserving top K search problem. It seeks a tradeoff with partial server-side result filtering in handling a modest-sized dataset. It is several orders of magnitude faster than the previous baseline solution [30] while accommodating four categories of previously-developed ranking signals. Searchable Encryption [2, 3] can be used to conduct privacy-preserving server-side query matching. Single-keyword search [13, 14] has existing solutions such as EDESE [60, 61], which encrypts the inverted index by replacing all keywords with tags generated by a secure PRF (pseudo-random function).

Before search, the client (data owner) prepares an inverted index and replaces each keyword in the index with a tag. During query evaluation, the client generates an sends the tag for the key word to be queried. The server looks up the tag and sends back the list of documents containing the keyword for that tag.

In the EDESE approach, the server does not learn the identities of the keywords to be queried, but learns everything else about the index. Prior work [29] shows how the encrypted EDESE index can be used to uncover the identities of plaintext keywords. Non-deterministically encrypting each posting list as a binary blob prevents many of these attacks on the index, but also prevents richer query techniques like multi-word conjunctive search.

4.1.1 Multi-word Queries

To support large systems with millions of documents more advanced IR techniques are needed such as multi-word search and ranking. Multi-word search is studied in many works including [15, 16, 17].

The OXT protocol [15] introduced by Cash et al. supports efficient boolean query

evaluation for both conjunctive and disjunctive queries. By using two separate private search indices much lower leakage can be obtained compared to prior work while maintaining reasonable search performance. The studies including OXT however do not support ranking, and in datasets where a single conjunctive query may contain up to millions of documents, ranking is a necessary feature to find documents of interest. Efficient and secure top K ranking remains an open research problem because of the challenge in achieving both.

4.1.2 Ranking

The main challenge to perform server-side privacy-preserving top K ranking is that advanced ranking involves arithmetic computation based on raw features (defined in Section 4.2) and hiding feature information through encryption prevents the server from performing effective scoring and result comparison. On the other hand, unencrypted feature values can lend themselves to privacy attacks [19, 20].

Homomorphic encryption [21, 22] addresses the problem of how to secure data while letting the server perform arithmetic calculations without decrypting the underlying data. For example, given feature values f_1 and f_2, the server that uses a homomorphic encryption $E()$ can compute $E(f_1 + f_2)$ using $E(f_1)$ and $E(f_2)$ without knowing f_1 and f_2. But such a scheme is still not computationally feasible when many numbers are involved, because each addition or multiplication is extremely slow, not mentioning the ability of comparing two results using scores computed with homomorphic encryption. For example, homomorphic encryption does not allow the efficient comparison of f_1+f_2 and $f_1'+f_2'$ at the server even it can securely compute $E(f_1 + f_2)$ and $E(f_1' + f_2')$.

We need a solution that addresses both score calculation (for multi-word queries) and top-K over the calculated scores. Order-preserving encryption (e.g. [53, 54, 55, 56]) allows

a sever to compare two encrypted numbers without knowing the actual numbers. For example, given feature values f_1 and f_2, the server that uses an order preserving encryption $E_{ope}()$ can compare the stored $E_{ope}(f_1)$ with the stored $E_{ope}(f_2)$ to compare the original features f_1, f_2 without learning either value. The problem with order preserving encryption for privacy-preserving ranking is that it doesn't support additions or multiplication of encrypted numbers, e.g. the server can't compute $E_{ope}(f_1+f_2)$ given $E_{ope}(f_1), E_{ope}(f_2)$, so can't directly support existing advanced IR techniques. Another problem with order preserving encryption is that it still has significant leakage, and given enough queries the server may be able to infer the distribution of the original data and even the approximate or exact values [62, 63].

Another challenge is that advanced ranking considers a variety of features (e.g. [23, 24, 25, 26, 27, 28]) and feature vectors are often sparse with many zero values. Space usage can grow explosively if zeros are explicitly stored. Using a compact data structure representation such as standard index compression techniques without a proper defense can leak statistic information about index, which may lead to leakage-abuse attacks [19, 20, 29].

The previous work on similarity-based secure ranking [30, 31, 32] converts each TFIDF-based feature vector into two random vectors. Index construction builds a forward index after multiplying each document feature vector with secret matrices. Online query processing transforms the given query vector with matrix multiplication also, and derives dot similarity of a transformed query vector with all encrypted document vectors, which results in time complexity as $O(T^2 + DT)$ where D is the number of documents and T is the dictionary size. To extend this model to consider proximity features such as word pairs, parameter T becomes very large. Inverted indexing is not feasible because offline and online matrix transformation with randomization yields dense feature and query vectors, and the space cost of the index becomes $O(DT)$.

$M^{-1}\vec{d}$, where M is a secret matrix, and online query processing derives similarity by computing $\vec{q}^t M$, and then $[\vec{q}^t M] * [M^{-1}\vec{d}]$. Thus the datasets tested in [30, 31, 32] are small with only thousands of documents or terms. The runtime cost is very high because the dimension of transformation matrix M is While transformed feature vectors preserve privacy, runtime computation is very time-consuming because the dimension of transformation matrix M is the number of all distinct raw features (e.g. single word terms and word pairs) in the dataset. A related technique is adopted in the secure k-nearest neighbor computation [64]. The main weakness for this matrix is that the search scheme iterates the matrix computation among all documents to select the best documents. This technique would not scale with a large number of terms, not mentioning such a size increases dramatically when word distance based features are considered as terms. Recent work in [57, 58] follows the above matrix transformation while considering multi-user data ownership and dynamic document update. The datasets tested in [30, 31, 32] are small with only thousands of documents or terms and high search cost with no inverted index support prohibits such an approach from handling a slightly larger dataset.

With a practical restriction towards fast response time, a server-hosted algorithm must deploy a compromised lightweight scheme to compute ranking scores and select results. While such a compromise leaks some information to the server, our goal is to make search as private as possible while maintaining efficiency in ranking a document set up to 1 million per user on a cloud shared by many users. Thus the design of a private ranking scheme needs to address such a situation and minimize the chance of feature leakage to the server. Our strategy to address this private ranking open problem is to leverage the previous searchable encryption research and strike various tradeoffs in developing an efficient scheme with limited information leakage to a server. We adopt a client-server collaborative approach in which the server conducts efficient matching, additive scoring, and partial ranking while the client does query preprocessing and the final top result

selection. Our techniques address sparsity of feature vectors through optional feature matching, conduct posting and query decomposition during query preprocessing. To seek balance between privacy and efficiency. Our design only uses one round of client-server communication since multi-round active communication between the server and client (e.g. [1, 59]) incurs a much higher communication cost and response latency.

We assume that a client owns a dataset and places the corresponding index on the cloud to be searchable by the client. This work does not consider the multi-ownership of documents [57] and it assumes the index on the cloud can be periodically refreshed, but without considering dynamic index update [13, 58].

Another related problem is that of Query Scrambling [65, 66]. In the Query Scrambling Problem the search algorithm must protect the privacy of the users, but assumes an unencrypted index of public information. The goal there is to protect the behavior and identities of users of existing internet search engines, for example protecting the identities of users searching for sensitive topics like "lawyers for victims of domestic abuse". In this work we assume the index data itself is also private (for example legal documents), but these can be complementary approaches. Future work could combine secure search with query scrambling to provide better user privacy even in the case where the server learns something about the index data.

Our presentation assumes that a query contains at least two words because it is relatively easy to handle single-word queries by storing encrypted pre-ranked document IDs for each word on the server.

We will work under the model where we have 2 parties: a trusted client which is also the data owner, and an untrusted server (possibly running as a cloud-VM) which we assume is trying to uncover private information. The server in this model is assumed to be honest-but-curious. Namely, the server honestly executes the protocol specification and hosted programs, but it may observe and infer the private information of a client

during execution by inspecting hosted data and all data sent by the client.

The **contribution** of this work is an indexing and online search scheme with linear additive scoring for this open private top K search problem. It seeks a tradeoff with partial server-side result filtering in handling a modest-sized dataset. It is several orders of magnitude faster than the previous baseline solution [30] while accommodating four categories of previously-developed ranking signals. While linear scoring [67] delivers a decent relevancy performance, non-linear scoring such as multiple additive trees [68] could deliver noticeable relevancy gains and developing private ranking based on nonlinear scoring remains to be an open problem.

4.2 Problem Definition and Design Considerations

Problem Definition: Given D document feature vectors with T features that a client owns, each document d has many feature values denoted as f_i^d and the client builds a searchable index and places it on the server. We focus on developing an indexing and top K search scheme so that the server can access encrypted matched document features for a query and compute their rank without knowing the underlying feature values within a reasonable response time for a modest-sized dataset. The server also should not learn the meaningful information when features are not involved in search.

We assume each search query contains a conjunction of keywords. For example, query "a b c" means that "a", "b", and "c" must appear in a matched document. We adopt this conjunction constraint to follow the convention that a search engine typically retrieves a document matching all search words typed by a user, or at least ranks them highest. A document returned by a search engine may not explicitly contain some of the search words due to query-side linguistic processing or internal word tagging; but the capability of conjunctive query processing is a desired core functionality. We define the formal

grammar rules as follows.

```
<query> ::=  <conjunctive expr>
          | <conjunctive expr> OR <query>
<conjunctive expr> ::=  <conjunctive query>
              <conjunctive query>
<conjunctive query> ::=  <word>
               | <word > <conjunctive query>
```

Given a set of conjunctive keywords to search, we perform a posting intersection of query words based on blinded trapdoors [15, 16] and hash-key matching to get a result set for ranking.

We adopt the posting intersection of query words based on a searchable encryption algorithm called OXT [15, 16]. Faster traditional intersection algorithms that traverse two or more postings [69] simultaneously are not adopted because such a traversal leaks more information to the server (the EDESE scheme can support this, but has higher leakage [60, 61, 29]).

During the search process, a standard technique to ensure privacy is to use a deterministic pseudo random function (called PRF) to hide information including term IDs and document IDs.

4.2.1 Ranking Formula

In this work we opt for a simple but popular rank score computing scheme which is a linear combination of document features. While such scoring [67] delivers decent relevancy performance, Ranking with multiple additive boosting trees (e.g. [68]) has been shown to be effective with visible gains over linear scoring. However, making such a nonlinear ranking method private involves not only score addition and but also value comparison. There is no known encryption method available that can solve both issues within a single framework, but our other work [70] focuses on how to make this practi-

cal by encrypting trees using a combination of comparison-preserving-mapping for tree traversal and additive-feature-blinding for score summation.

A linear combination formula computes ranking score in a form of $\sum \alpha_i f_i^d$ where α_i is a coefficient of feature value f_i^d for document d. We assume all coefficients are statically determined at index generation time. With this assumption coefficients can be embedded into feature values during index setup. Thus for the rest of the work we will ignore these coefficients, and the linear additive rank formula is simplified to $\sum f_i^d$.

4.2.2 Raw Ranking Features

We call a rank feature as *raw* if it is explicitly stored in the index and a rank feature as *composite* if it is computed based on other raw features. Our design is to make each basic feature in the above additive formula such as term weight or word pair weight as a raw feature and the server retrieves and simply adds them without knowing the role of these values. This minimizes the chance that the server understands their semantic contributions to ranking signals.

We use the above additive formula to support four categories of ranking features used in the information retrieval literature:

1. Term-frequency based composite features such TFIDF and BM25 [71]. They can be represented as the summation of weighted raw word frequency and thus the above additive scheme supports such features.

2. Proximity composite features based on the sum of raw proximity term features. Such a proximity term can be a n-gram within a certain distance [25] or a word pair [72, 26]. A traditional inverted index with positional information stores word positions explicitly [73] and online ranking computes composite proximity features based on word positions in each document. While this scheme is space efficient,

computing composite proximity features requires both order comparison and arithmetic calculation from word positions. As discussed in Section 4.1, there is a lack of encryption techniques to support both secure calculation and order comparison. Leaking relative word positions of each document may enable statistical attacks which reveal document content structure. Thus we opt to use a proximity formula expressed as the summation of raw features that directly model proximity terms. As a result, some of the previous proximity formulas are not be supported, for example, minimum aggregation [23] and span coverage [27].

3. Document query specific features such as document-query click through rate. The name of such a feature can be based on a query ID and this can be stored as a raw feature

4. Document specific features such as freshness and document quality. These features can either be stored as a raw feature with the name based on hashed document ID or can be embedded into other stored features (such as the term-frequency for the start-term of the query).

4.2.3 Handling Sparsity of Raw Ranking Features

Raw features in Category 2), 3) and 4) often have many zero values. If all encrypted zero values are stored explicitly, it would greatly simplify the privacy preserving design, but the space cost would be explosively high and such a scheme would become impractical.

One example is the proximity weight based on inverse of distance of two query words appearing in a document [23, 24, 25, 26, 27]. As distance exceeds a threshold, such a weight becomes insignificant and to avoid excessive space cost, a typical implementation does not store such a value. Many of distance based proximity features [23, 24, 25, 26, 27] fall into this category. Another example is a time-based feature indicating whether

the creation or modification time of a document is recent or in some year. A feature related to quality, authoritativeness, or popularity of a document can also be optional. Artificial terms may be created to handle each of these optional document features. Most documents have no or zero weight value for such features. Thus feature vectors are often sparse. Notice that we cannot just simply incorporate a default value in the search algorithm when an optional feature is not matched as the server could reason about the use of such default value. Then the server may be able to leverage such information and discover the value of the same feature for other documents. One option of handling feature sparsity is to compactly store nonzero feature values for each document as a forward index and once documents that match a query are identified during query processing, encrypted document features can be retrieved using a hashed document ID. Another option is to embed feature values in the posting entries of the traditional inverted index [73]. These options without a proper defense have a privacy risk that the server can inspect document feature vectors or postings directly and gather document statistical information such as word frequency distribution without client authorization and launch leakage-abuse attacks [20, 29].

Our design for optional features which are often sparse is to use an online key-value store. Namely each matched document d involves the following two types of weights.

1. Required individual feature weight f_i^d for document d. When a feature is required, every matched document has a feature value stored in the index. For example, the BM25 score of title and body in a document.

2. Optional feature weights O_t^d where $1 \leq t \leq m$. When a feature is optional, the default value is zero when this feature value of a document is not available from the index. For example, term-pairs (pairs of terms that occur nearby in a single document) are optional features since they can contribute to the relevance of ranking,

but may not be required for a correct query result.

Given a q-term query containing w_1, w_2, $\cdots$, w_q, the client side of our system converts this into the following required features f_1, f_2, $\cdots$, f_q and optional features $O_1, \cdots, O_m$. Total score for document d matched for the above conjunctive query is:

$$\sum_{i=1}^{q} f_i^d + \sum_{1 \le t \le m} O_t^d.$$

4.2.4 Feature Encryption with Mask Blinding

To preserve privacy of feature values, we blind each feature value using a random mask with modular addition to hide this value from the server. This mask is generated in a deterministic way using a PRF, and is known to the client only. Formally, we store each feature value f in the index as $[f + R]$ which is defined as $f + R \bmod N$ where R is the feature mask computed as a PRF of the term ID and the document ID in the range from 0 to $N - 1$. N is set to 2^{32} for our implementation.

The square bracket notation in $[f + R]$ also emphasizes that the server sees the computed value of expression $f + R \bmod N$ but the server is not able to derive individual f or R value since R is derived from client-owned secret keys. As we explain below, the above scheme will allow the server to compute the rank score by adding masked feature values and to conduct partial comparison if choosing masks judiciously with a tradeoff. We did not adopt existing homomorphic encryption scheme supporting additively-homomorphic operations (such as Paillier's scheme [22]) however, because it does not bring visible advantage while being less efficient and cannot support partial server-side ranking. While order preserving encryption (e.g. [53, 55, 56]) supports comparison, it is not considered due to lack of support for addition.

More specifically, for document d, weight of feature f_i^d in index is an integer and is

stored as $[f_i^d + R_i^d]$ with a random noise mask R_i^d and optional weight O_t^d is stored as $[O_t^d + RO_t^d]$ with a random noise mask RO_t^d. Given a query with q required features and m optional weights, the total rank score F plus the total score mask for document d including these masks under modulo N is:

$$[F + M] = [\sum_{i=1}^{q} [f_i^d + R_i^d] + \sum_{1 \le t \le m, O_t \in X} [O_t^d + RO_t^d]]$$

where $M = \sum_{1 \le t \le m, O_t \in X} O_t^d + \sum_{1 \le t \le m, O_t \in X} RO_t^d$ and $O_t \in X$ means that the corresponding optional feature can be found in the key-value store of index.

While the server can compute the above encrypted sum, as the feature masks are random and independent from one document to another, the server is not able to compare the relative rank order among matched documents. When the number of matched documents is modest, the server can send these results to the client along with a bitmap of optional features used for each document, which assists the client to remove the sum of masks in each rank score.

When there is a large number of matched documents in the server or client-server bandwidth is low, we want the server to conduct partial ranking so that results with low scores can be filtered out first before sending back to the client. That essentially becomes a two-stage ranking as a type of cascade ranking [74]. In next section, we explore this possibility with several tradeoffs necessary to balance privacy and query response time efficiency.

In order to allow the server to rank results and to not send an excessive number of top results back to the client, the server must conduct ranking at least partially to filter out results with low rank scores. Server-side filtering is vital when there is a large number of matched documents in the server or client-server bandwidth is low. Without this filtering, reasonable latency can be unobtainable under these cases. Table 4.1 lists

our proposed techniques which mainly seek for a tradeoff between privacy and efficiency in our design, and also strike a balance between efficiency and relevance.

tradeoffs	privacy	efficiency	relevance
server-side partial ranking	−	+	
term and query decomp.	−	+	
random s-terms	+	−	
limiting optional terms		+	−

Table 4.1: Impact of the proposed techniques

4.3 Indexing and Query Processing

In this section, we adopt the OXT searchable encryption [15, 16] for query term intersection and extend it for result scoring and ranking. We present an indexing and query processing scheme to support feature blinding with dynamic chunk-wide random masking. This scheme enables a client to safely trigger server-side partial ranking for selected queries involving popular features. Our design prevents the server from learning any information about the features and posting of an encrypted word if such a word has never been searched.

4.3.1 Basic Search Algorithm

The basic flow of search is as follows (see later in this section for formal description).

First the client does a lookup into the local R-store, and does a range-intersection on the terms. This gives a list of s-term-posting-chunks that need to be intersected with the x-terms.

The construction of the above data structure in the indexing process is described in Figure 4.4 while Figure 4.5 lists the steps of online query processing in accessing these data structures. This section presents the offline index construction and online

query processing algorithms. Table 4.2 lists the function and operator symbols used in our algorithms. They represent the characteristics of the given dataset, algorithm variables, and the system setting. We use standard hashing and encryption functions in cryptography

4.3.2 Notations and Definitions

Posting list a list of documents that a term appears in (and associated metadata for each term-document pair).

PRF a deterministic pseudo-random function that is (practically) non-invertible

Tag key for accessing a table/database/bloomfilter used for lookup (i.e. s-tag/x-tag).

Data structure setup during index construction.

4.3.3 Data Structure and High Level Search Flow

Before presenting the details on our privacy preserving mechanisms, we first outline the search flow in the proposed scheme, Our search flow involves 3 key-value stores:

1. R-store saves meta information in feature posting chunks such as document ID range of chunks. That facilitates query decomposition at the client side.

2. S-store contains required feature values and is used by the search algorithm to identify the candidate documents. We generalize the existing technique to judge set membership [15] as key-value store lookup. In particular, S-store is used to contain both posting of a required s-term and the feature values of the documents in the posting for this term. We choose store name "S" to emphasize that the lookup of this store is driven by s-terms.

3. X-store contains feature values accessible using a pair of document ID and feature ID.

Given a query, a client converts it into a sequence of subqueries using R-store. For each subquery, the client generates n required and optional feature IDs as $w_1, w_2, \cdots, w_n$ for the server to match documents and fetch their features. The server assumes required feature w_1 is the starting feature to fetch a posting list of candidate documents and their feature value from S-store. Let d be a candidate document ID that appears in the posting list of start feature w_1. To fetch another feature w_i of d where $2 \leq i \leq n$, the online algorithm pairs two IDs w_i and d together as the key to access X-store. To preserve privacy, all IDs and intermediate values are hashed or encrypted using operators summarized in Table 4.2 so that for any feature name and document ID, the server cannot access the posting list or feature values from the hosted X-store and S-store without authorization from the client. In addition, given a query which has not been searched in the past, the server cannot perform query processing or ranking without authorization from the client. The server should not learn the identity of the query terms used during online search.

$\|\|$	Bitwise concatenation of two values.
$PRF(k, a)$	A deterministic pseudo-random function of "a" using key k (e.g. SHA256 of $a\|\|k$). Among PRF keys used, $k_0, \ldots, k_9$ are secret known by the client only. k_u is public to generate feature weight mask R_x.
$E(k, a)$	Non-deterministic symmetric encryption of plaintext a using key k (e.g. AES using random initialization vector)
$D(k, b)$	Symmetric decryption of b using key k s.t. $D(k, E(k, a)) = a$ for any plaintext a and key k
$S(stag)$	Look up S-store with key $stag$, and return a chunk of encrypted feature posting.
$X(xtag)$	Look up X-store with key $xtag$, and return the encrypted feature value if exists.
$(x_i)_{i=1}^n$	A list of elements x_i from $i = 1$ to n.

Table 4.2: Function and operator symbols

4.3.4 Encrypted Inverted Index Setup

The input data set with feature vectors is converted into an inverted index format. The indexer, controlled by the client, builds the encrypted index and allows the server to host the S-store and X-store. For each feature w of document d with value f, this indexer constructs a key-value pair for the X-store as follows. Define p as the position count of d in the posting and c is the chunk ID where $c = \lfloor p/csize \rfloor$ and $csize$ is the chunk size.

- The key is $xtag = g^{PRF(k_5,w)PRF(k_2,d)}$ where k_5 and k_2 are secret hashing keys. Base g is a Diffie-Hellman constant for pseudo-random mapping [75].
- The value accessible through above key is: $S(xtag) = f + R_c + R_x \mod N$ where $R_x = PRF(k_u, g^{PRF(k_1,w)PRF(k_2,d)})$ and $R_c = PRF(k_0, w\|c)$ represent the chunk specific and document specific masks, respectively.

When the above feature is required, the d and f values are also saved in the posting chunk c of document w in the S-store. The corresponding S-store key is $stag = PRF(k_7, w\|c)$. The corresponding value is a chunk list of posting entries and each S-term posting entry is an encrypted tuple (e, y, f_s) defined as follows.

1. $e = E(k_e, d)$ which represents the encrypted document ID d. The encryption for e is semantically secure so that the server cannot learn anything except the length of the original document ID.
2. $y = PRF(k_4, w\|p)^{-1}PRF(k_2, d)$ is a blinded bridging number used when w is selected as a start feature for deriving the key to access the X-store during intersection proposed in [15]. We extend its usage for feature fetching and will elaborate it later in the example below.

   When the server receives a x-token ($xtok_i$) from the client, the corresponding x-tag can be recovered as $xtag = xtok_i^y$ at Phase 2 of Figure 4.5.

3. Value y' is a blinded bridging number used to derive part of random mask R_x and enable server-side partial ranking when triggered. computed like y above, used to derive the key R_x to decrypt the feature weight stored in the X-store during queries with result filtering, where $xtoken_i = g^{PRF(k_9,w_i)PRF(k_4,s\|p)}$, $y = PRF(k_4, s\|p)^{-1}PRF(k_5, d)$.

4. Blinded feature $f_s = u_d + f + R_c + R_s \mod N$ where u_d is the sum of document-specific Category 4 feature weights discussed in Section 4.2. R_s is the document-specific mask computed as $PRF(k_3, w||p)$.

We introduce a R-store which maps each subterm to a document ID range based on the chunks in the posting of the original term and allows the client to do earlier range intersection of document ID ranges based on the subterms. Each subterm of a term refers to one chunk in the posting of this term, i.e., $w\|c$ is one subterm of the term w, where c is the chunk ID. We present here how to construct the R-store in Figure 4.2 and how to decompose the query into subqueries with existing R-store in Figure 4.3. The Rstore creates a mapping from terms to lists of document id ranges. The Rstore is partitioned in the same way as the S-store (by posting list length) so we can use the Rstore to determine which S-store partitions to load client-side at the start of the search. The R-store also memorizes the posting length for each feature and can derive subterms of a term based on chunking. During the query processing, this allows the client to derive the corresponding s-terms of a search term, which allows the lookup of the associated document ID ranges.

Variables:

d some document id - in the end, this is what the client wants to get back (for matching documents). although we never reveal d to server, you can think of this as a randomly assigned UUID, so there is no information in d (other than what the client needs to look up a document)

s s-term - term (word) that drives the search, usually term with shortest posting (as estimated by client)

c chunk size for postings

i index into the s-term's posting (since we store postings in chunks of size c, this is a chunk index)

m number of chunks in s-term's posting (so the chunks for w_s go from $i = 1$ to $i = m$)

x_j x-term - term that is intersected with s-term posting to find matching documents

x_j jth x-term

n number of x-terms in the search (so x_n is the last x-term)

$stag$ key into S-store (used to lookup the cth chunk of the s-term posting)
calculated as: $PRF(K_s, w_s|c)$

e docid nondeterministically encrypted using s-term
calculated as: $Enc(k_e, did)$

y did blinded using the s-term chunk id
calculated as: $PRF(K_z, w_s|p) * PRF(K_d, did)$

$xtag$ key into X-store (used to lookup exactly one ¡w_x, did¿ pair to see if document did contains term w_x)
calculated as: $g^{PRF(K_x,w_x)*PRF(K_d,did)}$

$xtok_{ij}$ xtoken = x-term j combined with deblind factor for s-term chunk i
calculated as: $g^{PRF(K_x,w_{xj})*(PRF(K_z,w_s|p)^{-1})}$

f_z blinded feature value

We also address its weaknesses when incorporating it for private ranking.

When the result set is small (or the security requirements are very high), we will support a server side-ranking protocol that returns the full (encrypted) result set to the client, and requires the client to decrypt the evaluation scores to determine the top-k documents.

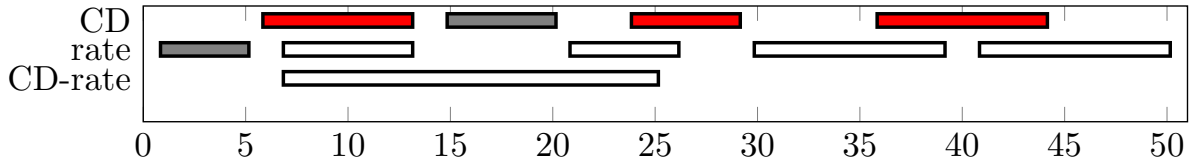
The OXT X-set protocol requires the client to keep sending candidate x-tokens until the server hits the end of the s-term posting, since neither side knows the length of the

s-term posting until the server locates the end of the posting in the T-set. This adds extra overhead to the query communication cost. If either side knew the approximate length of the s-term posting before search, the client would be able to send the correct number of x-tokens to the server up-front.

One of the key advantages of such a hash-table based intersection scheme is to hide the posting structure of the x-terms, which is a significant leakage.

CD	6	7	8	10	13	15	16	18	19	20	24	25	27	28	29	36	40	41	44							
rate	1	2	3	4	5	7	9	10	12	13	21	22	23	25	26	30	31	33	36	39	41	46	48	49	50
CD-rate	7	25																								

(a)Chunking examples



(b) Client-side range intersection for query decomposition

Figure 4.1: Chunking and query decomposition

The first step is to select an s-term. Let's say that in this case we select "cd" as the s-term since it is the least common word.

We use the example in Figure 4.1a to illustrate the partitioned search approach. Let's assume we are searching for "cd rate", and let each word appear in the document id's listed in the posting table. The first step in our partitioned search is to using the client-side R-store to do a range-intersection of the postings. In the example, we set the range chunk size to be 5, though in practice it would be much larger. For each range of the sorted postings, the client records the start and end document id in the R-store (e.g. for "cd" the following ranges are recorded: 6-13,15-20,24-29,36-45).

Figure 4.1b shows a graphical representation of the range-intersection, with the top 3 rows showing the ranges of each chunk for the 3 terms, and the 4th row showing the document id ranges where all terms intersect (at the chunk range level). Using these ranges, the client selects the term with the fewest intersecting chunks as the primary

term, since query cost is proportional to the posting length of the primary term. In this case "cd" is selected, and the client submits a conjunctive query to the server intersecting the matching chunks for the primary term (shaded gray) with the secondary terms.

Since we know the first term of the first chunk cannot match "rate", we can omit any subqueries containing that chunk. Similarly, the client can know just from the range information that document id 13 (the last posting in the first chunk) is a match to the query, so it can locally add it to the results. We cannot do the same with the gap at document id 40 however, since that is in the middle of the chunk and the client cannot know which posting index (if any) correspond to document id 40. This is because the client only has the end-points of each range. (In general, the client may be able to tell the server to skip the first/last posting of a chunk, or an entire chunk, but not postings inside the chunk.)

In this example, we reduce the cost of the query by more than half compared to the naive approach (that must compare all chunks of the primary with all secondary terms). Specifically, instead of 4 chunks * 5 postings/chunk = 20 postings compared, in our approach we only compare 8 postings across 2 chunks. If we partition the on-disk data structures by their corresponding posting-chunk, we can also tell the server which chunks it will have to load to complete the query.

After performing range-based intersection at the client side, the client sends a set of each conjunctive query expression with optional features to the server and the server sends back the top k relevant results, and the client decrypts them to get the docids.

The posting of keyword is sorted (by docid) and is partitioned. Each partition is encrypted and the information is saved in S-store and X-store.

Before we give a more formal definition of key and values used in the S-store and X-store at the server side, we illustrate the relationship between the S-store and X-store during online query processing.

Input: $DB = (w_i, posting_i)_{i=1}^{n}$; Document weights $(u_i)_{i=1}^{D}$.
Initialize an empty dictionary R, where the key is a string and the value is a list of integer pairs of strings.
for i = 1, 2, ..., and n **do**
    Initialize an empty list r.
    Suppose $posting_i = (docID_p, score_p)_{p=1}^{m}$.
    for p = 1, 2, ..., and m **do**
        if $p \mod csize = 1$ **then**
            Set $start = p$.
        end if
        if $p \mod csize = 0 \vee p = m$ **then**
            Append the pair $(docID_{start}, docID_p)$ to the list r.
        end if
    end for
    Set $R[w_i] = r$.
end for
Return R.

Figure 4.2: R-store Setup

Input: $query = (w_1, w_2, ..., w_n)$ and the R-store R.
Initialize an empty list res of tuples of n integers, and a tuple $ind = (ind_1, ind_2, ..., ind_n)$ where $\forall i = 1, 2, ..., n : ind_i = 0$.
Given $R[w_i][ind_i] = (first_i, last_i)$ for i from 1 to n, define: 1) If the intersection of intervals $\cap_{i=1}^{n}(first_i, last_i)$ is not empty, then denote $\cap_i^n R[w_i][ind_i] \neq \emptyset$. 2) If $last_i \geq last_j$ for some i and j, then denote $R[w_i][ind_i] \geq R[w_j][ind_j]$
while *true* **do**
    If $\cap_{i=1}^{n} R[w_i][ind_i] \neq \emptyset$, append $(ind_1, ind_2, ..., ind_n)$ to res.
    Find ind_m so that $\forall i = 1, 2, ..., n : R[w_m][ind_m] \leq R[w_i][ind_i]$
    Increment ind_m by 1.
    If $ind_m > |R[w_m]|$, break the while loop.
end while
Return res.

Figure 4.3: R-store Query Decomposition

Input: Postings $(w_i, posting_i)_{i=1}^{n}$; Document weights $(u_i)_{i=1}^{D}$.
Build R-store and initialize X-store and S-store.
for each feature w **do**
    Let the posting of this feature be $(d_p, f_p)_{p=0}^{m-1}$
    for $p = 0$ to $m - 1$ **do**
        Let chunk ID $c \leftarrow \lfloor p/csize \rfloor$ where $csize$ is chunk size
        Let feature mask $R_c \leftarrow PRF(k_0, w\|c)$
        $R_x \leftarrow PRF(k_u, g^{PRF(k_1,w)PRF(k_2,d_p)})$
        $R_s \leftarrow PRF(k_3, w||p)$
        $f_x \leftarrow f_p + R_c + R_x \mod N$
        $f_s \leftarrow u_{d_p} + f_p + R_c + R_s \mod N$
        $e \leftarrow E(k_e, d_p)$
        $y \leftarrow PRF(k_4, w||p)^{-1} PRF(k_2, d_p)$
        Add key-value pair $\{g^{PRF(k_5,w)PRF(k_2,d_p)}, f_x\}$ to X-store.
        if w is a required feature **then**
            Append $E(PRF(k_6, w\|c), \{e, y, f_s\})$ to the value of key $PRF(k_7, w\|c)$ in S-store.
        end if
    end for
end for

Figure 4.4: Encrypted inverted index setup

1. S-store - inverted index, stores encrypted posting list for each word. One term for each query (the s-term) is looked up in the S-store

2. X-store - Intersection lookup and feature store. Has two parts, bloomfilter for fast intersection, and feature store for ranking and exact intersection

3. R-store - Range set. This is one of our contributions, allowing client-side optimization and reducing leakage to server

Figure 4.4 presents the detailed steps in index construction, where we build the above three key-value stores while Figure 4.5 presents query processing (for server-side and client-side resp.) that uses these stores.

4.3.5 Online Search Procedure

Figure 4.5 shows the three phases of query processing. **Phase 1**, which is conducted at client side, forms required and optional features for a given query and performs an earlier range intersection to derive subqueries after R-store lookup. Assume each subquery has n features with n corresponding chunk IDs: $(w_i, c_i)_{i=1}^{n}$. All required features are placed before optional features in this feature sequence and w_1 is selected as the start required feature. Then the key to access S-store is $stag = PRF(k_7, w_1||c_1)$ and the key used in the server to decrypt an entry of posting $S(stag)$ is $skey = PRF(k_6, w_1||c_1)$. In addition, the client builds a special 2-dimensional token array for the server and each array element $tokens[i][j]$ is defined as: $xtoken_i = g^{PRF(k_5,w_i)PRF(k_4,w_1||p)}$, and $mtoken_i = g^{PRF(k_1,w_i)PRF(k_4,w_1||p)}$ where $2 \leq i \leq n$, $1 \leq j \leq csize$, $csize$ is the chunk size, and posting position $p = c_1 * csize + j$. The corresponding document specific mask for the start feature is $R_s = PRF(k_3, w_1||p)$. If the predicted result length does not exceed a specified threshold, the client will not turn on server-side result filtering. In that case, the second entry $mtoken_i$ of each token array element and R_s information are voided. Finally, the client sends $stag$ and $skey$ along with the token array and related mask R_s if needed to the server for the above subquery.

At **Phase 2**, after receiving the control information for each subquery, the server fetches the posting of w_1 from S-store based on function call $S(stag)$. For every posting entry (assume j-th entry) for w_1 and i-th other feature, it uses the received token array $tokens[i][j]$ to compute $xtag$ to access X-store and also the document specific mask R_x when needed. The server accumulates the rank score with mask subtraction when needed (e.g. Line 21 of Figure 4.5). Then it uses the x-token deblinding factors to compute a vector of x-tags for each document in the s-term posting. The server then looks up each x-tag in the X-store bloom-filter, and for any that do not match the server throws

out that document and moves on to the next document. After getting the list of likely matching documents and their x-tag vectors, the sever looks up the feature vectors for each document using the x-tags (along with the features from the posting). The server computes and compares the relevance scores of documents associated with each optional feature case from all subqueries, eliminates duplicates and impossible documents through result suppression.

When partial ranking is turned on, document scores under each optional feature lattice case are compared and filtered, and at most top K results are returned for each case. For each matched document sent back from the server to the client, the result tuple is in a format of (e, F, O, j), representing an encrypted document ID, an encrypted score, a bitmap on the optional features used, and the position this document appeared in the posting chunk of w_1. We also attach a simple subquery identifier for which these documents are matched. In our experiment, each result on average uses about 16 bytes and this includes an 8-byte encrypted document ID, 4-byte blinded score, 3-byte bitmap, and 1-byte position.

Phase 3 Client post-processing removes score masks and compares all received documents to select the final top K results.

We extend this basic search flow by adding optional x-terms to improve the ranking process. These optional x-terms include title-terms for all terms, and bigrams plus title bigrams. Bigrams are pairs of words that appear near each other in a document. For our evaluation, we set the maximum bigram distance at 9, which means in the worst case the size of the database is increased by 9. However, this greatly improves our relevance scores, so we make this tradeoff between space vs. relevance. To add the optional x-terms, the client adds optional xtokens to the xtoken vectors for each possible matching optional term. During search, the server looks up the optional xtags in the X-store, but does not use them as part of the intersection.

Phase 1: Client-side query preprocessing

Build query terms, decompose query, and enable server-side partial ranking if needed.
for each subquery **do**
    Send S-store access key *stag*, decryption key *skey*, *tokens* array, and related start feature mask R_s if needed to the server.
end for

Phase 2: Server-side subquery processing with scoring

Let $stag, skey, R_s, tokens$ be information received for one subquery.
for j-th entry Z in posting chunk returned by $S(stag)$ where $1 \leq j \leq csize$ **do**
    Decrypted tuple $(e, y, f_s) = D(skey, Z)$
    Initialize Score $F = f_s$. Initialize a bitmap O.
    if partial ranking is enabled **then**
        $F \leftarrow f_s - R_s \mod N$
    end if
    for i = 2, 3, ..., and n, $(xtoken_i, mtoken_i)$ in $tokens[j][i]$ **do**
        $xtag_i \leftarrow xtoken_i{}^y$
        if $X(xtag_i)$ does not exist and w_i is required **then**
            Skip this document for conjunctive requirement
        end if
        if $X(xtag_i)$ exists **then**
            Add i to optional feature bitmap O.
            if partial ranking is enabled **then**
                Let feature mask $R_x \leftarrow PRF(k_u, mtoken_i{}^y)$
                $F = F + X(xtag_i) - R_x \mod N$
            else
                $F = F + X(xtag_i) \mod N$
            end if
        end if
    end for
    Add the tuple (e, F, O, j) to the results list.
end for
if partial ranking is enabled **then**
    Select top K results for set of optional bitmap O and suppress unnecessary results.
end if
Send matched documents as list of (e, F, O, j) to client.
Repeat the above steps for all subqueries.

Phase 3: Client-side post processing

for each (e, F, O, j) of the result list from the server **do**
    Client decrypts document IDs and subtracts masks from their rank scores.
end for
Sort the results list and select top-K results.

Figure 4.5: Top K search

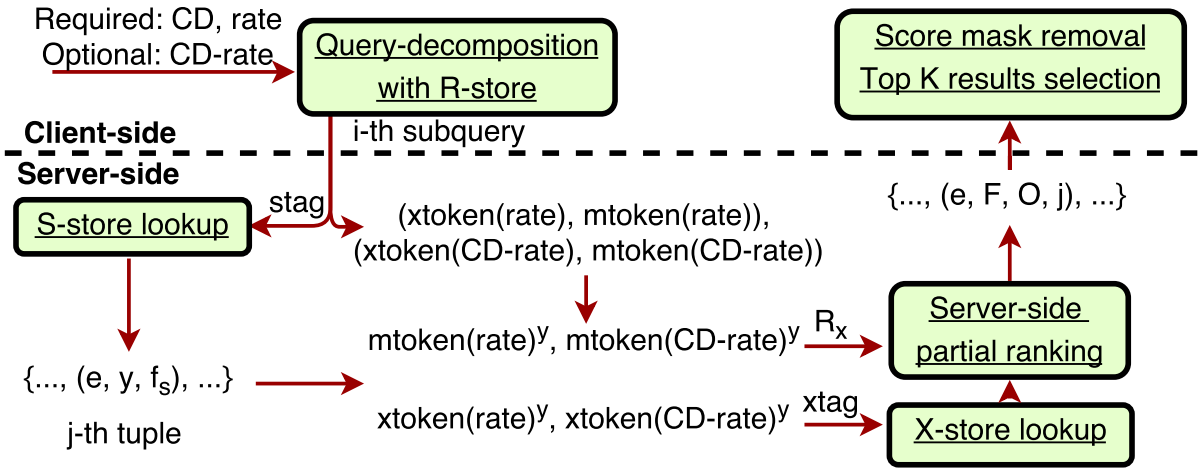


Figure 4.6: Example of client-server query processing

4.3.6 Ranking Blinding

To prevent leakage of feature values in the OXTF algorithm, we add blinding to each f value specific to each term through modular addition. More formally, we redefine f in the above algorithms as $f + RmodN$ with N set to 2^{32} for our implementation, and R computed as a keyed PRF of the term and document id. When the client receives back the results, they can deblind the final score by computing the deblinds for each term-document pair and subtracting out the deblinds from the blinded final score. This gives us secure scoring (but not ranking) when server-side ranking is not enabled. For server-side ranking, we compute R only as a function of the term, so the client can pre-compute score deblinding values and send those to the server along with the query. Then the server can deblind the final scores to sort the results and send back the top-k documents.

Note that while the Level 2 protection may offer some additional practical privacy, this scheme can be easily broken. Other areas of research (e.g. functional or homomorphic encryption) may be able to provide privacy preserving ranking instead of just privacy preserving score calculation (though even just leaking top-k doc-ids is significant leakage).

Example. Figure 4.6 gives a runtime processing example for query "CD rate" with optional term "CD-rate". After query decomposition and subqueries are generated, assume "CD" is the start feature of this subquery. The key to access S-store is $stag = PRF(k, \text{“}CD\text{”}\|c)$ for chunk c of this feature's posting.

After accessing S-store, assume d is a candidate document found from the posting of "CD" with corresponding tuple (e, y, f_s). To further access the X-store for features "rate", the server computes the access key as $xtag = xtoken_2^y$ and $R_x = PRF(k_u, mtoken_2{}^y)$. Since $y = PRF(k_4, \text{“}CD\text{”}\|p)^{-1}PRF(k_2, d)$ based on the index setup algorithm, we can verify that the server actually obtains the same values $xtag = g^{PRF(k_5,\text{“}rate\text{”})PRF(k_2,d)}$ and $R_x = PRF(k_u, g^{PRF(k_1,\text{“}rate\text{”})PRF(k_2,d)})$, which match the index setup algorithm in building X-store. It means that without knowing values k_1, k_2, and k_5, the server can fetch and add the feature weight of word "rate" successfully after the query-specific authorization information such as the token array is issued by the client.

The final result of $xtag_i$ is $g^{PRF(k_9,w_i)PRF(k_5,d)}$, which is an encrypted format of The key to lookup the X-store is an encrypted form of (w, d) and the corresponding value includes weight f_i^d.

Accessed with a stag key and its value includes a set of entries Each entry is the same as the Tset from the previous work [15], and stores an encrypted posting using the same locator and e, y values, plus the encrypted feature and y'.

As discussed in Section 4.2, the posting lists are divided into chunks and the storage implementation uses a partitioned hashtable. and the corresponding S-store is partitioned accordingly by posting list length to allow for $O(n)$ storage for a posting list with n entries. So if the partition size is 100, and the term appears in 10000 documents, the posting list will be split into 100 partitions which are stored separately in the S-store.

We store the S-store on disk as a hash-table of posting partitions. Each partition consists of a locator and a list of posting entries, where each entry has a label based on

the *stag* plus an encrypted tuple containing $\{e, y, E(k_{s,i}, f), y'\}$. e is stored using some non-deterministic encryption so the server cannot compare e values, but only send them back to the client as intersection results. y, y' are blinded per-term, so again the server cannot compare entries from different postings.

4.3.7 Search Complexity

The size of the index is proportional to the total number of nonzero feature values. The encrypted values cannot be compressed well because of randomization, so the index space cost is also directly proportional to the number of feature values. Given a query with n features, let $\sum(Posting(w_1))$ denote the sum of the posting length of the start feature w_1 for each subquery. The time cost for this query is $O((n-1)\sum|Posting(w_1)|)$. It is slower than that of [69] which does not need to consider privacy-preserving and this represents a tradeoff for private search.

4.3.8 Properties of Query Processing and Leakage Discussion

Theorem 1 *If a feature ID has never been used in any search query, the server cannot learn the corresponding feature weight of any document.*

The feature value f of document d stored in the X-store is encrypted as $f + R_c + R_x$ mod N where

$$R_x = PRF(k_u, g^{PRF(k_7,w)PRF(k_8,d)}).$$

As R_x is the document and term specific, without R_x value, the server cannot decrypt or discover $f + R_c$.

Similarly for the feature f of d stored in the S-store, it is encrypted as $f_s = f + R_c + R_s$ mod N where $R_s = PRF(k_3, w\|c)$. In addition there is a posting-entry encryption so

that $E(skey, \{e, y, f_s, y'\})\}$ where $skey = PRF(k_2, w\|c)$. Since p is different for different document IDs under the same term posting, without R_s and $skey$, the server cannot learn the feature value.

Theorem 2 *The server cannot learn the feature weight of a document for any unpopular word during or after query processing. That is true also for any popular word of a search query which involves at least one unpopular required word.*

From the previous theorem, the server does not learn any information if such a word is not involved in a search query.

For a query that involves an unpopular word, result filtering is not triggered and the server simply adds feature weights blinded with term and document specific masks and it cannot learn anything about the corresponding feature weight of each document and its final rank score.

The feature mask R_c for a popular word is shared among documents within the same chunk, but can only be computed using the secret key from the client. This ensures that the server cannot learn the masked weights of a feature unless result filtering is enabled for a query involving that term chunk. For any popular word that has only appeared in search queries without resulting filtering enabled, the server cannot learn anything about the corresponding feature.

Feature types		Unsearched	Searched		
			Filter off	Filter on	
				not candidate	candidate
Required	Popular	no	no	no	yes
	Unpopular	no	no	no	no
Optional	Popular	no	no	no	yes
	Unpopular	no	no	no	yes

Table 4.3: Chunk-wide feature difference leakage

If a feature from the above theorems, such as a word, has never appeared in any past search query, the server is unable to inspect its feature values so there is no leaked

information to the server about the values for this feature. For features have have been used only in queries with at least one unpopular term or queries without ranking enabled, the server also learns nothing about these feature values except their existence (from the intersection). In the case of queries that do not trigger partial server side ranking, the server learns no information about the individual feature values or the final scores of the documents, and can only calculate blinded scores to send back to the client.

In our evaluation with chunk-wide masks is 10,000, the percentage of popular words for CSIRO, TREC45, Aquaint, Clueweb and Enron are respectively 0.07%, 0.31%, 0.47%, 0.07%, and 0.1%. 8.7% of the tested TREC multiword queries enable server-side ranking. For such queries, chunk-wide random masks enable the comparison of rank scores of documents matched in each optional feature case under a subquery and then the server can learn their relative rank order. The server may also learn while we try to make server side ranking computation as private as possible. Our work leverages secure search techniques from the prior encryption research (e.g. [15]). One strategy to restrict such leakage to a smaller scope is to make the relative ratio of chunk size over the posting length threshold as small as possible. In our tested datasets, this ratio is set to 2.1% with chunk size 210, since this is only for words that appear in over 10,000 documents.

4.4 Evaluation

4.4.1 Implementation and Datasets

We have implemented a prototype system with C++. Our evaluations have the following objectives:

1. Examine another searchable encryption scheme with supporting ranking, in both theoretical and experimental ways, which supplies a comparison to evaluate the

efficiency of our system.

2. Demonstrate the average response time in query processing.

3. Assess the impact of query decomposition, random s-term selection, and server-side result filtering.

4. Illustrate the tradeoff when restricting the size of optional features.

5. Compare the relevance score of the linear additive framework with a tree boosting scheme. We report the query response time and relevance score such as NDCG@10.

6. Analyze the performance for serving multi-user in single machine.

Experiments are conducted on Linux Ubuntu 16.04 servers with 8 cores of 2.4 GHz AMD FX8320, 16GB memory. The code is compiled with optimization flag -O3. The following datasets are used: the TREC Disk 4&5 dataset with TREC Robust 2004 topics 301-450 &601-700, the CSIRO dataset with the TREC 2008 Enterprise Track Topics (CE051-CE127) queries, and the Aquaint dataset with TREC Robust 2005 query set of 50 topics. Table 4.4 summarizes their key characteristics after stemming and index generation. Row 2 is the number of documents. Row 3 is the number of term and document ID pairs. Row 4 is the number of composite term and document ID pairs. An optional term composed of two words appears in the X-store with a document if these two words appear in a document within distance 9. Note the majority of the encrypted index size comes from these optional terms. If lower ranking relevance is acceptable the distance limit could be reduced to significantly reduce the size because X-store size is linearly proportional to the total number of document-feature pairs. Row 5, 6 and 7 are the size of R-store, S-store and X-store in bytes. Row 8 is the total index size in bytes. As the machines we run experiments are shared in a cloud environment, we control the search memory

usage to under 2GB per machine. Due to memory constraints, we opt to use 1 server for CSIRO while distributing data to 8 and 5 servers respectively for Aquaint and TREC45. Intersection and score accumulation is conducted in parallel on those servers. At the end of this section, we also present our investigation in using a larger dataset from ClueWeb and the Enron email dataset.

TREC Disk 4&5 has 528,155 documents averaging 484 words/doc, CSIRO has 370,715 documents averaging 184 words/doc, and Aquaint has 1,033,461 documents averaging 440 words/doc. After stemming and inverted index generation, TREC Disk 4&5 contains 674875 unique terms with on average 165 entries/posting list, CSIRO contains 580377 unique terms with on average 39 entries/posting list, and Aquaint contains 716919 unique terms with on average 309 entries/posting list.

Dataset	CSIRO	TREC45	Aquaint
#Doc	0.37M	0.53M	1.03M
word-doc	22M	109M	216M
wordpair-doc	146M	712M	1,357M
R-Store	0.31GB	1.25GB	1.13GB
S-Store	1.12GB	5.56GB	11.02GB
X-Store	2.42GB	11.82GB	22.53GB
Total Size	3.85GB	18.63GB	34.68GB

Table 4.4: Size characteristics of datasets

For each document, we select features that perform well for text ranking based on the past work [23, 24, 25, 26, 27]. We generate the feature value of a word that this document contains based on the BM25 coefficient differentiated by where they appear (title and body). We also compute optional features based on word pairs that appear within a limit in a document. The feature value for the word pairs is based on the squared inverse of word distance following the work in [26] and if such a pair appears in the title or body of its document. We use AdaRank [67] to guide feature weighting and relevance evaluation follows 3-fold cross-validation. Notice that with the conjunctive query requirement, our document matching retrieves may miss some labeled relevant results in some queries

compared to a disjunctive assumption, which results in up-to 10% relevance difference in some cases.

At runtime, preprocessing of long queries can add a large number of word-pair based optional terms. This addition causes a long bitmap to record optional term matching and more matched results incomparable at the server side. We impose the following indexing and online constraints as a tradeoff to reduce server-client communication overhead and filter more results sent back to the client. A word pair is included in the index only when the word distance of such a pair is within a limit. Our evaluation has used limit 9 in processing three test datasets. We also aggregate the features under the same term as one feature. For example, text weights of a word pair that appears in different sections of a document are combined together. Now we limit the number of optional features during query processing. Given q words in a query, there are $\binom{q}{2}$ word pairs, and if we let $m = \binom{q}{2}$, there are $2^{\binom{q}{2}}$ lattice cases of using optional features. When $q = 5$, there are already 2^{10} cases. When $q = 7$, the lattice size explodes to 2^{21}. Thus it is not practical to let $m = \binom{q}{2}$ when q is large. Figure 4.7 shows the cases for 3 required and optional terms. The server prepares different top K results for different optional feature matching cases and sends all results back to the client. The goal of top K ranking is to minimize the number of results sent back to the client from the host and if we restrict the choices of optional terms, our evaluation in Section 4.4 shows that the relevance impact is small. To address the aforementioned challenge, we take a compromised approach. For online query processing, a client only demands optional word pairs that are within a distance limit L within the query.

Figure 4.8 depicts the use of query word distance limit $L = 2$ when $q = 5$. For example with a 5 word query $w_1, \cdots, w_5$, optional word pair terms considered with $L = 2$ only include: (w_1, w_2), (w_1, w_3), (w_2, w_3), (w_2, w_4), (w_3, w_4), (w_3, w_5), and (w_4, w_5). In general, the number of optional features m to consider based on word pairs is reduced from $\binom{q}{2}$ to

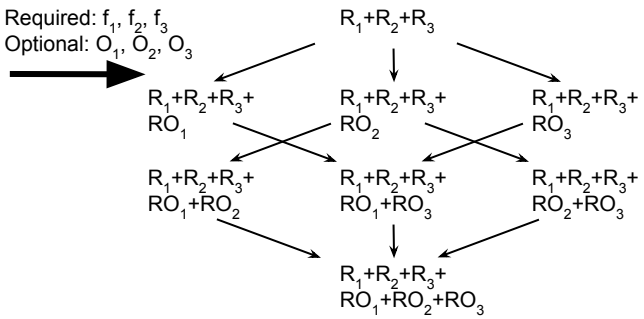


Figure 4.7: The lattice relationship for optional feature matching cases when $q = 3$ and $m = 3$.

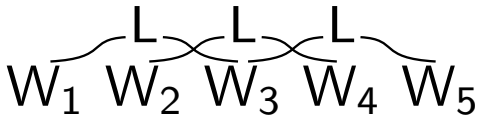


Figure 4.8: Distance restriction when $q = 5$ and $L = 2$

$\binom{L}{2} + L(q - L)$ if $q \geq L$, otherwise $\binom{q}{2}$. When we choose $L = 2$, $m = 2q - 3$. In practice, we can assume $q < 10$ because, typically, trimming a query longer than such a length does not affect the relevance. Thus $m \leq 15$ when optional terms are only based on word pairs. In some applications, additional optional terms may be needed to improve ranking relevance such as freshness or authoritativeness of a document. Thus m is typically under 24 and we use this number in our evaluation.

Each document result returned from the server on average uses about 16 bytes and this includes an 8-byte encrypted document ID, 4-byte blinded score, 3-byte bitmap, and 1-byte position. We select 10,000 as the result size threshold that triggers server-side partial ranking. Notice that the average Internet connection globally has reached an average speed of 7.2Mbps in 2017 [76]. With this average connection speed, it takes 0.18 seconds to transmit 10,000 matched results with 16 bytes per result.

4.4.2 Multiword Search Time

Table 4.5 lists the average search response time and its cost breakdown when the number of search words varies. A query response time includes the client-side time and the server side time for S-store lookup of candidate documents, and X-store lookup for intersection and score accumulation. The client time reported includes R-store intersection and the token generation for the first subquery, and the server starts to search as soon as it receives the first subquery.

There is a time overlap of client-side token generation for other subqueries and server-site subquery processing. We count the response time started from the time when the client sends the first subquery to the time when the client completes the top K selection after receiving all top candidate results. The S-store time to extract the document candidates depends on the length of postings identified from the S-store in processing the associated subqueries. From these 3 query sets, the number of candidate documents increases from $q = 2$ to $q = 3$, but decreases from $q = 3$ to $q = 3 or 4$. That impacts the X-store intersection cost. There is no cost involved in X-store operations for $q = 1$ and its time is listed so we can observe the average time difference between single-word and multiword queries. When q increases from 2 to 3, the X-store time increases as more optional features are involved. However, as there are less candidates found from S-store when q is 4 and 5 after earlier range intersection, the X-store time drops in all three query sets. The client-side time is modest, and it increases in all three datasets with more query words because there is more cost for range intersection and x-token generation. Time to compute a x-token is not insignificant because of integer exponentiation involved.

Table 4.6 lists the average search response time for 5 query posting length buckets. Each query falls into one column bucket based on the sum of the start feature posting length of its subqueries. This table validates that the query response time is

# Query words q		1	2	3	4-5
CSIRO	Client	30.53	59.98	74.89	101.16
	S-store	58.31	121.57	140.40	37.60
	X-store	0	59.21	137.87	64.09
	Total(ms)	89.62	283.86	427.98	248.51
TREC45	Client	18.89	45.18	65.56	107.22
	S-Store	146.79	191.30	222.33	119.67
	X-Store	0	85.56	284.15	260.11
	Total(ms)	166.42	405.64	717.34	693.00
Aquaint	Client	25.87	47.38	54.17	66.35
	S-store	261.81	147.60	146.67	90.60
	X-store	0	89.47	218.52	200.95
	Total(ms)	289.35	337.06	496.20	400.26

Table 4.5: Query processing cost of three datasets

approximately proportional to the posting length sum of the selected start features of all subqueries as discussed in Section 4.3.1.

# Posting length (thousands)	0 - 2	2 - 4	4 - 6	6 - 8	8 - 10
CSIRO (ms)	186.00	392.34	499.50	801.40	973.65
TREC45 (ms)	217.81	453.23	558.27	838.78	1049.26
Aquaint (ms)	134.47	343.56	479.21	770.15	909.97

Table 4.6: Impact of posting length on search time

4.4.3 A Comparison with the Baseline

We have also implemented and evaluated the baseline approach based on matrix transformation in [30, 58] with the same machine resource setting. It takes over 48, 26, and 46 hours to process a query on average for CSIRO, TREC45, and Aquaint with 1, 5, and 8 machines, respectively. The forward index space cost is 105, 285, and 815 terabytes respectively as encrypted feature vectors are dense. The high cost is caused by the large number of features (T), which is about 39M, 73M, and 108M respectively in these datasets. When restricting the offline word-pair distance to be within 3 instead of 10, T is reduced approximately by half on average and the above cost is also shortened by half. Our approach is still several orders of magnitude faster. An optimization using

tree search to speed up similarity computing as $O(T \log D)$ is discussed in [58], but the worst case is still $O(TD)$. Recall that D is the number of documents.

Table 4.7 shows the in-memory search time and index space cost. The second column is the total number of features. The search time only includes in-memory time with one core to compute dot-product similarity with all dense document vectors. It does not include the query transform time which can vary from a few minutes to up to 785 hours depending on the matrix choice, and does not include time to load data from disk to memory. The cost is so high so we have to sample and estimate the actual time it would take. For the TREC45 dataset with about 0.6 millions unique single terms, the baseline approach takes 53.71 mins on average to conduct a search. The most time consuming part is query preprocessing to conduct matrix-vector multiplications. When word pairs within distance 3 in TREC45 are considered, the baseline approach is too slow to respond. With the estimation on the TREC45 dataset that has about 42 millions proximity features, it would take 541.56 hours on average for search, where 489.10 hours are spent on query preprocessing and 52.45 hours are spent on computing similarities between query vectors and document vectors. In addition, the baseline approach requires huge storage space in the server-side, due to the fact that the server has to compute and store all document vectors in advance. Note that each document vector is a dense vector and its dimension is the number of features. For the TREC45 dataset, it costs 2.29 TB to save all these document vectors.

Dataset	#Features	Similarity time	Index size
CSIRO	16M	20.2 hours	58.9TB
TREC45	42M	52.5 hours	152.7TB
Aquaint	53M	66.5 hours	193.4TB

Table 4.7: Baseline similarity computing time and space cost

4.4.4 Impact of Query Decomposition

Query decomposition with chunked postings enables earlier range intersection, and S-store and X-store locality-aware partitioning. For range earlier intersection during query decomposition, we find that the intersection scope is reduced by 6%, 12%, and 21% for CSIRO, TREC45, and Aquaint, respectively. Without partitioning X-store, compared to a 1024-partition, search is about 42.5x, 17x, and 19.1x slower for queries with 2 words, 3 words, and 4-5 words, respectively. When the server hosts multiple encrypted indices for different users, there is a performance impact as multiple search programs for different users fight for the same memory resource and we have observed about up to 30% response time variation from a single-user environment to a 5-user. Thus the above decomposition is important to sustain low query response times.

While this reduction is highly data and query dependent, the main purpose of subquery conversion is to facilitate random s-term selection and X-set partitioning. Even s-term randomization makes search slower, earlier range intersection and X-store partitioning with subqueries, as shown below, brings efficiency advantages.

The R-store lets the client compute the number of subqueries to send to the server, so it does not have to keep sending subqueries until the server goes past the end of the s-term-posting. It trivially does this, since after range intersection the client knows exactly how many subqueries will be involved in the intersection.

The R-store reduces the total size of the query by letting the client skip subqueries or xtokens that are guaranteed to have no entries. Based on our evaluation of 3 corpuses and query sets, the effectiveness of the R-store at reducing query-size is highly data and query-dependent. However, for longer queries the average savings of the R-store are greater, which makes sense because more query terms means smaller (or equivalent) intersection set, and more query terms also means more optional terms which may or

may not intersect with the required terms. The R-store makes possible selecting different query terms as the s-term for different subqueries. So long as the set of subqueries covers all potentially intersecting document id ranges, correct search will be achieved, and a smaller number of s-term posting chunks can be leaked by correct s-term selection. We leave the analysis of this s-term switching as future work.

#Results returned		TREC Queries	Synthetic
CSIRO	No filter	11,607	33,093
	Chunk 105	2,504	8,884
	Chunk 210	1,288	6,159
TREC45	No filter	20,985	127,538
	Chunk 105	14,151	28,876
	Chunk 210	8,708	20,596
Aquaint	No filter	32,896	185,139
	Chunk 105	25,561	38,333
	Chunk 210	16,112	22,437

Table 4.8: Return result reduction in top-10 search with threshold 10,000.

4.4.5 Effectiveness of Server Partial Ranking

The average number of results returned from the sever for all 154 TREC queries without top K filtering in CSIRO, TREC45, and Aquaint is 849, 5490, and 5704 respectively. Table 4.9 shows the average number of results for top K search with $K = 10$. The threshold to trigger partial ranking is 10,000 as discussed above. Column 3 is the average number of results for the 23 TREC queries that trigger partial ranking with posting chunk size 105 and 201 respectively. Rows marked as "No filter" are for the setting that search does not use server-side partial ranking.

The number of results in other queries without triggering partial ranking is below the predefined threshold 10,000. Below such a threshold, the system does not trigger the server-side result filtering and it simply returns all matched results with rank scores to the client.

For Aquaint, there are 13 TREC queries triggered result filtering and 16,112 results are returned on average with chunk size 210 after filtering out 51% of all matched results. For a larger chunk size, there are more comparable matched documents under each subquery and there are more results filtered during partial ranking.

To evaluate the extreme situations, Column 4 is for handling 15 synthetic queries per dataset built based on only popular and stop words with a much larger number of return results. The partial ranking scheme is able to filter out more unnecessary return results also.

For Aquaint, the server returns 22,437 results on average for synthetic queries after filtering out 87.9% of all matched results. In this case with a 7.2Mbps average global Internet connection speed [76], result communication takes about 0.39 seconds.

The R-store lets the client compute the number of subqueries to send to the server, so it does not have to keep sending subqueries until the server goes past the end of the s-term-posting. It trivially does this, since after range intersection the client knows exactly how many subqueries will be involved in the intersection.

The R-store reduces the total size of the query by letting the client skip subqueries or xtokens that are guaranteed to have no entries. Based on our evaluation of 3 corpuses and query sets, the effectiveness of the R-store at reducing query-size is highly data and query-dependent. However, for longer queries the average savings of the R-store are greater, which makes sense because more query terms means smaller (or equivalent) intersection set, and more query terms also means more optional terms which may or may not intersect with the required terms. The R-store makes possible selecting different query terms as the s-term for different subqueries. So long as the set of subqueries covers all potentially intersecting document id ranges, correct search will be achieved, and a smaller number of s-term posting chunks can be leaked by correct s-term selection. We leave the analysis of this s-term switching as future work.

#Results returned		TREC Queries	Synthetic
CSIRO	No filter	11,607	33,093
	Chunk 105	2,504	8,884
	Chunk 210	1,288	6,159
TREC45	No filter	20,985	127,538
	Chunk 105	14,151	28,876
	Chunk 210	8,708	20,596
Aquaint	No filter	32,896	185,139
	Chunk 105	25,561	38,333
	Chunk 210	16,112	22,437

Table 4.9: Return result reduction in top-10 search with threshold 10,000.

4.4.6 Effectiveness of Server Partial Ranking

The average number of results returned from the sever for all 154 TREC queries without top K filtering in CSIRO, TREC45, and Aquaint is 849, 5490, and 5704 respectively. Table 4.9 shows the average number of results for top K search with $K = 10$. The threshold to trigger partial ranking is 10,000 as discussed above. Column 3 is the average number of results for the 23 TREC queries that trigger partial ranking with posting chunk size 105 and 201 respectively. Rows marked as "No filter" are for the setting that search does not use server-side partial ranking.

The number of results in other queries without triggering partial ranking is below the predefined threshold 10,000. Below such a threshold, the system does not trigger the server-side result filtering and it simply returns all matched results with rank scores to the client.

For Aquaint, there are 13 TREC queries triggered result filtering and 16,112 results are returned on average with chunk size 210 after filtering out 51% of all matched results. For a larger chunk size, there are more comparable matched documents under each subquery and there are more results filtered during partial ranking.

To evaluate the extreme situations, Column 4 is for handling 15 synthetic queries per dataset built based on only popular and stop words with a much larger number of return

results. The partial ranking scheme is able to filter out more unnecessary return results also.

For Aquaint, the server returns 22,437 results on average for synthetic queries after filtering out 87.9% of all matched results. In this case with a 7.2Mbps average global Internet connection speed [76], result communication takes about 0.39 seconds.

Impact of posting decomposition and client-side range intersection. By doing range intersections before search, the number of X-Store chunks needed to be searched is minimized. Here we present the impact of range intersection. Here the max bigram distance is 10. We partition a X-Store into 1024 parts.

We assess the relevance of the linear additive ranking model with tree-boosting LambdaMART [68] in using these datasets. Our system generates features based on TFIDF and BM25 of terms differentiated by where they appear (title and body). We also compute optional features based on word pairs that appear within a limit in a document, and gather the frequency, TFIDF, and BM25 of these composite terms. In addition we gather features based on the inverse of word distance, with several variations following We use AdaRank [67] to guide feature weighting and compare this linear ranking model with LambdaMART. All experiments are using 3 folds cross-validation. The number of trees in LambdaMART and number of iterations in AdaRank range up to 200.

The accuracy of using the linear additive model for our dataset is still modestly competitive to the state-of-the-art GBRT model.

4.4.7 Relevancy Impact with Linear Additive Scoring and the Number of Optional Terms

Table 4.10 illustrates the NDCG@10 score of the linear model and LambdaMART [68] under different query word distance constraints $L = 2, 5, \infty$. The distance restriction does

not negatively degrade performance. In the test model, the BM25 based features carry more significance. Compared to LambdaMART, the linear scoring model delivers decent performance. Our solution performs more competitively for TREC45 and Aquaint while it is worse by about 0.05 point for CSRIO compared to LambdaMART. While linear scoring is the best ranking model we can support so far for private ranking, it is interesting to investigate private tree boosting in the future, and our work in [70] starts this.

NDCG@10		L=2	L=5	L=∞
CSIRO	LambdaMART	0.4836	0.4842	0.4789
	Linear	0.4311	0.4046	0.4317
TREC45	LambdaMART	0.3823	0.3898	0.3866
	Linear	0.4121	0.4012	0.3808
Aquaint	LambdaMART	0.3256	0.3246	0.3131
	Linear	0.3118	0.3227	0.317

Table 4.10: Relevance with different numbers of word-pairs

4.4.8 ClueWeb

We have also investigated how our scheme handles a larger dataset using ClueWeb09 Category B dataset [77] with 50 million web documents. The average posting length of words in this dataset is 180 with 0.07% of them as popular words. We evaluate relevancy of linear additive formula with query word distance restriction $L = 2, 5$, and ∞ using queries and relevance set from the 2009 Million Query TREC [78] prels relevance judgments. The 2009 Million Query TREC collection [78]. The relevancy raw features include BM25-weighted single word weights and word pairs in the title and body sections, and document PageRank scores. The top team in the 2011 Web TREC has accomplished NDCG@20 of 0.24282 in [79] using the TREC 2011 Queries plus 450 out of the 684 queries from Million Query (MQ) TREC 2009 [78]. LambdaMART using our raw features obtains NDCG@20 of 0.2547, 0.2554, and 0.2601 with query word distance restriction

$L = 2$, $L = 5$, and $L = \infty$, respectively. The performance gap is partially caused by the conjunctive assumption we follow. The additive ranking scheme we use obtains 0.2512, 0.2530, and 0.2585 under these L values which still delivers a decent performance within an acceptable gap compared to LambdaMART. For our ranking scheme, we tested over the 2009 MQ TREC queries for an NCDG@20 of 0.2087 using word distance restriction $L = 2$, 0.2097 using $L = 5$, and 0.2105 using $L = \infty$. Our results are close to state of the art search techniques [79].

The search time for ClueWeb data is still reasonable as the client-side query processing does not require large memory and the server side can use more machines to distribute S-store and X-store for parallel processing. The client-server communication cost is a key concern and thus we select several different subsets of this ClueWeb dataset to assess the impact of database size change on server-side partial ranking. For all 684 queries from MQ TREC 2009 with judgment labels, Table 4.11 lists the average return result size from Row 3 to Row 4 with two chunk sizes when the database size chosen varies from 3M to 50M ClubWeb documents. Row 6 and Row 7 show the average return result size for queries for top 10% of the largest result sizes after server-side filtering. When the database size is 5M and chunk size is 210, it would take about 0.46 seconds on average for the server to send 25,394 results back to the client with today's global average Internet speed 7.2 Mbps. The slowest 10% of queries can take 1.93 seconds with 107,195 results. Notice the top country-level average Internet connection is 28.6 Mbps [76] and with such a speed, the slowest 10% would take 0.486 seconds to deliver. The proposed scheme can be suitable for clients with higher speed Internet connection. Considering 5G mobile network is being tested for 5Gbps or 10Gbps connections,

In specific, we simulate the results filtering algorithm and test its effectiveness on several subsets of ClueWeb datasets with 1 million, 3 millions, 5 millions, 10 millions, 30 millions and 50 millions. We also varies the chunk size to evalute its impact on filtering

effectiveness.

# Results returned		3M	5M	10M	30M	50M
All	No Filter	65,449	81,445	113,173	218,027	321,769
	Chunk 105	25,273	31,866	46,442	90,138	134,240
	Chunk 210	19,992	25,394	37,413	73,017	108,892
Top 10%	No Filter	393,650	506,452	752,328	1,619,203	2,465,084
	Chunk 105	97,710	147,156	234,872	508,292	812,515
	Chunk 210	69,056	107,376	173,195	374,309	608,736

Table 4.11: Return result sizes in top-10 search with threshold 10,000 for ClueWeb subsets varying from 3M to 50M

4.4.9 Enron Email Dataset

We have also studied the corporate email collection [80] with about 0.5M messages from about 150 users. The average posting length of words in this dataset is 65 with 0.1% of them as popular words. Like other datasets we have studied, the posting length follows a long-tail Zipf distribution. We use 100 personal names from the email collection as test queries, and find all queries only contain unpopular words. Namely all of them have results less than 10,000 and the server only computes rank scores without filtering. If we lower the threshold to 1000, then the result filtering ratio after server-side partial ranking is over 50% on average. Because each email message is short, searching this dataset is much faster than TREC45 with a similar size.

4.5 Concluding Remarks

The contribution of this work is a novel private top K ranking scheme with various tradeoffs to efficiently search cloud hosted data. A privacy analysis on limited leakage is provided. The proposed term and query decomposition with chunked postings enables reduced key-value lookup cost to support efficient private document scoring. For queries that only involve popular words and may have too many results to send back with

reasonable latency, a tradeoff is adopted to reduce the return set size for top K ranking while leaking some information from the chunk-wide masking. Posting decomposition also allows blocked X-store distribution to speedup the intersection and feature value lookup performance. The client-server collaboration allows the server to conduct the first round of top result selection, and to speedup the server-side intersection and scoring while preserving the privacy of the feature values. Our experiment shows that the response time is reasonable, and partial server ranking can effectively reduce the return result size for the tested datasets. This demonstrates the benefits of the earlier client-side range intersection, and it also shows the server result filtering reduces the number of results that need to be sent to the client. The number of optional features does affect efficiency and our experiment with word distance pairs shows a tradeoff is possible while still achieving a good relevance.

Feature value blinding with random offsets allows the server to conduct score addition with partial score comparison. Can such a method be useful for storing relative word positions of each document in the index and deriving word distance or other proximity formulas more space efficiently? Unfortunately it is not safe and leaking relative word positions of each document can lead to a statistics-based leakage abuse attack [19] and may reveal the word structure of a document. The developed techniques address this open problem for a modest sized dataset. For a significantly larger dataset, the current techniques require fast client-server connection since the result size returned from the server even after partial ranking can be significant when all query words are popular. Deployment of faster Internet connection such as mobile 5G networks could extend the applicability of the proposed approach together with additional improvements. Another potential strategy to support a larger dataset is to adopt a trusted proxy with a fast server-proxy communication connection, if available. Then final supplemental ranking could be completed at such a proxy and the ranked results can be sent to the client with a

smaller communication cost. Recently there is an advancement in neural network based ranking (e.g. [81]) and our techniques can be applicable for document pre-ranking used in such work or directly integrating neural network features.

Chapter 5

Conclusion and Future Works

5.0.1 Conclusion

Migrating existing systems to the cloud presents both new opportunities and new challenges. New opportunities are available for converging resources to simplify scaling and reduce the overall cost of server management, but new challenges present themselves including how to efficiently utilize these converged resources for managing storage. This dissertation investigates scalable techniques to address these challenges. In particular, our work focuses on 2 key problems: 1) how to efficiently deduplicate large numbers of daily snapshots for storage savings, 2) how to efficiently support rich search over cloud-stored data when such data is private and should not be leaked to the untrusted cloud server.

This dissertation starts with studying a low-profile parallel deduplication scheme for asynchronous backup. We distribute both the global index and the backup process over all machines to facilitate running the solution in a converged cloud architecture where backup is a secondary service and should not impact more primary services, which still achieving reasonable latency and high throughput.

I also presented a novel approach to the private search with top K ranking problem. Our solution extends prior private intersection techniques to support completely private document scoring, and also supports server-side partial top K filtering with some leakage to reduce the cost of sending results back to the client.

5.0.2 Future Work

This work includes the first efficient solution to the private search with top K ranking problem. In this work I only present an informal analysis of the damage that can be caused by the leakage from this scheme. Future work includes a more rigorous and theoretical analysis of the leakage, and more importantly concrete information that can be obtained through analysis of the leakage. The leakage itself is the source of the problem, but if an attacker can't reasonably learn anything non-trivial then the scheme could be said to be secure, while if a sufficiently advanced attacker could uncover the original plaintext of the corpus the scheme would be insecure, even if no such attack is currently known. Future work in this area needs to characterize the upper bound on what non-trivial knowledge an attacker could gain from this leakage, as well as characterize what non-trivial knowledge an attacker could reasonably obtain. Attackers with different levels of inside knowledge may have different upper bounds, so this also needs further study.

Bibliography

[1] M. Naveed, M. Prabhakaran, and C. A. Gunter, *Dynamic searchable encryption via blind storage*, SP '14, pp. 639–654, IEEE Computer Society, 2014.

[2] D. X. Song, D. Wagner, and A. Perrig, *Practical techniques for searches on encrypted data*, SP '00, IEEE Computer Society, 2000.

[3] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, *Searchable symmetric encryption: Improved definitions and efficient constructions*, CCS '06, pp. 79–88, ACM, 2006.

[4] "Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/."

[5] "Alibaba Aliyun. http://www.aliyun.com."

[6] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li, *Decentralized deduplication in san cluster file systems*, in *ATC'09*, USENIX.

[7] M. Vrable, S. Savage, and G. M. Voelker, *Cumulus: Filesystem backup to the cloud*, in *FAST'09*, pp. 225–238, USENIX.

[8] S. Quinlan and S. Dorward, *Venti: A New Approach to Archival Storage*, in *FAST'02*, pp. 89–101, USENIX.

[9] B. Zhu, K. Li, and H. Patterson, *Avoiding the disk bottleneck in the data domain deduplication file system*, in *FAST'08*, pp. 1–14, USENIX.

[10] "NetApp."

[11] EMC, *Achieving storage efficiency through EMC Celerra data deduplication. White Paper*, 2010.

[12] Nutanix, *Nutanix Complete Cluster, A Technical Whitepaper* , 2013.

[13] S. Kamara, C. Papamanthou, and T. Roeder, *Dynamic searchable symmetric encryption*, in *CCS'12*, pp. 965–976, 2012.

[14] S. Kamara and C. Papamanthou, *Parallel and dynamic searchable symmetric encryption*, in *FC 2013*, pp. 258–274, 2013.

[15] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, *Highly-scalable searchable symmetric encryption with support for boolean queries*, in *CRYPTO 2013*, pp. 353–373, 2013.

[16] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner, *Dynamic searchable encryption in very-large databases: Data structures and implementation*, in *NDSS 2014*, 2014.

[17] D. Cash and S. Tessaro, *The locality of searchable symmetric encryption*, in *EUROCRYPT 2014*, pp. 351–368, 2014.

[18] S. Kamara and T. Moataz, *Boolean searchable symmetric encryption with worst-case sub-linear complexity*, in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 94–124, Springer, 2017.

[19] M. S. Islam, M. Kuzu, and M. Kantarcioglu, *Access pattern disclosure on searchable encryption: Ramification, attack and mitigation*, in *NDSS 2012*, 2012.

[20] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, *Leakage-abuse attacks against searchable encryption*, in *CCS'15*, pp. 668–679, ACM, 2015.

[21] C. Gentry, *A Fully Homomorphic Encryption Scheme*. PhD thesis, 2009.

[22] P. Paillier, *Public-key cryptosystems based on composite degree residuosity classes*, in *EUROCRYPT '99*, pp. 223–238, 1999.

[23] T. Tao and C. Zhai, *An exploration of proximity measures in information retrieval*, SIGIR '07, pp. 295–302, ACM, 2007.

[24] S. Büttcher, C. L. A. Clarke, and B. Lushman, *Term proximity scoring for ad-hoc retrieval on very large text collections*, SIGIR '06, pp. 621–622, ACM.

[25] J. Bai, Y. Chang, H. Cui, Z. Zheng, G. Sun, and X. Li, *Investigation of partial query proximity in web search*, WWW '08, pp. 1183–1184, 2008.

[26] J. Zhao and J. X. Huang, *An enhanced context-sensitive proximity model for probabilistic information retrieval*, in *SIGIR*, pp. 1131–1134, 2014.

[27] K. M. Svore, P. H. Kanani, and N. Khan, *How good is a span of terms?: exploiting proximity to improve web retrieval*, SIGIR '10, pp. 154–161, ACM, 2010.

[28] E. Agichtein, E. Brill, S. Dumais, and R. Ragno, *Learning user interaction models for predicting web search result preferences*, in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, (New York, NY, USA), pp. 3–10, ACM, 2006.

[29] D. Pouliot and C. V. Wright, *The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption*, in *CCS'16*, pp. 1341–1352, ACM, 2016.

[30] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou, *Privacy-preserving multi-keyword ranked search over encrypted cloud data*, *IEEE Trans. Parallel Distrib. Syst.* **25** (2014), no. 1 222–233.

[31] W. Sun, B. Wang, N. Cao, M. Li, W. Lou, Y. T. Hou, and H. Li, *Verifiable privacy-preserving multi-keyword text search in the cloud supporting similarity-based ranking*, *IEEE Trans. Parallel Distrib. Syst.* **25** (2014), no. 11 3025–3035.

[32] C. Örencik and E. Savas, *An efficient privacy-preserving multi-keyword search over encrypted cloud data with ranking*, *Distributed and Parallel Databases* **32** (2014), no. 1 119–160.

[33] O. Goldreich and R. Ostrovsky, *Software protection and simulation on oblivious rams*, *J. ACM* **43** (May, 1996) 431–473.

[34] E. Stefanov, E. Shi, and D. X. Song, *Towards practical oblivious RAM*, in *NDSS 2012*, 2012.

[35] Y. Tan, H. Jiang, D. Feng, L. Tian, Z. Yan, and G. Zhou, *SAM: A semantic-aware multi-tiered source de-duplication framework for cloud backup*, in *39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010*, pp. 614–623, 2010.

[36] Y. Fu, H. Jiang, N. Xiao, L. Tian, F. Liu, and L. Xu, *Application-aware local-global source deduplication for cloud backup services of personal storage*, *IEEE Trans. Parallel Distrib. Syst.* **25** (2014), no. 5 1155–1165.

[37] J. Wendt, *A Candid Examination of Data Deduplication. White Paper, Symantec Corporation* , 2009.

[38] F. Guo and P. Efstathopoulos, *Building a high-performance deduplication system*, in *ATC'11*, USENIX.

[39] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane, *Tradeoffs in scalable data routing for deduplication clusters*, in *FAST'11*, pp. 2–2, USENIX.

[40] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge, *Extreme Binning: Scalable, parallel deduplication for chunk-based file backup*, in *MASCOTS'09*, pp. 1–9, IEEE.

[41] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, *Design tradeoffs for data deduplication performance in backup workloads*, in *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pp. 331–344, 2015.

[42] W. Zhang, H. Tang, H. Jiang, T. Yang, X. Li, and Y. Zeng, *Multi-level selective deduplication for vm snapshots in cloud storage*, in *IEEE CLOUD'12*, pp. 550–557.

[43] S. Rhea, R. Cox, and A. Pesterev, *Fast, inexpensive content-addressed storage in foundation*, in *ATC'08*, pp. 143–156, USENIX.

[44] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, *Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality*, in *FAST'09*, pp. 111–123, USENIX.

[45] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, *idedup: latency-aware, inline data deduplication for primary storage*, in *FAST'12*, pp. 24–24, USENIX.

[46] W. Zhang, T. Yang, G. Narayanasamy, and H. Tang, *Low-cost data deduplication for virtual machine backup in cloud storage*, in *HotStorage'13*, USENIX.

[47] D. Agun, T. Yang, and W. Zhang, *Low-profile source-side deduplication for virtual machine backup*, in *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[48] W. Zhang, D. Agun, T. Yang, R. Wolski, and H. Tang, *Vm-centric snapshot deduplication for cloud data backup*, in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, IEEE, 2015.

[49] P. Strzelczak, E. Adamczyk, U. Herman-Izycka, J. Sakowicz, L. Slusarczyk, J. Wrona, and C. Dubnicki, *Concurrent deletion in a distributed content-addressable storage system with global deduplication*, in *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, pp. 161–174, 2013.

[50] B. H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, *Commun. ACM* **13** (1970), no. 7 422–426.

[51] M. O. Rabin *et. al.*, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[52] F. C. Botelho, P. Shilane, N. Garg, and W. Hsu, *Memory efficient sanitization of a deduplicated storage system*, in *FAST'13*, pp. 81–94, USENIX.

[53] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, *Order-preserving encryption for numeric data*, in *SIGMOD 2004*, pp. 563–574, 2004.

[54] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, *Cryptdb: Protecting confidentiality with encrypted query processing*, SOSP '11, pp. 85–100, ACM, 2011.

[55] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, *Order-preserving symmetric encryption*, EUROCRYPT '09, pp. 224–241, 2009.

[56] R. A. Popa, F. H. Li, and N. Zeldovich, *An ideal-security protocol for order-preserving encoding*, SP '13, pp. 463–477, IEEE Computer Society, 2013.

[57] W. Zhang, Y. Lin, S. Xiao, J. Wu, and S. Zhou, *Privacy preserving ranked multi-keyword search for multiple data owners in cloud computing*, *IEEE Trans. Computers* **65** (2016), no. 5 1566–1577.

[58] Z. Xia, X. Wang, X. Sun, and Q. Wang, *A secure and dynamic multi-keyword ranked search scheme over encrypted cloud data*, *IEEE Trans. Parallel Distrib. Syst.* **27** (2016), no. 2 340–352.

[59] H. Hu, J. Xu, C. Ren, and B. Choi, *Processing private queries over untrusted data cloud through privacy homomorphism.*, in *ICDE*, pp. 601–612, 2011.

[60] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song, *Shadowcrypt: Encrypted web applications for everyone*, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1028–1039, ACM, 2014.

[61] B. Lau, S. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva, *Mimesis aegis: A mimicry privacy shield–a systems approach to data privacy on public cloud*, in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 33–48, 2014.

[62] M. Naveed, S. Kamara, and C. V. Wright, *Inference attacks on property-preserving encrypted databases*, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 644–655, ACM, 2015.

[63] M.-S. Lacharité, B. Minaud, and K. G. Paterson, *Improved reconstruction attacks on encrypted data using range query leakage*, in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 297–314, IEEE, 2018.

[64] W. K. Wong, D. W. Cheung, B. Kao, and N. Mamoulis, *Secure knn computation on encrypted databases*, in *SIGMOD 2009*, pp. 139–152, 2009.

[65] A. Arampatzis, P. S. Efraimidis, and G. Drosatos, *A query scrambler for search privacy on the internet*, *Information retrieval* **16** (2013), no. 6 657–679.

[66] A. Arampatzis, G. Drosatos, and P. S. Efraimidis, *Versatile query scrambling for private web search*, *Information Retrieval Journal* **18** (2015), no. 4 331–358.

[67] J. Xu and H. Li, *Adarank: a boosting algorithm for information retrieval*, SIGIR '07, pp. 391–398, ACM, 2007.

[68] Q. Wu, C. J. Burges, K. M. Svore, and J. Gao, *Adapting boosting for information retrieval measures*, *Inf. Retr.* **13** (June, 2010) 254–270.

[69] J. S. Culpepper and A. Moffat, *Efficient set intersection for inverted indexing*, *ACM Trans. Inf. Syst.* **29** (Dec., 2010) 1:1–1:25.

[70] S. Ji, J. Shao, D. Agun, and T. Yang, *Privacy-aware ranking with tree ensembles on the cloud*, in *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, pp. 315–324, ACM, 2018.

[71] K. S. Jones, S. Walker, and S. E. Robertson, *A probabilistic model of information retrieval: development and comparative experiments*, *Inf. Process. Manage.* **36** (Nov., 2000) 779–808.

[72] T. Elsayed, N. Asadi, L. Wang, J. J. Lin, and D. Metzler, *UMD and USC/ISI: TREC 2010 web track experiments with ivory*, in *Proceedings of The Nineteenth Text REtrieval Conference, TREC 2010, Gaithersburg, Maryland, USA, November 16-19, 2010*, 2010.

[73] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval (2nd Edition)*. Addison Wesley, 2011.

[74] L. Wang, J. Lin, and D. Metzler, *A cascade ranking model for efficient ranked retrieval*, in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, (New York, NY, USA), pp. 105–114, ACM, 2011.

[75] D. Boneh, *The decision diffie-hellman problem*, *Algorithmic number theory* (1998) 48–63.

[76] Akama.com, *Akamais State of the Internet. Q1 2017 Executive Summary*, 2017.

[77] L. T. I. at Carnegie Mellon University, "The clueweb09 dataset, http://boston.lti.cs.cmu.edu/data/clueweb09.."

[78] B. Carterette, V. Pavlu, H. Fang, and E. Kanoulas, *Million query track 2009 overview.*, in *TREC*, 2009.

[79] L. Boytsov and A. Belova, *Evaluating learning-to-rank methods in the web track adhoc task.*, in *TREC*, 2011.

[80] C. Project, "The enron email dataset, https://www.cs.cmu.edu/ enron/.."

[81] J. Guo, Y. Fan, Q. Ai, and W. B. Croft, *A deep relevance matching model for ad-hoc retrieval*, in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pp. 55–64, ACM, 2016.