University of California
Santa Barbara

Optimizing Efficiency of Privacy-aware Search with Additive and Neural Ranking

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

Doctor of Philosophy
in
Computer Science

by

Jinjin Shao

Committee in charge:

Professor Tao Yang, Chair
Professor Trinabh Gupta
Professor Xifeng Yan

December 2021

The Dissertation of Jinjin Shao is approved.

Professor Trinabh Gupta

Professor Xifeng Yan

Professor Tao Yang, Committee Chair

September 2021

Optimizing Efficiency of Privacy-aware Search
with Additive and Neural Ranking

To my family and friends.

Acknowledgements

In the first place, I owe my sincerest thanks to my Ph.D. advisor, Prof. Tao Yang, for his continuous guidance and support from the very fist day I came to UCSB. Tao has devoted significant time and efforts to advise me and spur me on. I always have my deepest appreciation for his invaluable ideas about my research work, his endless help on improving my presentation & writing skills and definitely those best advice in life.

Secondly, I want to give my huge thanks to my Ph.D. committee members: Prof. Trinabh Gupta and Prof. Xifeng Yan for the enlightenment and suggestions to improve this dissertation. Prof. Trinabh Gupta and Prof. Xifeng Yan have been my committee members since my Major Area Exam. I am deeply grateful to them.

Thirdly, I would like to thank all members in my research group during my Ph.D. study: Shiyu Ji, Yifan Qiao, Yingrui Yang, Xin Jin and Daniel Agun. Especially a huge thanks for Shiyu who was my research partner during my most time in UCSB. Also thanks Alvin Oliver Glova for his help when I begun my study on Intel SGX. What is more, I really want to express my thanks to my friends and roommates throughout my Ph.D. life. Without all of you, it would be much harder for me to reach here.

In the final place, my deepest gratitude goes to my parents, Richeng and Genlian, for their never-changing support. Additionally, a grateful thanks to Jiawen whom I am lucky to have in my life.

This thesis is supported in part by NSF IIS-1528041 and IIS-2040146, and by a Google faculty research award. This thesis used the Extreme Science and Engineering Discovery Environment [1] supported by NSF ACI-1548562. 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

Jinjin Shao

Education

2016 - 2021	University of California, Santa Barbara, USA Ph.D. in Computer Science
2013 - 2016	The Ohio State University, Columbus, USA B.S. in Computer Science and Engineering

Publications

Jinjin Shao, Yifan Qiao, Shiyu Ji, Tao Yang. "Oblivious Top-k Document Retrieval with Dynamic Pruning in a Trusted Execution Environment". In preparation for publication.

Yifan Qiao, Shiyu Ji, Changhai Wang, **Jinjin Shao**, Tao Yang. "Two-level Inverted Indexing for Privacy-aware Document Retrieval". In preparation for publication.

Yingrui Yang, Yifan Qiao, **Jinjin Shao**, Xifeng Yan, Tao Yang. "Lightweight Composite Re-ranking for Efficient Keyword Search with BERT". In preparation for publication.

Jinjin Shao, Yifan Qiao, Shiyu Ji, Tao Yang. "Window Navigation with Adaptive Probing for Executing BlockMax WAND". SIGIR'21, Pages 2323 - 2327, Virtual Event, Canada, July 2021.

Jinjin Shao, Shiyu Ji, Alvin Oliver Glova, Tao Yang, Tim Sherwood. "Inverted Index Obfuscation for Privacy-Preserving Keyword Matching in a Trusted Execution Environment". CIKM'20, Pages 1345 - 1354, Virtual Event, Ireland, October 2020.

Jinjin Shao, Shiyu Ji, Tao Yang. "Privacy-aware Document Ranking with Neural Signals". SIGIR'19, Pages 305 - 314, Paris, France, July 2019.

Shiyu Ji, **Jinjin Shao**, Tao Yang. "Efficient Interaction-based Neural Ranking with Locality Sensitive Hashing". WWW'19, Pages 2858 - 2864, San Francisco, CA, USA, May 2019.

Shiyu Ji, **Jinjin Shao**, Daniel Agun, Tao Yang. "Privacy-aware Ranking with Tree Ensembles on the Cloud". SIGIR'18, Pages 315 - 324, Ann Arbor, MI, USA, July 2018.

Daniel Agun, **Jinjin Shao**, Shiyu Ji, Stefano Tessaro, Tao Yang. "Privacy and Efficiency Tradeoffs for Multiword Top K Search with Linear Additive Rank Scoring". WWW'18, Pages 1725 - 1734, Lyon, France, April 2018.

Abstract

Optimizing Efficiency of Privacy-aware Search
with Additive and Neural Ranking

by

Jinjin Shao

Privacy considerations have become increasingly important for cloud-based information services. There are significant research challenges in top-k document search over outsourced large-scale datasets. It is because letting a cloud server access ranking features and perform advanced scoring computation may unsafely reveal privacy-sensitive information. With a practical restriction towards fast query response time, the heavy-weight cryptographic tools are often too expensive to deploy, and thus a server-hosted search system needs to seek optimized tradeoffs among privacy, efficiency, and relevance.

In this dissertation, a series of efficiency optimized document retrieval solutions is proposed with additive ranking or neural ranking when privacy protection is considered. We firstly introduce an efficiency-enhancing design that obfuscates the access pattern of the inverted index data during query processing in a trusted execution environment (TEE). Then this dissertation presents our work on ORAM-based top-k document retrieval with additive ranking in a TEE, and discusses techniques to accelerate matching with window navigation based index pruning and path caching. Finally, we discuss a privacy-aware neural ranking method with analytic and experimental studies. This dissertation includes evaluation results with TREC datasets on the efficiency and relevance of our proposed schemes against multiple baselines.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With a rising concern of privacy protection [2], one fundamental dilemma in many cloud-based information services (e.g., keyword search) is that a user may not fully trust a server provider while the user is still interested in leveraging the large-scale infrastructures. This distrust comes from the fact that the server hosting online services can have access to private user information and might suffer from cyberattacks which occasionally and accidentally invade the data privacy. In particular, there are significant research challenges in providing efficient privacy-aware search on outsourced large datasets. On one hand, although existing cryptographic tools such as homomorphic encryption [3, 4] can offer secure computation and data confidentiality, it is often infeasible to deploy such heavyweight techniques when the scalability and efficiency are highly desired in real-world applications. On the other hand, given the critical need for neural ranking models that have demonstrated a tremendous advancement [5, 6, 7, 8, 9] but result in significant complexity on both computation and storage, it is challenging to support an efficient scoring process during which a server cannot infer private information.

The previous work in searchable encryption (e.g. [10, 11, 12]) uses software-based solutions on a honest-but-curious server, and seeks a trade-off between privacy and time performance. These approaches have been shown vulnerable against many existing privacy attacks that can exploit the leaked information such as data access patterns (i.e. the way how user access the data during query processing). To mitigate the leakage of access pattern and optimize the query efficiency, Chapter 4 and Chapter 5 in this dissertation leverage the assistance of trusted hardware technology. A TEE provides a secure and isolated space where an application can run protected operations that deal with sensitive data. The OS of a server is blocked to inspect computations and data inside this TEE. Even though a TEE can provide a reasonably-trustable computing environment, a server can still observe the access traffic patterns. Data-dependent memory access pattern leakage can facilitate statistical privacy attacks for document search [11, 12].

This thesis addresses an open problem on how to optimize the efficiency of privacy-aware document search with additive and neural ranking. More specifically, there are three goals in this thesis: 1) A privacy-aware inverted index design is proposed for multi-keyword document retrieval in a TEE using additive ranking while avoiding the leakage of memory access patterns during online query processing. 2) We present a top-k document retrieval scheme with a hardware-assisted ORAM. It enables posting block pruning, with term-specific path caching that optimizes the ORAM access cost for dynamically-selected posting blocks. 3) A privacy-aware document search scheme leveraging neural signals is proposed, with countermeasuring those critical information leakages identified in interaction-based neural ranking models.

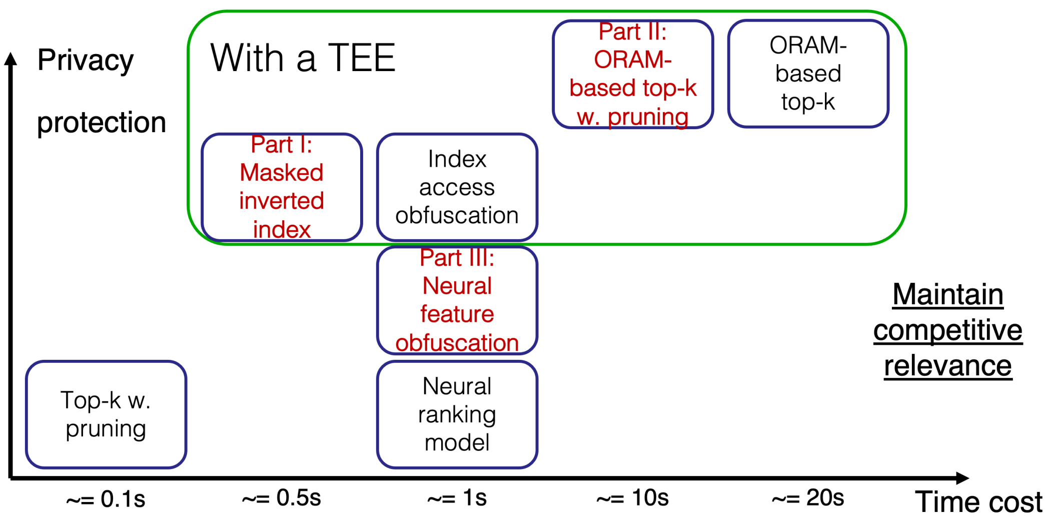


Figure 1.1: Overview of thesis contributions and a comparison with related work

1.2 Contributions and Thesis Organization

This thesis proposes a series of techniques to optimize the efficiency of privacy-aware document retrieval with additive and neural ranking. Fig. 1.1 shows an overview of all contributions here and compares them with related work in terms of privacy and efficiency. More details on contributions in each chapter are explained below.

1. Chapter 3 studies a boosting approach that further accelerates document retrieval by executing BlockMax WAND (BMW), or one of its variants, on a sequence of posting windows with an order prioritized to tighten the threshold bound earlier. This optimization could add benefits to safely eliminate more operations involved in posting block visitation and document score evaluation, which can also be applied

to ORAM-based top-k document retrieval as shown in Chapter 5. This chapter evaluates such index navigation for BMW and two of its variants.

2. Chapter 4 studies privacy-aware inverted index design and document retrieval for multi-keyword document search in a trusted hardware execution environment such as Intel SGX. The previous work uses time-consuming oblivious computing techniques to avoid the leakage of memory access patterns for privacy preservations in such an environment. This chapter proposes an efficiency-enhanced design that obfuscates the inverted index structure with posting bucketing and document ID masking, which aims to hide document-term association and avoid the access pattern leakage. This chapter describes privacy-aware oblivious document retrieval during online query processing based on such an index. Both privacy and efficiency analyses are provided, followed by evaluation results comparing proposed designs with multiple baselines.

3. Chapter 5 proposes a two-phase top-k document retrieval scheme that accomplishes both 1) dynamic pruning for safe document elimination and 2) an ORAM-based search scheme preventing the leakage of inter-term access pattern and intra-term access pattern. The previous oblivious search work does not consider skipping low-scoring documents, while dynamic pruning techniques have been shown to be essential for fast top-k query processing. This chapter describes evaluations with TREC datasets on the efficiency of our two-phase scheme and the impact of proposed techniques.

4. Chapter 6 analyzes the critical leakages in interaction-based neural ranking and studies countermeasures to mitigate such a leakage. The recent work on neural ranking has achieved solid relevance improvement, by exploring similarities between documents and queries using word embeddings. It is an open problem how to

leverage such an advancement for privacy-aware ranking, which is important for top K document search on the cloud. Since neural ranking adds more complexity in score computation, it is difficult to prevent the server from discovering embedding-based semantic features and inferring privacy-sensitive information. This chapter proposes a privacy-aware neural ranking scheme that integrates tree ensembles with kernel value obfuscation and a soft match map based on adaptively-clustered term closures. It also presents an evaluation with two TREC datasets on the relevance of the proposed techniques and the trade-offs for privacy and storage efficiency.

5. Chapter 7 concludes this dissertation and discuss possible future directions.

1.3 Permissions and Attributions

1. The content of chapter 3 is the result of a collaboration with Shiyu Ji, Yifan Qiao, and Tao Yang, and has previously appeared in SIGIR'21. It is reproduced here with the permission of ACM.

2. The content of chapter 4 and appendix A is the result of a collaboration with Shiyu Ji, Yifan Qiao, Alvin Oliver Glova, Tim Sherwood, and Tao Yang, and has previously appeared in CIKM'20. It is reproduced here with the permission of ACM.

3. The content of chapter 6 is the result of a collaboration with Shiyu Ji and Tao Yang, and has previously appeared in SIGIR'19. It is reproduced here with the permission of ACM.

Chapter 2

Problem Definition and Thesis Background

Problem definition. Given a set of query terms that correspond to a query, the goal of the top-k document retrieval problem is to find a set of top-k ranked documents that contain all query terms following the conjunctive query semantics, or at least one of these terms following the disjunctive query semantics. This thesis is focused on the disjunctive query semantic while the proposed schemes are extensible for conjunctive queries. A privacy-aware document search accomplishes document search while preserving privacy as much as possible. We will give more definitions on data structures and algorithms used for document retrieval as follows.

Inverted index. A common data structure used for document retrieval is an *inverted index* which contains a set of terms, and a *document posting list* of each term represents documents that contain such a term. Each term can be a single word used in a document or can represent other ranking features needed for a search system. Each entry in a posting list is a posting record. To support ranking, a posting record of each term contains not only an document ID but also its feature score associated with the term.

Top-k retrieval. Ranking for document retrieval uses a simple additive formula based on term features of each document which is $\sum_{t \in Q} w_t S(t,d)$, where Q is the set of all search terms, $S(t,d)$ is the feature score of term t for document d, and w_t is the weight for the corresponding term. For the simplicity of presentation, we assume $w_t = 1$, namely that the feature score of a term has already been scaled with its corresponding weight. An example of the additive formula is widely used BM25 [13]). Following the same assumptions as the previous work [14], after top-k results are retrieved, these results will be further re-ranked using more advanced ranking methods (e.g. the one we discuss in Chapter 6) in a multi-stage ranking scheme.

The previous top-k search studies have advocated earlier termination strategies for document retrieval to skip low-score documents, which cannot be on top-k, during the traversal of posting lists [15, 16]. The latest well-known scheme, BMW [14] and its variation [17], use the block-based document skipping technique based on the WAND algorithm [15]. In such schemes, the runtime index visit order follows document-at-a-time (DAAT) and postings are sorted by an increasing order by their document IDs. Ordering postings by document IDs also facilitates effective data compression. Another index organization strategy is impact-layered index where postings are divided by layers. Impact scores of documents in a lower layer are higher than that of lower layers, and postings within each layer is sorted by document IDs. For simplicity, this thesis assumes the document-sorted index.

Pruning of posting block visitations or document evaluations happens if the top-k threshold lower bound is higher than the estimated ranking score upper bounds for documents, and the effectiveness of pruning heavily depends on the tightness of the updated threshold lower bound and the accuracy of the estimated ranking score upper bound. BMW and its variants visit the posting lists typically following an increasing order of document IDs.

Privacy-preserving search with additive ranking and ensemble ranking. To support secure document matching functionality without ranking considerations, previous research efforts have studied searchable encryption (e.g. [18, 19, 10, 20, 21]) which are novel cryptographic protocols with provable leakage profile. Secure linear additive ranking based matrix transformation is studied in [22, 23, 24] and such techniques based on matrix transformation is only applicable for small datasets while the solution in [25] still requires a significant amount of post-ranking conducted at the client side with partial server-side ranking, which is not suitable for a large dataset. Privacy-preserving ranking with tree ensembles has been studied in [26].

Ranking requires arithmetic calculations based on feature vectors and homomorphic encryption [3, 4] is one idea offered to secure data while letting the server perform arithmetic calculations without decrypting the underlying data. But such a scheme is still not computationally feasible when many numbers are involved, because each addition or multiplication is extremely slow, meanwhile homomorphic encryption does not support the ability of comparing two results required by ranking. Neural ranking involves more computational complexity than a linear method or tree ensembles, and thus hiding feature computation becomes even harder.

Leakage abuse attacks. The previous works on plaintext or query attacks in [11, 12, 27] assume the adversary knows partial information on term occurrence in addition to a subset of plaintext documents. By preventing the leakage of the term occurrence information of documents in a hosted dataset, threats from these attacks can be removed or greatly alleviated. Islam et al. [11] proposed the query recovery attack called IKK which can be revised to launch a plaintext attack to identify some words in an encrypted document collection. This assumes that the adversary is the server who has some prior knowledge as follows. 1) The server knows plaintext of n_a words that appear in this document collection, but does not know the encrypted word IDs. 2) The server knows co-occurrence probabilities of these n_a words in this document collection. This can be approximated by using a public dataset. Note that, if those co-occurrence probabilities can be exactly computed when the adversary knows all documents, the recovery rate of this attack can be greatly improved. 3) The server has obtained encrypted IDs of documents that contain a subset of known n_a words. With the above three pieces of information, the server is able to recover the encrypted word IDs for a good percentage of these n_a words, and detect the set of document IDs containing these encrypted word IDs. Cash et al. [12] has improved the plain text recoverability of the IKK attack with extra information such as term frequency, and pointed out that the inverted index or occurrence probabilities for a set of words can be inferred when knowing the term frequency of English words in each document. There are also attacks exploiting leaked document similarities [27]. Their attacks only work if the adversary knows occurrence frequency and co-occurrence frequency of selected terms in the entire document set.

Trusted Execution Environment Recently, hardware-based Trusted Execution Environments (TEE), such as Intel Software Guard Extensions (SGX) and ARM TrustZone, have emerged as options for secure computation [28, 29, 30] even if the hosting operating system may still curiously seek for information. The motivation of using a TEE is to have a minimal Trusted Computing Base that only contains the trusted program processing private datasets in a TEE, even the operating system can be excluded from the trust model. The execution of the trusted program is isolated because there is private memory region provided and protected by the hardware mechanism [31]. Thus, any attempt to access the private memory region is denied even it is from privileged-level programs like operation systems.

Chapter 3

Window Navigation with Adaptive Probing for Top-k Retrieval

3.1 Introduction

Document retrieval for top-k search with disjunctive query semantics identifies all documents containing at least one query term, and it often uses a simple additive linear ranking to select top-k results for later-stage re-ranking. This initial stage of query processing is time-consuming in searching over a large document set. Dynamic pruning techniques such as MaxScore [32], WAND [15], and BlockMax WAND (BMW) [14] can greatly speed up the top-k document retrieval process by skipping the evaluation of low-scoring documents that are unable to appear in the final top-k results. As posting lists are often compressed in blocks, BMW [14] divides each posting list into blocks, stores the maximum score per block and leverages such scores to skip unnecessary decompression and inspections of posting blocks. There are various improvements following the work of BMW (e.g. [33, 17, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43]) and the next section discusses some of them.

Since an accurate estimation of the top-k threshold before posting list processing allows these dynamic pruning algorithms to skip more unnecessary operations [44, 38], BMW-t [45] statically divides the index into two tiers: Tier 1 has a sampled index for estimating threshold and Tier 2 has a full index for retrieving all top-k documents. By threshold sampling on Tier 1, and full search on Tier 2, BMW-t is shown to outperform BMW about 10%. The work in [40] uses a machine learning approach called BLR for threshold estimation. A recent comparison study in [41] shows that without using a past-query cache [46, 40], random sampling for threshold estimation can be highly competitive as BLR and obtain about 2% to 13.2% improvement over BMW. This chapter does not use past queries, and also does not consider unsafe threshold estimation, which is a fast approximation technique used in some of the above studies.

The main contribution of this chapter is a complementary scheme to safely accelerate BMW or its variants by changing their way of visiting the index through *window navigation with adaptive probing*. Without changing the internal core structure of BMW or its variants, we consider document retrieval flows through document ID windows and prioritizes a subset of windows that may hold high-scoring documents for a given query. That results in earlier pruning of many windows and the retrieval only navigates through unpruned windows while following a dynamically-selected order. We have evaluated the application of our technique in BMW with uniform block sizes, variable-sized BMW [17], and DBMW with document-ID-oriented blocks [47, 33]. Our evaluation examines the impact of such index navigation and finds that the proposed technique can significantly reduce the query time for short queries with 2 and 3 terms, and a certain range of k values.

3.2 Definitions and Background

BMW [14] assumes that a posting list, sorted in ascending order of document IDs, is compressed and stored in blocks. Some information of a posting block p can be accessible without decompression, and such information contains the maximum weighted term feature score among all documents and the maximum document ID in this block, denoted as $BlockMax(p)$ and $MaxDocID(p)$.

Our work is inspired by the candidate filtering approach [33, 48] which divides the scope of document IDs visited during document retrieval as a sequence of equal-sized windows. A query processing scheme to call a Live Block computation scheme that detects if a window can be pruned. Window pruning is related to interval-based pruning in [49]. These techniques prune a window or an interval when the estimated score upper bound is below the top-k threshold, and a similar technique is used in the local Block-Max method [42]. Following the posting block window setting of [33, 47], to further improve efficiency, LazyBM in [43] combines the WAND-based and MaxScore [32]-based pruning. The major difference of our effort compared to the above work is that we do not execute the document retrieval strictly in a sorted order of document IDs, and we dynamically detect and prioritize windows that may hold high-scoring documents. The work in [49] proposes to use a sorted order of its intervals in terms of interval score upper bound, which has an expensive sorting and execution logic, and outweighs its benefits as verified in [43]. Our work uses a fast selection strategy in a linear complexity to identify a set of windows that hold high scoring documents while we do not explicitly sort them.

Our work is orthogonal to the previous work that improves BMW as we can use a variation of BMW in our algorithm instead of the original BMW. One class of improvement is to estimate top-k threshold in advance before running BMW based on sampling, feature-based prediction, and past queries [44, 38, 40, 46, 40, 41]. The second class is to use a tiered index [45, 50, 51]. The third class is to improve pruning condition [52, 43]. The fourth class is to support different block structures: VBMW [17] uses variable sized blocks to reduce the gap between the $MaxScore$ value and the actual average document term weight in a block, and DBMW [47, 33] uses document-ID-oriented posting blocks and each of them covers a fixed number of consecutive document IDs. We will demonstrate the use of our work for VBMW and DBMW in addition to BMW.

3.3 Index Navigation with Probing

This section presents our proposed window-driven index navigation with adaptive probing to execute BMW, denoted as BMW^p. Besides the original BMW, a variant of BMW can also be applied in each phase of the algorithm described below. All returned documents satisfy the top-k scoring constraint with safe pruning.

Like [33], we divide the whole range of searchable document IDs into a disjoint set of document ID windows, and unlike [33], these windows are not equal-sized. A posting block p intersects with a window W, denoted by $p \in W$, if any document ID in this window is between $MinDocID(p)$ and $MaxDocID(p)$ (both inclusive). Given a set of posting blocks B, the maximum window score of a window W is defined as $WinMax(W) = \sum_{p \in B \wedge p \in W} BlockMax(p)$.

We focus on block-aligned windows. Our evaluation adopts the boundary IDs of each window to be identical to $MaxDocID(p)$ of a block p in this window. Fig. 3.1 shows 6 windows whose boundaries only aligned with the maximum IDs of posting blocks for two terms. The reason is that if we can skip the visitation of a window, the cost of decompressing a block can be avoided. Alternatively, we can opt to have windows aligned with the $MinDocID(p)$ of each block. Unlike [49], we do not use both the minimum and maximum IDs of posting blocks as window boundaries. We intend to let the number of block-aligned windows be as small as possible because such a number affects the probing time cost significantly as shown later. Compared to BMW, BMW^p adds field $MinDoc(p)$ as a small extra space overhead for each posting block.

We define $\Omega(Z)$ as the top-k threshold found by running BlockMax WAND through top Z windows with the highest $WinMax$ values, for a given set of windows. Define $\Gamma(W)$ as the highest rank score of documents that appear in window W.

If we arbitrarily select k windows, at least k distinct documents can be returned, and their minimum rank score can be used as a lower bound of the final top-k threshold. Our goal is to find most promising windows and probe these windows with BMW to yield documents whose rank scores can be close to those of final top-k documents. With N being the total number of windows, we set parameter value Z to limit the number of such promising windows with the largest $WinMax$ values to be probed.

The question is, how to find a good value for Z? A smaller and the earlier derivation of the top-k threshold bound will create more opportunities to prune the unnecessary visitations of posting blocks and their documents. Z needs to be sufficiently larger than k since the highest-scoring document of the top window with the largest $WinMax$ value may not be included in the final top-k documents. It is challenging to find an optimal solution and our idea is to find a reasonable cutoff value Z which satisfies

$$\min_{Z}\{Z \geq \alpha * k \text{ such that } \Omega(Z) \geq WinMax(W^Z)\}$$

where α is a constant and W^Z is the window which has the Z-th largest $WinMax$ value. Let NT^Z denote the set of the remaining $N - Z$ windows which are not in top Z. The above inequality means Z is sufficiently large to discover a tight bound for top-k threshold

θ. Then all windows in NT^Z can be pruned because

$$\Omega(Z) \geq WinMax(W^Z) \geq \max_{p \in NT^Z} WinMax(p) \geq \max_{p \in NT^Z} \Gamma(p).$$

To find the minimum value for Z following the above inequality efficiently, we first use an exponential probing strategy by guessing a series of cutoff point candidates $Z_1, Z_2, ..., Z_t$ where $Z_i = Z_1 * b^{i-1}$ and b is a constant. In our evaluation, we use $b = 8$ and $\alpha = 12.8$.

Description of BMW^p is presented as follows.

- Derive the block-aligned document ID windows based on the posting block metadata of searched terms.
- We let $Z_1 = \alpha k$ as a starting value, $Z_2 = Z_1 * b$, $\cdots$, and $Z_t = Z_1 * b^{t-1}$, until $Z_{t+1} > N$.
- Find the top Z_i window positions in the derived N windows where $1 \leq i \leq t$. That is done recursively as follows:
  - Find top Z_t windows among all N windows using a quick selection algorithm [53].
  - Recursively find top Z_i windows among top Z_{i+1} windows. This procedure ends when we find top Z_1 windows.
- Starting from Z_1, run BMW on posting blocks in selected windows Z_i from $i = 1$ to t until we find the smallest i value such that

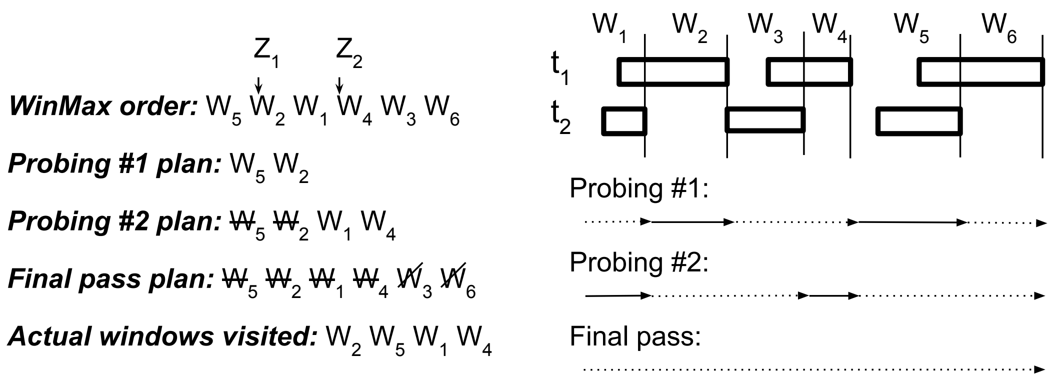
  $$\Omega(Z_i) \geq WinMax(W^{Z_i}).$$

  Let Z_j be the smallest Z_i found.

- In case that the j value derived above is $j = t$, and it does not satisfy $\Omega(Z_j) \geq WinMax(W_{Z_j})$, the final pass is to set $\theta = \Omega(Z_j)$, and prune any un-probed window W in NT^{Z_j} if it satisfies $\theta > WinMax(W)$. Then execute BMW through all the surviving windows, and continue to update the top-k threshold θ. During this process, prune any window W satisfying $\theta > WinMax(W)$.

Notice that posting blocks in windows that are inspected by an earlier round of probing will be annotated with a special mark on their metadata, and will not be re-inspected in a later round when running BMW. Thus during the final pass that has to run BMW through the surviving $N - Z_j$ windows, windows that have been examined earlier will not be re-examined again.

Example. Fig. 3.1 shows retrieval navigation for a two-term query with 6 windows $W_1, \cdots, W_6$ in ascending order of document ID ranges. The descending order of windows in their $WinMax$ values is W_5, W_2, W_1, W_4, W_3 and W_6, even though the algorithm does not explicitly derive this order. The left side of this figure shows that BMW^p plans to probe top Z_1 and Z_2 windows where $Z_1 = 2$, $Z_2 = 4$, $k = 1$, and $\alpha = 2$. The right side shows the actual windows visited during retrieval. The top window locating step identifies W_2 and W_4 are the 2nd and 4th top positions. Round 1 probes top 2 windows W_2 and W_5. Round 2 probes the top 4 windows W_1, W_2, W_4, and W_5 while skipping W_2 and W_5 marked with a strike-through (since they have been probed), and their corresponding BMW execution is depicted by dotted arrows. The details on how to do skipping these windows with a strike-through are discussed in the next paragraph. For the final pass, assuming the derived threshold $\Omega(4) \geq WinMax(W_4)$, the two un-probed windows W_3 and W_6 are pruned immediately and marked with slashes because $\Omega(4) \geq \Gamma(W_3)$ and $\Omega(4) \geq \Gamma(W_6)$. That is inferred by the fact that $\Omega(4) \geq WinMax(W_4)$, and we also know that $WinMax(W_4) \geq \max(WinMax(W_3), WinMax(W_6))$. In summary, index navigation with probing selects and visits a total of 4 windows in the order of W_2, W_5, W_1, and W_4 to safely complete the document retrieval.



Z1 Z2
WinMax order: W5 W2 W1 W4 W3 W6
Probing #1 plan: W5 W2
Probing #2 plan: ~~W5~~ ~~W2~~ W1 W4
Final pass plan: ~~W5~~ ~~W2~~ ~~W1~~ ~~W4~~ ~~W3~~ ~~W6~~
Actual windows visited: W2 W5 W1 W4
W1 W2 W3 W4 W5 W6
t1
t2
Probing #1:
Probing #2:
Final pass:

Figure 3.1: Navigation of windows with 2-round probing

Modified BMW for inspecting only selected windows. We apply BMW as described in [14] with some changes in its two functions *Next(d)* and *NextShallow(d).* in order to only traverse windows selected while directly skip others. Function *Next(d)* returns the first posting record whose document ID is no less than d. Function *NextShallow(d)*, described in Section 5.1 of [14], returns the first posting block whose maximum document ID is no less than d, without loading and decompressing the data of any block. To apply BMW on a specific set of windows, we first mark each posting block in all selected windows by examining the metadata of all posting blocks without any block data decompression. Next, we can apply BMW directly on posting blocks window by window from the beginning in ascending order of document IDs. The modified functions*NextMarked(d)* and *NextMarkedShallow(d)* return the first posting record or posting block, skipping all unmarked posting blocks.

Figure 3.2 shows an example of applying BMW on only marked posting blocks with *NextMarked(d)*. Note that three posting lists here should be sorted based on the document ID of the current iterator (marked by red rectangle box). Therefore, given the current top-k threshold is 3.1, the posting list with term upper bound 1.5 becomes the pivot since the sum of two term upper bounds can pass the threshold, i.e. $2.4 + 1.5 > 3.1$. Then after checking the block upper bound scores for these two blocks, the failure (i.e. $0.8 + 1.5 < 3.1$) results in executing the method *NextMarked(15)* on the posting list with term upper bound 1.5, which is for locating the next document with ID larger than 15. Since this method skips the unmarked block, even the next block in this list has a document whose ID is 16 larger than 15, the next block is skipped and the iterator goes directly to the block after the next block. Finally, the next iterator position reaches to document 18. Therefore the decompression and locating time for the unmarked block are saved through executing *NextMarked(15)*.

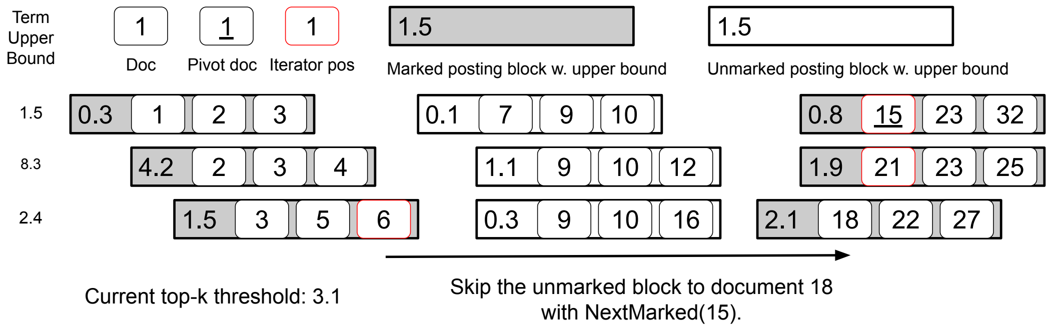

Term Upper Bound
1
Doc
1
Pivot doc
1
Iterator pos
1.5
Marked posting block w. upper bound
1.5
Unmarked posting block w. upper bound
1.5
0.3 1 2 3
0.1 7 9 10
0.8 15 23 32
8.3
4.2 2 3 4
1.1 9 10 12
1.9 21 23 25
2.4
1.5 3 5 6
0.3 9 10 16
2.1 18 22 27
Current top-k threshold: 3.1
Skip the unmarked block to document 18 with NextMarked(15).

Figure 3.2: Execution of BMW on marked posting blocks

Time cost of BMW^p. For simplicity, the cost of index loading from a disk is not included. Notice that many search systems pre-load compressed index into memory [54]. The total time cost for multi-round probing plus the final pass using BMW is $O(N) + O(T^{Z_j}_{BMW}) + O(T^{N-Z_j}_{BMW})$. Here $T^{Z_j}_{BMW}$ is the time cost for running BMW through top Z_j windows with the largest $WinMax$ values and $T^{N-Z_j}_{BMW}$ is the time for running BMW through the remaining windows. The item $O(N)$ is the time cost for driving windows from the posting block metadata of searched terms, and for locating the top scored window positions $Z_1, Z_2, ..., Z_t$. Cost of window generation is linear to the number of windows, which is proportional to the number of posting blocks. Finding the first position Z_t is $O(N)$ with a $O(N)$ quick selection algorithm, and the next position Z_{t-1} costs $\frac{N}{b}$. With the recursion ended at Z_1, the total time to locate these t positions are $O(N+\frac{N}{b}+\frac{N}{b^2}+\cdots)$ which is less than $O(\frac{bN}{b-1})$.

Since the documents in windows are only visited during window-driven index navigation, the total cost of running BMW in BMW^p is bounded by the total time for running an original BMW through all windows in a document ID sorted order while BMW^p should skip more operations related to block visits and document evaluation.

3.4 Evaluations

Datasets. The experiments are performed on the ClueWeb 2009 TREC category B with 33.6 million documents after filtering with Waterloo spam score [55] threshold 60. The header and body text for all documents is extracted using Indri [56]. All words are lowercased and stemmed using the Krovetz stemmer [57]. The inverted index is compressed using SIMD-BP128 [58]. We use TREC 2006 Efficiency Track topics where each query term has more than 100 postings. We randomly select 2500 queries containing two to six terms, where the length distributions are 16.8%, 28.2%, 27.7%, 17.3%, and 10.0%.

Testing environment. The algorithms evaluated are implemented in C++, and compiled with GCC 4.8.5 using the highest optimization settings. For the block partitioning in VBMW, we use the released code from [17]. The experiments are conducted with Intel Xeon E5-2680 v3 2.50GHz. Query processing times are reported in milliseconds using one CPU core on average with multiple runs. Ranking follows the linear additive formula BM25 [13]. Like the previous work [17, 43], all compressed posting lists and metadata for tested queries are loaded into memory before timing the queries.

Baselines. We apply our scheme to BMW with uniform block sizes (64 postings); VBMW [17] with variable-sized blocks (the average size is 64 postings); and DBMW [47, 33] for document-ID-oriented blocks where each block covers 1024 consecutive document IDs. We assume that the block metadata for VBMW is preloaded with space optimization described in [47, 33]. LazyBM [43] belongs to the same sub-family as DBMW with the same posting structure. We have not used LazyBM [43] as its implementation is not in public domain yet, and our current implementation of LazyBM is still slower than DBMW. We will study this more in the future.

Table 3.1: Mean query latency and tail latency in ms

Q. len.	2	3	4	5	6
			$k = 10$		
BMW	14.5 (84)	50.2 (229)	119.5 (413)	220.0 (735)	348.4 (1023)
BMW^p	4.2 (17)	33.1 (119)	102.8 (331)	189.6 (761)	309.7 (1042)
BMW^w	16.1 (57)	88.5 (470)	171.0 (487)	302.3 (923)	473.7 (1453)
BMW^{ws}	15.7 (60)	86.9 (328)	174.2 (545)	274.2 (890)	485.1 (1528)
VBMW	11.1 (80)	38.2 (203)	81.1 (321)	156.7 (479)	282.6 (671)
VBMW^p	1.4 (8)	13.1 (70)	57.3 (240)	105.7 (445)	251.5 (625)
DBMW	12.3 (63)	44.7 (166)	93.6 (241)	158.0 (378)	253.5 (537)
DBMW^p	1.1 (7)	15.9 (89)	45.3 (183)	99.9 (373)	180.7 (526)
WAND	33.7 (186)	74.3 (324)	144.0 (507)	244.5 (806)	484.5 (1157)
IBMW	25.4 (104)	96.6 (520)	210.2 (693)	345.1 (1289)	557.1 (1615)
			$k = 100$		
BMW	19.1 (108)	64.1 (277)	146.8 (460)	271.4 (800)	476.3 (1248)
BMW^p	9.6 (36)	57.1 (199)	153.1 (459)	326.2 (1045)	485.5 (1395)
VBMW	14.7 (96)	56.1 (246)	112.5 (352)	196.2 (518)	315.7 (709)
VBMW^p	7.4 (30)	49.2 (178)	121.9 (348)	236.2 (607)	368.7 (831)
DBMW	16.1 (83)	54.5 (182)	118.8 (289)	198.5 (435)	302.2 (632)
DBMW^p	7.2 (37)	36.7 (163)	90.0 (284)	175.1 (510)	289.6 (726)
			$k = 1000$		
BMW	33.3 (140)	101.5 (300)	224.4 (460)	409.9 (922)	690.0 (1426)
BMW^p	27.7 (78)	127.0 (367)	282.2 (656)	495.8 (1304)	722.0 (1710)
VBMW	32.4 (142)	101.9 (291)	218.9 (468)	350.1 (669)	534.1 (961)
VBMW^p	23.6 (64)	104.8 (283)	255.1 (518)	432.3 (834)	665.2 (1182)
DBMW	33.6 (166)	99.3 (252)	214.4 (439)	368.8 (653)	503.2 (802)
DBMW^p	23.0 (113)	76.4 (220)	178.7 (390)	327.0 (662)	470.7 (850)

Table 3.2: Mean time reduction ratio (-) or increase ratio (+)

	Q. len.	k=10	k=30	k=50	k=100	k=1000
$\frac{T(\text{BMW}^p)}{T(\text{BMW})}-1$	2	-71.3%	-58.2%	-52.6%	-49.6%	-16.6%
	3	-34.1%	-23.1%	-16.6%	-11.0%	+25.2%
	4+	-13.0%	-9.1%	+2.2%	+9.0%	+16.9%
$\frac{T(\text{VBMW}^p)}{T(\text{VBMW})}-1$	2	-87.2%	-69.6%	-63.8%	-49.9%	-27.3%
	3	-65.8%	-43.7%	-27.0%	-12.3%	+2.9%
	4+	-23.8%	-6.5%	+0.6%	+15.3%	+21.4%
$\frac{T(\text{DBMW}^p)}{T(\text{DBMW})}-1$	2	-90.9%	-76.7%	-73.8%	-55.4%	-31.6%
	3	-64.3%	-59.1%	-57.5%	-32.7%	-23.1%
	4+	-39.1%	-37.9%	-32.4%	-13.6%	-11.9%

Table 3.3: Mean time breakdown in ms for BMW^p

Query length	2	3	4	5	6
Num. of windows	41.5K	145.8K	323.7K	509.2K	737.7K
Window generation	1.4	7.6	19.2	32.1	38.1
Window selection	0.2	1.1	3.0	5.2	8.1
Adaptive probing	1.8	16.1	53.0	105.2	186.0
Final pass	0.7	8.3	27.6	47.2	77.5

A comparison with the baselines. Table 3.1 reports the mean query times in milliseconds for different algorithms with query lengths varying from 2 to 6. The window-driven execution of VBMW and DBMW is denoted as VBMW^p and DBMW^p respectively. This table also lists the 95th percentile *tail latency* for each query length within parentheses, corresponding to the response time occurring in the 95th percentile, as defined in [36]. Table 3.2 further shows the reduction ratio of the mean time for different query length and k values. A negative ratio indicates an improvement while a positive ratio indicates a degradation, and the overhead of probing and window navigation outweighs benefits.

BMW^p significantly outperforms BMW for 2 terms with all tested k values and for 3 terms with $k \leq 100$. For the mean time and 2 terms, BMW^p is 3.5x and 1.2x faster than BMW when $k = 10$ and $k = 1000$, respectively. For the tail latency and 2 terms, it is 4.9x and 1.8x faster when $k = 10$ and $k = 1000$, respectively. For 4 or more terms, BMW^p loses its advantages when $k \geq 50$. For long queries, $WinMax$ value is less accurate as a proxy for possible document scores. As k becomes larger, window navigation with probing becomes less effective, especially for long queries. One reason is that the gap of average score between top-k documents and remaining documents becomes smaller when k is large, and thus earlier probing becomes less effective for window pruning.

Like BMW, applying the proposed technique makes VBMW and DBMW significantly faster for queries with 2 terms and $k \leq 1000$ and with 3 terms and $k \leq 100$. When $k = 10$, VBMW becomes 7.8x and 10x faster for the mean and tail latencies, respectively. The reduction ratio when applied to VBMW is in general higher than to BMW because the variable block size design in VBMW makes $BlockMax$ value closer to the score average within a block and hence $WinMax$ value is more accurate as a proxy for possible document scores. DBMW^p also does very well for 2 and 3 terms, and because each equal sized block uses consecutive IDs, the cost of window generation is zero, which reduces optimization overhead.

Table 3.1 also lists WAND [15] and interval-pruning [49] integrated with BMW-based traversal (denoted as IBMW) in Rows 11 and 12 as a reference point. In general, they are slower than the other algorithms, and IBMW costs too much in interval generation.

Impact of adaptive probing. Row 5 of Table 3.1 lists the time of BMW^p without probing, denoted as BMW^w. It simply generates windows and runs the final BMW pass. Even though some windows are pruned, the overhead of generating windows and window pruning outweighs the benefits, and the overall time is even slower than BMW. Another alternative, denoted as BMW^{ws} in Row 6, is to sort all block-aligned windows first and visit them in a descending order of their $WinMax$ values. BMW^p outperforms BMW^{ws} significantly, which illustrates the benefits of adaptive probing. It is too expensive to sort and jump around to access the sorted windows.

Time cost breakdown. Table 3.3 shows the breakdown of query processing time for BMW^p, and the total number of windows handled. The percentage of window selection cost in the total cost is small, varying from 2.6% to 5.5%. If our fast $O(N)$ selection method is not used, sorting based on $WinMax$ would be $O(N \log N)$ cost, where N is the total number of windows, and this translates to about 7x to 10x more cost. Specifically, sorting all windows explicitly would cost 3.0ms, 11.8ms, 27.2ms, 44.1ms, and 65.1ms for query length from 2 to 6, respectively. That would add too much overhead.

The time cost distribution of VBMW^p is similar proportionally as that of BMW^p. For DBMW^p, there is no cost for window generation. As every posting block covers 1024 document IDs uniformly, there are total about fixed 33,000 windows for this ClueWeb dataset.

3.5 Summary

This chapter proposes and evaluates a performance boosting scheme for BMW or its variants by changing their top-k search flow with window navigation and adaptive probing. The evaluation shows this technique can offer a significant speed advantage for short queries with a certain range of k values. For the mean response time, the reduction ratio varies from 16.6% to 90.9% with 2 terms and $k \leq 1000$, and from 11.0% to 65.8% with 3 terms and $k \leq 100$. The reduction trend is similar for the 95th percentile tail latency.

Considering that the average number of words in queries of popular search engines is between 2 and 3 [59, 60], the proposed technique can be very effective for a search engine which deploys many disjoint index partitions and when k does not need to be large for each individual partition that contributes part of top results.

Chapter 4

Index Obfuscation for Oblivious Document Retrieval

4.1 Introduction

The previous work on the above searchable encryption work still cannot reach a fully secured level, because it leaks certain statistical query patterns, which can yield leakage-abuse attacks [11, 12, 61]. This chapter studies the usage of trusted hardware technology that can provide a reasonably-trustable computing environment. However, a server can still observe the access traffic patterns of its TEE to the main memory and to the TEE's internal buffer during server-side query processing. Various memory side-channel attacks with such hardware have been found (e.g. [62]). Data-dependent memory access pattern leakage can facilitate statistical privacy attacks for document search [11, 12]. To protect the leakage of data-dependent memory access patterns for keyword matching of documents, ORAM-based search solutions are proposed in [29, 63], but it is not computationally practical if the number of returned results is large. Another solution called REARGUARD [64] for single-word queries is proposed with oblivious memory accessing, in which a TEE intentionally scans a large number of posting lists to obfuscate access patterns, even if accessing only one of lists is needed. Oblivious computing in this solution also incurs time-consuming query processing cost.

The contribution of this chapter is to propose inverted index obfuscation and a relatively efficient oblivious document retrieval solution with additive ranking by leveraging trusted hardware. Our evaluation results with analyses show that our scheme is much faster than REARGUARD, though it is slower than the traditional top-k document retrieval based on the WAND and BMW optimization without privacy constraints [14, 15]. But these algorithms are vulnerable to privacy attacks, which we present in Section 4.3. To preserve privacy, our algorithm pursues the oblivious but exhaustive strategy, which represents a trade-off of efficiency for privacy.

4.2 Background and Related Work

4.2.1 Threat model

There are three entities in a cloud system: data owner, search user (client) and cloud server. The data owner has a collection of documents to be outsourced by the cloud server. To enable the searching and ranking functionality, the data owner needs to encrypt the documents, the inverted index, and the ranking model for the cloud server outsourcing. This proposal will initially assume that the data owner and the search user are the same entity. The client builds an encrypted but searchable index and lets a server host such index. The server is honest-but-curious, i.e., the server will honestly follow the client's protocol, but will also try to learn any private information from the client data based on what the server can observe from the hosted data and runtime information. To conduct a search query, the client sends several encrypted keywords and related information to the server. Our research targets at inexpensive client-server communication since extensive multi-round communication between the server and client (e.g. [65, 66]) incurs excessive high communication cost and response latency.

4.2.2 Known privacy attacks in document search

As the queries and documents are encrypted, the privacy abuse attacks can occur by acquiring statistical information to reveal search queries or documents partially. A server can always learn something by observing query processing. The question is if the learned information is sufficient for a server to recover original text words from encrypted index and content. The statistical patterns leaked during query processing can lead to the known attacks [11, 12, 61, 27] and we summarize them as follows.

- *Co-occurrence probability of search terms in a document* . Islam et al. [11] proposed the IKK plaintext attack to recover the plaintext of query words by computing the search word occurrence probability when knowing the result overlapping patterns of two single-word queries. The paper assumes that the adversary also knows the co-occurrence probability of dictionary words used in the dataset (e.g. approximated from a public dataset) and a plaintext mapping from a small set of words to their IDs.

- *Document frequency of search terms.* This information can be derived after knowing the length of a posting list for a term. The work in [12] shows that after knowing co-occurrence of search words in a document and their document frequency, an adversary can recover the plaintext of search terms without knowing the mapping from a small subset of search words to their word IDs.

- *Query equality pattern.* When a server knows about the repetition ratio of search queries, it can compare with some known background knowledge on query popularity, and deduce plaintext for some of queries as shown in [61].

- *Document similarities.* There are also attacks exploiting leaked similarities among documents [27]. Their attacks only work if the adversary knows occurrence frequency and co-occurrence frequency of selected terms in the entire document set.

All of these attacks require the leakage of term co-occurrence and term frequency, and knowing document sharing patterns among posting lists, the posting list length, and the list access frequency can lead to the above statistical information leakage. We will consider countermeasures in our design. We will also study how a traditional document retrieval algorithm without a privacy protection can reveal the above statistical patterns.

Searchable encryption (e.g. [18, 19, 10, 20, 67]) has been proposed to identify documents that match a user query from the encrypted index. These schemes do not support ranking and they still suffer from some degree of information leakage, which could cause a leakage-abuse attack discussed above. For example, OXT [10, 20] and IEX scheme [67] are the most comprehensive schemes to support multi-keyword queries. IEX for conjunctive queries leaks the overlapping of search terms from one query to another, thus can leak the co-occurrence probability of search terms and posting list access patterns. OXT for conjunctive queries leaks the overlapping of some search terms (e.g. leading terms) in a query history.

As discussed in the background, the previous work in [22, 23, 24] for secure additive ranking based matrix transformation is only applicable for small datasets, while the solution in [25] does not support full server-side additive ranking. None of the previous work including nonlinear ranking solutions in [26, 68] has solved the problem of privacy protection in document retrieval with additive ranking in dealing with a large dataset. Also, these schemes use deterministic IDs for search terms, thus leak the query equality patterns.

4.2.3 Trusted hardware platform

We assume there is a server platform that incorporates a processor where applications can create and make use of protected memory regions in their trusted execution environment. For example, data center providers typically use SGX-equipped Intel processors. Microsoft currently provides Intel SGX extensions in its cloud platform. A TEE has its own memory space and a host processor can monitor all accesses to the TEE's memory space, knowing which addresses are visited by the code running inside the TEE, but it cannot view the content accessed by the TEE's code.

To thwart attacks based on the leakage of posting list access patterns, [64] designs REARGUARD, an oblivious matching algorithm for one-word queries as follows. First it pads all posting lists with encrypted useless records so that all lists have the same lengths. Second, with one query word, the server scans all posting lists one by one from the whole inverted index. When the TEE has scanned all posting lists, only matched one is retained and the server cannot differentiate which one is matched, corresponding to the search terms. To alleviate the huge time cost to scale all posting lists, [64] suggests to partition indexed terms into groups and a server only scans a group of posting lists instead of the entire index. This trade-off brings a leakage on keyword group access patterns. To be effective for privacy protection, the group size still needs to be large, thus the time cost is still expensive.

4.3 Leakage-abuse Attacks and Design Considerations

We discuss possible leakage-abuse attacks during document retrieval with or without WAND-based document skipping [15], and present our design considerations in leveraging trusted hardware platforms. The main design challenge is that naively running document retrieval inside a TEE, even the index is encrypted, can leak unique query-dependent memory access traces, and an adversary server can take advantages and launch a leakage-abuse attack. We describe two attacks below: the first one reveals co-occurrence information and the second one reveals posting list access patterns. For the worst case, both attacks can recover plaintext queries from encrypted queries ([12] and [61] resp.).

Assumptions on adversary's knowledge. We assume the binary code of a program running inside TEE is known by a server adversary, and the server can observe all the memory accesses (namely, memory addresses that are read or written). Since some server can guess which instruction in TEE's code segment is read and executed, for the *worst case scenario*, we assume there exists an adversary who can trace program running step by step given a previously-issued client query. Hence an adversary can learn at least the number of search terms, and the number of the corresponding posting lists loaded to the TEE. With special obfuscation, the server can also learn the length of each posting list (the number of documents) approximately after decryption and decompression within the TEE.

We assume the run-time behavior of a document retrieval program is deterministic, only dependent on the input query terms. Also assume in a worst case, the binary code can be loaded into the TEE with the same starting memory address.

Attack revealing co-occurrence of search terms. We consider a standard document-at-a-time algorithm without using WAND, and it visits documents one by one in all posting lists in an increasing order of document IDs. The TEE program should have a code segment that adds the relevance score of matched postings from each posting list. The adversary can trace code execution and memory accesses within such a segment, and infer the intersection pattern of any two of the posting lists. Then the adversary can compute the co-occurrence probability of two search terms in a document as the number of common documents in two term lists divided by the total number of documents.

By now, the adversary knows the co-occurrence probability of encrypted search terms, and the term frequency based on the posting list length. The adversary can assume the dataset uses words from a dictionary (e.g. English), obtain the term co-occurrence probability based on the public knowledge, and recover the original plaintext of these encrypted search terms, following the work of [11, 12].

Attack revealing query equality patterns. For any document retrieval algorithm, there are two ways for a server to find the repeat probability of a query. One is to recognize the same set of search terms that has been used from one query to another. The other way is to observe and consider the memory address access sequence as a signature and detect if there is a repetition. With high probability, different queries will yield different memory access sequences. Then a server can use the derived repetition patterns to launch a query recovery attack following [61].

To minimize the chance of launching the above attacks, our countermeasure considerations are listed as follows.

- To hide document co-occurrence between posting lists, we can merge multiple terms into one large posting list. Hence the intersection size between two merged lists does not reflect any true co-occurrence information between two individual terms. Such merging essentially combines several terms together as a term bucket, which also hides the relationship of one-to-one mapping between index and query terms.

- To avoid leaking the signature of the memory address sequence, we pad the search results. For WAND/BMW based algorithms [15, 14, 17], it leaks a fixed memory address accessing sequence as a signature and one can infer the query repetition patterns. Initially, we considered using randomization of document skipping to yield different memory access patterns on the same input. However, given one query from the client, the server adversary can infer the statistical distribution of the memory access patterns by repeating such a query until the distribution can be inferred with a high confidence. Namely, the server can still approximate query repetition patterns and detect the query repetition.

As a result, we decide not to use WAND-based document skipping as a trade-off of efficiency for privacy. That essentially follows an exhaustive search, which has a visible efficiency loss. We view this efficiency loss acceptable with today's CPU speed and will evaluate the response time in our experiments.

We will reflect the above countermeasures in our algorithm design.

4.4 Index Bucketing and Masking

We propose *masked inverted index (MII)* with term bucketing to support efficient query processing, while avoiding the leakage of query equality pattern, co-occurrence probability pattern, and term frequency pattern. Our key ideas are: 1) To obfuscate the index, we group terms into buckets. Each bucket is linked to the union of posting lists of terms grouped in this bucket. The TEE of a server can conduct the term-bucket based posting list retrieval, and identify matched documents in an oblivious way to prevent the above-mentioned leakage; for example, the server cannot differentiate the identities of query terms. 2) To increase the degree of obfuscation, we duplicate each term and its posting list by k times, and randomly map duplicated copies to different term buckets. As a result, each term bucket contains mixed popular or unpopular terms, and the access pattern of each term bucket would be drastically different from that of original individual terms. Then, it is unlikely that a server can derive reasonably accurate query equality patterns and document frequency patterns. We let a TEE decrypt a posting record inside its trusted memory buffers to conduct protected query processing. Since the server is unable to observe the buffer content, it cannot detect co-occurrence probability patterns among term posting lists.

Table 4.1 lists frequently used symbols through this chapter.

Table 4.1: Frequently used notations

Symbols	Explanations
K	Number of top ranked results needed for document retrieval
V	Vocabulary size, namely, number of searchable terms
q	Number of search terms in a query
k	Number of duplicate copies for each term
b	Number of term copies hosted in each bucket
B	Number of term buckets
m	Mask code for indicating term association in a bucket
u_i	Term bucket ID for the i-th query term
$m(u_i)$	Selector for the i-th term hosted in Bucket u_i
Enc	Symmetric encryption with a random seed, e.g., AES256
reg_i	The i-th register

4.4.1 Inverted index with term buckets

We organize the index as follows.

- The index contains a set of term buckets. These buckets along with their posting lists will be compressed first, and then encrypted. Each a term bucket can represent b terms: $t_1, t_2, \cdots, t_b$. A term bucket data structure is accessed by a bucket ID and its value is a sequence of posting records. Each posting record is denoted as (d, m, f) where d is the document ID, m is a masking bit code with h bits indicates which of the corresponding h terms own this posting, and f is a sequence of features for these terms in d. Document d belongs to the posting list of term t_i if and only if the rightmost i-bit of masking code m is 1. The f component only stores the document features for terms t_i whose term posting list has d. Postings in this term bucket are sorted in an increasing order of document IDs.

- Given V terms in an inverted index, we intend to duplicate each term k times and randomly map these kV terms to a set of buckets so that each bucket has b terms. Our goal in forming these buckets is to have a random distribution of these term copies among these buckets. As a result, such randomness enhances privacy protection. After a random mapping, there exist some buckets where multiple copies of the same term appear in the same bucket, and we call this mapping collisions. A large number of collisions reduces the effectiveness of randomness for privacy. In the next subsection, we describe a collision tolerance condition, and present a bucket building algorithm that reduces collisions until the tolerance condition is satisfied.

- The index building process will derive a mapping as shown below from each searchable term t to the bucket location of its k copies: $u_1, u_2, \cdots, u_k$.

  $$map(t) = \{(u_1, m(u_1)), (u_2, m(u_2)), \cdots, (u_k, m(u_k))\},$$

  where for i from 1 to k, the $m(u_i)$-th term of bucket u_i represents term t. A client-side machine maintains such a map.

Figure 4.1 illustrates a masked inverted index in Part (d) derived from an original inverted index in Part (a). In Fig. 4.1(a), the posting list of term A contains documents d_1 and d_4. In Fig. 4.1(b), each term is duplicated twice. For example, A has two copies $A1$ and $A2$. The copies of these terms are mapped to 3 buckets. Bucket 1 contains $A1$ and $B1$. Fig. 4.1(c) is the client-side map based on the bucket layout from Fig. 4.1(b). "$A \rightarrow (1,1)(3,2)$" means that term A has one copy at Bucket 1 as the first term with a term selector 1, and another copy at Bucket 3 as the second term with a selector 2. Fig. 4.1(d) is the server-side collection of term buckets. Term bucket 1 has a posting list of four documents, where each document is associated with a mask code. For example, d_4

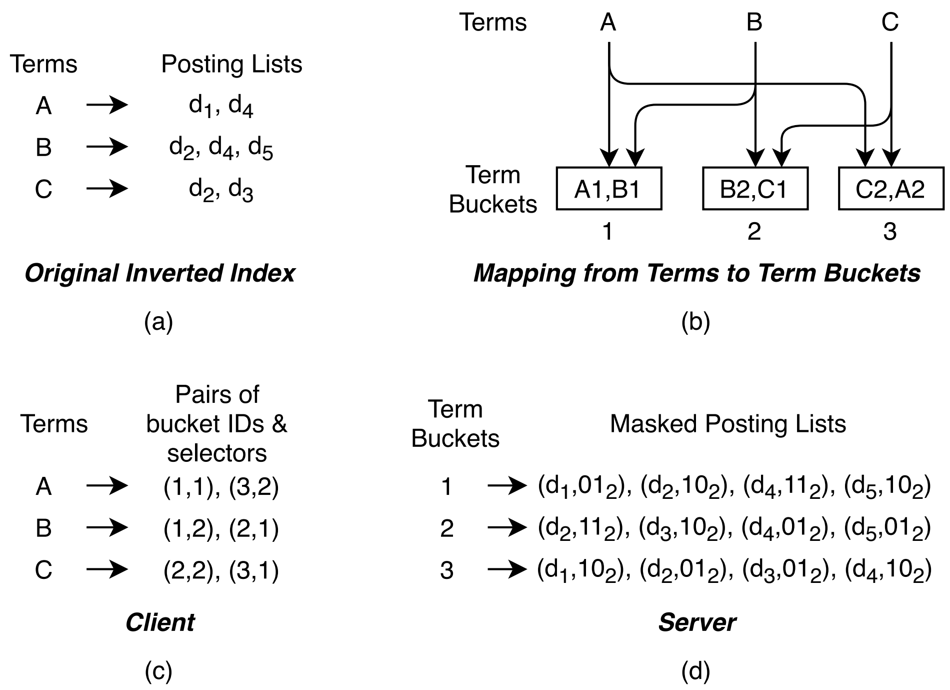


Figure 4.1: Masked inverted index with term buckets

has binary mask 11_2, and is retrievable with a selector 1 or 2, while d_5 is only retrievable with a selector 2. For simplicity, Figure 4.1 does not include the feature sequence in each posting record, which is discussed later in this section.

4.4.2 Term bucketing with limited collisions

Ideally speaking, a term t is duplicated to k copies and they are mapped to k distinct term buckets, and each term bucket has b distinct terms. As a result, search term t can be obfuscated by the other $(k \cdot b - 1)$ unique terms. That was the goal of our design. However, a randomized mapping process most likely fails to achieve the above optimal situation due to mapping collisions, thus as a compromise, we design a near-optimal solution.

Our term bucket building algorithm performs as follows. Let S be a sequence of combining the k duplicates of all terms $S = \{(i, j) : 1 \leq i \leq V, 1 \leq j \leq k\}$ where tuple (i, j) represents the j-th copy of the i-th term. We add fake terms if necessary to make the total number of term copies divisible by b and pad these fake documents at the end of S. We randomly shuffle the tuples in S, then group b consecutive tuples.

We define a collision pair as two copies of the same term are mapped to the same bucket. If the collision tolerance condition, defined below, is not satisfied in the current mapping, we will restart another random shuffle of tuples in S, and regroup them again. We repeat this random mapping until the collision condition is satisfied.

Collision tolerance condition: After executing a randomized mapping from term copies to buckets, let each term t be duplicated to the following k buckets: $u_1, u_2, \cdots$, and u_k. These buckets satisfy both of the following constraints: 1) The set of $\{u_1, u_2, \cdots, u_k\}$ has at least $(k-1)$ unique bucket IDs. 2) For each term bucket in $\{u_1, u_2, \cdots, u_k\}$, it has at least $(b-1)$ unique term copies from different terms, among all b term copies.

Let z be the number of times required to conduct remapping until the above collision tolerance condition is met. From Theorem 4.5.3 in the appendix, the probability of having z iterations is $(\frac{kb^2}{V}(\frac{1}{3} + \frac{k+b}{8}))^z$. For the datasets we tested, V is over 0.5 million, and kb^2 is small, so the probability of $z \geq 3$ is very small. Therefore, the random re-mapping process stops after a few iterations.

4.4.3 Oblivious online document retrieval

Figure 4.2 illustrates the client-server interaction with the presence of a TEE in a host server. Algorithm 1 is a description of our oblivious document retrieval scheme with additive rank score calculations for a disjunctive query. Given a query which corresponds to a number of search terms, a client first performs a map lookup for each term t as follows. Given $map(t) = \{(u_1, m(u_1)), (u_2, m(u_2)), \cdots, (u_k, m(u_k))\}$, the client randomly selects

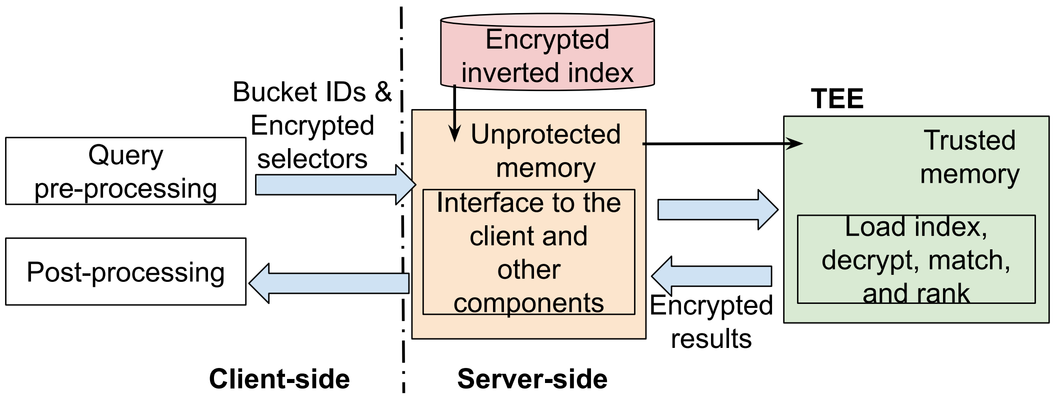


Figure 4.2: Online document retrieval with TEE

one of these k tuples, say, $(u_i, m(u_i))$, and then sends its encrypted form to the server.

Once the server receives all term bucket IDs and encrypted term selectors, its TEE loads encrypted posting lists for those term buckets to the trusted buffer space of the TEE, decrypts and decompresses them. For example, given $(u_i, Enc(m(u_i)))$, the TEE loads encrypted list for u_i, and decompresses the list after decryption. Then the TEE accesses each posting record of bucket u_i, say, (d, m, f). Assume $m(u_i)$ is the current term selector, the TEE considers d as a candidate when $m\&(1 << (m(u_i) - 1)) \neq 0$, where "&" is with a bit-wise AND. Then Algorithm 1 extracts a feature score from f obliviously using Algorithm 2.

For the example in Fig. 4.1, with a query including a keyword "A", a client can randomly choose and send term bucket ID 3 and a term selector 2 to the server. On the server side, a TEE loads, decrypts, and decompresses the posting list of bucket 3. For document d_1 with binary mask 10_2, it matches the term selector 2, and thus d_1 could be a candidate for the final result list.

Oblivious feature extraction. As discussed in Section 4.4.1, the third component, f, of each posting record (d, m, f) contains a sequence of rank score feature for document d corresponding to different terms specified by bit mask m of size b. To prove the oblivious

Algorithm 1: Oblivious top K retrieval for MII in a TEE

```
Input: An encrypted query that contains $q$ terms. The $i$-th query term is
       represented by a bucket ID and an encrypted term selector: $(u_i, Enc(m_i))$
For each $i$-th term where $1 \leq i \leq q$, use $u_i$ to locate and load the encrypted and
  compressed bucket posting list and decrypt term selector to get $m_i$;
Let $\mathbf{L}$ be the list of posting lists of term buckets after decryption and
  decompression such that $\mathbf{L}[i]$ is the posting list of the term bucket corresponding
  to the $i$-th query term;
while there are documents left in $\mathbf{L}$ do
    Let $d$ be the next document with minimal id in $\mathbf{L}$, and advance the pointer of
      $d$ in each list of $\mathbf{L}$ where $d$ appears;
    Let $D$ be the set of indices $i$'s such that for each $i$, $d$ appears in the posting list
      $\mathbf{L}[i]$;
    The relevance score for $d$ in $\mathbf{L}[i]$ where $i \in D$, i.e., $s_i$, can be obliviously
      obtained through Algorithm 2;
    The ranking score for $d$ is $\sum_{i \in D} s_i$;
end
Use oblivious sorting [69] to find the top $K$ ranked documents;
Return the encrypted top $K$ document IDs and rank information;
```

property of Algorithm 1 in Lemma 4.6.1, the feature score fetch procedure is required to be oblivious with respect to different term selectors. Given the number of bits in mask m is b, a naive oblivious method is to store all b features for each bucket posting record for which a special value is used for an invalid feature since some of bits in m can be zero. However, this approach requires higher storage cost given there are many 0 bits in term masks.

We devise a space-conscious oblivious method, without storing invalid features. We let f store a sequence of valid features following the non-zero pattern in the bit mask m, and design a bit manipulation procedure in Algorithm 2 which locates the position of the corresponding feature in f given a term selector. The complexity of Algorithm 2 is linear to the number of bits in m which is b. Given any document in any bucket, Algorithm 2 is oblivious to any term selector for this term bucket. Namely, Algorithm 2 gives an identical access pattern for different term selectors from 1 to b.

Example of Algorithm 2. Assume a posting record is (d, m, f) where mask $m = 0101_2$. Assume the input term selector is 3, which means to select the third bit starting the rightmost. The feature list f is $[f_1, f_3]$. Algorithm 2 first finds the feature offset $ind = 2$, since there are 2 ones among the first 3 bits from the right side. Then the extracted feature is $s = 1 \cdot f_3 \cdot 1 + 0 \cdot f_1 \cdot 1 = f_3$. If the term selector is 2, then $ind = 1$, and $s = 0 \cdot f_1 \cdot 0 + 1 \cdot f_3 \cdot 0 = 0$ implying no feature is selected.

Algorithm 2: Oblivious Feature Extraction for MII

```
Input: A term selector $m(u)$, ranging from 1 to $b$ (both inclusively). A mask
       code $m$ with $b$ bits, and a list of all valid features $f$ for the current bucket
       posting record.
$ind \leftarrow 0$ ;                    ▷ Count # bit 1 before the $m(u)$-th bit in $m$
for $c$ from 1 to $b$ do
    Let $maskBit$ be the $c$-th bit of $m$;
    $reg_1 \leftarrow c - m(u) - 1$;
    Let $signBit$ be the sign bit of $reg_1$, namely, 1 iff $reg_1 < 0$;
    $ind \leftarrow ind + (maskBit \& signBit)$, where & is bit-wise AND;
end
Let $test$ be the $m(u)$-th bit of $m$;
The selected feature score can be obliviously computed as
 $s \leftarrow \sum_{j=1}^{f.size} \texttt{Equal}(ind, j) \cdot f[j] \cdot test$;
Return extracted feature score $s$;

Function Equal($a$, $b$):
    Store $a$ to a register $reg_1$, and $b$ to a register $reg_2$;
    $reg_3 \leftarrow reg_1 \oplus reg_2$, where $\oplus$ is bit-wise logical XOR;
    Store logical NOR of the bits in $reg_3$ to $reg_4$;
    return $reg_4$;
```

4.5 Mapping from Term Copies to Buckets

We investigate the probability of restarting a random mapping from kV term copies to the kV/b buckets defined in Section 4.4.2. This mapping groups b consecutive term copies in a randomly shuffled sequence of kV term copies: $\{(i, j) : 1 \leq i \leq V, 1 \leq j \leq k\}$

where tuple (i, j) represents the j-th copy of the i-th term. Recall that a collision pair is defined as two copies of the same term mapped to the same bucket.

We also call **(k, b)-allocation** as a parameter pair with bucket size b and k duplicate degree of each term. We define **random variable** $Y_i^{(k,b)}$ as the number of collision pairs among the b term copies sent to the i-th bucket for this (k, b)-allocation.

Lemma 4.5.1 *For (k, b)-allocation,*

$$\Pr[Y_i^{(k,b)} \leq 1] = \frac{\binom{V}{b}k^b + \binom{V}{b-1}(b-1)\binom{k}{2}k^{b-2}}{\binom{kV}{b}}.$$

Proof: The total number of all possible combinations of the term copies sent to the i-th bucket is $\binom{kV}{b}$. The number of combinations that there is no collision among the b term copies sent to the i-th bucket is $\binom{V}{b}k^b$. Hence $\Pr[Y_i^{(k,b)} = 0] = \binom{V}{b}k^b / \binom{kV}{b}$. The number of combinations that there is exactly one collision among the b term copies sent to the i-th bucket is $\binom{V}{b-1}(b-1)\binom{k}{2}k^{b-2}$, which can be shown using a counting argument:

1. choose $b - 1$ terms;
2. choose one term among $b - 1$ for where the collision happens;
3. the collision can happen between any two of k copies of the chosen term;
4. each of the other $b - 2$ terms has k copies to choose from.

Hence $\Pr[Y_i^{(k,b)} = 1] = \binom{V}{b-1}(b-1)\binom{k}{2}k^{b-2}/\binom{kV}{b}$. ■

Lemma 4.5.2 *For (k, b)-allocation, if $\frac{b(b+1)}{2} - 1 << V$, then*

$$\Pr[Y_i^{(k,b)} > 1] \leq \frac{\binom{b}{3} + 3\binom{b}{4}}{V^2}.$$

Proof: By Lemma 4.5.1 we have

$$\Pr[Y_i^{(k,b)} \leq 1] = \frac{V \cdots (V-b+1) + V \cdots (V-b+2) b(b-1)\left(\frac{1}{2} - \frac{1}{2k}\right)}{V \cdots \left(V - \frac{b-1}{k}\right)}.$$

Since $\frac{b(b+1)}{2} - 1 << V$, we have $V - b + 1 >> \frac{b(b-1)}{2}$. Hence $\frac{b(b-1)}{2(V-b+1)} = o(1)$, where $o(1)$ denotes sufficiently small positive number given sufficiently large V. Then

$$\Pr[Y_i^{(k,b)} \leq 1] = \frac{V \cdots (V-b+1)(1+o(1))}{V \cdots \left(V - \frac{b-1}{k}\right)},$$

which asymptotically decreases as k increases. Hence the probability $\Pr[Y_i^{(k,b)} > 1]$ is upper bounded by $\Pr[Y_i^{(\infty,b)} > 1]$ for (∞, b)-allocation, which has infinite buckets, and each bucket chooses b terms independently uniformly. For (∞, b)-allocation, $\Pr[Y_i^{(\infty,b)} > 1]$ can be upper bounded by the following argument:

1. the probability that there are two pairs of copies, each of which comes from the same term, is $\binom{b}{4} \cdot 3V^{b-2}/V^b$, which can be shown using a counting argument: a) choose 4 copies for two collision pairs; b) there are 3 ways to arrange the chosen 4 copies for two pairs (see Figure 4.3); c) for each collision pair and each copy that is not chosen in a), its term has V possibilities;

2. the probability that there are three copies from the same term is $\binom{b}{3} V^{b-2}/V^b$, which can be shown by a counting argument similar to 1);

3. $Y_i^{(\infty,b)} > 1$ if and only if (1) or (2) happens.

Note that the events of (1) and (2) are overlapped, e.g., the case when more than 3 copies in one bucket come from the same term is in both (1) and (2). Hence we only claim an upper bound using union bound. ■

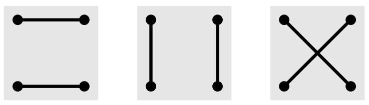


Figure 4.3: Three ways to arrange two collision pairs among four chosen term copies

Theorem 4.5.3 *If $\frac{b(b+1)}{2} - 1 << V$, the probability to start over the random mapping algorithm is at most*

$$\frac{b^2k}{V}\left(\frac{1}{3} + \frac{b+k}{8}\right).$$

Proof: Let X_i denote the number of collision pairs among the copies from the i-th term, and let Y_j denote the number of collision pairs among the copies to the j-th bucket. Then by Lemma 4.5.2

$$\Pr[X_i > 1] \leq \frac{\binom{k}{3} + 3\binom{k}{4}}{B^2} \leq \frac{1}{B^2}\left(\frac{k^3}{6} + \frac{k^4}{8}\right),$$

$$\Pr[Y_i > 1] \leq \frac{\binom{b}{3} + 3\binom{b}{4}}{V^2} \leq \frac{1}{V^2}\left(\frac{b^3}{6} + \frac{b^4}{8}\right).$$

By union bound and the fact $B = kV/b$,

$$\Pr[\max_{1\leq i\leq V} X_i > 1] \leq \sum_{i=1}^{V} \Pr[X_i > 1] = \frac{b^2}{V}\left(\frac{k}{6} + \frac{k^2}{8}\right),$$

$$\Pr[\max_{1\leq i\leq B} Y_i > 1] \leq \sum_{i=1}^{B} \Pr[Y_i > 1] = \frac{k}{V}\left(\frac{b^2}{6} + \frac{b^3}{8}\right).$$

Based on the collision tolerance condition in Section 4.4.2, the random term-bucket remapping restarts if one or both of the following conditions hold:

- 1) Each term has at least $(k-1)$ unique bucket locations if and only if the number of collision pairs among the copies of each term is more than 1, namely, $\max_{1\leq i\leq V} X_i > 1$.
- 2) Similarly, each bucket has at least $(b-1)$ unique terms if and only if $\max_{1\leq i\leq B} Y_i > 1$.

Thus the probability to start over re-mapping is at most

$$\Pr[\max_{1\le i\le V} X_i > 1 \vee \max_{1\le i\le B} Y_i > 1] \le \frac{1}{V}\left(\frac{b^2k}{3} + \frac{b^2k^2 + b^3k}{8}\right).$$

■

4.6 Privacy Analysis

4.6.1 Obliviousness of MII

Definition 4.6.1 Memory access pattern and obliviousness *[70, 71]. Memory access pattern is the sequence of memory accesses during the lifetime of algorithm execution. If the accessed memory is within TEE, each memory access contains two pieces of information: 1) access type (namely, read or write), and 2) memory address. TEE hides its memory content from the server. A document retrieval scheme is oblivious over an input query set if for any two queries from this query set, the memory access patterns are identical (for a deterministic algorithm) or identically distributed (for a randomized algorithm).*

Note that our main algorithm (Algorithm 1) is deterministic. Hence we need to show that our algorithm gives identical memory access patterns over some queries. If one works with a randomized algorithm, then to be oblivious, this algorithm needs to preserve the distribution of memory access patterns.

Lemma 4.6.1 *Replacing any queried term with any term in the same bucket cannot change the memory access pattern of Algorithm 1.*

Proof: Replacing any queried term with any term in the same bucket does not change **L** in Algorithm 1. The proof follows the observations as below:

1. Iterating over sorted bucket lists $\mathbf{L}$ is deterministic. Since we do not change $\mathbf{L}$, the memory access pattern of the iterating is not changed either. Computing feature aggregation $\sum_{i \in D} s_i$ is deterministic for each document given the same bucket lists $\mathbf{L}$.

2. The for loop in Algorithm 2 to compute the feature offset is oblivious to any input, since the memory access sequence is always a linear scan on the b bits of the bitmap.

3. Following the for loop, the feature aggregation in Algorithm 2 is oblivious if only selector is changed in the input data. Note that in Algorithm 1, as long as $\mathbf{L}$ is not changed, the sequence of documents that are passed to Algorithm 2 is not changed either. For each call of Algorithm 2, only selectors can be changed for different queries, while masks and feature lists keep the same. Hence the memory access pattern of feature aggregation in Algorithm 2 is not changed.

4. `Equal` has the same memory access pattern for any inputs a and b, namely, the sequence $(r1, \downarrow, r2, \downarrow, \downarrow, \downarrow, ret)$, where $r1$, $r2$ denote reading the memory locations of two arguments for `Equal` respectively, $\downarrow$ denotes reading the next instruction of the algorithm, and ret denotes returning to the caller of `Equal`.

5. Decryption, decompression, oblivious sorting and encryption give the same memory access pattern for any input lists with identical length.

■

Theorem 4.6.2 *For any query with q terms, there exist at least $b^q - 1$ other queries, where b denotes the number of terms in each bucket, such that Algorithm 1 is oblivious over all these b^q queries.*

Proof: By Lemma 4.6.1, for any query with q terms, each of the q terms can be replaced to b terms in the same bucket while preserving memory access patterns.

Hence the entire query can change to at least $b^q - 1$ other alternative queries preserving obliviousness of Algorithm 1. ■

4.6.2 Obfuscations of terms and queries

Theorem 4.6.3 *Assume each of V terms is duplicated to k buckets, and each bucket merges b terms. Then on average each query with q different terms is obfuscated over at least $\left((b-1)(k-1)(1-\frac{bk(k-1)}{2kV-2})\right)^q$ different queries, which are possible to match the same buckets during document retrieval.*

Proof: We first derive the lower bound on the number of terms which any query term is obfuscated over. Following the mapping process of Section 4.4.2, let $w_{i,j}$ denote the j-th term merged by the i-th matched bucket ($1 \le i \le k$ and $1 \le j \le b$). Hence

$$\Pr[w_{i,j} = w_{i',j'} | i > i'] = \frac{V\binom{k}{2}}{\binom{kV}{2}} = \frac{k-1}{kV-1}.$$

By union bound,

$$\Pr[\bigvee_{(i',j'):i>i'} w_{i,j} = w_{i',j'}] \le \sum_{(i',j'):i>i'} \Pr[w_{i,j} = w_{i',j'}] = \frac{(i-1)b(k-1)}{kV-1}.$$

Let indicator $\mathbf{1}_{i,j}$ denote that there exists no (i',j') such that $i > i'$ and $w_{i,j} = w_{i',j'}$. Hence

$$\mathbf{E}[\mathbf{1}_{i,j}] = 1 - \Pr[\bigvee_{(i',j'):i>i'} w_{i,j} = w_{i',j'}] \ge 1 - \frac{(i-1)b(k-1)}{kV-1}.$$

Following the collision tolerance condition enforced in the bucket building process of Section 4.4.2, and by arranging the orders of bucket locations for each term, and the term copies in each bucket, we can satisfy that

1. only the first two bucket locations for any term can be the same, and

2. only the first two term copies in each bucket can be the same.

Hence the average number of terms which the matched buckets are obfuscated over is at least

$$\mathbf{E}[\sum_{i=2}^{k}\sum_{j=2}^{b}\mathbf{1}_{i,j}] = \sum_{i=2}^{k}\sum_{j=2}^{b}\mathbf{E}[\mathbf{1}_{i,j}] \geq \sum_{i=2}^{k}\sum_{j=2}^{b}\left(1 - \frac{(i-1)b(k-1)}{kV-1}\right)$$

$$= (b-1)(k-1) - \frac{b(b-1)k(k-1)^2}{2kV-2}.$$

Since each of the q terms in the query has the above lower bound of obfuscation, raising to the power of q gives the lower bound for the entire query obfuscation. ■

From the above result, any term t with k duplicates in a given inverted index of V terms with b terms per bucket, on average term t is obfuscated by at least $(b-1)(k-1)(1-\frac{bk(k-1)}{2kV-2})$ different terms. Since a term in REARGUARD is obfuscated by $g-1$ terms in a group of size g, we can choose $(b-1)(k-1) \approx g$ so that privacy protection in MII is competitive to REARGUARD at least in terms of term and query obfuscation.

4.6.3 Leakage profile

We list all the following information that can be observed by a server when processing a query with MII. 1) *Static size information.* The number of term buckets in the index. The number of documents in each term bucket. 2) *Dynamic query size information.* The number of search terms in each query. The size of matched documents with padded results for a query. 3) *Term bucket access patterns.* The set of term buckets accessed for each query is exposed. The server can compute statistical information such as bucket access frequency. When two or more term buckets are queried, the set of documents appearing in these buckets may be leaked. Note that this only leaks positions of documents in these buckets, not real documents IDs. 4) *Feature size patterns.* The number of features of each document in each bucket posting list is exposed. This also tells the adversary how many of the b terms of the bucket are contained in each document.

Among all leakages specified above, there is no known privacy issue on learning the number of search terms and buckets used. For the length of each bucket, since the bucket has b randomly-mixed terms where $b > 1$, it is unlikely that the server can calculate the length of the posting list for a real term. For the result size information, since we pad unmatched or unselected documents, the server is not able to identify the real size. For the access patterns, since a bucket contains multiple terms that a server cannot differentiate, it is unlikely that the server can accurately compute the frequency of terms that appear in a history of queries, and calculate the document sharing pattern between postings of real terms.

4.7 Complexity Comparison

Table 4.2 gives a comparison of the index storage space and document matching time of MII, with the extended REARGUARD scheme [64] for handling q terms when the group size is g. The space cost is represented by the number of integers used to store the index before compression. The original REARGUARD does not deal with multiple search keywords or ranking also, and our extension is based on the best option we choose. Thus this comparison is illustrative to explain a cost advantage of MII over REARGUARD under certain assumptions. The time cost listed includes server side disk I/O and in-memory data processing. Notice that decryption of posting records can be conducted in the entire list for each term block, thus it is relatively fast. The client-side query processing cost of REARGUARD and MII is comparable, and is relatively insignificant.

Assumptions. Let n be the number of documents in a dataset, and V be the number of unique terms. It is known that the number of documents in each term's posting list often follows a Zipf-like distribution, and Table 4.2 illustrates the complexity difference under a simple Zipf distribution: assuming that the length of posting list for the i-th popular term is $\frac{n}{i}$. The longest posting list length of a term is n, and the average posting list length is $\frac{1}{V}\sum_{i=1}^{V}\frac{n}{i} \approx \frac{n\ln V}{V}$.

The space cost of MII is the sum of all posting lists multiplied by k because of term duplication. The query processing time cost of MII is proportional to the average posting list length multiplied by b and q.

For REARGUARD, with each posting list group of size g and following the Zipf distribution, we can show that the average posting list length of each group is $\Theta(\frac{ng}{V}\ln\frac{V}{g})$, considering V is large and $V >> g$. We explain the dominating space cost of REARGUARD as follows. With each posting list group of size g, define $G = V/g$ as the number of groups. Without loss of generality, suppose terms $w_1, \cdots, w_V$ are sorted in a descending order of their posting list lengths. Since these terms are randomly grouped, and these lists are padded uniformly to the same length for each group, we estimate the average posting list length after padding as follows. We examine the probability of each term w_i being the one with the longest posting list in its group, where i is from 1 to V. In that case, the corresponding group has a total list length of $g \cdot \frac{n}{i}$. The above case is not true when there exists j with $j < i$ such that w_i is in the same group of w_j, because the posting list of w_j is longer than that of w_i. Define indicator $\mathbf{1}_i$ as the event that w_i is not in the same group of any w_j for $j < i$. $\Pr[\mathbf{1}_i = 0] \leq \sum_{j=1}^{i-1}\frac{1}{G} = \frac{i-1}{G}$. Thus we have $\Pr[\mathbf{1}_i = 1] \geq \max\left\{0, 1-\frac{i-1}{G}\right\}$. Then let C be the space cost of padded group-based posting lists, and obviously C can be expressed as $C = \sum_{i=1}^{V}\mathbf{1}_i \cdot \frac{ng}{i}$. Therefore, the average posting list space cost $\mathbf{E}[C]$ is

$$\begin{aligned}\mathbf{E}[C] &= \sum_{i=1}^{V}\Pr[\mathbf{1}_i = 1]\frac{ng}{i} \geq \sum_{i=1}^{V}\max\left\{0, 1-\frac{i-1}{G}\right\}\cdot\frac{ng}{i}\\ &\geq \sum_{i=1}^{G}\left(1-\frac{i}{G}\right)\frac{ng}{i} = \Omega(ng(\ln G - 1)).\end{aligned}$$

Following the above analysis, we can further show that the upper bound of the average posting list length is $O(\frac{ng}{V}\ln G)$ for REARGUARD. Thus with $V >> g$ and large G, the average posting list length is $\Theta(\frac{ng}{V}\ln G)$. Given the fact that each posting list in a group has to be processed for each query term, the time cost is this number multiplied by g and q.

Table 4.2: A comparison of space and time complexity

	REARGUARD	MII
# of integers in the index	$\Theta(V + ng\ln\frac{V}{g})$	$\Theta(\frac{kV}{b} + kn\ln V)$
Server-side time	$\Theta(\frac{qng^2}{V}\ln\frac{V}{g})$	$\Theta(\frac{qbn}{V}\ln V)$

In general, b and k are small constants while g needs to be reasonably large. Assuming $n >> V$ and $V >> g$, the ratio of space cost of REARGUARD over MII is approximately $\Theta(\frac{g}{kb})$, thus the index storage space of MII should be the same as that of REARGUARD, under this simple Zipf distribution when choosing $g = kb$. In addition, the time cost ratio of REARGUARD over MII is about $\Theta(\frac{g^2}{b})$. As $g > b$, MII is significantly faster.

4.8 Evaluations

Experimental Settings. We evaluate disjunctive queries on MII and three other baselines, including block-max WAND, exhaustive OR, and REARGUARD. Given the fact that there is no standard IR toolkits supported by Intel SGX, we built our implementations for MII and all baselines with the C/C++ library under the SGX programming environment, which can make fair comparisons. The implementations of REARGUARD and MII access their corresponding encrypted indexes. All indexes are compressed using Simple-9 [72] before any encryption and this compression is simple with a reasonable effectiveness for our setting [73]. Our implementations of BMW and exhaustive OR directly access unencrypted indexes. We let $g = 85$ for REARGUARD following the setting in [64]. We also let $k = 18$ and $b = 6$ for MII. In this case, $g = (k-1)(b-1)$ and thus the privacy level of MII is approximately the same as that of REARGUARD. Experiments are conducted on a single machine with Intel Core i7-9700K CPU 3.60GHz, 32GB RAM, Samsung 970 NVMe SSD running Ubuntu 18.04 with Intel SGX Linux 2.7 SDK.

Datasets. Two TREC collections are evaluated: ClueWeb-09-Cat-B and TREC disks 4&5. TREC4&5 has about 0.5 million documents, and ClueWeb has nearly 30 millions documents after removing those with Waterloo spam score [55] below 40. We assume that ClueWeb can be hosted on multiple cores for parallel query processing. Thus we randomly partition this dataset and use one partition with 1 million documents for performance assessment. For TREC4&5, 530, 348 terms in total are indexed including Stop Words. For ClueWeb, with discarding terms whose frequency no more than 2, there are in total 1, 239, 769 terms indexed. For query processing, there are 837 queries for ClueWeb from WebTrack 2010-2012 and Millions Query Track. TREC4&5 uses 250 queries from Robust 2004. All test queries have 1-5 query keywords. Both documents and query words are stemmed using the Krovetz stemmer [57].

A comparison of query processing times. In Table 4.3, the query time for document retrieval with MII and three baselines are listed. An average time is reported in each different query length, and the rightmost column is the average time for all test queries. The time cost in Table 4.3 measures the duration starting from the time when the server receives (encrypted) queries, to the time when all top 1000 matched documents are derived. We make sure the index is not cached before each single query.

MII vs. REARGUARD. We observe that compared with REARGUARD, MII is 6.73x faster for TREC4&5, and 10.71x faster for ClueWeb. Notice that the time spent on the disk I/O to fetch posting lists occupies about 80% of time in REARGUARD, and about 56% in MII. The reduction ratio is not as high as the predicted number in Section 4.7, because the estimation does not include the startup cost of each I/O operation to access the SSD, and our test data is not exactly a simple Zipf distribution. The evaluation, however, still agrees on the trend that MII reduces the matching time significantly, including all the cases when the test query length varies.

Oblivious search vs. unprotected BMW. MII is slower than BMW by 3.32x and 4.90x mainly because of block-max WAND skipping documents based on top k thresholds and block-max scores. When compared with exhaustive OR, MII is 1.04x and 1.09x slower because there are posting duplicates in masked buckets. Notice that our implementation of block-max WAND is 3.18x and 4.48x faster than that of exhaustive OR, which is not as high as the ratio claimed by Ding and Suel [14]. This is because our query processing time includes the time of loading posting lists from disks, which is not considered in the block-max WAND paper.

Storage cost. As mentioned above, all indexes were compressed using Simple-9 compression technique. Using different compression methods did affect our reported storage sizes [73], but not by much. BMW and exhaustive OR use the same unencrypted index, which is 0.2GB for TREC4&5, and 11.8GB for ClueWeb. The storage size for MII is 3.1GB for TREC4&5, and 207.5GB for ClueWeb, which includes masked posting lists, vocabulary, and term buckets. These costs are around 15.5x and 17.6x as big as those of the unencrypted index in both datasets because of posting duplication. For REARGUARD, with the group size g as 85, the index size with group-wide posting padding is 8.5GB for TREC4&5, and 709.1GB for ClueWeb. Thus the index sizes of REARGUARD are about 2.7x and 3.4x as big as those of MII for our two datasets. These ratios are not too far away from the predicted ratio $\Theta(\frac{g}{k})$ in Section 4.7, which is around 4.7x.

Impact of different term bucket settings. Table 4.4 shows the impact of different term bucket settings, k and b, average query processing time and storage cost for our test datasets. The first setting, $k = 18$ and $b = 6$, is the one used in Table 4.3. The second one, $k = 6$ and $b = 18$, has the approximately same privacy level as that of the

Table 4.3: Comparing document retrieval time (milliseconds)

# Query Words		1	2	3	4-5	Average
TREC4&5	BMW	1	2	3	4	2.8
	Exhaustive OR	2	7	10	11	8.9
	REARGUARD	21	45	72	83	62.6
	MII	3	8	10	11	9.3
ClueWeb	BMW	31	73	134	254	125.7
	Exhaustive OR	116	311	583	1,212	560.1
	REARGUARD	620	2,112	6,409	16,723	6,557.9
	MII	133	318	631	1,327	612.4

Table 4.4: Comparing different term bucketing settings in MII

MII Settings		Avg. Query Time (ms)	Index Size (GB)
TREC4&5	k = 18, b = 6	9.3	3.1
	k = 6, b = 18	11.4 (1.2x)	0.9 (0.29x)
ClueWeb	k = 18, b = 6	612.4	207.5
	k = 6, b = 18	924.7 (1.5x)	67.1 (0.32x)

first setting. For storage sizes, close to what we predicted earlier, when k being 0.33x, MII costs around 0.29x and 0.32x as big as the storage cost of the first setting. This observation is consistent with Table 4.2 in Section 4.7 where MII space cost is proportional to k. With b being 3x larger, the second setting costs 1.2x and 1.5x query processing time compared to the first one on two datasets. The increasing trend modestly agrees with Table 4.2 and the increasing ratio is smaller than expected due to the discrepancy in posting length distribution estimation.

Evaluating oblivious feature extraction. As discussed at the end of Section 4.4, our oblivious feature extraction method is expected to reduce the storage cost greatly, in a situation where a document in a bucket posting record only matches some of those b terms in the bucket, compared with the naive oblivious method described in Section 4.4.3. Our experiments show that, when $k = 18$ and $b = 6$, the naive method costs around 2.5x and 2.6x storage space as those of our proposed oblivious method in two datasets, which shows the advantage of our space-conscious oblivious design. In addition, the time cost difference is insignificant to the total query processing time, as both of them are linear with the value of b.

4.9 Summary

This chapter proposes an oblivious document retrieval scheme with an obfuscated inverted index to hide document-term association and avoid pattern leakage of memory access operations for privacy protection. Our evaluation shows MII achieves an up-to 18.9x matching time speed-up over REARGUARD while the storage size is up-to 3.4x smaller. The oblivious but exhaustive approach in MII is still significantly slower than BMW [14], which represents a degradation of efficiency traded for privacy because WAND-based optimization in BMW designed without privacy constraints [14, 15, 17] is vulnerable to leakage-abuse attacks as studied in Section 4.3.

Intel SGX has around 90MB usable protected buffer memory [31], which is big enough to contain all posting records of the searched term buckets for our tested datasets. For hosting buckets with longer lists, there could be some slowdown in matching due to buffer pages being swapped into/out of the untrusted memory. TEEs like SGX still reside on the server machines and the risk such as physical attacks [31] exists, which is outside the scope of this study. Our solution provides an alternative approach to the software-only privacy-preserving solutions for document matching. A future work is to integrate with privacy-preserving ranking.

Chapter 5

Oblivious Top-k Document Retrieval with Dynamic Pruning

5.1 Introduction

To protect the leakage of data-dependent memory access patterns for keyword matching of documents, oblivious solutions that hide memory access pattern in document retrieval have been proposed in [64] and in Chapter 4. These solutions do not use the ORAM-based data storage [74], which can provide a higher level of security protection by hiding memory access patterns [75, 76, 77]. There is a tradeoff that ORAM based solutions become much more expensive with a significantly-longer processing time, and this chapter is focused on optimization for ORAM-based document retrieval. An ORAM-based document retrieval scheme has been proposed in [29], and that study does not consider top-k retrieval and uses exhaustive scanning of all matched posting records without skipping low-scoring documents. As document retrieval represents a significant portion of time cost in query processing, dynamic pruning techniques such as BlockMax WAND (BMW) [14, 15, 17] have been proposed in the information retrieval community and they are shown to be important for a faster query response. Our chapter 3 also describes how to boost the performance of BMW through window navigation with adaptive probing. However, so far there is no solution developed for oblivious document retrieval which can effectively prune unnecessary document visitations following a style similar to BMW.

The contribution of this chapter studies how to integrate document skipping in a privacy-preserving framework. To our best knowledge, this is the first effort to propose a top-k document retrieval scheme that conducts dynamic pruning with oblivious posting record access through the hardware-assisted ORAM. In detail, we discuss the privacy issue of inter-term and intra-term data access patterns, and present oblivious retrieval algorithms with additive ranking that mitigate the leakage of access patterns. To supporting dynamic pruning while reduce ORAM access time, we also present term-specific path caching for oblivious posting block retrieval in a skipping manner. Our evaluations on a large-scale dataset show that our techniques can optimize the efficiency of ORAM-based oblivious top-k retrieval in a TEE.

5.2 Background

Privacy Requirements and Threat model. There are three entities in a cloud system: data owner, search user (client) and cloud server. The data owner has a collection of documents to be outsourced by the cloud server. To enable the searching and ranking functionality, the data owner needs to encrypt the documents, the inverted index, and the ranking model for the cloud server outsourcing. This chapter assumes that the data owner and the search user are the same entity. The client builds an encrypted but searchable index, lets a server host such index, and a server only runs a query issued by such a client. The server is honest-but-curious, i.e., the server will honestly follow the client's protocol, but will also try to learn any private information from the client data based on what the server can observe from the hosted data and runtime information. To conduct a search query, the client sends several encrypted keywords and related information to the server. The privacy requirement is that the server still be able to search the encrypted inverted index and correctly find the desired documents without knowing these documents or search queries.

Efficiency requirements and assumptions. The query processing time for document retrieval should allow the interactive user experience, namely a response time around a second or less is required. Use of multiple parallel cores or multiple servers is allowed, and we assume the data index can be partitioned uniformly across multiple machines or cores so that query processing may be conducted in parallel and the parallel results may be aggregated at the end. We mainly describe on how query processing can be one conducted in one machine. Our research targets at inexpensive client-server communication since extensive multi-round communication between the server and client (e.g. [65, 66]) incurs excessive high communication cost and response latency.

TEE and Oblivious RAM. We assume a host server has a processor which contains a trusted computing environment (TEE) with a protected memory region. An example of TEE is Intel SGX and Microsoft cloud currently use SGX-equipped Intel processors. A TEE has its own memory space and a host processor can monitor all accesses to the TEE's memory space, knowing which addresses are visited by the code running inside the TEE, but it cannot view the content accessed by the TEE's code. One weakness is that an application that runs its privacy-sensitive code in TEE can still leak memory access patterns, which can lead malicious attacks.

A popular way to prevent the leakage of memory access patterns is to use Oblivious RAM (ORAM) [75, 76, 77] with TEE (e.g. [74]). An ORAM is a data storage which supports two simple operations: read and write a block of memory using its unique ID.

During such an operation, the server hosting this ORAM does not learn the data access pattern, namely a sequence of memory locations accessed. A popular implementation of ORAM uses Path ORAM [78] while other options are available [79] Note that the choice of a different ORAM does not have fundamental impact on our design.

The work of [74]) runs a trusted client in a TEE (Intel SGX) for a path ORAM which translates a clients logical data block request to a sequence of accesses to an untrusted host storage. It organizes untrusted host storage as a binary tree of height $O(\log N)$ where N is the number of data blocks, and each node in this tree can host some encrypted data blocks. The TEE maintains a position mapping table which maps a data block ID to a path in the tree, and also maintains a local store called *stash* that can save some data blocks as an "overflow" of the binary tree. The path ORAM implementation of a read request of a block is described briefly as follows and the write request is handled similarity. The client in a TEE looks up the position map to determine the path called P that the block is mapped to, downloads all data blocks along that path, decrypts and save them in the stash. Notice one of them is the desired block. After that, an extra step is performed to avoid the leakage of the data access patterns the client assigns a new random path in the tree to the accessed block, upload as many blocks as possible from the stash to the old path P of the binary tree with some position suffering. Each read or write block operation in a path ORAM costs $O(\log N)$ I/O operations. Because of random block position re-arrangement involved during data access, I/O involved for each block access operation is very expensive, and our design intends to minimize the amount of ORAM block access during document retrieval.

Oblivious Document Retrieval and Building Blocks. As mentioned in Section 5.1, oblivious solutions for document retrieval have been studied [64, 80], and the more secure one is the work of [29] which uses hardware-assisted ORAM to hide memory access patterns. All these studies have used exhaustive search and none of them has considered dynamic index pruning optimization. The goal of our work is to achieve oblivious document pruning similar as BMW while leveraging a hardware-assisted ORAM.

There are several building blocks commonly summarized below for achieving an oblivious algorithm design, i.e. those algorithms with memory access behaviors not depending on specific input data.

- **Oblivious Conditional Selection.** A conditional selection method takes three arguments as input, *cond*, *a*, and *b*. If *cond* is true, *a* is returned. Otherwise, *b* is returned. We note this basic function as o_select $(cond, a, b)$ in our paper. In order to make this method data oblivious, only register data manipulations should be used, after loading three inputs into registers. We leveraged x86-64 assembly languages to implement an oblivious version of conditional selection method, *test* and *cmovz()* in specific. The input data for this oblivious one can be 32-bit or 64-bit integers.

- **Oblivious Sorting.** We can use a sorting network to achieve oblivious sorting [69], for example, Bitonic Sorting. The time complexity is $O(n \log^2(n))$, where n is the number of element to be sorted. This sorting algorithm can also be implemented to run in parallel, which has a speed-up n, resulting in a time complexity as $\log^2(n)$.

5.3 Attacks on the Posting Access Pattern in Top-k Retrieval

We intend to store blocks of posting records as an ORAM block, and even the access pattern of such blocks is hidden, a privacy attack can still exist when document retrieval accesses the content within a block. In this section, we discuss privacy attacks on the posting access pattern revealed in the top-k retrieval process.

Definition and assumption. Given a query term whose posting list length is L, posting access pattern of this query term is the sequence of access operations on all postings in this list:

$$((op_1, id_1), ..., (op_L, id_L)),$$

where op_i can be one of $Skip$, $EvalFail$, and $EvalSucceed$. $Skip$ means this posting is skipped without being evaluated. $EvalFail$ means this posting is evaluated but not added into the top-k list. $EvalSucceed$ means this posting is evaluated and added into the top-k list. id_i is a unique identifier for each posting record.

We assume that the untrusted server can observe the posting access pattern for each term in a top-k query processed in a TEE. This assumption can exist if the retrieval program running in a TEE is not data oblivious, i.e. different instructions are executed depending on the value of input data in those branched code. More details on this assumption can be found in the work by Xu et al. [81]. Note that query keywords and an inverted index are always encrypted from the view of the untrusted server during query processing. We also assume the run-time behavior of a top-k retrieval program is deterministic, only dependent on the input query terms. Also assume in a worst case, the binary code can be loaded into the TEE with the same starting memory address.

Previous attacks. Once the untrusted server can obtain the posting access pattern for each query term, there are previous attacks that can be mounted:

- Clearly, a query repetition pattern can be induced from the posting access pattern if it is impossible to have two different queries whose posting access patterns are the same. The work by Liu et al. [61] presents an attack using a query repetition pattern if some prior knowledge can be available to the server.

- If the index being searched supports dynamical updates, e.g. deletions and additions of documents, since the posting access pattern reveals the length of posting lists and it is rare to have two posting lists with the same length, we can leverage the work by Blackstone et al. [82] that describes document injection attacks based on the volume leakage.

Our modified document injection attack. In addition to the previous attacks mentioned above, we hereby show a modified document attack based on the work by Zhang et al. [83] that proposes document injection attacks on the document identifier pattern, i.e. different document can be differentiated by their encrypted ids.

The goal of our attack is to recover the plaintext of encrypted queries. The general idea of our attack is: 1) Firstly, identify a set of encrypted queries Q that contain keywords appearing in a document d injected by the server. 2) By injecting another document d' that differs with d by having only one extra term t, the server continues to identify another set of encrypted queries Q' that contain keywords appearing in a document d'. Thus the server can know, for those encrypted queries making Q different from Q', they must contain that extra term t.

The following are detailed steps:

- **Step 1.** The server firstly collects posting access patterns for a number of queries that are processed within a range of time. Let $\mathcal{T}$ be a set of these patterns. As we assume above, each query term has a unique posting access pattern.

- **Step 2.** The server injects one document into the encrypted index in TEE. The content of this document is known to the server, which means the server knows every query keyword that appears in this document. Assume this new document is appended to the end of every posting list that needs to be updated.

- **Step 3.** After the new document is injected, when the client issues a new query, the server can record the posting access pattern, t. The server compares t with every traces collected in $\mathcal{T}$. The comparing result can be discussed as three cases: 1) The posting access pattern of this new query has an exact copy in $\mathcal{T}$. This means this query has been processed in the time range from the Step 1, and postings lists involved have not been changed. This indicates the injected file does not contain any keyword in this query. 2) There exists a posting access pattern t' in $\mathcal{T}$, such that t is a subsequence of t', In this case, the server can infer the injected document contains keywords in this query. 3) The above two cases do not exist. This indicate this new query has not been processed in that time range from the Step 1. We consider this case can be excluded if the length that time range is long enough.

- **Step 4.** By repeating Step 3, the server can collect a set of client's queries Q containing the keywords in the injected document.

- **Step 5.** Inject another document that differs from the last injected document by having only one extra term. Repeat Step 3 and 4 for this new injected document. the server can collect another set of client's queries Q'.

- **Step 6.** By comparing Q and Q', the server can pick up all client queries that have the extra term that make two injected documents different. Note that the plaintext of this extra term is known to the server (and injected into the known documents), thus the server can obtain the mapping from the plaintext to its encrypted number of this extra term. Once a server gathers the internal representation of many such terms used in the hosted index, it can learn more about the plaintext content of the corresponding dataset.

It should be noted that the previous work on attacks [12, 11] often rely on the co-occurrence analysis to reveal plaintext. Our above work provides an alternative attack based on the access patterns of posting lists. It also gives a motivation on why we need to avoid the leakage of the access patterns described in the beginning of this section, purely using ORAMs does not sufficiently hide such patterns. With such a reason, a traditional BMW based optimization leaks posting list access patterns, and can aid the above attack that exploits the leakage of posting list access patterns. To thwart such an attack, as considered in Section 5.4, our design emphasizes oblivious computing during index access while still supporting index pruning for maximized efficiency.

5.4 Design Considerations with Privacy Enhancement

There are two challenges in designing efficient document retrieval in a TEE. First, while dynamic pruning is a standard optimization technique, applying such a technique can easily leak data access patterns, which may compromise the privacy. Second, the use of ORAM hides data access patterns, but it carries a significant overhead especially for a dataset that has over billions of term-document pairs. In the paragraphs below we discuss several design issues further in details.

1. Data access patterns and privacy protection strategies. During query processing that conducts an intersection based on the posting lists of multiple search terms for document retrieval, the address sequence of accessing search terms and their posting lists in data storage becomes a data access pattern. Specifically there are two types of data access patterns leaked:

- **Inter-term data access patterns.** The recognition of different terms accessed allows the server to accumulate knowledge on the frequency of terms accessed. The work by Liu et al. [61] presents an attack that exploits query term repetition patterns under certain conditions.

- **Intra-term data access patterns.** The recognition of different posting records accessed within the posting list of a term allows the server to memorize a special data access sequence for each term and thus differentiate different terms using such sequences.

There are three strategies that can be used to hide the above access patterns.

- In the previous work, Sun et al. proposed REARGUARD [64] which can partially hides this leakage using posting list obfuscation. Shao et al. [80] proposed an efficiency-enhanced oblivious search protocol based on term-bucket obfuscation. However, these studies can still leak group-based access patterns. For example, in REARGUARD, all searchable terms are divided into multiple groups, and the access pattern of terms in the same group are hidden, while the group access pattern is still revealed.
- To completely prevent such leakage, Oblivious RAM (ORAM) [75, 76, 77] is known as a more robust way with more secure properties than other access pattern hiding techniques. Thus we will focus on ORAM-based search index design by hosting the searched index in ORAM blocks. The downside of using ORAM for intra-term data access patterns is the large number of posting records makes the time cost of accessing each individual posting data unaffordable compared to that in a no-privacy setting.
- **Oblivious access and computing.** Accessing an ORAM item is expensive with a complexity of $O(\log N)$ where N is the number of items in this ORAM. Thus each ORAM item tends to be coarse grained, in a number of data bytes, and is called an ORAM block. For an inverted index, each ORAM block typically holds a posting list or blocks of posting records within a posting list. Thus we can hide the access patterns of different search terms by storing their posting lists in different ORAM blocks, but the top-k search with an ORAM can still reveal the access pattern of different ORAM blocks through the access pattern of posting records within each block.

The solution for the above issue is oblivious computing, namely applying the same set of computations to every posting or every document in the top-k, which leverages those oblivious instructions presented in the background section.

2. Challenges with dynamic posting record pruning. It has been a standard optimization technique [15, 14] to prune unnecessary posting records during search. However, with dynamic pruning, there is no ideal solution currently that completely hides this access leakage even if the posting lists are stored in an ORAM. The challenge is that with dynamic pruning, the posting list of a query term is accessed once per each posting block that is not pruned, and also documents visited are typically in an increasing order of their ID values. That leaves a unique memory access trace pattern for each term searched. Thus we cannot just simply adopt the traditional pruning techniques such as BMW [14], and need to design a way that still prunes certain records while still hiding access patterns.

Our design idea is a two-phase retrieval process that prunes unwanted I/O accesses by executing uniform retrieval process that presents the same data access pattern for a group of queries, such that the server cannot differentiate any two from the same group.

3. Position map caching with pointer-based optimization. Accessing an ORAM block is still very expensive. To boost the top-k retrieval process, our next idea is motivated by the previous work on pointer-based optimization for ORAM. The goal is to speed up the ORAM access by locating an item in the ORAM quickly. Although it has been shown that this optimization can benefit the process of accessing a linked-list, it is no solution how to accelerate the process of accessing a posting list when pruning is adopted. We have developed a term-specific position caching technique to reduce the time cost to speedup ORAM access structured for posting lists.

5.5 Two-phase Oblivious Top-k Document Retrieval

5.5.1 Overview

One of the key ideas in dynamic pruning is to assess the minimum ranking score threshold of documents that appear in the top-k positions, and use such a threshold to prune documents whose estimated upperbound ranking scores are below the threshold. The earlier work includes the computation of such a threshold before document retrieval [41, 45, 40]. Our idea in designing the oblivious document retrieval with pruning is to view document retrieval as a visitation on a sequence of posting block windows. Each window includes all posting blocks that intersects the same range of the document IDs for a given set of query terms.

Our **T**wo-**P**hase (TP) protocol for oblivious top-k document retrieval is outlined below at a high level.

- Phase 1, which we call **Probing Phase**, finds Z promising posting block windows, and applies an oblivious top-k retrieval algorithm to these Z posting block windows, thereby producing a positive, hopefully tight top-k threshold Ω;
- Phase 2, which we call **Final Pass Phase**, gathers all unvisited posting blocks windows which are not selected in Probing Phase, and filters out those windows whose documents have no chance to get into the final top-k based on Ω. Then this phase applies an oblivious top-k retrieval on the surviving posting blocks, and produces the final top-k results.

Figure 5.1 gives the main data structure and computation flow of our two-phase scheme. The computation involved at each phase is oblivious. Namely for any two queries with the same length, the sequences of memory access behaviors they present are

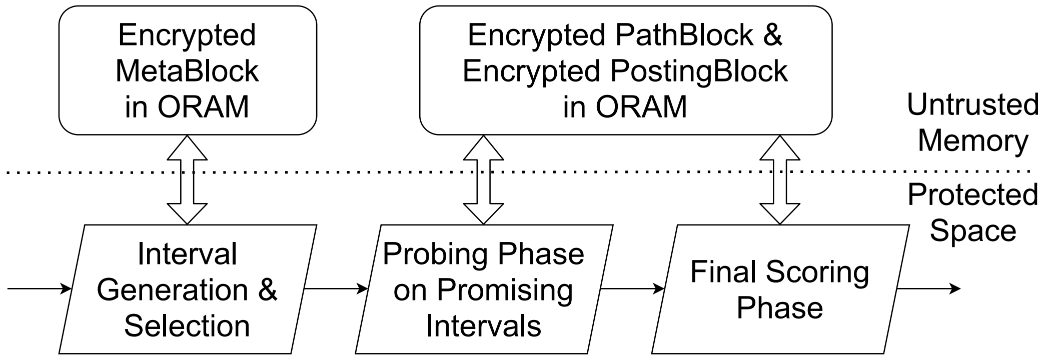

Encrypted MetaBlock in ORAM
Encrypted PathBlock & Encrypted PostingBlock in ORAM
Untrusted Memory
Protected Space
Interval Generation & Selection
Probing Phase on Promising Intervals
Final Scoring Phase

Figure 5.1: Server-side two-phase oblivious top-k retrieval flow

identical with using our framework. In addition, all data needed for query processing are encrypted when outside the enclave.

5.5.2 Data structures

We first present the data structure involved to facilitate two-phase document retrieval. As assumed in the previous work, the posting list of each term is sorted in ascending order of document IDs and divided into posting blocks. With leveraging ORAM to hide the access pattern, we package these posting blocks into encrypted ORAM blocks. For simplicity, we now assume each posting block is packaged as one ORAM block. The next section we discuss other approaches on how to package posting blocks.

Each posting record includes a document ID and its ranking score with a term. For simplicity, $PostingData$ stands for the data of this posting record. We call ORAM blocks holding posting blocks as $PostingBlock$, and each block can be represented as a tuple: ($BlockID$, $PostingData$, $PathID$) where

- $BlockID$ is a unique ID assigned to each posting block across all searchable terms, thus also a unique ID for each $PostingBlock$.
- $BlockID$ can be used as a key in ORAM to retrieve the corresponding $PostingBlock$.

- $PathID$ is the position information for $PostingBlock$ in the ORAM.

In order to support dynamic pruning, we package meta-information together for each posting block, and it is a tuple of four integers: ($BlockID$, $MinDocID$, $MaxDocID$, $BlockMaxScore$), where

- 1) $BlockID$ is the posting block ID, same as the one in $PostingBlock$;
- 2) $MinDocID$ is the smallest document ID in this block;
- 3) $MaxDocID$ is the largest document ID in this block;
- and 4) $BlockMaxScore$ is the highest ranking score among all documents in this block.

The above information for all posting blocks of one term is packed into one or a list of encrypted ORAM blocks, which we call $MetaBlock$, along with $TermID$ and $PathID$. $TermID$ is unique for each searchable term, and can be used to retrieve its $MetaBlock$ from the ORAM.

Note that in an ORAM-based architecture, all ORAM blocks has the same size that can typically vary from 4B to 10KB. Thus padding is needed for packaging $PostingBlock$ and $MetaBlock$ when there is not enough data for filling up a whole block. In addition, these ORAM data blocks are encrypted and can only be decrypted or re-encrypted inside a TEE with a secrete key.

5.5.3 Online oblivious top-k retrieval

Input and output of online top-k ranking. The input is the term IDs of all searched terms. Posting lists of searched terms can be retrieved by accessing corresponding $MetaBlock$s and then $PostingBlock$s from the ORAM. Among these posting lists, some postings can have the same document IDs. We assume each posting contains a document ID and its corresponding additive ranking score.

The top-k retrieval algorithm aggregates ranking scores for every document ID appearing in different posting lists, outputs a list of document IDs based on the descending order of all aggregated ranking scores, and returns the top-k document IDs.

Each rank score for document d is $\sum_i \alpha_i * f_i(d)$ where α_i is the weight for the i-th search term and $f_i(d)$ is the feature value of document d for this term. α_i is the corresponding feature weight. For simplicity, we assume α_i=1 in our paper because the weight can be embedded into the feature value.

Online oblivious top-k retrieval steps. Our query processing has the following four steps:

- **Load $MetaBlock$s from ORAM.** Given a set of query terms, the client firstly retrieves their $TermID$s through hashing or from a local key-value store which should be affordable in terms of the storage size. All these $TermID$s with encryption are sent to the server which contains a TEE. Then the TEE accesses the corresponding $MetaBlock$s through ORAM, which returns the metadata tuples for all posting blocks of the given query terms.

- **Window generation and selection.** See details in Section 5.5.4.

- **Execute Phase 1.** With all derived promising windows, the corresponding $PostingBlock$s of posting blocks intersecting promising intervals can be retrieved through ORAM. We apply an oblivious top-k computation algorithm discussed in Section 5.5.5 to all loaded posting blocks.

- **Execute Phase 2.** With a lower bound of top-k threshold derived from Phase 1, check all posting intervals in the second step against this lower bound: Intervals whose upper bound scores are lower than this thresh bound can be safely pruned. Since the list of intervals is already sorted obliviously, the location of intervals being pruned does not reveal any unique information for this query except the number of intervals. The TEE processes the remaining qualified intervals by loading posting blocks using their posting block IDs, applying an oblivious retrieval algorithm to these blocks, and finalizing the top-k result for this query.

5.5.4 Oblivious window generation and selection

Here we present how to retrieve the promising posting blocks for Phase 1. The previous work by Shao et al. [84] describes a method to select promising windows and probe all posting blocks intersecting with the selected windows. We follow a similar approach here while making every computation oblivious to prevent intra-term data access pattern leakage.

As discussed by the previous work [33, 84], a document ID window is a range of consecutive searchable document ID. In specific, we use block-aligned windows in our algorithm: given the whole range of searchable document IDs and all posting blocks for search terms, all distinctive $MaxDocID(p)$ act as the boundary IDs to produce a disjoint set of document ID windows, where p is any posting block involved and $MaxDocID(p)$ is the maximum document ID in p.

We also adopt the method of selecting promising windows. Firstly, a posting block p intersects with a window W, denoted by $p \in W$, if any document ID in this window is between $MinDocID(p)$ and $MaxDocID(p)$ (both inclusively). Given a set of posting blocks B, the maximum window score of a window W is defined as

$$WinMax(W) = \sum_{p \in B \wedge p \in W} BlockMax(p).$$

We can pick out those promising windows with highest $WinMax$ scores through oblivious sorting.

Algorithm 3: Oblivious window generation in a TEE

```
Input: Block metadata tuples for all $q$ query terms. A block metadata tuple is
       represented as $(minD, maxD, maxS, blockID)$, where $minD$ is the
       minimum document ID, $maxD$ is the maximum document ID, $maxS$ is
       the block max score, and $blockID$ is the ORAM block ID.
Let $\mathbf{L}$ contain all block metadata tuples, $\mathbf{W}$ be the first $q$ block metadata tuples in
  $\mathbf{L}$, Windows be an empty list, and $\mathbf{d}$ be 0, i.e. the smallest searchable document
  ID;
Obliviously sort $\mathbf{L}$ based on $minD$;
while $\mathbf{W}$ is not empty do
    Obliviously select $\mathbf{d}' = \min_{meta \in \mathbf{W}}(meta.maxD)$;
    Let $\mathbf{W}' \subseteq \mathbf{W}$ contain all blocks in $\mathbf{W}$ that intersect with the ID range $(\mathbf{d}, \mathbf{d}')$.
      Append a new window $(\mathbf{d}, \mathbf{d}', sumS, arrayBlockID)$ to Windows, where
      $sumS = \sum_{meta \in \mathbf{W}'}(meta.maxS)$ and $arrayBlockID$ contains all blockIDs of
      metadata in $\mathbf{W}'$;
    Remove the block metadata $meta$ in $\mathbf{W}$ if $meta.maxD = \mathbf{d}'$;
    Add the next block metadata in $\mathbf{L}$ to $\mathbf{W}$;
    Let $\mathbf{d}$ be $\mathbf{d}'$;
end
Return Windows;
```

Algorithm 3 presents our oblivious window generation algorithm whose computation complexity is the same as that of oblivious sorting. To ensure oblivious property, $arrayBlockID$ has to be of size q and can be padded using null values. Then if this algorithm is executed in a TEE, given two inputs that have the same number of block metadata tuples, their memory access patterns are identical.

5.5.5 Oblivious top-k computation

Top-k computation takes as input posting records and outputs k document IDs, assuming those posting records contain at least k distinctive document IDs. There are many existing algorithms (such as Block-Max WAND [14]) for efficient top-k computation without privacy considerations, but they may reveal intra-term data access pattern as described in Section 5.4.

```
Algorithm 4: Oblivious top-$k$ computation in a TEE
  Input: The value of $k$. Posting records for all query terms. A posting record is a
         tuple of two integers: one is document ID $d$ and the other is additive
         ranking score $s$.
  Form all posting records in a list as $\mathbf{L}$;
  Obliviously sort posting records in $\mathbf{L}$ on the document ID;
  for each posting $(\boldsymbol{d}, \boldsymbol{s})$ in $\mathbf{L}$ do
      Let $(\mathbf{d}', \mathbf{s}')$ be the next posting in $\mathbf{L}$, otherwise break the loop;
      Let $equalBit$ be 1 if $\mathbf{d} = \mathbf{d}'$, otherwise 0;
      $\mathbf{s}' = \mathbf{s}' + equalBit * \mathbf{s}$;
      $\mathbf{s} = \mathbf{s} - equalBit * \mathbf{s}$;
  end
  Obliviously sort posting records in $\mathbf{L}$ on the score descendingly;
  Return document IDs in the first $k$ posting records of $\mathbf{L}$;
```

Here we present Algorithm 4 which describes an oblivious algorithm to conduct top-k computation. In other words, if this algorithm is executed in a TEE, i.e. the value of all data is hidden from the server, given two inputs that have the same number of posting records, their memory access patterns are identical. Algorithm 4 has a computation complexity $O(n \log^2(n))$ where n is the number of input posting records, if bitonic sorter is used for the oblivious sorting. Notice that the complexity is not related to k.

We acknowledge that there are also other algorithms for oblivious top-k computation. For example, one simple way is to maintain a top-k array and obliviously update each entry in this array for each posting record. The complexity of such a algorithm is $O(n*k)$. Compared to Algorithm 4, this simple one can be more efficient when k is small.

Fig. 5.2 gives an example on how two postings lists are processed for a query using Algorithm 4. Here each shape represents one posting including a document ID and an additive ranking score. All rectangle shapes belong to one query term, and all ellipses belong to another query term. Fig. 5.2(a) shows that all posting records are merged

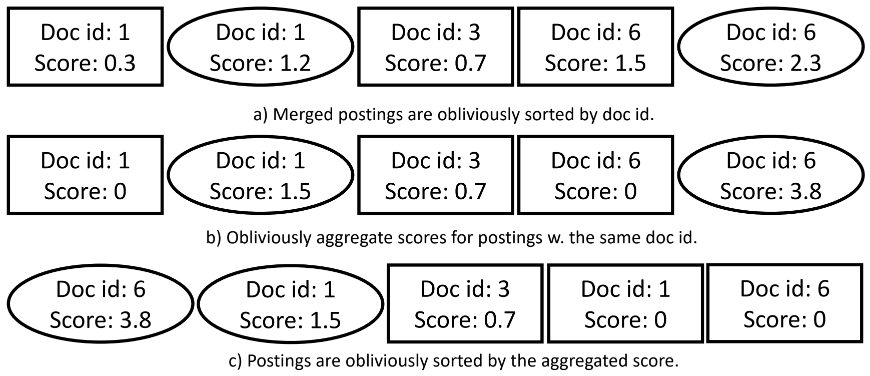


Figure 5.2: An example of oblivious top-k computation.

into one list, and they are obliviously sorted by the document IDs ascendingly. Fig. 5.2(b) corresponds to the state of **L** after the line 8 in Algorithm 4, and shows that the rank scores for the same document are aggregated to one posting record. For example, Document 1 has two postings, where one of them has a score of 1.5, which is the sum of 0.3 and 1.2, and the other record has a score of 0. In Fig. 5.2(c), all posting records in the merged list are obliviously sorted by their scores.

5.5.6 Posting block packing

Previous work [29] has shown that it is more efficient to pack all needed data in one ORAM block and access the ORAM once. Here given a partitioning of posting lists, i.e. posting blocks, we describe how to pack and store them in an ORAM, in order to optimize the data access cost during oblivious top-k retrieval. We define the packing size as the number of posting blocks in one ORAM block. Note that this packing size cannot be too large, which can result in many posting blocks being loaded when the posting list length of a query term is short, i.e. only a small portion of a large ORAM block is needed. On the other hand, this packing size being too small can make it inefficient for a query term that has a long posting list, i.e. it needs many ORAM blocks to fetch all posting blocks.

Here we describe a simple way to organize posting blocks into ORAM blocks: we bundle together those posting blocks that are next to each other based on document ID they contain. For example, given a list of posting blocks p_0, p_1, p_2, ..., if the packing size is 3, let p_0, p_1, p_2 be in the first ORAM block, and p_4, p_5, p_6 be in the second ORAM block, and so on. The intuition behind this is, those posting blocks in one ORAM block are very likely in the same window. Thus if a promising window is needed, we can use one ORAM access to fetch these posting blocks belonging to the same query term in this window.

5.5.7 Term-specific path caching

To optimize the ORAM access latency when considering skipping posting blocks, we propose term-specific path caching which is based on the pointer-based technique in [85].

Background on pointer-based ORAM. We hereby add details on the background of the pointer-based technique in ORAM [85]. Firstly, each ORAM access consists of three steps, with the input being a target block ID: 1) Retrieve a target path ID for the given block ID. 2) Fetch data blocks along the target path, and access the target block. 3) Evict data blocks back to the target path. Notice that, we can outsource the key-value store as another ORAM to the untrusted server such that minimizes the storage cost in the enclave. This approach is called recursive-ORAM, which can assure that the enclave has a constant size of storage cost, no matter the total number of ORAM data blocks is needed.

The pointer-based technique can optimize the access cost for accessing a linked list as the following: Assuming each linked list node is kept in one ORAM block, and the

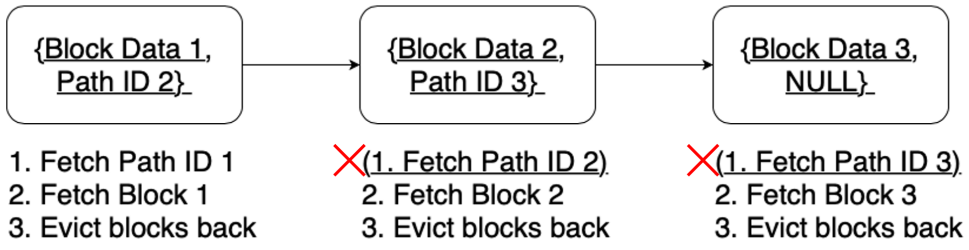


Figure 5.3: An example of pointer-based technique.

block b_2 has a node that is the successor of the node in the block b_1, we can augment the ORAM block b_1 with the path ID of b_1. In specific, the block b_1 contains the path ID of b_2 besides the node data in b_1. The benefit is that, when b_1 is accessed and fetched in the above step 2, we can read and overwrite the path ID of b_2 with no extra cost, which enables skipping the step 1 when accessing the block b_2. Fig. 5.3 shows an example of applying the pointer-based technique on a list of three ORAM blocks. Here Block 1 and Block 2 both have the path ID for their successive blocks, i.e. Block 2 and Block 3 respectively. Therefore, once Block 1 is accessed, the step 1 for accessing Block 2 and Block 3 can be omitted.

Although this technique can be applied directly to posting lists that are the same as linked lists [29], it cannot be applied to our algorithm when accessing and skipping posting blocks. It is because when a ORAM block b is skipped, the path ID information of the successive block cannot be retrieved. Thus we use term-specific path caching to overcome this issue.

Term-specific path caching. Since we have to retrieve some or all posting blocks for a query term from the ORAM, a special ORAM block called *PathBlock* for a query term can be constructed and maintained to contain the path IDs of all ORAM blocks that have posting blocks belonging to this term.

With the availability of this *PathBlock*, when executing Phase 1 and Phase 2, we can first fetch and load *PathBlock*s for all searched query terms, and then selectively and obliviously overwrite pid_b in *PathData* if the posting block b is in a promising window that is needed, such that when we do not need to pay the step 1 cost mentioned above. Here *PathData* means a path ID record as a tuple of 2 integers denoting the posting block ID and its path ID. We can have a separate ORAM keeping these *PathBlock*s, since it is acceptable to let the server distinguish between the access of *PostingBlock* and the access of *PathBlock*.

An example query processing with *PathBlock*. Fig. 5.4 illustrates an example with a two-term query. Fig. 5.4(a) shows the posting lists for these two terms, A and B. Each posting record contains a document ID and a ranking score. The first posting record for term A represents document ID 1 and ranking score 15. In addition, two posting lists are divided into posting blocks, which are separated by "-". We can observe that both terms have three posting blocks. Fig. 5.4(b) presents the query processing in three steps from left to right. The first step is loading *MetaBlock* containing document ID boundary and block max score for each posting block. After generating windows, we choose three posting blocks that are in the promising window: the first block and the second block in term A, and the first block in term B. After loading *PathBlock*, we fetched the path ID for these blocks, i.e. Path ID 1, Path ID 2, and Path ID 4. Lastly,

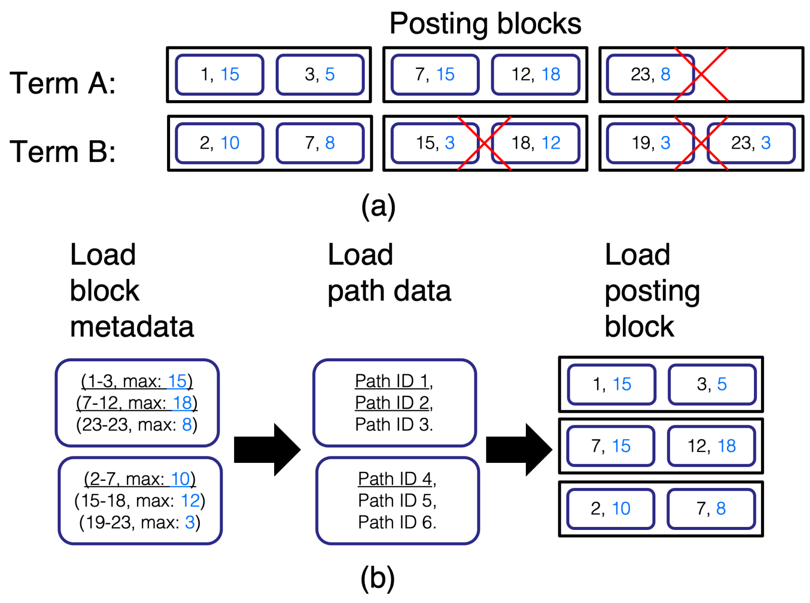


Figure 5.4: An example of ORAM data loading with path ID caching

three out of six $PostingBlocks$ are loaded for oblivious top-k computation, which saves the access cost for the skipped blocks.

5.6 Cost analysis

We discuss the cost model of our two-phase protocol without and with term-specific path caching. The cost is essentially measured by the number of ORAM blocks loaded for query processing. Assume a query has terms t_1, t_2, ..., t_m. The posting list length for t_i is pl_i, while the number of posting blocks for t_i is pb_i. Each ORAM block is of size B bytes. In addition, let Z be the number of posting blocks loaded after pruning. Let $Size_{metadata}$ be the number of bytes for one metadata tuple, and $Size_{pathdata}$ be the number of bytes for each path ID record.

- **Cost model without term-specific path caching.**

$$\sum_{i=1}^{m}(\lceil\frac{Size_{metadata} * pb_i}{B}\rceil * T_{MetaBlock} + Z * T_{PostingBlock}), \tag{5.1}$$

  where $T_{MetaBlock}$ is the access time for the ORAM storing $MetaBlock$, and $T_{PostingBlock}$ is the access time for the ORAM storing $PostingBlock$.

- **Cost model with term-specific path caching.**

$$\sum_{i=1}^{m}(\lceil\frac{Size_{metadata} * pb_i}{B}\rceil * T_{MetaBlock} + \lceil\frac{Size_{pathdata} * pb_i}{B}\rceil * T_{PathBlock} + Z * T'_{PostingBlock}), \tag{5.2}$$

  where $T_{PathBlock}$ is the access time for the ORAM storing $PathBlock$, and $T'_{PostingBlock}$ is the access time for the ORAM storing $PostingBlock$ without the path ID lookup time.

The first thing we can observe is that, compared to $\sum_{i=1}^{m}\lceil\frac{Size_{metadata}*pb_i}{B}\rceil * T_{MetaBlock}$, both $Z * T_{PostingBlock}$ and $Z * T'_{PostingBlock}$ should be more dominant in the cost models. It is because 1) $T_{PostingBlock}$ and $T'_{PostingBlock}$ can be much higher than $T_{MetaBlock}$ given the size of the ORAM storing $PostingBlock$ is much larger than the size of the ORAM storing $MetaBlock$; 2) Z is around $0.3 * \sum_{i=1}^{m} pb_i$ in our experiments for two-term queries, while $\frac{Size_{metadata}}{B}$ is much smaller, e.g. $Size_{metadata}$ is 16 and $B = 1024$. The second thing is that, if

$$\sum_{i=1}^{m}\lceil\frac{Size_{pathdata} * pb_i}{B}\rceil * T_{PathBlock} < Z * (T_{PostingBlock} - T'_{PostingBlock}), \tag{5.3}$$

then the second cost model has efficiency improvement over the first one. In our experiments, $(T_{PostingBlock} - T'_{PostingBlock})$ is about 2 ms, $T_{PathBlock}$ is about 3ms without path

ID lookup, and Z is around $0.3 * \sum_{i=1}^{m} pb_i$. Thus there should be significant efficiency gain through using term-specific path caching.

5.7 Privacy Analysis

5.7.1 Obliviousness of two-phase top-k retrieval

Recall that we define **memory access pattern and obliviousness** in Section 4.6. As both Algorithm 3) and Algorithm 4 are deterministic, below we show that these two algorithms present identical memory access patterns over some queries.

Lemma 5.7.1 *Replacing an input with any input that has the same number of block metadata tuples and has the same number of query terms, the memory access pattern of executing Algoirthm 3 does not change.*

Proof: Replacing an input with any input that has the same number of block metadata tuples, the length of $\mathbf{L}$ in Algorithm 3 is the same. Replacing an input with any input that has the same number of query terms, the length of $\mathbf{W}$ in Algorithm 3 is the same. When both $\mathbf{L}$ and $\mathbf{W}$ have their sizes unchanged, both the memory access pattern for the oblivious sorting before the *for* loop and the number of iterations in the *for* loop do not change. For each iteration, the proof of obliviousness follows the observations below:

1. For $\mathbf{W}'$ and $arrayBlockID$, through padding an empty block (or block ID resp.), we let them always be of size q which is the same as that of $\mathbf{W}$. $sumS$ can be calculated by scanning through $\mathbf{W}'$ and obliviously adding $maxS$ for those non-empty blocks.
2. To remove a block metadata, obliviously set the target metadata to empty values, apply oblivious sorting on all metadatas and remove the empty one which is always

in the same position after sorting. Adding block is deterministic through appending.

3. The returned $Windows$ are updated in a deterministic way.

■

Lemma 5.7.2 *With the same value of k, when replacing an input with any input that has the same number of posting records, the memory access pattern of executing Algoirthm 4 does not change.*

Proof: Replacing an input with any input that has the same number of posting records, the length of **L** in Algorithm 4 is the same. Thus, both the memory access pattern for the oblivious sorting before the *for* loop and the number of iterations in the *for* loop do not change. For each iteration, *equalBit* can be computed through oblivious selection, and therefore, both **s** and **s**′ can be computed in an oblivious way. At the end, with the same length of **L**, the oblivious sorting on **L** keeps the same memory access pattern, in addition to the deterministic way of returning top-k postings. ■

Following the above results, we can have the following theorem on the obliviousness of our proposed two-phase top-k retrieval with path caching.

Theorem 5.7.3 *Replacing any queried term with any term that has the same number of posting blocks, the memory access pattern of two-phase top-k retrieval does not change.*

Proof: The proof follows the observations below:

1. With the same number of posting blocks, the number of ORAM accesses for $MetaBlock$ does not change.

2. By Lemma 5.7.1, replacing any queried term with any term that has the same number of posting blocks, the total number of block metadata does not change. Thus, the memory access pattern of generating windows (i.e. Algorithm 3) does not change.

3. With the same number of posting blocks, the number of path IDs does not change. Therefore, the number of ORAM accesses for $PathBlock$ is still the same.

4. The selection of path IDs can be done obliviously.

5. By Lemma 5.7.2, replacing any queried term with any term that has the same number of posting blocks, the total number of posting records (including those empty ones used to filling the ORAM block) does not change. Thus, the memory access pattern of probing and the final pass (i.e. Algorithm 4) does not change.

■

As shown in Theorem 5.7.3, a query q_A can have the same memory access pattern as another query q_B if 1) they have the same number of query terms; 2) there is a one-to-one mapping between terms in q_A and terms in q_B such that two mapped terms have the same number of posting blocks. One observation is that, although this size information can be protected through padding, increasing posting block sizes does increase the number of queries that have the same memory access pattern in our proposed retrieval.

5.7.2 Leakage profile

Although we can assume a trusted execution environment has guarantees on data confidentiality and integrity, and our ORAM construction protects the data access pattern, there are still information that can be observed by a server in our proposed retrieval algorithm: 1) The number of query terms. 2) The number of ORAM blocks being accessed during query processing. 3) The number of posting records loaded during query processing. 4) The number of windows being generated and probed. 5) The number of matched documents. 6) The ORAM block length for $MetaBlock$, $PostingBlock$, and $PathBlock$. 7) The size of encrypted and decrypted term IDs, document IDs, additive ranking scores, ORAM block IDs and ORAM path IDs.

For the above leaked information, there is no known privacy attacks on 1), 6) and 7). From 2) and 3), it is unlikely that the server knows the exact length of posting lists. Padding strategy can help obfuscate the leakage in 4) and 5).

5.8 Evaluations

Datasets. All algorithms are evaluated on two datasets. One is ClueWeb 2009 TREC category B with 33.6 million documents after filtering with Waterloo spam score [55] threshold 60. Because the query processing time can be prohibitively slow when a single machine hosts the entire ClueWeb dataset, we choose to evenly split this dataset into ten partitions and let one single machine host one partition which is 3.36 million documents. The other is MS MARCO document collection with 3.20 million documents. The header and body text for all documents in two datasets are extracted using Indri [56]. All words are lowercased and stemmed using the Krovetz stemmer [57]. We randomly select 2500 queries from TREC 2006 Efficiency Track topics containing two to six terms. For all queries, each query term has more than 100 posting records.

Testing environment. We evaluated the performance of disjunctive queries on our proposed Two-Phase oblivious top-k retrieval and other baselines. Our implementations for PathORAM and retrieval algorithms including baselines are built with the C/C++ library supported by the SGX programming environment to make fair comparisons. All the experiments were conducted on a single machine with Intel Core i5-7260U 2.20GHz, running Ubuntu 18.04 with Intel SGX Linux 2.7 SDK. The indexes are encrypted and kept as ORAM blocks in the memory before query processing. All query processing times are reported in seconds in one single machine except specified otherwise. Before timing the queries, there is no index data being cached.

ORAM configuration. We firstly show three PathORAMs in our algorithm constructed with the configuration as following: 1) The one hosting $PostingBlock$ has a tree structure with height 21 and block capacity 2 million. Each block is of size 8 KB to store at most 32 posting blocks that has 32 posting records, i.e. packing size p is 32, where each posting record has two 4-byte integers representing document ID and additive feature score. The average block access time is 16.983 ms, and it becomes 15.118 ms if no path ID lookup is needed. 2) The one hosting $PathBlock$s has a tree structure with height 20 and block capacity 2 million. Each block is of size 1 KB that can store the path ID information, i.e. path ID (4-byte) and block ID (4-byte), for at most 128 posting blocks. The average block access time is 5.057 ms, and it becomes 3.192 ms if no path ID lookup is needed. 3) The one hosting $MetaBlock$ has a tree structure with height 20 and block capacity 1 million. Each block is of size 2 KB to store block metadata (four 4-byte integers) for at most 128 posting blocks. The average block access time is 6.088 ms, and it becomes 4.612 ms if no path ID lookup is needed.

The POSUP scheme proposed in [29] for document retrieval will be a baseline for us to compare as it conducts oblivious top-k computation discussed in Section 5.5.5. In $POSUP$, there is one PathORAM constructed to store all postings. Its height is 21 and block capacity is 2 million. Each ORAM block size is 8 KB to store at most 1024 posting records. The average block access time is 16.983 ms without using the pointer-based path optimization, and it becomes 15.118 ms if no path ID lookup is needed by using the pointer-based optimization.

For all the ORAMs in our experiment, the stash size is 80 to achieve negligible overflow probability that results in failed accesses [71]. We let the number of blocks in each ORAM bucket be 4 (which is referred as Z in the literature).

A comparison of query processing time. Table 5.1 lists the mean query times in seconds when k is 5 and 10 for different algorithms with query lengths varying from 2 to 6. Our proposed oblivious top-k retrieval is denoted as TP^o, and is denoted as TP^{BMW}

Table 5.1: Mean query latency and tail latency in second

Q. Len.	2	3	4	5	6
			$k = 10$, ClueWeb09		
$POSUP$	3.97 (15.1)	10.8 (40.7)	17.1 (47.9)	24.9 (60.1)	29.2 (65.7)
TP^o	2.35 (7.49)	7.64 (24.1)	14.3 (38.7)	21.7 (51.7)	27.2 (63.3)
TP^{BMW}	2.30 (7.34)	7.47 (23.5)	14.0 (37.6)	21.2 (50.3)	26.5 (61.7)
			$k = 5$, ClueWeb09		
$POSUP$	3.96 (15.1)	10.8 (40.6)	17.1 (47.8)	24.9 (60.0)	29.1 (65.6)
TP^o	2.02 (5.63)	6.81 (19.9)	13.3 (37.6)	20.4 (48.0)	25.5 (59.9)
TP^{BMW}	1.98 (5.52)	6.63 (19.5)	13.0 (36.6)	19.9 (46.8)	25.0 (58.4)
			$k = 10$, MS MARCO		
$POSUP$	3.26 (11.4)	7.45 (17.8)	11.7 (26.5)	16.8 (36.7)	21.8 (41.3)
TP^o	2.29 (6.36)	6.00 (14.4)	10.3 (21.5)	15.3 (32.6)	20.4 (40.5)
TP^{BMW}	2.24 (6.22)	5.86 (14.0)	10.1 (21.0)	14.9 (31.8)	19.9 (39.4)
			$k = 5$, MS MARCO		
$POSUP$	3.25 (11.3)	7.44 (17.8)	11.7 (26.4)	16.8 (36.6)	21.8 (41.2)
TP^o	1.97 (5.45)	5.35 (13.4)	9.43 (19.9)	14.1 (31.4)	19.0 (40.5)
TP^{BMW}	1.93 (5.34)	5.24 (13.1)	9.22 (19.4)	13.7 (30.6)	18.5 (39.4)

if there is no oblivious top-k computation. TP^{BMW} can be considered when the privacy requirement is different. The latency reported for $POSUP$ has used pointer-based path optimization. This table also lists the 95th percentile *tail latency* for each query length within parentheses, corresponding to the response time occurring in the 95th percentile, as defined in [36].

For both ClueWeb09 and MS MARCO, TP^o outperforms $POSUP$ for all query lengths with both tested k values. In particular, for short queries (i.e. 2 terms and 3 terms), TP^o shows more efficiency improvement compared to that for long queries (i.e. 4, 5, and 6 terms). For mean time and 2 terms in ClueWeb09, TP^o is 1.68x and 1.96x faster than $POSUP$ when $k = 10$ and $k = 5$, respectively. For mean time and 6 terms in ClueWeb09, TP^o is 1.07x and 1.14x faster than $POSUP$ when $k = 10$ and $k = 5$, respectively. For tail latency and 2 terms in ClueWeb09, TP^o gains a little more efficiency improvement: it is 2.02x and 2.68x faster than $POSUP$. As for tail latency and 6

Table 5.2: The impact of dynamic pruning on mean query latency (s) in TP^o

Q. Len.	2	3	4	5	6
$k = 10$, Posting Block Size = 32, $p = 32$					
W/O	3.94 (-)	10.8 (-)	17.1 (-)	24.8 (-)	29.2 (-)
W	2.35 (-40%)	7.64 (-29%)	14.3 (-16%)	21.7 (-12%)	27.2 (-7%)
$k = 5$, Posting Block Size = 32, $p = 32$					
W/O	3.93 (-)	10.7 (-)	17.0 (-)	24.8 (-)	29.1 (-)
W	2.02 (-49%)	6.81 (-37%)	13.3 (-22%)	20.4 (-18%)	25.5 (-12%)
$k = 5$, Posting Block Size = 64, $p = 16$					
W/O	3.95 (-)	10.8 (-)	17.1 (-)	24.8 (-)	29.1 (-)
W	2.32 (-41%)	7.63 (-29%)	14.5 (-15%)	21.9 (-11%)	27.0 (-7%)

Table 5.3: Mean query latency (s) and reduction rate of TP^o for different packing sizes

Q. Len.	2	3	4	5	6
$k = 5$, Posting Block Size = 32					
$p = 1$	6.26 (-)	24.1 (-)	53.7 (-)	88.9 (-)	114 (-)
$p = 4$	2.60 (-59%)	9.42 (-61%)	20.0 (-63%)	32.1 (-64%)	41.2 (-64%)
$p = 32$	2.02 (-68%)	6.81 (-72%)	13.3 (-75%)	20.4 (-77%)	25.5 (-78%)
$k = 5$, Posting Block Size = 64					
$p = 1$	4.19 (-)	15.6 (-)	33.0 (-)	52.5 (-)	68.1 (-)
$p = 2$	3.14 (-25%)	11.3 (-28%)	23.4 (-29%)	36.7 (-30%)	47.1 (-30%)

terms in ClueWeb09, however, TP^o shows only a marginal improvement which is 1.04x and 1.10x faster than $POSUP$. This trend aligns with the result in [84]: as the query length becomes larger or the k value becomes larger, the effectiveness of window probing can decrease. Thus the efficiency advantage of TP^o becomes marginal.

The impact of dynamic pruning. Table 5.2 shows the impact of dynamic pruning on mean query latency in our proposed two-phase oblivious retrieval for ClueWeb09. We compare mean query latency with (denoted as W in the table) and without (denoted as W/O in the table) pruning in three different settings: 1) $k = 10$, posting block size is 32, and packing 32 posting blocks into a single ORAM block. 2) $k = 5$, posting block size is 32, and packing 32 posting blocks into a single ORAM block. 3) $k = 5$, posting block size is 64, and packing 16 posting blocks into a single ORAM block, such that

Table 5.4: ORAM block read time with/without path caching in milliseconds

# of ORAM Blocks	512K	1M	2 M
ORAM Block Size = 512 B			
Without path caching	3.270	3.610	4.112
With path caching	2.020	2.133	2.247
ORAM Block Size = 1 KB			
Without path caching	4.132	4.513	5.057
With path caching	2.882	3.037	3.192
ORAM Block Size = 2 KB			
Without path caching	5.646	6.088	6.693
With path caching	4.396	4.612	4.827

Table 5.5: The impact of term-specific path-ID caching on mean query latency (s) for TP^o

Q. Len.	2	3	4	5	6
$k = 5$, Posting Block Size = 32, $p = 4$					
W/O	4.36 (-)	16.0 (-)	34.2 (-)	54.9 (-)	70.4 (-)
W	2.60 (-40%)	9.42 (-41%)	20.0 (-42%)	32.1 (-42%)	41.2 (-41%)
$k = 5$, Posting Block Size = 32, $p = 32$					
W/O	2.26 (-)	7.60 (-)	14.8 (-)	22.8 (-)	28.5 (-)
W	2.02 (-10%)	6.81 (-10%)	13.3 (-11%)	20.4 (-11%)	25.5 (-11%)
$k = 5$, Posting Block Size = 64, $p = 2$					
W/O	5.27 (-)	19.2 (-)	39.8 (-)	62.6 (-)	80.4 (-)
W	3.14 (-40%)	11.3 (-41%)	23.4 (-41%)	36.7 (-41%)	47.1 (-41%)

this setting has the same ORAM block size as the above two settings. For two-term queries, dynamic pruning can reduce the mean latency up to 49%, and for longer queries, it becomes less effective, e.g. only 7% for six-term queries. In addition, dynamic pruning is more effective when posting block size is smaller because the estimated window max score can be less accurate for large posting block sizes. Another observation is that, without dynamic pruning, our algorithm achieves the same level of performance as that of $POSUP$ shown in Table 5.1, indicating that dynamic pruning is the main reason of our algorithm outperforming $POSUP$.

The impact of packing posting blocks. In Table 5.3, the impact of packing posting blocks on mean query latency is presented for two different settings with using

ClueWeb09: 1) $k = 5$, posting block size is 32; 2) $k = 5$, posting block size is 64. For both settings, with increasing the number of posting blocks in a single ORAM block, mean query latency can be optimized significantly. It is because ORAM overhead makes it is inefficient to access small chunks of data in multiple times, compared with accessing a large chunk of data once. We can also observe that, for long queries, packing posting blocks can bring more time cost saving. This is due to that fact that, our algorithm tends to access more posting blocks for long queries since the pruning rate for long queries is not as low as that for short queries. In particular, the cost reduction brought by dynamic pruning can be less effective when packing too many posting blocks.

The impact of term-specific path caching. To assess our PathORAM implementation, Table 5.4 reports the average time in milliseconds for an ORAM block read operation studied through a micro benchmark when varying the block size and the number of block in a tested ORAM. The result indicates the block read time including the block lookup is reduced significantly with path ID caching. With path ID caching, the ORAM read time is much smaller without need of path-ID lookup. It also shows that the cost of path-ID lookup is only related to the number of ORAM blocks, and the ratio of path-ID lookup time cost over the total time cost becomes larger when the ORAM block size is small. For example, when ORAM block size is 512 bytes and there are 1 million ORAM blocks, the time cost of path-ID lookup is around 41% of the total access time. When ORAM block size is 1KB and there are 1 million ORAM blocks, the ratio becomes 33%. This trend is consistent with Table 5.5 that shows the impact of term-specific path ID caching on mean query latency in three different settings for ClueWeb09. Note that this path ID caching technique is for saving the cost of path-ID lookup. We can observe that when packing 4 posting blocks, the effectiveness of path ID caching is larger than that when packing 32 posting blocks, due to the fact that the total number of ORAM blocks becomes smaller and the ORAM block size becomes larger with packing more posting blocks.

Compared with top-k retrieval in no-privacy setting. We compare our algorithms with top-k retrieval algorithms in no-privacy setting to show the cost of executing an oblivious top-k retrieval algorithm against known attacks on the access pattern. Block-Max WAND (BMW) is a well-known top-k retrieval algorithm. In our evaluation, for one partition with 3.36 million documents, the query processing time is 1.34 ms, 4.36 ms, 7.16 ms, 10.4 ms and 12.7 ms for two to six terms when k value is 10. When k is 5, the query processing time is 1.14 ms, 4.39 ms, 7.14 ms, 9.54 ms and 12.1 ms. Our ORAM-based oblivious algorithm is about three orders of magnitude slower than BMW in no-privacy setting, and thus providing privacy protection with an ORAM-based solution does incur a significant cost, which also shows the importance of efficiency optimization for ORAM-based document retrieval. Chapter 4 exploits protection protection in a TEE without ORAM and our MII solution takes about 0.612 seconds to retrieve ranked results from a ClueWeb partition that has 1 million documents. Thus MII is one order of magnitude faster than our ORAM-based solution for ClueWeb09, which represents a tradeoff between privacy and time efficiency.

5.9 Summary

This chapter proposes and evaluates an oblivious top-k retrieval algorithm that leverages window navigation and probing to reduce the number of ORAM blocks being accessed. Our solution also includes two proposed techniques, posting block packing and term-specific path caching, in order to optimize the time cost of accessing ORAM blocks for posting lists in a skipping manner. The evaluation shows our algorithm can achieve faster query processing latency for all query lengths. In particular, it shows a significant efficiency advantage for short queries, especially in tail latency reduction.

Chapter 6

Privacy-aware Document Ranking with Neural Signals

6.1 Introduction

It is an open problem to develop privacy-aware ranking leveraging neural models. The previous research on neural ranking model falls into the following two categories: interaction-based or representation-based models [86]. The earlier work has focused on the representation-based models [87, 88] where each document and a query are separately represented as vectors through neural computation and the final ranking is based on the similarity of the two representative vectors. The recent studies have focused on interaction-based neural ranking models [5, 6, 7] where the word or term-level similarity of a query and a document is explored first based on their embedding vectors before applying additional neural computation. These studies have shown their interaction-based models outperform the earlier representation-based models and thus this chapter addresses privacy issues for three interaction-based models, more specifically DRMM [5], KNRM [6], and CONV-KNRM [7].

Recent secure neural network research [89] addresses image classification using homomorphic encryption and secure two-party computation with garbled circuits, to meet a different privacy requirement (where clients obtain predicted results without knowing the server decision model). The online processing time can cost 3.56 seconds for classifying each image and in addition, 296MB of data needs to be communicated between a client and a server for each image. While computed scores are still un-comparable at the server side, the cost is too expensive for ranking many documents, considering each document vector as an image vector. Order-preserving encryption techniques (e.g. [90, 91]) let a server compare the encrypted results but do not support arithmetic computation on encrypted numbers. There is a line of work perturbing feature values (e.g. [92]) for classification to achieve differential privacy. Our method is aimed at document search with a goal of preserving the exact ranking model and our design uses one round of client-server communication for faster response time.

Problem Statement. This chapter investigates how privacy consideration can be incorporated efficiently in neural ranking for top-k cloud data search. To our best knowledge, the proposed method in this chapter is the first effort to address privacy-aware interaction-based neural ranking. In specific, we identify statistical document information such as word frequency and occurrence that can be leaked during neural computation. Such information is required for a number of privacy attacks studied in the previous work [11, 12, 27]. To mitigate such a leakage, our techniques replace the exact kernel value with a privacy-aware tree ensemble model [93, 94, 95, 26]. We further propose a soft match map that captures non-exact similarity signals above a threshold while providing a privacy protection using term closures and kernel value obfuscation. Our evaluation using two TREC datasets shows the relevance of the proposed tree integration can even exceed the original baselines for NDCG scores when soft match maps are not used. There is a relevance trade-off when incorporating soft match maps.

6.2 Background and Problem Settings

Learning-to-rank algorithms. Given a matched document represented by a set of raw and/or composite features, a linear ranking model uses a linear combination of document features while a tree-based ensemble produces a set of decision trees using a boosting or bagging strategy [93, 94, 95].

An interaction-based neural ranking [5, 6, 7] can be formalized as performing the following computation flow:

$$\text{RankingScore} = NN(Ker(\vec{q} \otimes \vec{d})),$$

where $\vec{q}$ and $\vec{d}$ are two sequences of embedding vectors which can be representations for a unigram or a n-gram in a query and a document [5, 6, 7], or can be entity embeddings for existing entities in a query and a document [8]. Those embedding vectors can be learned from one of many existing neural network models such as Word2Vec [96], GloVe [97] and relevance based word embedding [98]. Embedding vectors for n-grams can be generated by a convolution operation described in [7]. NN is a forward neural network to compute the final ranking score.

Operator $\otimes$ is the interaction between query q and document d and its output is the similarity of a query term and a document term for all possible pairs from q and d. In [5, 6, 7], cosine similarity is used for measuring term similarity with a score varying from -1 to 1. Let $\langle t, w \rangle$ denote the cosine similarity between the term vector of t and that of another term w.

Operator Ker represents the kernel value calculation, extracts term-level matching signals based on the similarity of all term pairs from a query and a document, and generates a vector of real values being taken as input for the forward neural computation. There are two methods for kernel computation. In the *Histogram Pooling* [5] method, there are R kernels and each kernel associates with an interval within $[-1, 1]$, e.g., $[0.5, 0.6)$. The kernel value of the j-th kernel is the number of similarity values that fall into the j-th interval $[b_j, b_{j+1}]$: $K_j(t,d) = \sum_{w \in d} \mathbb{1}_{b_j \leq \langle t,w \rangle < b_{j+1}}$. For kernel pooling, we follow a definition in [6, 7] that each kernel associates with a Radial Basis Function (RBF). The RBF kernel for the j-th kernel is defined by μ_j and σ_j. Symbols μ_j and σ_j denote the mean and standard deviation respectively. Let exp be the natural exponential function and log be the natural logarithm function. The kernel value of the j-th kernel for a query term t is:

$$K_j(t,d) = \sum_{w \in d} \exp(-\frac{(\langle t,w \rangle - \mu_j)^2}{2\sigma_j^2}).$$

The output of kernel computation is a kernel vector of size R:

$$(\sum_{t \in q} \log K_1(t,d), \cdots, \sum_{t \in q} \log K_R(t,d))^T.$$

Privacy requirement and threat model. A client owns all data and wants to outsource the search service to a cloud server which is honest-but-curious, i.e., the server will honestly follow the client's protocol, but will also try to learn any private information from the client data. The client builds an encrypted but searchable index and lets a server host such index. This chapter does not consider the dynamic addition of new documents to the existing index, assuming the client can periodically overwrite the index in a cloud host server to include new content. To conduct a search query, the client sends several encrypted keywords and related information to the server. Our design only uses one round of client-server communication since multi-round active communication between the server and client (e.g. [65, 66]) incurs a much higher communication cost and response latency.

The biggest threat is the leakage of query and document plaintext. A server can also be interested in query access pattern and statistical information even if the query terms are encrypted. Finally the result patterns such as the overlapping of document IDs in multiple queries may also be interesting. This chapter is focused on providing a privacy protection to avoid the leakage of document plaintext and also important feature values during ranking process.

Since all of the above attacks require term occurrence and use term frequency if possible, this chapter will analyze the information leakage of interaction-based neural ranking methods on term frequency and occurrence in documents, and extend or redesign some of their components with a goal of hiding such statistical text information. It is worthy to note that some advanced techniques, e.g. ranking based on Convolutional Neural Network [9], require term positions and such information can be leaked during computation. Then term occurrences in a document can be easily inferred by a server. Thus this chapter does not investigate such ranking models.

6.3 Leakage Analysis and Design Considerations

We first examine the possible leakage of information in interaction-based neural ranking in terms of term occurrence and frequency.

Hiding term vectors. As mentioned above, there are three steps in the interaction-based model: interaction between query and document terms, kernel value calculation, and forward neural network computation. Our first thought is to hide term vectors using a hashing function while preserving cosine similarities (or other similarity metrics) between vectors. Such a protection can mask identities of term vectors; however, if a server is allowed to observe the result of interaction between a query term t and each term w of document d, it can easily infer frequencies and occurrences of all query terms as follows:

$$TF(t,d) = \sum_{w \in d} \mathbb{1}_{\langle t,w \rangle = 1},$$

where $\mathbb{1}_{\langle t,w \rangle=1}$ equals to 1 if $\langle t,w \rangle = 1$ and 0 otherwise.

Given the fact that we cannot store masked term vectors or explicitly store the interaction matrix elements, we resort the following strategy where the result of interaction between query and document terms is not computed by the server.

Kernel-level protection. Our next design idea is to provide a kernel-level protection of privacy by precomputing kernel values in advance. Thus only the output of Ker is exposed to the server, which hides the vector computation process and the result of interaction. Namely the owner of the dataset (as the client in our case) precomputes this kernel vector first for each term t that may interact with a document d, $\vec{f_{t,d}} = (a_1, , \cdots, a_j, \cdots, a_R)^T$ where $a_j = \log K_j(t,d)$. These vectors, after precomputed by a client, are uploaded and stored in a server. During the run time with a query q, then the kernel vector for this query can be constructed as $\sum_{t \in q} \vec{f_{t,d}}$. Then such a kernel vector is injected as an input to the forward neural computation step.

Leakage from kernel vectors. Unfortunately as we analyze below, the above kernel-level protection can still leak term frequency, which yields the leakage of term occurrence.

Notice for both histogram pooling and kernel pooling, the last kernel $K_R(t,d)$ is a special kernel representing the exact match of a query term with document terms. Without loss of generality, let this special one be the R-th kernel in this chapter. For histogram pooling, an interval defined as $[1,1]$ is associated with this special kernel. This means that last element a_R in $\vec{f_{t,d}}$ gets updated by one whenever a query term t exactly matches a document term in d. As a result, the server can infer the term frequency of a query term in any document by observing the result of a_R.

Proposition 6.3.1 *Given query term t and document d, in histogram pooling for the*

R-th kernel whose interval is $[1, 1]$*, term frequency* $TF(t, d) = \sum_{w \in d} \mathbb{1}_{\langle t,w \rangle = 1} = \exp(a_R)$.

For the R-th kernel derived with kernel pooling [6, 7], μ_R is 1.0, and σ_R is chosen to be a small positive real number, e.g., 0.001. We define the maximum cosine similarity between any two different terms in a vocabulary V where V is the collection of all terms in the given dataset:

$$\bar{S} = \max_{t,w \in V, t \neq w} \langle t, w \rangle .$$

The following theorem shows a server can still approximate term frequency $TF(t, d)$ using the value of last kernel a_R.

Theorem 6.3.2 *Given a query term t and a document d, in kernel pooling for the R-th kernel, if $\bar{S} < 1.0 - \sqrt{2\sigma_R^2 \ln \frac{n}{\epsilon}}$, then $|TF(t, d) - \exp(a_R)| < \epsilon$, where ϵ is a small real value.*

Proof of Theorem 6.3.2

Let tf denote TF(t, d) for simplicity here.

$$\begin{aligned}
&|\text{tf} - \exp(a_R)| = |\text{tf} - \exp(\log K_R(t, d))| = |\text{tf} - K_R(t, d)| \\
&= \left| \text{tf} - \sum_{w \in d} \exp(-\frac{(\langle t, w \rangle - \mu_R)^2}{2\sigma_R^2}) \right| \\
&= \left| \text{tf} - \sum_{w \in d, w = t} \exp(-\frac{(\langle t, w \rangle - 1.0)^2}{2\sigma_R^2}) - \sum_{w \in d, w \neq t} \exp(-\frac{(\langle t, w \rangle - 1.0)^2}{2\sigma_R^2}) \right| .
\end{aligned}$$

Since if $w = t$, then $\langle t, w \rangle = 1.0$, $\exp(-\frac{(\langle t,w \rangle - 1.0)^2}{2\sigma_R^2}) = \exp(0) = 1$. Thus if the length of d

is n, we have

$$\begin{aligned}
&\left|\mathrm{tf} - \sum_{w\in d, w=t} 1 - \sum_{w\in d, w\neq t} \exp(-\frac{(\langle t, w\rangle - 1.0)^2}{2\sigma_R^2})\right| \\
&= \left|\sum_{w\in d, w\neq t} \exp(-\frac{(\langle t, w\rangle - 1.0)^2}{2\sigma_R^2})\right| \leq (n - \mathrm{tf}) \exp(-\frac{(\bar{S} - 1.0)^2}{2\sigma_R^2}) \\
&\leq n \exp(-\frac{(\bar{S} - 1.0)^2}{2\sigma_R^2}) < \epsilon.
\end{aligned}$$

In our tested datasets in Section 6.6, $\bar{S}$ is below 0.9. Using the above theorem, condition $\bar{S} \leq 0.947 < 1.0 - \sqrt{2\sigma_R^2 \ln \frac{n}{\epsilon}}$ is true when $\sigma_R \leq 0.01$, $\epsilon = 0.01$, and $n \leq 10,000$, and a server can easily infer the frequency of a term in a document. Thus we are unable to use a_R and the next section will present a solution to address this.

It should be noted that the above analysis is true for computing the interaction between a unigram query term with all unigrams of a document in DRMM [5] and KNRM [6]. When computing the interaction of a h-gram from a query and a g-gram from a document where $h \neq g$ in CONV-KNRM [7], the cosine similarity of such a pair cannot be 1. Thus for CONV-KNRM, we only need to worry about the interaction of two terms with the same gram length.

6.4 Privacy-aware Neural Ranking

In this section, we propose three techniques for privacy-aware neural ranking: 1) replace the exact match kernel value with the ranking score from a traditional ranking method that leverages exact word matching; 2) provide a soft match index (which we call soft match map) to include a set of kernel values with similar terms while enhancing privacy; 3) obfuscate kernel values in the soft match map.

6.4.1 Replacement of the Exact Match Kernel

Neural signals can be considered to be composed of two parts: exact match component represented by last kernel value $K_R(t,d)$ and the soft match component represented by $K_1(t,d), \cdots, K_{R-1}(t,d)$ as shown in Fig. 6.1(a). Since Theorem 6.3.2 indicates that the root cause of term frequency leakage is the inclusion of kernel value K_R, we propose to drop this kernel value and compensate it by the including of a traditional ranking method that has better privacy protection, as illustrated in Figure 6.1(b).

We adopt a privacy-aware learning-to-rank tree ensemble model in [26] that encodes raw features with comparison preserving mapping (CPM) and derives a tree ensemble using encoded raw features. Raw ranking features mainly based on exact term matching are not leaked to the server. During our evaluation, these exact text matching features include BM25 for query words that appear in the title, BM25 for query words that appear in the body, and the proximity features [99, 100, 101, 102] with the minimum, maximum, and average of the squared min distance reciprocal of query word pairs in the title or in the body. We use word-pair or n-gram based features as a substitute to avoid composite proximity features based on word positions through arithmetic calculation. For ClueWeb09, extra raw features include PageRank and a binary flag indicating whether a document is from Wikipedia.

The above replacement is applied for computing the unigram-to-unigram interaction in DRMM and KNRM. It is also applicable to the interaction of a h-gram query term with a h-gram document term for CONV-KNRM. We discuss more on this in Section 4.8. This replacement comes with two advantages. First, it removes the source of term frequency leakage in a_R since an adversary is not able to recover feature values encoded with CPM [26]. Second, it can potentially boost ranking performance. Currently there is no known method to combine a traditional ranking method with a neural ranking model for

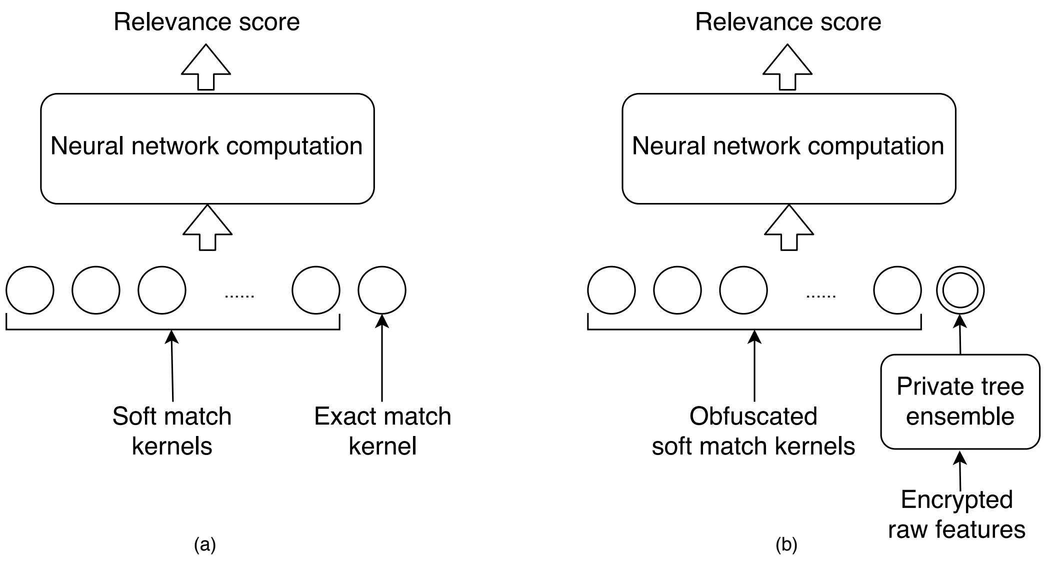


Figure 6.1: Replacement of the exact match kernel

a better relevance. A tree ensemble method has been proven to be effective before neural models gain more attention. For example, in the Yahoo! learning-to-rank challenge [103] in 2010, all winners have used some forms of tree ensembles. This replacement can provide a natural way to effectively combine a tree ensemble with a representation based neural model and we will evaluate the relevance impact in Section 4.8.

6.4.2 Obfuscation of Kernel Values

Even though we have precomputed the kernel value computation to avoid the leakage of term frequency to the server, there still exists a term frequency attack described below when the histogram pooling is used. In this attack, we describe how to recover the term frequency from closed soft match maps based on histogram pooling even though the exact match signals are removed. We assume that the interval $[-1, 1)$ for similarity value is divided as $[-1, b_2, \cdots, b_{R-1}, 1)$ where $-1 = b_1 \leq b_2 < b_3 < \cdots < b_{R-1} < b_R = 1$. Each kernel K_j is associated with an interval $[b_j, b_{j+1})$ where $1 \leq j \leq R-1$. Note that all intervals of these kernels are disjoint and their unions are interval $[-1, 1)$. Given a term

t, a document d, and a kernel vector $\vec{f_{t,d}}$ in a soft match map is $(a_1, , \cdots, a_j, \cdots, a_{R-1})^T$ and a_R is not included. Each kernel value $a_j = \log K_j(t,d) = \log(\sum_{w \in d} \mathbb{1}_{b_j \le \langle t,w \rangle < b_{j+1}})$. Thus

$$\sum_{j=1}^{R-1} \exp(a_j) = \sum_{j=1}^{R-1} (\sum_{w \in d} \mathbb{1}_{b_j \le \langle t,w \rangle < b_{j+1}}).$$

Since the union of all the disjoint intervals is $[-1, 1)$, we can have

$$\sum_{j=1}^{R-1} \exp(a_j) = \sum_{w \in d} \mathbb{1}_{-1 \le \langle t,w \rangle < 1}.$$

Let the length of d be n, and term frequency of t contained in d is $\text{TF}(t,d)$, we have $\sum_{j=1}^{R} \exp(a_j) = n$ and $\exp(a_R) = \text{TF}(t,d)$. Then $\text{TF}(t,d) = n - \sum_{j=1}^{R-1} \exp(a_j)$.

We also notice for any term t' that is not in document d, $\text{TF}(t',d) = 0$. But $\vec{f_{t',d}}$ is included in a soft match map because of term closure. Thus $n = \sum_{j=1}^{R-1} \exp(a'_j)$.

To launch this attack, we assume an adversary can scan the soft match map SMM to obtain all keys (t,d) such that $(t,d) \in SMM$, and then take the following actions: 1) For each key (t,d), obtain the kernel vector $\vec{f_{t,d}} = (a_1, \cdots, a_{R-1})$ in a soft match map. Compute the sum of elements in this vector. $S_t = \sum_{j=1}^{R-1} \exp(a_j)$, where $a_j = \log K_j(t,d)$. 2) Figure out the length of this document as $n = max_{t:(t,d) \in SMM}\{S_t\}$. 3) Compute term frequency of any t in d as $\text{TF}(t,d) = n - S_t$.

In the above attack, the frequency of a term queried can be uncovered by a server if it is able to find all or many of encrypted keys (t,d) where t is a soft or exact term of document d in a soft match map. While we have not found a term frequency attack for the kernel pooling, we still want to be cautious.

To minimize the chance of leaking exact kernel values, we propose a many-to-one mapping to obfuscate kernel vector values. Intuitively, if kernel vector values from multiple different term document interactions are indistinguishable, the adversary has to make random guesses on real kernel vector values. In specific, we add the ceiling function to convert the floating point number to an integer in forming the kernel vector, and this change allows the revised soft match kernel values to accomplish k-anonymization [104, 105]. For the j-th element of a kernel vector based on R kernels,

$$a_j = \begin{cases} \lceil \log_r(K_j(t,d)) \rceil, & \text{if } K_j(t,d) > 1, \\ 1, & \text{otherwise,} \end{cases}$$

where the logarithmic base r is a privacy parameter that can be adjusted. As we show later in Section 6.5.2, the anonymous factor k is $r^{R-1} - 1$, since all zero kernel values are converted to 1. A large r value will add more anonymity for privacy while it may degrade the relevance performance due to the lack of value differentiation in kernel vectors.

In DRMM, at the forward neural computation stage, the kernel values are re-scaled by document frequency weights of query terms. We use the same many-to-one function discussed above to obfuscate these weights. Our evaluation shows there is no visible relevance difference when $r = 10$.

6.4.3 Soft Match Maps

Kernel vector values are precomputed before query processing and such offline processing can be done efficiently on a parallel platform and/or with LSH approximation [106]. However, it is too expensive to store kernel vectors for all possible (t, d) pairs. For example, suppose there are $R = 20$ kernel values per term-document pair, for a dataset with 2M documents and a vocabulary of 250K terms, all these kernel vectors require 20 terabytes of space if each kernel value is stored using a reduced precision with 2 bytes. Although data compression may optimize this cost, the optimized storage cost is still excessively high.

Inspired by the inverted index, we propose a soft match index structure that contains kernel vectors of term-document pair (t, d) only if term t is reasonably related to document d, above a similarity threshold. This index data structure, called *soft match map*, has a key-value representation. The key of each entry is a hashed term-document pair (t, d), and the value is kernel vector $\vec{f_{t,d}}$ generated from Ker for a term-document pair. Notice a_R for R-th kernel is removed as discussed in Section 6.4.1 for interaction between a query term and a document term under the same gram length. We call a term for a document this map as *exact* term if this term appears in this document. If this term is not in this document but it is included in the map due to its similarity to another term in this document, we call this term as *soft* term of this document. We will access how a soft match map is accessed during search in Section 6.5.

Even though terms and documents in a soft match are encrypted and identified through numeral IDs, an adversary may infer the the occurrence of terms in a document which is a critical piece of information for privacy attack discussed in Section 6.2. In order to minimize the chance of leaking term occurrence in a document, we introduce the notion of *τ-similar term closure.*

Definition 1 *A set of terms C under vocabulary V is called a τ-similar term closure if for any term $t \in C$ and there exists another term $w \in V$ such that $\langle t, w\rangle \geq \tau$, then $w \in C$.*

Here V is the collection of terms in a given dataset and we will discuss a clustering algorithm shortly that groups a set of terms as a term closure.

Definition 2 *A soft match map SMM is closed under term closures if for any $(t, d) \in SMM$, for any w in the same term closure of t, $(w, d) \in SMM$.*

There are two advantages of a closed soft match map. 1) From the relevance point view, a document that contains a word which is similar to a query word that can get some rank credit as the privacy-aware ranking only uses this soft match map to identify semantically related documents. 2) As shown in the next section, an adversary would have a hard time to detect if a word ID that appears in the document or not because other similar words in its term closure have all appeared in the soft match map. We will analyze its privacy implication based on the notion of statistical indistinguishability in the next section.

In the rest of the paper, we will assume a soft match map is closed. We now describe how to partition a term vocabulary of a dataset into a disjoint set of term closures. There are trade-off factors to consider in controlling the size of term closures. With a larger size, the soft match map will accommodate more similar terms that can improve relevance, while creating more challenges for an adversary to distinguish and detect which term IDs in a soft match map appear in a document. On the other hand, a larger size demands more storage to host a soft match map. In the following, we discuss two algorithms that derive a disjoint set of term closures.

Clustering with a fixed similarity threshold. Given a clustering threshold τ, we cluster all terms in a closure C using a transitive closure computation as follows: if $t \in C$, and $\langle t, w \rangle \geq \tau$, then $w \in C$. This approach uses a uniform threshold for all clusters. With a small clustering threshold value, some clusters can have a very big size, which can result in a very large storage demand. With a large threshold value, some of term closures have a very small size, not big enough for the privacy purpose.

Adaptive clustering with multiple thresholds and closure size control. Given p as the targeted closure minimum size, x as the maximum size, and m sorted clustering thresholds $\tau_1 > \tau_2 > \cdots > \tau_m$. We first apply a similarity clustering with a fixed threshold τ_1. If some of clusters exceed the maximum size, we split and remove them so that each removed cluster is of size smaller than or equal to x, and at least the targeted minimum size, p. For the remaining terms, we apply a similarity clustering with a fixed threshold

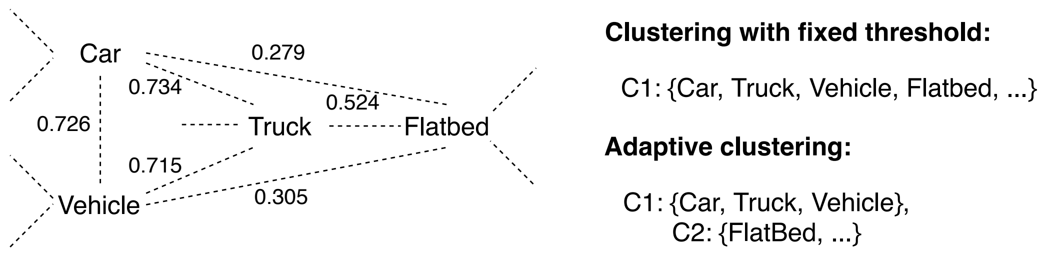

Car
Truck
Flatbed
Vehicle
0.279
0.734
0.524
0.726
0.715
0.305
Clustering with fixed threshold:
C1: {Car, Truck, Vehicle, Flatbed, ...}
Adaptive clustering:
C1: {Car, Truck, Vehicle},
C2: {FlatBed, ...}

Figure 6.2: Two clustering methods for term closure

τ_2. Repeat this process until all terms are removed into clusters, or we have applied clustering with all thresholds. Terms that are still left, are randomly organized into clusters of size smaller than or equal to x, and no less than p. This adaptive clustering provides a flexibility to group a sufficient number of similar terms in each closure while yielding a reduced storage demand. In our evaluation, $p = 5$ and $x = 30$ are used.

Figure 6.2 shows the difference of the above two clustering for a partial similarity graph of 4 terms from one of our testing datasets. The edges represent pairwise similarity scores. With fixed similarity threshold at 0.5, all four terms "Car", "Truck", "Vehicle", and "Flatbed" are clustered transitively into the same term closure. With adaptive clustering using a threshold set {0.9, 0.8, 0.7, 0.6, 0.5}, and closure target size 3, term "Flatbed" is not grouped with "Car", "Truck", and "Vehicle". This is because that when threshold 0.7 is used, terms "Car", "Truck", and "Vehicle" are grouped, reaching closure size 3 and they are removed to form a separate closure. "Flatbed" will then be grouped with other terms in the rest of the graph (which is not shown in this figure).

6.5 Leakage and Privacy of Soft Match Maps

6.5.1 Search Process and Leakage Profile

The top K search process is described as follows. A client first sends randomized tokens for query terms and related information (called trapdoor information [25]) to a server, and these terms include query unigrams and/or multi-grams when needed. The server cannot map tokens into query terms since tokens are randomized. After the server receives tokens for all the query terms, a privacy-aware document retrieval model based on [10, 25, 20, 107, 108] produces a set of document candidates for further ranking. The server first computes the keys to access the CPM-encoded features and run a tree ensemble, where the leakage profile is studied in [25]. Then the server computes the keys to access the soft match map, and fetches associated kernel vectors to drive the forward neural computation.

Depending on the query processing semantics and privacy requirement, there are two methods to pass the query term list and document list to neural ranking, and each of which has a different leakage profile. The first method is based on the private search work of [10, 25]. Based on trapdoor information sent from a client, the server can compute and obtain a key (t, d) to access the soft match map. Essentially the server obtains a list of keys representing (t, d) pairs where t is a query term ID and d is a candidate document ID, but the server cannot decompose each key into two parts to obtain its term ID. Finally the server returns a ranked list of encrypted document IDs and these IDs are available in the map values of keys. For this case, we list the leakage profile as follows.

- Initially the server does not know any key (t, d) for the map. After processing queries, the server gradually learns more keys of the soft match map.

- Once the server knows a key (t, d), it can retrieve the soft kernel vector of this key and an encrypted document ID.

The second method is based on the work in [20, 107, 108], the server receives a list of query term IDs and a list of candidate documents ID. It then computes each key (t, d) and learns about term ID t and document ID d. For this case, there is additional leakage.

- Gradually after processing more queries, the server learns keys and kernel vectors, and also is able to build a partial soft inverted index for all queried terms. Based on the soft inverted index, the server is able to estimate partial document similarities based on the overlapping degree of soft terms between two documents.

- Once all terms and documents are queried or processed, the server is able to build a complete soft forward index and soft inverted index. Namely give a list of documents that contain a soft term, and give a list of soft terms included in a document. By using the soft term postings, the server learns the membership information of each soft term closure.

Since the information listed above is slowly leaked as more queries are processed, we propose to re-index the dataset periodically and replace the index in the cloud with different term IDs during each update to mitigate the chance of letting the server exploit the entire soft inverted index. The next two subsections study the privacy properties with respect to exact term occurrence even if the entire soft index is leaked to a server adversary, since such information is required for the plaintext attacks discussed in the last part of Section 6.2. We will also discuss k-anonymity for kernel vector values.

6.5.2 k-anonymization of Kernel Value Vectors

We show our obfuscation mapping achieves k-anonymization (a standard notion from privacy literature [104, 105]) with respect to the kernel value $K_j(t, d)$.

Definition 3 *[104] Let V be a vector of real values representing $R-1$ kernel values namely, $V = [K_1(t,d), K_2(t,d), \cdots, K_{R-1}(t,d)]$. Let $V' = \mathcal{F}(V)$ where $\mathcal{F}$ is a transform function. V' is k-anonymous if and only if, for any V', there are at least k different V such that $V' = \mathcal{F}(V)$. An algorithm $\mathcal{F}$ is called k-anonymization algorithm, if it outputs a k-anonymous vector V' for any soft matching signals vector V.*

It is easy to show that the application of the above ceiling function to the above logarithmic mapping is k-anonymous. In the first group, there are r values $0, 1, 2, ..., (r-1)$ mapped to 1, and in the k-th group, there are $r^k - r^{k-1}$ values $r^{k-1}, ..., r^k - 1$ mapped to k for $k > 1$. Since $r \geq 2$, k-th group has $r^k - r^{k-1} > r(r-1) \geq r$ values mapped to k, there are at least r values being mapped into the same value for each group. As there are $(R-1)$ kernels, given a sequence of $\lceil \log_r(K_j(t,d)) \rceil$, there are r^{R-1} different sequences of $K_j(t,d)$. Thus here we can confirm that there are $r^{R-1} - 1$ sequences of kernel values that the adversary cannot distinguish from the real one. In our evaluation, we choose $r = 10$ and $R = 20$. Thus $r^{R-1} = 10^{19}$, which is a very large number.

Proposition 6.5.1 *The logarithmic mapping with ceiling obfuscation is k-anonymous with respect to soft match kernel values, where $k = r^{R-1}$.*

6.5.3 Obfuscation of Exact Term Occurrence

How strong a closed soft match map can be in avoiding the leakage of term occurrence? If an adversary including a server tries to detect an exact term of a document from a soft match map which contains both soft and exact terms with encrypted term IDs, we argue that other similar soft terms from the same closure behave closely as the server only knows the structure and the kernel values in this soft match map, and thus the adversary should have a hard time to differentiate. We justify this argument by showing that there are too many variations of a document set with *statistical indistinguishable* softmaps. This notion of statistical indistinguishability is used in the cryptographic literature [109, 110].

We define a *closure transformation* of a document set D as follows. Let all terms of document d in D be partitioned into a smallest set of term groups so that each group of terms belong to the same term closures. Assume a group of terms belong to term closure C, and then this transformation replaces this group of terms with any nonempty subset of C for document d. Such a change can be applied to multiple term groups in each document in D, which results in another dataset.

For each closure C, there is a total of $2^{|C|} - 1$ nonempty subsets and thus there are totally $2^{|C|} - 1$ ways to replace each term group. Notice this includes a case that the original term group is not changed. Let p be minimum closure size and thus there are at least $2^p - 1$ ways to replace each term group. If keys (t_1, d_1) and (t_2, d_1) are in a softmap, and t_1 and t_2 are in the same term closure, we will consider these two pairs fall into the same closure-document pair. Let N be the distinct number of closure-document pairs in a softmap. There are $(2^p - 1)^N$ possible document sets produced by applying a transformation to some of documents in D.

Definition 4 ***ε-statistical indistinguishability***. *The kernel vectors of document d in dataset D and its corresponding document d' after transformation are:*

$$\vec{f_{t,d}} = (a_1, , \cdots , a_j, \cdots , a_R)^T, \vec{f_{t,d'}} = (a'_1, , \cdots , a'_j, \cdots , a'_R)^T.$$

Define the statistical distance between $\vec{f_{t,d}}$ and $\vec{f_{t,d'}}$ as $SD(\vec{f_{t,d}}, \vec{f_{t,d'}}) = \frac{1}{2}\sum_{i=1}^{R-1} |a_i - a'_i|$ following [109, 110]. We call the softmaps of D and its transformed dataset D' are ε-statistically indistinguishable if $SD(\vec{f_{t,d}}, \vec{f_{t,d'}}) \leq \varepsilon$ for all corresponding pairs of d in D and d' in D'.

Theorem 6.5.2 *Given a document set D in which each term closure has at least p terms, there are N closure-document pairs in its closed softmap. Assume that for any document d, and its transformed document d' after a closure transformation satisfy $SD(\vec{f}_{t,d}, \vec{f}_{t,d'}) \leq \varepsilon$. There exist $(2^p - 1)^N$ different document sets D' such that the keys (term-document pairs) of its soft match map for D and D' are identical, while these two softmaps are ε-statistically indistinguishable.*

Proof of Theorem 6.5.2

For each term closure, if at least one term in that closure appears in document d, all terms in that closure would have precomputed kernel values with d. For each closure C, there is a total of $2^{|C|} - 1$ nonempty subsets. Thus there are totally $2^{|C|} - 2$ ways to replace the exact terms in closure C appeared in d with another subset of C containing soft terms, results in a different term occurrence pattern for all terms in closure C. When a server tries to guess the existence of exact term occurrence (t, d), for all $t \in C$, it has to locate the correct one from all $2^{|C|} - 1$ ways of forming d using exact or soft terms from C, which is at least $2^p - 1$. When a server has N closure-document pairs, there are $(2^p - 1)^N$ different document sets $\tilde{D}$ that produce the soft match maps with the same keys. Any kernel vector $\vec{f}_{t,d}$ for d in D is ε-statistically indistinguishable from the corresponding kernel vector $\vec{f}_{t,d'}$ for d' in any document sets $\tilde{D}$, since d' is a closure transformation of d.

Theorem 6.5.2 shows that for any soft match map generated by a document set D, there are at least $(2^p - 1)^N$ document sets with different term occurrences and co-occurrences, while having very similar soft match maps. Considering N is very large, it is less likely that a server can differentiate these datasets and derive the document occurrence of exact terms. In [12], the adversary queries 150 single-word queries to launch an IKK attack. Assuming the average posting length is 100, the number of term-document pairs guessed for a document set D is 15,000. With adaptive clustering, the size of all clusters can be limited within 30 in our tested datasets, thus $N = 15000/30 = 500$. If $p = 2$, then there are at least $3^{500} \approx 10^{238}$ document sets D' to correspond to the guessed inverted index. The adversary has to choose the correct one from these 10^{238} options. Note that any two such options disagree on term occurrence for at least one document. Hence there are 10^{238} term occurrence profiles for the adversary to choose from, which is very unlikely to succeed.

6.6 Evaluations

Here we evaluate relevance scores of the proposed privacy-aware neural ranking techniques using two TREC datasets and assess trade-offs of privacy and relevance, and storage and time cost.

Datasets. We use the following TREC test collections to do evaluations. 1) Robust04 uses TREC Disks 4 & 5 (excluding Congressional Records), which has about 0.5M news articles. 250 topic queries are collected from TREC Robust track 2004. 2) ClueWeb09-Cat-B uses ClueWeb09 Category B with 50M web pages. There are 150 topic queries from the TREC Web Tracks 2009, 2010 and 2011. Spam filtering is applied on ClueWeb09 Category B using Waterloo spam score with threshold 60. During indexing and retrieval, Krovetz Stemming [57] is used for both queries and documents.

Features and training. Candidate documents with their encrypted feature vectors are retrieved from the inverted index built for the above datasets, following the work in [25, 111, 21]. For a privacy-aware tree ensemble, we use CPM [26] with LambdaMART based on RankLib 2.5 [112]. In each fold of training ranking model, 5-fold cross validation is used to select the best model based on NDCG@20, varying the number of leaves from 2 to 30 and the number of trees from 100 to 500.

To evaluate the impact of feature choices with the integration of the tree ensemble on the final ranking relevance, we have three options of features listed as follows.

1) **G0 with term frequency features:** BM25 scores for query terms in the title field, and BM25 scores for query terms in the body field of each document. TF-IDF scores for query terms in the title field, and TF-IDF scores for query terms in the body field of each document. 2) **G1 with term frequency and proximity features:** All features from G0, the squared minimum distance reciprocal of query term pairs in the title field, and the squared minimum distance reciprocal [99, 100, 101, 102] of query term pairs in the body field of each document. 3) **G2 with term frequency, proximity, and page quality features:** All features from G1, PageRank, and a binary flag indicating whether a document is from Wikipedia. This group is only for ClueWeb09 Category B.

Neural Ranking Models. The baseline models are DRMM, KNRM, and CONV-KNRM trained with 5-fold cross validation. We also choose a variant of CONV-KNRM, denoted by CONV-KNRM*. For CONV-KNRM*, we only use the interactions between query unigrams and document unigrams, between query unigrams and document bigrams, and between query bigrams and document unigrams. The interaction between query bigrams and document bigrams are not included to reduce storage space need. For both histogram pooling and kernel pooling, R=30 kernels are used. All soft match kernels are equally distributed in the cosine range. In kernel pooling, σ is 0.10 for all soft match kernels. In CONV-KNRM, n-gram length is 2, and the number of CNN filters is 128 as used in the original work. All word embedding vectors are pre-trained and fixed in KNRM, CONV-KNRM and CONV-KNRM*. We use 300 dimension word embedding vectors trained on TREC Disks 4 & 5 or ClueWeb09 Category-Cat-B with Skip-gram + Negative sampling model [96]. All terms that appear less than 5 times are removed from embedding training.

We assess the use of the following 3 techniques denoted with T, O, and C where T stands for the replacement of the exact match kernel with LambdaMART/CPM, O stands for kernel value obfuscation, and C stands for using a closed soft match map. Notation A/T means ranking A with technique T while A/TOC means ranking A with all 3 techniques. All NDCG [113] values are within confidence interval ±0.01 with p-value < 0.05.

Table 6.1: Relevance impact of replacing exact match kernel for ClueWeb09-Cat-B

Model	Feature group	NDCG@1	NDCG@3	NDCG@5	NDCG@10
LambdaMART /CPM	G0	0.2498	0.2702	0.2571	0.2415
	G1	0.2818	0.2725	0.2688	0.2653
	G2	**0.2893**	**0.2828**	**0.2873**	**0.2827**
DRMM	Baseline	0.2586	0.2659	0.2659	0.2634
DRMM/T	G0	0.2635	0.2721	0.2623	0.2503
	G1	0.2838	0.2778	0.2772	0.2645
	G2	**0.2887**	**0.2857**	**0.2822**	**0.2793**
KNRM	Baseline	0.2663	0.2739	0.2693	0.2681
KNRM/T	G0	0.2736	0.2804	0.2798	0.2725
	G1	**0.3036**	0.2974	0.2951	0.2903
	G2	0.2999	**0.3097**	**0.3154**	**0.3147**
CONV-KNRM	Baseline	0.3155	0.3124	0.3126	0.3085
CONV-KNRM/T	G0	0.3031	0.3088	0.3154	0.3052
	G1	**0.3254**	0.3187	0.3177	0.3099
	G2	0.3225	**0.3200**	**0.3210**	**0.3216**
CONV-KNRM*	-	0.2884	0.2927	0.2906	0.2870
CONV-KNRM*/T	G0	0.3038	0.2998	0.2962	0.2933
	G1	**0.3276**	0.3099	0.3099	0.3117
	G2	0.3175	**0.3122**	**0.3239**	**0.3218**

Impact of replacing the exact match kernel with a LambdaMART/CPM tree ensemble. Table 6.1 and Table 6.2 show the NDCG relevance of the three neural ranking models as the baseline and relevance after the replacement of the exact match kernel with LambdaMART/CPM based on the three groups of features G0, G1, and G2. Soft match maps and kernel value obfuscation are not incorporated. The boldfaced numbers are the

Table 6.2: Relevance impact of replacing exact match kernel for Robust04

Model	Feature group	NDCG@1	NDCG@3	NDCG@5	NDCG@10
LambdaMART /CPM	G0	0.4819	0.4465	0.4257	0.3982
	G1	**0.5181**	**0.4610**	**0.4346**	**0.4044**
DRMM	Baseline	0.5049	**0.4872**	**0.4747**	**0.4528**
DRMM/T	G0	0.4993	0.4594	0.4425	0.4134
	G1	**0.5114**	0.4658	0.4501	0.4158
KNRM	Baseline	0.4983	0.4812	0.4647	0.4527
KNRM/T	G0	0.5158	0.4908	0.4768	0.4592
	G1	**0.5382**	**0.5063**	**0.4906**	**0.4673**
CONV-KNRM	Baseline	0.5373	0.4875	0.4742	0.4586
CONV-KNRM/T	G0	0.5402	**0.5057**	0.4894	0.4643
	G1	**0.5556**	0.5042	**0.4927**	**0.4693**
CONV-KNRM*	-	0.5007	0.4702	0.4601	0.4510
CONV-KNRM*/T	G0	0.5149	0.4827	0.4768	0.4535
	G1	**0.5404**	**0.5006**	**0.4892**	**0.4657**

Table 6.3: Impact of kernel value obfuscation for ClueWeb09-Cat-B

Model	Obfuscation base (r)	NDCG@1	NDCG@3	NDCG@5	NDCG@10
DRMM/TO	Yes(10)	0.2703	0.2731	0.2740	0.2732
	Yes(5)	0.2769	0.2757	0.2753	0.2722
	No	0.2887	0.2857	0.2822	0.2793
KNRM/TO	Yes(10)	0.2808	0.2929	0.2947	0.2906
	Yes(5)	0.2875	0.2968	0.2988	0.2971
	No	0.2999	0.3097	0.3154	0.3147
CONV-KNRM*/TO	Yes(10)	0.3121	0.3097	0.3165	0.3100
	Yes(5)	0.3178	0.3067	0.3161	0.3100
	No	0.3175	0.3122	0.3239	0.3218

Table 6.4: Impact of kernel value obfuscation for Robust04

Model	Obfuscation base (r)	NDCG@1	NDCG@3	NDCG@5	NDCG@10
DRMM/TO	Yes(10)	0.5078	0.4681	0.4449	0.4157
	Yes(5)	0.5110	0.4669	0.4446	0.4221
	No	0.5114	0.4658	0.4501	0.4158
KNRM/TO	Yes(10)	0.5117	0.4639	0.4393	0.4130
	Yes(5)	0.5100	0.4686	0.4451	0.4164
	No	0.5382	0.5063	0.4906	0.4673
CONV-KNRM*/TO	Yes(10)	0.5221	0.4980	0.4906	0.4623
	Yes(5)	0.5306	0.4987	0.4893	0.4613
	No	0.5404	0.5006	0.4892	0.4657

highest NDCG scores within each ranking model. From this table we observe that neural ranking with the use of LambdaMART/CPM trees outperforms the original baseline in NDCG at all Positions 1, 3, 5, and 10 for ClueWeb. For example, compared with CONV-KNRM, CONV-KNRM/T with G2 can improve NDCG@1, NDCG@3, NDCG@5 and NDCG@10 by 2.23%, 2.45%, 2.67% and 4.23% on ClueWeb. For Robust04, tree ensemble integration delivers up to 3.91% improvement for CONV-KNRM and up to 8.02% for KNRM, but degrades by up to -8.18% for DRMM. Comparing the use of $G0$, $G1$, and $G2$ for neural ranking integration, $G2$ is still most effective for ClueWeb and $G1$ is most effective for Robust04, which shows traditional signals still make a good contribution.

We examine NDCG scores reported in the previous work. For ClueWeb09-Cat-B, NDCG scores at Positions 1, 10 and 20 are 0.294, 0.289, 0.287 with CONV-KNRM in [7]. Our numbers are slightly higher, which can be caused by different data processing. Notice NDCG@20 in our run is 0.2950.

By comparing CONV-KNRM and CONV-KNRM*, the absence of bigram-bigram interaction does yield a loss of NDCG score. For example, the loss is 8.56%, 6.29%,

Table 6.5: NDCG score and storage demand for CONV-KNRM*/TOC

Similarity threshold	NDCG@1	NDCG@3	NDCG@5	NDCG@10	Storage (GB)
Robust04 & Clustering with fixed threshold					
0.3	0.5225	0.4974	0.4915	0.4621	45,021 (2,144)
0.5	0.5154	0.4883	0.4780	0.4543	24,793 (1,181)
0.7	0.4886	0.4644	0.4486	0.4169	287 (13)
0.9	0.4953	0.4594	0.4415	0.4091	261 (12)
Robust04 & Adaptive clustering					
0.3	0.5127	0.4892	0.4845	0.4582	1,512 (72)
0.5	0.5078	0.4845	0.4756	0.4498	1,133 (54)
0.7	0.4899	0.4608	0.4414	0.4110	278 (13)
0.9	0.4913	0.4558	0.4397	0.4090	261 (12)
ClueWeb09-Cat-B & Clustering with fixed threshold					
0.3	0.3136	0.3078	0.3149	0.3091	$1.7 \cdot 10^6$ (82,308)
0.5	0.3073	0.3069	0.3114	0.3069	$1.3 \cdot 10^6$ (61,877)
0.7	0.3064	0.3048	0.3122	0.3104	16,568 (792)
0.9	0.3069	0.3041	0.3105	0.3074	7,369 (354)
ClueWeb09-Cat-B & Adaptive clustering					
0.3	0.3052	0.3069	0.3142	0.3120	46,811 (2,231)
0.5	0.3056	0.3037	0.3113	0.3103	35,742 (1,705)
0.7	0.3067	0.3012	0.3088	0.3060	7,627 (366)
0.9	0.2963	0.3025	0.3117	0.3083	7,369 (354)

7.04% and 6.96% for ClueWeb at Positions 1, 3, 5, and 10, respectively. That represents a tradeoff of privacy and relevancy. Adding the tree ensemble integration, most NDCG scores for CONV-KNRM*/T can be on par with those of CONV-KNRM and are 4.31% better for ClueWeb09 at Position 10.

Impact of kernel value obfuscation. Table 6.3 and Table 6.4 show the impact of kernel value obfuscation on NDCG scores incorporating LambdaMART/CPM with G2 for ClueWeb and G1 for Robust04. We choose two different logarithmic base r here: 5 and 10. Overall, the relevance with $r = 5$ is slightly better than $r = 10$, while both of them result in degradation in ranking accuracy compared with no obfuscation. For NDCG@1, the degradation with $r = 10$ is 1.7% for CONV-KNRM, 6.36% for KNRM, and 6.3% for DRMM. For CONV-KNRM*, its degradation is relatively smaller.

Trade-offs between relevancy and storage efficiency. Table 6.5 studies the impact of using a closed soft match map with two clustering methods for term closures under different thresholds. This table is for CONV-KNRM*/TOC only as this model delivers the highest NDCG scores with all privacy preserving techniques. For example, with the obfuscation base being 10, and the clustering threshold being 0.7, for DRMM/TOC in ClueWeb, NDCG@5 and NDCG@10 are 0.2704 and 0.2720, respectively. For KNRM/TOC in ClueWeb, NDCG@5 and NDCG@10 are 0.2972 and 0.2912, respectively. In Table 6.5, Column 6 in the middle shows the storage need in gigabytes to store a soft match map and related data after clustering with fixed thresholds while last column on the right is the storage demand with adaptive clustering. Each entry has two numbers X(Y). Y is the total storage for unigram-unigram interaction while X is the total storage for unigram-unigram, unigram-bigram, and bigram-unigram interaction. Number Y also represents the amount of storage space needed for KNRM and DRMM.

While clustering with a fixed threshold yields some relevance improvement over adaptive clustering, it requires an excessive amount of storage space. The adaptive clustering threshold 0.7 is a good trade-off with an acceptable storage space for hosting the ClueWeb dataset. In this setting, the relevance of CONV-KNRM* is on par with the original CONV-KNRM baseline, lower than CONV-KNRM/T. That represents a trade-off of relevancy, privacy and storage cost. It still requires 7.627TB space and we can use a number of high-end SSDs with parallel I/O. The latest high-end SSD products from Intel [114] and Samsung [115, 116] have achieved 10-15μs IO latency with up-to 750K I/O operations per second. Thus for a soft match map hosted at a high-end SSD, the I/O access time of processing one query can still be reasonable.

Estimation of online query processing time. The online query processing time cost consists of 3 phases: 1) private result retrieval and preliminary ranking, 2) private tree ensemble scoring, 3) neural ranking. Based on [25], Phase I costs 460 ms on average for ClueWeb. For re-ranking top 1,000 candidate documents, there are about up-to 10K IO operations needed to fetch kernel vectors and features, and the total I/O time for accessing SSDs can take around 100 to 150ms with the above fast SSD performance parameters. Our experiments show that private tree ensemble scoring takes less than 2ms and all three neural models with TOC take about 10 ms or less. Thus overall the online query processing time is about 572ms to 622ms on average for ClueWeb. Notice that CONV-KNRM requires computation of term vectors and interaction matrices which could take 4-5 seconds. Thus even though our design pays extra cost in space, it does remove the expensive time spent for interaction computation.

6.7 Summary

The main contribution of this chapter is a privacy-aware neural ranking scheme integrated with a tree ensemble for server-side top K document search. The key techniques include the replacement of the exact kernel with a tree ensemble, a soft match map using obfuscated kernel values and term closures, and adaptive clustering for term occurrence obfuscation and storage optimization. Our design for privacy enhancement is to prevent the leakage of two critical text signals in terms of term frequency and occurrence needed for the attacks shown in the previous work and this chapter.

The evaluation with two TREC datasets shows that the NDCG can be improved noticeably by replacing the exact match kernel of neural ranking with a LambdaMART tree ensemble. The obfuscation of kernel values does carry a modest relevance trade-off for privacy. The adaptive clustering for term closures significantly reduces the storage demand with some trade-off in relevance.

Chapter 7

Conclusion and Future Works

In this dissertation, we have sought the optimized tradeoffs among privacy, efficiency, and relevance for privacy-aware search with additive and neural ranking. We firstly in Chapter 3 proposes and evaluates a top-k search flow with window navigation and adaptive probing. Next in Chapter 4, we present an oblivious document retrieval scheme MII that obfuscates the access pattern of inverted index data in a TEE. Chapter 5 proposes a top-k retrieval algorithm that supports dynamic pruning, seals the access pattern leakage using ORAM and achieves optimized query processing times for all query lengths. Finally in Chapter 6, a privacy-aware neural ranking scheme is designed to enable server-side top-k document search with advanced neural models.

This thesis has made several comprehensive approaches to improve the efficiency of top-k search in document retrieval and ranking while providing a privacy protection. There are still many important but challenging open problems that can be investigated in the future and some of them are listed as follows:

1. **Distributed oblivious top-k retrieval.** To further accelerate the query processing of oblivious top-k retrieval with the hardware-supported ORAM, a large-scale datasets can be distributed into multiple partitions, and let a single machine han-

dle one partition. However, if these machines process each partition separately, the benefit of dynamic pruning can be reduced. Thus, it is attractive to design a oblivious top-k retrieval that supports dynamic pruning in a distributed manner.

2. **Privacy-preserving neural ranking with a TEE.** Neural ranking models involves a huge overhead of computation and storage overhead as described in Chapter 6. Although a TEE can efficiently help secure computations, it is still difficult to design the advance neural ranking such as BERT in a privacy-preserving way.

3. **Accurate window navigation with probing.** The effectiveness of window navigation with probing has been shown in Chapter 3. However, one interesting question is how to accurately locate those promising windows for probing (instead of using $WinMax$) such that the final top-k documents can be quickly identified.

Bibliography

[1] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. Scott, and N. Wilkins-Diehr, *Xsede: Accelerating scientific discovery*, *Computing in Science & Engineering* **16** (2014), no. 05 62–74.

[2] T. P. Institute, "The 2018 global cloud data security study." https://www2.gemalto.com/cloud-security-research, 2018. Accessed: 2018-05-01.

[3] C. Gentry, *Fully homomorphic encryption using ideal lattices*, in *STOC '09*, pp. 169–178, ACM, 2009.

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

[5] J. Guo, Y. Fan, Q. Ai, and W. B. Croft, *A deep relevance matching model for ad-hoc retrieval*, in *Proceedings of CIKM'16*, pp. 55–64, ACM, 2016.

[6] C. Xiong, Z. Dai, J. Callan, Z. Liu, and R. Power, *End-to-end neural ad-hoc ranking with kernel pooling*, in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 55–64, ACM, 2017.

[7] Z. Dai, C. Xiong, J. Callan, and Z. Liu, *Convolutional neural networks for soft-matching n-grams in ad-hoc search*, in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pp. 126–134, ACM, 2018.

[8] C. Xiong, Z. Liu, J. Callan, and T.-Y. Liu, *Towards better text understanding and retrieval through kernel entity salience modeling*, in *The 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '18, pp. 575–584, ACM, 2018.

[9] L. Pang, Y. Lan, J. Guo, J. Xu, J. Xu, and X. Cheng, *Deeprank: A new deep architecture for relevance ranking in information retrieval*, in *Proceedings of CIKM'17*, pp. 257–266, ACM, 2017.

[10] 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.

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

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

[13] K. S. Jones, S. Walker, and S. E. Robertson, *A probabilistic model of information retrieval: development and comparative experiments*, in *Information Processing and Management*, pp. 779–840, 2000.

[14] S. Ding and T. Suel, *Faster top-k document retrieval using block-max indexes*, in *Proc. of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 993–1002, 2011.

[15] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien, *Efficient query evaluation using a two-level retrieval process*, in *Proc. of the 12th ACM International Conference on Information and Knowledge Management*, pp. 426–434, 2003.

[16] T. Strohman and W. B. Croft, *Efficient document retrieval in main memory*, in *Proc. of the 30th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 175–182, 2007.

[17] A. Mallia, G. Ottaviano, E. Porciani, N. Tonellotto, and R. Venturini, *Faster blockmax wand with variable-sized blocks*, in *Proc. of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 625–634, 2017.

[18] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, *Searchable symmetric encryption: improved definitions and efficient constructions*, *Journal of Computer Security* **19** (2011), no. 5 895–934.

[19] S. Kamara, C. Papamanthou, and T. Roeder, *Dynamic searchable symmetric encryption*, in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 965–976, ACM, 2012.

[20] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, *Dynamic searchable encryption in very-large databases: Data structures and implementation.*, in *NDSS*, vol. 14, pp. 23–26, Citeseer, 2014.

[21] 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.

[22] 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.

[23] 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.

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

[25] D. Agun, J. Shao, S. Ji, S. Tessaro, and T. Yang, *Privacy and efficiency tradeoffs for multiword top k search with linear additive rank scoring*, in *Proceedings of the 2018 World Wide Web Conference*, pp. 1725–1734, International World Wide Web Conferences Steering Committee, 2018.

[26] 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.

[27] G. Wang, C. Liu, Y. Dong, K.-K. R. Choo, P. Han, H. Pan, and B. Fang, *Leakage models and inference attacks on searchable encryption for cyber-physical social systems*, *IEEE Access* **6** (2018) 21828–21839.

[28] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, *Hardidx: Practical and secure index with sgx*, in *IFIP Ann. Conf. on Data and App. Security and Privacy*, pp. 386–408, 2017.

[29] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, *Hardware-supported oram in effect: Practical oblivious search and update on very large dataset*, *Proc. on Privacy Enhancing Technologies* (2019), no. 1 172–191.

[30] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, *Oblivious multi-party machine learning on trusted processors*, in *Proc. of 2016 USENIX Security Symp.*, pp. 619–636, 2016.

[31] V. Costan and S. Devadas, *Intel sgx explained.*, *IACR Cryptology ePrint Archive* (2016) 1–118.

[32] H. Turtle and J. Flood, *Query evaluation: Strategies and optimizations*, *Information Processing & Management* **31** (1995), no. 6 831850.

[33] C. Dimopoulos, S. Nepomnyachiy, and T. Suel, *A candidate filtering mechanism for fast top-k query processing on modern cpus*, in *Proc. of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 723–732, 2013.

[34] M. Crane, J. S. Culpepper, J. Lin, J. Mackenzie, and A. Trotman, *A comparison of document-at-a-time and score-at-a-time query evaluation*, in *Proc. of the 10th ACM International Conference on Web Search and Data Mining*, p. 201210, 2017.

[35] N. Tonellotto, C. Macdonald, and I. Ounis, *Efficient query processing for scalable web search*, *Foundations and Trends in Information Retrieval* **12** (2018), no. 4-5 319–500.

[36] J. Mackenzie, J. S. Culpepper, R. Blanco, M. Crane, C. L. A. Clarke, and J. Lin, *Query driven algorithm selection in early stage retrieval*, in *Proc. of the 11th ACM International Conference on Web Search and Data Mining*, p. 396404, 2018.

[37] J. Mackenzie and A. Moffat, *Examining the additivity of top-k query processing innovations*, in *Proc. of the 29th ACM International Conference on Information and Knowledge Management*, p. 10851094, 2020.

[38] A. Kane and F. W. Tompa, *Split-lists and initial thresholds for wand-based search*, in *Proc. of the 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*, p. 877880, 2018.

[39] A. Mallia, M. Siedlaczek, and T. Suel, *An experimental study of index compression and DAAT query processing methods*, in *Proc. of 41st European Conference on IR Research, ECIR' 2019*, pp. 353–368, 2019.

[40] M. Petri, A. Moffat, J. Mackenzie, J. S. Culpepper, and D. Beck, *Accelerated query processing via similarity score prediction*, in *Proc. of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, p. 485494, 2019.

[41] A. Mallia, M. Siedlaczek, M. Sun, and T. Suel, *A comparison of top-k threshold estimation techniques for disjunctive query processing*, in *Proc. of the 29th ACM International Conference on Information and Knowledge Management*, pp. 2141–2144, 2020.

[42] D. Shan, S. Ding, J. He, H. Yan, and X. Li, *Optimized top-k processing with global page scores on block-max indexes*, in *Proc. of the 15th ACM International Conference on Web Search and Data Mining*, p. 423432, 2012.

[43] O. Khattab, M. Hammoud, and T. Elsayed, *Finding the best of both worlds: Faster and more robust top-k document retrieval*, in *Proc. of the 43rd*

International ACM SIGIR Conference on Research and Development in Information Retrieval, p. 10311040, 2020.

[44] L. L. S. de Carvalho, E. S. de Moura, C. M. Daoud, and A. S. da Silva, *Heuristics to improve the BMW method and its variants*, *Journal of Information and Data Management* **6** (2015), no. 3 178–191.

[45] C. Rossi, E. S. de Moura, A. L. Carvalho, and A. S. da Silva, *Fast document-at-a-time query processing using two-tier indexes*, in *Proc. of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 183–192, 2013.

[46] E. Yafay and I. S. Altingovde, *Caching scores for faster query processing with dynamic pruning in search engines*, in *Proc. of the 28th ACM International Conference on Information and Knowledge Management*, pp. 2457–2460, 2019.

[47] C. Dimopoulos, S. Nepomnyachiy, and T. Suel, *Optimizing top-k document retrieval strategies for block-max indexes*, in *Proc. of the 6th ACM International Conference on Web Search and Data Mining*, p. 113122, 2013.

[48] A. Mallia, M. Siedlaczek, and T. Suel, *Fast disjunctive candidate generation using live block filtering*, in *Proc. of the 14th ACM International Conference on Web Search and Data Mining*, p. 671679, 2021.

[49] K. Chakrabarti, S. Chaudhuri, and V. Ganti, *Interval-based pruning for top-k processing over compressed lists*, in *Proc. of the 2011 IEEE 27th International Conference on Data Engineering*, p. 709720, 2011.

[50] C. M. Daoud, E. Silva de Moura, A. Carvalho, A. Soares da Silva, D. Fernandes, and C. Rossi, *Fast top-k preserving query processing using two-tier indexes*, *Information Processing & Management* **52** (2016), no. 5 855872.

[51] C. M. Daoud, E. S. Moura, D. Fernandes, A. S. Silva, C. Rossi, and A. Carvalho, *Waves: a fast multi-tier top-k query processing algorithm*, *Information Retrieval* **20** (2017), no. 3 292–316.

[52] E. Bortnikov, D. Carmel, and G. Golan-Gueta, *Top-k query processing with conditional skips*, in *Proc. of the 26th International Conference on World Wide Web Companion*, pp. 653–661, 2017.

[53] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition.* MIT Press, Cambridge, MA, USA, 2009.

[54] M. Fontoura, V. Josifovski, J. Liu, S. Venkatesan, X. Zhu, and J. Zien, *Evaluation strategies for top-k queries over memory-resident inverted indexes*, *Proc. VLDB Endow.* **4** (2011), no. 12 12131224.

[55] G. V. Cormack, M. D. Smucker, and C. L. Clarke, *Efficient and effective spam filtering and re-ranking for large web datasets*, *Information Retrieval* **14** (2011), no. 5 441–465.

[56] T. L. Project, *https://www.lemurproject.org/indri.php*, 2020.

[57] R. Krovetz, *Viewing morphology as an inference process*, *Artificial Intelligence* **118** (2000), no. 1-2 277–294.

[58] D. Lemire and L. Boytsov, *Decoding billions of integers per second through vectorization*, *Softw. Pract. Exp.* **45** (2015), no. 1 1–29.

[59] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz, *Analysis of a very large web search engine query log*, in *ACM SIGIR Forum*, vol. 33, pp. 6–12, ACM, 1999.

[60] B. J. Jansen and A. Spink, *How are we searching the world wide web? a comparison of nine search engine transaction logs*, *Information Processing & Management* **42** (2006), no. 1 248–263.

[61] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan, *Search pattern leakage in searchable encryption: Attacks and new construction*, *Information Sciences* **265** (2014) 176–188.

[62] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, *Software grand exposure: Sgx cache attacks are practical*, in *11th USENIX Workshop on Offensive Technologies, 2017*, 2017.

[63] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, *Oblix: An efficient oblivious search index*, in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 279–296, 2018.

[64] W. Sun, R. Zhang, W. Lou, and Y. T. Hou, *Rearguard: Secure keyword search using trusted hardware*, in *IEEE INFOCOM 2018*, pp. 801–809, 2018.

[65] M. Naveed, M. Prabhakaran, and C. A. Gunter, *Dynamic searchable encryption via blind storage*, in *2014 IEEE Symposium on Security and Privacy*, pp. 639–654, IEEE, 2014.

[66] 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.

[67] 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.

[68] J. Shao, S. Ji, and T. Yang, *Privacy-aware document ranking with neural signals*, in *Proc. of 2019 SIGIR*, p. 305314, 2019.

[69] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, *Opaque: An oblivious and encrypted distributed analytics platform*, in *NSDI 17*, pp. 283–298, 2017.

[70] O. Goldreich and R. Ostrovsky, *Software protection and simulation on oblivious rams*, *Journal of the ACM (JACM)* **43** (1996), no. 3 431–473.

[71] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, *Path oram: An extremely simple oblivious ram protocol*, *Journal of the ACM* **65** (2018), no. 4 1–26.

[72] V. N. Anh and A. Moffat, *Inverted index compression using word-aligned binary codes*, *Information Retrieval* **8** (2005), no. 1 151–166.

[73] J. Zhang, X. Long, and T. Suel, *Performance of compressed inverted list caching in search engines*, in *Proc. of the 17th International Conference on World Wide Web, WWW*, pp. 387–396, 2008.

[74] S. Sasy, S. Gorbunov, and C. W. Fletcher, *Zerotrace: Oblivious memory primitives from intel sgx.*, in *NDSS 2018*, 2018.

[75] O. Goldreich, *Towards a theory of software protection and simulation by oblivious rams*, in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pp. 182–194, 1987.

[76] R. Ostrovsky, *Software protection and simulation on oblivious RAMs.* PhD thesis, Massachusetts Institute of Technology, 1992.

[77] R. Ostrovsky, *Efficient computation on oblivious rams*, in *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, (New York, NY, USA), p. 514523, Association for Computing Machinery, 1990.

[78] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, *Path oram: an extremely simple oblivious ram protocol*, in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 299–310, 2013.

[79] X. Wang, H. Chan, and E. Shi, *Circuit oram: On tightness of the goldreich-ostrovsky lower bound*, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 850–861, 2015.

[80] J. Shao, S. Ji, A. O. Glova, Y. Qiao, T. Yang, and T. Sherwood, *Index obfuscation for oblivious document retrieval in a trusted execution environment*, in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pp. 1345–1354, 2020.

[81] Y. Xu, W. Cui, and M. Peinado, *Controlled-channel attacks: Deterministic side channels for untrusted operating systems*, in *2015 IEEE Symposium on Security and Privacy*, pp. 640–656, IEEE, 2015.

[82] L. Blackstone, S. Kamara, and T. Moataz, *Revisiting leakage abuse attacks.*, *IACR Cryptol. ePrint Arch.* **2019** (2019) 1175.

[83] Y. Zhang, J. Katz, and C. Papamanthou, *All your queries are belong to us: The power of file-injection attacks on searchable encryption*, in *25th USENIX Security Symposium (USENIX Security 16)*, pp. 707–720, USENIX Association, 2016.

[84] J. Shao, Y. Qiao, S. Ji, and T. Yang, *Window navigation with adaptive probing for executing blockmax wand*, in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2323–2327, 2021.

[85] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang, *Oblivious data structures*, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, p. 215226, Association for Computing Machinery, 2014.

[86] L. Pang, Y. Lan, J. Guo, J. Xu, and X. Cheng, *A deep investigation of deep ir models*, in *SIGIR 2017 Workshop on Neural Information Retrieval (Neu-IR'17)*, 2017.

[87] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, *Learning deep structured semantic models for web search using clickthrough data*, in *Proceedings of CIKM'13*, pp. 2333–2338, ACM, 2013.

[88] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil, *Learning semantic representations using convolutional neural networks for web search*, in *Proceedings of the 23rd International Conference on World Wide Web*, pp. 373–374, ACM, 2014.

[89] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, *Gazelle: A low latency framework for secure neural network inference*, in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 1651–1669, 2018.

[90] A. Boldyreva, N. Chenette, and A. O'Neill, *Order-preserving encryption revisited: Improved security analysis and alternative solutions*, in *Annual Cryptology Conference*, pp. 578–595, Springer, 2011.

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

[92] G. Jagannathan, K. Pillaipakkamnatt, and R. N. Wright, *A practical differentially private random decision tree classifier*, in *2009 IEEE International Conference on Data Mining Workshops*, pp. 114–121, IEEE, 2009.

[93] T.-Y. Liu *et. al.*, *Learning to rank for information retrieval*, *Foundations and Trends® in Information Retrieval* **3** (2009), no. 3 225–331.

[94] C. J. Burges, *From ranknet to lambdarank to lambdamart: An overview*, *Learning* **11** (2010), no. 23-581 81.

[95] M. Ibrahim and M. Carman, *Comparing pointwise and listwise objective functions for random-forest-based learning-to-rank*, *ACM Transactions on Information Systems (TOIS)* **34** (2016), no. 4 20.

[96] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, *Distributed representations of words and phrases and their compositionality*, in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[97] J. Pennington, R. Socher, and C. Manning, *Glove: Global vectors for word representation*, in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[98] H. Zamani and W. B. Croft, *Relevance-based word embedding*, in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 505–514, ACM, 2017.

[99] 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 19th Text REtrieval Conference, TREC 2010, Gaithersburg, Maryland, USA*, 2010.

[100] J. Bai, Y. Chang, H. Cui, Z. Zheng, G. Sun, and X. Li, *Investigation of partial query proximity in web search*, in *Proceedings of the 17th international conference on World Wide Web*, pp. 1183–1184, ACM, 2008.

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

[102] T. Tao and C. Zhai, *An exploration of proximity measures in information retrieval*, in *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 295–302, 2007.

[103] O. Chapelle and Y. Chang, *Yahoo! Learning to Rank Challenge Overview*, *J. of Machine Learning Research* (2011) 1–24.

[104] L. Sweeney, *k-anonymity: A model for protecting privacy*, *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* **10** (2002), no. 05 557–570.

[105] D. Di Castro, L. Lewin-Eytan, Y. Maarek, R. Wolff, and E. Zohar, *Enforcing k-anonymity in web mail auditing*, in *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pp. 327–336, ACM, 2016.

[106] S. Ji, J. Shao, and T. Yang, *Efficient interaction-based neural ranking with locality sensitive hashing*, in *The World Wide Web Conference*, WWW '19, (New York, NY, USA), pp. 2858–2864, ACM, 2019.

[107] R. Bost, $\sum o\varphi o\varsigma$*: Forward secure searchable encryption*, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1143–1154, ACM, 2016.

[108] R. Bost and P.-A. Fouque, "Thwarting leakage abuse attacks against searchable encryption – a formal approach and applications to database padding." Cryptology ePrint Archive, Report 2017/1060, 2017. `https://eprint.iacr.org/2017/1060`.

[109] D. Boneh and V. Shoup, *A graduate course in applied cryptography*, *Draft 0.2* (2015).

[110] O. Goldreich, *Foundations of Cryptography: Basic Tools.* Cambridge University Press, New York, NY, USA, 2000.

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

[112] V. Dang, "Ranklib." https://sourceforge.net/p/lemur/wiki/RankLib/, 2012. Accessed: 2018-05-20.

[113] K. Järvelin and J. Kekäläinen, *Cumulated gain-based evaluation of ir techniques*, *ACM Transactions on Information Systems (TOIS)* **20** (2002), no. 4 422–446.

[114] L. Smith, "Intel optane 800p nvme ssd review." `https://www.storagereview.com/intel_optane_800p_nvme_ssd_review`, 2018. Accessed: 2019-01-28.

[115] C. Mellor, "Samsung preps for z-ssd smackdown on intel optane drives." `https://www.theregister.co.uk/2018/01/30/samsung_launching_zssd_attack_on_intel_optane_drives`, 2018. Accessed: 2019-01-28.

[116] Samsung, "Samsung electronics begins mass production of industry's largest capacity ssd - 30.72tb - for next-generation enterprise systems." `https://bit.ly/2EFKp5N`, 2018. Accessed: 2019-01-28.