Embellishing Text Search Queries To Protect User Privacy

HweeHwa Pang Xuhua Ding
School of Information Systems
Singapore Management University
{hhpang, xhding}@smu.edu.sg

Xiaokui Xiao
School of Computer Engineering
Nanyang Technological University
xkxiao@ntu.edu.sg

ABSTRACT

Users of text search engines are increasingly wary that their activities may disclose confidential information about their business or personal profiles. It would be desirable for a search engine to perform document retrieval for users while protecting their intent. In this paper, we identify the privacy risks arising from semantically related search terms within a query, and from recurring high-specificity query terms in a search session. To counter the risks, we propose a solution for a similarity text retrieval system to offer anonymity and plausible deniability for the query terms, and hence the user intent, without degrading the system's precision-recall performance. The solution comprises a mechanism that embellishes each user query with decoy terms that exhibit similar specificity spread as the genuine terms, but point to plausible alternative topics. We also provide an accompanying retrieval scheme that enables the search engine to compute the encrypted document relevance scores from only the genuine search terms, yet remain oblivious to their distinction from the decoys. Empirical evaluation results are presented to substantiate the effectiveness of our solution.

1. INTRODUCTION

Today's text retrieval systems must have access to the queries and the index structures that facilitate document retrieval, and be trusted to not abuse the privilege. This arrangement is not always desirable. In a potent demonstration of the associated privacy risk, AOL recently released its Web log data, only to withdraw it soon after when it was shown that detailed user profiles could be constructed from the data [3]. Many users are thus justifiably wary that their queries could disclose personal or confidential information to the search engine, such as their interests or preferences.

Privacy in text search is a hard problem though. Most modern text search engines implement similarity retrieval, where result documents are ranked by their relevance to the user query. Similarity retrieval has been found to be more effective for general users than the Boolean model that finds only the documents containing all the search terms, with no relative ranking among the result documents [29]. To avoid computing the relevance of every document, search engines maintain an inverted list for each term, which contains its impact value in each document. This allows the result of a query to be composed from the inverted lists of the search terms, hence skipping the documents that do not contain any query term.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

The need to compute document relevance from inverted lists rules out encryption techniques that allow only Boolean matching, like those in [5], [10], [6] and [26]. Instead, query privacy is achieved through anonymization, as advocated in [12]. This approach was investigated systematically in [19], which proposed to formulate static sets of queries that cover diverse topics, and to replace each user query q with a query set S containing the query $\tilde{q}$ that best approximates q. $\tilde{q}$ serves as a surrogate that (hopefully) retrieves most of the result documents for q, while the remaining queries in S act as decoys. The scheme impacts precision-recall performance, because the result for $\tilde{q}$ is not necessarily a superset of q's result. Moreover, in practice only a small number of the exponentially increasing term combinations can be materialized, so the scheme is not designed for long queries that occur in general text search (e.g., TREC topics [27]) or query expansion [23, 28].

Problem formulation: Our objective is to deter a similarity-based text search engine from identifying the user intent from his query terms, without compromising precision-recall performance. Instead of hiding user queries among cover queries, we achieve higher anonymization by injecting decoy terms directly into each user query. At one extreme, the genuine and decoy terms in a query could span the entire dictionary. That offers complete privacy since the search engine has no idea which terms are being queried, but the concomitant overheads are too high. To be practical, we aim instead for anonymity and plausible deniability for the user intent, by expanding each query with a small subset of the dictionary terms that serve as decoys. This requires us to manage privacy risks from:

- *Semantically related search terms.* A query is likely to include multiple words that may be individually innocuous, but collectively point to a common topic. Such terms are easily distinguishable from random decoy terms. For instance, given the query {'accelerated', 'treadmill', 'radiation', 'flooding', 'saturn', 'therapy'}, the user is probably looking up 'accelerated radiation therapy' for some cancer, and the other terms are merely decoys. Another example of semantically related keywords is {'water', 'soaked', 'tissues'}, in the context of plant diseases.
- *Recurring high-specificity search terms.* In a search session, a user is likely to issue a series of related queries around some specific keywords. For example, a query "osteosarcoma symptoms" may be followed by another query "osteosarcoma therapy". With a large dictionary, the search term 'osteosarcoma" is unlikely to have been picked repeatedly as decoy by chance, so it is probably genuine. Since the term has a high *specificity*, i.e., it has a very specific meaning, its presence betrays the user's interest.

Contributions: To limit any inferences that the search engine may

draw from the search terms in a query, it is ineffective to throw decoy terms into it randomly. Instead, we counter the threat posed by semantically related search terms (e.g., 'accelerated', 'radiation', 'therapy') by adding, to each query, decoy terms that relate to plausible alternative topics (e.g., 'water', 'soaked', 'tissues'). As to the risk from recurring high-specificity search terms (e.g., 'osteosarcoma') in successive queries within a search session, our approach is to ensure that they also share decoys that are similarly specific (e.g., 'amaranthaceae'); consequently, intersecting the queries would yield diverse high-specificity terms. We derive the word specificity and semantic relations from a thesaurus.

After injecting decoys, all the genuine and decoy terms in the query are permuted randomly, to prevent the search engine from reinforcing its belief in the genuineness of a particular high-specificity term through the presence of other terms that are individually innocuous, but collectively relate to the same topic. For example, if a query contains both 'amaranthaceae' and 'hypocapnia', it is not obvious which is more plausible in light of other general or polysemous terms in the query like 'water', 'soaked' and 'tissues'.

When an embellished query arrives at the search engine, its query processing has to involve both the genuine and decoy terms since it cannot tell them apart. At the same time, the relevance of the query result cannot be compromised. To prevent the decoy terms in the embellished query from interfering with the documents' relevance scores, we present a secure retrieval scheme for the search engine to derive query results from only the genuine search terms, without discerning their distinction from the decoys.

The rest of the paper is organized as follows: Section 2 surveys related work, and defines the similarity text retrieval model. Our decoy injection mechanism is introduced in Section 3, while Section 4 presents the accompanying secure retrieval scheme. An empirical evaluation of our solution is reported in Section 5. Finally, Section 6 concludes the paper.

2. BACKGROUND

2.1 Related Work

Query privacy for Boolean text retrieval has been studied extensively. In that model, a query takes the form of a Boolean expression of search terms. Only documents that satisfy the Boolean expression qualify for the query result, and there is no ranking among the result documents. Query privacy is achieved by matching the encrypted or hashed keywords in place of their plaintext counterparts. Previous studies like [5], [10], [6] and [26] have proposed such schemes in public key and symmetric key settings. However, the Boolean model has been found wanting for general users; it is similarity retrieval which forms the basis of most modern document retrieval systems and Web search engines. Appendix B explains the distinction between Boolean and similarity retrieval models.

[22] proposes to project the documents and user queries from the original term space into a synthetic factor space formed with latent semantic indexing (LSI). As query processing involves similarity matching in the factor space, the server is oblivious of the actual term composition of the queries and result documents. However, LSI is known to perform well only for small homogeneous corpora [13], and is not suitable for large document collections that span multiple subject domains. Hence there is still a need for a privacy-preserving scheme that works with the conventional similarity retrieval mechanism involving an inverted index implementation.

TrackMeNot [12] protects user queries from search engines by hiding them among randomly generated 'ghost' queries. The authors conceded that the ghost queries often can be ruled out easily because their term combinations are not meaningful.

In [19], Murugesan and Clifton proposed to construct static groups of canonical queries, such that the queries in each group cover diverse topics. At runtime, a user query is substituted by the closest canonical query, while the other queries in the same group serve as cover queries to mask the user intent. The query groups are constructed by (a) mapping the dictionary terms into a 30-factor space with LSI; (b) forming canonical queries from terms that are in close proximity of each other in the factor space using a kd-tree nearest neighbor retrieval; and (c) selecting canonical queries with similar popularity from different parts of the factor space.

Our solution improves upon Murugesan and Clifton's scheme in several ways. First, in practice only a very small subset of term combinations can be materialized as canonical queries. This may be fine for Web search where queries are generally short, but not so for general text search. For example, the ad-hoc queries in TREC-2 and TREC-3 [27] contain up to 20 terms, and query expansion [23, 28] can produce even longer queries. In contrast, our approach of injecting decoy terms into each query does not suffer that restriction, as demonstrated in Section 5. Second, LSI exploits the heuristic that related terms tend to co-occur in documents; the strength of that heuristic depends on the corpus, and is not likely to capture all possible word relations. Moreover, LSI works best with 200 to 350 factors [9], whereas multi-dimensional index trees for finding nearest neighbors do not scale much beyond 10 dimensions [15]. We avoid these pitfalls by building instead on the WordNet database. Finally, substituting the user query with a canonical query affects precision-recall performance, as demonstrated in [19]. In contrast, our solution preserves the quality of the query results.

2.2 System Model

A search engine on a text corpus $\mathcal{D}$ typically employs an inverted index to process queries. The inverted index comprises a dictionary $\mathcal{T}$ of search terms, together with an inverted list for each term. The inverted list L_i for a term $t_i \in \mathcal{T}$ contains a sequence of $\langle d_j, p_{ij} \rangle$ pairs, where $p_{ij} \in \mathbb{R}$ is the impact of t_i in document $d_j \in \mathcal{D}$. If t_i appears in d_j, then $p_{ij} > 0$; otherwise $p_{ij} = 0$. For compactness, the documents d_j for which $p_{ij} = 0$ are not listed in L_i explicitly. In other words, if a d_j does not have an entry in L_i, then $p_{ij} = 0$. Appendix B elaborates on the index, including the derivation of p_{ij}.

For a query q with search terms $\{t_i\}$, the relevance of a document d_j is $S_{d_j,q} = \sum_{t_i \in q} p_{i,j}$. Consequently, only documents that appear in the inverted lists L_i's can accumulate a positive score, and be deemed relevant to the query. This means that query processing involves exactly the inverted lists for the search terms.

The inverted lists are assumed to be stored in plaintext on the search engine. We decided against encrypting the inverted lists because that would prevent the search engine from computing the relevance score of the documents, thus pushing the query processing function to the user. In any case, encryption would be ineffective here, because there are many clues that the search engine may exploit to map the encrypted inverted lists to their plaintext counterparts, including the varying length and access frequency of the lists.

We adopt the notation in Table 1 through the rest of the paper. The symbols are explained as they are used.

3. EMBELLISHING USER QUERIES WITH DECOY TERMS

This paper studies the problem of safeguarding the privacy of user queries in a similarity text search engine, so that an adversary is not able to observe or deduce the topic that is being searched on. We focus on protecting against the search engine, which constitutes the most powerful potential adversary because it possesses

Symbol	Meaning
N	# terms in the dictionary
t_i	The i-th term in the dictionary
L_i	The inverted list for t_i
d_j	The j-th document in the corpus
$score_j$	Relevance score of d_j
#Bkts	# buckets to map the dictionary terms to
BktSz	# terms per bucket
SegSz	# terms per segment
$E(.)$	Benaloh additively homomorphic encryption [4]
$KeyLen$	Length of cryptographic key for $E(.)$ (# bits)

Table 1: Notation

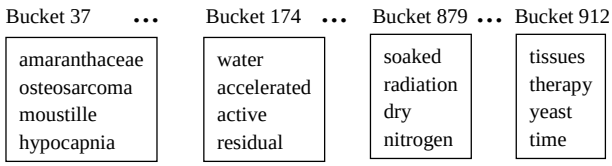


Figure 1: Illustration of Bucket Organization

the most information – it hosts the data structures, and it executes the query evaluation algorithms. We exclude tampering concerns, which have been addressed extensively in the context of query result authentication, e.g., [21]. Our problem is also orthogonal to user identity related privacy, which may be mitigated through query log anonymization [1], or by letting users connect to the search engine through an anonymous network such as mix-net [7] or Tor [8].

Conceptually, we aim to group the inverted lists of the dictionary terms in buckets, such that: (i) The terms within the same bucket are approximately equal in specificity, but differ significantly in meaning. For example, the terms in bucket 37 of Figure 1 are all very specific, whereas bucket 174 contains general terms. (ii) Given any two buckets, the semantic distance between the pair of terms in every slot i of the buckets are similar. Referring to the example in Figure 1, the distances between the corresponding terms in buckets 37 and 879 (i.e., {'amaranthaceae', 'soaked'}, {'osteosarcoma', 'radiation'}, {'moustille', 'dry'}, and {'hypocapnia', 'nitrogen'}) are roughly the same.

Given a query, all the terms that share the same bucket as any of the search terms are added as decoys. Therefore a high-specificity search term will always bring in the same set of decoy terms that are also specific, and a pair of semantically related search terms will attract pairs of decoy terms that are themselves semantically related. The net result is to cover the genuine search terms with decoy terms on other plausible topics. Referring to our running example in Figure 1 again, the user query 'osteosarcoma accelerated radiation therapy' relating to bone cancer would ideally be expanded with the decoy phrases 'amaranthaceae water soaked tissues', 'moustille active dry yeast' and 'hypocapnia residual nitrogen time', covering plant diseases, wine making and diving, respectively. After expansion, the terms in the query are permuted randomly, to raise the difficulty of isolating the genuine terms. Consequently, there is anonymity and plausible deniability for the user intent.

Obviously, constructing a bucket organization like the one in Figure 1 requires knowledge of term semantics and associations. As a realistic dictionary could easily exceed a hundred thousand distinct terms, organizing the buckets manually from scratch would be extremely tedious. In this work, we exploit the database of term associations in WordNet [18]. We could augment the database with relations extracted from text corpora [11] or the Web [25]; that is discussed briefly in Appendix C, and left for future work.

We begin by analyzing the rationale for our query embellishment approach in Section 3.1. Section 3.2 then gives a brief description of the WordNet database and the determination of term specificity. Following that, we propose a method for sequencing the dictionary in Section 3.3. The word sequence is then used in Section 3.4 to generate the buckets. The effectiveness of our bucket organization is evaluated in Section 5.1. For general corpora, it suffices to deploy the generated buckets directly; for sensitive applications where privacy is paramount, the buckets could be finetuned manually.

3.1 Rationale for Query Embellishment

Let $s = \langle q_1, q_2, \ldots, q_n \rangle$ be a sequence of n search queries issued by the user. Assume that q_i ($i \in [1, n]$) contains m_i keywords, i.e., $q_i = \{t_{i1}, t_{i2}, \ldots, t_{im_i}\}$. Our approach replaces each genuine term t_{ij} with a set (bucket) B_{ij} of terms, such that (i) $t_{ij} \in B_{ij}$, and (ii) $\forall t_{ij} = t_{xy}, B_{ij} = B_{xy}$ (each search term always brings in the same bucket). Upon observing $B_{i1}, \ldots, B_{im_i}$, the adversary knows that q_i must be in the set $Q_i = \{\{t'_{i1}, t'_{i2}, \ldots, t'_{im_i}\} \mid \forall j \in [1, m_i], t'_{ij} \in B_{ij}\}$. Let S denote the set of possible query sequences that can be formed by the queries in $Q_1 \times Q_2 \times \ldots \times Q_n$, i.e., $S = \{\langle q'_1, q'_2, \ldots, q'_n \rangle \mid \forall i \in [1, n], q'_i \in Q_i\}$. For any sequence $s' \in S$, let $\alpha(s')$ denote the adversary's prior belief in the event that s' is the sequence of queries issued by the user. Then, the adversary's posterior belief $\beta(s')$ in the same event is

$$\beta(s') = \frac{\alpha(s')}{\sum_{s^* \in S} \alpha(s^*)}. \quad (1)$$

In deciding which query sequence in S is the one issued by the user, the adversary would pick s' with $\beta(s')$ probability. Accordingly, we may quantify the privacy risk $risk(\{B_{ij}\})$ posed by our bucket organization $\{B_{ij}\}$ as the expected similarity between the sequence picked by the adversary and the genuine sequence s. Thus

$$risk(\{B_{ij}\}) = \sum_{s' \in S} \beta(s') \cdot sim(s', s), \quad (2)$$

where $sim(s', s)$ is a measure of the semantic similarity between s' and s, i.e., $sim(s', s)$ is large when s' and s are similar.

Ideally, we should construct $\{B_{ij}\}$ in a way that minimizes $risk(\{B_{ij}\})$. The computation of $risk(\{B_{ij}\})$, however, is practically difficult (if not impossible) due to two reasons. First, $risk(\{B_{ij}\})$ depends on the adversary's prior beliefs $\alpha(s')$ for all $s' \in S$, but $\alpha(s')$ is not known in advance and may vary among different adversaries. Second, although there exist standard methods for calculating the semantic distance between two queries (e.g., Formula 3 in Appendix B), quantifying the semantic similarity between two query sequences is an open problem, due to the complex correlations among the terms across queries.

Nonetheless, from Equation 2 we can make two important observations on how to lower $risk(\{B_{ij}\})$. First, $risk(\{B_{ij}\})$ decreases with $sim(s', s)$ for any query sequence q' in $S \setminus \{s\}$. In other words, $risk(\{B_{ij}\})$ is reduced if S encompasses a large number of query sequences with significant semantic differences from s. Intuitively, this can be achieved if each B_{ij} contains a semantically diverse set of terms, in which case the combinations of those terms lead to queries with drastically different meanings, thus decreasing the similarly among the query sequences in S. Second, $risk(\{B_{ij}\})$ is small when $\beta(s')$ is large for those sequences $s' \in S$ that are not similar to s. That is, we should ensure that the "camouflage" query sequences in S look as realistic as possible to the adversary. This, as explained in Section 1, may be attained only if each embellished query contains terms that are as specific as those in the genuine query. The above analysis justifies our solution approach to construct buckets in such a way that each B_{ij} contains terms with similar specificity but semantically different meanings.

3.2 Deriving Term Specificity from WordNet

Our bucket organization begins with a thesaurus. In this work, we pick the well-known WordNet [18]. It tags every term with one or more senses or meanings. Each sense has a corresponding *synset* which tracks the terms that share the meaning, and also its relationship with other synsets. For example, the term 'privacy' has two senses, one synonymous with 'seclusion', the other with 'secrecy' and 'concealment'. Relationships between synsets include:

- hypernym and hyponym, roughly corresponding to generalization and specialization;
- holonym and meronym, roughly corresponding to containment versus part-of; and
- topic and usage domain-membership.

In addition, antonyms and derivational relationships (e.g. 'man' and 'manhood') are identified.

The relationships in WordNet offer a basis for determining term specificity: The most general are those belonging to synsets that have no hypernyms. The more specific a term, the larger its number of levels of hypernyms. We represent the specificity of a term as a non-negative integer, determined as the length of the shortest path from the term's synset to a root in its hypernym hierarchy.

We focus on the part of WordNet involving nouns. (The other parts contain verbs, adjectives, and adverbs.) There are altogether 117,798 nouns that map to 82,115 synsets. Figure 2 shows that the specificity ranges from 0 to 18, with about one-third of the terms having a specificity value of 7. There is actually one synset with a specificity value of 0, and another 4 synsets with specificity of 1, although the corresponding bars are too short to notice in the figure.

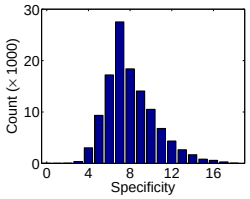


Figure 2: Distribution of Term Specificity

Another common approximation for term specificity is document frequency, defined as the percentage of documents in a corpus that contain that term. Studies such as [14] have reported a high correlation between the two methods. We adopt the hypernym method in this paper, because it has the advantage of being corpus-independent. If so desired, our bucket generation scheme will also work with the document frequency method.

3.3 Sequencing the Dictionary

From the WordNet database, Algorithm 1 produces term sequences in which related terms are clustered near each other. We process the synsets generally in decreasing number of relationships. The synsets with high connectivity (with other synsets) contain terms that are semantically rich. These synsets are used as seeds to pull related terms into the sequences. (The algorithm does not assume the existence of a root in the synset relation graph, from which we can initiate a depth-first or breadth-first traversal.)

For each synset, if the terms within it span multiple existing sequences, they are amalgamated into a longer sequence. If none of the terms in the synset have been encountered before, a new sequence is created. After adding the new terms from the current synset to the sequence, we go on to process its related synsets in order of closeness. This pulls in the derivationally related terms, antonyms, hyponyms, hypernyms, meronyms, followed by holonyms.

Algorithm 1 Sequence the terms in the dictionary

ProcessSynset(synset ss)
1: **if** the terms in ss appear in multiple existing sequences **then**
2: Concatenate these sequences.
3: Let sq denote the concatenated sequence.
4: **else if** none of the terms in ss is in an existing sequence **then**
5: Start a new sequence sq.
6: **else** one of the terms in ss is in an existing sequence
7: Let sq be that existing sequence.
8: Append the unprocessed terms in ss to sq.
9: Mark all the terms in ss as processed.
10: Mark ss as processed.
11: Return the sequence sq.

SequenceVocab(WordNet $wndb$)
12: Order the synsets in decreasing number of relationships.
13: Mark all the synsets as unprocessed.
14: Mark all the terms as unprocessed.
15: Let $SeqSet = \emptyset$.
16: **for all** unprocessed synset ss **do**
17: Let sq = ProcessSynset(ss); insert sq into $SeqSet$.
18: **for all** related synsets ss' in the order of derivational relations, antonyms, hyponyms, hypernyms, meronyms and holonyms **do**
19: Append any of the terms t in ss' to sq.
20: Mark t as processed.
21: Let sq = ProcessSynset(ss'); insert sq into $SeqSet$.
22: Return all the term sequences in $SeqSet$.

We skip the topic and usage domain memberships because these word associations tend to be less direct. The procedure is repeated until all the synsets, and hence all the terms, have been processed.

Running on WordNet, the algorithm groups all the 117,798 nouns into one long sequence, as they ultimately generalize to the same word – 'entity'. Below are a few snippets of the sequence:

- ... 'family tetragoniaceae', 'carpetweed family', 'amaranthaceae', 'family amaranthaceae', 'amaranth family', 'batidaceae', ...
- ... 'myosarcoma', 'neurosarcoma', 'malignant neuroma', 'osteosarcoma', 'osteogenic sarcoma', 'rhabdomyosarcoma', 'rhabdosarcoma', ...
- ... 'hypercapnia', 'hypercarbia', 'hypocapnia', 'acapnia', 'asphyxia', 'oxygen debt', 'hyperthermia', 'hyperthermy', ...
- ... 'foreign terrorist organization', 'terrorism', 'act of terrorism', 'terrorist act', 'abu hafs al-masri brigades', 'abu sayyaf', 'bearer of the sword', 'aksa martyrs brigades', ...

Evidently, the algorithm is effective in clustering related terms.

3.4 Bucket Formation

Next, we concatenate the term sequences generated by Algorithm 1 into one long sequence, and systematically assign its terms into buckets. Algorithm 2 gives the bucket formation procedure.

Suppose we want a bucket size BktSz, for some input parameter $1 \leq$ BktSz $\leq N/2$ where N is the total number of terms in the dictionary. The number of buckets #Bkts = N/BktSz. Since the terms within a bucket are meant to provide cover for each other, we simply assign the terms at positions 1, #Bkts+1, 2×#Bkts+1, ..., (BktSz-1)×#Bkts+1, in the sequence to slots 1, 2, ..., BktSz of bucket 1, respectively; then populate bucket 2 with the terms at positions 2, #Bkts+2, 2×#Bkts+2, ..., (BktSz-1)×#Bkts+2; and so on. Figure 3 illustrates the procedure. It has the advantage of maximizing the semantic diversity within each bucket. In addition, for any two buckets, the distance (and hence semantic difference)

Algorithm 2 Form buckets from the sequenced dictionary terms

GenerateBuckets(TermSequences {sq}, BktSz, SegSz)

1: Concatenate the input sequences into one long term sequence.
2: Let N = length of the concatenated sequence.
3: Set #Seg = N/SegSz.
4: Split the term sequence into equal segments, $S_1, S_2, \ldots, S_{\#Seg}$.
5: Sort the terms within each segment in decreasing specificity.
6: **for** i=1 to $N/$(BktSz×SegSz) **do**
7: ActiveSeg = $\emptyset$.
8: **for** j=1 to BktSz **do**
9: Register $S_{(j-1)N/(BktSz \times SegSz)+i}$ in ActiveSeg.
10: **for** j=1 to SegSz **do**
11: Create a new bucket B.
12: Insert into B the term at position j of each segment in ActiveSeg.
13: Output bucket B.

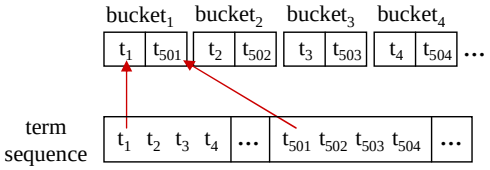


Figure 3: Bucket Formation – 1st Try ($N = 1000$, BktSz=2)

between the terms in their i-th slot is constant for $1 \leq i \leq$ BktSz. However, the terms within a bucket could vary widely in specificity, which is not desirable as explained at the beginning of this section.

To moderate the term specificity within buckets, we allow neighboring buckets to exchange the terms in their respective ith slots, on the condition that terms with the same specificity retain their original relative order. This is illustrated in Figure 4. As implemented by lines 3 to 5 of Algorithm 2, the procedure effectively allows term swapping among every batch of $1 \leq$ SegSz $\leq N/$BktSz consecutive buckets. In particular, we split the term sequence into #Seg=N/SegSz segments, order the terms by their specificity values (determined in Section 3.2), before populating each bucket with BktSz terms that are equally spaced in the modulated sequence. The result is that the initial buckets in each batch get the more specific terms, whereas the general terms gravitate to the later buckets.

To illustrate, below are a few of the buckets generated with BktSz=4 and SegSz=512; the numbers in parenthesis indicate the specificity of the corresponding terms.

- Bucket 1419: 'sir thomas wyatt' (7), 'hypocapnia' (6), 'ectozoon' (7), 'fool's gold' (6).
- Bucket 2076: 'love knot' (10), 'mainspring' (9), 'osteosarcoma' (14), 'yellow-breasted bunting' (14).
- Bucket 7927: 'huntsville' (9), 'pigeon loft' (7), 'brama' (7), 'terrorism' (9).
- Bucket 8106: 'smyrna' (7), 'lut desert' (6), 'acipenser' (7), 'abu sayyaf' (7).
- Bucket 14114: 'sign of the zodiac' (5), 'amaranthaceae' (8), 'american chestnut' (11), 'family eschrichtiidae' (7).

Thus, for example, a user query on {'abu sayyaf', 'terrorism'} would be embellished with the other terms from buckets 8106 and 7927. Even if the adversary manages to group the terms in the embellished query correctly – a nontrivial task in general – he is still faced with the combinations of {'smyrna', 'huntsville'}, {'lut desert', 'pigeon loft'}, and {'acipenser', 'brama'}, all of which are also plausible topics that explain the user's interest. Of course,

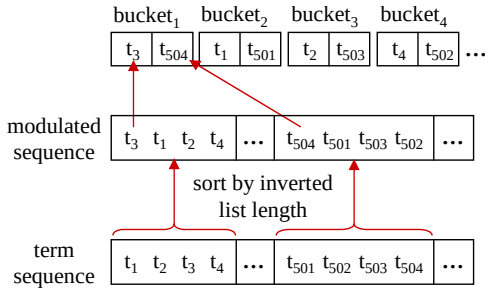


Figure 4: Final Bucket Formation (N=1000,BktSz=2,SegSz=4)

our bucket organization does not guarantee that all combinations of decoy terms are equally meaningful; indeed, that is probably impossible. Realistically, we can only aim to mask the user interest with as many meaningful decoys as possible. We will empirically measure the effectiveness of our proposed solution in Section 5.

Finally, our bucket formation algorithm requires two input parameters. The first, SegSz, balances between the two privacy risks: On one hand, a high SegSz leads to larger segments that provide flexibility to even out the specificity of the terms within each bucket. This reduces the risk from recurring high-specificity search terms in a query sequence. On the other hand, with finer segments the semantic distance between the terms occupying the same slot i in two buckets are likely to be similar across $1 \leq i \leq$ BktSz. The second parameter BktSz determines the bucket size. Since the search engine retrieves all the terms residing in the same bucket as any of the search terms, a larger bucket increases the number of decoy terms and thus privacy protection, at the expense of a heavier performance penalty. The impact of SegSz and BktSz on our bucket formation will also be investigated through experiments.

4. PRIVATE RETRIEVAL SCHEME

In this section, we introduce our private retrieval scheme for the search engine to compute the document relevance scores from only the genuine search terms in the query. The scheme includes procedures for query formulation, server processing, and post filtering.

Algorithm 3 User masks the genuine terms in the search query

Input: A set of genuine search terms t_i's.
Output: Embellished query q.

1: Let $q = \emptyset$.
2: **for all** genuine search terms t_i **do**
3: Let Bkt be the host bucket for t_i.
4: **for all** $t_j \in$ Bkt **do**
5: **if** $t_j == t_i$ **then** let $u_j = 1$
6: **else** let $u_j = 0$.
7: Let $E(u_j) = g^{u_j}\mu^r \bmod m$.
8: Insert $\langle t_j, E(u_j)\rangle$ into q.

Algorithm 3 lists the query formulation steps. From each bucket containing a genuine search term, the user client injects all the other terms in that bucket into the query as decoys. A query is unlikely to comprise only terms from the same bucket, because by construction the terms in a bucket relate to diverse topics; in case it happens, we will pick other buckets randomly to embellish the query. The query q is a set of $\langle t_j, E(u_j)\rangle$ pairs where $u_j = 1$ if t_j is a genuine term, and $u_j = 0$ for decoy term t_j. $E(u_j) = g^{u_j}\mu^r \bmod m$ is Benaloh's additively homomorphic encryption function [4], with suitably chosen parameters m, g, r, and random $\mu \in \mathbb{Z}_m^*$; a detailed definition of the encryption function is given in Appendix A. The embellished q is then submitted to the server.

Algorithm 4 Query processing by the search engine

Input: Embellished query q.
Output: A set R of candidate result documents, with their encrypted relevance scores.

```
1: Let R = ∅.
2: for all ⟨t_i, E(u_i)⟩ ∈ q do
3:     for all ⟨d_j, p_ij⟩ ∈ L_i do
4:         if ∃⟨d_j, E(score_j)⟩ ∈ R then
5:             E(score_j) = E(score_j) × E(u_i)^{p_ij}.
6:         else
7:             Insert ⟨d_j, E(u_i)^{p_ij}⟩ into R.
```

Algorithm 4 gives the query processing procedure that is executed by the search engine. The algorithm iterates through the (genuine and decoy) terms in the embellished query q. For each term t_i, the associated inverted list L_i contains the candidate result documents. For each such document d_j, its encrypted relevance score $E(score_j)$ is incremented by $E(u_i)^{p_{ij}}$ or, equivalently, $E(u_i \times p_{ij})$ as $E(.)$ is additively homomorphic.[1] For decoy terms t_i, $u_i = 0$ so $E(u_i)^{p_{ij}}$ changes only the ciphertext $E(score_j)$ but not the actual $score_j$. Consequently, $score_j$ accumulates only the impact values p_{ij} of the genuine terms t_i in q. The candidate result R is returned to the user.

As all the terms in the same bucket as any query term are injected into the query in Algorithm 3, the search engine should store the inverted lists for the terms of a bucket in common disk block(s). This allows Algorithm 4 to fetch the inverted lists of an entire bucket's worth of terms in one operation, thus lowering I/O costs.

Algorithm 5 Post filtering by the user

Input: A set R of candidate result documents, with their encrypted relevance scores.
Output: An ordered list of result documents.

```
1: for all ⟨d_j, E(score_j)⟩ ∈ R do
2:     Decrypt E(score_j) to recover score_j.
3: Sort the entries in R in decreasing relevance score_j.
4: Return the d_j of the top entries.
```

Upon receiving the result R, the user decrypts the relevance score of the candidate documents, and identifies those with the highest scores as result documents for the query. The procedure is given in Algorithm 5.

CLAIM 1. *Our retrieval scheme does not interfere with the relevance ranking of the search engine.*

Rationale Consider a result document d_j. As explained in Section 2.2, d_j must appear in the inverted list L_i of some genuine search term t_i. Since Algorithm 4 processes the inverted list of every term in the embellished query q, including L_i, the procedure is guaranteed to find d_j. With $u_i = 1$ for genuine term t_i and $u_i = 0$ for decoy term t_i, the score $E(score_j) = \sum_{t_i \in q} u_i \times p_{ij}$ is accumulated from the genuine terms. Therefore the user can determine correctly d_j's ranking with respect to the other documents. □

Alternate Retrieval Method: For retrieving the buckets that contain the search terms, we will benchmark our retrieval scheme against the Private Information Retrieval (PIR) method from [16]; the protocol is described briefly in Appendix A. With this method, each bucket Bkt is treated as a private "database", and the inverted lists within a bucket must be padded to exactly the same length. The database is treated as a matrix, with the columns corresponding to the inverted lists, and the ith row storing the ith bit of the lists.

Suppose we have a query that requires one genuine term in Bkt, with the remaining terms in the bucket serving as decoys. We send to the server a row vector, filled with QR (quadratic residue) values except for the column holding the genuine term which has a QNR (quadratic non-residue) value. The output is a column of QR/QNR values, corresponding to 0/1 bits in the target inverted list respectively. The generation of the output involves all the terms in the bucket, so the server cannot identify which term is being retrieved because it is computationally infeasible to differentiate QR from QNR without knowledge of the secret key.

Denoting the length (in bytes) of an inverted list L_i by $|L_i|$ and using a key size of $KeyLen$ bits, the return message to the user has a size of $KeyLen \times \max_{L_i \in \text{Bkt}} |L_i|$ bytes. This protocol can retrieve only one list per execution. Thus, if a query contains multiple genuine terms from the same bucket, their inverted lists have to be fetched one at a time.

[1]The cryptographic operation is defined only for non-negative integers p_{ij}. We assume that the impact values p_{ij}'s are discretized so that $p_{ij} \in \mathbb{N} \bigcup \{0\}$, as explained in [29].

5. EMPIRICAL EVALUATION

In this section, we present a two-part empirical evaluation of our solution. The first part focuses on the effectiveness of the decoy terms added to mask the user intent, and seeks to understand how the bucket size and segment size should be chosen. The second part of the evaluation quantifies the performance of the search engine that executes our private retrieval scheme.

5.1 Evaluation of Privacy Risk

As presented in Section 3, our solution (denoted as "Bucket") embellishes a user query with decoy terms from the same buckets as the genuine terms. We judge the plausibility of the resulting cover, compared to that provided by the same number of random decoy terms (denoted by "Random"). The evaluation metrics are:

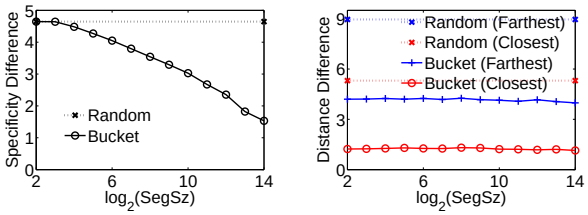
- Difference in intra-bucket specificity values. This measures the difference between the highest and lowest specificity values of the terms in the same bucket. A small difference is desirable, as it means that each genuine search term would attract decoy terms that are similarly specific, thus countering any inference from recurring high-specificity terms in a search session.
- Difference in inter-bucket distances. Recall that the terms in an embellished query are permuted randomly, to deter the adversary from recovering the logical grouping among the terms. Here, we assume conservatively that the adversary succeeds nevertheless. To counter any inference from them, all the groups should exhibit similar semantic patterns among their terms. Specifically, when a user queries for two terms t_1 and t_2 from a pair of buckets, the semantic distance $dist$ between t_1 and t_2 should be similar to the distance $dist'$ between the other pairs of terms from the two buckets. Again, we would like the distance difference $|dist - dist'|$ to be small, so that related query terms would evoke similarly related decoy terms. The smallest $|dist - dist'|$ is reported as 'closest cover', and the largest as 'farthest cover'.

We define the semantic distance between two terms t_1 and t_2 as the length of the shortest path between their corresponding synsets. We assign a weight of 1 to a hypernym-hyponym relationship, and weights of 0.5, 2 and 3 for antonym, holonym-meronym, and domain-member relationships, respectively, to reflect the different degrees of association that they represent. For a given bucket organization, we pick the terms in slot i of a pair of randomly selected buckets as query terms, with i varying uniformly between 1 and BktSz, and measure their semantic distance against that of the pairs of decoy terms at the other slots. (We pair the terms at the same slot

in the buckets because they are generally closer to each other in the term sequence, and hence are closer semantically, compared to word pairs across slots.) This measurement is repeated 1,000 times, and the average distance difference is reported.

For our experiments, we run the WordNet database through Algorithms 1 and 2 to generate the buckets. We first vary SegSz while keeping BktSz fixed at 4. Figure 5(a) gives the intra-bucket specificity differences, averaged over all the buckets. As expected, a larger segment size lowers the specificity differences by providing more leeway to swap terms within segments. This explains the lower specificity values compared to random decoy terms.

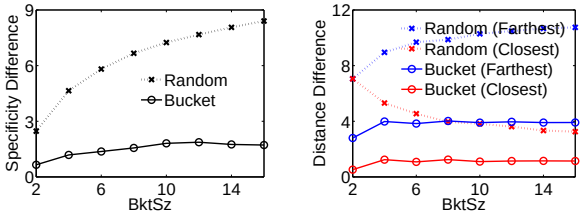
The corresponding distance differences are given in Figure 5(b). The results show that, on the average, the semantic distance of the closest cover differs from that of the user query by just one hypernym-hyponym hop, whereas the farthest cover is about 4 times longer. Surprisingly, the distance differences appear to be little affected by SegSz. On closer examination, we discover this is because line 5 of Algorithm 2 preserves the relative order among terms that tie on specificity, so a large SegSz causes entire synsets of terms to be reordered (by specificity), and allows closely related terms to remain clustered together in each segment. Again, the cover provided is much better than that of random decoy terms.



(a) Specificity Difference (b) Distance Difference

Figure 5: Effect of SegSz on Bucket Formation (BktSz = 4)

Since a larger segment size improves the specificity difference without worsening the difference in semantic difference, for the next experiment we maximize the segment size to N/BktSz while varying the bucket size. As shown in Figure 6, the specificity difference starts low. The implication is that small buckets tend to introduce decoys that match the genuine terms in specificity. As the bucket size increases, it becomes more difficult to maintain uniformity in term specificity within every bucket; nonetheless, the specificity difference is still small relative to random decoy terms.



(a) Specificity Difference (b) Distance Difference

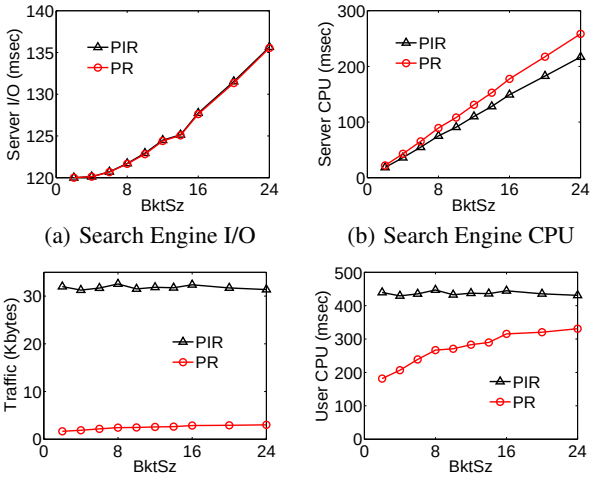
Figure 6: Effect of BktSz on Bucket Formation

5.2 Retrieval Performance

Having assessed the effectiveness of our decoy selection mechanism, we now investigate the performance of our Private Retrieval (PR) scheme. The experiment set-up is as follows:

- **Algorithms:** We benchmark PR against PIR. The two schemes are described in Section 4.
- **Dataset:** The corpus for the experiments is WSJ, comprising 172,961 articles published in the Walls Street Journal from December 1986 to March 1992. The combined size of the articles is around 513 Mbytes. We first load the corpus into the Lucene search engine [17], which parses the documents, performs stopword (common words like 'the' and 'a' that are not useful for differentiating between documents) removal but not stemming [2], and creates an inverted index. Next, we write out Lucene's index into a dictionary of terms, along with an inverted list for each term. This dictionary is intersected with the WordNet database, giving us a list of searchable terms with known semantic relationships (from WordNet). Using our mechanism in Section 3, the search terms are grouped into buckets.
- **Workload:** We form queries from the search terms randomly. The number of query terms is an experiment parameter. The queries are embellished systematically, according to Algorithm 3, before submission to the search engine.
- **System configuration:** Our search engine runs on a Redhat Linux box with dual Intel Xeon 3GHz CPUs, 1 Gbyte RAM, and a Seagate ST973401KC 73 Gbytes hard disk formatted with default 1-Kbyte blocks. The user machine has an Intel 1.33GHz CPU with 768 Mbytes of RAM, and runs Ubuntu Linux.
- **Metrics:** The performance metrics include the I/O and CPU costs of the search engine, the network traffic volume, and the user computation time. Each result is averaged over 1,000 queries.

For a start, we fix the query size at 12 terms and vary the bucket size. The results are plotted in Figure 7. PIR appears to enjoy a slight edge in server resource consumption; its I/O costs are virtually the same as PR's, while its server protocol imposes roughly 16% less CPU demand than the homomorphic encryption operations performed in PR. However, PR's server-user communication is an order of magnitude lower, whilst its saving in user computation ranges from 60% for BktSz=2, to 23% for BktSz=24. In particular, PR's increase in communication cost in Figure 7(c) is only sublinear to the bucket size, because its output corresponds to the union of the documents in the queried buckets.



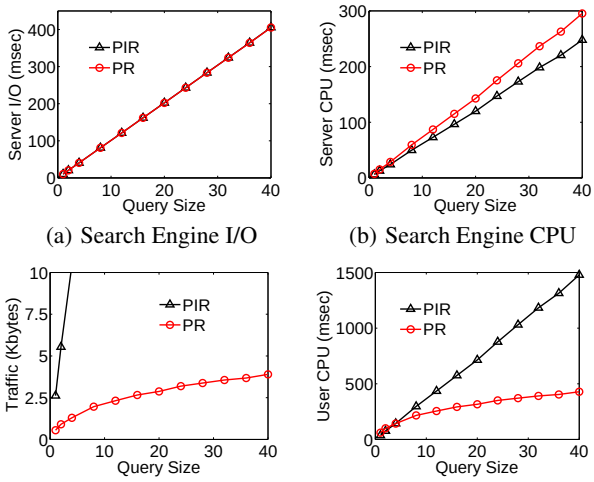
(a) Search Engine I/O (b) Search Engine CPU

(c) Network Traffic (d) User CPU

Figure 7: Performance Impact of BktSz

Next, Figure 8 profiles the sensitivity of the schemes to the query size, with the bucket size fixed at 8. Clearly, the disadvantages of PIR are exacerbated here, with its communication and user computation requirements both increasing linearly with the query size. In contrast, PR scales much more gracefully to long queries.

The experiment results clearly point to the superiority of PR. If most of the user queries comprise only a small number of terms, it is essential to adopt a large bucket size so that the embellished



(a) Search Engine I/O (b) Search Engine CPU

(c) Network Traffic (d) User CPU

Figure 8: Performance Impact of Query Size

queries include enough decoy terms to cover sufficiently diverse topics. On the other hand, if query expansion techniques (e.g., those reported in [23], [28]) are employed, the queries could well contain a few dozen search terms even before adding decoy terms. In both cases, the substantially lower network traffic and user computation imposed by PR would justify its server computation overheads. These advantages are particularly important in deployment scenarios where the communication channel may be a cellular network for which users are charged by traffic volume or the user protocol is executed within resource-constrained devices, whereas server computing resources can be provisioned much more easily.

6. CONCLUSION

This paper introduces a similarity text retrieval system that affords anonymity protection for the query terms, and hence the user intent, without compromising precision-recall performance. Our approach is to embellish each user query with decoy terms before submitting it to the search engine. Starting from a thesaurus, e.g., WordNet, we present a mechanism for selecting decoy terms that exhibit similar specificity spread as the genuine terms, yet point to plausible alternative topics. We also provide a novel retrieval scheme, using homomorphic encryption techniques, that enables the search engine to compute encrypted document relevance scores with respect to only the genuine search terms, but remain oblivious to their differentiation from the decoys. The effectiveness of our proposed methods are confirmed through experiments involving real datasets.

7. REFERENCES

[1] E. Adar. User 4xxxxx9: Anonymizing Query Logs. In *Query Log Analysis Workshop, WWW*, May 2007.

[2] R. Baeza-Yates and B. R. Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[3] M. Barbaro and T. Z. Jr. A Face Is Exposed for AOL Searcher No. 4417749. *The New York Times*, 9 August 2006.

[4] J. C. Benaloh. Dense Probabilistic Encryption. In *Workshop on Selected Areas of Cryptography*, May 1994.

[5] J. Bethencourt, D. Song, and B. Waters. New Constructions and Practical Applications for Private Stream Searching (Extended Abstract). In *IEEE Symposium on Security and Privacy*, May 2006.

[6] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public Key Encryption with Keyword Search. In *EUROCRYPT*, May 2004.

[7] D. L. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2), February 1981.

[8] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, August 2004.

[9] S. T. Dumais. Latent Semantic Indexing (LSI) and TREC-2. In *Second Text REtrieval Conference (TREC2), D. Harman, ed.*, March 1994.

[10] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword Search and Oblivious Pseudorandom Functions. In *Theory of Cryptography Conference*, Feburary 2005.

[11] T. Hasegawa, S. Sekine, and R. Grishman. Discovering Relations among Named Entities from Large Corpora. In *ACL*, July 2004.

[12] D. C. Howe and H. Nissenbaum. TrackMeNot. mrl.nyu.edu/ dhower/trackmenot/.

[13] P. Husbands, H. Simon, and C. H. Q. Ding. On The Use Of The Singular Value Decomposition For Text Retrieval. In *Proc. SIAM Computational Information Retrieval*, 2001.

[14] H. Joho and M. Sanderson. Document Frequency and Term Specificity. In *Recherche d'Information Assistée par Ordinateur Conference (RIAO)*, May 2007.

[15] F. Korn, B.-U. Pagel, and C. Faloutsos. On the 'Dimensionality Curse' and the 'Self-Similarity Blessing'. *IEEE TKDE*, 13(1), January 2001.

[16] E. Kushilevitz and R. Ostrovsky. Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In *FOCS*, October 1997.

[17] Lucene. Apache Lucene Search Engine. http://lucene.apache.org/java/docs/.

[18] G. A. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11), November 1995.

[19] M. Murugesan and C. Clifton. Providing Privacy through Plausibly Deniable Search. In *SDM*, April 2009.

[20] P. Paillier. Public-Key Cryptosystems based on Composite Degree Residuosity Classes. In *EUROCRYPT*, May 1999.

[21] H. Pang and K. Mouratidis. Authenticating Query Results for Text Search Engines. In *VLDB*, July 2008.

[22] H. Pang, J. Shen, and R. Krishnan. Privacy-Preserving, Similiarity-Based Text Retrieval. *ACM Transactions on Internet Technology*, 10(1), February 2010.

[23] Y. Qiu and H.-P. Frei. Concept Based Query Expansion. In *ACM SIGIR*, June 1993.

[24] S. E. Robertson, S. Walker, and M. Hancock-Beaulieu. Experimentation as a way of life: Okapi at TREC. *Information Processing and Management*, 36(1), 2000.

[25] B. Rozenfeld and R. Feldman. Self-Supervised Relation Extraction from the Web. *Knowl. Inf. Syst.*, 17(1), 2008.

[26] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *IEEE Symposium on Security and Privacy*, May 2000.

[27] TREC. Text REtrieval Conference. http://trec.nist.gov/.

[28] J. Xu and W. B. Croft. Query Expansion Using Local and Global Document Analysis. In *ACM SIGIR*, August 1996.

[29] J. Zobel and A. Moffat. Inverted Files for Text Search Engine. *ACM Computing Surveys*, 38(2), July 2006.

Appendix A. Cryptographic Primitives

A.1 Private Information Retrieval

The competitor to our proposed Private Retrieval scheme in the experiments is based on the Private Information Retrieval (PIR) construction by Kushilevitz and Ostrovsky (KO) in [16]. The PIR scheme is defined as follows.

Given n which is the product of two primes p_1 and p_2, a number $b \in \mathbb{Z}_n^*$ is a quadratic residue (QR) modulo n if $\exists w \in \mathbb{Z}_n^*$ such that $w^2 = b \bmod n$; otherwise b is a quadratic non-residue (QNR). Without knowledge of p_1 and p_2, it is computationally infeasible to distinguish a QR from a QNR.

Suppose that a server holds a private "database" B, organized as a $r \times c$ matrix of bits. A user, wishing to retrieve privately the bit $b_{xy} \in B$ in row x and column y of the database, executes the following protocol with the server:

- The user starts by picking two $(KeyLen/2)$-bit primes p_1 and p_2, then computes $n = p_1 p_2$. n is given to the server, while p_1 and p_2 are kept secret.
- The user generates c random numbers $q_1, \ldots, q_c$ such that q_y is a QNR and the other q_j's, $j \neq y$, are QRs. $q_1, \ldots, q_c$ are sent to the server.
- The server computes, for every row $1 \leq i \leq r$, a number $\gamma_i \in \mathbb{Z}_n^*$ as $\gamma_i = \prod_{j=1}^{c} v_{ij}$ where

$$v_{ij} = \begin{cases} q_j^2 & \text{if } b_{ij} = 0 \\ q_j & \text{if } b_{ij} = 1 \end{cases}$$

  $\gamma_1, \ldots, \gamma_r$ are returned to the user.
- With p_1 and p_2, the user can efficiently test whether γ_x is a QR. If so, he concludes that $b_{xy} = 0$; otherwise $b_{xy} = 1$.

A.2 Additively Homomorphic Encryption

Our Private Retrieval scheme in Section 4 builds on an additively homomorphic encryption function. Paillier's [20] and Benaloh's [4] are two of the well-known cryptosystems that provide additively homomorphic encryption. Our experiments employ the latter because it produces shorter ciphertexts and hence lower communication costs.

The Benaloh cryptosystem is constructed as follows for messages in $[0, r-1]$ where r is an integer. For key generation,

- Pick two large prime numbers p_1 and p_2 such that r divides $(p_1 - 1)$, r is co-prime with $(p_1 - 1)/r$, and r is co-prime with $(p_2 - 1)$.
- Set the modulus $n = p_1 p_2$.
- Select $g \in \mathbb{Z}_n^*$ such that $g^{(p_1-1)(p_2-1)/r} \bmod n \neq 1$.
- (n, g) is the public key, and (p_1, p_2) is the corresponding private key.

To encrypt a message $m \in \mathbb{Z}_r$,

- Choose a random $\mu \in \mathbb{Z}_n^*$.
- Set $E(m) = g^m \mu^r \bmod n$.

The random μ in the encryption function enables the same message m to map to different ciphertexts through different invocations of the function. This prevents an adversary from inferring m from knowledge of its frequency.

To recover a message m from its ciphertext $E(m)$, the decryptor exhaustively tests, for every $i \in \mathbb{Z}_r$, whether

$$(g^{-i} \cdot E(m))^{(p_1-1)(p_2-1)/r} = 1 \bmod n$$

The equation above only holds when $m = i$. Note that all $g^{-i} \bmod n$ can be pre-computed by the decryptor. With the baby-step giant-step algorithm, the decryption procedure requires $\mathcal{O}(\sqrt{r})$ time. However, for $r = 3^k$ for some $k \in \mathbb{N}$, [4] gives an optimized decryption procedure that requires only k modular exponentiations.

Given the ciphertext $E(m_1) = g^{m_1} \mu_1^r \bmod n$ and $E(m_2) = g^{m_2} \mu_2^r \bmod n$ for two messages m_1 and m_2, anyone can compute $E(m_1) \otimes E(m_2) = g^{m_1+m_2}(\mu_1 \mu_2)^r \bmod n = E(m_1 + m_2)$ to get the ciphertext of $(m_1 + m_2) \bmod r$. Therefore the cryptosystem is additively homomorphic.

Appendix B. Text Retrieval Models

This appendix gives a brief explanation of two classical text retrieval models that are related to existing work and our proposed solution.

B.1 Boolean Keyword Matching

The Boolean model for text retrieval by keyword matching is based on set theory and Boolean algebra. Suppose q is a query, in the form of a Boolean expression of keywords. Let q_{dnf} be the disjunctive normal form of q, and q_{cc} be a conjunct in q_{dnf}. The "similarity" between document d and query q is defined as:

$$S_{d,q} = \begin{cases} 1 & \text{if } \exists\ q_{cc} \in q_{dnf} \text{ such that } (\forall \text{ term } t \in q_{cc}, t \in d) \\ 0 & \text{otherwise} \end{cases}$$

The above function produces a binary decision without any notion of ranking. In other words, a document either matches the query or it does not; there is no *degree* of relevance that differentiates two matching documents. The limitation hinders good retrieval performance when users are not familiar with the exact terminology in the documents, and is especially problematic with casual users or large/heterogeneous corpora. This drawback of the Boolean model prompted the development of similarity-based retrieval systems.

B.2 Text Retrieval by Similarity

Let $\mathcal{D}$ denote a set of documents. Let $\mathcal{T}$ be the set of all terms (or keywords) in the dictionary for $\mathcal{D}$. A search engine on $\mathcal{D}$ and $\mathcal{T}$ takes a query q, which is a subset of $\mathcal{T}$ (i.e., $q \in 2^{\mathcal{T}}$), and returns a set of documents whose similarity scores are above a pre-defined threshold, e.g., the 20th highest score. Most text search engines rate the similarity of each document $d \in \mathcal{D}$ to a query q, based on these considerations:

- Terms that appear in many documents are given less weight;
- Terms that appear many times in a document are given more weight; and
- Documents that contain many terms are given less weight.

The considerations are encapsulated in a similarity function, which uses some composition of the following statistical values:

- $f_{d,t}$, the number of times that term t appears in document d;
- f_t, the number of documents that contain term t;
- N, the number of documents in the data set $\mathcal{D}$.

A similarity function that is effective in practice defines the score of a document d with respect to a query q, $S_{d,q}$, to be the cosine of the angle between the corresponding document vector and query vector in multi-dimensional term space. One of the most well-known variations is:

$$S_{d,q} = \frac{\sum_{t \in \mathbf{q}} w_{d,t} \cdot w_t}{W_d} \tag{3}$$

where $w_t = ln\left(1 + \frac{N}{f_t}\right)$, $w_{d,t} = 1 + ln(f_{d,t})$, and $W_d = \sqrt{\sum_{t \in d} w_{d,t}^2}$. Okapi [24] is another well-known scoring function. Similarity models have been studied extensively and proven successful in TREC experiments [29]. Note that our solution presented in this paper applies generally to similarity retrieval models that judge similarity from the query and document vectors, including Okapi.

Given a query, a straightforward evaluation algorithm is to compute $S_{d,q}$ for each document d in turn, and return those documents with the highest similarity scores at the end. The execution time of this algorithm is proportional to N, which is not scalable to large collections. Instead, search engines typically make use of an index that maps the search terms to the documents containing them. The most efficient index structure for this purpose is the *inverted index*. Our work uses the impact-ordered inverted index, one of the variations recommended in [29] which provides a comprehensive survey on search engine algorithms.

Impact-Ordered Inverted Index. The index consists of two components – a *dictionary* of terms and a set of *inverted lists*. The dictionary stores, for each distinct term t,

- a count f_t of the documents that contain t, and
- a pointer to the head of the corresponding inverted list.

The inverted list for a term t is a sequence of *impact pairs* $\langle d, p_{d,t} \rangle$ where

- d is the identifier of a document that contains t,
- $p_{d,t}$ is the associated *impact* of the term t in document d, defined as

$$p_{d,t} = \frac{w_{d,t} \cdot w_t}{W_d} \quad (4)$$

Note that w_t is independent of q. Moreover, each inverted list is sorted in decreasing $p_{d,t}$ values. Figures 9 and 10 give an example of an impact-ordered inverted index and an algorithm for finding the top-k matching documents with the inverted index, both adapted from [29]. The algorithm repeatedly pops the highest impact value from the inverted lists involved in a query, and accumulates the relevance score of the documents encountered. This continues until all the inverted lists are exhausted, at which point the documents with the highest cumulative scores are identified for the query result.

The similarity scoring model with the inverted index implementation are used extensively in modern document retrieval systems. They also form the foundation of Web search engines. That is why we pick the model as basis for our privacy-preserving text retrieval solution in this paper.

Appendix C. Merging Multiple Sources of Term Relations

Our decoy injection mechanism requires as input a thesaurus of term relations. For this paper, we use the database from WordNet [18]. As the term relations in this database have been derived manually, they are accurate but may not be comprehensive enough. Depending on the search application, it may be necessary to additionally incorporate domain-specific or emerging term relations. These can be obtained through relation extraction from text corpora [11] or the Web [25]. To combine the two sets of term relations, one way is to translate the various types of term relations in WordNet to appropriate numeric strengths; then, the extracted relations are also assigned ratings in the same strength scale according to their occurrence counts. Finally, line 18 of Algorithm 1 needs to be modified to iterate from the strongest term relations, down to some minimum strength threshold. The detailed procedure for merging multiple sources of term relations is left to a future extension of the paper.

Term t	f_t		Inverted List for t
and	1	$\mapsto$	$\langle 6, 0.159 \rangle$
big	2	$\mapsto$	$\langle 2, 0.148 \rangle$ $\langle 3, 0.088 \rangle$
dark	1	$\mapsto$	$\langle 6, 0.079 \rangle$
did	1	$\mapsto$	$\langle 4, 0.125 \rangle$
gown	1	$\mapsto$	$\langle 2, 0.074 \rangle$
had	1	$\mapsto$	$\langle 3, 0.088 \rangle$
house	2	$\mapsto$	$\langle 3, 0.088 \rangle$ $\langle 2, 0.074 \rangle$
in	5	$\mapsto$	$\langle 6, 0.159 \rangle$ $\langle 2, 0.148 \rangle$ $\langle 5, 0.088 \rangle$ $\langle 1, 0.088 \rangle$ $\langle 3, 0.088 \rangle$
keep	3	$\mapsto$	$\langle 5, 0.088 \rangle$ $\langle 1, 0.088 \rangle$ $\langle 3, 0.088 \rangle$
keeper	3	$\mapsto$	$\langle 4, 0.125 \rangle$ $\langle 5, 0.088 \rangle$ $\langle 1, 0.088 \rangle$
keeps	3	$\mapsto$	$\langle 5, 0.088 \rangle$ $\langle 1, 0.088 \rangle$ $\langle 6, 0.079 \rangle$
light	1	$\mapsto$	$\langle 6, 0.079 \rangle$
never	1	$\mapsto$	$\langle 4, 0.125 \rangle$
night	3	$\mapsto$	$\langle 5, 0.177 \rangle$ $\langle 4, 0.125 \rangle$ $\langle 1, 0.088 \rangle$
old	4	$\mapsto$	$\langle 2, 0.148 \rangle$ $\langle 4, 0.125 \rangle$ $\langle 1, 0.088 \rangle$ $\langle 3, 0.088 \rangle$
sleep	1	$\mapsto$	$\langle 4, 0.125 \rangle$
sleeps	1	$\mapsto$	$\langle 6, 0.079 \rangle$
the	6	$\mapsto$	$\langle 5, 0.265 \rangle$ $\langle 1, 0.263 \rangle$ $\langle 3, 0.263 \rangle$ $\langle 6, 0.159 \rangle$ $\langle 2, 0.148 \rangle$ $\langle 4, 0.125 \rangle$
town	2	$\mapsto$	$\langle 1, 0.088 \rangle$ $\langle 3, 0.088 \rangle$
where	1	$\mapsto$	$\langle 4, 0.125 \rangle$

Figure 9: Example of Impact-Ordered Inverted Index

To find the top k matching documents for a query q, using an impact-ordered inverted index.

(1) Fetch the first $\langle d, p_{d,t} \rangle$ entry in each query term t's inverted list.
(2) While there remain entries on any query term's inverted list,
  (a) Identify the inverted list entry $\langle d, p_{d,t} \rangle$ with highest $p_{d,t}$, breaking ties arbitrarily.
  (b) If d has not been encountered before, create an accumulator A_d and initialize it to zero.
  (c) $A_d \leftarrow A_d + p_{d,t}$.
  (d) Fetch the next entry in term t's inverted list.
(3) Identify the k largest A_d values and return the corresponding documents.

Figure 10: Query Evaluation on Impact-Ordered Inverted Index