Tiptoe++: A Learning-Augmented Protocol for Private Vector Search

Xiaochen Zhu Vihan Lakshman Dewei Feng Sarah Mokhtar
Massachusetts Institute of Technology
{xczhu, vihan, dewei, sarah04}@mit.edu

Abstract

We study the problem of *private vector search*. In this setting, we have a collection of vectors (such as text or image embeddings) stored on a third-party *server* and a *client* can query the server to receive the top k approximate nearest neighbor vectors in the collection. For privacy, we additionally require that the server learns *nothing* about the contents of the client's query while nevertheless returning relevant results. This problem has received heightened attention in recent years, but existing private vector database protocols all sacrifice substantial amounts of either performance or search quality. In this paper, aim to address this gap by designing a private vector search protocol that strikes a better balance between efficiency and search quality. To that end, we introduce Tiptoe++, a private search system that enhances the existing Tiptoe protocol, via machine learning. As an early proof-of-concept of our solution, we implemented Tiptoe++ on top of the Tiptoe system and found that our solution produced up to a 20-point improvement in Recall@10 and MRR@10 over Tiptoe on a million-scale sample of the popular MS MARCO Web Search Benchmark. For future work, we hope to refine our techniques and implementation and evaluate our proposed method over a larger set of retrieval benchmarks in the private setting and compare with more recent works.

1 Introduction

With the proliferation of neural representation learning, vector search has emerged as the state-of-the-art in information retrieval systems, typically outperforming classical keyword search by large margins. Recently, there has been heightened interest in developing protocols for *private* vector search that hide user queries from untrusted search engine servers while still enabling effective retrieval.

While several systems for private vector search have previously been proposed in the literature, each of these prior approaches present significant shortcomings in terms of either 1) search quality, 2) system performance (such as latency and memory), or 3) security. In this work, we aim to address this gap in the literature by developing a new private vector search protocol that offers full cryptographic security guarantees while offering a more favorable tradeoff between search quality and system performance.

In particular, we focus our efforts on improving the search quality of the Tiptoe [6] protocol while retaining Tiptoe's benefits of strong system performance and strong security guarantees. Specifically, as we will discuss in more detail below, Tiptoe leverages linearly homomorphic encryption to guarantee, under standard cryptographic assumptions, that the server will *nothing* about the contents of the client's search query. However, the main drawback of Tiptoe is that its search quality, while nontrivial and useful in practice, considerably lags behind other methods. For

instance, the Pacmann private search system [12] reports approximately 90% search recall on a standard vector retrieval benchmark while Tiptoe only achieves roughly 35% recall. This gap in search quality motivates the key research investiation we investigate in this work: *Can we improve the Tiptoe protocol to improve its search quality while retaining its core benefits of security and system performance*.

In this paper, we provide initial results suggesting that the answer to this question is in fact an affirmative. In particular, we make the observation that the centroid routing algorithm at the core of Tiptoe is suboptimal because it ignores the fact that queries might be distributed rather differently from the database elements. For example, web search queries tend to be very short while documents on the web tend to contain a significant amount of text. Thus, we propose to leverage recent advances in the near neighbor search community on *learning-augmented* routing functions that learn to map queries to the appropriate shard. Moreover, we also investigate improving the underlying clustering algorithm of Tiptoe by leveraging balanced graph partitioning. Finally, we also make an improvement to the core Tiptoe protocol by performing local ranking over a larger set of candidates at no additional server-side or communication costs, with only slight increase in the client-side computational cost. Taken together, we combine these optimizations into a new system that we call Tiptoe++ as it retains the core benefits of Tiptoe while notably improving the search quality.

1.1 Our contributions

We hypothesize that we can achieve an improvement in the search quality vs. system latency tradeoff in Tiptoe by reconsidering the choice of vector indexing algorithm. Specifically, we apply ideas from a recent line of work in combining balanced graph partitioning with a *routing algorithm* [3,5,8]. We adapt this paradigm to the private search setting via the following high-level recipe:

1. Given the collection of input embeddings, convert these vectors into an approximate k-nearest neighbors proximity graph.

2. Apply a balanced graph partitioning algorithm to break up this graph into clusters of roughly equal size such that the number of edges crossing between clusters is minimized. This problem is NP-Hard, but very good software tools exist in practice such as METIS [7], which we leverage in our implementation.

3. Build an approximate near-neighbor search index over each of these sub-partitions.

4. Learn a *routing function* r that maps a query q to a subset of the partitions. In the literature, this router can either be an explicit algorithm or a machine learning classifier.

5. A search proceeds as follows: Given a query, apply the routing function $r(q)$ on the client side to map the query to the graph partition that is most likely to contain its nearest neighbors. Then, retrieve the similarity scores of the vectors in the routed partition from the server via an off-the-shelf LHE scheme. The client then performs a local ranking over the retrieved scores and identify the IDs of the top-k nearest neighbors.

Our approach in Tiptoe++ is a generalization of the high-level protocol proposed by Tiptoe. In Tiptoe, the clusters are determined via k-means and the routing function is simply a comparison between the query vector and each of the $O(\sqrt{N})$ cluster centroids. We propose to replace the k-means clustering component with a higher quality graph partitioning algorithm (such as the ones proposed in [5]) and either learn a routing function via training a neural network or using the locality-sensitive based router proposed in [5]. Additionally, we improve the Tiptoe search protocol to allow the client to perform local ranking at a larger set of candidates. Otherwise, we can keep the remainder of the Tiptoe protocol the same and aim for higher search recall. Essentially, our proposal trades off more preprocessing time (in the form of performing the graph construction, graph partitioning, and router function learning) and slight increase in client-side post-processing time (in the last step of local ranking) for improved search quality.

2 Related Work

Several prior works have studied the private vector search problem, but under varying assumptions and threat models. The post notable related work for this paper is Tiptoe [6], a system that utilizes cryptographic techniques to perform private vector search without revealing any information about the query to the search engine. More recently, the Pacmann [12] system achieves improvements over Tiptoe via a new graph-based search index and novel client-side processing (at the cost of more client-server communication). Another recent system Wally [1] achieves improved scalability for private semantic search by relaxing the security guarantee to differential privacy, but uses a clustering-based search approach as opposed to more performant graph-based indexes as in Pacmann. Finally, Compass [13] also proposes a private ANN search scheme under a different security model where the client sends an encrypted search index to the server (in the other systems mentioned above, the database is public).

One major challenge with these private vector search protocols is that they all sacrifice performance (in computation and communication costs as well as retrieval quality) in exchange for security. In particular, there are several intertwined design decisions that affect the performance of the system. Perhaps the most consequential design choice is what model of security to adopt. The Tiptoe [6] and Pacmann [12] protocols, for example, guarantee that the search engine's servers learn *nothing* about the search query. Alternatively, Wally [1] does leak information about the queries, but issues noisy fake queries as well to hide individual user searches under differential privacy. This model is much weaker than full cryptographic privacy, but can allow for substantial gains in performance by circumventing standard private information retrieval (PIR) lower bounds.

On the other hand, Compass [13] considers an even more stringent threat model where both the query and the search database are kept hidden from the server. This setting has heightened practical relevance due to the recent popularity of retrieval-augmented generation (RAG) with large language models (LLMs) where individuals may want to search over proprietary company data as opposed to the public web and ask an LLM to synthesize this knowledge in some manner.

Another critical design lever impacting the performance of a private vector search scheme is the choice of vector search algorithm. Tiptoe and Wally both adopt an approach of partitioning the database vectors into $O(\sqrt{N})$ clusters using k-means. While k-means clustering can be computed very efficiently, it can have poor quality in practice. In the case of plaintext vector search, graph-based search indexes, where one builds a network by adding connections with vectors that maintain

close proximity in the metric space, typically achieve the state-of-the-art performance on established benchmarks [2]. Based on this observation, Pacmann develops a custom graph index and proposes to search by storing the graph on the server and communicating subgraphs to the client for traversal via a PIR scheme. Similarly, Compass makes modifications to the popular HNSW graph search index [9] to perform search over oblivious RAM (ORAM) [4]. In the Compass setup, the client will make Path ORAM calls to the server to access nodes for graph search and then decrypt and perform the traversal locally. Because Path ORAM hides access patterns, the documents remain completely hidden from the server, but this protocol comes at a performance overhead. The Compass authors report sub-second search latency at small scales, but up to 13 seconds of latency at the scale of approximately 100M documents.

2.1 Our focus: cryptographically secure search over public documents

Given the large space of design decisions in private vector search, we must make a deliberate choice over where to focus our attention. To that end, we choose to focus on the setting where the documents are *public* and the security guarantee is one of full cryptographic privacy, meaning that, unlike in differential privacy and similar definitions, the server learns nothing about the query in our case. This is also the threat model considered by Tiptoe and Pacmann. We choose this particular setting because we see it as an immediate direction for potential real-world deployment in the future as similar systems have already received attention in private web search and private image search. Thus, we choose to focus on designing Tiptoe++, an improvement for the Tiptoe protocol that addresses precisely this setting of public documents and full query privacy.

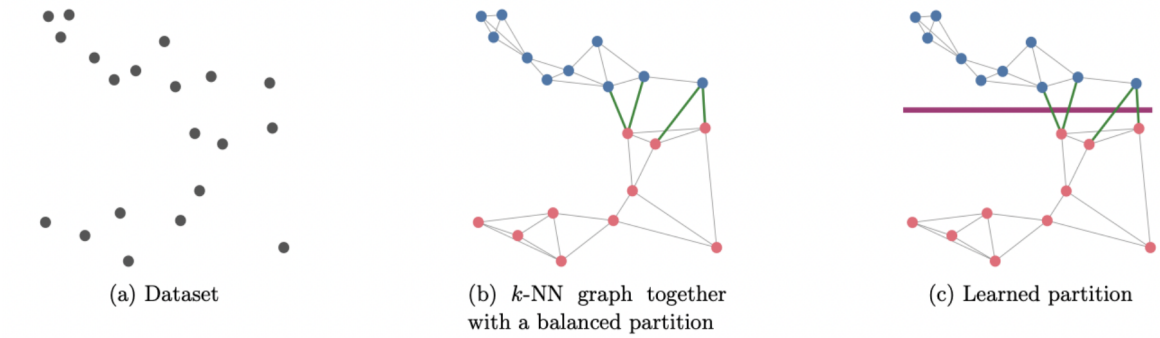
3 Methodology

At a high level, our system operates in three stages: index construction via clustering, query routing via a routing function, and private retrieval of similarity scores for local ranking. This architecture allows us to systematically decouple the tasks of partitioning, routing, and retrieval, making each component independently optimizable. In the following subsections, we present our improved clustering (Section 3.1), routing (Section 3.2), and retrieval protocol (Section 3.3).

3.1 Improved clustering

The original Tiptoe protocol adopts a simple k-means clustering scheme to partition the database vectors into roughly $\sqrt{N}$ clusters. While computationally efficient, k-means clustering introduces several critical issues in the context of private nearest neighbor search. First, the resulting clusters can be highly imbalanced in size, which complicates the bin-packing process used in Tiptoe to enforce uniform retrieval sets. Second, k-means relies on Euclidean distance to assign vectors to the nearest centroid, which can lead to suboptimal clustering structure in high-dimensional embedding spaces. Third, k-means is sensitive to its random initialization, which affects the search quality. These factors collectively degrade retrieval quality, especially when only a small number of clusters are probed for efficiency.

To address this, we adopt a balanced clustering strategy inspired by Neural LSH [3], which combines k-NN graph partitioning with supervised routing. The method consists of three steps:



(a) Dataset (b) k-NN graph together with a balanced partition (c) Learned partition

Figure 1: Stages of the improved clustering framework [3]: (a) the raw dataset, (b) the k-NN graph with balanced partitioning, and (c) the learned partition boundary extended to the ambient space.

(1) construct a k-NN graph over the dataset, where each node represents a vector and edges connect to its k closest neighbors; (2) apply balanced partitioning to minimize the number of inter-cluster edges under balance constraints; and (3) train a classifier to predict cluster IDs for unseen queries.

An overview of these stages is illustrated in Figure 1, where a k-NN graph is constructed over the dataset (Figure 1b), followed by a learned routing function that extends the partitioning to out-of-sample queries (Figure 1c).

This approach yields clusters that are both semantically coherent and size-balanced. In our system, this improved clustering forms the foundation for better routing (Section 3.2) and more effective private vector retrieval (Section 3.3). While it introduces additional preprocessing time for graph construction and partitioning, it significantly improves search quality and leads to more uniform communication cost. Importantly, the clustering step does not alter the structure of the search index: we retain the same search index architecture as Tiptoe, changing only how vectors are grouped. As a result, our system inherits all of Tiptoe's efficiency benefits, including compact index size, low communication overhead, and fast online performance. In our experiments, we used the METIS graph partitioning library [7] due to its ease of use. In the future, we could experiment with other clustering libraries and techniques as well.

3.2 Learned routing

Given a clustered dataset (Section 3.1), the next step is to route queries to clusters likely to contain their nearest neighbors. Traditional methods, such as Tiptoe, use centroid-based routing—assigning each query to the cluster with the most similar centroid. While efficient, this approach can be suboptimal in high-dimensional spaces where centroid proximity may not reflect actual neighbor density.

To improve routing quality, we adopt a learning-to-rank (LTR) approach proposed by Vecchiato et al. [11], which frames routing as a classification problem. The routing function is defined as $\tau(q; W) = Wq$, where $q \in \mathbb{R}^d$ is the query, $W \in \mathbb{R}^{L \times d}$ is a learnable weight matrix, and L is the number of clusters. Each row of W represents a learned cluster embedding, producing a score vector used to rank clusters by relevance.

Following this approach, we derive our training by identifying, for each query, the cluster

containing its exact nearest neighbor. Then, a model is trained using a softmax cross-entropy loss:

$$\mathcal{L}(q) = -\log\left(\frac{\exp(\tau_{i^*}(q))}{\sum_{j=1}^{L}\exp(\tau_j(q))}\right),$$

where i^* is the index of the correct cluster. This objective corresponds to optimizing Mean Reciprocal Rank (MRR) under a single-label setting and can be extended to top-k routing with softened targets.

Learned routing improves recall under fixed probe budgets and integrates easily with existing systems without introducing additional client-side cost compared to centroid-based routing, as the learned matrix W can replace the centroid matrix. At query time, the client computes a matrix-vector product $\tau(q; W) = Wq$, which has the same computational complexity as computing inner products with a set of centroids. Storage overhead is also minimal: the learned routing matrix $W \in \mathbb{R}^{L\times d}$ replaces the centroid matrix and is of the same size. Thus, learned routing improves accuracy without increasing client-side latency or storage requirements.

In our system, we implement the learned router on top of the graph-partitioned clusters described in Section 3.1. This allows the routing function to learn how to best map queries to partition IDs, rather than relying solely on geometric proximity to centroids. The model is trained using labeled query-partition pairs derived from exact search and integrated into the Tiptoe-style protocol as a drop-in replacement for centroid similarity routing.

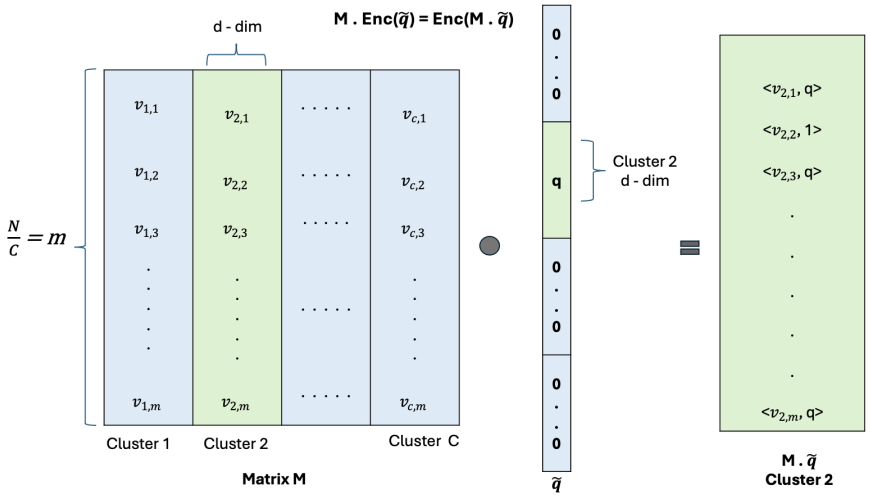
3.3 Improved search protocol

We also improve the search protocol of Tiptoe [6] by proposing a more effective client-side ranking strategy. Before describing our improvement, we first review the private vector search protocol of Tiptoe.

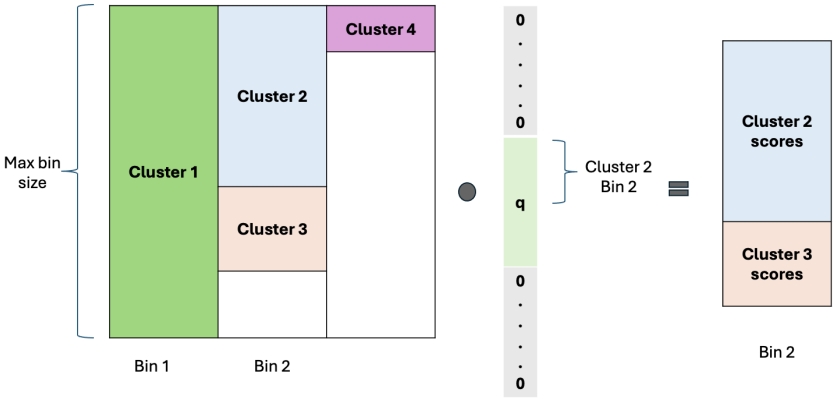
In the idealized setting where all C clusters are of equal size N/C, the server organizes the document vectors into a matrix $M \in \mathbb{R}^{(N/C)\times(C\cdot d)}$, where each cluster occupies a contiguous block of d-dimensional vectors. The client, given a query vector $q \in \mathbb{R}^d$, identifies the target cluster via the routing function and constructs a padded query vector $\tilde{q} \in \mathbb{R}^{C\cdot d}$, where only the d-dimensional block corresponding to the routed cluster is non-zero (and contains q). Using a linearly homomorphic encryption (LHE) scheme, the client encrypts $\tilde{q}$ and sends it to the server. The server then computes the encrypted inner products $M \cdot \tilde{q}$ via LHE, which correspond to similarity scores between the query and the documents in the routed cluster. The client decrypts the result and identifies the top-k vectors. This process is illustrated in Figure 2a.

However, in practice, clusters often vary significantly in size, making the above formulation inefficient. To address this, Tiptoe employs a bin-packing strategy: it groups clusters into fixed-size bins to ensure uniformity in encrypted computations. At query time, the client pads the query to retrieve all scores within the bin that contains the routed cluster. This process is shown in Figure 2b.

Despite retrieving all scores in the bin, Tiptoe only ranks and selects results from the routed cluster. Our key observation is that this discards valuable information already available to the client. We propose a simple yet effective improvement: instead of limiting local ranking to the routed cluster, the client performs ranking over the entire bin, which potentially spans multiple clusters. This expands the candidate set, and brings guaranteed improvement in retrieval quality. While this introduces a small increase in client-side computation, the added cost is minimal and



(a) Tiptoe with same-sized clusters



(b) Packing clusters into bins

Figure 2: The Tiptoe search protocol

fully local, with no additional communication or server computation. The benefit of this approach is confirmed in our evaluation, where bin-level ranking consistently improves search quality with negligible latency overhead.

4 Results

4.1 Implementation details

We implement our protocol in Go 1.20 with support for homomorphic encryption using the SEAL library, leveraging the same cryptographic parameters as in Henzinger et al [6]. We simulate a single-client, single-server setting on a single machine with 96 cores and 1 terabyte of RAM. We experiment with our protocol on the MS MARCO passage ranking dataset [10]. To increase the agility of our experiments, we down-sampled the 8.8 million documents of the MS MARCO

Table 1: Recall@10 and MRR@10 for different combinations of our three improvements. The baseline (Tiptoe) corresponds to all features disabled (C1), and Tiptoe++ corresponds to all enabled (C6).

	C1	C2	C3	C4	C5	C6
Improved clustering	✗	✗	✗	✗	✓	✓
Learned routing	✗	✗	✓	✓	✓	✓
Bin ranking	✗	✓	✗	✓	✗	✓
Recall@10 (%)	38.1	43.4	43.6	48.2	49.7	52.8
MRR@10 (%)	35.6	41.4	47.7	50.5	51.4	55.3

benchmark to 1 million. We then embedded into 384-dimensional vectors using the popular `All-MiniLM-L6-v2` embedding model [1] from the `sentence-transformers` library. Finally, we partitioned the 1 million documents into $\sqrt{N} = 1000$ clusters. For reproducibility, our code is open sourced at https://github.com/DeweiFeng/6.5610-project.

4.2 Search quality

We evaluate the search quality of our protocol on the MS MARCO dataset using recall@10 and MRR@10 over 100 queries. Recall@10 measures the fraction of queries for which at least one of the top-10 retrieved results is relevant, reflecting the system's ability to return correct answers within a small candidate set. MRR@10 (Mean Reciprocal Rank) complements this by measuring the average inverse rank of the first relevant result, truncated at rank 10. It captures not only whether relevant results are retrieved, but how early they appear in the ranked list, where higher values indicate better performance. Both metrics are standard in evaluating approximate nearest neighbor search (ANNS) algorithms. To isolate the effect of each design improvement, we measure the search quality under all different combinations of the three core modifications relative to Tiptoe: (1) balanced clustering via graph partitioning in place of k-means, (2) a learned routing function instead of nearest-centroid assignment, and (3) ranking over a bin instead of a single cluster.

Table 1 reports the recall and MRR for each configuration. When all three improvements are disabled, the system is equivalent to Tiptoe and serves as our baseline. Notably, Tiptoe achieves a Recall@10 of only 38.1% and MRR@10 of only 35.6%, indicating limited effectiveness in retrieving relevant results within the top 10 candidates. This is largely due to suboptimal clustering, rigid routing via nearest centroids, and narrow candidate pools at ranking time.

Enabling each improvement individually leads to consistent quality gains. Among them, the **learned routing function** yields the largest single-component improvement, suggesting that learned routing over centroids plays a critical role in driving performance. We hypothesize that this is the case because queries, especially in web search, are distributed very differently from documents, so using document centroids as the routing mechanism is suboptimal. Combining improvements yields further gains, and the full system with all three features enabled, namely Tiptoe++, achieves a Recall@10 of 52.8% and MRR@10 of 55.3%, outperforming Tiptoe by a substantial margin.

These results confirm that each of our design choices contributes meaningfully to search accuracy, and that the combination of improved clustering, learned query routing, and broader

[1] https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2

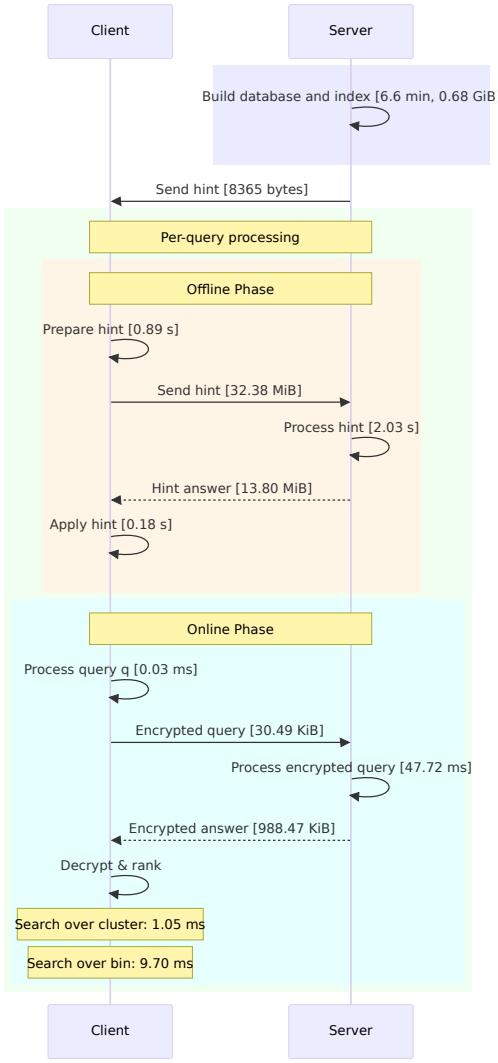


Figure 3: System performance of Tiptoe++

candidate ranking leads to substantial improvements in retrieval quality.

4.3 System performance

Our protocol retains the core architecture of Tiptoe and inherits its key efficiency benefits: compact search index, minimal online communication overhead and end-to-end latency. As in Tiptoe, the server performs a one-time pre-processing step to build the index and initialize the client with a small hint. Before each query, the client refreshes cryptographic keys and sends a hint to the server, which updates its internal state. This offline phase, while non-trivial in cost, is query-independent and can be performed in advance, effectively shifting the heavier parts of the protocol away from the online stage. The online phase remains lightweight, with only 57.45 ms of end-to-end latency, 30.49 KiB of uplink and 988.47 KiB of downlink communication, enabling interactive performance.

Compared to Tiptoe, the only added overhead lies in the ranking step. Instead of selecting the top-k scores from a single cluster, the client processes candidates from a full bin, which can increase

computation slightly. This increases the cost of the final sort from $O(k \log C)$ to $O(k \log B)$, where C and B denote the cluster and bin sizes, respectively. In practice, the bin size is only modestly larger than the cluster size, so the impact on computation is minimal. Indeed, bin-ranking results in a modest increase in the last step of client computation – from 1.05 ms in Tiptoe to 9.70 ms in our protocol. Despite this increase, the total online latency remains low, and the improved search quality from ranking a broader candidate set justifies the added cost.

The breakdown of system operations and their associated cost at each stage is illustrated in Figure 3.

5 Conclusion

In this project, we presented a private vector search protocol that extends Tiptoe with three key improvements: graph-based clustering, learned query routing, and bin-based ranking. These enhancements significantly improve search quality while retaining Tiptoe's efficiency advantages, including compact index size, low communication overhead, and fast online response. Our protocol achieves higher recall with only modest additional client-side computation, making it well-suited for latency-sensitive applications.

Private vector search remains a rapidly evolving area with significant open challenges. Future directions include scaling to larger datasets, comparisons with more state-of-the-art baselines, optimizing the underlying cryptographic protocols for throughput and latency, and designing more adaptive binning strategies – for example, grouping semantically related clusters to further improve search quality. We believe this work contributes a step toward practical secure search and highlights promising opportunities for further research.

6 Author contribution

All authors have contributed to the project:

- Xiaochen Zhu proposed the bin-ranking improvement and implemented the cryptographic search protocol in Go (server, client, coordinator, and performance measurement). He wrote Section 4 and Section 5 and edited other sections of the report.
- Vihan Lakshman implemented the core clustering, graph partitioning, and model training code and evaluated the results on the sampled MS MARCO benchmark. He also contributed to writing the first two sections of the report.
- Dewei Feng read the papers proposed by Xiaochen and Vihan. He collected the data for MS MARCO. He contributed to Section 3.1 and 3.2 of the report.
- Sarah Mokhtar contributed to the client-side implementation of the cryptographic search protocol in Go. She also contributed to Section 3.3 of the report.

References

[1] Hilal Asi, Fabian Boemer, Nicholas Genise, Muhammad Haris Mughees, Tabitha Ogilvie, Rehan Rishi, Guy N Rothblum, Kunal Talwar, Karl Tarbe, Ruiyu Zhu, et al. Scalable private search with wally. *arXiv preprint arXiv:2406.06761*, 2024.

[2] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

[3] Yihe Dong, Piotr Indyk, Ilya Razenshteyn, and Tal Wagner. Learning space partitions for nearest neighbor search. In *8th International Conference on Learning Representations (ICLR)*, 2020.

[4] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[5] Lars Gottesbüren, Laxman Dhulipala, Rajesh Jayaram, and Jakub Lacki. Unleashing graph partitioning for large-scale nearest neighbor search. *arXiv preprint arXiv:2403.01797*, 2024.

[6] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th symposium on operating systems principles*, pages 396–416, 2023.

[7] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[8] Vihan Lakshman, Choon Hui Teo, Xiaowen Chu, Priyanka Nigam, Abhinandan Patni, Pooja Maknikar, and SVN Vishwanathan. Embracing structure in data for billion-scale semantic product search. *arXiv preprint arXiv:2110.06125*, 2021.

[9] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[10] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. Ms marco: A human-generated machine reading comprehension dataset. 2016.

[11] Thomas Vecchiato, Claudio Lucchese, Franco Maria Nardini, and Sebastian Bruch. A learning-to-rank formulation of clustering-based approximate nearest neighbor search. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '24)*, pages 1–5. ACM, 2024.

[12] Mingxun Zhou, Elaine Shi, and Giulia Fanti. Pacmann: Efficient private approximate nearest neighbor search. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

[13] Jinhao Zhu, Liana Patel, Matei Zaharia, and Raluca Ada Popa. Compass: Encrypted semantic search with high accuracy. *Cryptology ePrint Archive*, 2024.