

CAROUSEL: GPU-Accelerated Private Vector Search via Homomorphic Sketching

Sohaib
sohaib@ucsb.edu
UC Santa Barbara

Divyakant Agrawal
divyagrawal@ucsb.edu
UC Santa Barbara

Amr El Abbadi
amr@cs.ucsb.edu
UC Santa Barbara

Soamar Homsi
soamar.homsi@us.af.mil
Air Force Research
Laboratory

ABSTRACT

Vector search on untrusted infrastructure exposes the *query embedding*, a faithful summary of the user’s intent. Fully homomorphic encryption can hide the query, but existing approaches still require a full scan followed by ranking of encrypted scores. Scalable private systems avoid encrypted ranking by either sending scores from a single cluster to the client or moving the search to the client, which must first stream the entire vector index for preprocessing. We present CAROUSEL, which instead scores the full corpus under encryption and compresses the results for client-side ranking. The server scores every vector under encryption, raises the scores to a power that preserves their order by magnitude while suppressing all but a handful, and compresses the resulting sparse vector into a logarithmic number of ciphertexts using homomorphic additions alone. Because every vector is scored, CAROUSEL incurs no recall loss from discarding candidates. Since clients hold no database-dependent state, insertions and deletions take effect immediately at the cost of a single vector write. CAROUSEL searches 1 billion vectors of BIGANN SIFT1B across multiple GPUs and achieves recall@100 of 0.9715. At 100 million SIFT vectors, it achieves recall@100 of 0.9845 with a query latency of 3.735 seconds on a single GPU.

1 INTRODUCTION

Modern information retrieval systems increasingly rely on vector similarity search as their principal operation. Documents, images, code, and product catalogs are embedded into a high-dimensional space in which semantic similarity is inner-product proximity, and a query is answered by finding the stored vectors nearest to the query’s embedding. Search engines, recommendation systems, and retrieval-augmented generation pipelines all depend on this primitive, and vector databases serving hundreds of millions of embeddings are now routine infrastructure [31]. The first step of every such query, however, is to hand the query embedding to the server and that embedding is a faithful summary of what the user wants to know. This is a privacy risk as search queries reveal sensitive personal information such as where they are, what they are doing, or what they are living through [5, 13].

Efficient vector search algorithms [27] typically rely on query dependent access patterns, which requires that the query remain in plaintext. For a search to be completely private, the server’s actions must be completely independent of the query itself. This means the server cannot skip over any part of its index, because the decision to skip would reveal information about the search i.e. which part of the index were not relevant to the query. Private search therefore forces a linear scan on every request, and recent literature is organized around either paying for that scan or avoiding it by preprocessing.

Every approach considered so far in the literature costs something that the user notices.

These costs usually appear as either massive communication overhead or reduced search accuracy. Systems that offer the strongest privacy, where the server learns nothing at all about the query, face this tradeoff clearly. For example, Tiptoe [15] suffers from a drop in search quality, while PACMANN [33] relies on heavy preprocessing that requires the server to stream the whole index to the client. Even relaxing the privacy guarantees does not solve the problem. Wally [6], which uses weaker differential privacy, does not improve search quality significantly. Thus, the fundamental challenge remains unsolved, providing complete privacy and high search accuracy without a heavy communication cost.

To resolve this challenge, this paper presents CAROUSEL, a GPU-accelerated vector search system that achieves complete privacy with high search accuracy and low communication overhead. The tradeoff prior systems face is not strictly necessary. Rather, it arises from how they choose to reduce communication overhead. The fundamental problem is that an exhaustive scan produces a relevance score for every document, creating a response far too large for transmission. Prior systems (Tiptoe, Wally) address this by pruning the database before the query arrives, estimating which documents are relevant and this filtering causes loss of accuracy. CAROUSEL avoids this by scoring every document and compressing only the final answer, discarding nothing before ranking is complete.

CAROUSEL executes this secure scan using Fully Homomorphic Encryption (FHE) [9, 12], which enables the server to perform additions and multiplications directly on ciphertexts. Selecting the best matches from the resulting scores, however, requires comparison, which FHE does not support natively. A single comparison must be approximated by a high-degree polynomial, making the sort needed to rank even a few hundred encrypted scores cost seconds [28]. Avoiding this sort leaves the server unable to select the top matches, so it would have to return every score for the client to rank. To overcome the bottleneck of returning millions of encrypted relevance scores, CAROUSEL employs a *sketch*, a compact summary of a large number of vectors that preserves its largest entries for later recovery. However, sketching techniques fundamentally rely on data sparsity, assuming that the vast majority of entries are near zero. A list of relevance scores violates this assumption as most documents exhibit a modest, non-zero similarity to the query, and even the top matches sit only marginally higher than this baseline.

To make the score list sparse without altering which documents rank highest, we introduce a step called *sharpening*. The server raises every encrypted score to a power, preserving their ordering by absolute magnitude. Since all scores fall between -1 and 1, larger

values shrink far more slowly than smaller ones. After this operation, the broad middle of poor-to-modest matches collapses rapidly toward zero, while the highest scores retain their magnitude. This widens the numerical gap between the top matches and the rest, allowing a sketch to compress the results while preserving high retrieval accuracy.

The sketch itself is constructed using only addition, the cheapest operation FHE offers. The server compacts the sparse scores into a few compact summary rows and returns those in place of the millions of initial scores. The client then decrypts the rows and recovers the top matches locally. Therefore, every comparison in the system occurs on the client after decryption. This ensures that the server never has to compare or sort encrypted values, avoiding the step that is expensive under encryption.

The matches the client recovers, however, are positions in the index rather than the documents themselves. To fetch the indexed data without revealing which the client wants, it uses *private information retrieval* (PIR), a protocol that allows a client to read an entry from a server-held database while the server learns nothing about which entry was read [16, 24, 32]. CAROUSEL addresses only the ranking step and composes with any PIR scheme for the retrieval that follows. The two together form an end-to-end private pipeline, the query stays hidden during search, and the retrieved documents stay hidden during retrieval.

CAROUSEL’s design also enables efficient online updates. It maintains a flat array of vectors, without a clustering or graph index to update. Insertions append a vector, and deletions zero its coordinates. When encoded diagonals are cached, only the affected blocks need to be refreshed. These changes require no client-side refresh of corpus-dependent search metadata. Tiptoe [15] describes support for continuous updates by assigning new or modified documents to clusters and distributing updated centroids and metadata to clients, but does not evaluate update latency or query performance under concurrent updates. We evaluate both the cost of insertions and deletions and their impact on query latency under sustained update workloads.

We implement CAROUSEL using CKKS [9], a fully homomorphic encryption scheme, and support both single- and multi-GPU execution. On MS MARCO, CAROUSEL reaches an MRR@100 of 0.303, matching a non-private graph index and exceeding Tiptoe at 0.13, Wally at 0.18, and Pacmann at 0.266. On SIFT-100M, it achieves recall@10 of 0.9785, compared with 0.90 for Pacmann, and completes each query in 3.735 seconds on a single GPU. On BIGANN SIFT1B, CAROUSEL searches all 1 billion vectors across multiple GPUs and achieves recall@100 of 0.9715.

CAROUSEL makes the following contributions:

- A private vector-search design that compresses the answer with a sketch rather than pruning the database, matching non-private search quality with logarithmic communication and cryptographic query privacy.
- An encrypted pipeline that sparsifies scores through power-based sharpening and moves all ranking to the client, avoiding encrypted comparison.
- A dynamic, GPU-accelerated implementation that supports single-write updates and multi-GPU execution, evaluated on SIFT, MS MARCO, and GIST at scales up to 10^9 vectors.

Table 1: Notation.

Sym.	Meaning	Sym.	Meaning
N	database vectors	n	CKKS ring dimension
d	vector dimension	ℓ	slots, $\ell = n/2$
k	results returned	L	levels in the chain
q, v_i	query and i th vector	Δ	CKKS scaling factor
s_i	score of v_i against q	p	sharpening exponent
B	blocks, $\lceil N/\ell \rceil$	M	fold matrix
R	fold rows returned		

2 BACKGROUND AND PRELIMINARIES

2.1 Private Vector Search

A vector search index holds a database $\mathcal{D} = \{v_1, \dots, v_N\}$ of N vectors in $\mathbb{R}^d$, obtained by passing documents, images, or other content through a pretrained embedding model. A client holding a query vector $q \in \mathbb{R}^d$ wants the identifiers of the k database vectors most similar to q , which for the embedding models used in this paper means largest inner product $\langle q, v_i \rangle$. All vectors are normalized to unit length, so the inner product coincides with cosine similarity and lies in $[-1, 1]$.

In the private version of this problem, the server must learn absolutely nothing about the query, q . We guarantee this using standard semantic security for the query. Our threat model assumes a single *semi-honest* server that follows the protocol, but still attempts to deduce the query from its observations.

Definition 2.1 (Query privacy). Let $\text{View}_{\mathcal{S}^*}[C(q)]$ denote the view of a server $\mathcal{S}^*$ in its interaction with a client that searches for q . A private vector search protocol satisfies query privacy if, for every $\mathcal{S}^*$ and every pair of queries q_0, q_1 , the distributions $\text{View}_{\mathcal{S}^*}[C(q_0)]$ and $\text{View}_{\mathcal{S}^*}[C(q_1)]$ are computationally indistinguishable.

The view includes every message the server receives, the memory it touches, and the sizes and timing of the responses it returns. Definition 2.1 forces all of these to be independent of the query, which is the formal statement of the linear scan requirement discussed in Section 1. Table 1 collects the notation used throughout.

2.2 Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) allows a server to compute on encrypted data without decrypting it. Given an encryption of m and public evaluation keys, the server can evaluate a supported function f to obtain an encryption of $f(m)$, without interacting with the key holder. Many practical schemes derive their security from the ring learning with errors assumption [26].

We use CKKS [9], which supports approximate arithmetic on real or complex numbers and is well suited to computing similarity scores. A fresh ciphertext consists of two polynomials in the ring $R_Q = \mathbb{Z}_Q[X]/(X^n + 1)$, where n is a power of two and Q is the ciphertext modulus. CKKS packs up to $\ell = n/2$ values into the *slots* of a single plaintext polynomial, allowing additions and multiplications to operate on all slots simultaneously in a SIMD manner. During encoding, a scaling factor Δ controls the precision with which these values are represented in the integer polynomial ring.

Decryption and decoding recover approximate values, and homomorphic operations introduce and propagate numerical errors. The encryption parameters must therefore provide sufficient precision for the intended computation.

CKKS supports a bounded number of multiplication stages through a *modulus chain*, $Q = q_0 q_1 \cdots q_L$. Multiplying two ciphertexts at scale Δ produces a ciphertext at scale Δ^2 . A subsequent *rescaling* operation divides the scale and modulus by one prime, bringing the scale back near Δ and consuming one level. The circuit’s *multiplicative depth*, the largest number of sequential multiplication stages along any computation path, must therefore fit within the available level budget. Longer computations require *bootstrapping* to refresh the ciphertext and restore usable levels, at substantial computational cost. We design CAROUSEL to complete its encrypted computation within the initial level budget, avoiding bootstrapping entirely.

Table 2 lists all the standard homomorphic operators we use in our system, along with their required keys and relative computation costs. These costs vary widely and this variation directly shapes how we design our pipeline. For example, rotation is highly expensive because it relies on key switching, which dominates both runtime and memory. Every distinct rotation shift requires the server to hold a separate, multi-megabyte evaluation key. In our setup, each key is about 10 MB, so minimizing the total number of rotation keys is a critical goal. Finally, computation cost scales with the size of the ciphertext, meaning it is always cheaper to compute at a lower multiplicative level than at a higher one.

Table 2: Homomorphic operators, the key material each requires, the levels it consumes, and its relative cost.

operator	key	levels	cost
Add(ct, ct/pt)	—	0	negligible
MulPt(ct, pt)	—	0	low
Mul(ct ₀ , ct ₁)	—	0	moderate
Relin(ct, RK)	relin.	0	high
Rescale(ct)	—	1	low
Rot(ct, RK _{step})	one per shift	0	high
Conj(ct, CK)	conj.	0	high

The asymmetry in computational costs among FHE operators introduces two fundamental constraints that shape our system architecture. First, an operator built exclusively from Add and MulPt consumes no evaluation keys, adds no depth, and introduces no key switching error, which makes it qualitatively cheaper than one that uses rotations. Second, comparison is not a native FHE operation. Sign, minimum, maximum and argmax must be approximated by compositions of high degree polynomials [10], each consuming several levels, which is why selecting the top k entries of an encrypted vector is expensive even for a few hundred candidates [28]. This is the reason CAROUSEL performs no comparison under encryption.

2.3 Prior Approaches to Private Vector Search

Private nearest-neighbor search splits into two families. Interactive protocols built on secure multi-party computation can exploit an index and so avoid touching the whole database, at the price of many

rounds and of trust assumptions such as non-colluding servers. Non-interactive protocols built on homomorphic encryption need one round and one server, but must compute over every record. CAROUSEL belongs to the second family.

Private search over a public corpus. Our design directly compares to existing architectures that hide only the query while relying on a public corpus and a single untrusted server. Coeus [1] first posed the problem for text, privately scoring Wikipedia articles with tf-idf at a cost that does not reach web scale. Tiptoe [15] replaced tf-idf with semantic embeddings, reducing private search to private nearest-neighbor search, and answered it with a preprocessing-friendly linearly homomorphic scheme [16] together with clustering to bound the response size. Clustering partitions the corpus offline, typically by k -means, which groups vectors around a set of centroids so that each vector belongs to the centroid nearest it. At query time the client probes only the cluster whose centroid is closest to its query, so the server effectively scores a fraction of the corpus rather than all N . Tiptoe [15] uses $\sqrt{N}$ clusters, and a relevant document outside the probed cluster cannot be retrieved. Wally [6] keeps the clustering and relaxes the guarantee instead, revealing the probed cluster under differential privacy using cover queries routed through an anonymity network. Its overhead falls as the concurrent client population grows but it requires an anonymity network to strip identifiers and shuffle requests. Pacmann [33] moves graph traversal to the client. Starting from an entry node, the client repeatedly uses preprocessing PIR [32] to fetch the neighbor list of the node closest to the query until the search converges. This shifts most computation and state to the client and before querying, it must stream the entire graph index and store 0.6 GB of hints for three million vectors or 2.9 GB for one hundred million. Each query also requires 20–30 sequential network rounds. These three systems form our baselines.

Other private nearest-neighbor protocols. Another line of work uses interactive multi-party computation (MPC). SANNS [8] combines distributed oblivious RAM and garbled circuits, while Panther [25] reduces communication through batch PIR. Both use a single server and prevent the client from learning anything about the corpus beyond the search results. Servan-Schreiber et al. [30] instead require two non-colluding servers to obtain sublinear communication. Compass [34] considers a different setting, where the client encrypts and uploads its own corpus, protecting it from the server. GraSS [21], the first fully homomorphic graph index, still takes hours per query on datasets much smaller than ours. Because of this cost, prior work viewed fully homomorphic scans as impractical. HANNS [29] also assumes that non-interactive search requires a full scan, but lowers its cost by ordering clusters to reduce encrypted comparisons and quantizing vectors to reduce storage.

3 THE CAROUSEL QUERY PIPELINE

3.1 Overview

The server holds a database of N vectors $v_1, \dots, v_N$ in $\mathbb{R}^d$, and a client holds a query $q \in \mathbb{R}^d$. The client wants to retrieve the indices of the k database vectors most similar to q , where similarity is the inner product $\langle q, v_i \rangle$, and must learn them without revealing q to the server. Once the indices are retrieved, the client can use

PIR techniques to retrieve the corresponding records. All vectors are normalised to unit length, so this inner product is the cosine similarity and lies in $[-1, 1]$. CAROUSEL solves this with fully homomorphic encryption where the server computes on the encrypted query and the scores stay encrypted as well.

CAROUSEL answers a query in four stages. The first three run on the server under encryption but the last runs on the client after decryption.

- **Scan.** The server computes the inner product of the encrypted query against every database vector, producing an encrypted score $s_i = \langle q, v_i \rangle$ for every i .
- **Sharpen.** The server raises each encrypted score to a power, $s_i \mapsto s_i^p$. This preserves the ranking of the strong matches while driving the rest toward zero, turning a dense score vector into a sparse one.
- **Fold.** The server compresses the sparse scores into R summary rows using a fixed binary matrix M , needing only homomorphic addition: $\text{row}_r = \sum_i M_{r,i} s_i^p$.
- **Decode.** The client decrypts the R rows and recovers the top- k identifiers locally. Every comparison in the system happens here, in plaintext.

The scores of ℓ consecutive vectors are packed into the slots of a single ciphertext. We call such a group of ℓ vectors a *block*, so a corpus of N vectors occupies $B = \lceil N/\ell \rceil$ blocks. We illustrate the pipeline throughout with a corpus of $N = 64$ vectors of dimension $d = 4$ and a ciphertext of $\ell = 8$ slots, so the similarity scores for a given query occupies $B = 8$ blocks.

3.2 Data Layout

The client encrypts its query into a single ciphertext. The dimension d of a query (and the vectors in the database) is much smaller than the number of slots ℓ in a ciphertext. Instead of placing the query once and leaving the remaining slots empty, the client tiles the query across all slots, repeating its d coordinates ℓ/d times. Figure 1 shows a query with $d = 4$ coordinates and a ciphertext has $\ell = 8$ slots, so the pattern q_0, q_1, q_2, q_3 appears twice where q_i refers to the i -th coordinate of the query vector q .

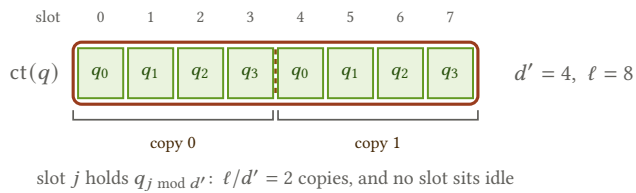


Figure 1: A four-dimensional query packed into a ciphertext with eight slots, tiled twice to fill all slots.

The corpus is laid out to match this tiled query. CKKS multiplies two operands slot by slot, so a product is formed only between values in corresponding slots. Consider storing the corpus in row-major order, with each vector's d coordinates placed contiguously across slots. Figure 2 shows what this produces where $v_{i,j}$ refers to the j -th coordinate of the i -th vector, v_i . Slot 0 holds q_0 against $v_{0,0}$, slot 1 holds q_1 against $v_{0,1}$, and slot 4 holds q_0 against $v_{1,0}$. Every slot

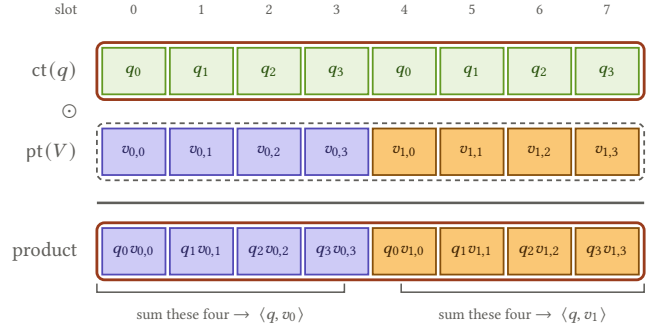


Figure 2: Row-major layout. The tiled query multiplies elementwise against two vectors laid out in consecutive slots.

Algorithm 1 Server processing of one block b

Require: rotated query $RQ[0..d-1]$, fold matrix M , row accumulators $\text{rows}[0..R-1]$

```

1: for  $k = 0$  to  $d - 1$  do
2:    $\text{DIAG}[k] \leftarrow \text{ENCODE}(v_k)$  ▷ Eq. 1
3: end for
4:  $\text{acc} \leftarrow RQ[0] \odot \text{DIAG}[0]$ 
5: for  $k = 1$  to  $d - 1$  do
6:    $\text{acc} \leftarrow \text{acc} + RQ[k] \odot \text{DIAG}[k]$ 
7: end for ▷  $\text{acc}[i] = \langle q, v_i \rangle$  for vector  $i$  of the block
8: for  $s = 1$  to  $nsq$  do
9:    $\text{acc} \leftarrow \text{SQUARE}(\text{acc})$  ▷ sharpen: raise scores to  $p = 2^{nsq}$ 
10: end for
11: for  $r = 0$  to  $R - 1$  do
12:   if  $M[r][b] = 1$  then
13:      $\text{rows}[r] \leftarrow \text{rows}[r] + \text{acc}$  ▷ fold into selected rows
14:   end if
15: end for
```

carries a product from a different vector coordinate, and the d products belonging to one vector are scattered across d separate slots. Recovering a single inner product would require summing these 4 slots together, and summing within a ciphertext requires rotations, which need rotation keys and are computationally expensive.

To avoid this limitation, we adapt the Halevi–Shoup diagonal representation for homomorphic matrix–vector multiplication [14]. Instead of storing vectors in row-major order, we build d plaintext diagonals, each containing one coordinate from every vector in the block. Diagonal k is

$$\text{diag}[k][i] = v_i[(i+k) \bmod d], \quad (1)$$

so slot i carries one coordinate of vector i . The offset by i staggers which coordinate each slot takes, which makes a one-slot shift of the query line up with the next diagonal, thus simplifying the product and summation operations and avoiding expensive and unnecessary rotations. Figure 3 shows all four diagonals of a block of eight four-dimensional vectors, each filling all $\ell = 8$ slots.

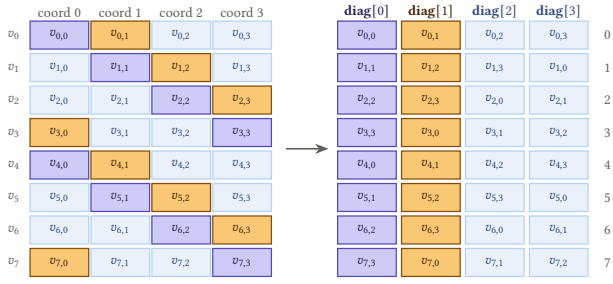


Figure 3: Diagonal plaintexts with the first two diagonals highlighted.

3.3 Encrypted Scan

Query expansion. The scan needs one copy of the query aligned to each of the d diagonals. The client uploads only one ciphertext, so the server produces the rest itself. The uploaded query repeats with period d , so rotating it by one slot advances every slot to the next coordinate. The server therefore builds copy k by rotating the uploaded query k slots, giving $RQ[0], \dots, RQ[d-1]$. Every copy is one further single-slot rotation of the previous one, so the whole chain only needs a single rotation key. The expansion runs once per query and its output is reused across all B blocks. This costs memory since, the d rotated copies stay resident for the duration of the scan.

Scoring. Copy k and diagonal k now match slot by slot. In slot i , copy k holds $q[(i+k) \bmod d]$ and diagonal k holds $v_i[(i+k) \bmod d]$, so their product is a matching pair of coordinates from the same vector. Multiplying the two and accumulating over all d diagonals gives one block of scores,

$$\text{acc} = \sum_{k=0}^{d-1} RQ[k] \odot \text{DIAG}[k], \quad (2)$$

where $\odot$ is the slot-wise product of a ciphertext against a plaintext. Slot i of acc now holds $\langle q, v_i \rangle$, the score of the i -th vector of the block. Figure 4 shows the four products and their sum on the running example.

This diagonal arrangement allows the system to independently accumulate the inner products in parallel. Every product landing in a given slot belongs to the same vector, so no slot ever mixes contributions from two different vectors. For instance, consider slot 0 in Figure 4. Across the four products it holds $q_0 v_{0,0}$, $q_1 v_{0,1}$, $q_2 v_{0,2}$ and $q_3 v_{0,3}$, which are the four terms of $\langle q, v_0 \rangle$. Summing the four ciphertexts therefore places $\langle q, v_0 \rangle$ in slot 0, and the same holds for all other slots for its own vector. Since, no operation moves a value between slots, the score of vector i stays in slot i for the remainder of the pipeline. Since a CKKS slot holds a complex number while a score is real, we pack two blocks per ciphertext and separate them after the scan with one conjugation, halving the ciphertexts scanned at the cost of a conjugation key.

The scan operation multiplies each plaintext diagonal against the query ciphertext. However, CKKS cannot multiply a ciphertext directly by a list of numbers. The plaintext must first be converted into the number-theoretic transform (NTT) domain, a representation required for CKKS multiplication. This transform is applied

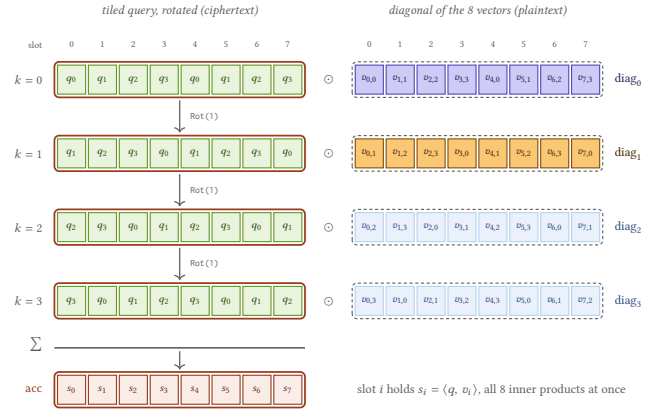


Figure 4: The scan

once for every prime in the modulus chain. The resulting representation is much larger than the coordinates it came from. We call this *diagonal encoding*. This is a computationally expensive step and dominates query latency, so the encoded diagonals are cached and reused across queries when they fit in device memory.

3.4 Sharpening the Score Vector

The vector inner products after the scan are not sparse. A large number of documents score modestly and the good matches are only slightly above them. A linear sketch cannot compress scores with this distribution. We therefore reshape it by raising every score to a power p , which the server does by squaring $nsq = \log_2 p$ times. Each squaring costs one level and one relinearisation. Squaring contracts every score, but it contracts small scores faster than large ones. A score of 0.95 raised to the power 16 becomes 0.44, while a score of 0.60 falls to 2.8×10^{-4} and a score of 0.30 to 4.3×10^{-9} . The gap between the head and the bulk therefore widens by orders of magnitude. After nsq squarings only a handful of entries remain within a few percent of the peak, and the rest are numerically zero. The score vector is now sparse enough for the sketch to compress.

The operation also preserves the order of the surviving scores, so sharpening costs no accuracy in the ranking. This holds for non-negative scores. An even power maps $-s$ and s to the same value, so a score close to -1 would survive sharpening and be reported as a strong match. In practice this does not arise, because the most negative score a query encounters is far smaller in magnitude than its largest positive score and the ratio is suppressed by the power p .

The exponent depends on the distribution of the cosine scores of the corpus. A small p leaves the bulk uncrushed, and the decoder cannot separate the true matches. On the other hand, a large p drives even the strong scores below the noise floor, where they are lost. The usable range depends on how cleanly a corpus's true neighbors separate from the bulk, and on the corpus size. The exponent also fixes the encryption level, $L_{\text{enc}} = nsq + 1$, so changing p changes the depth of the modulus chain.

3.5 The Fold

After sharpening, the server holds one ciphertext of scores for each of the B blocks, referred to as *block ciphertexts*. Only a few entries

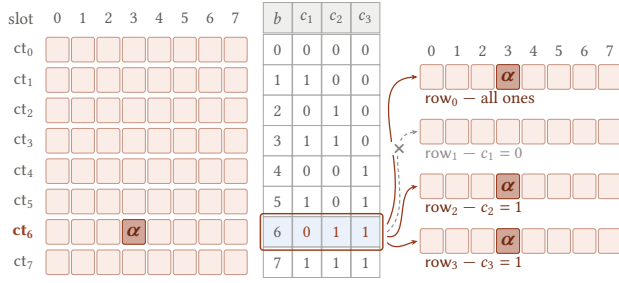


Figure 5: Folding $B = 8$ blocks into four rows.

carry significant magnitude, referred to as *heavy items*. The client only needs to recover the top- k scores, but sending all B block ciphertexts constitutes a significant overhead. The fold compresses the B ciphertexts into a smaller number of ciphertext sketches called *rows*. Each row is a sum of a different subset of blocks. The goal for the client is to decrypt the rows and recover only the heavy entries.

The first row of the fold is the sum of all B block ciphertexts. Slot i of this row holds the sum of all scores at position i across all B block ciphertexts. Since only a few entries are heavy, the slots of larger magnitude in this row are the slots occupied by heavy items. The first row therefore identifies all candidate slots that the client needs to examine. Once we have identified which slots the heavy items come from, the next step is to recover which of the B block ciphertexts contains the heavy item in slot i .

The remaining rows identify which block ciphertext contains the heavy item. A block index below B requires $m = \lceil \log_2 B \rceil$ bits. To illustrate the basic construction, assign each block its m -bit binary codeword and create one fold row per bit position. Row r sums the block ciphertexts whose r -th codeword bit is one, so each block is added to a unique combination of rows. These codewords form the columns of a binary *fold matrix* M , where $M[r][b] = 1$ if block ciphertext b is added to row r . Figure 5 illustrates this uncoded construction for $B = 8$, which requires three codeword rows in addition to row_0 . Block ciphertext 6 has codeword $(0, 1, 1)$ (LSB first) and is therefore added to row_2 and row_3 , but not to row_1 . A heavy item from this block appears in the corresponding pattern of rows, allowing the client to recover its block index.

In the ideal case, a heavy item occupies a given slot in exactly one block ciphertext, while every other block ciphertext holds a value near zero at that position. The client decrypts the rows and inspects row_0 to identify the heavy slots. For each candidate slot, the client evaluates which of the remaining rows contain a large value and which remain near zero. This binary pattern directly reconstructs the codeword of the originating block ciphertext. For example, consider slot 3 in Figure 5, where all block ciphertexts except block ciphertext 6 hold near-zero values, and block ciphertext 6 holds a magnitude of α . Examining slot 3 across the folded rows, row_0 will contain α , flagging it as a slot of interest. Extracting the values across all rows at this slot yields the vector $\langle \alpha, 0, \alpha, \alpha \rangle$. By matching this non-zero pattern to the known codewords, the client determines that the item originated from block ciphertext 6.

The idealised assumption fails due to two reasons. First, the block ciphertexts that do not hold the heavy item do not contribute zero. Each row sums many block ciphertexts, and every block carries

a residual bulk left over from sharpening, so a slot that should contain a near zero value instead holds the accumulated bulk of all the block ciphertexts added into that row, together with the encryption noise of CKKS. Therefore the entries are not exact but large and small values, and the more block ciphertexts contribute to a row, the larger the small values grow. Reading each row as an exact bit becomes unreliable due to noise corrupting some of the values. Second, two heavy items can occupy the same slot in two different block ciphertexts. Their codewords are then added together, and the client sees the sum of two codewords rather than one. As a result, the collision of two heavy items in the same slot, introduces a secondary challenge for the client-side decoding phase.

The first problem is resolved by replacing standard binary labels with a more robust codeword assignment. With plain binary codewords, two block codewords can differ in a single bit, for example, codewords of blocks ciphertext 4 and 5 differ only in the last position. This means a single unreliable row can turn one block's pattern into another's. The solution is to read the rows not as exact bits, but as a pattern to be matched to the nearest codeword. To ensure that this nearest match remains unambiguous under noise, we draw the codewords from a code of minimum Hamming distance three[20]. Because any two codewords now differ in at least three positions, the true codeword stays closest to the observed pattern even if one row is completely corrupted. This spacing makes the decode robust to the residual bulk and encryption noise.

CAROUSEL constructs the robust codewords using a randomized procedure. It begins with the m -bit representation of each block ID and repeatedly samples a random linear combination of the ID bits, appending the resulting bit to every block's codeword. Sampling continues until every pair of codewords has Hamming distance at least 3, producing an L -bit base codeword. CAROUSEL then samples r_p additional combinations in the same manner and appends them as extra measurement bits, yielding a codeword $c(b)$ of length $L + r_p$. Each codeword position defines one fold row, so together with the all-ones row_0 , the server returns $R = 1 + L + r_p$ rows. The augmented codewords form the columns of $M \in \{0, 1\}^{(R-1) \times B}$. Because appending coordinates cannot decrease Hamming distance, the augmented code retains a minimum distance of at least 3.

At any identified candidate slot, the client extracts the corresponding values from each row to construct an observation vector y . This vector is composed of three distinct contributions. First, a heavy item of magnitude α residing in block ciphertext b contributes α exclusively to the rows where its assigned codeword $c(b)$ contains a one, generating a structured component $\alpha c(b)$. Second, because every code row aggregates the residual bulk from its constituent block ciphertexts, this accumulation introduces a uniform baseline offset β across all rows, represented by the term $\beta \mathbf{1}$. Finally, the inherent CKKS encryption noise and minor inter-row variations in the bulk are absorbed into a single error vector ϵ . The observed vector is therefore formalized as

$$y = \alpha c(b) + \beta \mathbf{1} + \epsilon, \quad (3)$$

which describes a scaled codeword shifted by a constant offset and perturbed by noise. Recovering the originating block ciphertext thus reduces to determining which valid codeword $c(b)$ best aligns with this observation signal.

Algorithm 2 Client decoding of one query**Require:** decrypted rows $C_0 \dots C_{R-1}$, fold matrix M , budget C

```

1:  $cand \leftarrow$  indices of the  $C$  largest entries of  $C_0$ 
2: for all  $i \in cand$  do
3:    $y \leftarrow (C_1[i], \dots, C_{R-1}[i])$ 
4:    $\tilde{y} \leftarrow y - \text{mean}(y)$ 
5:    $\hat{b}_i \leftarrow \arg \max_b \langle \tilde{y}, \tilde{c}(b) \rangle^2 / \|\tilde{c}(b)\|^2$ 
6:    $\hat{s}_i \leftarrow \langle \tilde{y}, \tilde{c}(\hat{b}_i) \rangle / \|\tilde{c}(\hat{b}_i)\|$ 
7: end for
8: return  $\{\hat{b}_i \cdot \ell + i\}$  ordered by decreasing  $\hat{s}_i$ 

```

To determine which codeword best aligns with the observation, we treat the evaluation of each candidate as a two-parameter fit. Assuming the heavy item originated in a specific block ciphertext b , the vector y should approximate $\alpha c(b) + \beta \mathbf{1}$ for some magnitude α and offset β , disregarding the error term ϵ . For a given codeword, the optimal fit is the pair (α, β) that minimizes the distance to y . The quality of this fit is measured by the residual:

$$\text{res}(b) = \min_{\alpha, \beta} \|y - \alpha c(b) - \beta \mathbf{1}\|^2. \quad (4)$$

This residual is small only when $c(b)$ can be scaled and shifted to reproduce the observed pattern. The client therefore evaluates $\text{res}(b)$ for every block ciphertext and selects the candidate with the minimum residual. The fitted magnitude α at that optimal block then provides an estimate of the item's score, allowing the client to rank the recovered matches. The block index from the residual fit and the slot index from the candidate position together give the item's global identifier, $\text{id} = b \cdot \ell + i$. This is repeated over the candidate slots and ordering by the fitted magnitude yields the top matches.

The second problem, a collision, is when two heavy items occupy the same slot in different block ciphertexts. The observation by the client is a superposition, $y \approx \alpha_1 c(b_1) + \alpha_2 c(b_2) + \beta \mathbf{1} + \epsilon$, and the single-codeword fit of Equation 4 can match at most one of them. A collision is a birthday event over the ℓ slots of a block ciphertext. It depends on the number of surviving heavy items and the slot count, and not on N or B , so it does not become more frequent as the corpus grows. The rate also remains low since $k \ll \ell$ items survive sharpening. Among the top hundred scores, the expected number of colliding pairs in our settings is 0.15 per query, so most queries have no collision at all and the few that do, result in at most one match. We recover the second item by a second pass over the residual. Having fitted the first item's magnitude α_1 at a slot, the client subtracts its contribution $\alpha_1 c(b_1)$ from the observation and refits the remainder against every codeword, excluding the block ciphertext already chosen. Subtracting the first item makes the second fit informative since, any correlation that survives is evidence of a genuinely different source. A second item is accepted only when its fitted magnitude is significant compared to the first.

The residual fit of Equation 4 has a closed form that the client can evaluate. In order to minimize the residue over the offset β , we subtract the mean from both y and the codeword, which removes the constant bulk term $\beta \mathbf{1}$ and leaves only the block-dependent part. Minimising over the magnitude α is then a projection of the centred

observation onto the centred codeword. Therefore, the block index with smallest residual is the one with largest normalised correlation.

$$\hat{b} = \arg \min_b \text{res}(b) = \arg \max_b \frac{\langle \tilde{y}, \tilde{c}(b) \rangle^2}{\|\tilde{c}(b)\|^2}, \quad (5)$$

where $\tilde{y}$ and $\tilde{c}(b)$ are the mean-centred observation and codeword. The fitted magnitude $\hat{s} = \langle \tilde{y}, \tilde{c}(\hat{b}) \rangle / \|\tilde{c}(\hat{b})\|$ estimates the item's score. This is a matched filter, correlating the observation against each candidate codeword and selecting the largest is the optimal linear detector when a single codeword is present [4]. The r_p additional rows give the matched filter more measurements, improving its robustness to residual bulk and CKKS noise.

In Algorithm 2, we show the client side decode. It selects candidate slots from C_0 , and at each one recovers the block index using Equation 5 and its score by the estimated magnitude (α). The collision pass (excluded in algorithm 2) extends this without any changes. After the first block index is fitted, the client subtracts $\hat{s} \tilde{c}(\hat{b})$ from $\tilde{y}$ and applies the same Equation 5 to the residual, excluding the block index already chosen.

3.6 Updates

CAROUSEL maintains a flat index which is a array of vectors divided into many blocks. These blocks are encoded into diagonals and each query reads from these encoded diagonals. Each group of d diagonals contain coordinates from ℓ vectors of the same block. Since, a vector contributes one entry to each of its block's d diagonals, the cost of an update is re-encoding of these d diagonals.

Insert. An insert appends the new vector to the array. If the last block has available rows, the vector fills the first spare row. Otherwise, a new block is allocated for it. Because the vector index and the downstream PIR store share a single identifier space, they must remain synchronized. The record is appended to the PIR store first, and then the new vector is added to the vector index. This ordering guarantees that every identifier returned by the search is already present in the PIR. Consequently, the underlying PIR scheme must support append operations [22, 23]. If the block count B increases after any insertion, the fold matrix M and the number of rows R also need to be recomputed. Since M is public and entirely generated by a shared seed, the server transmits only the new B and each client regenerates M locally.

Delete. A delete zeroes all of the d coordinates of the target vector. A zero vector has inner product zero with every query, so the vector scores zero before and after sharpening, and has no effect on the fold row. Thus, once the block is re-encoded the index of the vector is unreachable. Because the search never exposes the identifier again, the matching PIR record does not need to be removed immediately. It can be lazily reclaimed later during compaction.

4 EVALUATION

In this section, we evaluate CAROUSEL's retrieval quality, efficiency, scalability, and update performance. We also conduct various ablation studies to examine its principal tradeoffs, including projection width versus retrieval quality and response size versus recall. Finally, we compare CAROUSEL with prior private vector-search systems.

Experimental setup. We run all server-side computations on a single NVIDIA RTX 5090 GPU with 32GB of device memory under

CUDA 13.2, hosted on a machine with 96GB of RAM, unless otherwise noted. Homomorphic evaluation uses Cheddar [11], a CKKS library using 32-bit words, with ring degree 65,536 and scaling factor 2^{30} . We measure all client-side latencies using a single core of an AMD Ryzen 9950X3D processor.

Datasets and workloads. We evaluate CAROUSEL on four workloads. SIFT-100M [19] contains 10^8 128-dimensional vectors, which is searched at native width. For the multi-GPU evaluation, we use the full BIGANN SIFT-1B dataset [19], containing 10^9 vectors of the same dimensionality. MS MARCO [7] contains 3.21×10^6 768-dimensional vectors, which we project to 128 and 256 dimensions using PCA. GIST1M [18] contains 10^6 960-dimensional vectors, which we project to 256 and 512 dimensions.

Evaluation metrics. Each configuration is evaluated on 1,000 queries and we report $\text{recall}@k$, the fraction of the exact top- k neighbors appearing in the returned set. For MS MARCO, we additionally report $\text{MRR}@100$, following prior private search systems [6, 15, 33]. A query receives a score of $1/j$ if its ground-truth passage appears at rank $j \leq 100$, and zero otherwise. $\text{MRR}@100$ is the mean of these scores across all queries. Because CAROUSEL searches over projected vectors, we distinguish three metrics to separate the sources of retrieval loss. The *projection ceiling* compares the exact top- k neighbors in the projected space with those in the original space, measuring the quality retained after dimensionality reduction alone. *Projected recall* compares CAROUSEL’s output with the exact top- k in the projected space, isolating loss introduced by the search pipeline. Finally, *true recall* compares its output with the exact top- k in the original, unprojected space, capturing the combined end-to-end loss from projection and the encrypted pipeline.

Security of Deployed Parameters. The deployed instance uses ring degree 2^{16} , a ternary secret of Hamming weight 1024, Gaussian error of standard deviation 3.2, and evaluation keys modulo a product reaching 2^{610} . Under the lattice estimator, the cheapest known attack on this configuration costs 2^{391} , above the 256-bit category of the homomorphic encryption standard [2, 3].

4.1 Retrieval Quality

Table 3 reports CAROUSEL’s retrieval quality on the three datasets, using the native dimensionality for SIFT and two projected dimensionalities each for MS MARCO and GIST. Projected recall ranges from 0.9411 to 0.9887, showing that the encrypted pipeline recovers at least 94% of the neighbors returned by an exact search in the projected space. Moreover, true recall remains close to the corresponding projection ceiling, indicating that dimensionality reduction, rather than the encrypted pipeline, accounts for most of the end-to-end quality loss.

On SIFT, CAROUSEL searches the native 128-dimensional vectors and achieves true recall of 0.9845 at rank 100 and 0.9785 at rank 10. On MS MARCO, increasing the projection width from 128 to 256 raises true recall@100 from 0.6337 to 0.8610 and $\text{MRR}@100$ from 0.2763 to 0.3031. Exact search over the full 768-dimensional vectors achieves an $\text{MRR}@100$ of 0.3125, so Carousell reaches 97.0% of this plaintext ceiling.

GIST exhibits the same projection-width tradeoff. Increasing the projected dimension from 256 to 512 improves true recall@100 from

Table 3: Retrieval quality over 1,000 queries per configuration.

dim. $\rightarrow d$	SIFT	MS MARCO		GIST	
	128	768 $\rightarrow$ 128	768 $\rightarrow$ 256	960 $\rightarrow$ 256	960 $\rightarrow$ 512
projection ceiling, $r@100$	–	0.6367	0.8639	0.8045	0.9240
projected $r@100$	–	0.9797	0.9887	0.9411	0.9413
true $r@100$	0.9845	0.6337	0.8610	0.7903	0.8996
true $r@10$	0.9785	0.6885	0.8740	0.7531	0.8842
$\text{MRR}@100$	–	0.2763	0.3031	–	–

0.7903 to 0.8996 and true recall@10 from 0.7531 to 0.8842. Meanwhile, projected recall remains essentially unchanged at approximately 0.94. Thus, the wider projection improves the information retained from the original vectors without changing the accuracy of the encrypted pipeline.

4.2 Query Performance and Scalability

Table 4 reports the average per-query cost across 1000 queries for each evaluated configuration. CAROUSEL supports two execution modes depending on whether the encoded database diagonals fit in device memory. In the *cached mode*, the server retains these diagonals and reuses them across queries, removing diagonal encoding from the query path. In the *recompute mode*, the server retains only the flat vector array and reconstructs the diagonal encodings for every query. The choice between these modes is determined by device-memory capacity. MS MARCO and GIST use cached diagonal encodings, achieving server latencies of 0.031–0.045 seconds and 0.034–0.057 seconds, respectively. On MS MARCO, server latency is dominated by the encrypted scan and sharpening stages. However on GIST, query expansion accounts for approximately half of the total server latency. Query expansion scales with vector dimensionality, whereas scanning and sharpening scale with dataset size. Because GIST uses equal or higher projected dimensions but contains fewer vectors than MS MARCO, query expansion constitutes a larger fraction of its latency.

SIFT represents the capacity-limited mode. Since, its encoded diagonals require approximately 512 GB and do not fit on our 32 GB GPU, CAROUSEL recomputes the diagonal encodings for each query. Of its 3.735-second server latency, encoding accounts for 1.993 seconds or 53.4%, followed by sharpening at 1.083 seconds or 29.0% and scanning at 0.553 seconds or 14.8%. Folding adds only 0.097 seconds because it requires neither ciphertext rotations nor multiplications.

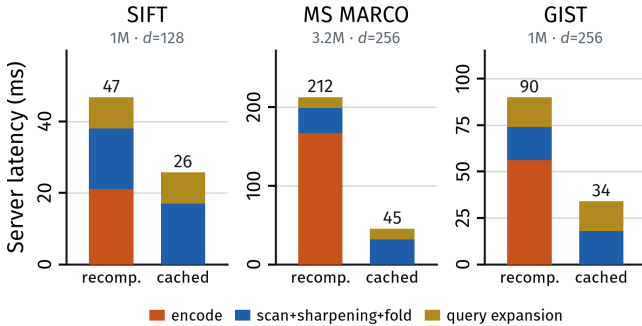
Client-side computation remains efficient in all of the configurations. Building the query takes 18.9–23.3 ms and decrypting and decoding the response takes 8.5–12.0 ms across the five configurations. Queries upload 3.2–5.2 MB and download 8.4–12.6 MB. The upload size varies because the corpus-specific sharpening exponent determines the required multiplicative depth, which in turn determines the ciphertext modulus and size.

Effect of caching. To isolate the benefit of caching, Figure 6 evaluates both execution modes on three configurations whose encoded diagonals fit in device memory. Caching eliminates the encoding stage while leaving query expansion, scanning, sharpening, and folding unchanged. It reduces server latency from 46.8 ms to 26.2 ms on SIFT-1M (the first one million vectors of SIFT), from 211.7 ms

Table 4: Per-query cost averaged over 1000 queries.

	SIFT	MS MARCO		GIST	
vectors N	10^8	3.21M	3.21M	1M	1M
dim. $\rightarrow d$	128	768 $\rightarrow$ 128	768 $\rightarrow$ 256	960 $\rightarrow$ 256	960 $\rightarrow$ 512
blocks B	3052	99	99	31	31
cached diagonals (GB)	512	10.07	20.13	9.66	19.33
sharpening exponent (p)	128	8	8	64	64
encode	1.993	--	--	--	--
query expansion	0.0078	0.0063	0.0125	0.0151	0.0303
scan	0.553	0.0116	0.0191	0.0083	0.0162
sharpening	1.083	0.0097	0.0096	0.0089	0.0089
fold	0.097	0.0027	0.0027	0.0007	0.0007
finalize	0.0014	0.0010	0.0010	0.0009	0.0009
server total (sec)	3.735	0.031	0.045	0.034	0.057
client query build (ms)	23.31	19.09	18.91	21.77	22.10
client decrypt & decode (ms)	12.01	9.78	9.55	8.50	8.56
upload / download (MB)	5.24/12.58	3.15/10.49	3.15/10.49	4.72/8.39	4.72/8.39
eval. keys (MB)	31.46	18.88	18.88	28.31	28.31

to 44.8 ms on MS MARCO, and from 90.3 ms to 33.8 ms on GIST. These reductions correspond to speedups of 1.79 $\times$, 4.73 $\times$, and 2.67 $\times$, respectively. The improvement is largest on MS MARCO because it uses a sharpening exponent of only $p = 8$, compared with $p = 128$ on SIFT-1M and $p = 64$ on GIST. Since, sharpening requires less compute, encoding accounts for 79% of the recomputed latency on MS MARCO, compared with 62% on GIST and 45% on SIFT-1M. Caching eliminates this stage and thus removes the corresponding share of the latency. The cached representations occupy 5.37 GB, 20.13 GB, and 9.66 GB of device memory, respectively.

**Figure 6: Server latency with recomputed and cached diagonal encodings.**

Scaling corpus size(N). We evaluate how CAROUSEL scales with the corpus size N . Figures 7a and 7b report server latency and recall@100 as we vary N on MS MARCO (cached mode) and SIFT (recompute mode).

On SIFT, server latency grows linearly from 0.343 s at 10M vectors to 3.735 s at 100M. The decompositions shows that both the encoding stage and the combined scan-sharpening-fold stage scale proportionally with N , resulting in a linear growth in end-to-end server latency. We observe that, recall depends on the sharpening exponent. At $p = 64$, recall declines from 0.9883 to 0.9809 as the corpus grows to 24.2M vectors, since the accumulated residual bulk exceeds what this exponent can suppress. We therefore increase the exponent to $p = 128$ at 29.0M vectors, as marked by the dashed

line. Recall subsequently recovers and remains between 0.9829 and 0.9873 as we scale further towards 100M vectors.

MS MARCO exhibits the same linear scaling at a fixed exponent of $p = 8$. With diagonal encodings cached, increasing the corpus from 0.328M to 3.21M vectors raises server latency from 17.2 ms to 44.4 ms and recall from 0.8185 to 0.8610. The server latency consists of a fixed component and a component that scales with the corpus: query expansion remains constant at 12.4–12.6 ms, whereas the scan-sharpening-fold stage grows with N . Because encoding is removed from the query path and query expansion does not scale with N , a 9.80 $\times$ increase in corpus size raises latency by only 2.58 $\times$.

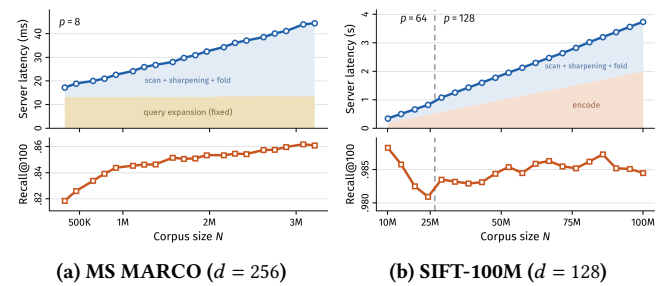
These results confirm the expected behavior of an exhaustive encrypted scan that is server latency scales linearly with N , while retrieval quality remains stable.

4.3 Response Size and Recall

Recall that Section 3.5 constructs a distance-3 base code using randomized parity rows and augments it with r_p additional measurement rows. Although every generated code satisfies the same minimum-distance guarantee, its sampled structure can affect the matched-filter decoder. We therefore evaluate the variance across generated codes and the effect of r_p on retrieval quality.

Figure 8 shows that the additional rows primarily improve robustness across random folding matrices. Without additional rows, an unfavorable fold matrix can substantially degrade retrieval quality: on GIST, the worst construction achieves 0.481 recall against a mean of 0.763. At the deployed setting $r_p = 4$, the standard deviation decreases by as much as 25.6 $\times$ relative to $r_p = 0$, while mean recall increases by at most 0.029. Increasing r_p to 8 reduces the standard deviation by as much as 217 $\times$, but improves mean recall by at most 0.0028 beyond $r_p = 4$. The effect is weakest on SIFT, where the longer $L = 19$ code already exhibits little variability at $r_p = 0$ and four additional rows improve mean recall by only 0.0007.

Therefore, the response contains $R = 1 + L + r_p$ fold rows. CAROUSEL packs two fold rows into each CKKS ciphertext by placing one in the real component and the other in the imaginary component. Each response ciphertext occupies 1.05 MB, giving an amortized cost of 0.525 MB per fold row. As a result, the download

**Figure 7: Scaling with corpus size. Server latency grows linearly with N on both datasets. Recall@100 increases with N on MS MARCO and remains stable on SIFT-100M. The dashed line marks the change from $p = 64$ to $p = 128$.**

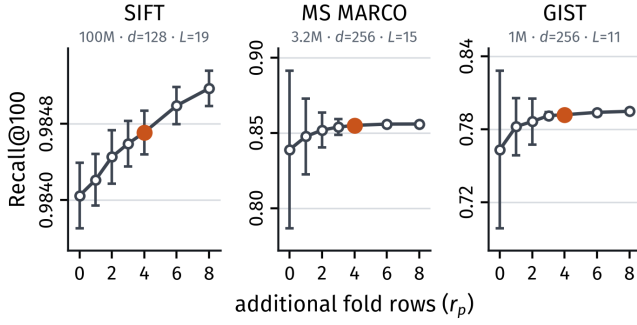


Figure 8: Effect of additional random fold rows on true recall@100. Each point reports the mean over 20 independently generated folding matrices, with each matrix evaluated on 1000 queries. Error bars show one standard deviation across matrices, and the orange markers denote the deployed setting $r_p = 4$.

grows in 1.05 MB steps, and the deployed choice $r_p = 4$ adds 2.1 MB to the response.

4.4 Updates and Freshness

Section 3.6 describes CAROUSEL’s update mechanism: insertions append vectors to the database, while deletions zero the corresponding rows. Ensuring next-query freshness presents different challenges in the cached and recompute modes. We evaluate the update cost and behavior under sustained write loads in each mode below.

Recompute mode. When the encoded diagonals exceed device memory capacity, CAROUSEL constructs them on demand for each query. Although recomputation increases query latency, it simplifies updates because each update is an $O(d)$ write to the flat vector array. Because the next query constructs its diagonals directly from this array, it automatically observes the updated data without cache invalidation or derived-state maintenance. We measure this write path against the SIFT-100M database, which contains 100M 128-dimensional vectors and occupies 25.6 GB. A single insertion takes $4.00\mu s$, corresponding to 250,000 insertions per second and a deletion takes $3.25\mu s$, or 308,000 deletions per second. Batching further amortizes the write overhead: batches of 10,000 insertions and deletions complete in $8.8\mu s$ and $7.2\mu s$, corresponding to throughputs of 1.14 and 1.38 billion updates per second, respectively. Concurrent updates have little effect on query performance in the recompute mode. When the writer is running on a separate CUDA stream, SIFT-100M query latency increases from 3.7244s (without updates) to 3.7341s at 10000 updates per second and 3.7538 s at 50000 updates per second. These increases correspond to overheads of only 0.26% and 0.79%, respectively.

Cached mode. When the encoded diagonals fit in device memory, CAROUSEL caches and reuses them across queries. An update can therefore make the cached representation of the affected block stale. An update can therefore make the cached representation of the affected block stale. Append-only insertions only affect the final block, whereas random deletions can make blocks throughout the dataset stale. To preserve next-query freshness, CAROUSEL marks

that block as dirty and re-encodes it before it is next read. Because each vector contributes to every diagonal of its block, modifying a single row requires re-encoding the entire block. This takes 3.35 ms on MS MARCO-256 and 3.53 ms on GIST-256. Multiple updates to the same block before it is next read coalesce into a single re-encoding.

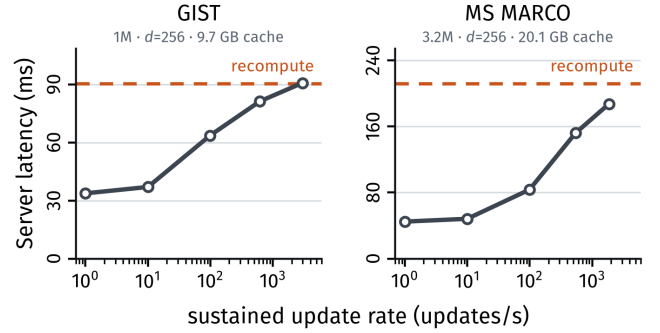


Figure 9: Server latency under sustained update load in the cached mode. Each point reports the mean over 100 queries while updates execute concurrently on a separate CUDA stream. The dashed lines show the corresponding recompute baselines.

Figure 9 shows how refreshing dirty blocks in cached mode affects query performance. Without concurrent updates, caching reduces server latency from 90.3 to 33.8 ms on GIST and from 211.7 to 44.8 ms on MS MARCO. At 10 updates per second, cached execution remains $2.44\times$ faster on GIST and $4.39\times$ faster on MS MARCO. At 100 updates per second, the cached mode remains $1.42\times$ and $2.54\times$ faster than the recompute mode on GIST and MS MARCO, respectively. As the update rate increases, more blocks become dirty between queries and must be re-encoded. GIST reaches the recompute baseline near 3,000 updates per second. On MS MARCO, the writer saturates at 1,851 updates per second, where cached execution remains $1.13\times$ faster than recomputation. Separate batch-size measurements indicate that its crossover would occur only near 3×10^5 updates per second. Thus, lazy re-encoding preserves the caching benefit over a wide range of update rates while maintaining next-query freshness.

For write-heavy deployments, CAROUSEL can move re-encoding off the query path. Updates accumulate into batches, while a background CPU thread performs the encoding. The query thread then transfers the re-computed diagonal encodings to the GPU and swaps them into the cache. This makes query latency largely independent of the update rate. Even at 10,000 updates per second, query latency remains nearly unchanged, increasing from 33.8 to 35.1 ms on GIST and from 44.8 to 50.3 ms on MS MARCO. This mode trades freshness for throughput. Queries continue using the previous cached version of an affected block until its replacement is ready, producing staleness windows of 1.9 s on GIST and 3.7 s on MS MARCO.

4.5 Multi-GPU Scaling to One Billion Vectors

Setting. We evaluate CAROUSEL on the full BIGANN SIFT-1B corpus using ten GPUs. The corpus contains 10^9 128-dimensional vectors, which CAROUSEL scans in their native representation without projection. We partition these corpus evenly across the ten GPUs, assigning 100M vectors to each GPU. Each GPU stores 25.6 GB of vectors, with a total corpus size of 256 GB across all 10 GPUs.

Each GPU receives the same encrypted query and computes a partial fold over its shard. The ten partial folds are then added homomorphically before a single response is returned. Because folding is linear, the aggregated response is equivalent to folding the complete corpus, and the client decodes once over all blocks. The sharpening exponent used is $p = 128$. The diagonal encodings are recomputed for every query. The latency and recall results are measured over 1,000 queries on the ten-GPU deployment.

Table 5: SIFT-1B on ten GPUs compared with SIFT-100M on one GPU.

	SIFT-1B	SIFT-100M	change
N	10^9	10^8	$10\times$
GPUs	10	1	$10\times$
fold rows R	29	24	+5
true recall@100	0.9715	0.9845	-0.0130
true recall@10	0.9699	0.9785	-0.0086
recall p10 / median / p95	0.970 / 0.990 / 1.000	0.970 / 0.990 / 1.000	—
server latency	3.765 s	3.735 s	$1.01\times$
per shard	3.735 s	3.735 s	$1.00\times$
query expansion	0.008 s	0.008 s	$1.00\times$
encode	1.993 s	1.993 s	$1.00\times$
scan	0.553 s	0.553 s	$1.00\times$
sharpening (7 squarings)	1.083 s	1.083 s	$1.00\times$
fold	0.097 s	0.097 s	$1.00\times$
cross-shard reduction	0.030 s	—	—
client query build	23.3 ms	23.3 ms	$1.00\times$
client decrypt and decode	15.8 ms	12.0 ms	$1.32\times$
client matched filter ($C = 2000$)	375 ms	31 ms	$12.1\times$
upload	5.24 MB (1 ct)	5.24 MB	$1.00\times$
download	15.73 MB (15 ct)	12.58 MB	$1.25\times$

Results. Table 5 compares SIFT-1B with the single-GPU SIFT-100M configuration. Server latency remains nearly constant after sharding. The ten-fold larger corpus takes 3.765 s per query, only $1.01\times$ the SIFT-100M latency, because each GPU processes approximately the same number of vectors as the single-GPU SIFT-100M configuration. Processing each shard takes 3.735 s, while aggregating the partial folds adds only 0.030 s. Thus, on the server side, scaling from 100 million to one billion vectors requires additional hardware rather than additional query latency.

Communication grows much more slowly than the corpus. The upload remains 5.24 MB because the client sends one ciphertext regardless of N , and the server generates the remaining rotations. The download increases by only 25%, from 12.58 to 15.73 MB, because the response contains $\lceil R/2 \rceil$ ciphertexts, where R is the number of fold rows, and R grows logarithmically with the number of blocks. Consequently, a ten-fold increase in corpus size leaves upload unchanged and increases download by only $1.25\times$. The client computation increases with corpus size. Query construction remains unchanged, while decryption and decoding increase only slightly,

from 12.0 to 15.8 ms. The matched-filter stage grows more substantially, from 31 to 375 ms, because it scores candidate slots against codewords from ten times as many blocks. This measurement uses a single CPU core, and the work can be parallelized across additional cores to reduce the latency. On the other hand, reducing the number of candidate slots from $C = 2000$ to $C = 200$ substantially lowers the matching cost with negligible loss in recall@10. These two options mitigate the dominant client-side computation cost at one billion vectors.

Retrieval quality remains stable even at this scale. True recall@100 decreases from 0.9845 to 0.9715, while recall@10 decreases from 0.9785 to 0.9699. The loss is concentrated in a small subset of queries as median recall remains 0.990, the 95th percentile remains 1.000, and the fraction of queries achieving at least 0.95 recall decreases from 97.3% to 95.7%.

4.6 Comparison with Prior Systems

Table 6 compares CAROUSEL with Tiptoe [15], Wally [6], and Pacmann [33] on the public MS MARCO corpus, with NGT [17] included as a non-private quality reference. Tiptoe, Pacmann and CAROUSEL hide the client’s query cryptographically, whereas Wally reveals the probed cluster and bounds the leakage with differential privacy, relying on an anonymization service and a large concurrent client population. We report the performance metrics exactly as published by each system on its respective hardware. The only exception is Tiptoe because its original paper omits latency for the MS MARCO dataset, we use the measurement reported by WALLY. Since these evaluations were conducted across highly different hardware architectures, we report the metrics in their native units and refrain from drawing direct performance comparisons. CAROUSEL reaches an $MRR@100$ of 0.303 at $d = 256$, the highest among the private systems and within 0.007 of NGT. Even at a smaller projected dimension of $d = 128$, CAROUSEL reaches $MRR@100$ of 0.276, higher than all prior systems. Its communication (13.6MB) is below Tiptoe and Pacmann, but exceeds Wally.

Tiptoe shares our setting and guarantees: a single semi-honest server, a public corpus and a single round protocol. The main difference is that while Tiptoe scores all vectors, the client only retrieves one cluster. This affects the quality of retrieval compared to CAROUSEL: yielding 0.13 against 0.303. Tiptoe’s clustering also requires the client holding 0.61MB of centroid metadata that a corpus change invalidates, whereas CAROUSEL’s client holds none. WALLY relies on this same clustering idea and suffers from the same retrieval quality problem. To escape it, WALLY allows the client to make multiple queries to retrieve different clusters. This recovers some quality but still remains low compared to CAROUSEL (0.18 at 5 queries against 0.303). WALLY’s only advantage is its lower communication cost, but this is due to its differential privacy approach, which requires a massive anonymity set of 5×10^5 clients to be efficient. Pacmann offers the same cryptographic guarantees but executes the search on the client, hiding each access with PIR. Therefore, the computational burden falls entirely on the client. Furthermore, Pacmann requires a preprocessing phase before any query can be issued, where the client streams a 2.7 GB index and retains 614 MB of hint state. In addition, each query requires 20 sequential network rounds and 63.2MB of communication. CAROUSEL

Table 6: Comparison on MS MARCO ($\approx 3.2\text{M}$ documents), as reported by each system. Tiptoe’s cost figures are Wally’s simulation. Wally’s ranges cover probing 1 to 5 clusters.

	Tiptoe	Wally	Pacmann	CAROUSEL	NGT
Privacy	crypto	DP + anon anon hop,	crypto	crypto	none
Assumptions	none	5×10^5 clients	none	none	—
Rounds	1	1	20	1	—
Offline pass	none	none	2.7 GB	none	—
Client state	0.61 MB	0.04 MB	614 MB	none	—
Comm. (MB)	17.4	0.56–2.60	63.2	13.6	—
Compute	11.0 core-s	0.39–1.50 core-s	0.29 core-s	0.045 gpu-s	—
MRR@100	0.13	0.12–0.18	0.266	0.303	0.31

outperforms Pacmann in retrieval quality (0.303 against 0.266) and matches it even when operating at a reduced dimensionality of $d = 128$ (MRR@100: 0.276). Pacmann also evaluates on SIFT, scaling to 10^8 vectors with parameters tuned to reach recall@10 of 0.90, at a cost of a 60 GB preprocessing download and 3 GB of client hint state. CAROUSEL reaches recall@10 of 0.9785 with 5.2 MB of upload and 12.6 MB of download per query.

No prior system evaluates at the billion scale. Tiptoe evaluates 360 million embeddings on a 45-server cluster, requiring 2.7 seconds, 56.9 MB of bandwidth, and 68 MB of client-cached metadata. Pacmann scales up to 10^8 vectors, requiring a 60 GB preprocessing download and 3 GB of hint state. Projecting Pacmann to 10^9 vectors based on its scaling laws ($\tilde{O}(\sqrt{n})$ for hints, linear for preprocessing) dictates roughly 10 GB of state and a massive 600 GB download. In contrast, CAROUSEL queries 10^9 vectors in 3.77 seconds on ten GPUs, downloading only 15.7 MB with zero client state. None of the prior systems evaluate updates to the corpus. CAROUSEL is the only system in this comparison that provides full cryptographic guarantees, requires only a single network round, and maintains zero corpus-derived client state. Moreover, it achieves higher retrieval accuracy than every private baseline while remaining practical at the billion scale.

5 DISCUSSION AND LIMITATIONS

CAROUSEL has four main limitations. Two of them are related to sharpening and two to deployment. Although ordinary insertions and deletions require no client action, changes to the encryption parameters require clients to re-upload their evaluation keys. The server stores three such keys per client: a multiplication key for sharpening, a rotation key for query expansion, and a conjugation key for the packing split. When there is change in the sharpening exponent p , it alters the modulus-chain depth and invalidates the existing keys. As the corpus grows, folding combines scores from more blocks, increasing the accumulated bulk noise that sharpening must suppress and potentially requiring a larger p . Section 4.2 shows a transition from $p = 64$ to $p = 128$ beyond 29 million vectors, after which $p = 128$ remains sufficient through 1 billion vectors. This is CAROUSEL’s only nonincremental reconfiguration. Across our SIFT scaling experiment, it occurs only once, requires no re-indexing, and transfers less than 32 MB of replacement keys per client. A second limitation is that sharpening orders scores by

absolute magnitude rather than signed value. Because the exponent is an even power of two, a strongly negative score could become heavy and returned as a match. In practice, this issue does not arise, as the most negative score a query encounters is consistently much smaller in magnitude than its largest positive match. While using an odd exponent would preserve the sign, it requires one additional ciphertext multiplication and increases the overall multiplicative depth. This configuration is omitted as no corpus requires it.

The remaining constraints are deployment-driven. Device memory limits whether encoded diagonals can be cached, forcing larger configurations to use recompute mode. Client decoding scales with the block count B and the candidate budget C , the number of slot candidates the client evaluates. Matched-filter latency increases from 31 ms at 10^8 vectors to 375 ms at 10^9 , where it dominates client cost. Reducing C lowers this latency but can reduce recall.

6 CONCLUSION

CAROUSEL shows that single-round private nearest-neighbor search with FHE is practical at billion-vector scale. Prior systems avoid fully homomorphic evaluation and instead use linearly or somewhat homomorphic encryption. By using GPU-accelerated CKKS, CAROUSEL makes exhaustive FHE scanning practical and achieves high recall by not discarding potential candidates. However ranking those scores still remains expensive, so CAROUSEL introduces sharpening and folding to complete the private search pipeline efficiently. CAROUSEL also keeps no corpus-derived state on the client. The client holds only its keys and a public projection, so a change to the corpus invalidates nothing it holds. This makes updates practical, whereas prior work in this setting either treats them as an open problem or briefly discusses them as an expensive operation. Overall, this design achieves the highest retrieval quality among private search systems, matching that of a non-private index, and is the only one scalable upto a billion vectors. CAROUSEL could be applied to a wide range of applications including private web search, recommendation, and retrieval-augmented generation.

ACKNOWLEDGMENTS

This work was supported by the Air Force Research Laboratory / Information and Spectrum Directorate (AFRL/RF), Rome, NY. Any statements or claims made in this publication are solely those of the authors and do not represent the views or policies of the United States Government.

REFERENCES

- [1] Ishtiaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Coeus: A System for Oblivious Document Ranking and Retrieval. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*. 672–690.
- [2] Martin Albrecht, Melissa Chase, Hao Chen, et al. 2019. Homomorphic Encryption Standard. Cryptology ePrint Archive, Paper 2019/939. <https://eprint.iacr.org/2019/939>
- [3] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the Concrete Hardness of Learning with Errors. *Journal of Mathematical Cryptology* 9, 3 (2015), 169–203. <https://doi.org/10.1515/jmc-2015-0016>
- [4] Matthew Aldridge, Oliver Johnson, and Jonathan Scarlett. 2019. Group Testing: An Information Theory Perspective. *Foundations and Trends in Communications and Information Theory* 15, 3–4 (2019), 196–392. <https://doi.org/10.1561/01000000099>
- [5] Michael Arrington. 2006. AOL Proudly Releases Massive Amounts of Private Data. TechCrunch.
- [6] Hilal Asi, Fabian Boemer, Nicholas Genise, Muhammad Haris Mughees, Tabitha Ogilvie, Rehan Rishi, Guy N. Rothblum, Kunal Talwar, Karl Tarbe, Akshay Wadia, Ruiyu Zhu, and Marco Zuliani. 2024. Wally: Batched Private Nearest Neighbor Search at Scale. <https://doi.org/10.48550/arXiv.2406.06761> arXiv:2406.06761v7 [cs.CR] arXiv preprint, version 7, 10 July 2026.
- [7] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. 2016. MS MARCO: A Human Generated Machine Reading Comprehension Dataset. *arXiv preprint arXiv:1611.09268* (2016). <https://doi.org/10.48550/arXiv.1611.09268>
- [8] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya Razenshteyn, and M. Sadegh Riazi. 2020. SANNS: Scaling Up Secure Approximate k -Nearest Neighbors Search. In *29th USENIX Security Symposium*. 2111–2128.
- [9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*. Springer, 409–437.
- [10] Jung Hee Cheon, Dongwoo Kim, Duhyeon Kim, Hun Hee Lee, and Keewoo Lee. 2019. Numerical Method for Comparison on Homomorphically Encrypted Numbers. In *Advances in Cryptology – ASIACRYPT 2019*. 415–445.
- [11] Wonseok Choi, Jongmin Kim, and Jung Ho Ahn. 2026. Cheddar: A Swift Fully Homomorphic Encryption Library Designed for GPU Architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. <https://doi.org/10.1145/3760250.3762223>
- [12] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*. 169–178.
- [13] Katie Hafner. 2006. Researchers Yearn to Use AOL Logs, but They Hesitate. The New York Times.
- [14] Shai Halevi and Victor Shoup. 2014. Algorithms in HELIB. In *Advances in Cryptology – CRYPTO 2014 (Lecture Notes in Computer Science)*, Juan A. Garay and Rosario Gennaro (Eds.), Vol. 8616. Springer, 554–571. https://doi.org/10.1007/978-3-662-44371-2_31
- [15] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nikolai Zeldovich. 2023. Private Web Search with Tiptoe. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*. Koblenz, Germany, 396–416. <https://doi.org/10.1145/3600006.3613134>
- [16] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. 2023. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. In *32nd USENIX Security Symposium*. 3889–3905.
- [17] Masajiro Iwasaki and Daisuke Miyazaki. 2018. Optimization of Indexing Based on k -Nearest Neighbor Graph for Proximity Search in High-Dimensional Data. *CoRR* abs/1810.07355 (2018).
- [18] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. <https://doi.org/10.1109/TPAMI.2010.57>
- [19] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in One Billion Vectors: Re-Rank with Source Coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. 861–864. <https://doi.org/10.1109/ICASSP.2011.5946540>
- [20] William H. Kautz and Richard C. Singleton. 1964. Nonrandom Binary Superimposed Codes. *IEEE Transactions on Information Theory* 10, 4 (1964), 363–377. <https://doi.org/10.1109/TIT.1964.1053689>
- [21] Duhyeon Kim, Yujin Nam, Wen Wang, Huijing Gong, Ishwar Bhati, Rosario Cammarota, Tajana S. Rosing, Mariano Tepper, and Theodore L. Willke. 2024. GraSS: Graph-based Similarity Search on Encrypted Query. Cryptology ePrint Archive, Paper 2024/2012. <https://eprint.iacr.org/2024/2012>
- [22] Dmitry Kogan and Henry Corrigan-Gibbs. 2021. Private Blocklist Lookups with Checklist. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 875–892.
- [23] Arthur Lazzaretti and Charalampos Papamanthou. 2024. Single Pass Client-Preprocessing Private Information Retrieval. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 5967–5984.
- [24] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. 2024. Hintless Single-Server Private Information Retrieval. In *Advances in Cryptology – CRYPTO 2024 (Lecture Notes in Computer Science)*, Leonid Reyzin and Douglas Stebila (Eds.), Vol. 14928. Springer Nature Switzerland, Cham, 183–217. https://doi.org/10.1007/978-3-031-68400-5_6
- [25] Jingyu Li, Zhicong Huang, Min Zhang, Cheng Hong, Jian Liu, Tao Wei, and Wenguang Chen. 2025. Panther: Private Approximate Nearest Neighbor Search in the Single Server Setting. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 365–379. <https://doi.org/10.1145/3719027.3765190>
- [26] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010*. Springer, 1–23.
- [27] Yu. A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836. <https://doi.org/10.1109/TPAMI.2018.2889473>
- [28] Federico Mazzone, Maarten Everts, Florian Hahn, and Andreas Peter. 2025. Efficient Ranking, Order Statistics, and Sorting under CKKS. In *34th USENIX Security Symposium*. 8541–8558.
- [29] Haowen Pan, Ruiqi Gan, Yunhao Fu, Yintai Sun, Zhou Zhang, Yuxiang Wang, Yi Chen, Bo Zhang, Haoyi Zhou, Yongxin Tong, Zhenyu Guan, Jin Dong, and Song Bian. 2026. HANNs: Low-Storage Non-Interactive Approximate Private Nearest Neighbor Search with Sublinear Comparison Complexity. Cryptology ePrint Archive, Paper 2026/1397. <https://eprint.iacr.org/2026/1397>
- [30] Sacha Servan-Schreiber, Simon Langowski, and Srinivas Devadas. 2022. Private Approximate Nearest Neighbor Search with Sublinear Communication. In *43rd IEEE Symposium on Security and Privacy (S&P)*. 911–929.
- [31] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xi-angyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. 2021. Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 2614–2627.
- [32] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. 2024. Piano: Extremely Simple, Single-Server PIR with Sublinear Server Computation. In *45th IEEE Symposium on Security and Privacy (S&P)*. 4296–4314.
- [33] Mingxun Zhou, Elaine Shi, and Giulia Fanti. 2025. Pacmann: Efficient Private Approximate Nearest Neighbor Search. In *13th International Conference on Learning Representations (ICLR)*. 99396–99416.
- [34] Jinhao Zhu, Liana Patel, Matei Zaharia, and Raluca Ada Popa. 2025. Compass: Encrypted Semantic Search with High Accuracy. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Boston, MA, 915–938.