SIMSPARSE: SEMANTIC SEARCH CAN STAY SPARSE END TO END

Anonymous authors
Paper under double-blind review

ABSTRACT

Scaling semantic retrieval requires balancing corpus storage against the computation needed to access it. Conventional graph indexes retain dense document embeddings for low-latency search, while low-storage approaches often shift part of this cost to query-time document re-encoding. Learned sparse representations offer a different design point: their compact codes support direct scoring without document re-encoding, but compact scoring alone does not determine which documents should be evaluated. We introduce SIMSPARSE, a sparse-native graph retrieval system that uses sparse features to organize and navigate the corpus. Shared active features generate candidate connections, active query features select entry points, and sparse inner products guide graph construction, traversal, and ranking. The deployed index retains only sparse codes, bounded-degree graph links, and short routing lists, without dense corpus embeddings or online document re-encoding. Across six benchmarks and two embedding backbones, SIMSPARSE achieves competitive retrieval quality with low storage and query latency, while feature-induced topology substantially improves retrieval when sparse representations are held fixed. On the 60-million-document RPJ-Wiki corpus, at matched performance, SIMSPARSE achieves single-digit-GiB storage and millisecond-scale latency, approaching the storage efficiency of recomputation-based indexing while retaining the low latency of dense graph search.

1 INTRODUCTION

External memory allows AI systems to access knowledge beyond their parameters. Retrieval-augmented models consult external documents, while agents and persistent assistants retrieve information across tasks and interactions (Lewis et al., 2020; Yao et al., 2022; Packer et al., 2023). As the corpora supporting these applications grow, storing the representations used for semantic matching becomes increasingly costly. Reducing this storage footprint, however, can shift the burden to query-time computation needed to preserve retrieval quality. Therefore, balancing these demands becomes a central consideration in how external memory is represented and accessed at scale.

Existing systems make different choices about what to retain and what to compute at query time. Conventional approaches, including HNSW and DiskANN, retain full dense embeddings for scoring, supporting fast retrieval at a substantial storage cost (Malkov & Yashunin, 2018; Jayaram Subramanya et al., 2019). LEANN reduces the full embedding storage by using compressed vectors for candidate screening and recomputing selected documents' full embeddings to compensate for compression-induced accuracy loss, which leads to additional query-time computation (Wang et al., 2025). This suggests a different design point: use a compact representation that can serve directly as the scoring representation, rather than as a proxy from which dense embeddings must be recovered. Learned sparse representations such as CSR provide this possibility by learning semantic matching in a high-dimensional space with only a few active features per document (Wen et al., 2025; Guo et al., 2026). Stored as feature identifiers and values, these codes support direct sparse inner-product scoring without online document re-encoding.

At large scale, however, compact scoring alone does not make retrieval efficient: a query may still need to evaluate many documents. At tens of millions of documents, exhaustive sparse scoring remains costly, and inverted indexes can require scanning long posting lists. Graph search offers an alternative by restricting scoring to a small explored subset of the corpus: starting from a few entry

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

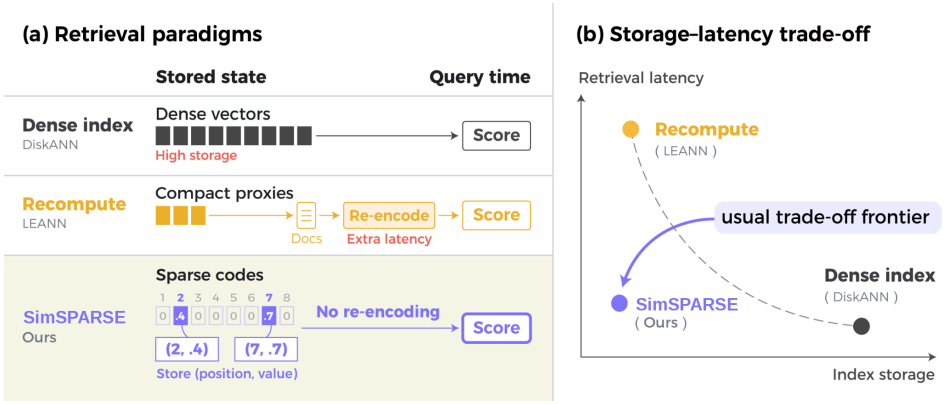


Figure 1: **Overview of SIMSPARSE. (a) Retrieval paradigms.** Dense indexes (e.g., DiskANN (Jayaram Subramanya et al., 2019), HNSW (Malkov & Yashunin, 2018)) retain document embeddings for direct scoring, while recomputation-based indexes (e.g., LEANN (Wang et al., 2025)) reduce stored representations by shifting document encoding to query time. SIMSPARSE instead performs graph search directly over compact sparse codes, using the same codes for routing, traversal, and scoring. **(b) Storage-latency trade-off.** SIMSPARSE reaches a new operating point, combining low index storage with low query latency at matched retrieval performance.

points, search follows document-to-document connections toward promising regions rather than evaluating all possible candidates. Therefore, its efficiency depends on whether the graph provides useful connections and starting points that make relevant documents reachable within a small search budget. Our key insight is that sparse features provide useful structure for both: shared active features identify potential neighbors, while active query features identify promising starting points. These features can therefore support not only compact semantic scoring but also the organization and navigation of the corpus. This suggests a route to combining low storage with low latency: use the structure of sparse representations to guide access to them.

In this paper, we introduce **SimSPARSE**, which uses sparse features to construct graph connections, select query-conditioned entry points, and score traversal. Its deployed index retains only sparse codes, bounded-degree graph links, and short routing lists. Across six benchmarks and two embedding backbones, SIMSPARSE achieves competitive retrieval quality while maintaining low query latency and index storage. We then evaluate scalability on the 60-million-document RPJ-Wiki corpus, measuring storage and query latency at matched retrieval performance. As shown in Figure 1, SIMSPARSE approaches the storage efficiency of recomputation-based indexing while retaining the low query latency characteristic of dense graph search. Our contributions are:

- We identify and realize a new storage–latency operating point for large-scale semantic retrieval: sparse codes serve directly as the stored scoring representation while also providing the structure for efficient navigation, avoiding the need to trade dense embedding storage for query-time document re-encoding.
- We introduce SIMSPARSE, a sparse-native graph retrieval system that turns the structure of learned sparse representations into a navigable topology. Shared active features induce document connectivity, while active query features provide query-conditioned entry points, enabling the same sparse representation to govern both corpus organization and search.
- We evaluate SIMSPARSE across six retrieval benchmarks, two embedding backbones, and the 60M-document RPJ-Wiki corpus, showing competitive retrieval quality and, at matched performance, storage approaching recomputation-based methods with latency comparable to dense graph indexes.

2 RELATED WORK

Compact representations and index storage. Reducing retrieval storage requires deciding both how representations are compressed and what scoring information remains available. PQ (Jegou

et al., 2010) replaces subvectors with codebook assignments, while MRL (Kusupati et al., 2022) learns dense embeddings that remain effective at reduced dimensionalities. At the system level, compact representations play different roles: DiskANN (Jayaram Subramanya et al., 2019) uses compressed vectors for in-memory navigation while retaining full-precision vectors on SSD, whereas LEANN (Wang et al., 2025) further reduces stored embedding state by selectively reconstructing document representations at query time. These designs trade stored scoring information for storage access or recomputation. SIMSPARSE instead uses sparse document codes directly for deployed scoring, avoiding dense corpus embeddings and online document re-encoding.

Sparse representations and candidate access. Learned sparsity makes semantic representations addressable through a small set of active features, but this does not by itself determine how many documents must be examined. SPLADE (Formal et al., 2021b) exposes vocabulary-aligned features through learned weighting and expansion, while CSR (Wen et al., 2025; Guo et al., 2026) learns latent sparse coordinates, with CSRv2 pushing toward the ultra-sparse regime. Existing sparse indexes primarily exploit these coordinates through selective posting access. Seismic (Bruch et al., 2024a) prunes and groups postings into geometric blocks whose summaries guide candidate selection, and SeismicWave (Bruch et al., 2024b) augments this process with ordered block traversal and one-hop graph expansion. SIMSPARSE instead uses postings mainly to expose structure: full postings are consumed offline to construct graph connectivity, while only short feature-conditioned routing lists are retained online. Retrieval then shifts from posting traversal to repeated graph expansion under the same sparse scoring function.

Graph topology and query routing. Once retrieval is formulated as graph search, efficiency depends on both which paths the graph makes available and where the query enters them. HNSW (Malkov & Yashunin, 2018) creates multi-scale routes through a hierarchy of proximity graphs, while NSG (Fu et al., 2017) improves navigability through neighbor pruning and connectivity repair. Query-dependent initialization provides a complementary mechanism: LADR (Kulkarni et al., 2023) uses lexical retrieval to seed dense-scored graph exploration, with its adaptive variant repeatedly expanding promising neighborhoods. These methods typically separate access organization from query scoring or routing. SIMSPARSE instead derives both from the sparse representation itself: shared active features propose graph connections, active query features identify entry candidates, and sparse inner products govern traversal and ranking.

Appendix E further situates SIMSPARSE along three interacting axes: the information retained for scoring, the mechanisms used to expose candidates, and the construction and traversal of search paths.

3 BACKGROUND

Indexed similarity search. Let $\mathcal{D} = \{d_1, \ldots, d_N\}$ be a document corpus and q a query. Document and query encoders ϕ_d and ϕ_q map them to a common representation space $\mathbb{R}^D$, giving $\mathbf{d}_i = \phi_d(d_i)$ and $\mathbf{q} = \phi_q(q)$. Given a similarity function $s : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$, the exact retrieval target $\mathcal{R}_k^\star(q) \subseteq \mathcal{D}$ consists of the k documents with the largest scores $s(\mathbf{q}, \mathbf{d}_i)$. An index $\mathcal{I} = \mathrm{Index}(\{\mathbf{d}_i\}_{i=1}^N)$ aims to recover this set with less work than exhaustive scoring, returning an approximate size-k result $\mathcal{R}_k(q; \mathcal{I}) \subseteq \mathcal{D}$. Search can be accelerated through inverted lists or partitions (Sivic & Zisserman, 2003), scoring with quantized vectors (Jegou et al., 2010), or graph traversal (Wang et al., 2025; Jayaram Subramanya et al., 2019; Malkov & Yashunin, 2018).

Storage–recomputation trade-off. An index stores document representations together with the structures used to access them. Retaining one D-dimensional dense vector per document at b bytes per coordinate requires approximately:

$$B_{\mathrm{mat}} \approx NDb + B_{\mathrm{struct}}^{\mathrm{mat}}, \tag{1}$$

where $B_{\mathrm{struct}}^{\mathrm{mat}}$ denotes the persistent index state beyond the stored dense vectors.

LEANN (Wang et al., 2025) reduces retained vector storage through compact proxies and graph pruning, and re-encodes selected documents during search. For an already encoded query that re-encodes m selected documents, the corresponding storage and query-time costs can be summarized as:

$$\begin{aligned} B_{\mathrm{rec}} &\approx NB_{\mathrm{proxy}} + B_{\mathrm{struct}}^{\mathrm{rec}}, \\ T_{\mathrm{query}} &\approx T_{\mathrm{search}} + T_{\mathrm{reenc}}(m), \end{aligned} \tag{2}$$

where B_{proxy} is the per-document proxy storage and $B^{\text{rec}}_{\text{struct}}$ denotes the other persistent state required by the search structure. T_{search} captures search over this retained state, while $T_{\text{reenc}}(m)$ captures the time spent encoding the m selected documents. Thus, recomputation-based indexing reduces persistent representation storage by introducing query-dependent document encoding.

Compact document representations. Reducing the size of the scoring representation provides another way to reduce storage. Product Quantization (PQ) (Jegou et al., 2010) represents subvectors using codebook identifiers, while Matryoshka Representation Learning (MRL) (Kusupati et al., 2022) learns embeddings that remain effective when truncated to shorter dimensionalities. Sparse representations take a different approach, retaining a high-dimensional feature space while activating only a small subset of coordinates. For example, CSR (Wen et al., 2025) maps a pretrained document embedding $\mathbf{h} \in \mathbb{R}^D$ to a sparse code $\mathbf{z} \in \mathbb{R}^H$ using a learned encoder:

$$\mathbf{z} = \mathrm{TopK}_{K_d}(\mathrm{ReLU}(\mathbf{W}_{\text{enc}}\mathbf{h} + \mathbf{b}_{\text{enc}})), \qquad \|\mathbf{z}\|_0 \le K_d \ll H. \tag{3}$$

where TopK_{K_d} retains the K_d largest activations and sets the remaining coordinates to zero. The encoder is trained with reconstruction of the input embedding together with contrastive learning in the sparse space. The resulting code can be stored using only its active feature identifiers and values, while supporting direct scoring in the sparse space without reconstruction of the original dense embedding.

4 METHOD

SIMSPARSE uses sparse features to organize both document connections and query entry points (Section 4.1). We describe how the index is constructed from fixed sparse codes (Section 4.2) and how queries enter and traverse the resulting graph (Section 4.3), then analyze its storage and search costs (Section 4.4).

4.1 OVERVIEW

SIMSPARSE turns fixed sparse representations into a graph that can be entered and navigated using the same feature space, where each document d_i is represented by a sparse vector $\mathbf{z}_i$. Shown in Figure 2, during index construction, these active features are used to propose graph connections in three complementary ways: local windows connect documents with similar activation levels on a shared feature, landmarks expose strongly activated documents, and signatures identify documents that share combinations of strong features. The resulting candidates are scored and pruned to form a directed graph with at most M outgoing links per document.

At query time, the sparse query vector $\mathbf{z}_q$ activates only a small set of features. Each active feature retrieves a short list of representative documents, and the highest-scoring candidates are used as entry points for graph search. Both entry selection and graph traversal use the same sparse inner product:

$$s(q, i) = \mathbf{z}_q^\top \mathbf{z}_i = \sum_{t \in S_q \cap S_i} z_{q,t} z_{i,t}, \tag{4}$$

where S_q and S_i denote the sets of active features in the query and document, respectively, and $z_{q,t}$ and $z_{i,t}$ are their activation values on feature t. Sparse features thus play a dual role: feature identities determine graph connections and query entry, while activation values score entries, traversal paths, and results.

4.2 GRAPH CONSTRUCTION

We construct a directed graph over N documents, with at most M outgoing links per document. For each feature t, a posting list P_t contains documents activating t, ordered by increasing activation magnitude. Document i selects up to F active features for posting-based candidate generation, favoring strong activations and rare features through inverse document frequency (IDF) (Robertson, 2004):

$$T_i = \operatorname*{Top}_{F,\, t \in S_i} |z_{i,t}| w_t^\alpha, \qquad w_t = 1 + \log \frac{N+1}{|P_t|+1}. \tag{5}$$

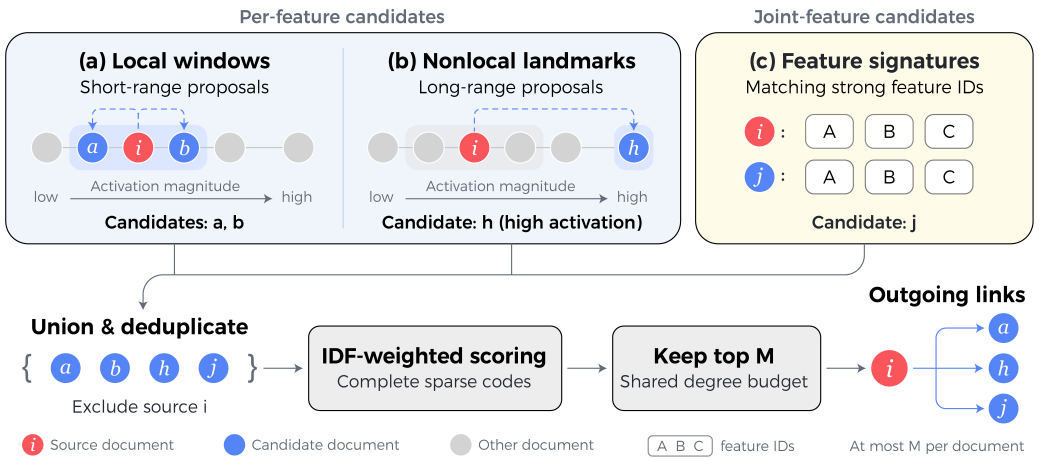


Figure 2: **Graph candidates from sparse features.** **(1) Local neighbors** come from windows around a document's position in activation-ordered postings. **(2) Landmarks** are high-activation documents selected independently of these windows. **(3) Joint matches** share signatures of strong feature identifiers. The candidate sets are merged and scored using the complete sparse codes; at most M outgoing links are retained per document.

where S_i is the active feature set of document i, $|P_t|$ counts documents activating t, and α controls the rarity preference. Top_m returns the identifiers of up to m highest-scoring items.

Local windows. Selecting only the strongest activations would concentrate proposals on the same few documents per feature. Local windows provide source-dependent, short-range proposals along each posting. Let $r_t(i)$ be the position of document i in P_t. A radius-R window selects up to R documents on either side:

$$C_{\mathrm{local}}(i;R) = \bigcup_{t\in T_i} \{j \in P_t : 0 < |r_t(j) - r_t(i)| \le R\}. \tag{6}$$

Starting at $R = R_0$, we double R while the pool contains fewer than ρM documents, where ρ is an oversampling factor relative to the degree budget. We stop after an expansion adds fewer than δ new documents, or before doubling would exceed the maximum radius $R_{\max}$.

Nonlocal landmarks. For our nonnegative codes, stronger activation on a shared feature increases that feature's score contribution. We therefore add up to L_h documents with the largest activation magnitudes from each selected posting. These landmarks provide direct proposals to strong feature representatives that could otherwise require several local steps to reach. Selection is independent of the source's position, so landmarks can overlap with local candidates. Their contribution is bounded by FL_h proposals before deduplication.

Feature signatures. Agreement on several strong features provides evidence of similarity that is not captured by proximity along any single posting. We therefore represent document i by the identifiers of its L strongest active features:

$$\sigma(i) = \mathrm{sort}_{\mathrm{ID}}\left(\mathrm{Top}_{\substack{L\\ t\in S_i}} |z_{i,t}|\right). \tag{7}$$

Sorting by identifier gives the selected feature set a canonical representation. We retrieve candidates from the bucket sharing the same nonempty signature. To improve coverage when exact L-feature combinations are sparse, we also index an L'-feature backoff key, with $L' < L$, obtained from the canonical signature. Each exact or backoff lookup contributes at most B documents in document-ID order, independently of posting position.

Selecting graph links. Let $C(i)$ be the union of all three candidate sets, with duplicates and the source document removed. We score every candidate using all shared active features, including

those that did not generate the candidate:

$$s_{\text{build}}(i,j) = \sum_{t \in S_i \cap S_j} z_{i,t} z_{j,t} w_t^{\beta},$$
$$A_i = \operatorname*{Top}_{M}{}_{j \in C(i)} s_{\text{build}}(i,j), \qquad |A_i| \le M. \tag{8}$$

where A_i contains the outgoing neighbors of document i and $\beta \ge 0$ controls IDF weighting in edge scores independently of feature selection. Appendices A.2 and Appendix D.1 give further construction details and graph diagnostics.

4.3 Query Entry Selection and Graph Search

Graph construction determines available routes, while query routing determines where traversal begins. To accommodate queries aligned with different graph regions, we derive entry candidates from their active features. For each feature t, an *entry router* stores up to L_r documents with the largest activation magnitudes on that feature. The active query features therefore expose a small set of feature-conditioned candidates:

$$L(q) = \bigcup_{t \in S_q} R_t, \qquad E(q) = \operatorname*{Top}_{E}{}_{i \in L(q)} s(q,i). \tag{9}$$

The router lists provide candidate starting points, while the complete sparse query score determines which of them initialize search. Because $s(q,i)$ uses all shared active features, entry selection can combine evidence across coordinates rather than committing to any single query feature. The entry budget E is shared across features, allowing the query to start from several promising regions of the graph. If no router candidate is available, we use a stored maximum-norm document as a fallback.

Starting from these query-conditioned entries, we perform best-first graph traversal (Malkov & Yashunin, 2018). At each step, we expand the highest-scoring unexpanded document and score its newly reached neighbors using Eq. 4. Traversal stops after at most C expansions, or when no further document remains to be expanded, and returns the top-k scored documents encountered during search. Thus, active feature identifiers determine where search enters the graph, while the same sparse similarity used for retrieval guides subsequent traversal. Appendix A.3 gives the exact traversal and stopping rules.

4.4 Storage and Search Complexity

SimSPARSE keeps only the state needed for sparse scoring and graph navigation at deployment: the sparse document codes, at most M outgoing links per document, and at most L_r routed documents per feature. Let K_d bound the number of active features per document, b_s denote the storage per active feature identifier–value pair, and b_n the storage per document identifier. With N documents and H possible features, the deployed index therefore satisfies

$$B_{\text{index}} \le \underbrace{NK_d b_s}_{\text{sparse codes}} + \underbrace{NMb_n}_{\text{graph links}} + \underbrace{HL_r b_n}_{\text{entry router}} + B_{\text{offsets}}, \tag{10}$$

where postings, feature signatures, and other structures used to construct the graph are not included, since they are all discarded once construction is complete.

Search is similarly controlled by the routing and traversal budgets. A query with K_q active features obtains at most $K_q L_r$ routed candidates, while at most C graph expansions expose CM outgoing links. Each encountered document is scored by intersecting the query and document supports, which takes $O(K_q + K_d)$ time for sorted sparse codes. With score reuse, the resulting sparse-scoring work is bounded by

$$T_{\text{score}} = O((K_q L_r + CM + 1)(K_q + K_d)). \tag{11}$$

The first factor bounds how many documents can be exposed by routing and graph expansion, while the second is the cost of evaluating one sparse similarity. Appendix A.4 develops these bounds further, including exact storage accounting and the resource costs of graph construction and search.

Backbone	Method	Latency	Storage	HotpotQA	NQ	ClimateFever	Fever	DBpedia	MSMARCO	Avg.
Contriever	HNSW	3.88×	28.57×	15.16	26.45	9.83	33.87	**33.99**	26.67	24.33
	DiskANN	2.79×	39.38×	14.48	26.43	9.98	34.07	33.31	**27.52**	24.30
	IVF	1.16×	23.60×	13.93	30.78	16.17	28.95	30.74	19.98	23.43
	PQ	0.86×	0.67×	11.97	23.07	8.94	22.42	19.81	17.60	17.30
	LEANN	201.27×	2.11×	18.75	36.15	23.54	**59.76**	27.27	21.41	31.15
	SIMSPARSE	1.00×	1.00×	**28.76**	**39.25**	**33.49**	44.89	24.44	23.08	**32.32**
BGE	HNSW	6.95×	29.49×	28.34	**59.02**	30.11	**83.72**	39.25	45.46	47.65
	DiskANN	8.17×	40.65×	28.13	57.39	31.76	82.29	41.99	44.74	**47.72**
	IVF	1.23×	27.38×	27.12	55.19	30.08	74.50	**44.79**	**45.47**	46.19
	PQ	1.04×	0.58×	26.01	47.57	25.48	74.54	34.89	38.10	41.10
	LEANN	1505.68×	2.03×	19.46	51.92	31.68	73.05	36.22	35.72	41.34
	SIMSPARSE	1.00×	1.00×	**31.89**	54.76	**41.37**	71.49	33.50	42.35	45.89

Table 1: **Comparison on six BEIR (Thakur et al., 2021) benchmarks with two backbones.** Retrieval quality is evaluated with nDCG@10 (%) while latency and index storage are normalized independently within each backbone to SIMSPARSE for a clearer comparison. The first/second largest value on each benchmark is **bold** / underlined.

5 EXPERIMENTS

5.1 BENCHMARK PERFORMANCE

End-to-end benchmark evaluation. For a controlled end-to-end comparison, we use Contriever (Izacard et al., 2021) and BGE-base-en-v1.5 (Xiao et al., 2024) as backbones, following the standard setup of LEANN (Wang et al., 2025). Table 1 compares SIMSPARSE with dense, compressed, and recomputation-based indexes across six BEIR (Thakur et al., 2021) benchmarks. With Contriever, SIMSPARSE achieves the highest average nDCG@10 (32.32). With BGE, it reaches 45.89, close to the strongest dense baselines while reducing index storage by roughly 30–40×. Detailed experiment setup is available in Appendix C.

Matched-performance evaluation at scale. Motivated by LEANN (Wang et al., 2025), we use hybrid reference top-3 neighbors as ground truth and tune each method to the smallest search-time setting that reaches 90% Recall@3, reporting latency and storage at this matched-recall operating point. As shown in Table 2, SIMSPARSE achieves single-digit-GiB storage with millisecond-scale latency across both backbones, approaching the footprint of recomputation-based indexing while retaining the speed of dense search. The CSR baselines further show that compact sparse scoring alone is insufficient for efficient retrieval at this scale: exhaustive CSR search requires roughly 192 s per query, while CSR-IVF reduces this to about 90 ms; SIMSPARSE reaches about 9 ms at comparable storage, demonstrating the benefit of using sparse features not only for scoring but also for corpus navigation. Appendix C provides additional details on the evaluation procedure.

5.2 DISCUSSION ON THE SETTING OF KEY PARAMETERS IN SIMSPARSE

Graph degree. The graph degree controls the maximum number of outgoing neighbors retained for each document. A larger degree improves connectivity but increases the number of neighbors scored per expansion. Figure 3 (a) shows that performance improves rapidly at small degrees and then saturates as the graph becomes denser. we therefore set graph degree $M = 16$, which is near the knee of the performance–latency curve.

Candidate queue size. The candidate queue controls how many promising discovered documents are retained during best-first search for possible future expansion. A larger queue allows more alternative search paths but increases overhead. Figure 3 (b) shows rapidly diminishing performance gains beyond moderate sizes, despite continued growth in latency. We therefore choose a queue size of 128, which preserves most of the performance improvement while keeping the search substantially cheaper than larger queue settings.

Backbone	Method	Index Storage	NQ	GPQA	TriviaQA	HotpotQA	Avg.
Contriever	HNSW	189.59	0.016	**0.007**	0.012	0.011	0.012
	DiskANN	271.48	0.009	0.012	0.021	0.011	0.013
	IVF	171.93	2.17	1.89	2.76	2.53	2.338
	LEANN	3.81	2.78	2.94	2.77	3.08	2.893
	CSR-exhaustive	3.28	192.37	186.42	203.05	187.90	192.433
	CSR-IVF	3.32	0.098	0.084	0.096	0.093	0.093
	SIMSPARSE	3.49	**0.008**	0.013	**0.009**	**0.007**	**0.009**
BGE	HNSW	187.43	0.007	0.015	0.018	0.011	0.013
	DiskANN	281.9	**0.005**	0.008	0.013	**0.007**	**0.008**
	IVF	175.72	2.41	2.08	2.29	2.18	2.24
	LEANN	4.17	2.53	2.81	2.92	3.04	2.83
	CSR-exhaustive	3.28	191.84	188.22	202.76	186.31	192.28
	CSR-IVF	3.34	0.086	0.091	0.085	0.097	0.09
	SIMSPARSE	4.02	**0.005**	**0.011**	**0.012**	0.008	0.009

Table 2: **Matched-performance comparison on the 60M-document RPJ-Wiki corpus**. Index storage is calculated in GiB and latency is evaluated across four query domains in seconds per query. The first/second smallest latency on each benchmark is **bold**/underlined.

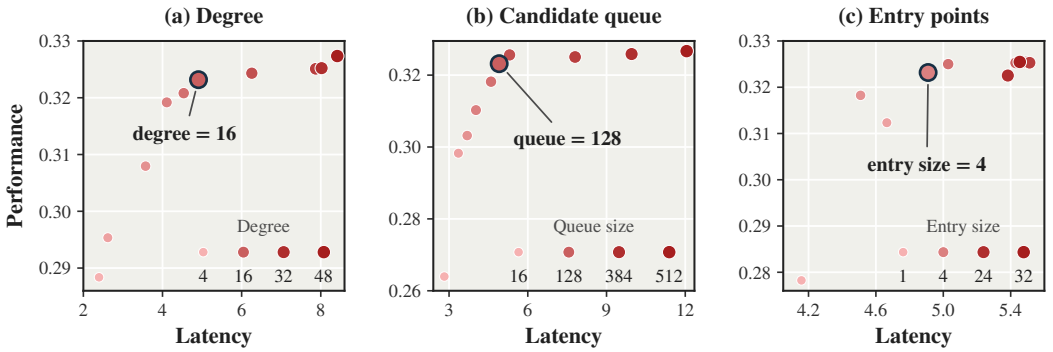


Figure 3: **Performance–latency trade-offs for SIMSPARSE parameters. (Left)** graph degree; **(Middle)** candidate queue size; **(Right)** number of entry points.

Number of entry points. The entry count E specifies how many query-conditioned starting documents initialize graph traversal. Multiple entries reduce reliance on a single starting region but add initial scoring and frontier candidates. As shown in Figure 3 (c), moving from one to a few entry points provides a noticeable performance gain, whereas increasing the entry count further brings only limited improvement. We therefore set $E = 4$, which provides sufficient diversity in query initialization with only a modest latency overhead.

5.3 EMPIRICAL ANALYSIS

Ablation. Table 3 reports ablations of SIMSPARSE components on six BEIR (Thakur et al., 2021) benchmarks with Contriever. IDF-aware construction improves feature selection, while landmarks and signatures add nonlocal routes beyond short posting neighborhoods. Their cumulative gains show that effective retrieval benefits from complementary graph structure. Multi-entry search provides a further gain on top of the fully constructed graph, indicating distinct roles for topology and routing: construction determines which useful paths exist, whereas query-conditioned entries determine where search begins.

Storage-aware Precision. Figure 4 (Left) shows that the preferred numerical precision depends strongly on the available per-document storage budget. Under tight budgets, aggressive quantization is most effective, as lower bit-width leaves more capacity for storing nonzero activations. As the budget increases, the optimal choice shifts to higher precision. This pattern suggests a simple

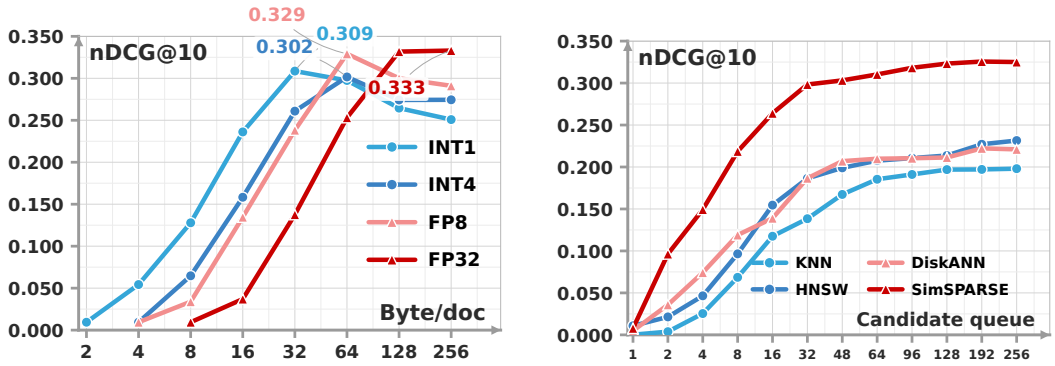


Figure 4: **(Left)**: Storage-aware precision selection under different per-document budgets. **(Right)**: Performance of different graph topologies with fixed sparse representations and varying queue sizes.

Graph Construction				Search	Performance	
Local	IDF	Landmark	Signature	Multi-entry	nDCG@10 (%)	Latency (ms)
✓	✗	✗	✗	✗	28.10	4.27
✓	✓	✗	✗	✗	29.72	3.72
✓	✓	✓	✗	✗	30.49	3.61
✓	✓	✓	✓	✗	31.27	3.34
✓	✓	✓	✓	✓	32.32	3.86

Table 3: **Ablation of SIMSPARSE's graph construction and query routing.** Construction components are added cumulatively, with multi-entry routing enabled in the final stage. Average nDCG@10(%) across six benchmarks in BEIR (Thakur et al., 2021) and average query latency (ms) are reported.

storage-aware rule: use lower precision when representation capacity is the bottleneck, and increase precision once sufficient storage is available to preserve activation values. Appendix D.2 gives the exact mapping from each storage budget and numerical precision to its supported TopK, together with a complementary visualization of this capacity-precision trade-off.

Graph Topology. Figure 4 (Right) isolates the role of graph topology by holding the Contriever-based CSR (Wen et al., 2025) representations fixed and changing only how the graph is constructed. Increasing the candidate queue improves all methods by allowing search to explore more of the graph, but the gap between different topologies remains substantial even under larger search budgets. In particular, SimSPARSE reaches an average nDCG@10 above 0.32, while the alternative constructions remain below 0.24. This controlled comparison shows that sparse representations alone do not guarantee effective graph retrieval: their signal must be organized into navigable paths. SIMSPARSE's feature-induced topology makes more of the same representation accessible within a limited search budget. Additional fixed-representation comparisons and graph connectivity diagnostics are provided in Appendix D.1.

6 CONCLUSION

We introduced SIMSPARSE, a sparse-native graph retrieval system that uses sparse codes for both scoring and navigation. Feature-induced graph connections and query-conditioned entry points eliminate dense corpus embeddings and online document re-encoding while retaining efficient graph search. Across benchmarks and backbones, SimSPARSE achieves competitive retrieval quality with low storage and latency, scaling to 60M documents with a single-digit-GiB index and millisecond-level search. Controlled topology analysis further shows that compact sparse representations alone are insufficient: effective retrieval also requires organizing their signal into navigable paths. These results motivate the joint design of sparse representations and retrieval topology.

AI USE STATEMENT

In this work, we used generative AI tools for assisting in the writing of proofs, implementing methods, helping develop theoretical models and assisting with translation. We have not used generative AI tools for proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, supporting qualitative and thematic data analysis, or interpreting results. Generating synthetic datasets and cleaning or reformatting datasets are not applicable to this work. Additionally, we used generative AI tools for modifying scientific figures and summarizing existing literature. We have reviewed all AI-assisted work. For example, all AI-assisted proofs were independently reviewed and verified by the authors, and language edits were reviewed to ensure that they preserved the intended technical meaning and claims. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work adheres to the ICLR Code of Ethics. Although our proposed methods are broadly applicable, their deployment in real-world scenarios may carry societal considerations, particularly regarding bias, fairness, and privacy. We advocate for responsible application of our techniques and disclose that we have no conflicts of interest.

REPRODUCIBILITY STATEMENT

We provide comprehensive details of our methodology, datasets, model architectures, and evaluation protocols in both the main text and Appendix. Full mathematical derivations and additional experimental results are included in the Appendix. Our code and scripts will be released upon acceptance.

REFERENCES

Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, et al. Ms marco: A human generated machine reading comprehension dataset. *arXiv preprint arXiv:1611.09268*, 2016.

Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia-a crystallization point for the web of data. *Journal of web semantics*, 7(3):154–165, 2009.

Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. Efficient inverted indexes for approximate retrieval over learned sparse representations. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 152–162, 2024a.

Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. Pairing clustered inverted indexes with κ-nn graphs for fast approximate retrieval over learned sparse representations. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pp. 3642–3646, 2024b.

Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. *Advances in Neural Information Processing Systems*, 34:5199–5212, 2021.

Leonardo Delfino, Domenico Erriquez, Silvio Martinico, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. kann olo: Sweet and smooth approximate k-nearest neighbors search. In *European Conference on Information Retrieval*, pp. 400–406. Springer, 2025.

Thomas Diggelmann, Jordan Boyd-Graber, Jannis Bulian, Massimiliano Ciaramita, and Markus Leippold. Climate-fever: A dataset for verification of real-world climate claims. *arXiv preprint arXiv:2012.00614*, 2020.

Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. Splade v2: Sparse lexical and expansion model for information retrieval. *arXiv preprint arXiv:2109.10086*, 2021a.

Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. Splade: Sparse lexical and expansion model for first stage ranking. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*, pp. 2288–2292, 2021b.

Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

Cong Fu, Changxu Wang, and Deng Cai. High dimensional similarity search with satellite system graph: Efficiency, scalability, and unindexed query compatibility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(8):4139–4150, 2021.

Jianyang Gao and Cheng Long. Rabitq: Quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2946–2953, 2013.

Lixuan Guo, Yifei Wang, Tiansheng Wen, Yifan Wang, Aosong Feng, Bo Chen, Stefanie Jegelka, and Chenyu You. Csrv2: Unlocking ultra-sparse embeddings. *International Conference on Learning Representations*, 2026.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pp. 3887–3896. PMLR, 2020.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning. *arXiv preprint arXiv:2112.09118*, 2021.

Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in neural information processing Systems*, 32, 2019.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, 2017.

Hrishikesh Kulkarni, Sean MacAvaney, Nazli Goharian, and Ophir Frieder. Lexically-accelerated dense retrieval. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2023. doi: 10.1145/3539618.3591715. URL https://arxiv.org/abs/2307.16779.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. Matryoshka representation learning. *Advances in Neural Information Processing Systems*, 35:30233–30249, 2022.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

Jimmy Lin and Xueguang Ma. A few brief notes on deepimpact, coil, and a conceptual framework for information retrieval techniques. *arXiv preprint arXiv:2106.14807*, 2021.

Kejing Lu, Chuan Xiao, and Yoshiharu Ishikawa. Probabilistic routing for graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2402.11354*, 2024.

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.

Antonio Mallia, Omar Khattab, Torsten Suel, and Nicola Tonellotto. Learning passage impacts for inverted indexes. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*, pp. 1723–1727, 2021.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Stephen Robertson. Understanding inverse document frequency: On theoretical arguments for IDF. *Journal of Documentation*, 60(5):503–520, 2004. doi: 10.1108/00220410410560582.

Sivic and Zisserman. Video google: A text retrieval approach to object matching in videos. In *Proceedings ninth IEEE international conference on computer vision*, pp. 1470–1477. IEEE, 2003.

Mariano Tepper, Ishwar Singh Bhati, Cecilia Aguerrebere, Mark Hildebrand, and Ted Willke. Leanvec: Searching vectors faster by making them fit. *arXiv preprint arXiv:2312.16335*, 2023.

Mariano Tepper, Ishwar Singh Bhati, Cecilia Aguerrebere, and Ted Willke. Gleanvec: Accelerating vector search with minimalist nonlinear dimensionality reduction. *arXiv preprint arXiv:2410.22347*, 2024.

Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. Beir: A heterogenous benchmark for zero-shot evaluation of information retrieval models. *arXiv preprint arXiv:2104.08663*, 2021.

James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. Fever: a large-scale dataset for fact extraction and verification. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 809–819, 2018.

Yichuan Wang, Zhifei Li, Shu Liu, Yongji Wu, Ziming Mao, Yilong Zhao, Xiao Yan, Zhiying Xu, Yang Zhou, Ion Stoica, et al. Leann: A low-storage vector index. *arXiv preprint arXiv:2506.08276*, 2025.

Tiansheng Wen, Yifei Wang, Zequn Zeng, Zhong Peng, Yudi Su, Xinyang Liu, Bo Chen, Hongwei Liu, Stefanie Jegelka, and Chenyu You. Beyond matryoshka: Revisiting sparse coding for adaptive representation. *arXiv preprint arXiv:2503.01776*, 2025.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. C-pack: Packed resources for general chinese embeddings. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*, pp. 641–649, 2024.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 2369–2380, 2018.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Qiang Yue, Xiaoliang Xu, Yuxiang Wang, Yikun Tao, and Xuliyuan Luo. Routing-guided learned product quantization for graph-based approximate nearest neighbor search. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 4870–4883. IEEE, 2024.

Minjia Zhang, Wenhan Wang, and Yuxiong He. Grasp: Optimizing graph-based nearest neighbor search with subgraph sampling and pruning. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, pp. 1395–1405, 2022.

A ALGORITHMIC DETAILS OF SIMSPARSE

In this section, we discuss the complete construction and search procedures for SIMSPARSE. Starting from fixed sparse document codes, we first organize documents by their active features and use this structure to generate candidate graph links. We then build a feature-conditioned entry router and perform best-first search from query-dependent starting points. Finally, we give the exact storage accounting and computational costs of construction and retrieval. Table 4 summarizes the notation used throughout this section, while Algorithms 1 and 2 give the full procedures.

Table 4: Notation for sparse codes, routing structures, and resource budgets.

Symbol	Meaning
N, H	Number of documents and possible sparse features.
$\mathbf{z}_i, S_i$	Document i's sparse code and its nonzero feature identifiers.
$\mathbf{z}_q, S_q$	Query code and its nonzero feature identifiers.
K_d, K_q	Maximum document support size and the query support size.
$P_t, r_t(i), w_t$	Feature t's ordered posting, document i's zero-based rank, and IDF weight.
T_i, F	Features used to propose candidates for i, and their maximum number.
$C(i), A_i, M, Q$	Candidate set, outgoing neighbors, degree bound, and post-score candidate cap.
L, L', B	Exact signature size, backoff size, and cap on each signature bucket.
L_h, L_r	Landmark candidates per feature and stored router entries per feature.
E, W, C, k	Entry count, candidate-queue capacity, expansion budget, and requested result count.

A.1 ORGANIZING SPARSE FEATURES FOR CONSTRUCTION

Sparse-code representation. The construction procedure takes fixed sparse document codes $\{\mathbf{z}_i\}_{i=1}^N$ as input. We represent each document by its nonzero feature identifiers and activation values, with duplicate coordinates merged, explicit zeros removed, and feature identifiers stored in a canonical order. These conventions do not change the sparse representation; they ensure that posting construction, tie breaking, and sparse intersections are deterministic. When needed, document codes may additionally be normalized before index construction. In our implementation, construction operates on a float32 CSR view of the codes; the released code supports both sparse NPZ and fixed-support compact storage formats.

Selecting informative construction features. For each document, we use only a small subset of its active features to propose graph candidates. Intuitively, a useful construction feature should be both strongly activated for the document and selective across the corpus. We capture the latter using an IDF-style weight.

For each feature t, let P_t denote the set of documents with $z_{i,t} \neq 0$, ordered by increasing activation magnitude $|z_{i,t}|$. Let $r_t(i)$ be the position of document i in this ordered posting. We assign feature t the rarity weight

$$w_t = 1 + \log \frac{N+1}{|P_t|+1}, \tag{12}$$

so that features active in fewer documents receive larger weight.

Document i then selects up to F active features for candidate generation:

$$T_i = \operatorname*{Top}_{F,\, t \in S_i} \left(|z_{i,t}| w_t^{\alpha} \right), \tag{13}$$

where $\alpha \geq 0$ controls how strongly feature rarity influences selection. Thus, construction prioritizes coordinates that are salient for the document while avoiding over-reliance on very common features.

A.2 CANDIDATE GENERATION AND LINK SELECTION

For each document, SIMSPARSE first constructs an overcomplete pool of candidate neighbors and only then chooses the outgoing graph links. The three candidate sources are designed to provide

complementary connectivity: local posting neighborhoods capture nearby documents along individual features, signatures capture agreement across multiple strong features, and landmarks provide direct access to strong feature representatives. All candidates are subsequently compared under the same sparse similarity score and compete for the same degree budget.

Local candidates along feature postings. Documents with similar activation values on the same feature appear near one another in the corresponding ordered posting. We therefore use short windows around the source document to obtain feature-specific local candidates.

For a radius R, define

$$C_{\mathrm{local}}(i; R) = \bigcup_{t \in T_i} \{j \in P_t : 0 < |r_t(j) - r_t(i)| \le R\}. \tag{14}$$

Thus, each selected feature contributes documents that are close to i in activation rank, while the source document itself is excluded.

Rather than fixing one radius for all documents, we adaptively enlarge the window when the candidate pool is too small. Starting from R_0, we double R until the union contains at least ρM candidates, an expansion adds fewer than δ new documents, or the next radius would exceed $R_{\max}$. This produces enough local proposals for later pruning without requiring every document to inspect a large posting neighborhood.

Multi-feature signature candidates. Local posting windows consider one feature at a time. Two documents can, however, agree on several of their strongest features while being far apart in every individual posting. To expose such joint matches, we assign each document a compact signature formed from its strongest active feature identifiers.

Let $\sigma(i)$ contain the identifiers of the L largest-magnitude active features of document i, sorted by feature identifier to obtain a canonical key. We also define a shorter backoff key $\kappa(i)$ of size $L' < L$ from this canonical signature. Documents sharing either key form candidate buckets:

$$\begin{aligned} C_{\mathrm{sig}}(i) = {} & \mathrm{First}_B\{j : \sigma(j) = \sigma(i)\} \\ & \cup \mathrm{First}_B\{j : \kappa(j) = \kappa(i)\}, \end{aligned} \tag{15}$$

followed by removal of the source document.

The exact signature favors stronger multi-feature agreement, while the shorter backoff key improves coverage when exact signatures are rare. Each bucket contributes at most B candidates, so this mechanism adds at most $2B$ proposals before deduplication.

Nonlocal landmark candidates. Local windows favor nearby activation ranks, so they may require several graph steps to reach documents that strongly represent a feature. To provide direct long-range proposals, we retain a small set of high-activation landmarks for each feature.

Let H_t contain the L_h documents with the largest activation magnitude on feature t. For source document i,

$$C_{\mathrm{land}}(i) = \left(\bigcup_{t \in T_i} H_t \right) \setminus \{i\}. \tag{16}$$

These candidates are independent of the source document's position in the posting, and therefore complement the short-range proposals from local windows.

Scoring and selecting outgoing links. For each document i, we union the local, signature, and landmark candidates and remove duplicates. Candidate generation only determines which documents are considered; the final graph links are selected using the complete sparse codes. Specifically, every candidate j is scored with $s_{\mathrm{build}}(i, j)$ from Eq. (8), which aggregates all shared active features, including those that did not generate the candidate.

If the candidate set exceeds Q, we first retain the top Q candidates by this score and then keep at most M outgoing neighbors. All candidate sources therefore compete under a shared degree budget. The resulting graph is directed: outgoing degree is bounded by M, while no corresponding bound is imposed on in-degree.

Algorithm 1 Construction of the SIMSPARSE graph and entry router

Require: Sparse document codes $Z = \{\mathbf{z}_i\}_{i=1}^N$; construction budgets in Table 5
Ensure: Directed graph $A = \{A_i\}_{i=1}^N$; feature routers $\{R_t\}_{t=1}^H$; fallback entry e_0

```
Build activation-ordered postings P_t, ranks r_t(i), and feature weights w_t
Build exact/backoff signature buckets and landmark sets H_t
for each document i = 1, ..., N do
    Select construction features T_i using Eq. (13)
    C_local ← ∅, R ← R_0
    repeat
        n_old ← |C_local|
        C_local ← C_local ∪ C_local(i; R)
        n_new ← |C_local| − n_old
        R ← 2R
    until |C_local| ≥ ρM or n_new < δ or R > R_max
    C_i ← (C_local ∪ C_sig(i) ∪ C_land(i)) \ {i}
    Score every j ∈ C_i using s_build(i, j)
    Retain the top min(Q, |C_i|) candidates by construction score
    A_i ← the top min(M, |C_i|) remaining candidates
end for
for each feature t = 1, ..., H do
    R_t ← up to L_r documents with largest |z_{i,t}|
end for
e_0 ← arg max_i ||z_i||_2^2
return A, {R_t}, e_0
```

A.3 QUERY ROUTING AND GRAPH TRAVERSAL

Graph construction determines which routes are available, but retrieval also depends on where a query enters the graph. SIMSPARSE uses the query's active features to retrieve a small set of candidate starting documents, then uses the full sparse query score to choose among them.

For each feature t, the router stores up to L_r documents with the largest activation magnitude on that feature. Given query $\mathbf{z}_q$ with active support S_q, the union of its feature-conditioned router lists is

$$L(q) = \bigcup_{t \in S_q} R_t. \tag{17}$$

We score these candidates using the same sparse inner product used during retrieval and retain up to E starting documents:

$$E(q) = \operatorname*{Top}_{E}_{i \in L(q)} \sum_{t \in S_q \cap S_i} z_{q,t} z_{i,t}. \tag{18}$$

The router therefore determines which regions of the graph are exposed to the query, while the complete sparse score determines which exposed documents are used to initialize traversal. If no router candidate is available, we fall back to the stored maximum-norm document e_0.

Starting from the selected entries, SIMSPARSE performs bounded best-first graph search. At each step, it expands the highest-scoring unexpanded document in the candidate queue, scores any previously unseen outgoing neighbors, and inserts promising neighbors back into the queue. The queue retains at most W candidates, discarding the lowest-scoring candidates when this capacity is exceeded. This bound limits the number of alternative search paths retained for future expansion.

Search terminates after at most C expansions or when no unexpanded candidate remains. We separately maintain a size-k result heap containing the best documents scored so far. Thus, W controls the breadth of the active search frontier, whereas C controls the total number of graph expansions.

Algorithm 2 maintains two priority structures: a max-priority frontier for choosing the next document to expand and a size-k min-priority heap for the best results seen so far. We distinguish *scored*, *queued*, and *expanded* documents. A document is scored when it is first reached through routing or

a graph edge; it is queued if it survives the candidate-queue capacity constraint; and it is expanded when its outgoing links are inspected. A visited set prevents the same document from being scored repeatedly.

Algorithm 2 Query routing and bounded best-first traversal in SIMSPARSE

Require: Query code $\mathbf{z}_q$; deployed index $(Z, A, \{R_t\}, e_0)$; result count k; entry budget E; queue capacity W; expansion budget C
Ensure: Approximate top-k document identifiers

$L \leftarrow \bigcup_{t \in S_q} R_t$; deduplicate candidates
Compute and cache $s(q, i)$ for each $i \in L$
$S \leftarrow$ the top E candidates in L by score
if $S = \emptyset$ **then**
    $S \leftarrow \{e_0\}$
end if
Initialize capacity-W max-priority candidate queue F_q, capacity-k result heap H_k, and visited set V
for each $i \in S$ **do**
    Add i to V; insert $(s(q, i), i)$ into F_q and H_k
end for
$c \leftarrow 0$
while $F_q \neq \emptyset$ **and** $c < C$ **do**
    $(u, i) \leftarrow \mathrm{PopMax}(F_q)$
    if early stopping is enabled **and** $|H_k| = k$ **and** $u < \mathrm{MinScore}(H_k)$ **then**
        break
    end if
    for each $j \in A_i$ with $j \notin V$ **do**
        Add j to V; compute or reuse $v \leftarrow s(q, j)$
        Insert (v, j) into F_q; if $|F_q| > W$, discard its lowest-scoring item
        Insert (v, j) into H_k, evicting its minimum if necessary
    end for
    $c \leftarrow c + 1$
end while
return documents in H_k in decreasing score order

A.4 Deployment Storage and Computational Cost

At deployment, SIMSPARSE retains only three corpus-dependent components: the sparse document codes, the bounded-degree graph, and the feature-conditioned entry router. All postings, signature tables, and landmark caches used during construction are discarded. We first give the exact storage accounting for these retained structures, then summarize the offline construction cost and the per-query search cost.

Persistent index storage. Let

$$K_{\mathrm{tot}} = \sum_i |S_i|, \qquad m = \sum_i |A_i|, \qquad r = \sum_t |R_t|$$

denote the actual numbers of stored sparse activations, graph links, and router entries, respectively. If feature identifiers, activation values, document identifiers, and row offsets require b_f, b_v, b_n, and b_o bytes, then the persistent array storage is

$$\begin{aligned} B_{\mathrm{arrays}} = K_{\mathrm{tot}}(b_f + b_v) + \big[mb_n + (N+1)b_o\big] \\ + \big[rb_n + (H+1)b_o\big] + B_z, \end{aligned} \tag{19}$$

where B_z accounts for sparse-code row offsets when they are stored explicitly.

The degree and router budgets imply $m \leq NM$ and $r \leq HL_r$. For fixed-support sparse codes, document row locations may be implicit, giving $B_z = 0$; variable-support CSR storage instead requires row offsets. Importantly, the construction postings, signature buckets, and landmark tables do not contribute to deployed index storage.

Table 5: Default SIMSPARSE construction and search parameters. The entry count E and candidate queue length W are query-time search parameters and do not affect graph construction.

Parameter	Value	Parameter	Value
Degree bound M	16	Selected features F	8
Feature exponent α	0.75	Edge-score exponent β	0.5
Initial / maximum radius	8 / 96	Oversampling ρ	2
Minimum new candidates δ	2	Post-score cap Q	128
Signature sizes L / L'	3 / 2	Bucket cap B	256
Landmarks per feature L_h	4	Cached representatives	64
Router entries per feature L_r	128	Recommended entry count E	4
Candidate queue W	128		

Offline construction cost. Construction cost is dominated by candidate generation and sparse scoring. Let

$$R_* = R_0\, 2^{\lfloor \log_2(R_{\max}/R_0) \rfloor}$$

be the largest radius reachable by the doubling schedule. Before deduplication and score-based truncation, a document considers at most

$$|C(i)| \le 2FR_* + 2B + FL_h \equiv P_{\max}, \tag{20}$$

where the three terms correspond to local windows, exact/backoff signature buckets, and feature landmarks.

Thus, with bounded document support size K_d, scoring all construction candidates requires

$$O(NP_{\max}K_d)$$

sparse-intersection work, up to constant factors from the sparse scoring implementation. Building the activation-ordered postings additionally requires

$$O\left(\sum_t n_t \log n_t\right), \qquad n_t = |P_t|.$$

All auxiliary posting, signature, and landmark structures are temporary and are discarded after construction.

Per-query search cost. Let $K_q = |S_q|$ be the query support size. Router lookup exposes at most $K_q L_r$ candidate entries, while at most C graph expansions inspect CM outgoing links. Each newly encountered document is scored at most once, and the candidate queue retains at most W scored documents for possible future expansion.

For sorted sparse supports, evaluating one query–document inner product takes $O(K_q + K_d)$ time. The total sparse-scoring work is therefore bounded by

$$T_{\text{score}} = O((K_q L_r + CM + 1)(K_q + K_d))\,. \tag{21}$$

The queue capacity W bounds the number of active expansion candidates rather than the number of documents that may be scored: a newly reached neighbor must be scored before the search can decide whether it should remain in the queue.

Maintaining the bounded candidate queue requires $O(W)$ temporary state, while the result heap requires $O(k)$. Queue and heap updates contribute logarithmic factors of $O(\log W)$ and $O(\log k)$ under heap-based implementations. These structures are temporary per-query state and are not included in persistent index storage.

B WHY FEATURE-INDUCED ROUTES WORK

This section isolates the structural properties that make feature-induced navigation possible. Sparse support first provides an exact address space for all documents that can receive a nonzero query score. Within this address space, local posting neighborhoods, feature signatures, and landmarks expose complementary routes at different posting distances. Finally, construction-time reweighting changes which of these routes are retained without changing the underlying sparse support or introducing an additional deployed representation. These statements concern the indexing mechanism; retrieval quality is evaluated empirically in Section 5 and Appendix D.1.

B.1 SPARSE SUPPORT PROVIDES AN EXACT SCORE ADDRESS SPACE

Proposition 1 (support determines score addressability). Let S_q and S_i denote the nonzero supports of arbitrary real-valued sparse codes $\mathbf{z}_q$ and $\mathbf{z}_i$, and let $P_t = \{i : z_{i,t} \neq 0\}$ be the posting associated with feature t. Any document with a nonzero query score must occur in at least one posting activated by the query:

$$s(q,i) \neq 0 \quad \Longrightarrow \quad i \in \bigcup_{t \in S_q} P_t. \tag{22}$$

For nonnegative codes, the converse also holds: membership in the union implies a strictly positive score. Thus the sparse feature identifiers provide an exact address space for all documents that can contribute to the sparse scoring rule, while the stored activation values suffice to evaluate their scores.

Proof. Expanding the sparse inner product gives $\mathbf{z}_q^\top \mathbf{z}_i = \sum_{t=1}^{H} z_{q,t} z_{i,t}$. A nonzero contribution can occur only at a coordinate in $S_q \cap S_i$. Hence, if the total score is nonzero, the intersection must be nonempty, and i belongs to P_t for at least one $t \in S_q$. For nonnegative codes, any shared active coordinate contributes a strictly positive term and all remaining terms are nonnegative, so a nonempty intersection also implies a positive score. □

Addressability is not exhaustive retrieval. Proposition 1 identifies where a nonzero-scoring document can occur, but it does not require all such documents to be visited at query time. SimSPARSE retains only short feature-conditioned routing lists as entry candidates and uses the graph to reach additional documents under a bounded search budget. Thus, exact sparse scoring and exhaustive sparse search are separate questions: the former follows directly from the stored sparse codes, whereas the latter is replaced by approximate graph traversal.

B.2 COMPLEMENTARY CANDIDATE SOURCES CREATE LOCAL AND NONLOCAL ROUTES

Multi-feature agreement can be nonlocal in every individual posting. The conjunction case in Figure 2 already occurs in four dimensions. Consider the normalized sparse vectors over coordinates (A, B, U, V),

$$\begin{aligned} \mathbf{z}_i = (0.6, 0.8, 0, 0), \qquad \mathbf{z}_j = (0.8, 0.6, 0, 0), \\ \mathbf{z}_i^\top \mathbf{z}_j = 0.6 \times 0.8 + 0.8 \times 0.6 = 0.96. \end{aligned} \tag{23}$$

Both documents have $\{A, B\}$ as their two strongest feature identifiers. Now fix any integer $m \geq 1$. For m distinct values $a \in (0.6, 0.8)$, add documents $(a, 0, \sqrt{1-a^2}, 0)$ and $(0, a, 0, \sqrt{1-a^2})$. These unit-norm documents insert m positions between i and j in the A and B postings, respectively, while having strongest-feature sets $\{A, U\}$ and $\{B, V\}$. Consequently, any local window with radius $r < m+1$ fails to propose i and j through either posting, even though a signature on $\{A, B\}$ places them in the same bucket.

The example shows that agreement across strong features is not reducible to proximity along any one feature ordering. Signatures expose such joint matches without requiring wider single-feature windows. The two-feature construction is only illustrative; the default signature size is given in Table 5.

Proposition 2 (candidate routes before degree pruning). Fix an activation-ordered posting P_t and suppose that every document under consideration selects feature t during construction and uses a local window of radius at least $r \geq 1$. Before degree pruning, two posting positions $a < b$ are connected by a sequence of at most

$$\left\lceil \frac{b-a}{r} \right\rceil \tag{24}$$

local candidate links. By contrast, a signature match can be proposed in one step independently of the documents' distance in any individual posting, and a landmark can be proposed in one step independently of the source document's position in P_t.

Proof. From position a, a radius-r window contains position $\min(a + r, b)$. Repeating this argument decreases the remaining posting distance by at least r until b is reached, requiring at most $\lceil (b-a)/r \rceil$ candidate steps. Signature lookup depends on equality of feature-identifier tuples rather than posting distance, while landmark lookup selects directly from H_t. Both mechanisms can therefore propose a one-step candidate that lies outside the current local window. □

Shared signatures certify multi-feature overlap. An exact signature match shares every identifier in a nonempty size-L strongest-feature set σ. For nonnegative sparse codes,

$$\mathbf{z}_i^\top \mathbf{z}_j \;\geq\; \sum_{t \in \sigma} z_{i,t} z_{j,t} \;>\; 0. \tag{25}$$

The first inequality follows because all coordinate-wise products outside σ are nonnegative. Thus an exact signature does more than bypass posting distance: it identifies documents with simultaneous overlap on several strong coordinates. This is qualitatively different from locality in a single activation-ordered posting.

Candidate routes are hypotheses, not guaranteed edges. The preceding statements concern the routes exposed by candidate generation. The deployed graph retains only the highest-scoring candidates under the shared degree budget, so a particular candidate path persists only when its constituent links survive pruning. The three sources therefore provide complementary opportunities rather than separate graph layers: local windows expose rank-contiguous neighbors, signatures expose multi-feature matches outside those windows, and landmarks expose strong feature representatives independent of source rank. All retained links are stored in the same bounded-degree adjacency structure.

A concrete routing and traversal example. Once a candidate link survives degree pruning, it can be used independently of the mechanism that originally proposed it. Consider a sparse query with nonzero coordinates

$$z_q = \{A : 0.8,\ C : 0.6\},$$

and three documents

$$z_1 = \{A : 0.9,\ B : 0.1\}, \qquad z_2 = \{A : 0.4,\ C : 0.8\}, \qquad z_3 = \{A : 0.7,\ C : 0.7\},$$

where all unlisted coordinates are zero. Suppose the feature-conditioned router stores $R_A = [d_1]$ and $R_C = [d_2]$. The query therefore exposes d_1 and d_2 as candidate entries. Their complete sparse inner products with the query are

$$s(q, 1) = 0.8(0.9) = 0.72, \qquad s(q, 2) = 0.8(0.4) + 0.6(0.8) = 0.80.$$

With a single-entry budget, d_2 is selected as the starting document.

Now suppose the deployed graph contains the stored link $d_2 \to d_3$. Expanding d_2 exposes d_3, whose query score is

$$s(q, 3) = 0.8(0.7) + 0.6(0.7) = 0.98.$$

Thus traversal reaches a document that was absent from the query's router lists but scores higher than the selected entry. The router only determines which documents are initially exposed; subsequent access is determined by the retained graph links, while both entry selection and traversal are scored using the same sparse inner product. The values above are schematic and serve only to illustrate this interaction between feature-conditioned routing and graph navigation.

B.3 Construction Reweighting Preserves the Sparse Representation

Proposition 3 (reweighting preserves sparse support). For positive feature weights w_t and exponent $\beta \geq 0$, define the conceptually reweighted code

$$\widetilde{z}_{i,t} = w_t^{\beta/2} z_{i,t}. \tag{26}$$

Then

$$s_{\mathrm{build}}(i,j) = \widetilde{\mathbf{z}}_i^\top \widetilde{\mathbf{z}}_j, \qquad \mathrm{supp}(\widetilde{\mathbf{z}}_i) = \mathrm{supp}(\mathbf{z}_i) = S_i. \tag{27}$$

Thus IDF-weighted link selection changes the geometry used during construction without changing which sparse coordinates are active. If the final adjacency retains at most M document identifiers per row and each feature router retains at most L_r identifiers, the deployed graph and router contain at most NM and HL_r identifiers, respectively.

Proof. For each coordinate,

$$\widetilde{z}_{i,t}\widetilde{z}_{j,t} = w_t^{\beta/2} z_{i,t}\, w_t^{\beta/2} z_{j,t} = w_t^{\beta} z_{i,t} z_{j,t}, \tag{28}$$

and summing over t gives $s_{\mathrm{build}}(i,j)$. Because every w_t is strictly positive, reweighting neither creates nor removes nonzero coordinates. The storage bounds follow by summing the per-document adjacency cap over N documents and the per-feature router cap over H features. □

Construction changes edge preference, not deployed representation. The reweighted view is only a construction-time interpretation. SimSPARSE does not store $\widetilde{\mathbf{z}}_i$ in addition to $\mathbf{z}_i$: online search uses the original sparse codes together with the bounded adjacency and router arrays produced during construction. Consequently, construction can favor rare or informative shared coordinates without introducing a second corpus-level scoring representation. With at most C expansions and degree bound M, search exposes at most CM outgoing links, yielding the scoring-work bound in Eq. (21) independently of the lengths of the construction postings.

The post-score cap does not alter top-M link selection. For $Q \geq M$, retaining the top Q candidates by s_{build} and then selecting the top M by the same score yields the same top-M set as selecting the top M directly from the full candidate pool, provided ties are resolved by the same fixed total order. Hence the intermediate cap affects only the materialized post-score candidate set; it does not change the selected outgoing links or reduce the number of candidate scores that must first be evaluated.

C Experiment Details

C.1 Datasets

We evaluate retrieval quality on six million-scale datasets from the BEIR benchmark (Thakur et al., 2021), and use a 60-million-passage Wikipedia corpus derived from RedPajama (RPJ-60M) for the large-scale storage–latency evaluation.

BEIR retrieval benchmarks. We select six datasets whose corpora contain millions of documents and span question answering, fact verification, entity retrieval, and web passage retrieval. We use the standard BEIR corpus, query, and relevance-judgment files and evaluate on the official BEIR test split.

- **HotpotQA** (Yang et al., 2018): a multi-hop question answering benchmark constructed from Wikipedia. Its BEIR retrieval version contains approximately 5.23M passages and 7,405 test queries. A query may be associated with multiple relevant passages, reflecting the supporting evidence required by the original multi-hop questions.
- **Natural Questions (NQ)** (Kwiatkowski et al., 2019): an open-domain question answering benchmark containing naturally occurring search questions paired with evidence from Wikipedia. The BEIR version contains approximately 2.68M passages and 3,452 test queries.
- **Climate-FEVER** (Diggelmann et al., 2020): a fact-verification benchmark consisting of climate-related claims and evidence drawn from Wikipedia. The BEIR version contains approximately 5.42M passages and 1,535 test queries.
- **FEVER** (Thorne et al., 2018): a large-scale fact-verification benchmark in which textual claims are associated with supporting or refuting evidence from Wikipedia. Its BEIR retrieval corpus contains approximately 5.42M passages and 6,666 test queries.
- **DBpedia-Entity** (Bizer et al., 2009): an entity-retrieval benchmark built over DBpedia. Queries cover several entity-search intents, including named entities, natural-language questions, and list queries. The BEIR version contains approximately 4.63M entities and 400 test queries.
- **MS MARCO** (Bajaj et al., 2016): a large-scale passage retrieval benchmark derived from real search queries. The BEIR passage corpus contains approximately 8.84M passages and 6,980 evaluation queries.

The six corpora therefore contain between approximately 2.68M and 8.84M documents, allowing us to evaluate graph retrieval under million-scale candidate spaces while retaining standard human relevance judgments.

RPJ-60M scalability benchmark. For the large-scale experiment, we construct a shared datastore from the Wikipedia portion of RedPajama. The raw Wikipedia text is segmented into chunks of 256 tokens, producing approximately 60 million passages from roughly 76 GB of source text. Each passage is encoded once by the corresponding pretrained backbone before sparse representation learning and index construction.

We evaluate this datastore using queries from four domains: Natural Questions (Kwiatkowski et al., 2019), TriviaQA (Joshi et al., 2017), GPQA (Rein et al., 2023), and HotpotQA (Yang et al., 2018). These query sets cover open-domain factual questions, trivia questions, graduate-level scientific questions, and multi-hop questions, respectively. All four query domains are evaluated against the same RPJ-60M datastore and they do not define separate document collections.

C.2 Evaluations

BEIR retrieval quality. For the six BEIR benchmarks, we report normalized Discounted Cumulative Gain at rank 10 (nDCG@10), using the official relevance judgments provided by BEIR. For a

query q, let $\mathrm{rel}_{q,i}$ denote the relevance grade of the document appearing at rank i. We compute

$$\mathrm{DCG@10}(q) = \sum_{i=1}^{10} \frac{2^{\mathrm{rel}_{q,i}} - 1}{\log_2(i+1)}, \tag{29}$$

and normalize it by the DCG of the ideal ranking:

$$\mathrm{nDCG@10}(q) = \frac{\mathrm{DCG@10}(q)}{\mathrm{IDCG@10}(q)}. \tag{30}$$

We report the mean over evaluation queries. All nDCG@10 values in the main tables are reported as percentages.

Hybrid reference neighbors for RPJ-60M. RPJ-60M does not provide passage-level relevance judgments for the four query domains. We therefore construct a proxy reference set directly from the dense and sparse representation spaces rather than treating either representation as an independent source of ground truth.

For each query q and document d_i, we first compute the dense-space similarity

$$s_{\mathrm{dense}}(q, i) \tag{31}$$

using the corresponding pretrained backbone, and the sparse-space similarity

$$s_{\mathrm{sparse}}(q, i) = \mathbf{z}_q^\top \mathbf{z}_i \tag{32}$$

using the learned sparse codes.

Because the two similarity functions may have different numerical ranges, we normalize them independently for each query using min–max normalization over the complete RPJ-60M corpus:

$$\widetilde{s}_x(q, i) = \frac{s_x(q, i) - \min_j s_x(q, j)}{\max_j s_x(q, j) - \min_j s_x(q, j)}, \qquad x \in \{\mathrm{dense}, \mathrm{sparse}\}. \tag{33}$$

We then combine the two normalized similarities with equal weight:

$$s_{\mathrm{hybrid}}(q, i) = \widetilde{s}_{\mathrm{dense}}(q, i) + \widetilde{s}_{\mathrm{sparse}}(q, i). \tag{34}$$

The reference top-3 set is defined as

$$R_3^\star(q) = \mathrm{Top3}_i \; s_{\mathrm{hybrid}}(q, i). \tag{35}$$

This construction provides a proxy for semantic relevance when manually annotated passage-level qrels are unavailable. The dense and sparse representations capture related but non-identical similarity structures, while independent normalization prevents either space from dominating the combined score merely because of its numerical scale. Their equal-weight combination therefore favors documents that are strongly supported across both representation spaces.

Recall on RPJ-60M. Given an approximate top-3 result $R_3(q)$ and the hybrid reference set $R_3^\star(q)$, we compute

$$\mathrm{Recall@3}(q) = \frac{|R_3(q) \cap R_3^\star(q)|}{3}. \tag{36}$$

We average Recall@3 over the queries in each evaluation domain.

For each indexing method, we vary its search-time budget and identify the smallest setting that reaches at least 90% average Recall@3. Storage and query latency are then reported at this matched-recall operating point. This protocol controls search fidelity across methods while allowing their internal retrieval procedures to differ, and avoids conflating approximate index quality with the absence of manually annotated relevance judgments on RPJ-60M.

Latency measurement. Query latency excludes offline representation training and index construction. It measures only the online retrieval procedure starting from an already encoded query. For RPJ-60M, latency is measured independently on Natural Questions, GPQA, TriviaQA, and HotpotQA and reported in seconds per query.

C.3 Implementation Details

Backbone representations. We conduct experiments with Contriever (Izacard et al., 2021) and BGE-base-en-v1.5 (Xiao et al., 2024). The dense output of each pretrained backbone is used as the input representation from which the corresponding sparse codes are learned. The sparse encoder maps the dense representation into a higher-dimensional nonnegative space and retains only a prescribed number of active coordinates through a Top-K operator. Once trained, these sparse codes are fixed and shared by graph construction, query routing, graph traversal, and final sparse scoring.

Sparse autoencoding objective. Our sparse representation training follows the Top-K sparse-autoencoder formulation used in CSR (Wen et al., 2025). Given a dense backbone embedding $\mathbf{h} \in \mathbb{R}^d$, the sparse encoder produces

$$\mathbf{z}_k = \mathrm{ReLU}(\mathrm{TopK}_k\left(W_{\mathrm{enc}}(\mathbf{h} - \mathbf{b}_{\mathrm{pre}}) + \mathbf{b}_{\mathrm{enc}}\right), \tag{37}$$

and reconstructs the dense representation as

$$\widehat{\mathbf{h}}_k = W_{\mathrm{dec}}\mathbf{z}_k + \mathbf{b}_{\mathrm{pre}}. \tag{38}$$

The reconstruction objective is evaluated at multiple sparsity levels:

$$\mathcal{L}_{\mathrm{recon}} = \mathcal{L}(k) + \frac{1}{8}\mathcal{L}(4k) + \beta\mathcal{L}_{\mathrm{aux}}, \tag{39}$$

where

$$\mathcal{L}(k) = \left\|\mathbf{h} - \widehat{\mathbf{h}}_k\right\|_2^2. \tag{40}$$

The auxiliary reconstruction term uses otherwise inactive latent units to reconstruct residual error, providing training signal to coordinates that would otherwise be selected infrequently by the Top-K operator.

Supervised sparse contrastive learning. Reconstruction encourages the sparse representation to preserve information from the original backbone embedding, but does not directly optimize the relative similarity between retrieval queries and relevant documents. We therefore additionally use supervised contrastive learning following the retrieval-supervised formulation used in CSRv2 (Guo et al., 2026).

For each training query q, a document d^+ marked relevant by the training relevance judgments is treated as a positive example, while other documents available in the minibatch are used as negatives. The supervised contrastive loss is

$$\mathcal{L}_{\mathrm{sCL}} = -\frac{1}{|\mathcal{B}|}\sum_{q\in\mathcal{B}} \log \frac{\exp(s(\mathbf{z}_q, \mathbf{z}_{d^+})/\tau)}{\sum_{d\in\mathcal{N}(q)\cup\{d^+\}} \exp(s(\mathbf{z}_q, \mathbf{z}_d)/\tau)}, \tag{41}$$

where $\mathcal{N}(q)$ denotes the negative documents available for query q, τ is the temperature, and $s(\cdot,\cdot)$ is the sparse-space similarity used for retrieval.

The complete sparse-representation objective is

$$\mathcal{L} = \mathcal{L}_{\mathrm{recon}} + \gamma_{\mathrm{s}}\mathcal{L}_{\mathrm{sCL}}. \tag{42}$$

D Additional Empirical Analysis

D.1 Graph Topology and Navigation Diagnostics

The main experiments show that SimSPARSE benefits from organizing sparse representations into a feature-induced graph rather than relying on the sparse codes alone. Here we isolate this effect more directly by holding the sparse representations fixed and varying only the graph topology, and then examine structural diagnostics of a conventional sparse nearest-neighbor graph.

Topology comparison with fixed sparse representations. Table 6 compares different graph constructions using exactly the same sparse document and query representations. We evaluate KNN, HNSW, DiskANN, and SimSPARSE on six million-level-document benchmarks under both Contriever-based and BGE-based sparse codes. Because the representations are fixed, differences in retrieval quality arise from how the same sparse similarity signal is organized and accessed by the graph rather than from differences in representation learning.

SimSPARSE obtains the highest nDCG@10 on all six benchmarks with Contriever and on five of the six benchmarks with BGE. The gap is particularly large on several datasets: for example, with Contriever, SimSPARSE reaches 32.61 on NQ compared with 17.46 for DiskANN and 14.63 for HNSW, while on HotpotQA it reaches 21.19 compared with 11.48 and 11.03, respectively. The same pattern persists with BGE, where SimSPARSE substantially improves over the alternative topologies on NQ, Climate-FEVER, FEVER, and DBpedia.

These results indicate that sparse representations do not automatically become effective graph-search representations simply by connecting nearest neighbors under the sparse similarity. The arrangement of edges determines which parts of the sparse similarity structure can be reached within a bounded search budget. SimSPARSE explicitly uses active-feature structure to create local and nonlocal routes, making a larger fraction of the information already present in the fixed sparse codes accessible during search.

Backbone	Method	HotpotQA	NQ	Climate-FEVER	FEVER	DBpedia	MSMARCO
Contriever	KNN	6.81	11.50	34.67	27.33	25.79	12.04
	HNSW	11.03	14.63	25.65	30.50	30.17	16.23
	DiskANN	11.48	17.46	23.80	25.68	30.25	17.59
	SimSPARSE	**21.19**	**32.61**	**42.38**	**43.12**	**37.20**	**23.62**
BGE	KNN	8.49	26.01	8.43	15.12	14.63	34.64
	HNSW	17.63	38.31	22.38	27.06	26.82	45.07
	DiskANN	18.73	41.81	10.92	32.66	36.13	**48.29**
	SimSPARSE	**26.02**	**52.71**	**35.68**	**76.10**	**41.51**	47.55

Table 6: **Graph topology comparison with fixed sparse representations.** All methods use the same sparse document and query representations; only the graph topology changes. Retrieval quality is nDCG@10 ($\times 100$) on six million-level benchmarks. Bold indicates the highest value for each backbone and benchmark.

Navigation diagnostics of sparse nearest-neighbor graphs. To understand why a conventional sparse nearest-neighbor topology can remain difficult to search, Table 7 reports navigation statistics on HotpotQA and Quora. In addition to retrieval quality, we measure the fraction of vertices reachable from the designated graph entry and the fraction of vertices with zero in-degree.

The designated entry reaches 73.12% of vertices on HotpotQA and 76.27% on Quora, leaving a substantial portion of each graph inaccessible from that starting point. At the same time, 22.28% and 18.41% of vertices, respectively, have zero in-degree. These structural limitations are consistent with the gap between sparse-kNN retrieval and the stronger topology-based results: even when relevant documents have high sparse similarity, they cannot contribute to search if the graph does not expose a traversable route to them within the search process.

This observation motivates the two access mechanisms used by SimSPARSE. Feature-induced candidate generation creates routes that are not restricted to nearest-neighbor connectivity alone, while

query-conditioned entry selection reduces dependence on a single corpus-wide starting point. Together, these mechanisms address complementary sources of search failure: the graph topology determines which routes exist, whereas query routing determines which regions are exposed at the beginning of traversal.

Dataset	Dense-topology nDCG@10	Sparse-kNN nDCG@10	Sparse-kNN R@100	Reachable ↑	Zero in-degree ↓
HotpotQA	0.2432	0.1617	0.2322	0.7312	0.2228
Quora	0.7845	0.4500	0.5569	0.7627	0.1841

Table 7: **Navigation diagnostics for sparse nearest-neighbor graphs.** Reachability is the fraction of vertices reachable from the designated entry; zero in-degree is the fraction of vertices with no incoming graph links. Sparse-kNN R@100 measures retrieval recall under the corresponding sparse-kNN topology.

D.2 Storage Budgets, Numerical Precision, and Sparse Capacity

The storage–precision analysis in Section 5.3 compares different numerical formats under a fixed per-document storage budget. Here we make explicit how each byte budget translates into the number of sparse activations that can be retained, and provide a complementary visualization of this capacity.

From a byte budget to TopK. A sparse document code stores, for every active coordinate, both its feature identifier and its activation value. In our storage accounting, a feature identifier occupies 16 bits. If the activation value is represented with p bits, the nominal storage cost of one active coordinate is therefore $16+p$ bits. Under a budget of B bytes per document, the largest support size that fits within the representation budget is

$$K(B,p) = \left\lfloor \frac{8B}{16+p} \right\rfloor, \qquad p \in \{1,4,8,32\}, \tag{43}$$

corresponding to INT1, INT4, FP8, and FP32, respectively. Configurations for which Equation 43 gives $K < 1$ are infeasible and are omitted.

This conversion makes the storage trade-off especially transparent. For example, at 64 Bytes/doc, the same representation budget permits $K = 30$ with INT1, $K = 25$ with INT4, $K = 21$ with FP8, but only $K = 10$ with FP32. At 128 Bytes/doc, these capacities increase to 60, 51, 42, and 21, respectively. Lower numerical precision therefore allocates more of a fixed storage budget to retaining additional active coordinates, whereas higher precision spends a larger fraction of the budget preserving each activation value more accurately.

Visualizing the precision–capacity trade-off. Figure 5 gives a complementary view of the storage-aware precision experiment in Figure 4 (Left). The two figures use the same underlying retrieval measurements; the difference is in how the experimental configuration is displayed. For each combination of per-document budget and numerical precision, Figure 5 places the corresponding TopK from Equation 43 directly inside the marker, while the marker shading represents the resulting average retrieval performance. A dash denotes a budget–precision combination that cannot store even one active coordinate. The outlined marker at each budget indicates the precision achieving the highest performance under that budget.

The visualization clarifies why the preferred precision changes with storage budget. Under tight budgets, the dominant constraint is representational capacity: using fewer bits per activation allows substantially more nonzero coordinates to be retained, so aggressive quantization can outperform more precise values despite introducing larger quantization error. As the budget increases, support size becomes less restrictive and the marginal benefit of additional active coordinates diminishes. Preserving activation magnitudes more accurately then becomes comparatively more important, causing the preferred format to shift from INT1 toward FP8 and eventually FP32. Thus, the storage-aware transition observed in Figure 4 (Left) can be interpreted as a change in the relative value of *sparse capacity* and *activation precision* as the available bytes per document increase.

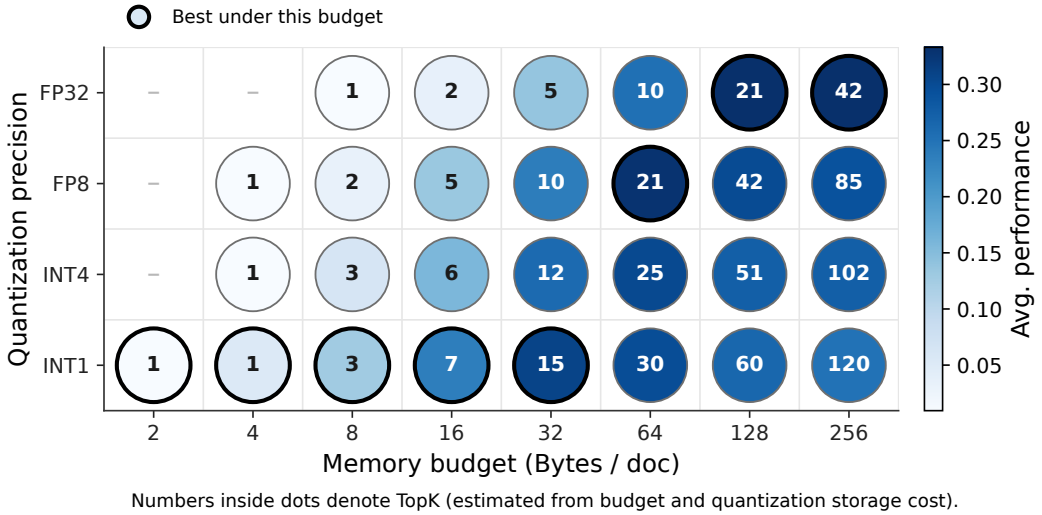


Figure 5: **Sparse capacity under different storage budgets and numerical precisions.** Numbers inside markers show the maximum TopK supported by each budget–precision combination according to Equation 43; marker shading shows the corresponding average retrieval performance used in Figure 4 (Left). A dash denotes an infeasible configuration, and a black outline identifies the highest-performing precision at each storage budget.

E Extended Related Work

The comparisons in Section 2 connect three design choices: what an index retains, how a representation exposes candidates, and which paths search can follow. These choices interact: a shorter code may still require a larger representation for reranking, and a sparse representation may support either posting traversal or graph search. We elaborate on these interactions to locate SimSPARSE within existing work on compact and navigable retrieval.

E.1 Compact Representations and the Information Retained for Scoring

Vector compression is useful to retrieval insofar as it preserves the information needed to distinguish promising candidates. Within the product-quantization family, this begins with representing subvectors by codebook identifiers (Jegou et al., 2010) and extends to learning the decomposition and codebooks to reduce distortion, as in OPQ (Ge et al., 2013). The relevant notion of distortion can itself depend on the retrieval task. Anisotropic vector quantization (Guo et al., 2020), for example, weights reconstruction error by its effect on large inner products, directing capacity toward maximum inner product search. Randomized binary quantization offers another approach: RaBitQ (Gao & Long, 2024) supports an unbiased squared-distance estimator with a probabilistic error bound. Together, these methods show that code length is only one part of compression; the objective or estimator determines which aspects of the original geometry the code preserves.

Reducing dimensionality similarly connects representation size to the work performed during search. MRL (Kusupati et al., 2022) learns nested embedding prefixes that remain useful at several lengths, allowing the representation budget to vary without prescribing a particular storage or reranking policy. LeanVec (Tepper et al., 2023) combines linear projection with vector quantization and considers both matching and differing query–database distributions. GleanVec (Tepper et al., 2024) develops this direction through piecewise-linear mappings and the closed-form LeanVec-Sphering method. Their multi-stage search retrieves candidates using reduced representations and reranks them using higher-dimensional vectors. The smaller representation therefore reduces candidate-search costs, while the retained reranking state also contributes to the deployed footprint.

At the system level, the same question becomes when and where accurate embeddings are available. DiskANN (Jayaram Subramanya et al., 2019) places compressed vectors in memory and full vectors

on disk, coupling graph navigation to storage access. LEANN (Wang et al., 2025) moves part of this cost to computation: high-degree-preserving graph pruning and compact-proxy filtering support selective document re-encoding at query time. This reduces retained embedding state while making recomputation part of retrieval. SIMSPARSE instead uses sparse codes as the final scoring state throughout online search. Construction auxiliaries are discarded, and deployment retains the codes, bounded-degree graph, and short entry lists. This choice makes both the retained corpus state and the online scoring path explicit. Its quality is assessed empirically; sparse-code ranking does not imply exact recovery of the original dense ranking.

E.2 From Learned Sparse Features to Candidate Access

Learned lexical retrieval builds on the addressability of vocabulary terms while learning which matches should matter. Document expansion increases the set of possible matches, and learned impacts control their contribution, as in DeepImpact (Mallia et al., 2021). Contextual representations can also be reduced to scalar term weights compatible with inverted indexes, as in uniCOIL (Lin & Ma, 2021), which can additionally use expansion. SPLADE (Formal et al., 2021b) learns vocabulary weighting and expansion jointly under sparsity regularization; SPLADE v2 (Formal et al., 2021a) strengthens this approach through max pooling and distillation and explores document-only encoding to remove neural query-encoding cost. Across these designs, vocabulary coordinates provide a common interface between learned relevance and inverted access.

Latent sparse representations retain this feature-based interface without requiring coordinates to denote words. CSR (Wen et al., 2025) learns codes through sparse autoencoding and contrastive learning, and CSRv2 (Guo et al., 2026) pushes toward ultra-sparsity through k-annealing, supervised contrastive learning, and optional backbone finetuning. For an index, the useful structure is the set of active feature identifiers and their activation values: shared coordinates identify potential matches even when those coordinates lack lexical meanings. SIMSPARSE operates on these fixed codes and exploits this structure during construction and query entry selection. The representation-learning objective and the access algorithm thus remain distinct design choices.

Turning feature overlap into efficient retrieval requires controlling the candidates exposed by each active coordinate. Inverted search addresses this problem by organizing and selectively visiting postings. Seismic (Bruch et al., 2024a) combines posting pruning with geometrically coherent blocks, uses summaries to select blocks, and evaluates candidates through a forward index containing their full sparse vectors. SeismicWave (Bruch et al., 2024b) refines this path by ordering blocks in the first visited posting according to query–summary scores and then performing one-hop nearest-neighbor expansion of the result set. This already connects sparse inverted retrieval to graph-based refinement. The distinction in SIMSPARSE is where the main search takes place: postings organize candidate generation during construction, while online retrieval proceeds through repeated document-graph expansion after a bounded feature-list lookup. Full construction postings are no longer retained at deployment. The shared use of sparse scores therefore coexists with a different lifetime and role for the inverted structure.

E.3 Constructing Search Paths and Controlling Traversal

A useful proximity graph must support travel beyond the nearest local neighborhood within a limited edge budget. HNSW (Malkov & Yashunin, 2018) provides hierarchical routes, while NSG (Fu et al., 2017) combines pruning with connectivity repair. The importance of edge arrangement is further developed by SSG (Fu et al., 2021), which studies angularly distributed outgoing edges for indexed and unindexed queries and introduces NSSG as a practical large-scale construction. Search behavior can also guide graph refinement directly: GraSP (Zhang et al., 2022) uses training queries, subgraph sampling, and iterative refinement to identify edges whose removal improves expected search efficiency. These approaches connect graph quality to geometric coverage or observed traversal behavior, beyond the similarity of individual linked pairs.

For sparse codes, feature membership supplies additional structure from which candidate paths can be built. SIMSPARSE uses activation-ordered posting windows to propose local links, shared-feature signatures to expose joint matches outside those windows, and high-activation landmarks to propose nonlocal links. Candidates compete under one degree budget after similarity-based scoring. These mechanisms specify how sparse feature structure generates edges; they do not establish the

geometric guarantees studied for SSG or guarantee that all proposed routes survive pruning. kANNolo's support for dense and sparse HNSW (Delfino et al., 2025) also makes clear that sparse-vector compatibility is already available in graph search. The focus here is how active coordinates shape graph construction and access.

Once a graph is built, retrieval cost depends on both its starting points and the neighbors evaluated during traversal. LADR (Kulkarni et al., 2023) uses lexical candidates to support dense-scored graph exploration, demonstrating how one access mechanism can guide another. PEOs (Lu et al., 2024) acts within traversal, using partitioned random projections to probabilistically filter adjacent candidates before exact distance evaluation. Compression can also be optimized for this process: RPQ (Yue et al., 2024) learns a differentiable product quantizer using losses derived from graph neighborhoods and routing features. Representation and graph behavior are therefore already coupled in prior work. SIMSPARSE couples them through feature identifiers: the query's active coordinates address short document lists, and sparse inner products select entries and score subsequent traversal. This requires neither a separately trained router nor online recovery of dense document embeddings.

The organization of access matters across storage tiers as well. SPANN (Chen et al., 2021) keeps centroids in memory and posting lists on disk, using balanced clustering, closure-based posting expansion, and query-aware list pruning to balance candidate coverage against disk access. Here, routing selects corpus partitions whose postings form the main retrieval path; DiskANN (Jayaram Subramanya et al., 2019) instead follows graph links to access disk-resident vectors. SIMSPARSE's feature lists serve a narrower online purpose: they select starting documents for a corpus-wide graph search. This places its design at the intersection of feature-addressed access and graph navigation, with the same stored sparse representation supplying the scores throughout retrieval.