Approximate Similarity Joins over Dense Vector Embeddings

Douglas Rolins de Santana[1], **Leonardo Andrade Ribeiro**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG) – Goiânia – GO – Brazil

douglasrolins@discente.ufg.br,laribeiro@inf.ufg.br

Abstract. *We consider the problem of efficiently answering similarity join queries over vector data generated by machine learning models. Owing to the high dimensionality and density of such vectors, approximate solutions are prevalent for dealing with large datasets. In this context, we investigate how to evaluate similarity joins using the Hierarchical Navigable Small World (HNSW), a state-of-the-art, graph-based index designed for approximate k-nearest neighbor (kNN) queries. We explore the design space of possible solutions, ranging from alternatives on top of HNSW to deeper integration of similarity join processing into this structure. Experimental results show that our proposal achieves substantial speedups with negligible accuracy loss.*

1. Introduction

Recently, there has been a fast-growing interest in processing dense, high-dimensional vectors. This trend is fueled by modern machine learning models that embed complex unstructured data, such as text, image, and audio, into vector representations retaining semantically meaningful information. This data processing paradigm based on vector embeddings provides the backbone of a wide range of applications, including recommendation, video search, image-text retrieval, and question answering, among many others [Wang et al. 2021].

Queries over vector embeddings are typically based on similarity or distance measures such as Cosine similarity and Euclidean distance. Such queries come in two flavors: given an input vector v, the k-nearest neighbor (kNN) search returns the k vectors most similar to v, and the threshold-based search returns all vectors whose similarity with v is not less than a given threshold. Such flavors naturally apply to join queries, in which the k most similar vector pairs or all the vector pairs whose similarity is not less than the threshold are returned. In this paper, we focus on threshold-based join queries, called henceforth similarity joins.

Vectors generated by embedding models have intrinsic characteristics that make similarity join evaluation on large datasets challenging. First, these vectors are high-dimensional, which renders many indexing methods ineffective owing to the well-known "curse of dimensionality". Second, embedding models generate dense vectors, i.e., all dimensions contain non-zero values, in contrast to traditional term-based tokenization methods that generate sparse vectors, i.e., most of the values in the dimensions are zero. Several similarity join algorithms, such as AllPairs [Bayardo et al. 2007] and L2AP [Anastasiu and Karypis 2014], exploit vector sparsity to derive filters and reduce the comparison space. Unfortunately, such filters exhibit little to no pruning power on dense vectors, leading to a drastic drop in performance [Santana and Ribeiro 2022].

A common approach to dealing with the above issues is resorting to approximate solutions. Instead of always producing an exact answer, approximate algorithms may miss some valid results to trade accuracy for performance. In this context, the Hierarchical Navigable Small World (HNSW) [Malkov and Yashunin 2020] is among the most popular indexing structures for approximate kNN search. In a nutshell, HNSW is an in-memory, hierarchical organization of Delaunay graph approximations with tunable parameters for controlling recall-performance tradeoffs. Previous benchmark studies reported HNSW as state-of-art [Aumüller et al. 2020], which also matches our own experimental comparison (see Section 4).

In this paper, we investigate how to evaluate approximate similarity joins using HNSW. This indexing structure is designed for kNN search and does not natively support threshold-based queries. Thus, we first propose evaluating similarity joins on top of HNSW by judiciously selecting and incrementing the k parameter. Then, we explore the design space of deeper integration of similarity join processing into HNSW by modifying its internal algorithms for proximity graph building and searching. Our experimental study on several real-world datasets shows that the best-performing version of our proposal achieves up to 300x and 26x speedups over a exact method and external approach, respectively, with negligible accuracy loss.

The rest of this paper is organized as follows. In Section 2, we provide background material and formally define the problem of our focus. In Section 3, we present our contributions. Our experimental evaluation is presented in Section 4. We discuss related work in Section 5 and wrap up with the conclusions in Section 6.

2. Background

In this section, we first discuss techniques for embedding data into vectors before providing formal definitions of the problems considered in this paper. Finally, we overview the HNSW structure.

2.1. Embedding Techniques

A vast body of techniques for embedding text into numerical vectors has been developed over the years. Earlier approaches, such as Bag of Words and TF-IDF, relied on statistical methods to project text onto a discrete vector space. Such embedding models usually involve several thousands of dimensions as each term in the corpus corresponds to a dimension; thus, vectors representing sentences or documents are very sparse.

Embedding techniques based on neural networks project text onto a continuous vector space. Compared to statistical methods, the resulting embeddings have much lower dimensionality and are dense; note that these vectors still have high dimensionality, e.g., several hundreds of dimensions. State-of-the-art techniques employ language models based on the Transformer architecture [Vaswani et al. 2017]. These models are pre-trained on large text corpora, such as Wikipedia, in an unsupervised manner; BERT is the most popular pre-trained language model based on Transformers [Devlin et al. 2019]. As embeddings are generated considering the whole input sequence, they capture semantic and contextual information, including intricate linguistic aspects, such as polysemy and synonymy, that are not detected by earlier approaches. Besides text, the Transformer

architecture has been employed to produce embeddings for other modalities, such as image, audio, and video, as well as combinations of different data types in multimodal embeddings. In this paper, we focus on vector embeddings generated by neural networks; in the following, we assume that the input data has already been mapped to vectors.

2.2. Definitions and Terminology

We assume a set of real-valued vectors $\mathcal{V}$ of fixed dimensionality n. Given two vectors x and y, let $sim\,(x, y)$ be a commutative similarity function that returns a value in $[0, 1]$. We formally define relevant similarity operations to our context as follows.

Definition 1 (kNN Search) *Given a query vector x and an integer k, a kNN search over $\mathcal{V}$ returns the answer set $\mathcal{A}_{knn} = \{(\{y_1, ..., y_k\} \subseteq \mathcal{V} : \forall y \in \mathcal{A}_{kNN}$ and $\forall y' \in \mathcal{V} - \mathcal{A}_{kNN}, sim\,(x, y) \geq sim\,(x, y')\}$.*

Definition 2 (Similarity Search) *Given a query vector x and a similarity threshold τ, a similarity search over $\mathcal{V}$ returns the answer set $\mathcal{A}_{ss} = \{y \in \mathcal{V} : sim\,(x, y) \geq \tau\}$.*

Definition 3 (Similarity Join) *Given a similarity threshold $\tau \in [0, 1]$, a similarity join over $\mathcal{V}$ returns the answer set $\mathcal{A}_{sj} = \{(x, y) \in \mathcal{V} \times \mathcal{V} : sim\,(x, y) \geq \tau\}$.*

Obviously, we can compute the similarity join by performing a similarity search for each $x \in \mathcal{V}$ in a nested-loop join fashion. The following lemma expresses the condition for containment and equivalence between kNN and similarity search and, in turn, establishes a connection between kNN search and similarity join.

Lemma 1 *Given a query vector x, consider two search queries, kNN and SS, over $\mathcal{V}$: the former is a kNN search and returns the answer set $\mathcal{A}_{knn}$ for a given k value, and the latter is a similarity search and returns the answer set $\mathcal{A}_{ss}$ for a given threshold value τ. Let y_k be the vector with lowest similarity in $\mathcal{A}_{knn}$, i.e., $y_k \in \mathcal{A}_{knn}$ and $\forall y \in \mathcal{A}_{knn}, sim\,(x, y_k) \leq sim\,(x, y)$. Further, let y_{k+1} be the vector with highest similarity not in $\mathcal{A}_{knn}$, i.e., $y_{k+1} \notin \mathcal{A}_{knn}$ and $\forall y' \notin \mathcal{A}_{knn}, sim\,(x, y_{k+1}) \geq sim\,(x, y')$. If $sim\,(x, y_{k+1}) < \tau$, then SS is contained in kNN as $\mathcal{A}_{ss} \subseteq \mathcal{A}_{knn}$. If $sim\,(x, y_{k+1}) < \tau$ and $sim\,(x, y_k) \geq \tau$, then SS and kNN are equivalent as $\mathcal{A}_{ss} \subseteq \mathcal{A}_{knn}$ and $\mathcal{A}_{knn} \subseteq \mathcal{A}_{ss}$.*

Proof 1 (Sketch) *We prove Lemma 1 by contradiction. Suppose that $sim\,(x, y_{k+1}) < \tau$ and there exists a vector $y \in \mathcal{A}_{ss} \setminus \mathcal{A}_{knn}$, i.e., $\mathcal{A}_{ss} \not\subseteq \mathcal{A}_{knn}$. If $y \in \mathcal{A}_{ss}$, then $sim\,(x, y) \geq \tau$; but if $y \notin \mathcal{A}_{knn}$, then $sim\,(x, y_{k+1}) \geq sim\,(x, y) \geq \tau$, which contradicts with $sim\,(x, y_{k+1}) < \tau$. Now, suppose that $sim\,(x, y_k) \geq \tau$ and there exists a vector $y \in \mathcal{A}_{knn} \setminus \mathcal{A}_{ss}$, i.e., $\mathcal{A}_{knn} \not\subseteq \mathcal{A}_{ss}$. If $y \in \mathcal{A}_{knn}$, then $sim\,(x, y_k) \leq sim\,(x, y)$; but if $y \notin \mathcal{A}_{ss}$, then $sim\,(x, y_k) \leq sim\,(x, y) < \tau$, which contradicts with $sim\,(x, y_k) \geq \tau$.*

In this paper, we focus on the cosine similarity and assume that all input vectors have been normalized to unit length, i.e., $||x|| = 1, \forall x \in \mathcal{V}$. Thus, the cosine similarity between two vectors x and y corresponds to their dot product: $sim\,(x, y) \equiv dot\,(x, y) \equiv \sum_{i=1}^{n} x_i \times y_i$, where x_i is value of the *i*th dimension of x. Finally, we note that the techniques proposed in this paper also apply to the Euclidean distance.

2.3. HNSW

HNSW is an indexing structure based on a layered organization of proximity graphs. At each layer, an approximation of the Delaunay graph is built by preserving only links to the closest neighbors of a node. HNSW heuristically inserts long-range links to satisfy small-world navigation properties and, thus, achieve logarithmic complexity scaling of search operations. Links are organized into hierarchical layers according to their length. Figure 1 illustrates the HNSW multi-layer organization, which resembles a probabilistic skip list structure in which proximity graphs replace linked lists. The maximum layer of a node is determined through a random choice with exponential decay probability distribution. Search starts on the top layer, which contains only the longest links, and proceeds down the hierarchy every time a minimum local is reached in its greedy traversal, finishing at the bottom layer (Layer 0).

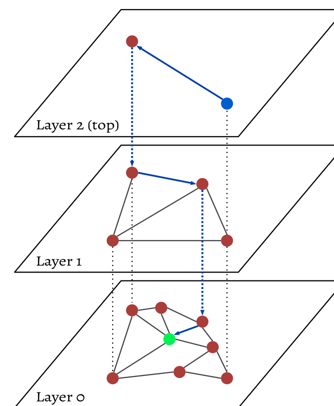

Figure 1. HNSW structure

HNSW is designed for approximate kNN search, which means the answer set $\mathcal{A}_{knn}$ may contain elements that are not within the actual k-nearest neighbors to the query vector. The parameters `efConstruction` and `ef` determine the number of neighbors considered during proximity graph building and searching, respectively, thereby controlling recall-performance tradeoffs. For each inserted element, the number of connections established with its closest neighbors is determined by the parameter `M`. In all layers, the maximum number of connections per node is also parameterized, which ensures the logarithmic complexity. HNSW is considered state-of-the-art for approximate kNN search [Aumüller et al. 2020] — we compare HSWN against competing solutions in the context of similarity join evaluation in Section 4 – and has been incorporated by modern vector DBMSs (e.g., [Wang et al. 2021]).

3. Approximate Similarity Join Algorithms

In this section, we present our HNSW-based similarity join (HSJ) algorithms. As mentioned in the previous section, HNSW returns approximate results; hence our proposed solutions are also approximate. Further, as HNSW is a kNN indexing structure and does not support threshold-based queries, we devise strategies to use this structure for similarity join. First, we propose an approach to evaluating similarity joins on top of HNSW based on the judicious handling of the k parameter. Then, we present algorithms that adapt the internals of HNSW to similarity search.

3.1. Similarity Join on Top of HNSW

A basic approach to evaluating similarity join using HNSW is to first index $\mathcal{V}$ and, then, perform a kNN search for each vector to find its similar counterparts, i.e., the $\mathcal{A}_{ss}$ answer set. To this end, one has to use for each kNN search a k value that establishes an equivalence or at least containment relationship with a similarity search based on the fixed threshold (recall Lemma 1); in the latter, some dissimilar vector have to be removed

from the answer set $\mathcal{A}_{knn}$. Of course, such k value is not known beforehand. An intuitive strategy is to issue multiple kNN searches with increasing k till all similar vectors are retrieved. The choices of the initial k value and its increment along the searches play a crucial role in the overall performance: if the chosen values are too small, a greater number of kNN searches are needed to retrieve $\mathcal{A}_{ss}$; if they are too large, many dissimilar vectors are retrieved incurring in unnecessary overhead.

Given the above observations, we now propose heuristics for selecting the initial k value and defining its increment at each interaction. For the initial k value, we use the HNSW parameter `ef` increased by a factor determined by the similarity threshold: $k_{initial} = ef + (1 - \tau) * ef$. The idea is to use a greater $k_{initial}$ for smaller thresholds. For the increment of k, we look at y_k, the least similar element in $\mathcal{A}_{knn}$. The k's increment follows the similarity of y_k to the current query vector, which monotonically decreases at each interaction: $k_{new} = k + ((1 - \tau) * k)/(1 - sim(x, y_k))$. The intuition behind this heuristic is to decrease k's increment as $sim(x, y_k)$ approaches τ.

Algorithm 1 describes HSJ-Ext, the external variant of HSJ that executes on top of HNSW without modifying its internal structure. First, all vectors are indexed in the HNSW graph (Line 4). Then, $\mathcal{V}$ is scanned again (Line 6), and for each vector x, the corresponding $\mathcal{A}_{knn} \supseteq \mathcal{A}_{ss}$ is computed by issuing kNN searches with x and increasing k (Lines 7–14); the initial k value and its increment are calculated according to the heuristics previously described (Line 5 and 11, respectively). The vectors in $\mathcal{A}_{knn}$ with similarity higher than τ are joined with vector x and inserted into $\mathcal{A}_{sj}$ (Lines 15-18).

HSJ-Ext fully benefits from the highly efficient kNN search procedure of HNSW. However, even with the proposed heuristics for determining the k value, a large number of searches may be needed for low thresholds. Each search requires a complete traversal of the proximity graph hierarchy from the top to the bottom layer. To avoid such repeated searches, we propose integrating similarity join processing into HNSW, as described next.

3.2. Similarity Join Integrated into HNSW

Algorithm 2 describes HSJ-Ths algorithm, which directly performs similarity searches within a modified HNSW structure. As in the previous algorithm, HSJ-Ths first indexes all vectors (Line 5) and subsequently pass over $\mathcal{V}$ again, issuing a similarity search query for each vector (Line 6). In contrast to HSJ-Ext, $\mathcal{A}_{ss}$ is now computed inside HNSW by the new RANGE-SEARCH method (Line 7). This modification allows HNSW to traverse the graph only once, finding similar elements above the specified threshold. Using the RANGE-SEARCH method, we can directly retrieve the $\mathcal{A}_{ss}$ set of similar elements, eliminating the need for additional post-processing.

Algorithm 4 details the RANGE-SEARCH method, which is based on the KNN-SEARCH method from the original HNSW paper. RANGE-SEARCH follows a similar approach in searching for objects in the upper layers (Lines 5–8) until it reaches the entry point in the bottom layer. At this layer (Line 9), it calls the new RANGE-SEARCH-LAYER method (Algorithm 5), which was based on the SEARCH-LAYER method in HNSW, to search for nodes similar to the query vector with the threshold as the stopping condition for graph traversal. The RANGE-SEARCH-LAYER method incorporates the following modifications: (1) the local minimum is only used as a stopping condition if the similarity is less than the threshold (Line 8); (2) it also checks if the similarity of the

Algorithm 1: HSJ-Ext

Input : Set of vectors $\mathcal{V}$, a threshold τ, the HNSW's parameters $M, efC,$ and ef
Output: The answer set $\mathcal{A}_{sj}$

1 $\mathcal{A}_{sj} \leftarrow \emptyset$
2 $\mathcal{A}_{knn} \leftarrow \emptyset$
3 $hnsw \leftarrow buildHNSW(\mathcal{V}.dimension, M, efC, ef)$
4 $hnsw.insertAll(\mathcal{V})$
5 $k_{initial} = ef + ((1 - \tau) * ef)$
6 **foreach** $x \in \mathcal{V}$ **do**
7 $\quad k \leftarrow k_{initial}$
8 $\quad \mathcal{A}_{knn} \leftarrow k$NN-SEARCH$(hnsw, x, k)$
9 $\quad y_k \leftarrow$ vector with the lowest similarity in $\mathcal{A}_{knn}$
10 $\quad$ **while** $sim(x, y_k) \geq \tau$ **do**
11 $\quad\quad k \leftarrow k + ((1 - \tau) * k)/(1 - sim(x, y_k))$
12 $\quad\quad \mathcal{A}_{knn} \leftarrow k$NN-SEARCH$(hnsw, x, k)$
13 $\quad\quad y_k \leftarrow$ vector with the lowest similarity in $\mathcal{A}_{knn}$
14 $\quad$ **end**
15 $\quad$ **foreach** $y \in \mathcal{A}_{knn}$ **do**
16 $\quad\quad$ **if** $sim(x, y) \geq \tau$ **then**
17 $\quad\quad\quad \mathcal{A}_{sj}.add(x, y)$
18 $\quad\quad$ **end**
19 $\quad$ **end**
20 **end**
21 **return** $\mathcal{A}_{sj}$

Algorithm 2: HSJ-Ths

Input : Set of vectors $\mathcal{V}$, a threshold τ, the HNSW's parameters $M, efC,$ and ef
Output: The answer set $\mathcal{A}_{sj}$

1 $\mathcal{A}_{sj} \leftarrow \emptyset$
2 $\mathcal{A}_{ss} \leftarrow \emptyset$
3 $ef = ef + ((1 - \tau) * ef)$
4 $hnsw \leftarrow buildHNSW(\mathcal{V}.dimension, M, efC, ef)$
5 $hnsw.addAll(\mathcal{V})$ `// insert all vectors with hnsw's INSERT algorithm`
6 **foreach** $x \in \mathcal{V}$ **do**
7 $\quad \mathcal{A}_{ss} \leftarrow$ RANGE-SEARCH$(hnsw, x, ef, \tau)$
8 $\quad$ **foreach** $y \in \mathcal{A}_{ss}$ **do**
9 $\quad\quad \mathcal{A}_{sj}.add(x, y)$
10 $\quad$ **end**
11 **end**
12 **return** $\mathcal{A}_{sj}$

current element to the query is greater than the threshold as a condition for including it as a candidate, in addition to the other conditions (Line 15); (3) it only removes candidates from the dynamic list if their similarity is smaller than the threshold (Line 18); and (4) includes in the answer set only elements with similarity above the threshold (Line 25). We found that the parameter `ef` cannot be disregarded in the search with the threshold as the stopping condition, as it defines a minimum number of candidates to consider and prevents reaching a false local minimum.

Finally, we propose HSJ-Ths^{Inc} (Algorithm 3), which uses the RANGE-SEARCH method in an incremental way. In contrast to the previous approach that performs a complete indexing of the vectors before the search, HSJ-Ths^{Inc} performs an incremental search at the moment each vector is added to the index (Lines 7 and 12, respectively). This approach has significant advantages, such as the reduction of the search scope, limited to the data already indexed, and avoidance of repeated processing of similar pairs.

4. Experiments

In this section, we present an experimental study of our algorithms for approximate similarity join. The objectives of our evaluation are: 1) to assess the performance of our techniques in terms of execution time and recall rate, i.e., the percentage of results returned by the approximate method compared to the exact method.; 2) to compare the performance of HNSW against competing non-graph solutions in the context of similarity join; 3) to compare the performance of the HSJ-Ths and HSJ-Ths^{Inc} versions with the external approach HSJ-Ext and also with the brute-force approach for exact results; and 4) to test the scalability of our proposals.

Algorithm 3: HSJ-Ths^{Inc}

Input : Set of vectors $\mathcal{V}$, a threshold τ, the HNSW's parameters $M, efC,$ and ef
Output: The answer set $\mathcal{A}_{sj}$

1. $\mathcal{A}_{sj} \leftarrow \emptyset$
2. $\mathcal{A}_{ss} \leftarrow \emptyset$
3. $ef = ef + ((1 - \tau) * ef)$
4. $hnsw \leftarrow buildHNSW(\mathcal{V}.dimension, M, efC, ef)$
5. **for** $x \in \mathcal{V}$ **do**
6.   **if** $hnsw \neq \emptyset$ **then**
7. $\mathcal{A}_{ss} \leftarrow$ RANGE-SEARCH($hnsw, x, ef, \tau$)
8.   **foreach** $y \in \mathcal{A}_{ss}$ **do**
9. $\mathcal{A}_{sj}.add(x, y)$
10.   **end**
11.   **end**
12.   INSERT($hnsw, x$) `// hnsw's INSERT algorithm`
13. **end**
14. **return** $\mathcal{A}_{sj}$

Algorithm 4: RANGE-SEARCH

Input : index hnsw, query vector x, size of the dynamic candidate list ef and threshold τ
Output: The answer set $\mathcal{A}_{ss}$

1. $W \leftarrow \emptyset$
2. $\mathcal{A}_{ss} \leftarrow \emptyset$
3. $ep \leftarrow$ get enter point for index
4. $L \leftarrow$level of ep
5. **for** $lc \leftarrow L$ ... *1* **do**
6. $W \leftarrow$ SEARCH-LAYER($x, ep, ef = 1, lc$) `// hnsw's SEARCH-LAYER alg.`
7. $ep \leftarrow$ get nearest element from W to x
8. **end**
9. $\mathcal{A}_{ss} \leftarrow$ RANGE-SEARCH-LAYER($x, ep, ef, lc = 0, \tau$)
10. **return** $\mathcal{A}_{ss}$

Algorithm 5: RANGE-SEARCH-LAYER

Input : query element x, enter points ep, size of the dynamic candidate list ef, layer number lc, threshold τ
Output: The answer set $\mathcal{A}_{ss}$

1. $v \leftarrow ep$ `// set of visited elements`
2. $C \leftarrow ep$ `// set of candidates`
3. $W \leftarrow ep$ `// dynamic list of found nearest neighbors`
4. $\mathcal{A}_{ss} \leftarrow \emptyset$
5. **while** $|C| > 0$ **do**
6. $c \leftarrow$ extract the most similar element from C to x
7. $f \leftarrow$ get the least similar element from W to x
8.   **if** *sim*$(c, x) <$ *sim*(f, x) *and sim*$(f, x) < \tau$ **then**
9.   break `// all elements in W are evaluated and threshold reached`
10.   **end**
11.   **foreach** $e \in$ *neighbourhood(c) at layer lc* **do**
12.   **if** $e \notin v$ **then**
13. $v \leftarrow v \cup e$
14. $f \leftarrow$ get the least similar element from W to x
15.   **if** *sim*$(e, x) \geq$ *sim*(f, x) *or* $|W| < ef$ *or sim*$(e, x) \geq \tau$ **then**
16. $C \leftarrow C \cup e$
17. $W \leftarrow W \cup e$
18.   **if** $|W| > ef$ *and sim*$(f, x) < \tau$ **then**
19.   remove the least similar element from W to x
20.   **end**
21.   **end**
22.   **end**
23.   **end**
24. **end**
25. **foreach** $y \in W$ **do**
26.   **if** $y.sim \geq \tau$ **then**
27. $\mathcal{A}_{ss}.add(y)$
28.   **end**
29. **end**
30. **return** $\mathcal{A}_{ss}$

4.1. Experimental Setup

We used four databases, three of which were semi-synthetically generated (DBLP[1], IMDB[2], and Spotify[3]), and one widely used benchmark dataset (Glove). For the Glove dataset, we utilized the version with 100 dimensions and 1,183,514 vectors provided by the ANN-Benchmarks[4] project. The characteristics of the semi-synthetic data are

[1] `http://dblp.uni-trier.de`
[2] `https://www.imdb.com/interfaces`
[3] `https://research.atspotify.com/datasets/`
[4] `https://github.com/erikbern/ann-benchmarks`

shown in Table 1, and the normalized dense vectors were generated using the Sentence-Transformers framework [Reimers and Gurevych 2019]. The experiments were conducted on a server equipped with an Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz (12 CPU) processor and 32 GB of RAM.

Datasets	Attributes	Min Size	Max Size	Avg Size	Records	Duplicates	Total	Dimensions
DBLP	Title, Author	6	1081	125	50000	2	100000	768
IMDB	Title, Actors, Description	31	477	161	20000	5	100000	384
Spotify	Artist_name, Track_name	5	204	33	50000	2	100000	384

Table 1. Description of semi-synthetic datasets

4.2. Similarity Joins with ANN Algorithms

Approximate nearest neighbor (ANN) algorithms play a crucial role in real-time applications and are specifically designed for kNN search. In the context of similarity join, it is possible to adapt the application of these algorithms to achieve approximate similarity join (recall Lemma 1). One simplified approach is to perform repeated top-k search calls, increasing the value of k at each iteration, until the least similar object returned in the search is below the established threshold. Some indexing structures provided by Faiss, such as the inverted file index (IVF) and the flat index, allow threshold-based searches. However, the HNSW, Annoy, and ScanN structures do not have this capability natively.

Threshold	**nmslib.brute_force**		**faiss.IndexLSH**		**faiss.IVFFlat**	
	Recall	Time (s)	Recall	Time (s)	Recall	Time (s)
0.6	100%	3431	95.18%	1190	92.99%	153
0.7	100%	2609	99.98%	1127	95.29%	153
0.8	100%	2023	100.00%	1127	97.81%	152
0.9	100%	1780	100.00%	1126	99.49%	152
Threshold	**Annoy**		**nmslib.HNSW**		**ScanN**	
	Recall	Time (s)	Recall	Time (s)	Recall	Time (s)
0.6	94.57%	327	97.43%	47	79.49%	88
0.7	97.33%	235	98.23%	39	97.29%	80
0.8	99.06%	186	99.12%	34	99.03%	71
0.9	99.26%	143	99.42%	31	99.81%	63

Table 2. Similarity Join Experiments with ANN Algorithms

To evaluate the application of HNSW compared to Faiss, Annoy, and ScanN in similarity join, we conducted initial experiments using the DBLP database indicated in Table 1. For each indexing structure, we implemented a two-step algorithm: (1) indexing all the vectors and (2) traversing each vector in the database and querying the index for similar objects above the similarity threshold using the cosine function. We employed the top-k search approach for the threshold, as mentioned earlier. In the case of Faiss with the IVF index, we performed a range-search query. For these experiments, we utilized the official libraries of Faiss[5], Annoy[6], HNSW[7], and ScanN[8], with Python bindings. We

[5]`https://github.com/facebookresearch/faiss`
[6]`https://github.com/spotify/annoy`
[7]`https://github.com/nmslib/nmslib`
[8]`https://github.com/google-research/google-research/tree/master/`

used the brute-force version of the nmslib library to obtain exact results. In Faiss, we employed approximate indexes, such as the LSH type, which applies locality-sensitive hash functions to maximize collisions and group vectors, and the IVFFlat type, which reduces the search scope through inverted file and clustering.

The results of these experiments can be seen in Table 2 and indicate that the HNSW structure achieved a better tradeoff between execution time and recall rate. HNSW obtained better execution times, from 31s to 47s, maintaining recall above 97%, even at lower thresholds. ScanN also achieved low execution times, but when lowering the threshold to 0.6 the recall drops dramatically. These results correspond to previous work that demonstrated that the HNSW structure provides state-of-the-art solutions compared to other competing methods. [Aumüller et al. 2020].

4.3. Similarity Joins with HNSW

We conducted experiments to evaluate the performance of our techniques in terms of execution time and recall rate, comparing them to the exact brute-force method and the external approach. The implementation was done in Java using the HNSW library available on GitHub[9] and exploiting the multicore processor parallelism. We used the values 64, 32, and 32 as parameters `M`, `efC`, and `ef`, respectively, in all executions. We performed multiple algorithm executions, varying the threshold from 0.5 to 0.9 with an increment of 0.1, and the recorded times represent the average of five executions. A maximum limit of 100 hours was defined for the execution of the algorithms. If this limit is reached, the execution is terminated. The algorithms are indicated in the graphs as follows: `BF` for brute force, `EXT` for HSJ-Ext, `THS` for HSJ-Ths, and `INC` for HSJ-Ths^{Inc}. The execution of the brute-force method on the GLOVE dataset with a threshold of 0.5 was interrupted due to reaching the time limit. Therefore, it was not possible to obtain the recall rates of the methods for this threshold and dataset.

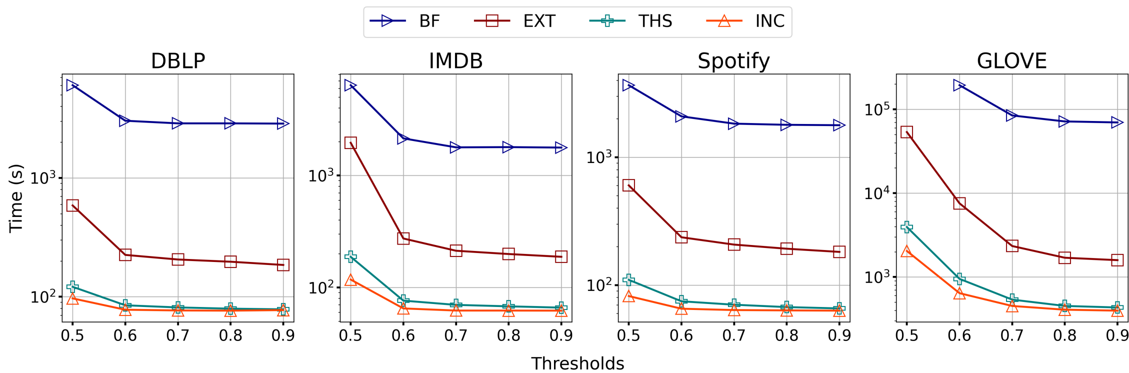

Figure 2. Runtime of methods

The runtimes are shown in Figure 2, and the recall rates are available in Table 3. The results demonstrate that our techniques significantly outperform the exact method, with recall rates approaching 100% for the different thresholds evaluated. In the GLOVE dataset and threshold 0.6, the exact method ran in 54 hours, while the HSJ-Ths^{Inc} took

scann
[9]https://github.com/jelmerk/hnswlib

Threshold	DBLP				IMDB			
	BF	EXT	THS	INC	BF	EXT	THS	INC
0.9	100%	99.99%	99.97%	99.94%	100%	99.77%	99.44%	99.95%
0.8	100%	99.95%	99.93%	99.87%	100%	99.45%	99.28%	99.81%
0.7	100%	99.92%	99.91%	99.61%	100%	99.21%	99.19%	98.83%
0.6	100%	99.90%	99.89%	99.41%	100%	99.03%	98.98%	98.48%
0.5	100%	99.37%	99.23%	98.80%	100%	99.25%	98.85%	98.71%
Threshold	**SPOTIFY**				**GLOVE**			
	BF	**EXT**	**THS**	**INC**	**BF**	**EXT**	**THS**	**INC**
0.9	100%	99.97%	99.91%	99.93%	100%	99.75%	99.74%	99.75%
0.8	100%	99.90%	99.87%	99.88%	100%	99.69%	99.67%	99.59%
0.7	100%	99.87%	99.80%	99.61%	100%	98.82%	99.03%	98.89%
0.6	100%	99.86%	99.75%	99.17%	100%	98.01%	98.33%	98.17%
0.5	100%	98.61%	98.24%	97.50%	-	-	-	-

Table 3. Methods recall rate

10.6 minutes (300x) with a recall of 98.17%. Additionally, we observed that the HSJ-Ths and HSJ-Ths^{Inc} versions, which utilize the new HNSW graph search method, the latter employing incremental indexing, outperformed the external approach HSJ-Ext in all datasets, achieving maximum speedups of 13x and 26x, respectively. These speedups were achieved in the GLOVE dataset and threshold of 0.5, with runtimes of 34, 65.8, and 894 minutes for HSJ-Ths^{Inc}, HSJ-Ths, and HSJ-Ext, respectively.

Regarding multicore parallelism, non-incremental approaches with all vectors pre-indexed avoid synchronization issues between processes, allowing independent searches across the entire index. However, the incremental approach may face challenges in synchronization, potentially leading to incomplete results in certain searches. Nonetheless, experimental results demonstrate that despite parallelism in the incremental version, the recall rate remains comparable to that of the non-incremental version. This outcome is likely due to the distribution of input vectors, which mitigates synchronization problems. Further evaluations with diverse datasets are necessary to validate these findings.

The HNSW index's memory consumption is crucial to consider during its construction. The number of connections determined by the parameter M affects the average memory consumption per element. In addition, the dimensionality and number of vectors also affect. In the experiments carried out, in the dataset of 100,000 vectors with 384 and 768 dimensions, the index size was 375 Mb and 675 Mb, respectively. For 1,183,514 vectors with 100 dimensions, the index size reached 1,815 Mb.

5. Related Work

L2AP [Anastasiu and Karypis 2014] is a state-of-the-art algorithm for exact similarity join over sparse vectors using based on cosine similarity. It introduces, primarily, the filtering based on the Cauchy-Schwarz inequality, achieving superior performance even when compared to approximate techniques. However, L2AP relies on the characteristics of sparse vectors to derive the filters, and it has been shown to have poor efficiency in dense vector spaces [Santana and Ribeiro 2022].

Numerous works have been carried out to improve the performance of similarity join on set-represented data, especially for exact results [Ribeiro and Härder 2011,

Christiani et al. 2018]. For dense vectors, the focus has been on parallelism and the use of GPUs to achieve exact results [Johnson et al. 2019], as well as the application of dimensionality reduction or quantization for approximate results [Paparrizos et al. 2022]. However, the quantity and dimensionality of the data still have a direct impact on the execution time for obtaining exact results. Furthermore, when applying quantization and dimensionality reduction techniques, despite significant performance gains, the recall rate experiences a drastic decrease.

With the increasing use of dense vector embeddings, particularly in the fields of data science and machine learning, there have been works aimed at optimizing similarity search over such vector spaces. An important aspect of these research efforts is the development of indexing structures and algorithms for approximate nearest neighbor (ANN) search. Among the prominent algorithms developed are FAISS [Johnson et al. 2019], Annoy [Bernhardsson 2015], Hierarchical Navigable Small World (HNSW) [Malkov and Yashunin 2020], and ScanN [Guo et al. 2020]. These methods have been designed for kNN search and are not specifically optimized for similarity join.

Among the mentioned algorithms, the HNSW stands out for offering significant improvements in search time with minimal reduction in recall [Echihabi et al. 2019, Aumüller et al. 2020]. It has been widely used in various applications, ranging from real-time searches to data discovery in data lakes [Fan et al. 2023]. Considering these characteristics, this work explores the use of this structure for performing similarity joins.

6. Conclusions

In this paper, we presented algorithms for approximate similarity join using the HNSW graph structure. We propose three algorithms for approximate similarity join and a novel graph search method based on a threshold as the stopping criterion. Experimental results demonstrate that approximate similarity join with HNSW is an efficient and accurate approach for retrieving similar object pairs in large datasets. Compared to the exact method and external approach, the proposed technique reduced execution time by up to 300x and 26x, respectively, while maintaining recall rates close to 100%. These results highlight the potential of leveraging the HNSW structure for similarity joins, expanding its applicability beyond kNN search. In future work, we suggest exploring additional optimizations, such as exploiting the threshold in graph construction and investigating other indexing structures to further enhance similarity join operation on dense vectors.

References

Anastasiu, D. C. and Karypis, G. (2014). L2AP: Fast Cosine Similarity Search with Prefix L-2 Norm Bounds. In *Proceedings of the ICDE Conference*, pages 784–795.

Aumüller, M., Bernhardsson, E., and Faithfull, A. J. (2020). ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *Information Systems*, page 101374.

Bayardo, R. J., Ma, Y., and Srikant, R. (2007). Scaling up All Pairs Similarity Search. In *Proceedings of the WWW Conference*, page 131–140.

Bernhardsson, E. (2015). Spotify/Annoy: Approximate Nearest Neighbors in c++/python Optimized for Memory usage and Loading/saving to Disk. https://github.com/spotify/annoy.

Christiani, T., Pagh, R., and Sivertsen, J. (2018). Scalable and Robust Set Similarity Join. In *Proceedings of the ICDE Conference*, pages 1240–1243.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186.

Echihabi, K., Zoumpatianos, K., Palpanas, T., and Benbrahim, H. (2019). Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *Proceedings of the VLDB Endowment*, page 403–420.

Fan, G., Wang, J., Li, Y., Zhang, D., and Miller, R. J. (2023). Semantics-Aware Dataset Discovery from Data Lakes with Contextualized Column-Based Representation Learning. *Proceedings of the VLDB Endowment*, page 1726–1739.

Guo, R., Sun, P., Lindgren, E., Geng, Q., Simcha, D., Chern, F., and Kumar, S. (2020). Accelerating Large-Scale Inference with Anisotropic Vector Quantization. In *Proceedings of the International Conference on Machine Learning*, pages 3887–3896.

Johnson, J., Douze, M., and Jégou, H. (2019). Billion-scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, pages 535–547.

Malkov, Y. A. and Yashunin, D. A. (2020). Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 824–836.

Paparrizos, J., Edian, I., Liu, C., Elmore, A. J., and Franklin, M. J. (2022). Fast Adaptive Similarity Search through Variance-Aware Quantization. In *Proceedings of the ICDE Conference*, pages 2969–2983.

Reimers, N. and Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 3982–3992.

Ribeiro, L. A. and Härder, T. (2011). Generalizing Prefix Filtering to Improve Set Similarity Joins. *Information Systems*, pages 62–78.

Santana, D. and Ribeiro, L. (2022). Junções por Similaridade em Espaços Vetoriais Semânticos. In *Anais Estendidos do XXXVII Simpósio Brasileiro de Bancos de Dados*, pages 147–153.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is All you Need. In *Annual Conference on Neural Information Processing Systems*, pages 5998–6008.

Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., Yu, K., Yuan, Y., Zou, Y., Long, J., Cai, Y., Li, Z., Zhang, Z., Mo, Y., Gu, J., Jiang, R., Wei, Y., and Xie, C. (2021). Milvus: A Purpose-Built Vector Data Management System. In *Proceedings of the SIGMOD Conference*, pages 2614–2627.