Graph Querying or Similarity Search? Both!

Vicente Calisto[1], Sebastián Ferrada[1,2,5], Gonzalo Navarro[1,3], Juan L. Reutter[1,4], Juan Pablo Sánchez[1,4], and Domagoj Vrgoč[1,4]

[1] Millennium Institute for Foundational Research on Data (IMFD), Chile
[2] IDIA, Universidad de Chile, Santiago, Chile
[3] DCC, Universidad de Chile, Santiago, Chile
[4] Pontificia Universidad Católica de Chile, Santiago, Chile
[5] National Center for Artificial Intelligence Research (CENIA), Chile
`vicente.calisto@imfd.cl, {sebastian.ferrada,gnavarro}@uchile.cl, jreutter@ing.puc.cl, {jpsanchez, vrdomagoj}@uc.cl`

Abstract. Extracting information from knowledge graphs is a significant algorithmic challenge, especially when dealing with multimodal knowledge graphs that integrate images, text, and/or videos. While current graph management systems can efficiently evaluate graph queries, they struggle with multimedia data. To address this, systems rely on metadata, such as vector embeddings, for similarity search. While both graph pattern evaluation and similarity search work well independently, real-world applications often require their combination to retrieve media based on both the graph structure and specific similarity criteria.

This paper studies the problem of querying multimodal knowledge graphs by combining graph patterns with similarity constraints. We formalize this as an extraction task where some nodes in the graph pattern are filtered by similarity, and then the results must be ordered by a similarity score. While a straightforward approach is to evaluate the graph pattern first and then sort by similarity, we introduce alternative algorithms that evaluate both tasks jointly, leveraging indices for efficient similarity computation. Our implementation employs an approximate version of these indices, and our experiments show that graph database systems can efficiently integrate semantic similarity constraints into their queries.

1 Introduction

Similarity search plays a crucial role in Information Retrieval, Machine Learning, and Retrieval-Augmented Generation (RAG). It involves finding objects in a dataset that are *similar* to a given query, which is essential for document retrieval [7], recommendation systems [1], and pattern recognition [41] tasks. In RAG, similarity search enables the retrieval of knowledge relevant to the query to enhance generative models, leading to more accurate and contextually informed outputs [32]. Further, with the rise of vector representations of text, images, and other media, the quest for efficient similarity search has become increasingly relevant for querying large-scale, high-dimensional data [55,26,13,8,22].

```
SELECT ?tower ?img ?dist WHERE {
?tower wdt:P31/wdt:P279* wd:Q12518; wdt:P17 ?country; wdt:P18 ?img. ?country wdt:P361 wd:Q18.
wd:Q151356 :hasVector ?vector.
?img proc:hnswIterator ("vectorIndex" ?vector ?dist)
} LIMIT 10
```

Fig. 1: Query to get the S. American towers most similar to the Berlin TV Tower.

Knowledge Graphs (KGs) are structured representations of information where entities are connected through relationships. KGs facilitate complex queries, reasoning, and inferences [30]. KGs are used in semantic search [56], recommendation systems [24], and natural language processing [46], etc. In this work, we focus on Multimodal Knowledge Graphs (MKGs), where the base data remains in RDF, and other data modalities (such as images, text, or videos) are incorporated into the graph via precomputed vector embeddings. These embeddings are treated as additional attributes associated with RDF entities, allowing for more nuanced and expressive queries, where users may wish to retrieve subgraphs that satisfy structural constraints and exhibit similarity in vector spaces. Consequently, supporting similarity-aware querying in MKGs requires novel query semantics and evaluation strategies that go beyond traditional graph pattern matching or vector retrieval in isolation. Examples of MKGs include TIVA-KG [53], Richpedia [52], and IMGpedia [20].

In this paper, we investigate the problem of how to efficiently evaluate a graph pattern query Q over an MKG G while also imposing a similarity-based restriction on the results. Specifically, the query results $Q(G)$ must be sorted by the distance of the vectors associated with the bindings of a fixed variable in Q and an input vector or with the vector associated with a given entity $a \in G$. We are mostly interested in the case when the k best answers are wanted. The integration of graph pattern matching with similarity search introduces new computational challenges, including efficiently combining the two tasks and leveraging existing indices for optimization, presenting new opportunities for improving query evaluation strategies in MKGs. Furthermore, such techniques have broader applications, such as improving KG-based RAG [39] by incorporating both graph structure and similarity constraints into the retrieval process.

For instance, consider combining Wikidata [48], a large KG with structured information about various domains, and IMGpedia [20], an MKG that links Wikimedia Commons images and their corresponding embeddings to Wikidata entities. Suppose we want to retrieve images of communication towers in South America that are visually similar to an image of the Berlin TV Tower. To achieve this, we first construct a graph pattern query Q that identifies entities classified as "tower" (`wd:Q12518`) and located in South America (`wd:Q18`). Next, we apply a similarity condition to the images of these towers, requiring them to be among the 10-nearest neighbors of the Berlin TV Tower's image in Wikimedia Commons (`wd:Q151356`) based on their embeddings from IMGpedia. This approach allows us to retrieve 254 images of towers that match the graph pattern and, from them, the 10 most similar to the Berlin TV Tower. Fig. 1 illustrates how we express this example query using our own SPARQL extension with similar-

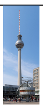
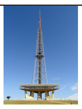
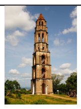
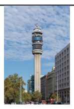
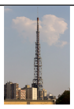
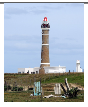
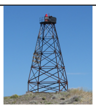
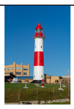
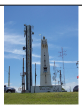
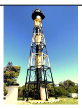
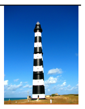
Index	0	1	2	3	4	5	6	7	8	9	10
`?image`											
`?sim`	1.0	0.763	0.727	0.727	0.722	0.717	0.710	0.708	0.708	0.703	0.702

Fig. 2: Results of evaluating the query from Fig. 1 over Wikidata.

```
SELECT ?pact (COUNT(?twt) AS ?nb) (AVG(?dist) AS ?mean) WHERE {
 ?member telar:hasAccount ?account ;
      telar:politicalGroupConvention ?pact.
 ?twt telar:postedBy ?account ; telar:createdAt ?date .
 FILTER(?date>="2022-02-17T02:51:20"
     && ?date<="2022-02-17T14:51:20")
 telar:twp_1494142390663360512 :hasVector ?vector
 ?twt proc:hsnwIterator ( "tweets_sbert" ?vector ?dist )
} GROUP BY ?pact
```

`?pact`	`?nb`	`?mean`
Frente Amplio	95	0.50
Partido Comunista	41	0.51
Vamos por Chile	27	0.64
UDI	11	0.68

Fig. 3: Query searching for polarization on opinions about regionalization.

ity indices and vectors. In our solution, RDF triplestores are extended with a special vector literal with a datatype that appears in the object position. The similarity part of the query is expressed using the `proc:hnswIterator` clause, which must receive as input the vector embedding of the image of the Berlin TV Tower (`wd:Q151356`), which is achieved through the predicate `:hasVector`. This clause mandates that similar items are to be extracted through an HNSW vector index [33] Fig. 2 contains the final 10 images of the answer.

Our setting also allows for similarity searches w.r.t. an input vector instead of a node in the KG. This problem appears continuously in the context of RAG. For example, consider a user searching for political discourses in Congress, asking for a summary of interventions on climate change. Using a RAG-based approach, we can store the vector embedding of each discourse in Congress at indexing time and use the same embedding model to embed the query "summary of discussions about climate change" at query time. Then, we retrieve the 10 most similar interventions and passed them to an LLM for summarization. Now, suppose the user asks for a "summary of discussions about climate change by congresspeople over 60." Here, a standard RAG approach might not work as intended since it could retrieve interventions from younger members of Congress. By leveraging a KG, we could first filter members of Congress over 60, ensuring only their interventions are considered. Then, we can retrieve the 10 most similar discourses from this filtered set, refining the search before passing the results to the LLM.

As a final example, we use TelarKG [2], an MKG of the Chilean constitutional process enriched with S-BERT [44] embeddings for convention members' social media posts and interventions. We query it to retrieve the members' opinions on regionalization, based on a seed neutral tweet, and compute their average similarity by political pact. The query combines graph patterns (linking members, posts, and pacts) with vector similarity over text. Figure 3 shows the query and select results, revealing the expected ideological polarization of opinions.

In this paper, we design algorithms to evaluate these queries jointly. We compare our algorithms to a *query-and-select* baseline, which first evaluates the graph pattern and then selects the results with the shortest distances. Notably, our algorithms offer theoretical guarantees in a combined setting: they are worst-case optimal in the number of results and range-optimal in distance computations. We also provide a complete implementation of our approach on top of an existing open-source SPARQL engine [35]. Experiments show that our algorithms significantly accelerate MKG query answering with broad applicability.

Before detailing our algorithms (Sections 5–6), we cover related work (Section 2), preliminaries (Section 3), and the problem statement (Section 4). Implementation and evaluation are in Sections 7–8, and conclusions in Section 9.

2 Related Work

The problem of retrieving objects based on nearest neighbors has recently received a lot of attention in the context of Vector Databases, which are systems built specifically for obtaining the nearest neighbors of (embeddings of) objects. For more information on the latest advances in the area, we refer to the survey by Pan et al. [38]. Notably, some of these systems allow for *filtered similarity search*, wherein users can input certain filter conditions together with the input vector, and the system retrieves only the nearest neighbors that satisfy the filter conditions [23,19,40,51,25]. Further, ChromaDB [15], a popular vector database, has specified a formal language to carry out these operations: here documents have JSON metadata, and filter conditions refer to the presence of certain values in this metadata. For example, one writes `{gender: female}` to ask only for documents where the `gender` metadata corresponds to `female`. We note that our problem is more general, as filter conditions can be expressed as triple patterns over KGs, but complex graph queries cannot be transferred to filter conditions unless precomputed at indexing time, which is not compatible with the idea that systems should support evaluating any possible pattern with similarity. Moreover, Vector Databases normally use a version of one of our algorithms (Iterate-and-Query, see Algorithm 3) to evaluate filtered similarity search, and hence studying other options for solving filtered similarity search should also be of interest to Vector Databases implementing this feature.

We are aware of only two systems that support answering graph queries based on similarity conditions of an arbitrary vector, which is the focus of this paper: MillenniumDB [49,50], the database system we use as the framework to experiment with, and pgvector [42], a Vector Database built as a PostgreSQL component that allows answering a subset of SQL queries based on similarity constraints. Using the formalization of the problem that we provide in this paper, we see that these systems evaluate queries using the baseline that we present in Section 5. Interestingly, the need for more efficient algorithms to answer queries based on similarity conditions has already been stated by contributors of pgvector [17]. We expect our work to guide future contributions to these systems and pave the path for more systems supporting similarity search in MKGs.

Lastly, Arroyuelo et al. study query answering where patterns feature similarity conditions between variables [3]. This work relies on precomputed indices for processing similarity conditions and hence does not support querying for arbitrary input vectors. Likewise, previous work on MKGs (e.g., [53,52,20]) does not efficiently support queries with arbitrary vectors not present in the KG.

3 Preliminaries

RDF and SPARQL. RDF [16] is the standard graph data model for the Semantic Web. RDF terms can be either IRIs ($\mathcal{I}$), literal values ($\mathcal{L}$), or blank nodes ($\mathcal{B}$). An RDF triple is a tuple $(s, p, o) \in \mathcal{IB} \times \mathcal{I} \times \mathcal{IBL}$. An RDF Graph is a finite set of RDF triples. The fundamental querying primitives in SPARQL [27] are Basic Graph Patterns (BGPs). Let $\mathcal{V}$ be a universe of variables disjoint from $\mathcal{IBL}$. A BGP Q is a set of triple patterns $(x, y, z) \in \mathcal{IV} \times \mathcal{IV} \times \mathcal{ILV}$. The output $Q(G)$ of the BGP when evaluated in a graph G is a set of solution mappings μ, where $\mu : \mathcal{V} \cup \mathcal{I} \to \mathcal{I}$ is an assignment such that for each triple pattern $(x, y, z) \in Q$, it holds that $(\mu(x), \mu(y), \mu(z)) \in G$, where $\mu(x) = x$ for all $x \in \mathcal{I} \cup \mathcal{L}$. We denote as $\mathsf{vars}(Q)$ the set of variables appearing in the triple patterns of Q.

Worst-case-optimal joins. The *AGM bound* [5] establishes the maximum output size of a join query. This bound can also be applied to graph patterns by regarding each triple pattern as a relation formed by the triples matching its constants [31]. Formally, the AGM bound of a query Q over a graph database G, denoted as $Q^*(G)$, or just Q^* if G is understood from context, is the maximum size $Q(G')$ could have over any database instance G' with at most the number of edges in G. An algorithm to process queries is *worst-case optimal (wco)* if it has a running time in $\tilde{O}(Q^*)$, where $\tilde{O}$ ignores polylogs and data-independent factors. In this paper, we use Leapfrog TrieJoin (LTJ), which was shown to be worst-case optimal for any join query, including graph patterns, by Veldhuizen [47].

Similarity search. Similarity search over a set of objects $\mathcal{D}$ from a universe $\mathcal{U}$ relies on a similarity score, the higher the score, the more similar the objects are. Similarity scores are often measured as distances. Given a *distance function* $\delta : \mathcal{U} \times \mathcal{U} \to \mathbb{R}_0^+$, smaller distances indicate greater similarity.

A common similarity task is the search for the k *nearest neighbors* in $\mathcal{D}$ of a query object $q \in \mathcal{U}$, which are k elements of $\mathcal{D}$ that are closest to q w.r.t. distance δ. Formally, the problem is to find the set $k\text{-NN}(q)$ such that $q \notin k\text{-NN}(q)$, $|k\text{-NN}(q)| = k$, and $\forall x \in k\text{-NN}(q), \forall y \in \mathcal{D} \setminus k\text{-NN}(q), y \neq q$, $\delta(q, x) \leq \delta(q, y)$.

A naive search algorithm computes the distance from q to all $x \in \mathcal{D}$; computing $n = |\mathcal{D}|$ distances. To reduce the cost, objects in $\mathcal{D}$ can be *indexed* to prune objects that are too far from q without computing their distance to q. In *metric spaces*, the triangle inequality $\delta(x, y) \leq \delta(x, q) + \delta(q, y)$ is used to prune distance computations, so that if we compute $\delta(q, x)$ and the index knows a bound $\delta(x, y) \geq d$, we know that $\delta(q, y) \geq d - \delta(q, x)$ and might avoid computing it. Some relaxations of the triangle inequality can be used for non-metric spaces.

Several indices exist for metric spaces [45,55], including the particular case where the space is $\mathbb{R}^d$ and the distance function is an L_p metric, where a famous example is the R-tree [26], which works well for low values of d; its analogous for general metric spaces is the M-tree [13]. Most indices are designed to solve *range queries*, which, given $q \in \mathcal{U}$ and $r > 0$, return all $x \in \mathcal{D}$ such that $\delta(q,x) \leq r$. k-NN queries can be solved on those indices in various ways, for example, with a sequence of range searches with increasing values of r until k or more elements are retrieved. There are also solutions that backtrack on the index, looking for the k elements closest to q. We call those solutions *KNN indices*.

A well-known technique [28,29] that works on hierarchical indices (like the R-tree) keeps a *priority queue* with the maximal nodes of the hierarchy yet to be traversed, sorted by a lower bound on the distance from q to any object in the node. The algorithm iterates, extracting the first node from the queue and reinserting its children in the queue; leaves of the index insert objects in the queue, computing their distance to q. Objects extracted from the queue are reported at once; it can be seen that the first k reported objects are k-NN(q).

An important property of this technique is that it is *range-optimal*. It finds k-NN(q) with the same number of distance computations the index would use to compute a range search that retrieves those k objects. Further, this method is *incremental*: it successively retrieves the next closest element to q, so we obtain k-NN(q) by stopping after k objects are retrieved, but we can also use it as an iterator that yields the next closest element to q. We call this operation $\mathit{next}_{\mathcal{D}}(q)$, and call *INN* those indices supporting this incremental search.

On *high-dimensional* spaces (where the histogram of δ is very concentrated), indices may fail to avoid performing many distance computations. This is called the *curse of dimensionality*. A way to measure the *intrinsic* dimensionality of a dataset in a metric space is $\mu^2/(2\sigma^2)$, where μ and σ^2 are the mean and variance of the histogram of δ [11], which is $\Theta(d)$ on random vectors in the $(\mathbb{R}^d, L_p)$ metric space. An extreme example is the distance $\delta(x,x) = 0$ for all x and $\delta(x,y) = 1 + \epsilon(x,y)$ for all $y \neq x$, where $\epsilon(x,y) \ll 1$ is a random value; it is impossible to save any distance computation in this space. In general, an intrinsic dimension over 20 is considered to be intractable.

In those high-dimensional spaces, finding an *approximation* to k-NN(q) may be acceptable because modeling similarity with distance functions is already an approximation to the desired answers. A good approximation may offer much lower search costs at the expense of a small difference between the correct and the returned set k-NN(q). There exist various algorithms to compute k-NN(q) approximately on L_p metrics, with various sorts of guarantees [4,54,43], as well as with no approximation guarantees but performing well in practice [22,8]. Some techniques for general metric spaces offer probabilistic approximation guarantees [14,12], while others offer no guarantees but perform well in practice [10].

Cost model. In range or nearest-neighbor searches, the number of evaluations of the distance δ dominates all the other CPU costs; thus, it is customary to count only the number of evaluations of δ. This is not the case in our problem: solving graph patterns involves a significant CPU cost, which can be superlinear on the database size, whereas the number of evaluations of δ is at most linear. For this reason, we describe the time complexities of algorithms as a pair $\langle c, e\rangle$, where e is the number of evaluations of δ and c includes all other CPU costs.

4 Top-*k* Graph Similarity Search

Let us now formally introduce the problem we study in the paper. We assume our MKG G includes a vector store τ, which assigns vector embeddings to some (or all) nodes in G. For a node b in G, $\tau(b)$ represents its vector embedding. We assume a distance function δ to measure the similarity among the vectors in τ. Additionally, τ is indexed to support incremental search for the nearest neighbors of a given vector t, which we denote as $\text{next}_\tau(t)$.

With these definitions, we want to allow users to add a similarity clause to graph patterns. Recall that $Q(G)$ denotes the evaluation of a graph pattern Q over an MKG G. Let $x \in \mathsf{vars}(Q)$ be a fixed variable. We call (Q, x) a graph similarity pattern. Note that we can write the tuples in $Q(G)$ as $(\bar{a}, b)$, where b is the element bound to x and $\bar{a}$ are the elements bound to the variables in $\mathsf{vars}(Q) \setminus \{x\}$. Now, we select the tuples from $Q(G)$ that have the lowest value for $\delta(t, \tau(b))$: the distance from an input vector t to $\tau(b)$. Particularly, we want the k-best answers from $Q(G)$ (handling ties in any way). The problem Top-k Graph Similarity Search (or k-GSS) consists of finding such k best answers to an arbitrary graph similarity pattern w.r.t. their distance to an arbitrary vector:

Top-k Graph Similarity Search (k-GSS)	
Given:	Graph G, distance function δ, vector store τ.
Input:	Graph pattern (Q, x) with similarity, vector t and integer k.
Output:	k best answers $(\bar{a}, b) \in Q(G)$, ordered by $\delta(t, \tau(b))$

Revisiting the query from Fig. 1, the input graph pattern is stated in a `WHERE` clause. The last triple pattern sets `?img` as the variable that is subject to similarity, and the vector bound to `?vector` is given as the embedding of node `wd:Q151356`. `?dist` is the fresh variable to which the distance from the vector of `?img` and `?vector` is bound. As discussed in Section 7, the `proc:hsnwIterator` function supports both user-provided vectors (`type:tensorFloat` constants) and vectors retrieved from the MKG (e.g., the embedding of a given node).

5 A first baseline: Query-and-select

A naive algorithm for k-GSS is to (1) evaluate Q over G, (2) compute $\delta(t, \tau(b))$ for each $(\bar{a}, b) \in Q(G)$, and (3) select those with the k shortest distances.

This procedure is outlined in Algorithm 1. The computation of d can be done with any classic linear-time algorithm for computing quantiles. If there are ties in the distances, the k tuples are selected as follows: in a first pass over **Ans**,

Algorithm 1 Baseline k-GSS via query-and-select

```
Require: Graph $G$, distance $\delta$, vector store $\tau$, Query $(Q, x)$, vector $t$ and integer $k$
Ensure: Top $k$ answers $(\bar{a}, b) \in Q(G)$, ordered by $\delta(t, \tau(b))$
  Ans ← $Q(G)$                                          ▷ using some wco algorithm
  for $(\bar{a}, b)$ in Ans do
      associate with $(\bar{a}, b)$ the value $dist = \delta(t, \tau(b))$.
  end for
  $d$ ← the $k$th smallest $dist$ value in Ans
  OAns ← $k$ tuples $(\bar{a}, b) \in$ Ans with $dist \leq d$
  Return OAns
```

we report the tuples $(\bar{a}, b)$ with $dist < d$. Say we reported $k' < k$ tuples in this pass; then, in a second pass, we report the first $k - k'$ tuples with $dist = d$.[6]

This algorithm can be pipelined using a max-priority queue P with the k tuples $(\bar{a}, b)$ with the smallest $dist$ values seen so far. For each new tuple $(\bar{a}, b)$ found by the wco algorithm, we compute $dist{=}\delta(t, \tau(b))$ and, if $dist < \max(P)$, we extract the maximum of P (if $|P|{=}k$) and insert $(\bar{a}, b)$. Despite the $O(\log k)$ extra CPU time per tuple in $Q(G)$, this version likely performs better in practice.

Overall, this query-and-select algorithm takes time $\tilde{O}(Q^*)$ to compute $Q(G)$, then it performs $|Q(G)|$ distance computations, and finally it spends $O(|Q(G)|) \subseteq O(Q^*)$ (or $O(|Q(G)| \log k) \subseteq \tilde{O}(Q^*)$ if pipelined) time to compute d and find the k smallest distances. Its time complexity, using our model, is then $\langle \tilde{O}(Q^*), |Q(G)| \rangle$.

An advantage of this algorithm is its simplicity, not needing any KNN index. However, it must find *all* the results in $Q(G)$, which may be prohibitive for queries with many results. Further, it requires computing the distance of the vector t to *every* query result. Notably, its time complexity is essentially independent of k, unlike most KNN indices that perform better on smaller k.

Throughout the paper, we assume the use of a wco join algorithm for the graph query evaluation to analyze the theoretical cost of our methods. We acknowledge that such algorithms are not currently adopted in many existing graph database systems. Nevertheless, we emphasize that our algorithms are compatible with other join strategies as well; while doing so may forgo the theoretical guarantees, the usefulness of the algorithms is preserved.

Next, we explore algorithms that compute k-GSS faster, especially for small values of k. We also consider combining the wco algorithm with approximate KNN indices to further speed up queries at the cost of slight result degradation.

6 Combining wco and KNN search

In this section, we provide alternatives to the query-and-select algorithm. These alternatives combine a wco algorithm for the graph pattern with INN indices that support incremental KNN searches from t in our vector store τ via the iterator

[6] Even if all the distances to t are distinct, there can be many tuples $(\bar{a}, b)$ for the same value of b, and they will all appear in **Ans**.

Algorithm 2 k-GSS via query-and-iterate

```
Require: Graph $G$, distance $d$ and indexed vector store $\tau$, $next_\tau()$
Require: Query $(Q, x)$, vector $t$ and integer $k$
Ensure: Top $k$ answers $(\bar{a}, b) \in Q(G)$, ordered by $\delta(t, \tau(b))$
  Ans ← $Q(G)$                                         ▷ using some wco algorithm
  $H$ ← dictionary of tuples $(\bar{a}, b) \in$ Ans with $b$ acting as key
  OAns ← $\emptyset$
  for each next element $b = next_\tau(t)$ do
    if $b$ is a key in $H$ then
      Add to OAns every tuple of the form $(\bar{a}, b) \in$ Ans
      if $|$OAns$| \geq k$ then
        Return OAns capped to the first $k$ tuples
      end if
    end if
  end for
  Return OAns                                          ▷ if less than $k$ results
```

$next_\tau(t)$ described in the Preliminaries. Both searches are integrated into a single algorithm. For the analysis, we assume the INN algorithm is range-optimal.

6.1 Query-and-iterate

The first algorithm we explore uses an INN to reduce the $|Q(G)|$ distance evaluations computed by query-and-select. It is most useful when the number of answers to the query, $|Q(G)|$, is large. As before, we first compute $Q(G)$, but instead of probing each result, we repeatedly use $next_\tau(t)$ to find the next neighbor of t. Each such neighbor is looked up in $Q(G)$ until k of them are found. Lookup can be implemented, for example, with a hash table for tuples $(\overline{a}, b) \in Q(G)$, with b acting as the hash key. This method is described in Algorithm 2.

Let d be the distance to the k-th nearest neighbor of t that occurs in $Q(G)$, we define $R(d) = \{b \mid \delta(t, \tau(b)) \leq d\}$ as the set of elements in the vector store up to the farthest element to t we answer. A range-optimal INN computes the required distances (for the specific index) to find $R(d)$; we call this number $D(d) \leq |Q(G)|$. Let us assume its CPU cost is within $\tilde{O}(D(d)) \subseteq \tilde{O}(Q^*)$, which is mostly the case. This includes the hash table search time.[7] Then, we immediately obtain:

Proposition 1. *the complexity of Algorithm 2 is* $\langle \tilde{O}(Q^*), D(d) \rangle$

This is a considerable reduction in the number of distance evaluations w.r.t. the query-and-select algorithm.

6.2 Iterate-and-query

Our second proposal aims to reduce CPU time by working the other way around: we begin by iterating over $next_\tau(t)$ to retrieve the next-nearest neighbor of t. For

[7] To obtain worst-case search times, we can use instead binary search on a sorted array, whose construction and searches are still within $\tilde{O}(|Q(G)|) \subseteq \tilde{O}(Q^*)$.

Algorithm 3 Approximate k-GSS via Iterate-and-query

```
Require: Graph $G$, distance $d$ and indexed vector store $\tau$, $next_\tau()$
Require: Query $(Q, x)$, vector $t$ and integer $k$
Ensure: Top $k$ answers $(\bar{a}, b) \in Q(G)$, ordered by $\delta(t, \tau(b))$
  OAns ← $\emptyset$
  for each next element $b = next_\tau(t)$ do
      Ans($b$) ← $Q_{x=b}(G)$                    ▷ using some wco algorithm
      Add Ans($b$) to OAns
      if |OAns| $\geq k$ then
          Return OAns capped to the first $k$ tuples
      end if
  end for
  Return OAns                                   ▷ if less than $k$ results
```

each such neighbor b, we run a restricted query $Q_{x=b}$, where the variable x is replaced with the constant b. This method is detailed in Algorithm 3. It is most useful when the query Q is complex and produces many results.

Let us define $Q_{x \in R(d)}(G) \subseteq Q(G)$ as the set of answers in $Q(G)$ where x is restricted to be in $R(d)$. We show:

Proposition 2. *The time complexity of Algorithm 3 is* $\langle \tilde{O}(Q^*_{x \in R(d)}), D(d) \rangle$

Proof. The worst-case size of the output satisfies $Q^*_{x \in R(d)} \leq Q^*$. A wco algorithm that handles the query $Q' = Q \ltimes U$, where U is a unary relation with the attribute x and the rows $R(d)$, works in time $\tilde{O}(Q^*_{x \in R(d)})$. We do not proceed in this way but instead invoke the wco algorithm with $Q_{x=b}$ for each $b \in R(d)$. By Friedgut's inequality [36] it holds that: $Q^*_{x \in R(d)} \geq \sum_{b \in R(d)} Q^*_{x=b}$. The time complexity of this algorithm is then $\langle \tilde{O}(Q^*_{x \in R(d)}), D(d) \rangle$.

Although worst-case time complexities always favor iterate-and-query over query-and-iterate, actual wco algorithms may require more work to compute $Q_{x=b}(G)$ for many values of b than to compute the whole $Q(G)$. Which way works better in practice depends on this relation. A tradeoff can be obtained by evaluating in batches: for each batch $B = b_1, \ldots b_\ell$ of consecutive near neighbors returned by $next_\tau(t)$, we run the query $Q_{x \in B}$, by treating B as a unary relation associated with variable $x \in \mathsf{vars}(Q)$. The worst-case analysis stays as for the non-batched version, except that we may extract up to $R(d) + \ell$ elements from the INN (in exchange for packing the candidates into fewer queries).

7 Implementation

Algorithm 1 can be implemented on top of any existing graph database system that has a vector store by defining a new operator allowing the computation of distances to a fixed vector. In contrast, implementing Algorithms 2 and 3 requires the coordination of two components: a graph database management system with support for vectors and an INN. Practical applications such as RAG impose considerable strain on the query throughput, so we resort to ANN indices, which are faster (though less precise) and widely available in popular vector databases. We use MillenniumDB [35] for both components, a multimodal graph database supporting RDF/SPARQL which extends RDF with a vector datatype allowing to store vectors as literals. In MillenniumDB we store vectors as literals in triples, using a special datatype to denote it is a vector. Regarding queries, the engine uses Leapfrog Trie-Join (LTJ) [47] as the wco algorithm to compute the answers of graph patterns, and HNSW [33] to implement the ANN index. Naturally, these two components work independently, and thus, another implementation may use other approaches for query processing or for vector retrieval. All storage and retrieval of data is achieved through the standard disk-buffer architecture, meaning that the database is fully persistent.

Query-and-Select. Implementation of Algorithm 1 in MillenniumDB uses LTJ. As results are produced, we compute the distance to the specified vector t and keep a priority queue of size k, storing the most similar results seen so far. After evaluating Q, the queue contains the final output as described in Section 5.

Query-and-Iterate. For Algorithm 2, the dictionary H used to probe for query tuples is built using extendable hashing [21] (which is buffered to disk as needed). LTJ populates this table as it produces results. We also build an iterator on top of MillenniumDB's ANN index by extending the classical top-k querying for HNSW [33] with the ability to iterate over all the vectors in decreasing order of similarity to a fixed vector. As we iterate through the elements most similar to our query vector t, we probe H until the k most similar results are found.

Iterate-and-Query. To implement Algorithm 3, we modify the variable ordering of Leapfrog Trie-Join so that it starts with variable x (which is the one subject to similarity search). When instantiating this variable, we iterate over all possible assignments for it using the ANN index instead of using the database indices in MillenniumDB. Our implementation explicitly alters the variable ordering of LTJ, which is known to be important for performance [37]. As we assume $R(d)$ to be much smaller than the number of nodes in the graph, it should be the case that starting with variable x leads to a better performance.

Query Language. MillenniumDB fully supports SPARQL. We designed and implemented a SPARQL extension in MillenniumDB to add similarity search clauses. Our extension offers support for the following four new capabilities:

1. *Vector datatype.* As already mentioned, we implement a vector datatype that can be used in RDF triples. A sample syntactic specification of a vector of length 3 is `"[1,2,3]"^^type:tensorFloat`.
2. *Distance functions.* We implement special functions to calculate similarity between vectors using common distance functions. For example, the function call `fn:manhattanDistance(?vector, "[1,2,3,4]"^^type:tensorFloat)` computes the Manhattan distance between the vector stored in `?vector` and the vector `[1,2,3,4]`. We currently support several standard distance functions, such as the Manhattan and Euclidean metrics.

3. *HNSW iterators.* To support iterative algorithms, we also extend the HNSW index with the ability to return all the vectors in the vector store sorted by (approximate) distance from a fixed vector. For this, we implement the procedure: `proc:hnswIterator("index" vector ?distance)`. This procedure returns *graph nodes* in decreasing order of distance to `vector` in the HNSW index called `index` and binds this distance to `?distance`. In contrast to traditional HNSW iterators [33], which only return top-k nodes, to support our algorithms, we need to potentially iterate over all the nodes in the graph.

We now illustrate how to use these capabilities. Let us say we wish to retrieve 100 people most similar to Geoffrey Hinton, who also won a Turing Award. The following query expresses this using the HNSW iterator from item 3 above. Note that the nodes returned by `proc:hnswIterator` are bound to variable `?person`.

```
SELECT ?person ?dist WHERE {
  ?person :won :Turing_Award . :Geoffrey_Hinton :hasVector ?vector.
  ?person proc:hnswIterator("indexName" vector ?dist)
} LIMIT 100
```

Our language also supports the Query-and-Select algorithm, and also the classical HNSW method to retrieve the top-k most similar nodes. For space reasons we do not show the corresponding SPARQL queries, but they will be made avaliable as part of the documentation of our code.

8 Experimental Results

In this section, we see how our algorithms perform against the Query-and-Select baseline. Regarding query time, we expect to confirm the hypothesis that Algorithms 2 and 3 are better suited than the baseline for computing top-k queries for small k. Furthermore, we also expect our variants to thrive on queries with high selectivity, especially when running over limited time constraints. We also believe it is important to study the selectivity thresholds that mandate when to use one algorithm over the others, as this has a direct impact on the development of fully-fledged query engines looking for the best plan to evaluate each query.

In our implementation, we focus on speed by relying on ANN indices. Hence, as we cannot guarantee the retrieval of all the nearest neighbors of a vector, another natural question is how our alternatives compare to the baseline regarding the quality of the answers retrieved by the approximate methods. In summary, in this section, we use our experiments to answer the following questions:

Q1: How much faster can Algorithms 2 and 3 be for solving k-GSS in practice?
Q2: For increasing values of k, when does the baseline become a better algorithm? Regarding the number of answers, what is the selectivity range where our alternatives perform better?
Q3: How does the ANN version of Query-and-Iterate and Iterate-and-Query compare to the baseline in terms of the quality of the answers?

8.1 Data

For our experiments, we used the combined RDF graphs of Wikidata and IMGpedia. IMGpedia is an MKG that provides embeddings for 15 million images from Wikimedia Commons and also links each image to its related entities in Wikidata [20]. We use a truthy dump of Wikidata containing 617,065,092 triples.

Regarding queries, we mine them from a Wikidata query log [34,9]. We select all queries with nonempty results that involve an image (e.g., by using predicate `wdt:P18`) that has an available embedding in IMGpedia. We extract only the BGPs in these queries. Each of these real-world BGPs is transformed into a similarity pattern by selecting a random variable in the BGP that we know (from Wikidata's ontology) must be mapped to an image, selecting a random node from the answer of this BGP, and constructing a similarity clause for this variable and the vector of this node. We produced 200 queries.

In terms of similarity, we sampled a million random pairs of vectors from the HOG vectors in IMGpedia (192 dimensions). We obtain an estimated mean Manhattan (L_1) distance of 42.6, and a standard deviation of 8.1. This gives us an intrinsic dimensionality of 13.8, which is at the higher end of the tractable range, further justifying the choice of ANNs for our implementation.

8.2 Setup

We use the implementation described in Section 7 (see [35]). The experiments were run on a commodity server (Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz and 118GB of RAM, Devuan Chimaera 4.0). MillenniumDB used 48GB of RAM for buffering and single-threaded execution. OS caches were cleared between the algorithm versions, but remained hot within the runs. This means that each batch of queries is run in succession for a specific algorithm and similarity restriction, but caches are cleaned when switching versions. Queries have a timeout of 1 minute; only results found within that time are reported. The HNSW index [33] was built with $m = 48$ edges and *efConstruction*= 256.

8.3 Query Evaluation Time

Fig. 4 shows the average execution time over all queries in our benchmark. As expected, the smaller the value of k, the better our algorithms perform. On average, Query-and-Iterate shows only a slight advantage for top-1 queries, while Iterate-and-Query shows an advantage for up to top-50 queries. This already gives a direct answer to **Q1**, as Iterate-and-Query seems an obvious alternative to use in cases where $k < 50$. However, the running time of our algorithms depends heavily on the selectivity of these queries, so we look into this.

From our analysis on selectivity, an important shortcoming of the Query-and-Select baseline is that it requires computing the distances between all the $|Q(G)|$ answers of the pattern and the input vector, whereas both Query-and-Iterate and Iterate-and-Query require an optimal (when using INNs) number of computations, which we call $R(d)$. Hence, the larger the number of answers,

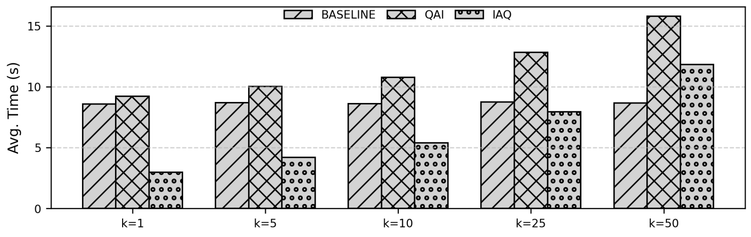


Fig. 4: Running time of Query-and-Select (baseline), Query-and-Iterate (QAI) and Iterate-and-Query (IAQ). Average overall queries in the benchmark.

the more unnecessary computations are made by the baseline. Further, we also expect Iterate-and-Query to use far less processing time (compared to the baseline) when $R(d)$ is smaller than $|Q(G)|$: this is because we are specifically starting LTJ with the variable subject to similarity. This is a good idea if $R(d)$ is small compared to the selectivity of other variables, but it can be a poor choice if other variables provide much stricter selectivity. This fact does not show up in the wco bounds, and hence, it was not predicted by the theory.

To verify these claims experimentally, we partition the 200 queries in the benchmark into deciles w.r.t. the number of answers of the underlying graph pattern in each query and compute the proportion of time taken by Iterate-and-Query and Query-and-Select, as well as Query-and-Iterate and the baseline. Results in Table 1 show a striking difference in the running time of algorithms depending on the selectivity of the graph patterns. For queries with low selectivity (i.e., **D8**–**D10**), Iterate-and-Query can be up to 50x faster than the baseline. On the other hand, this is reversed in the higher-selectivity deciles, with fewer answers ($< 4,000$). In these cases, it is conceivable that the database is better off computing these 4,000 answers and then ordering them by similarity than traversing the index. This analysis also explains why Query-and-Iterate exhibits the worst performance on average: in **D8**–**D10**, where these algorithms shine the most, Query-and-Iterate still has the overhead of processing hundreds of thousands of tuples into a hash join, which slows down the process. With this experiment, we answer **Q2**: for the settings in our server, the selectivity threshold of queries for using Iterate-and-Query is over 248,000 expected answers.

One important observation from Table 1 is that query selectivity in our dataset is highly skewed, with 7 deciles having fewer than 4000 answers. Remarkably, even though 70% of queries are extremely selective, the running time of Iterate-and-Query (IAQ) remains competitive up to $k = 25$. This highlights the inefficiency of Query-and-Select under low selectivity. Assuming an oracle that predicts the decile of each query, combining Query-and-Select and IAQ would reduce the average running time for $k = 25$ to around 2.5 seconds, a 66% improvement over IAQ alone, currently the best option (see Figure 4).

On average, the baseline takes 0.1s for queries in **D1**–**D7** and up to 130 minutes for queries in **D8**–**D10**. IAQ is slightly slower on smaller queries (0.8s on **D1**–**D7**), but remains fast as size grows (8s on **D8**–**D10**).

Table 1: Proportion of median time taken by Query-and-Select (baseline), Query-and-Iterate (QAI), and Iterate-and-Query (IAQ), grouped over deciles of 20 queries, sorted by number of answers. **MNA** is the median number of answers to the queries in each decile.

		Time: IAQ/baseline					**Time: QAI/baseline**				
D.	**MNA**	$k = 1$	$k = 5$	$k = 10$	$k = 25$	$k = 50$	$k = 1$	$k = 5$	$k = 10$	$k = 25$	$k = 50$
1	459	16.31	132.38	302.82	861.22	1438.98	19.22	115.97	254.32	761.39	1219.95
2	793	6.84	23.19	37.82	58.12	130.14	7.13	20.33	33.45	49.12	106.28
3	976	5.90	17.48	35.34	107.01	173.68	6.78	12.22	15.16	36.86	71.53
4	1,303	9.05	19.55	25.51	80.80	216.00	9.79	18.62	22.85	70.08	178.57
5	1,606	26.62	47.63	84.57	174.20	248.96	24.05	38.22	68.28	141.47	195.50
6	2,095	45.99	64.58	80.71	189.45	333.63	42.45	55.39	69.84	164.54	293.63
7	3,186	17.10	44.34	79.40	145.64	291.32	16.98	39.58	73.97	124.71	242.92
8	248,004	**0.79**	1.08	1.67	2.76	4.59	1.01	1.48	2.70	3.38	3.21
9	2,572,812	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**	**0.91**	**0.91**	**0.91**	**0.87**	1.08
10	7,625,570	**0.03**	**0.03**	**0.03**	**0.03**	**0.03**	1.20	1.17	1.21	1.19	1.19

8.4 Quality of the Approximated Methods

In this section, we answer **Q3** by measuring the distance between the answers retrieved by our (ANN-based) implementation of Algorithms 2 and 3 and the ground truth baseline. We remark that these results mostly depend on the choice of ANN index: the more precise the index, the better the quality of the answers. Nevertheless, the results we obtain for the HNSW index in MillenniumDB reveal a trend that should be common to any implementation that uses similar indices [6].

To evaluate the quality of the answers of our algorithms, we measure the percentage of correct k-NN objects found relative to the baseline. In this context, this percentage reflects both precision and recall. Fig. 5 presents the precision distribution per algorithm and per value of k. QAI has a mean average precision (MAP) of 27.5%, while IAQ reaches 28.5%. As shown in the figure, precision increases with larger values of k, which is expected since, for smaller k, a single wrong answer has a larger impact. The figure also shows a high variance in precision, averaging 31%. These results reflect the inherent tradeoff between query evaluation time and precision [6]. We leave improving the precision metrics for future work, possibly with parameter tuning or by incorporating more sophisticated ANN indices. Moreover, for cases where precision is most important, one can always resort to deterministic non-approximated INN indices.

9 Concluding Remarks

In this paper, we formalize and study efficient algorithms for the k-GSS problem. Our theoretical analysis makes a case for Iterate-and-Query as the algorithm of choice, especially when k is small and the number of answers to patterns is large. We confirm this fact through an experiment using IMGpedia [20], an MKG that

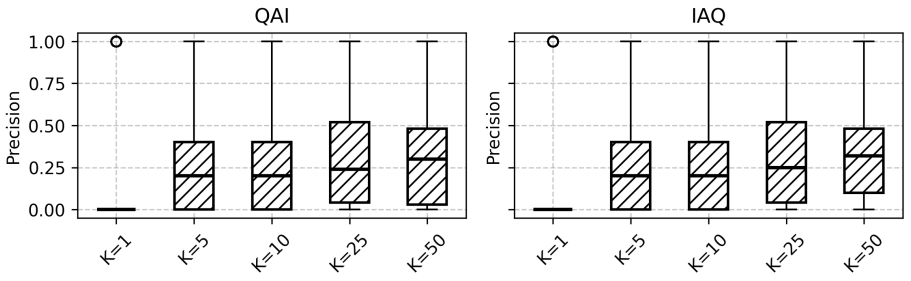


Fig. 5: Precision of Query-and-Iterate (QAI) and Iterate-and-Query (IAQ).

incorporates image embeddings into Wikidata. We believe our paper will pave the way for implementing k-GSS in graph databases, especially in those that currently feature vector stores, such as Neo4j or Amazon Neptune.

Improved query algorithms. So far, we have considered plug-and-play combinations between LTJ and an INN algorithm, but more involved arrangements may yield improved results. For example, we can enhance the LTJ algorithm so that the similarity condition is incorporated at the time of variable binding. This can be done in the style of Query-and-Select (i.e., generate all the values of a variable x that survive Leapfrog's intersection, sort them by distance to t, and then bind the values $x = b$ in that order), Query-and-Iterate (generate the values for x as before, then use the INN index to produce the neighbors of t sorted by distance, and bind the values $x = b$ that are found in the intersection), or Iterate-and-Query (generate the consecutive neighbors with the INN, and check each one to see if it survives the intersection of LTJ).

More similarity conditions. Both algorithms and the query language can be extended to support more similarity conditions. Assume we have a graph G, a distance δ, and a vector store τ. For query patterns, we can now fix several of its variables, which are to be compared with several individual vectors. To see how answers are sorted, suppose we have such a pattern query $(Q, x_1, \ldots, x_n)$, vectors $t_1, \ldots t_n$, and an aggregation function $F : \mathbb{R}^n \to \mathbb{R}$. Then each answer $(\bar{a}, b_1, \ldots, b_n)$ is ordered according to the value of $F(b_1, \ldots, b_n)$. For the query-and-select baseline, we compute F at the time answers are retrieved and keep the set of best-k answers seen so far in a priority queue. For the other algorithms, we replace the $next_\tau(t)$ iterator with a top-1 variant of the Threshold Algorithm [18] using $next_\tau(t)$ to separately explore candidates in each dimension of F.

Supplemental Material Statement: All sources and scripts available at [35].

Acknowledgments. This work was funded by ANID – Millennium Science Initiative Program – Code ICN17_002. S. Ferrada was supported by Fondecyt Grant 11251322; by CENIA FB210017, Basal ANID; and by the Vicerrectoría de Investigación y Desarrollo (VID) of U. de Chile, project code UI-016/24. G. Navarro was supported by Fondecyt Grant 1-230755. J. L. Reutter and D. Vrgoč were supported by Fondecyt Grant 1221799. D. Vrgoč was also supported by Fondecyt Grant 1240346.

References

1. Amer, A.A., Abdalla, H.I., Nguyen, L.: Enhancing recommendation systems performance using highly-effective similarity measures. Knowledge-Based Systems **217**, 106842 (Apr 2021). `https://doi.org/10.1016/j.knosys.2021.106842`
2. Angles, R., Calisto, V., Díaz, J., Ferrada, S., Hogan, A., Pinto, A., Reutter, J., Rojas, C., Rosales-Méndez, H., Sarmiento, H., Toussaint, E., Vrgoč, D.: TelarKG: A Knowledge Graph of Chile's Constitutional Process. In: Proceedings of the 7th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). pp. 1–5. ACM, Santiago AA Chile (Jun 2024). `https://doi.org/10.1145/3661304.3661899`
3. Arroyuelo, D., Bustos, B., Gómez-Brandón, A., Hogan, A., Navarro, G., Reutter, J.: Worst-case-optimal similarity joins on graph databases. Proceedings of the ACM on Management of Data **2**(1), 1–26 (2024)
4. Arya, S., Mount, D., Netanyahu, N., Silverman, R., Wu, A.: An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In: Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA'94). pp. 573–583 (1994)
5. Atserias, A., Grohe, M., Marx, D.: Size bounds and query plans for relational joins. SIAM Journal on Computing **42**(4), 1737–1767 (2013)
6. Aumüller, M., Bernhardsson, E., Faithfull, A.: ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. Information Systems **87**, 101374 (Jan 2020). `https://doi.org/10.1016/j.is.2019.02.006`
7. Baeza-Yates, R., Ribeiro-Neto, B., et al.: Modern information retrieval, vol. 463. ACM press New York (1999)
8. Bawa, M., Condie, T., Ganesan, P.: LSH forest: Self-tuning indexes for similarity search. In: Proc. 14th International Conference on World Wide Web (WWW). p. 651. ACM Press (2005). `https://doi.org/10.1145/1060745.1060840`
9. Bonifati, A., Martens, W., Timm, T.: Navigating the maze of wikidata query logs. In: The World Wide Web Conference. pp. 127–138 (2019)
10. Chávez, E., Figueroa, K., Navarro, G.: Effective proximity retrieval by ordering permutations. IEEE Transactions on Pattern Analysis and Machine Intelligence **30**(9), 1647–1658 (2008)
11. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquin, J.L.: Searching in metric spaces. ACM Computing Surveys **33**(3), 273–321 (2001)
12. Ciaccia, P., Patella, M.: PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In: Proc. 16th International Conference on Data Engineering (ICDE). pp. 244–255 (2000)
13. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proc. 23rd International Conference on Very Large Data Bases (VLDB). pp. 426–435 (1997)
14. Clarkson, K.L.: Nearest neighbor queries in metric spaces. Discrete Computational Geometry **22**(1), 63–93 (1999)
15. Core, C.: Chroma - the open-source embedding database. `https://github.com/chroma-core/chroma` (2024)
16. Cyganiak, R., Wood, D., Lanthaler, M., Klyne, G., Carroll, J.J., McBride, B.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, W3C (Feb 2014)
17. Discussion on the pgvector project: `https://github.com/pgvector/pgvector/issues/259`
18. Duong, Q.H., Liao, B., Fournier-Viger, P., Dam, T.L.: An efficient algorithm for mining the top-k high utility itemsets, using novel threshold raising and pruning strategies. Knowledge-Based Systems **104**, 106–122 (2016)

19. Engels, J., Landrum, B., Yu, S., Dhulipala, L., Shun, J.: Approximate nearest neighbor search with window filters. arXiv preprint arXiv:2402.00943 (2024)
20. Ferrada, S., Bustos, B., Hogan, A.: IMGpedia: A Linked Dataset with Content-Based Analysis of Wikimedia Images. In: The Semantic Web – ISWC 2017, vol. 10588, pp. 84–93. Springer International Publishing, Cham (2017). `https://doi.org/10.1007/978-3-319-68204-4_8`
21. Garcia-Molina, H., Ullman, J., Widom, J.: Database systems: the complete book. Pearson Education India (2008)
22. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK. pp. 518–529. Morgan Kaufmann (1999), `http://www.vldb.org/conf/1999/P49.pdf`
23. Gollapudi, S., Karia, N., Sivashankar, V., Krishnaswamy, R., Begwani, N., Raz, S., Lin, Y., Zhang, Y., Mahapatro, N., Srinivasan, P., et al.: Filtered-diskann: Graph algorithms for approximate nearest neighbor search with filters. In: Proceedings of the ACM Web Conference 2023. pp. 3406–3416 (2023)
24. Guo, Q., Zhuang, F., Qin, C., Zhu, H., Xie, X., Xiong, H., He, Q.: A Survey on Knowledge Graph-Based Recommender Systems. IEEE Transactions on Knowledge and Data Engineering **34**(8), 3549–3568 (Aug 2022). `https://doi.org/10.1109/TKDE.2020.3028705`
25. Guo, R., Luan, X., Xiang, L., Yan, X., Yi, X., Luo, J., Cheng, Q., Xu, W., Luo, J., Liu, F., et al.: Manu: a cloud native vector database management system. arXiv preprint arXiv:2206.13843 (2022)
26. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data - SIGMOD '84. p. 47. ACM Press, Boston, Massachusetts (1984). `https://doi.org/10.1145/602259.602266`
27. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language (Mar 2013), `https://www.w3.org/TR/sparql11-query`
28. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Transactions on Database Systems **24**(2), 265–318 (1999)
29. Hjaltason, G.R., Samet, H.: Incremental similarity search in multimedia databases. Tech. Rep. CS-TR-4199, University of Maryland, Computer Science Department (2000)
30. Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutiérrez, C., Kirrane, S., Labra Gayo, J.E., Navigli, R., Neumaier, S., Ngonga Ngomo, A.C., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J.F., Staab, S., Zimmermann, A.: Knowledge Graphs. No. 22 in Synthesis Lectures on Data, Semantics, and Knowledge, Springer (2021). `https://doi.org/10.2200/S01125ED1V01Y202109DSK022`, `https://kgbook.org/`
31. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: Proc. 18th International Semantic Web Conference (ISWC). pp. 258–275 (2019)
32. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive nlp tasks. In: Proc. 34th International Conference on Neural Information Processing Systems. Curran Associates Inc., Red Hook, NY, USA (2020)
33. Malkov, Y.A., Yashunin, D.A.: 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)

34. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In: The Semantic Web–ISWC 2018: 17th International Semantic Web Conference, Monterey, CA, USA, October 8–12, 2018, Proceedings, Part II 17. pp. 376–394. Springer (2018)
35. Millennium Institute for Foundational Research on Data: MillenniumDB. The link to the repository can be found at: `https://anonymous.4open.science/r/WCO-SimilaritySearch-2025`
36. Ngo, H.Q.: Worst-case optimal join algorithms: Techniques, results, and open problems. In: Proc. 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS). pp. 111–124 (2018)
37. Nguyen, D., Aref, M., Bravenboer, M., Kollias, G., Ngo, H.Q., Ré, C., Rudra, A.: Join processing for graph patterns: An old dog with new tricks. In: Proc. 3rd International Workshop on Graph Data Management Experiences and Systems (GRADES). pp. 2:1–2:8 (2015)
38. Pan, J.J., Wang, J., Li, G.: Survey of vector database management systems. The VLDB Journal **33**(5), 1591–1615 (2024)
39. Pan, S., Luo, L., Wang, Y., Chen, C., Wang, J., Wu, X.: Unifying Large Language Models and Knowledge Graphs: A Roadmap. IEEE Transactions on Knowledge and Data Engineering **36**(7), 3580–3599 (Jul 2024)
40. Paraschakis, D., Ros, R., Borg, M., Runeson, P.: Fuserank (demo): Filtered vector search in multimodal structured data. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. pp. 404–408. Springer (2024)
41. Pelillo, M., Hancock, E.R. (eds.): Similarity-Based Pattern Recognition, Lecture Notes in Computer Science, vol. 7005. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). `https://doi.org/10.1007/978-3-642-24471-1`
42. PostgreSQL Global Development Group: pgvector - open-source vector similarity search for postgres. `https://github.com/pgvector/pgvector` (2024)
43. Prokhorenkova, L., Shekhovtsov, A.: Graph-based nearest neighbor search: From practice to theory. In: Proc. 37th International Conference on Machine Learning, (ICML). Proceedings of Machine Learning Research, vol. 119, pp. 7803–7813 (2020)
44. Reimers, N., Gurevych, I.: Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks (2019). `https://doi.org/10.48550/ARXIV.1908.10084`
45. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann (2006)
46. Schneider, P., Schopf, T., Vladika, J., Galkin, M., Simperl, E., Matthes, F.: A decade of knowledge graphs in natural language processing: A survey. In: Proc. 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing. vol. 1, pp. 601–614 (2022)
47. Veldhuizen, T.L.: Triejoin: A simple, worst-case optimal join algorithm. In: Proc. 17th International Conference on Database Theory (ICDT). pp. 96–106 (2014)
48. Vrandečić, D., Krötzsch, M.: Wikidata: A free collaborative knowledgebase. Communications of the ACM **57**(10), 78–85 (Sep 2014)
49. Vrgoč, D., Rojas, C., Angles, R., Arenas, M., Arroyuelo, D., Buil-Aranda, C., Hogan, A., Navarro, G., Riveros, C., Romero, J.: MillenniumDB: An Open-Source Graph Database System. Data Intelligence **5**(3), 560–610 (Aug 2023). `https://doi.org/10.1162/dint_a_00229`, `https://direct.mit.edu/dint/article/5/3/560/117375/MillenniumDB-An-Open-Source-Graph-Database-System`

50. Vrgoč, D., Rojas, C., Angles, R., Arenas, M., Calisto, V., Farías, B., Ferrada, S., Heuer, T., Hogan, A., Navarro, G., Pinto, A., Reutter, J., Rosales, H., Toussiant, E.: Millenniumdb: A multi-modal, multi-model graph database. In: Companion of the 2024 International Conference on Management of Data. p. 496–499. SIGMOD '24, Association for Computing Machinery, New York, NY, USA (2024). `https://doi.org/10.1145/3626246.3654757`, `https://doi.org/10.1145/3626246.3654757`
51. Wang, J., Yi, X., Guo, R., Jin, H., Xu, P., Li, S., Wang, X., Guo, X., Li, C., Xu, X., et al.: Milvus: A purpose-built vector data management system. In: Proceedings of the 2021 International Conference on Management of Data. pp. 2614–2627 (2021)
52. Wang, M., Wang, H., Qi, G., Zheng, Q.: Richpedia: A Large-Scale, Comprehensive Multi-Modal Knowledge Graph. Big Data Research **22**, article 100159 (Dec 2020)
53. Wang, X., Meng, B., Chen, H., Meng, Y., Lv, K., Zhu, W.: TIVA-KG: A Multimodal Knowledge Graph with Text, Image, Video and Audio. In: Proc. 31st ACM International Conference on Multimedia. pp. 2391–2399 (2023)
54. Yianilos, P.: Locally lifting the curse of dimensionality for nearest neighbor search. In: Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 361–370 (2000)
55. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search - The Metric Space Approach, Advances in Database Systems, vol. 32. Kluwer (2006)
56. Zheng, W., Zou, L., Peng, W., Yan, X., Song, S., Zhao, D.: Semantic SPARQL similarity search over RDF knowledge graphs. Proceedings of the VLDB Endowment **9**(11), 840–851 (Jul 2016). `https://doi.org/10.14778/2983200.2983201`