# Worst-Case-Optimal Similarity Joins on Graph Databases

Diego Arroyuelo
DCC, Escuela de Ingeniería, Pontificia Universidad Católica & IMFD
Santiago, Chile
diego.arroyuelo@uc.cl

Benjamin Bustos
Department of Computer Science, University of Chile & IMFD
Santiago, Chile
bebustos@dcc.uchile.cl

Adrián Gómez-Brandón
Universidade da Coruña & CITIC & IMFD
A Coruña, Spain
adrian.gbrandon@udc.es

Aidan Hogan
Department of Computer Science, University of Chile & IMFD
Santiago, Chile
ahogan@dcc.uchile.cl

Gonzalo Navarro
Department of Computer Science, University of Chile & IMFD
Santiago, Chile
gnavarro@dcc.uchile.cl

Juan Reutter
Department of Computer Science, PUC & IMFD
Santiago, Chile
jreutter@ing.puc.cl

## ABSTRACT

We extend the concept of worst-case optimal equijoins in graph databases to the case where some nodes are required to be within the $k$-nearest neighbors ($k$-NN) of others under some similarity function. We model the problem by superimposing the database graph with the $k$-NN graph and show that a variant of Leapfrog TrieJoin (LTJ) implemented over a compact data structure called the Ring can be seamlessly extended to integrate similarity clauses with the equijoins in the LTJ query process, retaining worst-case optimality in many relevant cases. Our experiments on a benchmark that combines Wikidata and IMGpedia show that our enhanced LTJ algorithm outperforms by a considerable margin a baseline that first applies classic LTJ and then completes the query by applying the similarity predicates. The difference is more pronounced on queries where the similarity clauses are more densely connected to the query, becoming of an order of magnitude in some cases.

## CCS CONCEPTS

• **Theory of computation → Database query processing and optimization (theory)**; **Data structures and algorithms for data management**.

## KEYWORDS

Worst-case optimal joins; Leapfrog Triejoin; graph patterns; graph databases; graph indexing; similarity joins; nearest-neighbor graphs

**ACM Reference Format:**
Diego Arroyuelo, Benjamin Bustos, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, and Juan Reutter. 2018. Worst-Case-Optimal Similarity Joins on Graph Databases. In *Proceedings of ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD '24).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Graph databases [5] are enjoying a resurgence, seen in the emergence of novel graph query languages [19] and new commercial graph database systems [8, 11]. Open knowledge graphs, such as Wikidata [54], receive millions of SPARQL queries per day [38]. These developments call for (1) time- and space-efficient techniques to evaluate queries over graph databases [40], and (2) new features that enhance the expressivity of graph query languages [4].

Regarding efficiency, *worst-case optimal* (*wco*) *join* algorithms [44] have provided notable reductions in runtimes for evaluating complex graph patterns compared to traditional methods [3, 6, 7, 25, 33, 34, 39, 45]. While such algorithms typically require extra index permutations, and thus more space when compared to, e.g., pairwise joins, recent works address this limitation through on-the-fly indexing [24], and compact data structures [6, 7].

Regarding expressivity, wco algorithms have mainly been studied in the context of evaluating *equijoins*. Their suitability for other types of joins is not yet well understood. Wco guarantees have been proven for *relaxed joins* [44], whereby not all of the tuples of the join query need to be satisfied. Such guarantees have also been studied for top-$k$ queries [50], where only the top $k$ results in some ordering are returned. Though interesting variants, both relaxed and top-$k$ queries are still based on equijoins. Similar guarantees have recently been studied in the context of theta-joins [51], which allow inequalities, non-equalities, etc., in join conditions. It remains of interest to study wco guarantees for other types of join.

Our goal in this paper is to push the envelope for wco join algorithms by studying their applicability for *similarity joins*, which relax equijoins by matching elements of the database that are "similar" (according to some predefined criteria), rather than precisely equal. Two variants of similarity joins are commonly considered: *range-based similarity joins* (or $\epsilon$-joins) match pairs of elements within a certain distance, while *$k$–nearest neighbor joins* (or $k$-NN joins) match, for each element in the left relation, its $k$ most similar elements in the right relation [48]. Such joins have been widely studied since the mid 1990's [56], having been folklore for longer, with works citing applications for multimedia databases [56], time-series databases [56], DNA databases [56], spatial databases [31], text mining [15], clustering [31], record linkage [13, 15], and more besides. Supporting similarity joins could then open up a wide range of such applications for graph databases [23].

*Contributions:* Our specific contributions are as follows: (i) we present a compact data structure and novel algorithms for evaluating $k$-NN similarity joins over graph databases; (ii) we prove the worst-case optimality of these techniques under certain conditions of (a)cyclicity; (iii) we create a pseudo-real-world benchmark for this task that combines graphs with multimedia (image) content; (iv) we show that our algorithm clearly outperforms a baseline using wco join algorithms that postpone similarity joins until last.

*Motivating examples.* Graph patterns, the most typical queries on graph databases, look for partially instantiated subgraphs in the labeled graph. For instance, Twitter searches for diamond patterns in order to make recommendations about whom to follow [29, 39]: if $(x, \mathsf{Follows}, y)$ denotes a graph edge $x \xrightarrow{\mathsf{Follows}} y$, then

$$(x,\mathsf{Follows},y),\ (x,\mathsf{Follows},z),\ (y,\mathsf{Follows},z),\ (y,\mathsf{Follows},w),\ (z,\mathsf{Follows},w),$$

might indicate that user $w$ is a good recommendation for user $x$ to follow, based on the topology of its social network. However, one would also want to take advantage of certain similarities among users in order to improve suggestions. For instance, in the previous example it might be the case that user $y$ does not necessarily follow user $z$, but they are similar in some sense (for example, they have similar interests, they produce similar posts about certain topics, or they live in the same region or country). The same happens with users $z$ and $w$. Hence, one could issue a query like

$$(x, \mathsf{Follows}, y), (x, \mathsf{Follows}, z), y \sim z, (y, \mathsf{Follows}, w), z \sim w,$$

where $\sim$ denotes similarity among the involved variables. This would allow us recommend $x$ to follow $w$ not only based on the graph topology, but also considering certain similarities. Indeed, based on the same query we might also want to suggest $y$ to follow $z$ (and vice versa), and even $z$ to follow $w$.

Beyond this example, combining similarity joins with graph patterns can be useful in diverse domains, for instance, to find: (1) stadia of German football clubs whose *geographically closest* stadium is of a team in the same league; (2) *similar messages* posted by bot accounts and politicians they follow on a social network; (3) *visually similar* works by Henri Matisse and works by a Cubist compatriot. Combining criteria via multiple similarity joins, we can find, for example: (4) pairs of songs with *similar tonality and lyrics* by Asian artists, or (5) countries *similar in terms of population and area* that are neighbors. Using similarity joins on distinct entities we can find, for example, (6) pairs of stars and their orbiting exoplanets with *similar mass*, resp., to a solar planet and our sun; (7) researchers working on *similar topics* at *geographically close* institutes. Our goal is to evaluate such queries efficiently in time and space.

*Related work.* Kiefer et al. [35] propose iSPARQL, which adds an `IMPRECISE` clause to SPARQL that allows for specifying a similarity join. Ferrada et al. [23] extend SPARQL with a syntax, semantics and set of rewriting rules for similarity joins. Other works extend graph databases with domain-specific similarity joins in the context of query relaxation [32], record deduplication [26, 47], multimedia databases [22], and geographic databases [10, 58], among others.

In terms of novelty, little work has been done on optimizing similarity joins within graph patterns [23] (or indeed, in the relational setting [48]), and no work that we are aware of has looked at wco guarantees for join queries with similarity clauses.

## 2 LEAPFROG TRIEJOIN AND THE RING

### 2.1 Graph databases and BGP matching

We introduce key concepts and notation needed for this paper.

Definition 1. *Let $U$ be a universe of constants. A* graph database *is a labeled graph $G(V, E)$, where $V \subseteq U$ is a finite set of nodes and $E \subseteq V \times U \times V$ is a finite set of labeled edges; $(u, p, v) \in E$ denotes $u \overset{p}{\to} v$. We call $\mathrm{dom}(G) = \{u, p, v \mid (u, p, v) \in G\}$ the subset of $U$ used as constants in $G$ and $D = |\mathrm{dom}(G)|$. Furthermore, we call $n = |V|$ the number of nodes in $G$ and $N = |E|$ the number of edges.*

To simplify, we assume $U = [1..D]$; note $n \le D \le 3N$, $N \le D^3$.

A graph database $G$ is often used to search for patterns of interest, that is, subgraphs of $G$ that are homomorphic to a basic graph pattern $Q$. We define a basic graph pattern formally as follows.

Definition 2. *Let $G(V, E)$ be a graph database, $U$ be its universe of constants, and $W$ be a universe of variables disjoint from $U$. A basic graph pattern (BGP) $Q$ is a set of triple patterns $(x, y, z)$, where $x, y, z \in U \cup W$. The output $Q(G)$ of the BGP is the set of all assignments $A : W_Q \to U$, where $W_Q \subseteq W$ are the variables that appear in $Q$, such that for each triple pattern $(x, y, z) \in Q$, it holds that $(A'(x), A'(y), A'(z)) \in G$, where $A'(x) = x$ for all $x \in U$ and $A'(x) = A(x)$ for all $x \in W_Q$.*

Given a BGP $Q$ over a graph database $G$, the task is to enumerate $Q(G)$. A BGP $Q$ is equivalent to a join query, as follows. Each triple pattern in $Q$ is an atomic query over $G$, equivalent to equality-based selections on a single ternary relation. Then, a BGP corresponds to a conjunctive query (i.e., a *join query* plus simple selections) over the relational representation of the graph.

The *AGM bound* [9] establishes the maximum output size of a join query free of self joins. This bound can also be applied to BGPs, which feature self joins, constants in $U$, and multiple occurrences of a variable in a triple pattern. The idea is to regard each triple pattern as a relation formed by the triples matching its constants [33]. Thus, the AGM bound of $Q$ over a graph database instance $G$, denoted $Q^*$, is the maximum size $Q(G')$ could have over any database instance $G'$ of size $|G'| \le |G|$, where $|\cdot|$ denotes the number of edges of a graph. A join algorithm is *worst-case optimal (wco)* if it has a running time in $\tilde{O}(Q^*)$, where $\tilde{O}$ ignores polylogs and data-independent factors. Atserias et al. [9] proved that for queries as simple as $\{(x, p, y), (y, p, z), (z, p, x)\}$ (for some constant $p$), no classical plan involving only pair-wise joins can be wco.

### 2.2 Leapfrog Triejoin (LTJ)

Leapfrog Triejoin [53] (`LTJ`, for short) is a worst-case optimal algorithm for computing natural joins in relational databases that has been adapted for evaluating BGPs [33] as described next. Assume that the graph database has been stored using a trie (or digital tree) data structure, such that for each edge $(u, p, v) \in E$ there is a path of length 3 in the trie storing the values $u$, $p$, and $v$, respectively. In the RDF notation, these values are called s (*subject*), p (*predicate*), and o (*object*), respectively. So, the above is called the spo trie, as tuples are stored following that order. Indeed, for `LTJ` to work properly, one needs to store $3! = 6$ different tries, corresponding to the 6 different permutations of the values s, p, and o. Now, let us consider a BGP $Q = \{t_1, \ldots, t_q\}$ whose set of variables is $\{x_1, \ldots, x_v\}$. `LTJ` uses a so-called *variable elimination* approach, carrying out $v$ iterations, each handling a particular variable. This implies defining a total order $\langle x_{i_1}, \ldots, x_{i_v} \rangle$ in which the variables will be processed.

Each triple pattern $t_i$ has an associated trie $\tau_i$ whose edge values have been stored in a manner consistent with the given variable ordering. LTJ starts at the root of every $\tau_i$ and descends by the children that correspond to the constants in $t_i$. It then proceeds to the variable elimination phase. Let $Q_j \subseteq Q$ be the triple patterns that contain variable $x_{i_j}$. Starting with the first variable in the order, $x_{i_1}$, LTJ finds each $c_1 \in \mathrm{dom}(G)$ such that for every $t \in Q_1$, if $x_{i_1}$ is replaced by $c_1$ in $t$, the evaluation of the modified triple pattern $t$ over $G$ is non-empty (i.e., there may be answers to $Q$ where $x_{i_1}$ is equal to $c_1$). To find such a $c_1$, we must intersect the children of the current nodes in all the tries $\tau_i$, for $t_i \in Q_1$. During execution, we keep a mapping $\mu$ that binds variables already processed. As we find each constant $c_1$ suitable for $x_{i_1}$, we set $\mu = \{(x_1 := c_1)\}$ and branch on this value $c_1$, going down by $c_1$ in all the tries $\tau_i$, for $t_i \in Q_1$. We now repeat the same process with $Q_2$, finding suitable constants $c_2$ for $x_{i_2}$ and extending the mapping to $\mu = \{(x_1 := c_1), (x_2 := c_2)\}$, and so on. Once we have eliminated all variables, $\mu$ is a solution for $Q$ (solutions can be found on each branch for distinct values of $c_1, \ldots, c_v$). If for some variable $x_{i_j}$ there is no value $c_j$ in the intersection, the algorithm backtracks and continues with the next value for $Q_{j-1}$. When the process finishes, the algorithm has reported all the solutions for $Q$.

LTJ carries out the intersection at the trie nodes using the primitive `leap`$(\tau_i, c)$, which finds the next smallest constant $c_i \geq c$ within the children of the current node in trie $\tau_i$; if there is no such value $c_i$, `leap`$(\tau_i, c)$ returns a special value $\perp$. Veldhuizen [53] showed that LTJ is wco if `leap()` runs in polylogarithmic time.

## 2.3 Fundamental operations on strings

Let $B$ be a bit vector of length $|B|$. On it we define the following operations, for $b \in \{0,1\}$: (1) $\mathrm{rank}_b(B, i)$, with $1 \leq i \leq |B|$, counts the number of bits with value $b$ in $B[1..i]$, and (2) $\mathrm{select}_b(B, j)$, with $1 \leq j \leq \mathrm{rank}_b(B, |B|)$, yields the position in $B$ of the $j$th bit with value $b$ from the left. These operations, as well as accessing $B[i]$, can be supported in $O(1)$ time using $|B| + o(|B|)$ bits of space [14, 41]. They can also be extended to a string $S[1\,..\,N]$ over an alphabet $\Sigma = [0, D)$, as $\mathrm{rank}_c(S, i)$ and $\mathrm{select}_c(S, j)$, for $c \in \Sigma$. *Wavelet trees* (WT, for short) are the paradigmatic data structure supporting these operations efficiently, specifically in $O(\log D)$ time and using $N \log D + o(N \log D)$ bits of space [28] (we use logarithms in base 2). WTs efficiently support an extended set of operations [42], including: (1) `range_next_value`$(S, r_b, r_e, c)$, which finds, for $c \in \Sigma$, the smallest symbol $c' \geq c$ that occurs in range $S[r_b\,..\,r_e]$, in $O(\log D)$ time; and (2) `range_symbols`$(S, r_b, r_e)$, which counts the number of different values in $S[r_b\,..\,r_e]$ in $O(\log N)$ time and using $O(N \log D)$ additional bits of space.

## 2.4 The Ring data structure

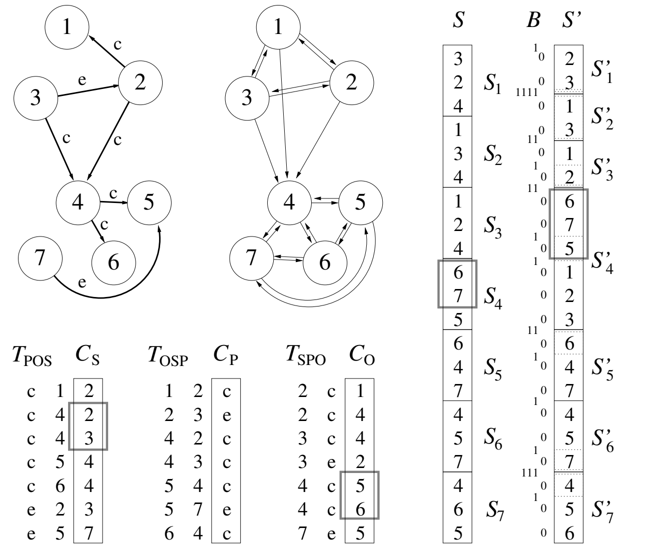
The `Ring` [6] data structure supports the six orders needed by LTJ using a single bidirectional cyclical index that uses space close to the raw data representation (and even less in some cases), while supporting the `leap()` operation needed by the algorithm.

The data structure is essentially a column store, built as follows for a graph $G(V, E)$. Let $T_{\text{SPO}}[1\,..\,N][1\,..\,3]$ be the table storing the $N = |E|$ edges $(u, p, v)$ of the graph, sorted according to the SPO order. Let $C_{\text{O}}$ denote the last column of $T_{\text{SPO}}$, which intuitively corresponds to the last level (i.e., the one corresponding to O) of the trie for SPO. Next, the process moves the third column to the front in the table, making it the first column. The table is then re-sorted to obtain table $T_{\text{OSP}}$. Let $C_{\text{P}}$ denote the last column of this table, which corresponds to the last level of the trie for the order OSP. Finally, the third column in table $T_{\text{OSP}}$ is moved to the front and the table is re-sorted once again, obtaining table $T_{\text{POS}}$ and column $C_{\text{S}}$. The `Ring` index is then formed by the sequences $C_{\text{S}}$, $C_{\text{O}}$, and $C_{\text{P}}$, which are represented using wavelet trees, with a total space requirement of $3N \log D + o(N \log D)$ bits. It also contains arrays $A_j$, for each $C_j$ with $j \in \{\text{S}, \text{P}, \text{O}\}$, defined as $A_j[k] = |\{i \in [1\,..\,N],\ C_j[i] < k\}|$, for $k = 1, \ldots, D+1$. These arrays store the cumulative number of occurrences of the symbols of $U$ in $C_j$ and are represented using bit vectors with $3(N + D) + o(N + D)$ bits. The total space is thus close to the $3N \log D$ bits of a plain representation of the graph $G$.

By using $C_j$ and $A_j$, for $j \in \{\text{S}, \text{P}, \text{O}\}$, we can switch between tables using the function $F_j : [1\,..\,N] \rightarrow [1\,..\,N]$, defined as $F_j(i) \equiv A_j[c] + \mathrm{rank}_c(C_j, i)$, where $c = C_j[i]$. So, function $F_{\text{O}}$ maps a position in table $T_{\text{SPO}}$, using $A_{\text{O}}$ and $C_{\text{O}}$, to the corresponding one in $T_{\text{OSP}}$. Similarly, $F_{\text{P}}$ maps from $T_{\text{OSP}}$ to $T_{\text{POS}}$ and $F_{\text{S}}$ maps from $T_{\text{POS}}$ back to $T_{\text{SPO}}$. The mappings work in $O(\log D)$ time if we compute $\mathrm{rank}_c$ using the WT. We can also move in the other direction in $O(\log D)$ time using the inverse function of $F_j$: $F_j^{-1}(i') \equiv \mathrm{select}_c(C_j, i' - A_j[c])$, where $c$ is such that $A_j[c] < i' \leq A_j[c+1]$.

Every node $v$ in any of the 6 tries corresponds to a range $C_j[b\,..\,e]$ in some of the three columns. Consider, for example, the trie $T_{\text{SPO}}$, whose leaves are enumerated in column $C_{\text{O}}$. If $v$ is the root, then its range is the whole $C_{\text{O}}[1\,..\,N]$. If $v$ is in the first level and corresponds to the subject $\text{S} = x$, then its range $C_{\text{O}}[b\,..\,e]$ is that of all triples starting with $x$, $[b\,..\,e] = [A_{\text{S}}[x] + 1\,..\,A_{\text{S}}[x+1]]$. If $v$ is in the second level and corresponds to $(\text{S}, \text{P}) = (x, y)$, then $C_{\text{O}}[b\,..\,e]$ is the range of the triples in $T_{\text{SPO}}$ starting with $xy$. A leaf node denoting the triple $(\text{S}, \text{P}, \text{O}) = (x, y, z)$ corresponds to the single position in $C_{\text{O}}$ where $T_{\text{SPO}}$ contains $xyz$. The same holds, analogously, for the tries $T_{\text{OSP}}$ and $T_{\text{POS}}$. The other tries can also be simulated with ranges. Consider $T_{\text{SOP}}$, for example. A first-level node $v$ by $\text{S} = x$ corresponds to the same range in $C_{\text{O}}$ as before, but a second-level node corresponding to $(\text{S}, \text{O}) = (x, z)$ is equivalent to $(\text{O}, \text{S}) = (z, x)$, which is a node in $T_{\text{OSP}}$, and thus to a range in $C_{\text{P}}$. Then, if we descend from $\text{S} = x$ to $(\text{S}, \text{P}) = (x, y)$, we restrict the range in $C_{\text{O}}$, but if we descend to $(\text{S}, \text{O}) = (x, z)$, we switch to a range in $C_{\text{P}}$. The new column ranges are computed with extensions of the functions $F_j$ and $F_j^{-1}$, all in $O(\log D)$ time; see the original paper [6] for details.

EXAMPLE 1. *Consider the graph on the top-left of Figure 1, where the labels indicate (c)heap or (e)xpensive travel routes from the source to the target node. The columns $C_{\text{S}}$, $C_{\text{P}}$, and $C_{\text{O}}$ are shown on the bottom-left; ignore the rest of the figure for now. The BGP $Q = \{(x, \text{c}, y), (y, \text{c}, z)\}$ looks for places $(y, z)$ we can reach from $x$ at low cost with at most one stop. Both triples $(x, \text{c}, y)$ and $(y, \text{c}, z)$ have the initial range $C_{\text{S}}[1\,..\,5]$, corresponding to $\text{P} = \text{c}$. Say we first eliminate variable $y$. For $(y, \text{c}, z)$, the candidate subjects $\{2, 3, 4\}$ are the distinct elements in $C_{\text{S}}[1\,..\,5]$, whereas for $(x, \text{c}, y)$, the candidate objects*



| $T_{\text{POS}}$ | | $C_S$ | $T_{\text{OSP}}$ | | $C_P$ | $T_{\text{SPO}}$ | | $C_O$ |
|---|---|---|---|---|---|---|---|---|
| c | 1 | 2 | 1 | 2 | c | 2 | c | 1 |
| c | 4 | 2 | 2 | 3 | e | 2 | c | 4 |
| c | 4 | 3 | 4 | 2 | c | 3 | c | 4 |
| c | 5 | 4 | 4 | 3 | c | 3 | e | 2 |
| c | 6 | 4 | 5 | 4 | c | 4 | c | 5 |
| e | 2 | 3 | 5 | 7 | e | 4 | c | 6 |
| e | 5 | 7 | 6 | 4 | c | 7 | e | 5 |

**Figure 1: A graph $G$ and its $K$-NN graph with $K = 3$ at its right. Below, the `Ring` index of $G$. On the right, the representation of the $K$-NN graph using $S$, $S'$, and $B$. The dotted lines mark the internal divisions in each $S'_x$ for the different values of $t$. The grayed ranges correspond to Examples 1 and 3.**

*$\{1, 4, 5, 6\}$ are the distinct elements in $C_O$ that are mapped to $C_S[1..5]$ by $F_S^{-1}$. The `Ring` efficiently finds the intersection $\{4\}$. We then bind $y := 4$. The new range associated with $(y, c, z) = (4, c, z)$ is $C_O[5..6]$ – corresponding to $(s, p) = (4, c)$ in $T_{\text{SPO}}$ – whereas the one associated with $(x, c, y) = (x, c, 4)$ is $C_S[2..3]$ – corresponding to $(p, o) = (c, 4)$ in $T_{\text{POS}}$. Those ranges are outlined. We continue later in Example 3.*

In the `Ring`-supported LTJ algorithm, then, each triple pattern of $Q$ is associated with some range $C_j[b..e]$, and the intersections of trie nodes correspond, intuitively, to finding the common values in the ranges of all the triple patterns that share the next variable to bind. Operation `leap()`, which powers the intersection, is supported in $O(\log D)$ time by using, in particular, the WT operation `range_next_value`, which finds the smallest $c_x \geq c$ within a range $C_j[b..e]$. The LTJ intersection algorithm works in time bounded by the size of the smallest intersected range multiplied by the number of intersected ranges and by the cost of `leap()`, which yields worst-case optimality. For compatibility, we state their results using $\log N = \Theta(\log D)$ instead of $\log D$.

Theorem 1 ([6]). *Let $G$ be a graph database with $N$ edges and $Q$ be a BGP. Then, a representation using $3N + o(N)$ words of space can compute $Q(G)$ in $O(Q^*|Q|\log N)$ time, where $Q^*$ is the maximum possible output (AGM bound) of $Q$ on some graph of $N$ edges.*

## 3 LTJ WITH SIMILARITY JOINS

### 3.1 Modeling similarity

Based on the observation that users struggle with distance-based similarity given that distances – particularly in high-dimensional abstract spaces – can be difficult to conceptualize [23, 57], we model similarity via $k$-nearest neighbors, which allows us to abstract away details of particular distance functions.

Definition 3. *Let $V$ be a set and $d : V \times V \longrightarrow \mathbb{R}^+$ be a distance function on $V$. For $u, v \in V$ and any integer $1 \leq k < |V|$, we define $k$-NN$(u)$ as a set satisfying $u \notin k$-NN$(u)$, $|k$-NN$(u)| = k$, and $\forall v \in k$-NN$(u)$, $\forall v' \notin \{u\} \cup k$-NN$(u)$, $d(u, v) \leq d(u, v')$. That is, $k$-NN$(u)$ is the set of the $k$ elements closest to $u$, with ties broken arbitrarily.*

In order to have a consistent definition of $k$-NN for all values of $k$ in presence of ties, we define the concept of a $K$-NN graph.

Definition 4. *Given an integer $1 \leq K < |V|$, a $K$-NN graph of $V$ is a directed graph whose node set is $V$ and the out-neighbors of every $u \in V$ is a set $K$-NN$(u)$. We additionally regard $K$-NN$(u)$ as ordered by nondecreasing distance to $u$, so we say that $K$-NN$(u)[j]$ is the $j$th closest element to $u$ and, for any $1 \leq k \leq K$, we say that $v \in k$-NN$(u)$ iff $v = K$-NN$(u)[j]$ for some $1 \leq j \leq k$.*

We consider the $K$-NN graph as part of the input, just like the graph $G$. Except for simple cases like, say, geographic distance, the distance function is given by an expert on the data domain. In some cases, the expert directly gives the $K$-NN graph, as it might be easier to rank by closeness than to come up with a similarity function [49]. When a similarity function is given, the $K$-NN graph is computed only once, at index construction time, not for each query. The naïve approach for building the $K$-NN graph takes time $\Theta(n^2)$, by computing all pairwise distances between nodes in $V$. Paredes et al. [46] present methods for building $K$-NN graphs for general metric spaces, taking empirical times of $O(n^{1.27})$ for low-dimensional spaces and $O(n^{1.90})$ for high-dimensional ones, using $O(n(K + \log n))$ space. In the case of $\mathbb{R}^d$, Vaidya [52] proposes an algorithm for $K = 1$ that takes $O((cd)^d n \log n)$ time, for a constant $c$ [18]. Dickerson and Eppstein [20] compute the $K$-NN graph in $\mathbb{R}^d$ in $O(Kn + n \log n)$ time, although they leave open the dependence of the bound on $d$. There are several algorithms for computing approximated $K$-NN graphs, such as NN-Descent [21] (arbitrary similarity measure, empirical time $O(n^{1.14})$), multiple random divide & conquer and neighborhood propagation [55] (data in $\mathbb{R}^d$, time $O(dn \log n)$), and a method based on Locality Sensitive Hashing [59] (data in $\mathbb{R}^d$, time $O(\ell(d + \log n)n)$ with $\ell$ a parameter).

We will enrich the classic BGPs of graph databases by assuming a given $K$-NN graph on the same nodes of the graph database and permitting, in addition to the triple patterns of the BGPs, zero or more expressions of the form $x \lessdot_k y$, where $x$ and $y$ can be constants or variables and $1 \leq k \leq K$ is an integer. The expression $x \lessdot_k y$ means that $y$ is among the $k$ closest elements to $x$, that is, $y \in k$-NN$(x)$. Let us define our extended BGPs and their worst-case optimality.

Definition 5. *Let $G(V, E)$ be a graph database, $U$ be its universe of constants, and $W$ be a universe of variables disjoint from $U$. An extended BGP $Q$ is a set of triple patterns $(x, y, z)$, where $x, y, z \in U \cup W$, and a set of clauses $x \lessdot_k y$, where $x, y \in U \cup W$, $x \neq y$, and $k \geq 1$ is an integer. The output $Q(G)$ of the extended BGP is the set of all assignments $A : W_Q \longrightarrow U$, where $W_Q \subseteq W$ are the variables that appear in $Q$, such that (1) for each triple pattern $(x, y, z) \in Q$, it holds that $(A'(x), A'(y), A'(z)) \in G$, and (2) for each clause $x \lessdot_k y \in Q$, it holds that $A'(y) \in k$-NN$(A'(x))$, where $A'(x) = x$ for all $x \in U$ and $A'(x) = A(x)$ for all $x \in W_Q$.*

Definition 6. *Given a graph database $G$ with $N$ triples and a $K$-NN graph for a metric distance $d$ on its nodes $V$, an algorithm to compute $Q(G)$ for an extended BGP $Q$ is* worst-case optimal (wco) *if its time complexity is $\tilde{O}(Q^*)$, where $Q^*$ is the maximum size of $Q(G')$ on any graph $G'$ with $N' \leq N$ triples and the $K$-NN graph of any metric $d'$ on the nodes $V'$ of $G'$.*

Though we use $x \lhd_k y$ as our similarity primitive, its asymmetric nature may be unintuitive for final users. We thus build upon it the following more intuitive symmetric operator:

$$x \sim_k y \iff x \lhd_k y \wedge y \lhd_k x,$$

that is, $x$ is among the $k$ nearest neighbors of $y$ and vice versa. We simply convert any clause $x \sim_k y$ per its definition in order to handle it in the extended BGPs we just defined.

We will assume for simplicity that all the nodes $V$ participate in the similarity. Our techniques can handle $K$-NN graphs defined on a subset of $V$ only, assuming the $k$-NN predicates involving other nodes are always false. We could also have fewer than $K$ neighbors for some nodes of $V$, for example to disregard neighbors that are too far away in terms of the distance $d$. Furthermore, we could have various independent $K$-NN relations and refer to them in the same queries. Our techniques can work with any $k$-NN relation, without requiring that it corresponds to some distance $d$; therefore they are useful to model similarity functions that are non-metric, for example. Our optimality proofs are valid even in the more restrictive case of metric distances (the more restrictive the relation, the harder to be wco because not every input table is possible).

We can also extend our results to distance-based similarity joins, that is, indicating that two elements $x$ and $y$ must be within distance $d$ to each other. We return to this point at the end of Section 3.3.

## 3.2 A basic idea

In principle, any indexing scheme solving BGPs in wco time can be extended to handle the similarity clauses $x \lhd_k y$. At index construction time, the value $K$ is chosen and a suitable representation of the $K$-NN graph is built. At query time, for every clause $x \lhd_k y$, we *materialize* the relation $k\mathrm{NN}(\cdot,\cdot)$ containing all the pairs $(a,b)$ such that $b \in k\text{-NN}(a)$, and replace the clause $x \lhd_k y$ by the expression $k\mathrm{NN}(x,y)$. We then run the wco algorithm on the modified query.

*Materializing.* A first problem is how to efficiently materialize $k\mathrm{NN}(\cdot,\cdot)$ from the $K$-NN graph, because the value $k$ used at query time can be much smaller than $K$ (additionally, each clause may use a different $k$ value). For the discussion, let us regard the $K$-NN graph as a table of triples $(u,v,j)$, meaning that $v = K\text{-NN}(u)[j]$. To materialize $k\mathrm{NN}(\cdot,\cdot)$ we must extract all the triples $(u,v,j)$ for $1 \leq j \leq k$, and then sort them by $(u,v)$ and by $(v,u)$ to build the two LTJ tries representing $k\mathrm{NN}(\cdot,\cdot)$. The most efficient way is to maintain the $K$-NN graph sorted by $j$, so the extraction takes $O(kn)$ time and the sorting for trie construction takes time $O(kn \log n)$.

While this time is proportional to the input size and thus within wco bounds in theory, the approach is totally impractical because in most useful cases the output is much smaller than the input. As an example, in our experimental setup of Section 6, just copying the part of the $K$-NN graph and sorting it twice, for $k = 50$, takes 260 seconds, only after which the actual query processing starts. Instead, our proposed index handles the complete query process in as little as 1.3 seconds for the fastest queries we consider, or as much as 103 seconds for the most expensive ones.

We show in Section 3.3 how our data structures manage to *simulate* the desired tables $k\mathrm{NN}(\cdot,\cdot)$, without ever materializing them, by representing the $K$-NN graph in a specific way, using WTs. Additionally, we seamlessly extend the `Ring` to emulate the tries of these simulated tables without building them at query time.

*Optimality.* Using LTJ is not wco in this case, because the relations $k\mathrm{NN}(\cdot,\cdot)$ satisfy what is known as a *degree constraint*: there are only $k$ tuples in the relation sharing their same first component. When the $k$-NN constraints follow an ayclicity property one can retain worst-case optimality by choosing an order that respects such acyclicity [1, 43]. In the general case, optimality can be obtained by using PANDA [2], an algorithm that is optimal for the setting considered in this paper. This is, again, impractical, however. While the PANDA algorithm may work with theoretical guarantees in data complexity, the authors themselves note that this involves huge query-dependent factors, and that it is important to find algorithms that work faster in practice [2]. The approach we develop in Section 4 is to construct a good variable ordering for LTJ in a greedy fashion, taking advantage of the fact that the `Ring` can retrieve the number of instantiations for each triple pattern before processing them. We show that our approach remains worst-case optimal (just as PANDA) for a relevant class of extended BGPs that includes queries with cyclic $k$-NN constraints. Our experiments in Section 6 show that our variable ordering strategy can improve query computation time in practice, even for queries where our strategy is not necessarily worst-case optimal.

## 3.3 Our solution

In order to extend the LTJ algorithm to handle the similarity constraints $x \lhd_k y$, we merge the classic `Ring` representation of the database graph (Section 2.4) with a representation of the $K$-NN graph. We choose $K$ at index construction time, and then can handle queries $x \lhd_k y$ for any $1 \leq k \leq K$. Importantly, depending on the elimination order of the variables in LTJ, it may be the case that we need to compute the relation $x \lhd_k y$ in a backwards fashion: instead of looking for the $k$ nearest neighbors of a node $u$, we may have to look for those nodes of which $u$ is a $k$-nearest neighbor. Thus, our $K$-NN graph representation consists of two sequences, $S[1..Kn]$ and $S'[1..Kn]$, plus a bitvector $B[1..2nK]$, which record the $K$-NN graph and its transpose. For simplicity, let us identify the graph nodes $V$ with the integers in $[1..n]$.

Definition 7. *For any $u \in [1..n]$, let $S_u[1..K]$ be such that $S_u[j] = K\text{-NN}(u)[j]$. We then define $S[1..Kn] = S_1 \cdot S_2 \cdots S_n$, thus $S_u[j] = S[(u-1)K + j]$.*

Definition 8. *For any $u \in [1..n]$, let $S'_u$ be the sequence of elements $v$ such that $S_v[j_v] = u$ for some $1 \leq j_v \leq K$, sorted by increasing value of $j_v$ with ties broken arbitrarily. We then define $S'[1..Kn] = S'_1 \cdot S'_2 \cdots S'_n$. To distinguish the different values of $j_v$ in $S'$, let $S'_u$ contain $s_t$ elements $v$ with $j_v = t$, then we define bitvector $B_u = 10^{s_1}10^{s_2}\cdots 10^{s_K}$ and $B[1..2nK] = B_1 \cdot B_2 \cdots B_n$. Then $S'[i]$ corresponds to the $i$th $0$ in $B$; note $B$ contains $Kn$ $0$s and $Kn$ $1$s.*

Example 2. *In the middle of Figure 1 we show a 3-NN graph, where each node $u$ points to the nodes in 3-NN$(u)$ using Euclidean* *distance on the plane. For example, the three nearest neighbors of node $u = 1$ are, from closest to farthest, $S_1 = 324$, and those of node $u = 2$ are $S_2 = 134$. These strings of length $K = 3$ are concatenated into the string $S$, shown vertically on the right of the graph. Now consider $S'_4$ on the right of the figure, which contains the nodes $v$ for which $u = 4$ is in 3-NN$(v)$, i.e., such that $u$ appears in $S_v$. We see that $u = 4$ appears in $S_1, S_2, S_3, S_5, S_6$, and $S_7$, at positions $j_1 = 3$, $j_2 = 3$, $j_3 = 3$, $j_5 = 2$, $j_6 = 1$, and $j_7 = 1$, respectively. We write those values of $v$ in $S'_4 = 675123$ by increasing order of $j_v$, that is, from smallest to largest value of $k$. Then, $S'_4[1..2] = 67$ are the nodes $v$ for which $u \in$ 1-NN$(v)$, $S'_4[1..3] = 675$ are those for which $u \in$ 2-NN$(v)$, and $S'_4[1..6]$ are those for which $u \in$ 3-NN$(v)$. These limits inside $S'_4$, at positions 2, 3, and 6, are marked in unary by $B_4 = 100101000$. As another example, we have $S'_1 = 23$ with $B_1 = 10011$ because $1 \in$ 1-NN$(2)$ and $1 \in$ 1-NN$(3)$.*

This arrangement in $S'$ allows us to have a range for the values $x$ such that $x \lhd_k y$ when $y$ is fixed.

Lemma 1. *For any $v$, the values $u$ such that $K$-NN$(u)[t] = v$ are written in $S'$ from position $S'[p_v(t)]$, with $p_v(t) = \mathrm{select}_1(B, (v-1)K + t) - (v-1)K - t + 1$.*

Proof. Let $s_t$ be as in Def. 8; then the first of the desired elements $u$ is at $S'_v[p]$, with $p = s_1 + \cdots + s_{t-1} + 1$. By the definition of $B_v$, it holds that $p = \mathrm{select}_1(B_v, t) - t + 1$. Since $B_v$ starts at the $((v-1)K + 1)$th 1 of $B$ and the 0s of $B$ correspond to the positions in $S'$, $S'_v$ starts at $S'[\mathrm{select}_1(B, (v-1)K + 1) - (v-1)K]$. Therefore, $S'_v[p]$ corresponds to $S'[p_v(t)]$. □

Lemma 2. *The following are equivalent.*

- $(a)$ $v \in k$-NN$(u)$,
- $(b)$ *$v$ is in $S[(u-1)K + 1..(u-1)K + k]$ and*
- $(c)$ *$u$ is in $S'[p_v(1)..p_v(k+1) - 1]$.*

Proof. Conditions $(a)$ and $(b)$ are equivalent by Def. 7. Condition $(a)$ is equivalent to $v$ appearing in $S_u[1..k]$, i.e., $S_u[t] = v$ for some $1 \le t \le k$. By Lemma 1, $u$ appears in $S'[p_v(t)..p_v(t+1) - 1]$. Taking the union of those ranges for $1 \le t \le k$, we have that condition $(a)$ is equivalent to $u$ appearing in $S'[p_v(1)..p_v(k+1)-1]$. □

Consider a clause of the form $x \lhd_k y$ in the query. Our algorithm proceeds exactly as if we had materialized the relation $k$NN$(x, y)$. Therefore, whenever $x$ or $y$ is bound, our LTJ algorithm must simulate the binding of the first or the second component of $k$NN$(\cdot, \cdot)$ to $x$ or $y$, respectively. In LTJ, this would correspond to representing $k$NN$(x, y)$ with two tries, $T_{xy}$ with order $xy$ and $T_{yx}$ with order $yx$, and descending by $T_{xy}$ if $x$ is materialized first and by $T_{yx}$ if $y$ is materialized first. We do not materialize those tries either, however. Just as the Ring represents every node of the tries $T_{\mathrm{SPO}}$, etc. as some range $C_j[b..e]$, we represent the nodes of $T_{xy}$ and $T_{yx}$ as ranges in $S$ or $S'$. Precisely, by Lemma 2, if $x$ is bound first, we simulate descending in $T_{xy}$ by associating the range $S[(x-1)K+1..(x-1)K+k]$ with the clause $x \lhd_k y$. If, instead, $y$ is bound first, we simulate descending in $T_{yx}$ by associating the range $S'[p_y(1)..p_y(k+1) - 1]$ with the clause $x \lhd_k y$. Those ranges will then be included in the corresponding intersections when the variable $y$ (in the first case) or $x$ (in the second case) is bound, exactly as any other column range $C_j[b..e]$ corresponding to triple patterns in $Q$. The WT operation `range_next_value` allows us running intersections on the ranges in $S$ and $S'$ without the need of sorting the values.

Example 3. *Consider again Figure 1, and consider now the extended BGP $Q = \{(x, \mathrm{c}, y), (y, \mathrm{c}, z), y \sim_2 z\}$, which looks for* nearby *places $(y, z)$ we can go consecutively from $x$ at low cost. We start the process as in Example 1, but when we bind $y := 4$, we also descend by $y = 4$ in the tries of $T_{yz}$ and $T_{zy}$, as we have the clause $y \sim_2 z \equiv y \lhd_2 z \land z \lhd_2 y$. This corresponds to associating the range $S_4[1..2] = S[10..11]$ with $4 \lhd_2 z$ and the range $S'_4[1..3] = S'[7..9]$ with $z \lhd_2 4$. These ranges are also outlined in the figure. Say we now eliminate $z$. The* Ring *intersects the ranges $C_{\mathrm{O}}[5..6]$ associated with $(4, \mathrm{c}, z)$, $S[10..11]$ associated with $4 \lhd_2 z$, and $S'[7..9]$ associated with $z \lhd_2 4$. The intersection yields the candidate set $\{6\}$. We then bind $z := 6$, associating $C_{\mathrm{O}}[6]$ with the triple $(4, \mathrm{c}, 6)$, $S[10]$ with $4 \lhd_2 6$, and $S'[7]$ to $6 \lhd_2 4$. We finally eliminate $x$, which has two bindings in $C_{\mathrm{S}}[2..3] = \{2, 3\}$. The solutions are then $(x, y, z) = (2, 4, 6)$ and $(x, y, z) = (3, 4, 6)$. If we used $y \sim_3 z$ we would have also found the solutions $(2, 4, 5)$ and $(3, 4, 5)$.*

In order for those ranges in $S$ and $S'$ to be seamlessly integrated into the LTJ algorithm supported by the Ring, we represent the sequences $S$ and $S'$ using wavelet trees, whereas the bitvector $B$ must be represented supporting constant-time select queries. The total space of $S$, $S'$ and $B$ adds up to $2nK + o(nK)$ words.

*Range-based similarity.* While as discussed in Section 3.1 the $k$-NN model is preferred in many cases, there are others (e.g., geographic distances) where using distance constraints may be more intuitive. Our scheme can be extended to support range-based similarity joins, with clauses of the form $dist(x, y) \le d$, where $d$ is bounded by some maximum distance of interest, $d_{\max}$. To support it, we could store a *distance graph* represented as the WT of a sequence $D$, much like $S'$ in the $K$-NN graph, where for every $u$ we store all the nodes $v$ within distance at most $d_{\max}$ from $u$, in increasing distance order, and a bitvector similar to $B$ to mark the region of every node $u$ in $D$. Whenever $x$ (or $y$) is bound in the clause $dist(x, y) \le d$, we find the range of $x$ (or $y$) in $D$ and binary search the prefix of the nodes at distance at most $d$ from $u$. This range is added to participate in the intersection when later $y$ (or $x$) is bound. If computing the distances $d(u, v)$ takes non-constant time, we could store them in an array parallel to $D$. Overall, this adds $O(\log N)$ extra time per binding of $x$ (or $y$), which does not alter the total complexity. The resulting time is then the same as if we had set clauses $x \lhd_k y$ and $y \lhd_{k'} x$, where $x$ and $y$ have $k$ and $k'$ nodes within distance $d$, respectively. Note that $k$ and $k'$ will be known to the algorithm and could be used to choose the variable elimination order. Since $k$ and $k'$ depend on each binding of $x$ and $y$, however, the analysis is messier; this is why we fix $k$ for simplicity of exposition.

## 4 OPTIMAL VARIABLE ORDERINGS

Previous work on the LTJ algorithm in the graph context shows that the order in which variables are instantiated makes no difference in the worst-case optimality of the algorithm (although it does have an impact in practice [33, 44]). We will show that, on the contrary, the variable elimination order does make a difference in our extended LTJ algorithm due to the degree constraints naturally present in the $k\mathrm{NN}(\cdot,\cdot)$ tables, and thus we face the problem of finding variable orderings that reach worst-case optimality.

Our variable ordering strategy builds from, and extends, previous strategies designed for queries with acyclic degree constraints [1, 43]. It is based on instantiating variables in an adaptive fashion, choosing at each step the variable with the fewest bindings among those that can be chosen in a topological traversal of the graph of query constraints, whenever possible (i.e., avoiding binding $y$ before $x$ in clauses $x \triangleleft_k y$). Our resulting LTJ extension, which can handle any extended BGP, not only inherits the worst-case optimality of the strategies that handle acyclic topologies, but also produces wco strategies on some classes of queries with cyclic constraints.

Compared to PANDA [2], which can deal with arbitrary degree constraints and always achieves worst-case optimality, our extended LTJ strategy provides a simpler way of handling extended BGPs. While our strategy is not optimal for all queries, it is known that the running time of PANDA includes factors of important magnitude that depend on the query. Hence, our extended LTJ strategy can be seen as a lightweight alternative to PANDA, which is asymptotically optimal for a relevant class of queries.

## 4.1 Size bounds for extended BGPs

We can reason about output size bounds for our queries by regarding, again, each clause $x \triangleleft_k y$ as a relation $k\mathrm{NN}(x,y)$. As explained, the degree of this relation is at most $k$. Relations with degree constraints usually restrain the number of tuples in the output.

Example 4. *Consider $Q = (x,R,y),\ (y,S,z),\ x \triangleleft_k z$, which corresponds to the classic triangle query where one of the relations is replaced by a $k$-NN constraint. Let $G$ be a graph with $N$ edges. If we treat the constraint $x \triangleleft_k z$ as a virtual relation $k\mathrm{NN}(x,z)$ and apply the classic AGM result, we obtain the bound $O(N^{3/2})$.*

*This bound, however, is not tight, because $k\mathrm{NN}(x,z)$ satisfies a degree constraint: each constant eliminating $x$ can be connected only to its $k$ nearest neighbors. This reduces the size of the answers, which are now tightly bound by $O(kN)$: there are at most $N$ edges matching $(x,R,y)$, where for each such edge we find the $k$ nearest neighbors from $x \triangleleft_k z$, and thereafter $(y,S,z)$ can (at most) filter results.*

In their seminal paper, Atserias et al. [9] showed that the maximum number of output tuples of an equijoin was always bounded by the result of a linear program depending on the query and the database instance; that bound was further shown to be tight.

For the case of extended BGPs, we can also bound their number of answers by using a specific linear program. Moreover, while the problem of devising tight lower bounds for BGPs with degree constraints is open (see, e.g., [1]), we will show that our program produces an upper bound that is tight (in data complexity) for a wide class of extended BGPs covering many queries one would expect in practice. Our linear program extends work by Ngo [43] with an additional restriction on dependencies following a cyclic constraint. Let us begin with some definitions.

Definition 9. *The* constraint graph *of an extended BGP $Q$ has the variables $W_Q$ as nodes, and one directed edge $x \longrightarrow y$ per constraint $x \triangleleft_k y$ where both $x$ and $y$ are variables. We say that the constraints of $Q$ are* acyclic *iff its constraint graph is acyclic. A constraint is* cyclic *if its edge participates in a cycle in the constraint graph of $Q$.*

To specify our linear program, let us assume first that our queries are *safe*: for every clause $x \triangleleft_k y$ in $Q$ there must be a triple pattern mentioning $x$; we will later explain how to deal with unsafe queries. Let $Q$ be an extended BGP with $M$ triple patterns over a graph with $N$ tuples. We associate two sets of weights with $Q$: a weight $w_i$ for $t_i$, the $i$th triple pattern in $Q$, and a weight $\delta_{xy}$ for each constraint $x \triangleleft_k y$ in $Q$. We write $x \in t_i$ to indicate that variable $x$ appears in $t_i$. The program associated with $Q$ is then:

$$\begin{aligned}\text{minimize } & \sum_{i=1}^{M} w_i \log N + \sum_{x \triangleleft_k y \in Q} \delta_{xy} \log k \\ \text{where } & \sum_{i,\, x\in t_i} w_i + \sum_{z \triangleleft_k x \in Q} \delta_{zx} \ \geq\ 1 \text{ for each variable } x \text{ in } Q, \\ & \sum_{i,\, x\in t_i} w_i \ - \delta_{xy} \ \geq 0 \text{ for each cyclic constraint } x \triangleleft_k y \text{ in } Q.\end{aligned} \tag{1}$$

Let $\rho^*(Q,N)$ denote the optimal solution to this linear program. The value $Q^* = 2^{\rho^*(Q,N)}$ was shown to be a tight upper bound on $|Q(G)|$ when the constraints of $Q$ are acyclic [43]. The following lemma transfers this result into our framework.

Lemma 3 (cf. [43]). *The number of answers to an extended BGP $Q$ whose constraints are acyclic, over any graph with $N$ tuples, is bounded by $Q^*$, and this bound is tight.*

Note that the only restriction we are considering on the tables $k\mathrm{NN}(\cdot,\cdot)$ is their degree constraint, whereas worst-case optimality in Def. 6 refers to valid $K$-NN graphs that correspond to some metric on $V$. The lower bound still holds because there is always a suitable metric $d$ for every desired table $k\mathrm{NN}(\cdot,\cdot)$: On the trivial metric $d(x,x) = 0$ and $d(x,y) = 1$ for all $x \neq y$, where all the nearest-neighbor comparisons are ties, every table $k\mathrm{NN}(\cdot,\cdot)$ is valid.

The program may overestimate the number of answers when the constraint graph of $Q$ has cycles, however. While there are better bounds for queries with general degree constraints [1, 2], our program provides a simpler bound that can nevertheless be shown to be tight in several practical cases. We further show desirable properties of this program for queries with small cycles, and empirically show that it leads to efficient practical algorithms.

Abo Khamis et al. assume in their analysis that queries are safe. To deal with unsafe queries, we add a predicate $\mathrm{Dom}(x)$ for each constraint $x \triangleleft_k y$, which is instantiated as the *domain* of the graph. Adding weights $s_{xy}$ for each unsafe constraint, the program is then:

$$\begin{aligned}\text{minimize } & \sum_{i=1}^{M} w_i \log N + \sum_{x \triangleleft_k y \in Q} (\delta_{xy} \log k + s_{xy} \log D) \\ \text{where } & \sum_{i,\, x\in t_i} w_i + \sum_{z \triangleleft_k x \in Q} \delta_{zx} + \sum_{x \triangleleft_k y \in Q} s_{xy} \ \geq 1; \text{var } x \text{ in } Q, \\ & \left( \sum_{i,\, x\in t_i} w_i + \sum_{x \triangleleft_k y \in Q} s_{xy} \right) - \delta_{xy} \ \geq 0;\ x \triangleleft_k y \text{ cyclic.}\end{aligned} \tag{2}$$

Note that this program is equivalent to to the program (1) when queries are safe and $N \leq D$: in this case there is always an optimal solution where all weights $s_{xy}$ are set to 0.

## 4.2 Queries with acyclic constraint graphs

Even in the case of acyclic constraint graphs, variable orderings are important for the evaluation of queries. For instance, when running LTJ to evaluate the query of Example 4 over graphs with $N$ edges, the order $y, z, x$ requires up to $N^{3/2}$ eliminations of variables, whereas the order $y, x, z$ requires only $kN$, because the constraint $x \triangleleft_k z$ restricts to only $k$ bindings for $z$ for each binding of $x$.

Previous work on processing queries with degree constraints has shown that wco time can be obtained on acyclic queries by instantiating variables according to the topological ordering of the query (i.e., always instantiating $x$ before $y$ if there are at most $k$ bindings of $y$ per value of $x$). Since our data representation (Section 3.3) allows us to intersect $k$-NN relations while using `leap()` in LTJ, we can use the same variable ordering strategy (i.e., respecting the order of the edges of the constraint graph) to achieve worst case optimality. Such a topological order on the constraint graph can be computed in $O(|Q|)$ time. We can now state our result about optimality for queries with acyclic constraints.

Theorem 2. *Let $G$ be a graph database with $n$ nodes and $N$ edges and $K$ be an integer. Then, a representation using $3N+2nK+o(N+nK)$ words of space can compute $Q(G)$ for extended BGPs $Q$ with acyclic constraints of the form $x \triangleleft_k y$, with $1 \leq k \leq K$, in $O(Q^*|Q|\log N)$ time, where $Q^*$ is the solution to the program (2).*

Before proving the theorem we develop some additional notation; we will use it again in the following section when we extend Theorem 2 for queries with cycles.

Definition 10. *For an unbound variable $x$, let $Q_x$ be the set of (partially instantiated) triples $t \in Q$ such that $x \in t$, and let $t(x)$ be the set of distinct values in the database to which $x$ can be instantiated in triple $t \in Q_x$ (i.e., the answers to $t$ as if every other variable were existentially quantified). Further let $\ell_x = \min_{t \in Q_x} |t(x)|$.*

We are ready for the proof.

Proof of Theorem 2. We simulate LTJ with the Ring, as explained in Section 3.3, from where the space follows. We use a variable ordering that is compatible with a traversal of the constraint graph of $Q$ in topological order.

Let $G$ be a graph of $N$ edges over a domain of size $D$, $Q$ a query, and $\{w_i\}_{i=1}^{M}$, $\{\delta_{xy}\}_{x \triangleleft_k y \in Q}$, $\{s_{xy}\}_{x \triangleleft_k y \in Q}$ be a (not necessarily optimal) solution to the linear program (2). Letting $|t|$ denote the number of triples matching $t$, we show that the algorithm runs in time bounded by

$$|Q|\,\log N \;\cdot\; \prod_{i=1}^{M} |t_i|^{w_i} \prod_{x \triangleleft_k y \in Q} k^{\delta_{xy}} \prod_{x \triangleleft_k y \in Q} D^{s_{xy}}. \qquad (3)$$

The proof is by induction on the number of steps performed by the algorithm, binding one variable $x$ at a time.

For the base case, $Q$ has a single variable $x$. The algorithm computes the intersection of the sets $t(x)$ for every triple $t \in Q_x$, and also of the sets $k$-NN$(a)$ for every clause $a \triangleleft_k x$, with constant $a$, and the sets $\{b \mid x \in k\text{-NN}(b)\}$ for every clause $x \triangleleft_k b$ with constant $b$. Using the Ring, we can intersect all these sets in time bounded by the size of the smallest set we intersect, times a $|Q|\log N$ factor (recall Section 2.4). Given that those sets are always bounded by the minimum between $\ell_x$ and $D$, the size of $k$-NN$(a)$ is bounded by $k$, and the size of $\{b \mid x \in k\text{-NN}(b)\}$ is bounded by $D$ (actually, by $n \leq D$), we have that the time is bounded by $\min(\ell_x, k, D)$, or $\min(\ell_x, D)$ if no clauses of the form $a \triangleleft_k x$ exist in $Q$. We have assumed weights $w_i$, $\delta_{xy}$ and $s_{xy}$ are an admissible solution, so they verify the constraint $\sum_{t_i \in Q_x} w_i + \sum_{a \triangleleft_k x \in Q} \delta_{ax} + \sum_{x \triangleleft_k b \in Q} s_{xb} \geq 1$ by Eq. (2). This means that

$$\min(\ell_x, k, D) \;\leq\; \prod_{t_i \in Q_x} |t_i|^{w_i} \prod_{a \triangleleft_k x \in Q} k^{\delta_{ax}} \prod_{x \triangleleft_k b \in Q} D^{s_{xb}},$$

because $\ell_x$ is smaller than each $|t_i|$ and the minimum of a set of reals is bounded by their geometric mean.

The inductive case follows from the proof of Ngo [43, Thm. 5.1], adapted to our base case. □

## 4.3 Queries with constraint cycles

We now consider the general case, where the constraint graph of $Q$ can have cycles. Knowing how to operate optimally when the constraint graph is acyclic, we break the cycles in the graph of $Q$ adaptively, following a topological ordering of the strongly connected components, and from each component, binding the variable that yields the minimum number of candidates. We start with some definitions.

Definition 11. *Given nodes $x$ and $y$ in a directed graph $C$, we say that $x \prec_C y$ if there is no path from $y$ to $x$ in $C$. Furthermore, we say that node $x$ is $C$-*minimal *if $x \prec_C z$ for every other node $z$ in $C$.*

We then proceed adaptively as follows, where $C$ is the constraint graph of the *current* query $Q$, that is, the query $Q$ with all the variables already bound replaced by their corresponding constraints:

(1) If there are $C$-minimal variables, choose the $C$-minimal variable $x$ with minimum value $\ell_x$.
(2) Otherwise, choose the variable $x$ with minimum value $\ell_x$.

Note that, in the second case, we are forced to bind some $x$ before $z$ in a constraint $z \triangleleft_k x$. In either case, once we bind $x$, new variables may become $C$-minimal because the edges in the constraint graph consider only constraints where both variables are (yet) unbound.

In order to enable such a strategy, we need to be able to compute the quantity $\ell_x$ for every candidate variable $x$. We can do this with the Ring, as it can retrieve any $|t(x)|$ in $O(\log N)$ time using the operation `range_symbols` on the range corresponding to $t(x)$; recall Section 2.3. It can also compute the cardinalities of $k$NN$(a, x)$ or $k$NN$(x, a)$ in constant time because they are the sizes of the corresponding ranges in $S$ or $S'$. Every time a variable $x$ is bound along the adaptive algorithm, the LTJ algorithm recomputes the range of each tuple $t_x \in Q_x$ in time $O(\log N)$, so the cost of recomputing $|t(x)|$ and updating $\ell_x$ is subsumed in the current cost of LTJ. The space of the Ring, in this case, grows but is still $O(N)$.

In section 6 we show that this strategy can lead to fast query resolution in practice, even though it is not necessarily wco: when several constraints form a cycle it could be the case that the variable $x$ minimizing $\ell_x$ leads to more total instantiations than other orderings. However, we can show that the running time of this algorithm is still bounded by our program in the particular case where there is a single "maximal" cycle of length two. This case is especially interesting because the symmetric clauses $x \sim_k y$, with variables $x$ and $y$, form cycles of length 2.

Definition 12. *The constraint graph of $Q$ is* single 2-cyclic *iff it has at most one cycle, it is of the form $\{x \triangleleft_k y, y \triangleleft_k x\}$, and there are no edges $x \triangleleft_k z$ or $y \triangleleft_k z$, with a variable $z \notin \{x, y\}$.*

Theorem 3. *Let $G$ be a graph database with $n$ nodes and $N$ edges and $K$ be an integer. Then, a representation using $O(N) + 2nK + o(nK)$ words of space can compute $Q(G)$ for extended BGPs $Q$ with constraints of the form $x \triangleleft_k y$, with $1 \leq k \leq K$, and forming a single 2-cyclic graph, in $O(Q^*|Q| \log N)$ time, where $Q^*$ is the solution to the linear program (2).*

Proof. Given the structure of $Q$, we can think of the algorithm as a sequence of steps where it binds one minimal variable $x$ at a time, finishing with a step where it either binds one variable (if $Q$ is acyclic), or the two variables of the only 2-cycle $(x, y)$.

As in the proof of Theorem 2, we consider any solution $\{w_i\}_{i=1}^M$, $\{\delta_{xy}\}_{x \triangleleft_k y \in Q}$, $\{s_{xy}\}_{x \triangleleft_k y \in Q}$ to the linear program (2), and prove that the algorithm runs in time bounded by Eq. (3).

The proof is again by induction on the steps performed by the algorithm. This time, for the base case we have two options: either $Q$ has a single variable, or it has two variables forming a 2-cycle. The proof for the case of a single variable is exactly as in Theorem 2.

If the query features two variables $(x, y)$ forming a 2-cycle, we proceed as follows. Assume $\ell_x \leq \ell_y$; the proof in the other case is analogous. The algorithm would then first iterate over all bindings $I$ for $x$, that is, $I$ is the intersection between all sets $t(x)$, all sets $k$-NN$(a)$ for constraints $a \triangleleft_k x$ in $Q$, and all sets $\{b \mid x \in k\text{-NN}(b)\}$ for constraints $x \triangleleft_k b$ in $Q$. Let us call $\iota = |I|$.

Then, for each such instantiation $c \in I$, the algorithm processes $Q[x \longrightarrow c]$ (the query $Q$ where we replace every $x$ by $c$), which implies intersecting the following sets; $t[x \longrightarrow c]$ denotes the triple $t$ as (possibly) instantiated in $Q[x \longrightarrow c]$.

- Set $t[x \longrightarrow c](y)$ for every triple $t \in Q_y \cap Q_x$.
- Set $t[x \longrightarrow c](y) = t(y)$ for every triple $t \in Q_y \setminus Q_x$.
- Set $k$-NN$(a)$ for each constraint $a \triangleleft_k y$ in $Q[x \longrightarrow c]$, with constant $a$. Note there exists (at least) one such set, for $a = c$.
- Set $\{b \mid y \in k\text{-NN}(b)\}$ for each constraint $y \triangleleft_k b$.

For an instantiation $x := c$, let $t_c$ be the minimum number of values over all the sets $t[x \longrightarrow c](y)$. As explained, the `Ring` can process this intersection in time $O(\min(t_c, k, D) \cdot |Q| \log N)$, and hence the total running time is bounded by $|Q| \log N \sum_{c \in I} \min(t_c, k, D)$. As for the previous base case, for any $p, q, r$ such that $p + q + r \geq 1$, this entails that the running time is bounded by $|Q| \log N$ times

$$\sum_{c \in I} t_c^p\, k^q\, D^r \;\leq\; k^q\, D^r \sum_{c \in I} t_c^p\, 1^{q+r} \;\leq\; k^q\, D^r \left(\sum_{c \in I} t_c\right)^p \left(\sum_{c \in I} 1\right)^{q+r}$$

where the last part follows by Holder's inequality; to apply it we introduced the factor $1^{q+r}$ in the second term.

We note that the rightmost summation is exactly $\iota$. Then, by the same reasoning as in the case where we bind one variable, we have that $\iota^{q+r} \leq \min(\ell_x, k, D)^{q+r}$, or $\min(\ell_x, D)^{q+r}$ if there were no constraints $a \triangleleft_k x$ with $a \neq y$ in $Q$. Now for any numbers $\alpha + \beta + \gamma \geq q + r$ we have that $\min(\ell_x, k, D)^{q+r} \leq \ell_x^\alpha\, k^\beta\, D^\gamma$, and $\min(\ell_x, D)^{q+r} \leq \ell_x^\alpha\, k^\beta\, D^\gamma$ with $\beta = 0$. Summing up, we have:

$$k^q\, D^r \left(\sum_{c \in I} t_c\right)^p \left(\sum_{c \in I} 1\right)^{q+r} \;\leq\; k^{q+\beta}\, D^{r+\gamma} \ell_x^\alpha \left(\sum_{c \in I} t_c\right)^p.$$

Because our weights satisfy program (2), we have that $\delta_{yx} \leq \sum_{t_i \in Q_y} w_i + \sum_{y \triangleleft_k z \in Q} s_{yz}$. We identify three cases, depending on the value of $\delta_{yx}$.

**Case 1** holds when $\delta_{yx} \leq \sum_{t_i \in Q_y \setminus Q_x} w_i$.
**Case 2** holds when $\sum_{t_i \in Q_y \setminus Q_x} w_i < \delta_{yx} \leq \sum_{t_i \in Q_y} w_i$.
**Case 3** holds when $\sum_{t_i \in Q_y} w_i < \delta_{yx}$.

Consider case 1; we explain how to deal with the other cases shortly. Let $p_1 = \sum_{t_i \in Q_y \setminus Q_x} w_i - \delta_{yx}$, and take

$$\begin{aligned} p &= \textstyle\sum_{t_i \in Q_y} w_i - \delta_{yx} & \alpha &= \textstyle\sum_{t_i \in Q_x \setminus Q_y} w_i - p_1 + \delta_{yx} \\ q &= \textstyle\sum_{z \triangleleft_k y \in Q} \delta_{zy} + \delta_{yx} & \beta &= \textstyle\sum_{z \triangleleft_k x \in Q} \delta_{zx} - \delta_{yx} \\ r &= \max(1 - p - q, 0) & \gamma &= \textstyle\sum_{x \triangleleft_k z \in Q} s_{xz} \end{aligned}$$

Note that our assumptions guarantee that each of these values is nonnegative. Moreover, we have that $p_1 = \sum_{t_i \in Q_y} w_i - \sum_{t_i \in Q_y \cap Q_x} w_i - \delta_{yx}$, and thus $\alpha + \beta + \gamma$ is

$$\sum_{t_i \in Q_x} w_i + \sum_{z \triangleleft_k x \in Q} \delta_{zx} + \sum_{x \triangleleft_k z \in Q} s_{xz} - \left(\sum_{t_i \in Q_y} w_i - \delta_{yx}\right) \geq 1 - p \geq q + r.$$

Before finishing we need to further bound the term $(\sum_{c \in I} t_c)^p$. Writing $p_2 = \sum_{t_i \in Q_x \cap Q_y} w_i$, so that $p = p_1 + p_2$, we have that $(\sum_{c \in I} t_c)^p = (\sum_{c \in I} t_c)^{p_1} (\sum_{c \in I} t_c)^{p_2}$.

Recall that $p_1 \geq 0$ in case 1. Thus, we can find a set of weights $w_i' \geq 0$ such that $p_1 = \sum_{t_i \in Q_y \setminus Q_x} w_i'$. Then, $(\sum_{c \in I} t_c)^{p_1}$ can be bounded by $\ell_x^{p_1} \prod_{t_i \in Q_y \setminus Q_x} |t_i|^{w_i'}$, because $\iota \leq \ell_x$ and $t_c \leq |t_i|$ for any triple $t_i$ that only mentions variable $y$ and not $x$. Moreover, $\sum_{c \in I} t_c \leq |t_i|$ for any $t_i \in Q_x \cap Q_y$. Putting everything together, we obtain

$$k^{q+\beta}\, D^{r+\gamma} \ell_x^\alpha \left(\sum_{c \in I} t_c\right)^p \leq k^{q+\beta}\, D^{r+\gamma}\, \ell_x^\alpha \ell_x^{p_1} \prod_{t_i \in Q_y \setminus Q_x} |t_i|^{w_i'} \prod_{t_i \in Q_y \cap Q_x} |t_i|^{w_i}. \quad (4)$$

Finally, note that $\alpha + p_1 = \sum_{Q_x \setminus Q_y} w_i + \delta_{yx}$. Thus, we rewrite $\ell_x^\alpha \ell_x^{p_1}$ as $\ell_x^{\sum_{Q_x \setminus Q_y} w_i} \ell_x^{\delta_{yx}}$. We know that $\ell_x \leq |t_i|$ for any triple $t_i \in Q_x \setminus Q_y$. Given that $\ell_x \leq \ell_y$ and that $\ell_y \leq |t_i|$ also holds for any triple $t_i \in Q_y \setminus Q_x$, we have that $\ell_x \leq |t_i|$ for any such triple, and we can then write

$$\ell_x^\alpha \ell_x^{p_1} \prod_{t_i \in Q_y \setminus Q_x} |t_i|^{w_i'} \leq \prod_{t_i \in Q_x \setminus Q_y} |t_i|^{w_i} \prod_{t_i \in Q_y \setminus Q_x} |t_i|^{w_i},$$

by redistributing back the weight $\delta_{yx}$ to each of the weights $w_i'$. Substituting back in Eq. (4), we obtain

$$k^{q+\beta}\, D^{r+\gamma} \ell_x^\alpha \left(\sum_{c \in I} t_c\right)^p \leq k^{q+\beta}\, D^{r+\gamma} \prod_{t_i \in Q_x \cup Q_y} |t_i|^{w_i},$$

which finishes the proof of case 1. Indeed, $q + \beta$ contains all weights of constraints associated with $x$ or $y$, $\sum_{z \triangleleft_k y \in Q} \delta_{zy} + \sum_{z \triangleleft_k x \in Q} \delta_{zx}$, where $z$ is either $x$ or a constant. Likewise, $r + \gamma$ is bounded by the sum of all the relevant weights $s_{xz}$ and $s_{yz}$.

The other two cases are proved using the same ideas. For case 2, we set instead $p_1 = \delta_{yx}$, and for case 3 we further need to redistribute weights between $p$ and $r$ so as to subtract $\delta_{yx}$.

As the inductive case only features instantiations of variables out of a cycle, we can prove it by combining the proof of Ngo [43, Thm. 5.1] with the techniques introduced in our base case. ◻

We remark that the proof of this theorem, and hence the good properties of our variable ordering, holds for a much wider class of BGPs, in which we permit any number of 2-cycles as long as one of the variables in each of these cycles is also the target of a $k$-NN constraint. The proof is omitted for lack of space.

## 5 IMPLEMENTATION

Our data structure is an extension of the `Ring`, whose implementation [6] is coded in C++ using several structures from the SDSL library [27]. We extend the `Ring` data structure with those for the sequences $S$, $S'$, and bitvector $B$. The sequences are represented with the same wavelet trees used by the `Ring`. Bitvector $B$ is implemented in plain form, with the `bit_vector` class of SDSL. To support select, we use `select_support_mcl`, which takes constant time by using 20% extra space. We compile our code using `gcc` version 6.3.0 with `-O9` optimization.

An important aspect of a practical implementation of `LTJ`-style algorithms is the order in which variables are bound, as explained. We use the following rule [6]: The next variable to bind is the $x$ with minimum $\ell_x$ value in the current (i.e., partially instantiated) query $Q$. Finally, we leave for the end the *lonely* variables, that is, those that appear only once in $Q$ [33]. This algorithm is *adaptive* in the sense that, after binding the first variable $x$ with each value $c$, the next variable to bind may differ on each $Q[x \rightarrow c]$.

Per the `Ring`, we do not compute $|t(x)|$ precisely for the triple $t$ in order to compute $\ell_x$, but rather use the size $e - b + 1$ of the range $[b\,..\,e]$ corresponding to $t$ in the current $Q$. In the case of similarity clauses $x \triangleleft_k a$ or $a \triangleleft_k x$, the size of the corresponding ranges in $S$ or $S'$ are the exact number of different values $x$ can be bound to.

We implemented two variants of our algorithm, and a baseline.

### 5.1 Ring-KNN-S

This variant is a faithful implementation of the technique we describe in Section 3.3, but it does not incorporate the restrictions derived from our optimality analysis in Section 4. That is, we use the variable binding order just described. Although Section 4 suggests that we should aim to eliminate only $C$-minimal variables when similarity clauses are involved, this comes from the fact that the condition $x \triangleleft_k y$ can only bound to $k$ the number of candidates for $y$ given $x$, not the other way. On average, however, the number of values for $x$ given $y$ is also $k$, because there are exactly $kn$ tuples in the virtual relation $k\mathrm{NN}(\cdot,\cdot)$.

Therefore, we expect a similar performance on average when disregarding this restriction, although this could lead to some bad cases and higher variance. On the other hand, having more freedom to choose the next variable to bind may lead to better query plans.

### 5.2 Ring-KNN

This is the full implementation of our technique, observing the restrictions of Section 4.

Each time we must choose the next (non-lonely) variable to eliminate, we pass through the edges $x \longrightarrow y$ of the current constraint graph (i.e., the clauses $x \triangleleft_k y$ where both $x$ and $y$ are variable), marking the variables $y$ as not yet ready to be bound. At the end, if there are unmarked variables, we choose to eliminate the unmarked variable $x$ with the least value $\ell_x$. If all the variables are marked, instead, we choose the (marked) variable $x$ with the least value $\ell_x$.

### 5.3 Baseline

Not having an implementation to compare with, we developed a baseline with a simple solution to the problem that avoids materializing the $k\mathrm{NN}(\cdot,\cdot)$ relation, as discussed in Section 3.2. The idea is to first solve the extended BGP as a BGP, ignoring the similarity clauses, and then postprocessing the solutions with the similarity clauses. Our baseline also builds on the `Ring` for BGPs; it thus solves the query in the following two phases:

(1) With the `Ring`, it computes the full solution to the query without taking into account any similarity constraint.
(2) Then, it filters or extends the previous results by checking the direct and reverse nearest neighbor graph. Both graphs are represented as adjacency vectors in plain form.

Note that a clause $x \triangleleft_k y$ where both $x$ and $y$ appear in other triple patterns of $Q$ will have both variables bound in the final solution, and thus we only have to filter the solution by checking whether $y \in k\text{-NN}(x)$. Instead, if only one variable is bound, we must extend the result with all the possible values of the other, using the direct or the reverse graph. This may bind the variable of another similarity clause, and so on. Our baseline does not support similarity clauses that are disconnected from the rest of the query.

Filtering should be prioritized, as it may eliminate the solution before wasting time extending it. For step 2, then, we first classify the similarity clauses into *2-ready*, *ready* and *sim*: the clauses $x \triangleleft_k y$ where both $x$ and $y$ are bound are in *2-ready*; those with one of them bound are in *ready*; and those with both unbound are in *sim*. We start filtering the results with the clauses in *2-ready*, which can preempt the whole query process if they fail. When *2-ready* becomes empty, we continue with *ready*, generating all the possible values for the other variable by means of the stored $K$-NN graph. When *ready* becomes empty, we continue with *sim*. We process first the clauses that contain the variable that participates in most triples. Note that when we extend a variable from *sim*, others can move to *ready*. Therefore, every time we extend a variable, we update both groups. The process finishes when all groups are empty.

Note that, in essence, the baseline corresponds to solving the triples of the query and leaving all the similarity clauses to the end, whereas our general technique fully incorporates those clauses into the LTJ process, aiming at processing them at the best moment.

## 6 EXPERIMENTS

We compared our implementations on a benchmark we created for this purpose, lacking any standard one. In this section we describe the benchmark and the results of the comparisons, as well as some experiments on the quality of our similarity operators.

## 6.1 Benchmark

We created a benchmark from a dataset that combines the Wikidata graph [54] and IMGpedia [22], a linked-dataset that incorporates images from the Wikimedia Commons dataset and their nearest neighbor graph computed based on visual descriptors of the image content. The identifiers of these images appear as nodes in the Wikidata graph, for example, to indicate that a particular entity (person, building, artwork, etc.) is depicted by a given image. Our dataset contains $617,065,092$ triples which, using 32-bit numbers for the identifiers, occupies 6.9 GB. Additionally, storing the $K$-NN graph for $K = 50$ requires 5.3 GB of extra space. In total, our data set occupies 12.2 GB in plain form.

Regarding queries, based on a real-world query log [12, 38], we keep only those queries that involve some image, obtaining a total of 2,942 queries. Note that those BGPs do not contain similarity constraints. We aim to generate realistic extended BGPs by adding, to those real queries, one or more similarity constraints between image nodes. We extended the queries with different similarity patterns in order to show different situations that affect query evaluation performance, and use $k = 50$ throughout. We obtain a total of 1,470 queries, classified as follows ($q_A$ denotes that $q$ contains, at least, the variables in the set $A$).

- *Q1*: contains 100 queries that join two queries by using the operator $x \lhd_k y$. Given two BGPs $q_{\{x\}}$ and $q_{\{y\}}$, where $x$ and $y$ are images, we produce the extended BGP $q_{\{x\}} \cup \{x \lhd_k y\} \cup q_{\{y\}}$. The variant *Q1b* uses $\{x \sim_k y\}$ instead of $\{x \lhd_k y\}$.
- *Q2*: contains 14 queries that join three queries by using the operator $x \lhd_k y$ twice in a path. Given BGPs $q_{\{x\}}, q_{\{y\}}, q_{\{z\}}$, where the three variables are images, we produce $q_{\{x\}} \cup \{x \lhd_k y\} \cup q_{\{y\}} \cup \{y \lhd_k z\} \cup q_{\{z\}}$. The variant *Q2b* replaces unidirectional by bidirectional similarity operators. We omit a third variant, *Q2t*, that closes the triangle with $\{z \lhd_k x\}$, because it gives almost the same results as *Q2*.
- *Q3*: contains 307 queries that extend a query with a triangle involving similarity. Given a BGP $q_{\{x,y\}}$ that contains $(x, p, y)$, where $y$ is an image, we extend it with the patterns $\{(x, p, y'), y \lhd_k y'\}$. We get close pairs $y, y'$ related to $x$ by $p$.
- *Q4*: contains 20 queries that extend a query as in *Q3*, but now looking for $y'$ with all the properties of $y$. Given a query $q_y$ where $y$ is an image and participates in more than one triple pattern (to avoid duplicating *Q3*), we add a new variable $y'$ and, for each triple pattern that mentions $y$, we add a copy that mentions $y'$ instead. We finally add a clause $y \lhd_k y'$.
- *Q5*: contains the same 307 queries of *Q3*, further extended with a triple pattern $(y, l_1, l_2)$, where $l_1$ and $l_2$ are variables that only participate in this triple (i.e., they are lonely). This extracts all the information about $y$ for each pair $y, y'$.

## 6.2 Results

The experiments were conducted on an Intel® Core$^{\text{TM}}$ i7-3820 CPU @ 3.60GHz (4 cores) with 10MB of cache and 256 GB of RAM, running Debian GNU/Linux 9 with kernel 4.9.0-8 (64 bits).

In space, both Ring variants need 12.15 GB to store the Ring and the $K$-NN graph. This is almost the same space to store the raw data (which our index replaces, as any edge of $G$ or of the $K$-NN graph can be retrieved from the wavelet trees). The baseline uses more space, 17.99 GB, as it stores the $K$-NN graph in plain form.

We evaluated the 1,470 queries on the baseline and our Ring variants. All the queries are run with a timeout of 10 minutes and without limiting the number of results. Figure 2 shows the query time distributions using violin plots [30] (which show a symmetric histogram of values along the $y$ axis) plus averages and medians.

*Queries Q1.* In these queries Ring-KNN is on average 15% faster than the baseline. The queries have the form $q_{\{x\}} \cdot x \lhd_k y \cdot q_{\{y\}}$, so the baseline will fully compute $q_{\{x\}}$ and $q_{\{y\}}$ and finally filter the pairs $(x, y)$ that do not satisfy the constraint. The Ring variants, instead, may restrict the results earlier using the similarity clause. Per Section 4, Ring-KNN cannot bind $y$ before binding $x$, whereas Ring-KNN-S is free to do so. This makes Ring-KNN-S 60% faster than the baseline, and also significantly faster than Ring-KNN.

On *Q1b*, again per Section 4, Ring-KNN will not bind $x$ nor $y$ until the end, as they form a 2-cycle $x \sim_k y$. Still, $x$ and $y$ are always bound before the lonely variables, that is, Ring-KNN does not compute all the results of $q_{\{x\}} \cdot q_{\{y\}}$, while the baseline does. This makes Ring-KNN still 10% faster than the baseline on average. Due to its flexibility, Ring-KNN-S still outperforms Ring-KNN by 60% on average. The difference in their plans is illustrated by the fact that, on average, Ring-KNN-S and Ring-KNN bind the first variable involved in a similarity constraint after processing 36% and 68% of the variables, respectively.

A closer analysis of the time distribution reveals additional benefits of the Ring-KNN plans, as its median is 40% lower than the baseline. This means that, although there are bad cases that make its average closer to that of the baseline, Ring-KNN solves many queries faster. Ring-KNN-S times also distribute much better than the baseline, with the medians being 2.6 times lower. The better distribution of times for Ring-KNN and the even-better distribution for Ring-KNN-S are observed in the violin plots.

*Queries Q2.* These queries feature two or three independent similarity predicates between three variables, which the baseline is forced to leave to the end. The average difference between Ring-KNN and the baseline stretches to 55%, while Ring-KNN-S is now only 12% faster than Ring-KNN on average. The fact that the median of Ring-KNN is "only" 20% faster than that of the baseline shows that the former is also more stable. It is also more stable than Ring-KNN-S, as the difference in their medians reaches 45%, but in exchange the latter is considerably faster in most queries. The violin plots clearly show that the guards of Ring-KNN sharply limits the bad cases, which reach the timeouts for Ring-KNN-S. Interestingly, there is little difference between *Q2*, where $x$, $y$, and $z$ form a chain, and *Q2t*, where they form a triangle.

Both Ring variants worsen on the symmetric queries of *Q2b*. The cycles force Ring-KNN to leave the three variables to the end (still before the lonely ones), so its distance to the baseline decreases to 20% in average and 25% in medians. Ring-KNN-S is 10% faster than Ring-KNN, but its median is worse by 13%. The violin plot shows that, again, the variable ordering of Ring-KNN limits its worst cases; the distribution is in general preferable to that of Ring-KNN-S.

*Queries Q3.* These queries represent a different situation, where the similarity clause appears "in an extreme" of the query, instead

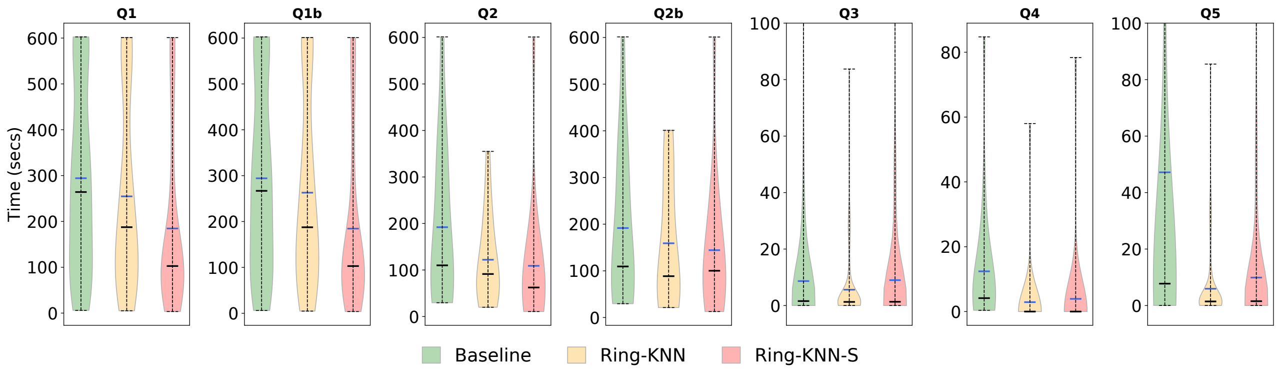


**Figure 2: Query time distribution per query type (600 seconds is the timeout). The upper segment in each plot is the mean and the lower is the median. The Baseline and Ring-KNN-S reach the timeout in *Q3* and *Q5*.**

of "in the middle" connecting two queries as in *Q2*. The baseline generates and tests all the pairs $y, y'$ where both are connected to $x$ by predicate $p$. Ring-KNN can bind $y$ at any moment, and then $y'$. Ring-KNN-S, instead, binds $y$ and $y'$ with no restrictions.

As a result, Ring-KNN is 55% faster than the baseline on average, and its median is also 25% lower. Interestingly, it is also 60% faster than Ring-KNN-S on average, although the medians are nearly the same. This shows that the query plans of Ring-KNN, which cares about binding $y$ before $y'$, feature more stable times than those of Ring-KNN-S, as it avoids some bad cases that worsen the average of the latter. The violin plots clearly illustrate the better stability and overall performance of Ring-KNN over Ring-KNN-S.

*Queries Q4.* In this case, $y$ and $y'$ are more densely connected to the query than in *Q3*, which yields fewer candidate pairs to check by the baseline, but more triple patterns to handle. In this query Ring-KNN and Ring-KNN-S become more than 4 and 3 times faster than the baseline on average, respectively. This is the only set of queries that includes empty results, which is detected very early by the Ring variants, but not by the baseline: the median of the baseline is over 4 seconds, whereas those of the Ring variants are $3 \cdot 10^{-5}$. We also note that Ring-KNN performs better than Ring-KNN-S: 35% faster on average and with better worst cases, plus a slightly more stable violin plot.

*Queries Q5.* The last type of query is designed specifically to illustrate how bad a simple baseline that separates the similarity clauses from the main query plan can perform, since it must produce all the instantiations of $l1$ and $l2$ before checking if $y$ satisfies the similarity clause. As expected, the baseline is an order of magnitude slower than the Ring variants. As in *Q3*, Ring-KNN clearly outperforms the simple Ring-KNN-S, being 50% faster on average. The distributions are similar (the median of Ring-KNN being only 7% lower), which again shows that Ring-KNN-S incurs many more bad cases that raise its average. This is confirmed in the violin plots.

*Summary.* Our experiments demonstrate that incorporating the similarity clauses in the main body of the LTJ query algorithm performs much better than a naive strategy that uses LTJ over triples as a black box and postpones the similarity checks until the end. While the difference is moderate in simple cases (*Q1*), it becomes more noticeable when the clauses are more connected to the query (*Q2*, *Q3*), and it may become very sharp (*Q4*, *Q5*).

The comparison between the simple and the full Ring variants also depends on the complexity of the query. In simpler cases (*Q1*), Ring-KNN-S is more effective by exploiting the opportunity of binding the variables involved in similarity clauses earlier. The full Ring-KNN is slower in practice on those queries. In particular, the cycles further restrict Ring-KNN's plans and make it closer to the baseline, as it must bind all the other non-lonely variables first. As the queries get more complicated, however, with more similarity constraints or with constraints involved in cycles (*Q2* onwards), the careful variable ordering of Ring-KNN protects it against bad cases and makes it preferable over Ring-KNN-S. The latter is completely outperformed in the more involved queries (*Q3*–*Q5*).

## 6.3 On the quality of the symmetric similarity

Per the possible unintuitiveness of the asymmetric operator $x \triangleleft_k y$ for users, we introduced a symmetric version $x \sim_k y$ that intersects the results of $x \triangleleft_k y$ and $y \triangleleft_k x$. Implementing the symmetric version brought a number of challenges, both theoretical (to achieve worst-case optimality, Section 4.3) and practical (restricted query plans, Section 6.2) . A natural question is about the quality of the results obtained with this symmetric operator compared to the classic $k$-NN results. We present an experiment to address that question.

For this experiment, we use real datasets that serve as a ground truth about which returned results are considered "good" or "bad". We use two different datasets for this purpose, which divide the data into classes. We assume that a result returned from the same class of the query is good, otherwise it is bad.

- Anuran Calls dataset [16, 17]: It consists of 7,195 vectors of dimension 22, formed by audio features (Mel Frequency Cepstral Coefficients or MFCCs) extracted from syllabes of frog calls. There are 10 unbalanced classes in the dataset (size of smaller class is 68, size of larger class is 3,478), each corresponding to a different species of frogs.
- Dry Bean dataset [36, 37]: It consists of 13,611 vectors of dimension 16, formed by features extracted from dry bean grains. There are 7 unbalanced classes in the dataset (size of smaller class is 522, size of larger class is 3,546), each

corresponding to a different class of beans. We normalized linearly the values of each feature in the range $[0, 1]$ .

We built the $K$-NN graph of both datasets, for $K = 100$, using Euclidean distance on the vectors, and then queried every object $x$ in each dataset for each value $k$ in $\{5, 10, \ldots, 100\}$. For each query object $x$ and value of $k$, we performed queries according to four strategies: (1) $k$-NN (labeled kNN and corresponding to $x \triangleleft_k y$), where we return the first $k$ neighbors $y$ of $x$ in the $K$-NN graph; (2) reverse $k$-NN (labeled `reverse` and corresponding to $y \triangleleft_k x$), where we return those $y$ that list $x$ among their first $k$ neighbors in the $K$-NN graph; (3) the intersection of $k$-NN and reverse $k$-NN (labeled `intersection` and corresponding to $x \sim_k y$); and (4) the union of $k$-NN and reverse $k$-NN (labeled `union`, another symmetric alternative to the intersection). For each dataset and $k$, we average the precision value (fraction of the elements returned that belong to the same class of $x$) over all the query objects $x$. The higher the precision, the better is the quality of the operation.

Figure 3 shows the average precision for each strategy on both datasets; see the four lines that reach $k = 100$ for now. We observe that the precision of the $k$-NN strategy diminishes with $k$, which is expected because a larger answer is more likely to contain objects from other classes (which lowers the precision). This is the case of the other strategies as well, though some start growing for low values of $k$. The reverse $k$-NN strategy, although returning on average $k$ answers, consistently displays less precision than that of $k$-NN. The same occurs with the union strategy, the other symmetric option we disregarded. The intersection, instead, is competitive with the $k$-NN strategy and outperforms it from some value of $k$ between 25 and 30, more markedly on the the Anuran Calls dataset.

Note that the strategy $k$-NN returns exactly $k$ objects, the reverse strategy returns on average $k$ objects (and thus exactly $k$ objects in our plots that query for all the objects), the intersection returns at most $k$ objects, and the union returns at least $k$ objects. Thus comparing them all for the same $k$ may be seen as unfair. The figure then also shows the results for union and intersection classified according to the average number of values returned by the queries. The relative results are similar as before, although this time the $k$-NN strategy outperforms intersection on the Dry Bean dataset.

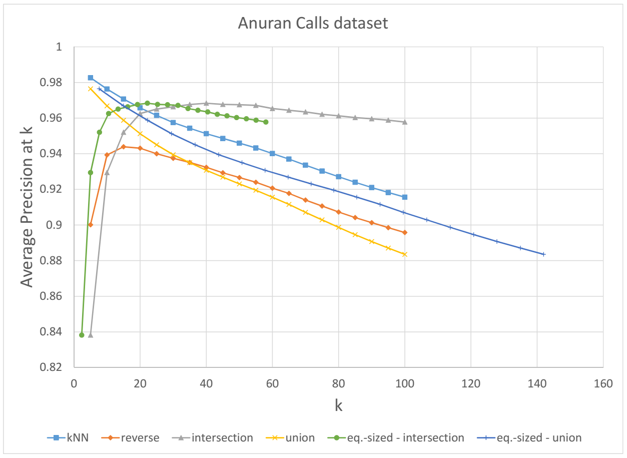
On both datasets, the difference on average precision comparing equally-sized rankings is not so high: the maximum difference, for $k \geq 10$, is below 8% on the Anuran Calls dataset. These results evidence that all tested similarity join strategies could be meaningful and useful in practice. Still, kNN and `intersection`, the two we implemented, perform consistently better (note also that `union` is not as natural to support with LTJ as `intersection`). There is no conclusive difference in quality between kNN and `intersection`. While the $x \triangleleft_k y$ operation is more efficient, the $x \sim_k y$ operator may be more intuitive for some users. In particular, since the `reverse` strategy is consistently worse than kNN, it turns out that the $y$'s that match each $x$ in the result of $x \triangleleft_k y$ are of better quality that the $x$'s that match each $y$; such an asymmetry also reduces intuitiveness.

## 7 DISCUSSION AND FUTURE WORK

Our theoretical results show that the acyclic queries are easily solved in wco time by just taking the similarity constraints $x \triangleleft_k y$ in topological order. When the query has cycles, we could only prove bounds for queries featuring one 2-cycle by binding its variables at the end (save for lonely variables). Our algorithm Ring-KNN extends this strategy to general queries, by never binding $y$ before $x$ if possible. While the general wco of this strategy is not established, it proves to be superior in practice – for all but the simplest cases – when compared with the basic Ring-KNN-S that treats the constraints as any other triple. In simplest cases, Ring-KNN-S is faster as it has more freedom to choose the binding order.

Our experiments also show that, though the symmetric operator $x \sim_k y$ (which produces, precisely, 2-cycles) might be more intuitive than the basic asymmetric $x \triangleleft_k y$ version, they are equivalent in terms of retrieval quality (and better than another symmetric operator defined as the union of $x \triangleleft_k y$ and $y \triangleleft_k x$). Our experiments also show that the cost per delivered tuple is 2–5 times higher with the symmetric operator with all Ring strategies. This situation leads us to consider other ways to define a symmetric operator that is equally intuitive and easier to handle algorithmically.

One choice, for example, is to interpret any similarity clause in either direction, whichever appears to be more convenient at query time. In particular, if the user does not specify the direction of a similarity clause and the system can define it as $x \triangleleft_k y$ or $y \triangleleft_k x$, we



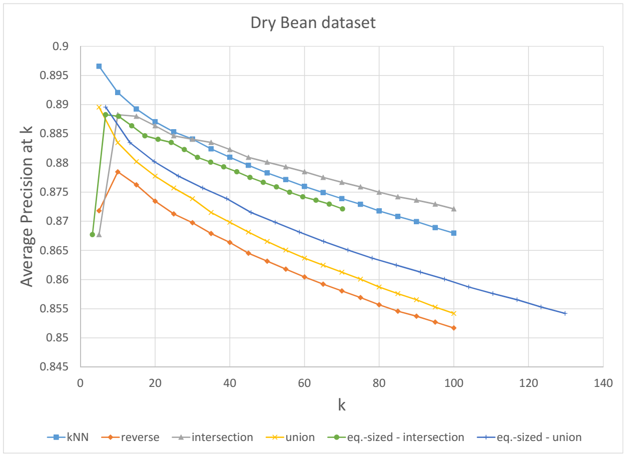


**Figure 3: Average Precision@k for both datasets.**

can always make the query acyclic and solve it in wco time. Query answers may differ slightly depending on which order is chosen, so this approach can be seen as a way of producing faster, approximate answers, akin to the technology used in vector databases to perform similarity searches. Further setting the direction of the constraint opens new avenues for query optimization, looking for the fastest query plan regardless of its comparative quality.

Towards devising further similarity operators that can be powered with our algorithmic machinery, we will pursue eliminating the need to specify $k$ at the low-level operations $x \triangleleft_k y$ or $x \sim_k y$; the query would instead ask for the $k^*$ "best" results, where the nodes involved in similarity constraints are most similar. For example, in the query $\{\text{Chile} \triangleleft_3 y, (y, \text{continent}, \text{Europe})\}$, which looks for European countries similar (under some metric) to Chile, a user may want to select the best three results, even if no European country is among the $k = 3$ countries most similar to Chile worldwide. In order to enable this semantics, the system would increase the value of $k$ until $k^* = 3$ results are obtained.

## ACKNOWLEDGMENTS

Funded by ANID – Millennium Science Initiative Program – Code ICN17_002. A.G. is funded in part by MCIN/AEI/10.13039/5011000-11033: grant PID2020-114635RB-I00 (EXTRACompact); by MCIN/AEI/10.13039/501100011033 and EU/ERDF "A way of making Europe": PID2022-141027NB-C21 (EARTHDL); by MCIN/AEI/10.13039/501100011033 and "Next-GenerationEU"/ PRTR: grants TED2021-129245B-C21 (PLAGEMIS), PDC2021-120917-C21 (SIGTRANS) and by GAIN/Xunta de Galicia: GRC: grants ED431C 2021/53, and CIGUS 2023-2026. G.N. was funded in part by Fondecyt Grant 1-230755. J.R. was funded in part by Fondecyt Grant 1221799.

## REFERENCES

[1] M. Abo Khamis, H. Q. Ngo, and D. Suciu. Computing join queries with functional dependencies. In *Proc. 35th ACM Symposium on Principles of Database Systems (PODS)*, pages 327–342, 2016.

[2] M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do Shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. 36th ACM Symposium on Principles of Database Systems (PODS)*, pages 429–444, 2017.

[3] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, 2018.

[4] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

[5] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1:1–1:39, 2008.

[6] D. Arroyuelo, A. Hogan, G. Navarro, J. L. Reutter, J. Rojas-Ledesma, and A. Soto. Worst-case optimal graph joins in almost no space. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 102–114. ACM, 2021.

[7] D. Arroyuelo, G. Navarro, J. L. Reutter, and J. Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Transactions on Database Systems*, 47(2):8:1–8:53, 2022.

[8] B. Arsintescu. How LIquid connects everything so our members can do anything. LinkedIn Engineering Blog, 2023. https://engineering.linkedin.com/blog/2023/how-liquid-connects-everything-so-our-members-can-do-anything.

[9] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[10] R. Battle and D. Kolas. Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. *Semantic Web*, 3(4):355–370, 2012.

[11] B. R. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, S. Rondelli, A. Ryazanov, M. Schmidt, K. Sengupta, B. B. Thompson, D. Vaidya, and S. Wang. Amazon Neptune: Graph data management in the Cloud. In *Proc. ISWC Posters & Demonstrations*, 2018.

[12] A. Bonifati, W. Martens, and T. Timm. Navigating the maze of Wikidata query logs. In *Proc. World Wide Web Conference (WWW)*, pages 127–138, 2019.

[13] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. 22nd International Conference on Data Engineering, (ICDE)*, page 5, 2006.

[14] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

[15] W. W. Cohen and H. Hirsh. Joins that generalize: Text classification using WHIRL. In *Proc. 4th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 169–173, 1998.

[16] J. Colonna, E. Nakamura, M. Cristo, and M. Gordo. Anuran Calls (MFCCs). UCI Machine Learning Repository, 2017. DOI: https://doi.org/10.24432/C5CC9H.

[17] J. Colonna, T. Peet, C. A. Ferreira, A. M. Jorge, E. F. Gomes, and J. a. Gama. Automatic classification of Anuran sounds using convolutional neural networks. In *Proc. 9th International C* Conference on Computer Science & Software Engineering (C3S2E)*, page 73–78, 2016.

[18] A. Dashti, I. Komarov, and R. M. D'Souza. Efficient computation of k-nearest neighbour graphs for large high-dimensional data sets on GPU clusters. *PLOS ONE*, 8(9):1–12, 09 2013.

[19] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoc, M. Wu, and F. Zemke. Graph pattern matching in GQL and SQL/PGQ. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 2246–2258, 2022.

[20] M. T. Dickerson and D. Eppstein. Algorithms for proximity problems in higher dimensions. *Computational Geometry*, 5(5):277–291, 1996.

[21] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proc. 20th International Conference on World Wide Web (WWW)*, page 577–586, 2011.

[22] S. Ferrada, B. Bustos, and A. Hogan. IMGpedia: A linked dataset with content-based analysis of Wikimedia images. In *Proc. 16th International Semantic Web Conference (SWC)*, pages 84–93, 2017.

[23] S. Ferrada, B. Bustos, and A. Hogan. Extending SPARQL with similarity joins. In *Proc. 19th International Semantic Web Conference (ISWC)*, pages 201–217, 2020.

[24] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment*, 13(11):1891–1904, 2020.

[25] P. Fuchs, P. A. Boncz, and B. Ghit. EdgeFrame: Worst-case optimal joins for graph-pattern matching in Spark. In *Proc. 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 4:1–4:11, 2020.

[26] M. Galkin, M. Vidal, and S. Auer. Towards a multi-way similarity join operator. In *Proc. New Trends in Databases and Information Systems (ADBIS Short Papers and Workshops)*, pages 267–274, 2017.

[27] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.

[28] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[29] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. Lin. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 7(13):1379–1380, 2014.

[30] J. L. Hintze and R. D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181––184, 1998.

[31] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 237–248, 1998.

[32] A. Hogan, M. Mellotte, G. Powell, and D. Stampouli. Towards fuzzy query-relaxation for RDF. In *Proc. Extended Semantic Web Conference (ESWC)*, pages 687–702, 2012.

[33] A. Hogan, C. Riveros, C. Rojas, and A. Soto. A worst-case optimal join algorithm for SPARQL. In *Proc. International Semantic Web Conference (ISWC)*, pages 258–275, 2019.

[34] O. Kalinsky, A. Hogan, O. Mishali, Y. Etsion, and B. Kimelfeld. Exploration of knowledge graphs via online aggregation. In *Proc. 38th International Conference on Data Engineering (ICDE)*, pages 2695–2708, 2022.

[35] C. Kiefer, A. Bernstein, and M. Stocker. The fundamentals of iSPARQL: A virtual triple approach for similarity-based semantic Web tasks. In *Proc. International Semantic Web Conference (ISWC)*, pages 295–309, 2007.

[36] M. Koklu and I. A. Ozkan. Dry Bean Dataset. UCI Machine Learning Repository, 2020. DOI: https://doi.org/10.24432/C50S4B.

[37] M. Koklu and I. A. Ozkan. Multiclass classification of dry beans using computer vision and machine learning techniques. *Computers and Electronics in Agriculture*, 174:article 105507, 2020.

[38] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *Proc. 17th International Semantic Web Conference (ISWC)*, pages 376–394, 2018.

[39] A. Mhedhbi, C. Kankanamge, and S. Salihoglu. Optimizing one-time and continuous subgraph queries using worst-case optimal joins. *ACM Transactions on Database Systems*, 46(2):6:1–6:45, 2021.

[40] A. Mhedhbi and S. Salihoglu. Modern techniques for querying graph-structured relations: Foundations, system implementations, and open challenges. *Proceedings of the VLDB Endowment*, 15(12):3762–3765, 2022.

[41] I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

[42] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[43] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proc. 37th ACM Symposium on Principles of Database Systems (PODS)*, pages 111–124, 2018.

[44] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *Journal of the ACM*, 65(3):16:1–16:40, 2018.

[45] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *Proc. 3rd International Workshop on Graph Data Management Experiences and Systems (GRADES)*, pages 2:1–2:8, 2015.

[46] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 85–97, 2006.

[47] M. A. Sherif and A. N. Ngomo. A systematic survey of point set distance measures for link discovery. *Semantic Web*, 9(5):589–604, 2018.

[48] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *Proc. 26th International Conference on Data Engineering (ICDE)*, pages 892–903, 2010.

[49] T. Skopal and B. Bustos. On nonmetric similarity search problems in complex domains. *ACM Computing Surveys*, 43:34:1–34:50, 2011.

[50] N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proceedings of the VLDB Endowment*, 13(9):1582–1597, 2020.

[51] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Beyond equi-joins: Ranking, enumeration and factorization. *Proceedings of the VLDB Endowment*, 14(11):2599–2612, 2021.

[52] P. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4(2):101–116, 1989.

[53] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory (ICDT)*, pages 96–106, 2014.

[54] D. Vrandecic and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[55] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1106–1113, 2012.

[56] D. A. White and R. C. Jain. Similarity indexing with the SS-tree. In *Proc. 12th International Conference on Data Engineering (ICDE)*, pages 516–523, 1996.

[57] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.

[58] X. Zhai, L. Huang, and Z. Xiao. Geo-spatial query based on extended SPARQL. In *Proc. International Conference on Geoinformatics (GEOINFORMATICS)*, pages 1–4, 2010.

[59] Y.-M. Zhang, K. Huang, G. Geng, and C.-L. Liu. Fast knn graph construction with locality sensitive hashing. In *Machine Learning and Knowledge Discovery in Databases*, pages 660–674. Springer, 2013.