Extending SPARQL with Similarity Joins

Chapter · November 2020

DOI: 10.1007/978-3-030-62419-4_12

Sebastián Ferrada
Linköping University
6 PUBLICATIONS **44** CITATIONS

Extending SPARQL with Similarity Joins

Sebastián Ferrada[0000-0002-9834-8376], Benjamin Bustos[0000-0002-3955-361X], and Aidan Hogan[0000-0001-9482-1982]

Department of Computer Science, Universidad de Chile
Millenium Institute for Foundational Research on Data
{sferrada, bebustos, ahogan}@dcc.uchile.cl

Abstract. We propose techniques that support the efficient computation of multidimensional similarity joins in an RDF/SPARQL setting, where similarity in an RDF graph is measured with respect to a set of attributes selected in the SPARQL query. While similarity joins have been studied in other contexts, RDF graphs present unique challenges. We discuss how a similarity join operator can be included in the SPARQL language, and investigate ways in which it can be implemented and optimised. We devise experiments to compare three similarity join algorithms over two datasets. Our results reveal that our techniques outperform DB-SimJoin: a PostgreSQL extension that supports similarity joins.

Keywords: Similarity Joins, SPARQL

1 Introduction

RDF datasets are often made accessible on the Web through a SPARQL endpoint where users typically write queries requesting *exact* matches on the content. For instance, in Wikidata [27], a SPARQL query may request *the names of Nobel laureates that have fought in a war*. However, there are times when users need answers to a *similarity query*, such as requesting *the Latin American country with the most similar population and GDP to a European country*. The potential applications for efficient similarity queries in SPARQL are numerous, including: entity comparison and linking [23,24], multimedia retrieval [9,15], similarity graph management [10,7], pattern recognition [4], query relaxation [12], as well as domain-specific use-cases, such as protein similarity queries [2].

An important feature for similarity queries are *similarity joins* $X \bowtie_{\mathfrak{s}} Y$, which obtain all pairs (x, y) from the (natural) join $X \bowtie Y$ such that $x \in X$, $y \in Y$, and additionally, x is similar to y according to similarity criteria $\mathfrak{s}$. Similarities are often measured in terms of distance functions between pairs of objects in a d-dimensional vector space, with two objects being more similar the closer they are in that space. A distance function $\delta : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is called a *metric* when it is non-negative, reflexive, symmetric and satisfies the triangle inequality. There are two main types of similarity criteria $\mathfrak{s}$ considered in practice: *a*) in a *range-based* similarity join, $\mathfrak{s}$ specifies a range r such that the distance between output pairs must be below r; and *b*), in a *k-nearest neighbours* (k-nn) similarity join, $\mathfrak{s}$ specifies an integer k such that the pair (x, y) will be output if and only if there are fewer than k other elements in Y with lower distance to x.

Similarity joins for some metrics (e.g., Manhattan distance) can be expressed in SPARQL using built-in numeric operators, order-by, limit, etc. Other metrics can at best be approximated; for example, SPARQL offers no direct way to compute a square root for Euclidean distance. Even when similarity joins can be expressed, a SPARQL engine will typically evaluate these queries by computing distances for all pairs and then filtering by the specific criteria. Conversely, a variety of algorithms and indexes have been proposed to evaluate similarity joins in a more efficient manner than processing all pairs, where the available optimisations depend on the precise definition of $\mathfrak{s}$. Compiling similarity joins expressed in vanilla SPARQL into optimised physical operators would require showing equivalence of the SPARQL expression to the similarity join supported by the physical operator, which is not even clear to be decidable. Thus, dedicated query operators for similarity joins address both usability and efficiency.

Though similarity joins have been well-studied, a key challenge arising in the RDF/SPARQL setting is that of *dimensionality*, where we allow the user to select any number of dimensions from the data, including dynamic dimensions computed from the data (through functions, aggregations, etc.). Being dimension-agnostic introduces various complications; for example, indexing on all combinations of d dimensions would naively result in $O(2^d)$ different indexes, and would not support dynamic dimensions. Such challenges distinguish the problem of supporting similarity queries in SPARQL from typical usage in multimedia databases (based on fixed descriptors), and also from works on supporting domain-specific distances in query languages, such as geographic distances [1,29].

In this paper, we propose to extend SPARQL with multidimensional similarity joins in metric spaces, and investigate optimised techniques for evaluating such queries over RDF graphs. Most works thus far on extending SPARQL with similarity features have either focused on (1) unidimensional similarity measures that consider similarity with respect to one attribute at a time [14,26], or (2) domain-specific fixed-dimensional similarity measures, such as geographic distance [1,29]. Other approaches rather pre-compute and index similarity scores as part of the RDF graphs [7,12,20] or support metric distances measures external to a query engine [19,24]. To the best of our knowledge, our proposal is the first to consider multidimensional similarity queries in the context of SPARQL, where the closest proposal to ours is DBSimJoin [25]: a PostgreSQL extension, which – though it lacks features we argue to be important for the RDF/SPARQL setting (namely k-nn semantics) – we will consider as a baseline for experiments.

Section 2 discusses literature regarding efficient similarity join evaluation, and proposals to include such joins in database systems. In Section 3 we propose the syntax and semantics of a SPARQL extension that supports similarity joins. Section 4 presents our implementation, shows use-case queries and discusses possible optimisations. In Section 5 we perform experiments over two real datasets; we compare different evaluation algorithms, further adopting DBSimJoin as a baseline system. We conclude and outline future directions in Section 6.

2 Related Work

In this section we first describe works addressing the efficient evaluation of similarity joins. Thereafter, we discuss works on similarity queries and distance computation in SPARQL and other query languages for database systems.

Similarity Joins: The brute force method for computing a similarity join between X and Y is to use a *nested loop*, which computes for each $x \in X$ the distance to each $y \in Y$, outputting the pair if it satisfies the similarity condition, thus performing $|X| \cdot |Y|$ distance computations. For range or nearest-neighbour queries over metric distances, there are then three main strategies to improve upon the brute force method: *indexing*, *space partitioning*, and/or *approximation*.

A common way to optimise similarity joins is to index the data using tree structures that divide the space in different ways (offline), then pruning distant pairs of objects from comparison (online). Among such approaches, we highlight *vantage-point Trees* (*vp-Trees*) [28], which make recursive ball cuts of space centred on selected points, attempting to evenly distribute objects inside and outside the ball. vp-Trees have an average-case search time of $O(n^{\alpha})$ on n objects, where $0 \leq \alpha \leq 1$ depends on the distance distribution and dimensionality of the space, among other factors [17], thus having an upper bound of $O(n^{2\alpha})$ for a similarity join. Other tree indexes, such as the D-Index [6] and the List of Twin Clusters [21], propose to use clustering techniques over the data.

Other space partitioning algorithms are not used for indexing but rather for evaluating similarity joins online. The Quickjoin (QJ) algorithm [13] was designed to improve upon grid-based partition algorithms [3,5]; it divides the space into ball cuts using random data objects as pivots, splitting the data into the vectors inside and outside the ball, proceeding recursively until the groups are small enough to perform a nested loop. It keeps window partitions in the boundaries of the ball in case there are pairs needed for the result with vectors assigned to different partitions. QJ requires $O(n(1+w)^{\lceil \log n \rceil})$ distance computations, where w is the average fraction of elements within the window partitions. QJ was intended for range-based similarity joins and extending QJ to compute a k-nn similarity join appears far from trivial, since its simulation with a range-based join would force most of the data to fall within the window partitions, thus meaning that QJ will reach its quadratic worst case.

Another alternative is to apply approximations to evaluate similarity joins, trading the precision of results for more efficient computation. FLANN [16] is a library that provides several approximate k-nn algorithms based on randomised k-d-forests, k-means trees, locality-sensitive hashing, etc.; it automatically selects the best algorithm to index and query the data, based, for example, on a target precision, which can be traded-off to improve execution time.

Similarity in Databases: Though similarity joins do not form part of standard query languages, such as SQL or SPARQL, a number of systems have integrated variations of such joins within databases. In the context of SQL, DBSimJoin [25] implements a range-based similarity join operator for PostgreSQL. This implementation claims to handle any metric space, thus supporting various metric distances; it is based on the aforementioned index-free QJ algorithm.

A number of works have proposed online computation of similarity joins in the context of domain-specific measures. Zhai et. al [29] use OWL to describe the spatial information of a map of a Chinese city, enabling geospatial SPARQL queries that include the computation of distances between places. The Parliament SPARQL engine [1] implements an OGC standard called GeoSPARQL, which aside from various geometric operators, also includes geospatial distance. Works on link discovery may also consider specific forms of similarity measures [24], often string similarity measures over labels and descriptions [26].

Other approaches pre-materialise distance values that can then be incorporated into (standard) SPARQL queries. IMGpedia [7] pre-computes a k-nn self similarity join offline over images and stores the results as part of the graph. Similarity measures have also been investigated for the purposes of SPARQL query relaxation, whereby, in cases where a precise query returns no or few results, relaxation finds queries returning similar results [12,20].

Galvin et al. [8] propose a multiway similarity join operator for RDF; however, the notion of similarity considered is based on semantic similarity that tries to match different terms referring to the same real-world entity. Closer to our work lies iSPARQL [14], which extends SPARQL with `IMPRECISE` clauses that can include similarity joins on individual attributes. A variety of distance measures are proposed for individual dimensions/attributes, along with aggregators for combining dimensions. However, in terms of evaluation, distances are computed in an attribute-at-a-time manner and input into an aggregator. For the multidimensional setting, a (brute-force) nested loop needs to be performed; the authors leave optimisations in the multidimensional setting for future work [14].

Novelty: To the best of our knowledge, the two proposals most closely related to our work are DBSimJoin [25] and iSPARQL [14]. Unlike DBSimJoin, our goal is to introduce similarity joins to the RDF/SPARQL setting. Unlike both systems, we support k-nn semantics for similarity join evaluation, thus obviating the need for users to explicitly specify range values, which can be unintuitive within abstract metric spaces. We further outperform both systems (including under range semantics) by incorporating more efficient similarity join algorithms than the nested-loop joins of iSPARQL [14] and the Quickjoin of DBSimJoin [25]. Without the proposed extension, queries attempting to generate some kind of similarity search in SPARQL would be a) too verbose and b) too costly, since there is no clear strategy to avoid nested-loop executions.

3 Syntax and Semantics

In this section, we define the desiderata, concrete syntax and semantics for our proposed extension of SPARQL for supporting similarity joins.

3.1 Desiderata

We consider the following list of desiderata for the similarity join operator:

- *Closure*: Similarity joins should be freely combinable with other SPARQL query operators in the same manner as other forms of joins.
- *Extensibility*: There is no one-size-fits-all similarity metric [14]; hence the operator should allow for custom metrics to be defined.
- *Robustness*: The similarity join should make as few assumptions as possible about the input data in terms of comparability, completeness, etc.
- *Usability*: The feature should be easy for SPARQL users to adopt.

With respect to *closure*, we define a similarity join analogously to other forms of joins that combine graph patterns in the `WHERE` clause of a SPARQL query; furthermore, we allow the computed distance measure to be bound to a variable, facilitating its use beyond the similarity join. With respect to *extensibility*, rather than assume one metric, we make the type of distance metric used explicit in the semantics and syntax, allowing other types of distance metric to be used in future. Regarding *robustness*, we follow the precedent of SPARQL's error-handling when dealing with incompatible types or unbound values. Finally, regarding *usability*, we support syntactic features for both range-based semantics and k-nn semantics, noting that specifying particular distances for ranges can be unintuitive in abstract, high-dimensional metric spaces.

3.2 Syntax

In defining the syntax for similarity joins, we generally follow the convention of SPARQL for other binary operators present in the standard that allow for combining the solutions of two SPARQL graph patterns [11], such as `OPTIONAL` and `MINUS`. Besides stating the two graph patterns that form the operands of the similarity join, it is necessary to further define at least the following: the attributes from each graph pattern with respect to which the distance is computed, the distance function to be used, a variable to bind the distance value to, and a similarity parameter (search radius or number of nearest neighbours).

We propose the following extension to the SPARQL 1.1 EBNF Grammar [11], adding one new production rule (for `SimilarityGraphPattern`) and extending one existing production rule (`GraphPatternNotTriples`). All other non-terminals are interpreted per the standard EBNF Grammar [11].

```
SimilarityGraphPattern ::= 'SIMILARITY JOIN ON (' Var+ ') (' Var+ ')'
                             ( 'TOP' INTEGER | 'WITHIN' DECIMAL ) 'DISTANCE' iri 'AS' Var
                             GroupGraphPattern
GraphPatternNotTriples ::= GroupOrUnionGraphPattern | ··· | SimilarityGraphPattern
```

The keyword `ON` is used to define the variables in both graph patterns upon which the distance is computed; the keywords `TOP` and `WITHIN` denote a k-nn query and an r-range query respectively; the keyword `DISTANCE` specifies the IRI of the distance function to be used for the evaluation of the join, whose result will be bound to the variable indicated with `AS`, which is expected to be *fresh*, i.e., to not appear elsewhere in the `SimilarityGraphPattern` (similar to `BIND`). The syntax may be extended in future to provide further customisation, such as supporting different normalisation functions, or to define default parameters.

Depending on the metric, we could, in principle, express such queries as vanilla SPARQL 1.1 queries, taking advantage of features such as variable binding, numeric expressions, sub-selects, etc. However, there are two key advantages of the dedicated syntax: (1) similarity join queries in vanilla syntax are complex to express, particularly in the case of k-nn queries or metrics without the corresponding numeric operators in SPARQL; (2) optimising queries written in the vanilla syntax (beyond nested-loop performance) would be practically infeasible, requiring an engine that can prove equivalence between the distance metrics and semantics for which similarity join algorithms are optimised and the plethora of ways in which they can be expressed in vanilla syntax. We rather propose to make similarity joins for multidimensional distances a first class feature, with dedicated syntax and physical operators offering sub-quadratic performance.

3.3 Semantics

Pérez et al [22] define the semantics of SPARQL operators in terms of their evaluation over an RDF graph, which results in a set of *solution mappings*. We follow their formulation and define the semantics of a similarity join in terms of its evaluation. Letting $\mathbf{V}$, $\mathbf{I}$, $\mathbf{L}$ and $\mathbf{B}$ denote the set of all variables, IRIs, literals and blank nodes, respectively, then a solution mapping is a partial mapping $\mu : \mathbf{V} \to \mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$ defined for a set of variables called its *domain*, denoted $\mathrm{dom}(\mu)$. We say that two mappings μ_1, μ_2 are compatible, denoted $\mu_1 \sim \mu_2$, if and only if for all $v \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ it holds that $\mu_1(v) = \mu_2(v)$. Now we can define the following core operators on sets of mappings:

$$X \bowtie Y \coloneqq \{\mu_1 \cup \mu_2 \mid \mu_1 \in X \wedge \mu_2 \in Y \wedge \mu_1 \sim \mu_2\}$$

$$X \cup Y \coloneqq \{\mu \mid \mu \in X \vee \mu \in Y\} \qquad \sigma_f(X) \coloneqq \{\mu \in X \mid f(\mu) = \text{true}\}$$

$$X \setminus Y \coloneqq \{\mu \in X \mid \nexists \mu' \in Y : \mu \sim \mu'\} \quad X ⟕ Y \coloneqq (X \bowtie Y) \cup (X \setminus Y)$$

where f denotes a filter condition that returns true, false or error for a mapping.

The *similarity join expression* parsed from the aforementioned syntax is defined as $\mathfrak{s} \coloneqq (\mathcal{V}, \delta, v, \phi)$, where $\mathcal{V} \subseteq \mathbf{V} \times \mathbf{V}$ contains pairs of variables to be compared; δ is a distance metric that accepts a set of pairs of RDF terms and returns a value in $[0, \infty)$ or an error (interpreted as ∞) for incomparable inputs; $v \in \mathbf{V}$ is a fresh variable to which distances will be bound; and $\phi \in \{\mathtt{rg}_r, \mathtt{nn}_k\}$ is a filter expression based on range or k-nn. Given two solution mappings $\mu_1 \sim \mu_2$, we denote by $[\![\mathcal{V}]\!]^{\mu_1}_{\mu_2}$ the set of pairs $\{((\mu_1 \cup \mu_2)(x), (\mu_1 \cup \mu_2)(y)) \mid (x, y) \in \mathcal{V}\}$ (note: $[\![\mathcal{V}]\!]^{\mu_1}_{\mu_2} = [\![\mathcal{V}]\!]^{\mu_2}_{\mu_1}$). We say that $[\mu_1, \ldots, \mu_n] \in X_1 \bowtie \ldots \bowtie X_n$ if and only if $\mu_1 \in X_1$, $\ldots$, $\mu_n \in X_n$, and $\mu_i \sim \mu_j$ (for $1 \leq i \leq n$, $1 \leq j \leq n$). We can also interpret $\mu = [\mu_1, \ldots, \mu_n]$ as the mapping $\bigcup_{i=1}^{n} \mu_i$. We denote by $X_{\sim\mu} \coloneqq \{\mu' \in X \mid \mu \sim \mu'\}$ the solution mappings of X compatible with μ. We define $\mathrm{udom}(X) = \bigcup_{\mu \in X} \mathrm{dom}(\mu)$ and $\mathrm{idom}(X) = \bigcap_{\mu \in X} \mathrm{dom}(\mu)$. Finally we use v/d to denote a mapping μ such that $\mathrm{dom}(\mu) = \{v\}$ and $\mu(v) = d$.

Definition 1. *Given two sets of solution mappings X and Y, we define the evaluation of range and k-nn similarity joins, respectively, as:*

$$\underset{(\mathcal{V},\delta,v,rg_r)}{X \bowtie Y} \coloneqq \{[\mu_1,\mu_2,\mu_v] \in X \bowtie Y \bowtie \{v/\delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2})\} \mid \mu_v(v) \le r\}$$

$$\underset{(\mathcal{V},\delta,v,nn_k)}{X \bowtie Y} \coloneqq \{[\mu_1,\mu_2,\mu_v] \in X \bowtie Y \bowtie \{v/\delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2})\} \mid \mu_v(v) \le \kappa^{\delta,k}_{\mu_1,Y}\}$$

when $v \notin \mathrm{udom}(X) \cup \mathrm{udom}(Y)$ or error otherwise, where:

$$\kappa^{\delta,k}_{\mu_1,Y} \coloneqq \min\left\{\delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2}) \;\middle|\; \mu_2 \in Y_{\sim\mu_1} \wedge |\{\mu_2' \in Y_{\sim\mu_1} \mid \delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2'}) < \delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2})\}| < k\right\}.$$

We return an error when $v \notin \mathrm{udom}(X) \cup \mathrm{udom}(Y)$ to emulate a similar behaviour to `BIND` in SPARQL. Per the definition of $\kappa^{\delta,k}_{\mu_1,Y}$, more than k results can be returned for μ_1 in the case of ties, which keeps the semantics deterministic.

We define bag semantics for similarity joins in the natural way, where the multiplicity of $\mu \in X \bowtie_{\mathfrak{s}} Y$ is defined to be the product of the multiplicities of the solutions $\mu_1 \in X$ and $\mu_2 \in Y$ that produce it.

3.4 Algebraic Properties

We now state some algebraic properties of the similarity join operators. We use $\bowtie_{\mathfrak{r}}$, $\bowtie_{\mathfrak{n}}$, $\bowtie_{\mathfrak{s}} \in \{\bowtie_{\mathfrak{r}}, \bowtie_{\mathfrak{n}}\}$ to denote range, k-nn and similarity joins, respectively.

Proposition 1 *$\bowtie_{\mathfrak{r}}$ is commutative and distributive over $\cup$.*

Proof. Assume $\mathfrak{r} = (\mathcal{V}, \delta, v, \mathtt{rg}_r)$. For any pair of sets of mappings X and Y:

$$\begin{aligned} X \bowtie_{\mathfrak{r}} Y &\coloneqq \{[\mu_1,\mu_2,\mu_v] \in X \bowtie Y \bowtie \{[v/\delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2})]\} \mid \mu_v(v) \le r\} \\ &\equiv \{[\mu_2,\mu_1,\mu_v] \in Y \bowtie X \bowtie \{[v/\delta(\llbracket\mathcal{V}\rrbracket^{\mu_2}_{\mu_1})]\} \mid \mu_v(v) \le r\} \equiv \quad Y \bowtie_{\mathfrak{r}} X \end{aligned}$$

Which proves the commutativity. For left-distributivity over $\cup$:

$$\begin{aligned} X \bowtie_{\mathfrak{r}} (Y \cup Z) &\coloneqq \{[\mu_1,\mu_2,\mu_v] \in X \bowtie (Y \cup Z) \bowtie \{v/\delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2})\} \mid \mu(v) \le r\} \\ &\equiv \{[\mu_1,\mu_2,\mu_v] \in X \bowtie Y \bowtie \{v/\delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2})\} \mid \mu(v) \le r\} \cup \\ &\qquad \{[\mu_1,\mu_2,\mu_v] \in X \bowtie Z \bowtie \{v/\delta(\llbracket\mathcal{V}\rrbracket^{\mu_1}_{\mu_2})\} \mid \mu(v) \le r\} \\ &\equiv (X \bowtie_{\mathfrak{r}} Y) \cup (X \bowtie_{\mathfrak{r}} Z) \end{aligned}$$

Commutativity and left-distributivity over $\cup$ imply distributivity over $\cup$. □

Proposition 2 *$\bowtie_{\mathfrak{n}}$ is not commutative nor distributive over $\cup$.*

Proof. As counterexamples for commutativity and distributivity, note that there exist sets of mappings X, Y, Z with $|X| = n$, $|Y| = |Z| = 2n$, $n \ge k$ such that:

- Commutativity: $|X \bowtie_{\mathfrak{n}} Y| = nk$ and $|Y \bowtie_{\mathfrak{n}} X| = 2nk$.
- Distributivity: $|X \bowtie_{\mathfrak{n}} (Y \cup Z)| = nk$ and $|(X \bowtie_{\mathfrak{n}} Y) \cup (X \bowtie_{\mathfrak{n}} Z)| = 2nk$. □

Proposition 3 $\bowtie_{\mathfrak{n}}$ *is right-distributive over* $\cup$.

Proof. Assume $\mathfrak{n} = (\mathcal{V}, \delta, v, \mathtt{nn}_k)$. We see that:

$$\begin{aligned}(X \cup Y) \bowtie_{\mathfrak{n}} Z &\coloneqq \{[\mu_1, \mu_2, \mu_v] \in (X \cup Y) \bowtie Z \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})\} \mid \mu_v(v) \leq \kappa^{\delta,k}_{\mu_1,Z}\} \\ &\equiv \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Z \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})\} \mid \mu_v(v) \leq \kappa^{\delta,k}_{\mu_1,Z}\} \cup \\ &\quad\ \{[\mu_1, \mu_2, \mu_v] \in Y \bowtie Z \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})\} \mid \mu_v(v) \leq \kappa^{\delta,k}_{\mu_1,Z}\} \\ &\equiv (X \bowtie_{\mathfrak{n}} Z) \cup (Y \bowtie_{\mathfrak{n}} Z)\end{aligned}$$ □

Proposition 4 $(X \bowtie_{\mathfrak{s}} Y) \bowtie_{\mathfrak{s}'} Z \not\equiv X \bowtie_{\mathfrak{s}} (Y \bowtie_{\mathfrak{s}'} Z)$ *holds.*

Proof. As a counter example, consider that $\mathfrak{s}$ and $\mathfrak{s}'$ bind distance variables v and v' respectively such that $v' \in \mathrm{udom}(X)$, $v' \notin \mathrm{udom}(Y) \cup \mathrm{udom}(Z)$ and $v \notin \mathrm{udom}(X) \cup \mathrm{udom}(Y) \cup \mathrm{udom}(Z)$. Now $(X \bowtie_{\mathfrak{s}} Y) \bowtie_{\mathfrak{s}'} Z$ returns an error as the left operand of $\bowtie_{\mathfrak{s}'}$ assigns v but $X \bowtie_{\mathfrak{s}} (Y \bowtie_{\mathfrak{s}'} Z)$ will not. □

Finally, we discuss how the defined operators relate to other key SPARQL operators. The condition in claim 3 is analogous to well-designed queries [22].

Proposition 5 *Let* $\mathfrak{s} = (\mathcal{V}, \delta, v, \phi)$*. If each mapping in* $X \bowtie Y$ *binds all variables in* $\mathcal{V}$ *and* $v \notin \mathrm{udom}(X) \cup \mathrm{udom}(Y) \cup \mathrm{udom}(Z)$*, then the following hold:*

1. $(X \bowtie_{\mathfrak{s}} Y) \bowtie Z \equiv (X \bowtie Z) \bowtie_{\mathfrak{s}} Y$
2. $(X \bowtie_{\mathfrak{s}} Y) \setminus Z \equiv (X \setminus Z) \bowtie_{\mathfrak{s}} Y$ *if* $\mathrm{udom}(Z) \cap (\mathrm{udom}(Y) - \mathrm{idom}(X)) = \emptyset$
3. $(X \bowtie_{\mathfrak{s}} Y) ⟕ Z \equiv (X ⟕ Z) \bowtie_{\mathfrak{s}} Y$ *if* $\mathrm{udom}(Z) \cap (\mathrm{udom}(Y) - \mathrm{idom}(X)) = \emptyset$
4. $\sigma_f(X \bowtie_{\mathfrak{s}} Y) \equiv \sigma_f(X) \bowtie_{\mathfrak{s}} Y$ *if* f *is scoped to* $\mathrm{idom}(X)$*.*

Proof. We prove each claim in the following:

1. The third step here is possible as ϕ does not rely on Z (per the assumptions).

$$\begin{aligned}(X \bowtie_{\mathfrak{s}} Y) \bowtie Z &\coloneqq \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Y \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})\} \mid \phi(\mu_v)\} \bowtie Z \\ &\equiv \{[\mu_1, \mu_1', \mu_2, \mu_v] \in X \bowtie Z \bowtie Y \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})\} \mid \phi(\mu_v)\} \\ &\equiv \{[\mu_1, \mu_2, \mu_v] \in (X \bowtie Z) \bowtie Y \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})\} \mid \phi(\mu_v)\} \\ &\equiv (X \bowtie Z) \bowtie_{\mathfrak{s}} Y\end{aligned}$$

2. For a mapping $\mu = [\mu_1, \mu_2]$ such that $\mathrm{udom}(Z) \cap (\mathrm{dom}(\mu_2) - \mathrm{dom}(\mu_1)) = \emptyset$, there does not exist $\mu' \in Z$ such that $\mu \sim \mu'$ if and only if there does not exist $\mu' \in Z$ such that $\mu_1 \sim \mu'$. Taking $\mu_1 \in X$ and $\mu_2 = [\mu_2', \mu_2''] \in Y \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})$ from $X \bowtie_{\mathfrak{s}} Y$, the result then holds per the given assumptions.
3. The second step here uses the previous two results. The third step uses the right-distributivity of $\bowtie_{\mathfrak{r}}$ and $\bowtie_{\mathfrak{n}}$ over $\cup$ proven in previous propositions.

$$\begin{aligned}(X \bowtie_{\mathfrak{s}} Y) ⟕ Z &\coloneqq ((X \bowtie_{\mathfrak{s}} Y) \bowtie Z) \cup ((X \bowtie_{\mathfrak{s}} Y) \setminus Z) \\ &\equiv ((X \bowtie Z) \bowtie_{\mathfrak{s}} Y) \cup ((X \setminus Z) \bowtie_{\mathfrak{s}} Y) \\ &\equiv ((X \bowtie Z) \cup (X \setminus Z)) \bowtie_{\mathfrak{s}} Y \qquad \equiv (X ⟕ Z) \bowtie_{\mathfrak{s}} Y\end{aligned}$$

4. For a mapping $\mu = [\mu_1, \mu_2]$ and filter f scoped to $\mathrm{dom}(\mu_1)$, $f(\mu)$ is true if and only if $f(\mu_1)$ is true. Taking $\mu_1 \in X$ and $\mu_2 = [\mu_2', \mu_2''] \in Y \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})\}$ from $X \bowtie_{\mathfrak{s}} Y$, the result then holds per the given assumptions. □

```
SELECT ?c1 ?c2 ?d WHERE {
  { ?c1 wdt:P31 wd:Q6256 ;                          # Countries
        wdt:P2250 ?lifex1 ; wdt:P2131 ?ngdp1 ;    # Life expectancy, Nominal GDP
        wdt:P4010 ?gdp1 ; wdt:P2219 ?growth1 ;    # GDP, GDP growth rate
        wdt:P1081 ?hdi1 ; wdt:P361 wd:Q12585 }    # HDI, Latin America
  SIMILARITY JOIN
    ON (?lifex1 ?ngdp1 ?gdp1 ?growth1 ?hdi1)
       (?lifex2 ?ngdp2 ?gdp2 ?growth2 ?hdi2)
    TOP 1 DISTANCE sim:manhattan AS ?d              # 1-nn using Manhattan
  { ?c2 wdt:P31 wd:Q6256 ;                          # Countries
        wdt:P2250 ?lifex2 ; wdt:P2131 ?ngdp2 ;    # Life expectancy, Nominal GDP
        wdt:P4010 ?gdp2 ; wdt:P2219 ?growth2 ;    # GDP, GDP growth rate
        wdt:P1081 ?hdi2 ; wdt:P30 wd:Q46 }}       # HDI, Europe
```

?c1	?c2	?d
`wd:Q419` [Peru]	`wd:Q218` [Romania]	0.129
`wd:Q298` [Chile]	`wd:Q45` [Portugal]	0.134
`wd:Q96` [Mexico]	`wd:Q43` [Turkey]	0.195

Fig. 1. Query for European countries most similar to Latin American countries in terms of a variety of economic indicators, with sample results

4 Use-Cases, Implementation and Optimisation

Having defined the syntax and semantics of the similarity join operator, we now illustrate how the operator can be used, implemented and optimised.

4.1 Use-Case Queries

To illustrate the use of similarity joins in SPARQL, we will now present three use-case queries, demonstrating different capabilities of the proposal. All three queries are based on real-world data from Wikidata [27] and IMGpedia [7].

Similar Countries: In Figure 1 we present a similarity query for Wikidata [27] that, for each Latin American country, will return the European country with the most similar welfare indicators to it, considering life expectancy, Gross Domestic Product (GDP), nominal GDP, GDP growth rate and Human Development Index (HDI).[1] The query performs a 1-nn similarity join between both sets of countries based on the Manhattan distance over the given dimensions. The figure also presents three sample pairs of results generated by the query (though not returned by the query, we add English labels for illustration purposes).

Similar Elections: In Figure 2, we present a more complex similarity query over Wikidata to find the four most similar elections to the 2017 German Federal Election in terms of the number of candidates, parties and ideologies involved. The query involves the use of aggregates and paths in the operand graph patterns of the similarity join. The figure also presents the results of the query.

[1] We use prefixes as defined in `http://prefix.cc`.

```
SELECT ?e2 ?c1 ?c2 ?p1 ?p2 ?d
WHERE {
  { SELECT (wd:Q15062956 AS ?e1)
      (COUNT(DISTINCT ?candidate) AS ?c1)
      (COUNT(DISTINCT ?party) AS ?p1)
      (COUNT(DISTINCT ?ideology) AS ?i1) WHERE {
        wd:Q15062956 wdt:P726 ?candidate .                       # candidates
        ?candidate wdt:P102 ?party . ?party wdt:P1387 ?ideology.}}# parties, ideologies
  SIMILARITY JOIN ON (?c1 ?p1 ?i1) (?c2 ?p2 ?i2)
    TOP 4 DISTANCE sim:manhattan AS ?d                           # 4-nn using Manhattan
  { SELECT ?e2
      (COUNT(DISTINCT ?candidate) AS ?c2)
      (COUNT(DISTINCT ?party) AS ?p2)
      (COUNT(DISTINCT ?ideology) AS ?i2) WHERE {
        ?e2 wdt:P31/wdt:P279* wd:Q40231 ; wdt:P726 ?candidate .   # elections, candidates
        ?candidate wdt:P102 ?party . ?party wdt:P1387 ?ideology . # parties and ideologies
    } GROUP BY ?e2 }}
```

?e2	?c1	?c2	?p1	?p2	?d
`wd:Q15062956` [2017 German Federal Election]	10	10	8	8	0.000
`wd:Q1348890` [2000 Russian Presidential Election]	10	10	8	7	0.220
`wd:Q1505420` [2004 Russian Presidential Election]	10	6	8	8	0.240
`wd:Q19818995` [2017 Saarland State Election]	10	7	8	7	0.293

Fig. 2. Query for elections similar to the 2017 German Federal Election in terms of number of candidates, parties and ideologies participating, with results

Similar Images: Figure 3 presents a similarity query over IMGpedia [7]: a multimedia Linked Dataset. The query retrieves images of the Capitol Building in the US, and computes a 3-nn similarity join for images of cathedrals based on a precomputed Histogram of Oriented Gradients (HOG) descriptor, which extracts the distribution of edge directions of an image. The figure further includes a sample of results for two images of the Capitol Building, showing for each, the three most similar images of cathedrals that are found based on edge directions.

4.2 Implementation

The implementation of the system extends ARQ – the SPARQL engine of Apache Jena[2] – which indexes an RDF dataset and receives as input a (similarity) query in the syntax discussed in Section 3.2. The steps of the evaluation of an extended SPARQL query follow a standard flow, namely PARSING, ALGEBRA OPTIMISATION, ALGEBRA EXECUTION, and RESULT ITERATION. The PARSING stage receives the query string defined by a user, and outputs the algebraic representation of the similarity query. Parsing is implemented by extending Jena's Parser through JavaCC[3], wherein the new keywords and syntax rules are defined. The ALGEBRA OPTIMISATION can then apply rewriting rules (discussed presently) over the query, further turning logical operators (e.g., knnsimjoin) into physical

[2] `http://jena.apache.org`
[3] `https://javacc.github.io/javacc/`

```
SELECT ?img1 ?img2 WHERE {
  { ?img1 imo:associatedWith wd:Q54109 .                           # Capitol Building
    ?vector1 a imo:HOG ; imo:describes ?img1 ; imo:value ?hog1 .}    # HOG descriptor
  SIMILARITY JOIN ON (?hog1) (?hog2) TOP 3 DISTANCE sim:manhattan AS ?d # 3nn w/ Manhattan
  { ?cathedral wdt:P31 wd:Q2977 .                                    # Cathedrals
    ?img2 imo:associatedWith ?cathedral .
    ?vector2 a imo:HOG ; imo:describes ?img2 ; imo:value ?hog2 .}}   # HOG descriptor
```

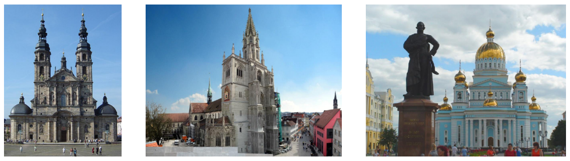
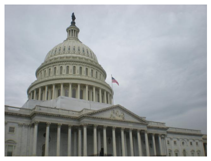
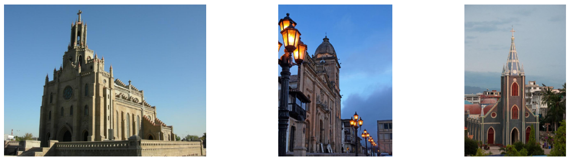
?img1	?img2 (*grouped*)
	
	

Fig. 3. Query for the three images of cathedrals most similar to each image of the US Capitol Building in terms of the HOG visual descriptor

operators (e.g., nestedloop). Next, Algebra Execution begins to evaluate the physical operators, with low-level triple/quad patterns and path expression operators feeding higher-level operators. Finally, Result Iteration streams the final results from the evaluation of the top-level physical operator. All physical similarity-join operators follow the same lazy evaluation strategy used for the existing join operators in Jena.

4.3 Similarity Join Optimisation and Query Planning

The Parsing phase will output either a knnsimjoin or rangesimjoin logical operator for the similarity join. The Algebra Optimisation phase must then select an algorithm with which to instantiate these operators. As previously discussed, for a similarity join $X \bowtie_{\mathfrak{s}} Y$, the naive strategy of computing a nested-loop join will require $|X| \cdot |Y|$ distance computations. The algorithms we include with our implementation of similarity joins are: nested loops, vp-Trees[4] and QJ for range queries; and nested loops, vp-Trees and FLANN[5] for k-nn.

Nested loops constitute a baseline for evaluating similarity joins without optimisation, as would be applied for similarity queries written in vanilla SPARQL

[4] We use the library provided by Chambers at `https://github.com/jchambers/jvptree`.

[5] We use the Java implementation provided by Stavrev at `https://gitlab.com/jadro-ai-public/flann-java-port.git`.

syntax or in iSPARQL [14]. On the other hand, QJ is used in DBSimJoin [25], and thus we also include it as a baseline measure, although it does not support k-nn, which we previously argued to be an important feature in this setting.

In initial experiments we noted that the results of similarity queries as defined herein sometimes gave unintuitive results when the magnitude of values in one dimension was naturally much greater or much smaller than that of other dimensions. Taking the query of Figure 1, for example, while the values for GDP tends to be in the order of billions or trillions, the values for HDI fall in $[0, 1]$; as defined, the HDI dimension would have a negligible effect on the results of the similarity join. To address this, we apply pre-normalisation of each dimension such that the values fall in the range $[0, 1]$.

Aside from adopting efficient similarity join algorithms, we can further optimise evaluation by applying query planning techniques over the query as a whole: given an input similarity query, we can try to find an equivalent, more efficient plan through query rewriting rules. While Jena implements techniques for optimising query plans in non-similarity graph patterns, we further explored both rewriting rules and caching techniques to try to further optimise the overall query plan. However, the techniques we have implemented had negligible or negative effect on runtimes, yielding a negative result, because of which further optimisations in this direction are left as future work.

Regarding rewriting rules, in fact there is little opportunity for such rules to optimise queries with similarity joins for two main reasons. One reason is that since similarity joins are a relatively expensive operation, optimally the results returned from the sub-operands should be as small as possible at the moment of evaluation; this often negates the benefit of delaying operations until after the similarity join. Another reason is that for k-nn similarity joins, the operation is not commutative, nor associative (see Section 3.3) preventing join reordering. The most promising rewritings from a performance perspective relate to properties 2 and 4 of Proposition 5, as they reduce the size of the similarity join by first applying the negation or filter, respectively, on the left operand; however, we did not encounter such cases (we further believe that queries are more likely to be written by users in the more optimal form).

Since the operands of similarity joins often have significant overlap in terms of triple patterns, an avenue for optimisation is evaluating this overlap once, reusing the results across both sub-operands. As an example, for the query in Figure 1, both operands have the same graph pattern except the last triple pattern, which distinguishes European and Latin American countries. We implemented such a technique, which we expected would reduce the runtime by avoiding redundant computation. However, experiments contradicted this expectation, where similar or slightly worse performance was found, for the following reasons. First, the overlap may be less selective than the non-overlapping patterns, making it disadvantageous to evaluate first; conversely, if we conservatively identify the maximal overlap maintaining the original selectivity-based ordering, the overlap will often be negligible. Second, even in favourable examples where a significant and selective overlap was present, no effect was seen due to lower-level caching.

5 Evaluation

We now present our evaluation, comparing the performance of different physical operators for similarity joins in the context of increasingly complex SPARQL queries, as well as a comparison with the baseline system DBSimJoin. We conduct experiments with respect to two benchmarks: the first is a benchmark we propose for Wikidata, while the second is an existing benchmark used by DBSimJoin based on colour information. All experiments were run on a Windows 10 machine with a 4-core Intel i7-7700 processor @2.80GHz and 16GB RAM. We provide code, queries, etc., online: `https://github.com/scferrada/jenasj`.

5.1 Wikidata: *k*-nn Self-Similarity Queries

In order to compare the relative performance of the three similarity join algorithms implemented: nested loop, vp-Trees and FLANN, we present performance results for a set of self-similarity join queries extracted from Wikidata[6]. To arrive at these queries, we begin with some data exploration. Specifically, from the dump, we compute the *ordinal/numeric characteristic sets* [18] by: (1) filtering triples with properties that do not take numeric datatype values, or that take non-ordinal numeric datatype values (e.g., Universal Identifiers); (2) extracting the *characteristic sets* of the graph [18] along with their cardinalities, where, specifically, for each subject in the resulting graph, we extract the set of properties for which it is defined, and for each such set, we compute for how many subjects it is defined. Finally, we generate k-nn self-similarity join queries from 1,000 ordinal/numeric characteristic sets containing more than 3 properties that were defined for more than 500 subjects. The values of k used are $1, 4, 8$. The joins were executed several times per algorithm to make a better estimation of the execution time, since vp-Trees and FLANN present randomised stages; we then report average runtimes over the runs.

Figure 4 presents the average execution time for differing *numbers of entities*, defined to be the cardinality of the solutions $|X|$ input to the self-similarity join $X \bowtie_{\mathtt{n}} X'$, where X' rewrites the variables of X in a one-to-one manner. Highlighting that the y-axis is presented in log scale, we initially remark that the value of k appears to have an effect on the execution time roughly comparable with the associated increase in results that can be expected. We can see a general trend that as the number of entities in the input initially increases, so too do the execution times. Comparing the algorithms, we see significant differences in performance depending on the similarity join algorithm, where (as expected) the nested loop strategy performs poorly. On the other hand, vp-Trees and FLANN are competitive with each other, showing similar performance; both see less sharp increases in time as the number of input entities increases. More specifically, FLANN is faster in 54.7% of the queries; however, we remark that, unlike vp-Trees, FLANN computes an approximation of the real result where, in these experiments, it gave 98% precision overall.

[6] We use a truthy dump available in February, 2020.

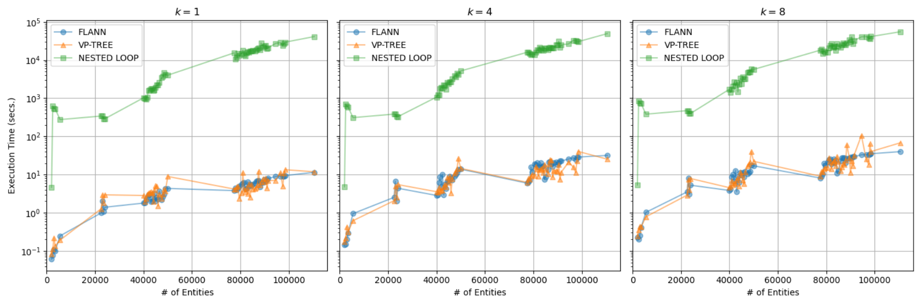


Fig. 4. Average execution time by number of input solutions (entities) for $k = 1, 4, 8$

In terms of absolute times, we find that both FLANN and vp-Trees can compute self-similarity joins $X \bowtie_{\mathfrak{n}} X'$ where $|X| < 20000$ and $1 \leq k \leq 8$ in less than one second (nested loops can take minutes on such queries), and can compute most of the queries analysed ($|X| < 120000$) within 10 seconds for $k = 1$, 40 seconds for $k = 4$, and 100 seconds for $k = 8$.

5.2 Corel Colour Moments: Range Similarity Queries

We compare our system with the closest found in literature, DBSimJoin [25], a PostgreSQL extension that supports range-based similarity joins in metric spaces implementing the QJ algorithm. As DBSimJoin only supports range queries, we compare it with the vp-Tree implementation in Jena (Jena-vp). The DBSimJoin system was originally tested with synthetic datasets and with the Corel Colour Moments (CCM) dataset, which consists of 68,040 records of images, each described with 9 dimensions representing the mean, standard deviation and skewness for the colours of pixels in the image. For CCM, the DBSimJoin paper only reports the effect of the number of QJ pivots on the execution time when $r = 1\%$ of the maximum possible distance in the space [25]. We compare the DBSimJoin implementation with our system for more general performance metrics, using CCM. We converted the CCM dataset to an RDF graph using a direct mapping.

To find suitable distances for the query, we compute a 1-nn self-similarity join on the data, where we take the maximum of minimum distances in the result: 5.9; this distance ensures that each object is paired with at least another object in a range-based query. We compare the runtime of both systems with increasing values of r, using $r = 5.9$ as an upper bound.

Table 1 presents the results: the execution time of DBSimJoin grows rapidly with r because of the quick growth of the size of the window partitions in the QJ algorithm. DBSimJoin crashes with $r \geq 0.4$ so we include results between 0.3 and 0.4 to illustrate the growth in runtime. The runtime of Jena-vp increases slowly with r, where more tree branches need to be visited as the result-set size increases up to almost $4 \cdot 10^8$. Our implementation does not crash with massive results because it fetches the bindings lazily: it obtains all pairs within distance

Table 1. Execution times in seconds for range similarity joins over the CCM Dataset.

Distance	# of Results	DBSimJoin (s)	Jena-vp (s)
0.01	68,462	47.22	6.92
0.1	68,498	84.00	7.45
0.3	72,920	775.96	9.63
0.39	92,444	1,341.86	11.17
0.4	121,826	–	11.30
1.0	4,233,806	–	35.01
5.9	395,543,225	–	1,867.86

r for a single object and returns them one by one, only computing the next batch when it runs out of pairs; on the other hand, QJ starts piling up window partitions that replicate the data at a high rate, thus causing it to crash.

Considering range-based similarity joins, these results indicate that our system outperforms DBSimJoin in terms of efficiency (our system is faster) and scalability (our system can handle larger amounts of results).

6 Conclusions

Motivated by the fact that users of knowledge graphs such as Wikidata are sometimes interested in finding similar results rather than crisp results, we have proposed and evaluated an extension of the SPARQL query language with multidimensional similarity joins. Applying similarity joins in the RDF/SPARQL setting implies unique challenges in terms of requiring dimension-agnostic methods (including dimensions that can be computed by the query), as well as data-related issues such as varying magnitudes amongst attributes. We thus present a novel syntax and semantics for multidimensional similarity joins in SPARQL, an implementation based on Apache Jena, a selection of optimised physical operators for such joins, along with use-case queries to illustrate the extension.

We evaluate three different strategies for implementing nearest neighbour joins: a brute-force method (nested loops), an online index-based approach (vp-Trees), and an approximation-based approach (FLANN). Of these, nested loops and vp-Trees can also be applied for range queries. Our experiments show that of these alternatives, vp-Trees emerge as a clear winner, being an exact algorithm that supports both k-nn and range similarity joins, as well as mostly outperforming the other algorithms tested. Our implementation with vp-Trees (and FLANN) outperforms the brute-force nested loop approach – which is how multidimensional distances expressed in vanilla SPARQL queries or queries in iSPARQL are evaluated by default – by orders of magnitude. Compared with the only other system we are aware of that implements multidimensional similarity joins as part of a database query language – DBSimJoin – our approach can handle k-nn queries, as well as much larger distances and result sizes.

Based on our results, we believe that similarity joins are an interesting direction to explore for SPARQL, where they could be used in applications such as entity matching, link prediction, recommendations, multimedia retrieval, query relaxation, etc.; they would also provide novel query functionality for users of SPARQL endpoints, allowing them to find entities in a knowledge graph that are similar within a metric space defined in the query by the user themselves.

Acknowledgements This work was partly funded by the Millennium Institute for Foundational Research on Data, by FONDECYT Grant No. 1181896 and by CONICYT Grant No. 21170616. We thank the reviewers for their feedback.

References

1. Battle, R., Kolas, D.: Enabling the geospatial Semantic Web with Parliament and GeoSPARQL. Semantic Web **3**(4), 355–370 (2012)
2. Belleau, F., Nolin, M.A., Tourigny, N., Rigault, P., Morissette, J.: Bio2RDF: towards a mashup to build bioinformatics knowledge systems. Journal of Biomedical Informatics **41**(5), 706–716 (2008)
3. Böhm, C., Braunmüller, B., Krebs, F., Kriegel, H.P.: Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In: SIGMOD Record. vol. 30, pp. 379–388. ACM (2001)
4. Böhm, C., Krebs, F.: Supporting KDD applications by the k-nearest neighbor join. In: International Conference on Database and Expert Systems Applications (DEXA). pp. 504–516. Springer (2003)
5. Dittrich, J.P., Seeger, B.: GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In: Special Interest Group on Knowledge Discovery in Data (SIGKDD). pp. 47–56. ACM (2001)
6. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: D-index: Distance searching index for metric data sets. Multimedia Tools and Applications **21**(1), 9–33 (2003)
7. Ferrada, S., Bustos, B., Hogan, A.: IMGpedia: a linked dataset with content-based analysis of Wikimedia images. In: International Semantic Web Conference (ISWC). pp. 84–93. Springer (2017)
8. Galkin, M., Vidal, M., Auer, S.: Towards a Multi-way Similarity Join Operator. In: New Trends in Databases and Information Systems (ADBIS). pp. 267–274. Springer (2017)
9. Giacinto, G.: A Nearest-neighbor Approach to Relevance Feedback in Content Based Image Retrieval. In: International Conference on Image and Video Retrieval (CIVR). pp. 456–463. ACM, New York, NY, USA (2007)
10. Guerraoui, R., Kermarrec, A., Ruas, O., Taïani, F.: Fingerprinting Big Data: The Case of KNN Graph Construction. In: International Conference on Data Engineering (ICDE). pp. 1738–1741 (April 2019)
11. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C Recommendation (Mar 2013), `https://www.w3.org/TR/sparql11-query/`
12. Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards Fuzzy Query-Relaxation for RDF. In: Extended Semantic Web Conference (ESWC). pp. 687–702 (2012)
13. Jacox, E.H., Samet, H.: Metric space similarity joins. ACM TODS **33**(2), 7 (2008)
14. Kiefer, C., Bernstein, A., Stocker, M.: The fundamentals of iSPARQL: a virtual triple approach for similarity-based Semantic Web tasks. In: International Semantic Web Conference (ISWC), pp. 295–309. Springer (2007)

15. Li, H., Zhang, X., Wang, S.: Reduce Pruning Cost to Accelerate Multimedia kNN Search over MBRs Based Index Structures. In: 2011 Third International Conference on Multimedia Information Networking and Security. pp. 55–59 (Nov 2011)
16. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISSAPP). pp. 331–340. INSTICC Press (2009)
17. Navarro, G.: Analyzing metric space indexes: What for? In: International Conference on Similarity Search and Applications (SISAP). pp. 3–10. IEEE Computer Society, Washington, DC, USA (2009)
18. Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: International Conference on Data Engineering (ICDE). pp. 984–994 (2011)
19. Ngomo, A.N., Auer, S.: LIMES – A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 2312–2317 (2011)
20. Oldakowski, R., Bizer, C.: SemMF: A Framework for Calculating Semantic Similarity of Objects Represented as RDF Graphs. Poster at ISWC (2005)
21. Paredes, R., Reyes, N.: Solving similarity joins and range queries in metric spaces with the list of twin clusters. Journal of Discrete Algorithms **7**(1), 18–35 (2009)
22. Pérez, J., Arenas, M., Gutiérrez, C.: Semantics and complexity of SPARQL. ACM TODS **34**(3), 16:1–16:45 (2009)
23. Petrova, A., Sherkhonov, E., Grau, B.C., Horrocks, I.: Entity Comparison in RDF Graphs. In: International Semantic Web Conference (ISWC). pp. 526–541. Springer (2017)
24. Sherif, M.A., Ngomo, A.N.: A systematic survey of point set distance measures for link discovery. Semantic Web **9**(5), 589–604 (2018)
25. Silva, Y.N., Pearson, S.S., Cheney, J.A.: Database similarity join for metric spaces. In: International Conference on Similarity Search and Applications (SISAP). pp. 266–279. Springer (2013)
26. Volz, J., Bizer, C., Gaedke, M., Kobilarov, G.: Discovering and Maintaining Links on the Web of Data. In: International Semantic Web Conference (ISWC). pp. 650–665. Springer (2009)
27. Vrandečić, D., Krötzsch, M.: Wikidata: A Free Collaborative Knowledgebase. Comm. ACM **57**, 78–85 (2014)
28. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Symposium on Discrete Algorithms (SODA). vol. 93, pp. 311–321 (1993)
29. Zhai, X., Huang, L., Xiao, Z.: Geo-spatial query based on extended SPARQL. In: International Conference on Geoinformatics (GEOINFORMATICS). pp. 1–4. IEEE (2010)