Similarity Joins and Clustering for SPARQL

Sebastián Ferrada [a,*], Benjamin Bustos [b] and Aidan Hogan [b]
[a] *Dept. of Computer and Information Science (IDA), Linköping University, Linköping, Sweden*
E-mail: sebastian.ferrada@liu.se
[b] *Millenium Institute for Foundational Research on Data, Department of Computer Science, University of Chile, Santiago, Chile*
E-mails: bebustos@dcc.uchile.cl, ahogan@dcc.uchile.cl

Abstract. The SPARQL standard provides operators to retrieve exact matches on data, such as graph patterns, filters and grouping. This work proposes and evaluates two new algebraic operators for SPARQL 1.1 that return similarity-based results instead of exact results. First, a similarity join operator is presented, which brings together similar mappings from two sets of solution mappings. Second, a clustering solution modifier is introduced, which instead of grouping solution mappings according to exact values, brings them together by using similarity criteria. For both cases, a variety of algorithms are proposed and analysed, and use-case queries that showcase the relevance and usefulness of the novel operators are presented. For similarity joins, experimental results are provided by comparing different physical operators over a set of real world queries, as well as comparing our implementation to the closest work found in the literature, DBSimJoin, a PostgreSQL extension that supports similarity joins. For clustering, synthetic queries are designed in order to measure the performance of the different algorithms implemented.

Keywords: Similarity Joins, Clustering, SPARQL

1. Introduction

Query languages such as SPARQL offer users the ability to match data using precise criteria. The results of evaluating the query contain *exact* matches that satisfy the specified criteria. This is useful, for example, to query for places with a population of more than one million (filters), for the names of famous musicians that fought in World War I (joins), or for the average life expectancy over all countries of each continent (grouping and aggregation). However, users may also require results that are based on imprecise criteria, such as similarity; for instance, obtaining countries with a population similar to Italy (similarity search), pairs of similar planetary systems from two different galaxies (similarity join), or groups of similar chemical elements based on certain properties (clustering). The potential applications for efficient similarity-based queries in SPARQL are numerous, including: entity comparison and linking [1, 2], multimedia retrieval [3, 4], similarity graph management [5, 6], pattern recognition [7], query relaxation [8], as well as domain-specific use-cases, such as protein similarity queries [9].

A similarity join $\bowtie_{\mathfrak{s}}$ operates over two sets of objects X, Y, bringing together all pairs $(x, y) \in X \times Y$ such that y is similar to x according to the given similarity criteria $\mathfrak{s}$ (which may specify a distance measure, a threshold, the attributes to compare, etc.). Clustering operates over a single set of objects and partitions them into groups, called clusters, such that within a cluster, all members are similar to each other with respect to a given similarity criteria.

Similarity is usually measured through distance functions defined over a metric or vector space, where two objects are as similar as they are close in the given space [10]. Distances are usually defined over real, d-dimensional vector

*Corresponding author. E-mail: sebastian.ferrada@liu.se.

spaces (e.g., L_p distances); however, there are distances to compare strings (e.g., edit distance), sets (e.g., Jaccard distance), and many other types of objects. Two main types of similarity criteria are distinguished: range-based and nearest neighbours. Range-based similarity queries require a distance threshold r and assume that two objects $x \in X, y \in Y$ are similar if the distance among them is at most r. Nearest neighbour similarity queries require a natural k and state that x and y are similar if there are fewer than k other elements in Y with lower distance to x.

It must be noted that *some* similarity joins could be expressed in SPARQL 1.1; for example, a range-based similarity join that uses the Manhattan distance might be expressed using subqueries, numeric operations and filters. However, general similarity queries over metric spaces cannot be expressed using SPARQL, and those that can be expressed will typically be evaluated by computing all pair-wise distances and applying filters to each one, since SPARQL engines are not able to produce efficient planning and optimisations for such queries. Although several techniques for the efficient evaluation of similarity joins in metric spaces are available [11–14], the appropriate techniques depend on the similarity criteria to be used, where in order to efficiently compile vanilla SPARQL queries that express a similarity join into optimised physical operators that include these techniques, the engine would need to prove the equivalence of the query to a particular similarity join, which is not clear to be decidable.

Some works have proposed extending SPARQL with similarity features. However, these works have either focused on (1) unidimensional similarity measures that consider similarity with respect to one attribute at a time [15, 16], or (2) domain-specific fixed-dimensional similarity measures, such as geographic distance [17, 18]. Other approaches rather pre-compute and index similarity scores as part of the RDF graphs [6, 8, 19] or support metric distance measures external to a query engine [2, 20].

In terms of clustering, Ławrynowycz [21] proposed an extension for SPARQL 1.0 that provides the solution modifier `CLUSTER BY`. This modifier receives several arguments, such as the clustering algorithm and its parameters; however their proposal was not implemented and it was not updated for SPARQL 1.1 so, for instance, it is unclear how the solution modifier works with the newly added modifiers such as `GROUP BY`. Thus, it is still unclear how the results of a SPARQL 1.1 query should be grouped with respect to similarity criteria, rather than exact criteria, and it is unclear what the performance cost of such an operator might be.

In this paper, two operators that enrich SPARQL with similarity-based queries are presented: similarity joins and clustering. Our work is, to our knowledge, the first to consider multidimensional similarity queries in the context of SPARQL, where the closest proposal to this one is DBSimJoin [22]: a PostgreSQL extension, which – though it lacks features arguably important for the RDF/SPARQL setting, namely nearest neighbours semantics – is considered as a baseline for experiments. Our work is also pioneer on the implementation and evaluation of clustering features on top of SPARQL with read-only capabilities (other works use SPARQL Update queries for some form of clustering [23]).

An example of what can be achieved by including similarity joins and clustering to SPARQL can be found in Figs. 2 and 13. In the former, we compute a similarity join between metals and non-metals that have similar atomic properties, such as boiling and melting points, mass, and electronegativity. In the later, we present a density-based clustering performed over stellar data used to create an HR-diagram, commonly used to visualise star evolution.

This paper is an extension of our previous work, where we proposed, defined, implemented and evaluated novel types of similarity joins in SPARQL [24]. Our primary novel contribution in this work involves defining, implementing and evaluating a clustering solution modifier. Other novelties include additional experiments for similarity joins, as well as new use-case queries and extended discussion throughout.

In Section 2 we present definitions regarding RDF, the SPARQL algebra and syntax, and similarity-based operations, and in Section 3, related work. The similarity join operator, its properties, and implementation are introduced in Section 4. Section 5 is dedicated to the clustering solution modifier: its definition, a proposed syntax, the implemented clustering algorithms, and use cases. Evaluation of the similarity join operator and the clustering solution modifier is presented and discussed in Section 6. Finally, Section 7 presents our concluding remarks about this work.

2. Preliminaries

In this section, the relevant definitions and notation that is used throughout this paper regarding RDF, SPARQL and Similarity Operations are presented.

2.1. RDF and SPARQL

RDF is the standard graph-based data model for the Semantic Web. *RDF terms* can be either IRIs (**I**), literal values (**L**) or blank nodes (**B**). An *RDF triple* is a tuple $(s, p, o) \in \mathbf{IB} \times \mathbf{I} \times \mathbf{IBL}$[1], where s is referred to as the *subject*, p as the *predicate*, and o as the *object*. A finite set of RDF triples makes an *RDF graph*.

SPARQL is the W3C standard query language for RDF [25]. The semantics of SPARQL operators are introduced by Pérez et al. [26], in terms of their evaluation over an RDF graph. The result of the evaluation of a SPARQL query is a set of *solution mappings*. Let **V** denote a set of query variables disjoint with **IBL**, a solution mapping is a partial function $\mu : \mathbf{V} \to \mathbf{IBL}$ defined for a set of variables, called its *domain*, $\mathrm{dom}(\mu)$. Two solution mappings μ_1, μ_2 are *compatible*, noted as $\mu_1 \sim \mu_2$, if for all $v \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2), \mu_1(v) = \mu_2(v)$. Letting X and Y denote sets of solution mappings, the SPARQL algebra is defined as follows:

$$
\begin{aligned}
X \bowtie Y &:= \{\mu_1 \cup \mu_2 \mid \mu_1 \in X \wedge \mu_2 \in Y \wedge \mu_1 \sim \mu_2\} \\
X \cup Y &:= \{\mu \mid \mu \in X \vee \mu \in Y\} \\
\sigma_f(X) &:= \{\mu \in X \mid f(\mu) = \text{true}\} \\
X \setminus Y &:= \{\mu \in X \mid @\,\mu' \in Y : \mu \sim \mu'\} \\
X \bowtie Y &:= (X \bowtie Y) \cup (X \setminus Y) \\
\pi_V(X) &:= \{\mu' \mid \exists \mu \in X : \mu' \subseteq \mu, \mathrm{dom}(\mu') = V \cap \mathrm{dom}(\mu)\} \\
\gamma_V(X) &:= \{(\mu, X') \mid \mu \in \pi_V(X),\ X' = \{\mu' \in X \mid \pi_V(\mu') = \mu\}\}
\end{aligned}
$$

where f is a filter condition that, given a solution mapping, returns true, false, unbound or an error.

We denote the evaluation of a query Q over an RDF graph G as $Q(G)$. Before defining $Q(G)$, first let $t = (s, p, o)$ denote a triple pattern, such that $(s, p, o) \in \mathbf{VIL} \times \mathbf{VI} \times \mathbf{VIBL}$; then by $vars(t)$ we denote the set of variables appearing in t and by $\mu(t)$ we denote the image of t under a solution mapping μ. Finally, we can define $Q(G)$ recursively as follows:

$$
\begin{aligned}
t(G) &:= \{\mu \mid \mu(t) \in G, \mathrm{dom}(\mu) = vars(t)\} \\
[Q_1 \text{ AND } Q_2](G) &:= Q_1(G) \bowtie Q_2(G) \\
[Q_1 \text{ UNION } Q_2](G) &:= Q_1(G) \cup Q_2(G) \\
[Q_1 \text{ OPTIONAL } Q_2](G) &:= Q_1(G) \bowtie Q_2(G) \\
\text{FILTER}_f(Q)(G) &:= \sigma_f(Q(G)) \\
\text{SELECT}_V(Q)(G) &:= \pi_V(Q(G)) \\
\text{GROUP}_V(Q)(G) &:= \gamma_V(Q(G))
\end{aligned}
$$

2.2. Similarity Search

In this section, we introduce the concept of similarity search over a universe of objects, as well as different similarity search operations.

Definition 2.1 (Similarity Search)**.** *Let* **U** *be the universe of objects, and* $\mathbf{D} \subseteq \mathbf{U}$ *a dataset. Given a query object* $q \in \mathbf{U}$*, similarity search obtains all elements* $x \in \mathbf{D}$ *such that x is similar to q.*

[1] Using that $\mathbf{IB} = \mathbf{I} \cup \mathbf{B}$

Mathematically, similarity is modelled as a score. Let $s_{x,y} \in [0,1]$ be the *similarity score* between objects x and y, where the higher the score, the more similar x and y are. Similarity scores are often measured in terms of distances within a given space; two objects being more similar if the distance among them is small. In this work, we consider a class of distance functions called metrics:

Definition 2.2 (Metric Distance)**.** *A distance function $\delta : \mathbf{U} \times \mathbf{U} \to \mathsf{R}$ is called a metric if the following properties hold for all $x, y, z \in \mathbf{U}$:*

1. $\delta(x,y) > 0$ *(positivity).*
2. $\delta(x,y) = 0 \iff x = y$.
3. $\delta(x,y) = \delta(y,x)$ *(symmetry).*
4. $\delta(x,y) \leq \delta(x,z) + \delta(z,y)$ *(triangle inequality).*

Examples of metric distances are the Euclidean distance over real vectors, the edit distance over strings, and the Jaccard distance over sets.

Metric distances induce a topology over the set they are defined upon:

Definition 2.3 (Metric Space)**.** *A* metric space *is a tuple $(\mathbf{U}, \delta)$ where $\mathbf{U} \neq \emptyset$, and δ is a metric distance function defined over pairs of elements of* $\mathbf{U}$.

There are two main types of similarity search queries in metric spaces: range and nearest neighbours queries.

Definition 2.4 (Range Similarity Search)**.** *Let $\mathbf{D} \subseteq \mathbf{U}$ be a dataset, and $q \in \mathbf{U}$ be a query object. The r-range similarity search, for $r \in \mathsf{R}$ is defined as:*

$$Q_{\mathsf{range}}(\mathbf{D}, q, r) = \{x \mid x \in \mathbf{D}, \delta(x,q) \leq r\}$$

Definition 2.5 (Nearest Neighbours Similarity Search)**.** *Let $\mathbf{D} \subseteq \mathbf{U}$ be a dataset, and $q \in \mathbf{U}$ be a query object. The k-nearest neighbours similarity search, for $k \in \mathsf{N}$ is defined as:*

$$Q_{\mathsf{nn}}(\mathbf{D}, q, k) = \{(x,y) \mid x \in X, y \in Y, \kappa_\delta(x, y, \mathbf{D}) \leq k\}$$

where $\kappa_\delta(x, y, \mathbf{D}) = |\{y' \in \mathbf{D} : x \neq y', \delta(x,y') < \delta(x,y)\}| + 1$ returns the rank of y w.r.t. its distance from x.

If instead of a single query object q we consider a set of query objects, we call this operation a similarity join. A similarity join is an operation defined over two *comparable* sets. Two sets are deemed comparable if a (metric) distance function can be defined over their Cartesian product.

Definition 2.6 (Similarity Join)**.** *Let X and Y subsets of a universe* $\mathbf{U}$*. Their* similarity join *is defined as follows:*

$$X \bowtie_{\mathfrak{s}} Y = \{(x,y) \mid x \in X, y \in Y, x \text{ is similar to } y \text{ according to } \mathfrak{s}\}$$

where $\mathfrak{s}$ can refer to a range-based or a k-nearest neighbours criteria, thus defining the following two operations:

$$X \bowtie_{\mathfrak{r}} Y := \{(x,y) \mid x \in X, y \in Y, \delta(x,y) \leq r\} \qquad X \bowtie_{\mathfrak{n}} Y := \{(x,y) \mid x \in X, y \in Y, \kappa_\delta(x,y,Y) \leq k\}$$

where δ is a distance function: $\mathbf{U} \times \mathbf{U} \to [0, \infty)$.

Throughout this work, two special cases of similarity join are distinguished: a) *query-by-example*, where $Y = \{y\}$ and b) *self-similarity join*, where $X = Y$.

3. Related Work

This section is divided in four parts: the first presents techniques to evaluate similarity joins; the second introduces works about the similarity operations available in query languages or database engines; the third is dedicated to clustering techniques; and the fourth to the inclusion of clustering in query languages.

3.1. Similarity Joins

The brute force method for computing a similarity join between X and Y is called 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 (or $|X| \cdot |Y|/2$ if it is a metric distance). For similarity joins over metric spaces, there are 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). Several tree structures have been proposed as indexes for optimising similarity queries [27–29]. Among such approaches, we can find *vantage-point Trees* (*vp-Trees*) [11], 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 query-by-example 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 [30], thus having an upper bound of $O(n^{2\alpha})$ for a similarity join. Other tree indexes, such as the D-Index [13] and the List of Twin Clusters [14], 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 [12] was designed to improve upon grid-based partition algorithms [31, 32]; 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 $O(n(1+w)^{d \log n e})$ 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 the similarity joins, trading the precision of the results for more efficient computation. FLANN [33] 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 based, for example, on a target precision, which can be traded-off to improve execution time.

3.2. 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 [22] 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 Quickjoin algorithm.

In the more specific context of the Semantic Web, a number of works have proposed online computation of similarity joins in the context of domain-specific measures. Zhai et. al [18] 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 [17] 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 [2], often string similarity measures over labels and descriptions [16].

Other approaches pre-materialise distance values that can then be incorporated into SPARQL queries. IMGpedia [6] 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 to those sought [8, 19].

Galvin et al. [34] 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 this work lies iSPARQL [15], 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 [15].

3.3. Clustering

Clustering is a process through which a set of objects can be automatically partitioned into groups – called *clusters* – such that the distance between objects of a cluster is minimised, and the distance between objects in different clusters is maximised. Clustering algorithms can be roughly divided into distance-based and density-based. A prime example of distance-based clustering algorithms is k-means [35], which first randomly selects k centroids and assigns each object to the cluster defined by the closest centroid; it then recomputes new centroids as the mean objects of each current cluster (which may not be a part of the set of objects), and reassigns the objects to the clusters defined by these new centroids until the centroids converge. The k-medoids algorithm is similar to k-means [36], but instead of centroids, it defines *medoids*: the element of a cluster such that its distance to every other element in the cluster is minimal. As for density-based algorithms, DBSCAN [37] groups together objects in dense regions (every object has several neighbours), separated by low density, sparse regions. DBSCAN also marks isolated objects as outliers, which are not included in any cluster.

3.4. Clustering in Databases

In databases, Ordonez and Omiecinski [38] provide an efficient disk-based k-means implementation for RDBMS using SQL `INSERT` and `UPDATE` queries. Also in SQL, Zhang and Huang [39] propose to extend the query language with a `CLUSTER BY` keyword, however, they do not provide an implementation. Li et al. [40] propose to extend SQL with fuzzy criteria for grouping and ordering, which they call clustering and ranking; they provide syntax and semantics, as well as a C++ implementation of the ranking and clustering algorithms based on so-called summary grids that provide a more time-efficient evaluation. Silva et al. [41] propose a similarity based `GROUP BY` that clusters the query results in three ways: 1) by defining in the query a maximum element separation or maximum cluster diameter, 2) by defining the centroids around which the data is clustered to, and 3) by defining a set of limit or threshold values. They provide a PostgreSQL implementation of these three clustering options.

Qi et al. [23] provide a k-means implementation for RDF data by using SPARQL Update queries. Ławrinowicz [21], proposed an extension to SPARQL 1.0, that introduces a `CLUSTER BY` solution modifier along with a clear syntax and semantics; however, they do not include an implementation nor evaluation, and it is not immediately clear how their proposal could be aligned with SPARQL 1.1 grouping and aggregates.

4. Similarity Joins in SPARQL

In this section, the similarity join operator for SPARQL is presented. Along with the algebraic definition of the operator and its properties, we will present options for its implementation and optimisation, and further introduce use-case queries to illustrate its application. We first define the following general criteria that a novel similarity join operator for SPARQL 1.1 should satisfy:

CLOSURE: the new operator should be freely combinable with other available operators.
EXTENSIBILITY: the similarity join should allow the user to specify the required dimensions and metrics to be used in the query, and define new metrics as needed.
ROBUSTNESS: the new operator should make as few assumptions as possible about the input or the data, regarding completeness, comparability, etc.
USABILITY: the new operator should be easy for users to adopt, offering high expressiveness while avoiding unnecessary verbosity.

With respect to CLOSURE, the similarity join will be defined in a manner analogous to other forms of joins that combine graph patterns in the `WHERE` clause of a SPARQL query; furthermore, the computed distance measure is allowed to be bound to a variable, facilitating its use beyond the similarity join (e.g., for solution modifiers such as `ORDER BY`, for aggregates such as `MAX`, `MIN`, `AVG`, for expressions within `FILTER` or `BIND`, etc). With respect to EXTENSIBILITY, rather than assume one metric, the type of distance metric used is explicit in the semantics and syntax, allowing other types of distance metrics to be used in future. Regarding ROBUSTNESS, the precedent of

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

Fig. 1. The extended production rules from SPARQL 1.1 to support similarity joins.

SPARQL's error-handling when dealing with incompatible types or unbound values is followed. Finally, regarding USABILITY, syntactic features for both range-based semantics and k-nn semantics are supported, noting that specifying particular distances for ranges can be unintuitive in abstract, high-dimensional metric spaces.

4.1. Syntax

To define the syntax for a similarity join, it is necessary to consider the syntax of other binary operators such as `OPTIONAL` or `MINUS` [25], as well as the required extra parameters: the variables representing the dimensions with respect to which the distance is to be computed, the distance function to be used, a fresh variable to bind the computed distances to, and the similarity parameter corresponding to a threshold distance (for range similarity joins) or to a threshold number of nearest neighbours (for k-nn similarity joins).

The syntax for the similarity join operator extends the SPARQL 1.1 EBNF grammar [25] by adding one production rule for `SimilarityGraphPattern`, and extending the rule for `GraphPatternNotTriples`. All other definitions are left unchanged. The exact changes to the grammar can be found in Fig. 1.

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`, similarly to `BIND`. The syntax may be extended in the future to provide further customisation, such as supporting different normalisation functions, or to define default parameters.

Depending on the metric, it could be possible, in principle, to express such queries as plain SPARQL 1.1 queries, by 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 would be 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. This is why we propose to make similarity joins for multidimensional distances a first class feature, with dedicated syntax and physical operators offering sub-quadratic performance in average/typical cases.

4.2. Semantics

Based on the SPARQL semantics presented in Section 2, and the syntax for the similarity joins from Section 4.1, we introduce the *similarity join expression*.

Definition 4.1 (Similarity Join Expression)**.** *A* similarity join expression *is a 4-tuple* $\mathfrak{s} := (\mathcal{V}, \delta, v, \phi)$*, 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 shall be bound; and* $\phi \in \{\mathtt{rg}_r, \mathtt{nn}_k\}$ *is a filter expression based on range or k-nn criteria.*

In the following, we present several definitions that build up to the formal specification of our similarity join operators. We begin with a definition that, given two compatible solutions from either side of the similarity join μ_1 and μ_2, and given a set of paired variables $\mathcal{V}$, produces a set of pairs of constants projected from the solutions over which distances can be computed.

Definition 4.2. Given two compatible solution mappings μ_1, μ_2, we define the set of mapping pairs to be compared in the similarity join as:

$$[\![V]\!]^{\mu_1}_{\mu_2} := \{((\mu_1 \cup \mu_2)(x), (\mu_1 \cup \mu_2)(y)) \mid (x, y) \in V\}$$

Note that per this definition $[\![V]\!]^{\mu_1}_{\mu_2} = [\![V]\!]^{\mu_2}_{\mu_1}$.

Next we define some useful notation for the result of a natural equi-join (applying a join with equality on shared variables, i.e., based on compatible solutions) that forms the basis of our similarity join operator.

Definition 4.3. We define the result of a natural equi-join as $[\mu_1, \ldots, \mu_n] \in X_1 / \ldots / X_n$ 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 also define $\mu = [\mu_1, \ldots, \mu_n]$ as the mapping $\bigcup_{i=1}^{n} \mu_i$.

We further define some notation for sets of compatible mappings.

Definition 4.4. Given a solution mapping μ and a set of solution mappings X, the subset of solution mappings X' of X that are compatible with μ is denoted by

$$X \sim \mu := \{\mu' \in X \mid \mu \sim \mu'\}.$$

Given a set of solutions, we now define notation for the union and intersection of the domains of the solutions (the sets of variables for which a solution is defined).

Definition 4.5. Given a set of solution mappings X, we define the union and intersection domains as:

$$\mathrm{udom}(X) := \bigcup_{\mu \in X} \mathrm{dom}(\mu) \qquad \mathrm{idom}(X) := \bigcap_{\mu \in X} \mathrm{dom}(\mu)$$

In order words, $\mathrm{udom}(X)$ is the set of variables that are defined by some solution mapping $\mu \in X$, while $\mathrm{idom}(X)$ is the set of variables that are defined for all solution mappings $\mu \in X$.

The next definition allows us to define a singleton solution mapping that maps a single variable to a single SPARQL term (similar to BIND). This will be used to bind the distance value to a specified (fresh) variable.

Definition 4.6. The expression $v \to d$ denotes a mapping such that $\mathrm{dom}(\mu) = \{v\}$ and $\mu(v) = d$.

With all these definitions in hand, we introduce the similarity join operators in SPARQL as follows:

Definition 4.7 (Range Similarity Joins in SPARQL). Given two sets of solution mappings X and Y, the evaluation of range similarity joins is defined as:

$$X \underset{(V, ?v, rg_r)}{/} Y := \{[\mu_1, \mu_2, \mu_v] \in X / Y / \{v \to (d \in [\![V]\!]^{\mu_1}_{\mu_2})\} \mid \mu_v(v) \leq r\}$$

when $v \in \mathrm{udom}(X) \cup \mathrm{udom}(Y)$ or error otherwise.

Definition 4.8 (k-nn Similarity Joins in SPARQL). Given two sets of solution mappings X and Y, the evaluation of k-nn similarity joins is defined as:

$$X \underset{(V, ?v, nn_k)}{/} Y := X \underset{(V, ?v, rg_{r'})}{/} Y$$

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

$$r' := \min\{d \in [\![V]\!]^{\mu_1}_{\mu_2} \mid \mu_2 \in Y \sim \mu_1 \text{ and } |\{\mu'_2 \in Y \sim \mu_1 \mid d' \in [\![V]\!]^{\mu_1}_{\mu'_2}, d' < d\}| > k\}$$

In case that $|Y \sim \mu_1| < k$, we say that $r' = \infty$.

The term r' intuitively refers to the distance that is required in order to return at least k nearest neighbours for μ_1 (or 1 if $|Y_1| < k$, effectively allowing the full set Y_1 to be returned). Per this definition, more than k results can be returned for μ_1 in the case of ties in distance, which keeps the semantics deterministic.

An error is returned when $v \in udom(X) \cap udom(Y)$ to emulate a similar behaviour to JOIN in SPARQL.

Bag semantics are defined for similarity joins in the natural way, where the multiplicity of $\mu \in X \,/_s\, Y$ is defined to be the product of the multiplicities of the solutions $\mu_1 \in X$ and $\mu_2 \in Y$ that produce it.

Following, we present several algebraic properties for the similarity join and its relationship with other SPARQL operators. The proofs for each of the claims can be found in Appendix A:

1. $/_r$ is commutative and distributive over $\cup$.
2. $/_n$ is not commutative nor is it distributive over $\cup$.
3. $/_n$ is right-distributive over $\setminus$.
4. $(X /_s Y) /_{s'} Z \neq X /_s (Y /_{s'} Z)$ holds, i.e., $/_s$ is not associative.
5. Let $s = (V, \cdot, v, \cdot)$. If each mapping in X / Y binds all variables in V and $v \in udom(X) \cap udom(Y) \cap udom(Z)$, then the following hold:
   (a) $(X /_s Y) / Z \equiv (X / Z) /_s Y$
   (b) $(X /_s Y) \bowtie Z \equiv (X \bowtie Z) /_s Y$ if $udom(Z) \setminus (udom(Y) \cup idom(X)) = \emptyset$;
   (c) $(X /_s Y) \setminus Z \equiv (X \setminus Z) /_s Y$ if $udom(Z) \setminus (udom(Y) \cup idom(X)) = \emptyset$;
   (d) $\sigma_f(X /_s Y) \equiv \sigma_f(X) /_s Y$ if f is scoped to $idom(X)$.

4.3. Implementation

The implementation of the similarity join operators 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 4.1. The steps of the evaluation of an extended SPARQL query follow a standard flow, namely Parsing, Algebra Optimisation, Algebra Execution, and Result Iteration. The implementation is available at https://github.com/scferrada/jena.

The Parsing stage receives the query string written by a user, and outputs the algebraic representation of the similarity query. Parsing is implemented by extending the Parser of Jena using JavaCC[3], where the new keywords and syntax rules are defined, as introduced in Section 4.1. The result of the Parsing stage is an abstract syntax tree of algebraic operators, e.g., `knnsimjoin(leftjoin(...), triplepattern(...))`.

In terms of physical operators to evaluate the similarity joins, initially we considered applying of ine indexing of literal values in the RDF graph in order to allow lookups of similar values. However, we discarded this option for the following two reasons:

- Indexing approaches typically consider a fixed metric space, with a fixed set of attributes specified for the distance measure (e.g., longitude and latitude). Conversely, in our scenario, the user can choose the attributes according to which similarity will be computed for a given query. In the setting of an RDF graph such as DBpedia or Wikidata, the user can choose from thousands of attributes. Indexing all possible combinations would lead to an exponential blow-up and would be prohibitively costly in terms of both time and space.
- Aside from selecting explicit attributes in the data – and per the SPARQL requirement – the user may often wish to compute attributes upon which distance can be measured. For example, when comparing countries, the user may wish to use the number of cities in each country (computed with COUNT), or the GDP per capita (dividing the GDP by the population). It would not be feasible to pre-index these computed attributes.

For this reason, we rather consider online physical operators that receive two sets of solution mappings and compute the similarity join. Some of these physical operators include indexing techniques, but these indexes are built online – for a specific query – over intermediate solutions rather than the RDF graph itself.

The similarity join physical operators apply a normalisation function to each of the dimensions of the space, in order to avoid differences in scale of these dimensions to affect disproportionately the similarity of two solution

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

Table 1
IRI pre xes used in the paper.

Pre x	IRI
wdt:	<http://www.wikidata.org/prop/direct/>
wd:	<http://www.wikidata.org/entity/>
sim:	<http://sj.dcc.uchile.cl/sim#>
imo:	<http://imgpedia.dcc.uchile.cl/ontology#>
tdb:	<https://triplydb.com/triply/iris/def/measure/>
stp:	<http://stars.dcc.uchile.cl/prop/>

mappings. For example, a similarity join query computing distances w.r.t. the dimensions human development index, HDI, (a number in $[0; 1]$) and population (ranging from thousands to even millions), as de ned, the HDI dimension would have a negligible effect on the results of the similarity join. However, as a future direction, we might provide special syntax to allow users to decide if the data should be automatically normalised or not.

Algebra Optimisation then applies static rewriting rules over the query, further turning logical operators (e.g., knnsimjoin) into physical operators. In terms of the physical operators, we implement three variants:

Nested Loopsinvolves a brute-force pairwise comparison, incurring quadratic cost for similarity joins. It computes exact results for both range and k-nn queries and is intended as a performance baseline.

vp-Trees is an index-based approach based on applying ball-cuts in the metric space. It computes exact results for both range and k-nn queries. Though the worst-case for similarity joins is still quadratic, depending on the distance distribution, dimensionality, etc., vp-Trees can approach linear performance.

FLANN is an ensemble of approximate methods for k-nn queries only, where the best approach is chosen based on the query, data, target precision, etc.

Algebra Execution begins to evaluate the physical operators chosen in the previous step, with low-level triple/quad patterns and path expression operators feeding higher-level operators. Finally, Result Iteration streams the nal 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.

Besides the implementation of sub-quadratic average-case similarity join physical operators, there are other avenues for query optimisation, namely: query rewriting and result caching. As for rewriting rules, there is little opportunity for applying such rules because of a) similarity joins are a relatively expensive operation to compute, thus the best option is to reduce the size of the outputs of the sub-operands as early as possible and therefore, making typical rewriting rules impractical (such as delaying lters and optionals); and b) we see in Section 4.2 that the k-nn similarity join operator is not commutative nor associative, further preventing rewriting such as join reordering.

4.4. Use-Case Queries

To illustrate the use of similarity joins in SPARQL, three use-case queries are presented, demonstrating different capabilities of the proposal. All three queries are based on real-world data from Wikidata [42] and IMGpedia [61]. All pre xes used in the queries of this paper are listed in Table 1.

Similar Chemicals Chemical elements contain several numeric/ordinal properties, such as boiling and melting point, mass, electronegativity and so on. In Fig. 2 we request for metals similar to non-metals in terms of those properties, along with sample results of the similar metal/non-metal query. Fig. 2 also presents a selection of results of the query, where we see that Selenium is most similar to Aluminium, and Hydrogen to Sodium.

Similar Elections: Fig. 3, presents a more complex similarity query over Wikidata to nd the most similar elections to the 2017 German Federal Election in terms of the number of candidates, parties and ideologies involved. The query uses aggregates and property paths in the operand graph patterns. Fig. 3 also presents the results of the query.

```
SELECT ?element1 ?element2 ?dist WHERE {
    {?element1 wdt:P31 wd:Q11344; #chemical elements
               wdt:P2102 ?boil1; wdt:P2101 ?melt1;
               wdt:P1108 ?eneg1;
               wdt:P2054 ?dens1; wdt:P2067 ?mass1.
               (wdt:P279+) wd:Q11426. # metals
    }
    SIMILARITY JOIN ON (?boil1 ?melt1 ?eneg1 ?dens1 ?mass1)
                       (?boil2 ?melt2 ?eneg2 ?dens2 ?mass2)
    TOP 1 DISTANCE sim:manhattan AS ?dist
    {?element2 wdt:P31 wd:Q11344;
               wdt:P2102 ?boil2; wdt:P2101 ?melt2;
               wdt:P1108 ?eneg2;
               wdt:P2054 ?dens2; wdt:P2067 ?mass2.
     MINUS {?element2 (wdt:P279+) wd:Q11426.}}} # non-metals
```

Element 1	Element 2	Distance
wd:Q876 [Se]	wd:Q663 [Al]	0.4644
wd:Q556 [H]	wd:Q658 [Na]	0.3107

Fig. 2. Query for non-metal chemical elements similar to metallic elements in terms of atomic properties.

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

Similar Images: Fig. 4 presents a similarity query over IMGpedia [6]: a multimedia Linked Dataset. The query retrieves images of the Entel Tower in Chile, and computes a 10-nn similarity join for images of towers based on a precomputed Histogram of Oriented Gradients (HOG) descriptor, which extracts the distribution of edge directions of an image. Note that this query uses a different distance funtion IRI than the previous two, since variables ?hog1 and ?hog2 are bound to string encodings of vectors, and thus function sim:manhattanvec deals with the parsing of such strings as well. The gure further includes a sample of results for an image of the Entel Tower (in index 0), showing for each similar image, the related Wikidata entity, its label, and the distance to the source image.

```
SELECT ?img2 ?tower ?dist WHERE {
  { ?img1 imo:associatedWith wd:1421270 .                          # Entel Tower
    ?vector1 a imo:HOG ;                                           # HOG descriptor
             imo:describes ?img1 ;
             imo:value ?hog1 .}
  SIMILARITY JOIN ON(?hog1) (?hog2) TOP10 DISTANCE sim:manhattanvec AS ?dist   # 10nn w/ Manhattan
  { ?tower wdt:P31/wdt:P279* wd:Q12518 .                           # Towers
    ?img2 imo:associatedWith ?tower .
    ?vector2 a imo:HOG ;
             imo:describes ?img2 ;
             imo:value ?hog2 .}}
```

Index	0	1	2	3	4	5
?img2						
?tower	wd:Q1421270	wd:Q635	wd:Q7793202	wd:Q7653429	wd:Q7011062	wd:Q4943563
Label	Torre Entel	Tower of Fourvière	Thomas Point Light	Swan Bells	New Presque Isle Light	Boon Island Light
?dist	0	22.282	24.224	24.439	25.052	25.911

Index	6	7	8	9	10
?img2					
?tower	wd:Q1413217	wd:Q7208057	wd:Q5250767	wd:Q6391315	wd:Q7092627
Label	i kov TV Tower	Point Judith Light	Deer Island Light	Kenosha North Pier Light	One Fathom Bank Lighthouse
?dist	25.984	26.397	26.850	26.940	27.016

Fig. 4. Query for the 10 images of towers most similar to each image of the Entel Tower in terms of the HOG visual descriptor

5. Clustering Solution Modi er

In this section, we analyse, implement and evaluate a CLUSTER BY solution modi er for SPARQL queries – inspired by the proposal of awrynowicz [21] – based on the aforementioned notions of similarity and distance. First, we discuss how this extension of SPARQL was initially conceived, how it can be formalised syntactically and semantically and how it interacts with the rest of the solution modi ers. We then discuss implementation details for the solution modi er, and present a prototype built on top of Apache Jena. Finally, use-case queries are presented.

As it has been argued before in this paper, users sometimes require non-exact matches over the data and instead wish for similar matches. This is also the case when using the solution modi er GROUP BY in SPARQL, which makes a partition of the resulting mappings of a query based on certain variable values that need to match exactly in order to assign a binding into a group. In SPARQL 1.1 [25] the algebra for GROUP BY is de ned as described in Section 2.

As presented, the GROUP BY modi er partitions the bindings into equivalence classes (the multisets) based on the different key values (the values of Ω_0). The similarity-based alternative for GROUP BY to perform clustering over the bindings, so that solution mappings that are similar (instead of equal) are assigned to the same cluster.

5.1. Syntax

In order to support clustering in SPARQL, we must rst de ne its syntax and semantics in a way that a user can express the clustering variables, and the algorithms and their required parameters. awrynowicz [21] proposes the following extension for SPARQL 1.0 [43], where the cluster identi er is bound AS a fresh variable (cf. Fig. 5).

```
SolutionModifier   ::= OrderClause? LimitOffsetClauses? ClusterByClause?
ClusterByClause    ::= 'CLUSTER' 'BY' Var+ 'AS' Var UsingClause?
UsingClause        ::= 'USING' IRIRef '(' (Params)? ')'?
Params             ::= TriplesBlock
```

Fig. 5. Syntactic extension to the SPARQL 1.0 grammar for clustering [21]

```
SolutionModifier   ::= ClusterByClause? GroupClause? HavingClause? OrderClause? LimitOffsetClauses?
ClusterByClause    ::= 'CLUSTER' 'BY' Var+ WithClause ClusterAlias
WithClause         ::= 'WITH' iriOrFunction
ClusterAlias       ::= 'AS' Var
```

Fig. 6. Syntactic extension to the SPARQL 1.1 grammar for clustering

The first thing that can be noticed from the definition is that, since the extension was conceived for SPARQL 1.0, it lacks the definition of the GROUP BY solution modifier. The definition of a clustering solution modifier for SPARQL 1.1 needs to take into account the existence of GROUP BY, and how it shall interact with CLUSTER BY.

Since clustering is the similarity-based version of grouping (which is based on exact matching), one alternative is to make the modifiers CLUSTER BY and GROUP BY be mutually exclusive in a query. However, the two modifiers can be made compatible by defining the clustering modifier as binding the cluster identifier of each mapping to a fresh variable, potentially allowing users to later group the results by the identifier value; in other words, CLUSTER BY extends the solution mappings with an extra variable bound to a cluster identifier, rather than directly grouping them. This decision is made since any new operation included into a query language should be as non-disruptive as possible, and to preclude the usage of GROUP BY when clustering is indeed disruptive.

Since the CLUSTER BY solution modifier is not mutually exclusive with GROUP BY, CLUSTER BY does not interact with the HAVING clause and the different aggregators, which can rather be computed after the clustering takes place. Conversely, in order to apply clustering with respect to values computed through aggregation functions, the aggregation must be computed in a subquery. Thus, following the SPARQL spec, the order of the solution modifiers including CLUSTER BY is clustering, grouping, having, ordering, projection, distinct, reduced, offset, and limit.

Pawrynowycz [21] makes use of the SPARQL Query Results Clustering Ontology (SQRCO) to define the clustering algorithm and its parameters as a set of triples in the query. However, at the time of writing, the ontology is not found on the Web and is not registered in prefix.cc. We instead take a syntax-based approach, using special keywords for the clustering configuration, similarly to what is done with the similarity join operator.

Our proposed extension of the SPARQL 1.1 syntax that includes CLUSTER BY is presented in Fig. 6. The extension adds a new production rule ClusterByClause, as an alternative in the SolutionModifier production rule. We make use of the Extension Function framework defined for SPARQL 1.1 to allow users to define the desired clustering algorithm and its parameters by using an iriOrFunction rule after the keyword WITH, thus allowing the function to be called without any parameters as well, which permits to define default values for each supported algorithm. Finally, we provide syntax to define the fresh variable on which the cluster identifier shall be bound AS. We permit only variables to be used as clustering attributes, but expressions could be supported in a future version. Nevertheless, expression could be indirectly used for clustering by using a BIND function.

An example of a SPARQL query that uses this extension can be appreciated in Fig. 7, where the countries available in Wikidata are clustered based on their life expectancy and economic growth. The query specifies that the clustering shall use k-means returning 4 clusters. Notice that we also obtain the 2-letter ISO code for each country (property wdt:P297) and we are able to project that variable even if it is not part of the CLUSTER BY variables.

5.2. Semantics

Given the SPARQL semantics presented in Section 2, as well as the discussed syntax for CLUSTER BY, we now propose the semantics for the CLUSTER BY solution modifier, per the following definition.

```
SELECT ?lifex ?growth ?code ?c WHERE {
    ?c1 wdt:P31 wd:Q6256 ; #countries
        wdt:P2250 ?lifex ;
        wdt:P2219 ?growth ;
        wdt:P297 ?code . }
CLUSTER BY ?lifex ?growth WITH sim:kmeans(4) AS ?c
```

Fig. 7. Example SPARQL query with the CLUSTER BY solution modifier

Definition 5.1. Given a multiset of mappings Ω, the clustering algorithm C, the variables $V = \{?v_1, \ldots, ?v_n\} \subseteq \text{dom}(\Omega)$ with respect to which the clustering is computed and a fresh variable $?c \in \mathcal{V}$ that stores the cluster identifier, the evaluation of the clustering solution modifier is defined as follows:

$$\text{eval}_{V,C,?c}(\Omega) := \{\mu[?c \to i] \mid i = C(\{\mu_V(\mu) \mid \mu \in \Omega\})[\text{index}(\mu_V(\mu))], \mu \in \Omega\}$$

where C is a clustering function that, given a multiset of solution mappings Ω, a solution mapping μ and a variable $?c$, returns a mapping $[?c \to i]$ where i is an RDF term used as the identifier of the cluster selected for μ by C. Concrete values for the identifiers are left to each specific algorithm, where a natural solution is to enumerate the clusters with integers. Note that the original solution mapping μ is being extended, not the projected version $\mu_V(\mu)$.

5.3. Implementation

Analogous to the similarity join operators, we implemented the CLUSTER BY solution modifier on top of Apache Jena's ARQ, where the code is publicly available in the following repository: https://github.com/scferrada/jena. The implementation supports three clustering algorithms: *k*-means, *k*-medoids, and DBSCAN. These algorithms are currently implemented in-memory. All three algorithms use the Manhattan distance as the dissimilarity measure. All three algorithms enumerate the clusters in no particular order using natural numbers starting from 1, and use said numbers as the identifiers of the clusters. For *k*-medoids we implemented the FastPAM1 algorithm proposed by Schubert and Rousseeuw [44] which is $O(k)$ faster than the naive algorithm and uses $O(k)$ extra memory.

Each algorithm requires different parameters, but the implementation offers default values for them, if the user does not wish to specify them. The parameters of each algorithm and the chosen default values are:

- k-means [35]: k, the number of clusters to be returned, default 3; n, the number of iterations to terminate the algorithm, default 10.
- k-medoids [36]: k, the number of clusters to be returned, default 3.
- DBSCAN [37]: ε, the distance to search for neighbours of an object, default 0; p, the minimum number of neighbours that an object needs to be considered part of a cluster, default 1.

DBSCAN is different from *k*-means and *k*-medoids in that it does not require the number of clusters to be specified, but also may mark some objects as outliers, not being in any cluster. To comply with the semantics of CLUSTER BY, it assigns all solution mappings marked as outliers to a special cluster (with identifier 1). Notice that the default parameters for DBSCAN result in all objects marked as outliers. The implementation of DBSCAN uses vp-Trees in order to improve performance, since for each element a range-based similarity search must be conducted.

5.4. Use-Cases

First, in Fig. 8 we display the result of the query presented in Fig. 7, that clusters the countries in Wikidata with respect to their life expectancy and economic growth rate. Some results were omitted to avoid clutter in the figure. Each point has a different colour depending on their assigned cluster (whose identifier is bound to variable ?c). It can be seen in this case that, since there are differences in the magnitudes of the values of life expectancy and economic growth rate, the largest values tend to weigh more in the distance value and thus, in this example, we see a stronger division along the vertical axis. To prevent that any given dimension has a disproportionate influence over the distance function, it is recommended to normalise each dimension before clustering. Additionally, we note that,

Fig. 8. Clusters of countries by growth and life expectancy

```
SELECT ?iris ?type ?plength ?pwidth ?c WHERE {
    ?iris a ?type;
          tdb:petalLength ?plength ;
          tdb:petalWidth  ?pwidth . }
CLUSTER BY ?plength ?pwidth WITH sim:kmedoids(3) AS ?c
```

Fig. 9. Extended SPARQL query to obtain three clusters of iris

unlike GROUP BY, we can keep and project any variables that are not part of the CLUSTER BY clause. In Fig. 8 we also present the 2-letter ISO code of each country (which is directly obtained from the query) annotating each point in the graph, so that the reader can know which countries belong to each cluster more easily.

Then, the Iris Dataset in RDF[4] is used to run a clustering query on petal length and width. In Fig. 9 the extended SPARQL query that divides the objects into three clusters using the k-medoids algorithm is presented. Fig. 10 shows the results of the clustering, where the green cluster is iris setosa, the blue cluster is mostly iris virginica, and the red cluster is mostly iris versicolor. Notice that the iris types can also be retrieved with the query.

Next, we present a Hertzsprung-Russell (HR) diagram, that is used in astronomy to classify stars into main sequence, giants, white dwarfs, etc. HR diagrams have the normalised B–V Colour (a measure of the temperature or "brightness" of the star) on the X-axis, and M_V (the absolute magnitude) on the Y-axis. To achieve this, we generate an RDF version of a real-world catalogue of stars[5]. In Fig. 11 a sample of the generated data can be found. The full RDF data is available at https://github.com/scferrada/starsrdfdata. Then, we cluster the stars by their absolute magnitude (M_V) and their B-V colour using DBSCAN, since the main sequence cluster is better obtained by density-based methods. We also filter in the query the stars that have a high parallax uncertainty (a measure of the chance that the distance of the star, and thus its magnitude, is incorrect), because these stars may have an unreliable magnitude measure and just add noise to the diagram. Since magnitude and colour have very different scales, we also apply min–max normalisation in two BIND clauses, and perform the clustering over the normalised data. The SPARQL query that represents this process can be found in Fig. 12. The plot of the result of the query can be seen

[4] Retrieved from https://triplydb.com/RosalinedeHaan/iris. Credit to Rosaline de Haan.

[5] The CSV data is available at http://burro.astr.cwru.edu/Academics/Astr221/HW/HW5/yaletrigplx.dat. The generated RDF data is available in the following repository: https://github.com/scferrada/starsrdfdata.

Fig. 10. Clusters obtained with Query 9

```
<http://stars.dcc.uchile.cl/star/2.00>  <http://stars.dcc.uchile.cl/prop/color>                1.04 .
<http://stars.dcc.uchile.cl/star/2.00>  <http://stars.dcc.uchile.cl/prop/absMagnitude>        0.261669 .
<http://stars.dcc.uchile.cl/star/2.00>  <http://stars.dcc.uchile.cl/prop/parallaxUncertainty> 149.0 .
```

Fig. 11. Extract of the RDF data for stars.

```
SELECT ?temp ?lum ?c WHERE {
    ?star  stp:color                ?temp ;
           stp:absMagnitude         ?lum ;
           stp:parallaxUncertainty  ?error .
    FILTER(?error < 75)
    BIND((?lum+1.4444)/(14.7121 + 1.4444)  AS ?lum2)
    BIND((?temp+0.17)/(1.88+0.17)  AS ?temp2)}
CLUSTER BY ?temp2 ?lum2 WITH sim:dbscan(0.05, 10)  AS ?c
```

Fig. 12. SPARQL query for clustering star data.

in Fig. 13, where the central purple cluster represents stars in the main sequence, the top right blue and sky blue clusters represent the giant and super giant stars, and the bottom-left red cluster contains the white dwarfs.

Finally, we showcase how the CLUSTER BY solution modifier can interact with GROUP BY in a SPARQL query. To do this, we modify the query from Fig. 7, which obtains the life expectancy and growth rate of the countries in Wikidata, to now GROUP BY cluster identifier and compute the average life expectancy and the minimum growth rate of each cluster. This query is presented in Fig. 14, and the results of its evaluation over Wikidata can be seen in Table 2 where cluster 1 corresponds to the dark blue cluster in Fig. 8, which includes Japan and Italy, and has the highest life expectancy and a minimum growth rate of 0.9%; cluster 2 is the pink cluster, which contains New Zealand and Chile; cluster 3 is the sky blue cluster, which includes Brazil and Bulgaria, and presenting the lowest growth rate at -18%; and cluster 4 is the red cluster, presenting the lowest life expectancy average.

6. Experimental Results

Experimental results for both the similarity join operator and the clustering solution modifier are presented in this section. Unless stated otherwise, experiments were run on a machine with Ubuntu 20, an 8-core Intel® Core™ i7-6700 CPU @3.4GHz, 16GB of RAM and a 931GB HDD. We provide the experiment code, queries, etc.,[6] online

[6] https://github.com/scferrada/SimilaritySPARQL

Fig. 13. Visual representation of the star clusters obtained with query from Fig. 12. Outliers not presented.

```
SELECT AVG(?lifex) MIN(?growth) ?c WHERE {
?c1 wdt:P31 wd:Q6256 ; #countries
    wdt:P2250 ?lifex ;
    wdt:P2219 ?growth .
}
CLUSTER BY ?lifex ?growth WITH sim:kmeans(4) AS ?c
GROUP BY ?c
```

Fig. 14. Example SPARQL query with the CLUSTER BY solution modifier and GROUP BY

Table 2
Results of query of Fig. 14

?c	AVG(?lifex)	MIN(?growth)
1	83.399	-5.42
2	78.696	-3.20
3	73.121	-18.00
4	61.909	-13.80

6.1. Similarity Join Evaluation

We compare 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 visual descriptors of images.

6.1.1. Wikidata: k-nn Self-Similarity Queries

Extending the performance experimentation presented in our previous work [24], we now present similar experiments using real SPARQL queries. We use a truthy version of the Wikidata RDF dump[7]. We start with SPARQL

[7]We use the dump from February 2020.

Fig. 15. Execution time for self-similarity join queries, for different values of k.

queries from the Wikidata query logs, and apply several filters: first, at least one numeric attribute must be projected in the SELECT clause; second, these numeric attributes must not be obtained OPTIONAL clause; third, they produce at least 8 solution mappings; and fourth, they are executed in less than 3 hours using Jena. We obtain 2,726 SPARQL queries. For each of these base queries, we write a self-similarity join version for our experiments.

We executed self-similarity joins with $k \in \{2, 4, 8\}$, and compare 3 different techniques: Nested Loops (NL), vp-Trees (VPT), and FLANN. We run the queries sequentially, to avoid caching. In Fig. 15 we present our findings, where we provide the execution time of the base query as a baseline for comparison. VPT perform the best in 2,841 out of 8,178 scenarios (34.7%), NL in 2,770 (33.9%), and FLANN in 2,567 (31.4%). NL tend to perform better when the number of solution mappings is relatively smaller, which makes sense, since building indices has an overhead cost. The mean set size of the cases where NL performs the best is 178 elements. FLANN tends to perform better, when the set is comparably larger (mean size of 2,155 elements). As VPT perform better in the majority of the times and provides an exact result it should be preferred for larger result sets, and NL for smaller result sets.

6.1.2. Corel Colour Moments: Range Similarity Queries

The system proposed in this work is compared with the closest found in literature, DBSimJoin [22], a PostgreSQL extension that supports range-based similarity joins in metric spaces implementing the Quickjoin (QJ) algorithm. As DBSimJoin only supports range queries, it is to be compared 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 (red, green and blue) of the pixels of the image. For CCM, the DBSimJoin paper only reports the effect of the number of QJ pivots on the execution time when $r = 10\%$ of the maximum possible distance in the space [22]. Here, the comparison between the systems takes into account more general performance metrics, using CCM. The CCM dataset was converted to an RDF graph using a direct mapping.

To find suitable distances, we first compute a 1-nn self-similarity join, where we take the maximum 1-nn distance in the result: 5.9; this distance ensures that each object has at least one neighbour in a range-based query with $r = 5.9$. Therefore, we compare the runtime of both systems with increasing values of $r \leq 5.9$ as an upper bound.

DBSimJoin is implemented and distributed an extension for PostgreSQL 8. When installed in the current hardware, it crashed with any similarity join query, making it impossible to re-run the experiments. Thus, we present the results previously reported [24]. Table 3 presents the results: the execution time of DBSimJoin grows rapidly with r because of the quick growth of the size and number of window partitions in the QJ algorithm. DBSimJoin crashes with $r > 0.4$ so results between 0.3 and 0.4 are included to illustrate the growth in runtime closer to the failing point. The runtime of Jena-vp increases slowly with r, where more tree branches need to be visited as the result-set

Table 3
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

Fig. 16. Execution time of range-based self-similarity joins, including polynomial trend line for DBSimJoin times beyond 4 hours.

size increases up to almost 400 million records. The implementation of Jena-vp does not crash with massive results because it fetches the bindings lazily: it obtains all pairs within distance ε 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 DBSimJoin to crash. In Fig. 16 the execution times of DBSimJoin and vp-Jena are presented visually, along with a polynomial trend of order 2 that predicts the execution times for when DBSimJoin crashes. Notice that both axes are in logarithmic scale.

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

6.2. Clustering Experimental Results

To gain insight on the performance of the clustering solution modifier, we measure the overhead produced by the clustering algorithms. To achieve this, we prepared 1,000 synthetic SPARQL queries that draw values from the numeric properties of Wikidata. The queries are composed of only one Basic Graph Pattern (BGP), and we apply clustering over its solutions. To produce these queries, we begin with some data exploration. Specifically, from a Wikidata dump, we compute the ordinal/numeric characteristic sets [45] 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 [45] 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 CLUSTER BY queries from 1,000 ordinal/numeric characteristic sets containing more than 3 properties that were defined for at least 500 subjects.

Then, we execute these queries with different parameters, in order to compare the execution time of the three implemented algorithms under various realistic configurations. These parameters are:

- For k-means: $k \in \{4, 8, 12, 16\}$, and $m \in \{10, 20, 30\}$.
- For DBSCAN: $\varepsilon \in \{0.01, 0.1, 0.5, 1\}$, and $p \in \{1, 10, 100\}$.
- For k-medoids: $k \in \{4, 8, 12, 16\}$.

Fig. 17. Execution time of the CLUSTER BY queries using k-means clustering

Fig. 18. Execution time of the CLUSTER BY queries using DBSCAN clustering

We use as a baseline the execution time of the query that counts the results of the underlying BGP of each benchmark query, so as to measure the overhead produced by the CLUSTER BY operators.

In Fig. 17, we present the results for the queries using the k-means algorithm. We use same-coloured lines for runs with the same value of k, and different dashing for different values of m. The baseline is shown with the light green line. We observe that the execution time increases as the value of k increases, and that the value of m does not significantly influence the execution time. This is expected, since k-means has an upper bound time complexity of O(knm) distance computations. However, k-means can terminate before m iterations if the clusters have converged.

The results for the queries using DBSCAN are harder to interpret, as the execution time also depends on the distance distribution of the space defined by the solution mappings obtained from evaluating each BGP. This is because DBSCAN's cost depends on the cost of performing range searches, and on the number of found neighbours. This

Fig. 19. Execution time of the CLUSTER BY queries using k-medoids clustering

can be noticed in Fig. 18, which presents a plot where queries with the same value for k are shown in the same colour, and queries with the same value for p are shown with the same line dashing. The baseline is shown with the light green line. In Fig. 18, we observe unpredictable changes in the magnitude of the execution time, even for queries with a similar n. This can also be explained by the fact that we use fixed values for this evaluation instead of using distance values depending on the metric space derived from the query evaluation, as it is recommended [46]. Notice that normalisation of the underlying metric space would not guarantee more consistent results, as the cost of DBSCAN depends on the distance distribution of the space. Nevertheless, queries with a larger p and a smaller p tend to take more time. For these results, we marked queries taking more than 3 hours as timeouts, and thus not reported in the graph. This leads to most queries with p = 1 to not be included.

k-medoids is the costliest clustering algorithm of the three, as it minimises intra-cluster distance while maximising inter-cluster distance. Since the implementation requires to compute all pairwise distances, $O(n^2)$ memory is required, with an extra memory of $O(k)$ used by the FASTPAM algorithm [44]. This memory requirement resulted in queries with n larger than 5,584 to fail with out of memory errors. In Fig. 19, we present the execution times for those queries that finished successfully. It can be noticed that k-medoids takes much longer to execute than the underlying query, and around four orders of magnitude more time than k-means. For $k = 16$, queries with $n > 2908$ take longer than 12 hours and were manually cancelled and therefore not reported.

Given these experimental results, we can argue that the preferred way to cluster query results would be to use k-means, which shall have a similar execution time as the underlying query. Moreover, the k-medoids clustering algorithm should be avoided given its intensive memory and time usage. For the case of DBSCAN, it is necessary to be familiar with the vector space that is being generated, and carefully choose the parameters, for which we refer to the recommendations of Schubert et al. [46], of using $p = 2 \cdot d$, where d is the dimensionality of the space, and ε as the "elbow" value in a graph of the distance of each object to its 1-nearest neighbour.

7. Concluding Remarks

In this paper we have presented two extensions to SPARQL that incorporate two new similarity-based operators: similarity join and clustering. We define each operator's syntax and semantics as part of the SPARQL grammar and the SPARQL algebra, and show how these new operators can be combined with the rest. In the case of the similarity join, we prove algebraic properties and rewriting equivalences. For the case of the clustering solution modifier, we discuss how it interacts with the other solution modifiers, such as GROUP BY. We motivate the importance of these extensions by defining use-case queries ranging several domains (images, stellar data, encyclopaedic data).

These operators have been implemented on top of Apache Jena, and the implementation is publicly available. We provide experimental results to evaluate the performance of the implementation, where we show that our similarity join outperforms the closest system to ours, DBSimJoin. We also offer comparisons among different methods and con gurations to evaluate both the similarity join and the clustering. We determine that the better way to compute the similarity join is the vp-tree when there is a large number of solution mappings, and the nested loop when there are fewer solution mappings. For these experiments, we use 2,726 real SPARQL queries obtained from the logs of Wikidata. We also show that, when clustering, users may prefer to use the k-means algorithm, as it is faster and consumes less memory than k-medoids. We conclude by observing that DBSCAN is rather unpredictable in terms of execution time, and that it can produce useful and fast results if the parameters are carefully chosen.

As a future direction for this work, we plan to implement the physical operators using algorithms for secondary memory. This would allow for more ef cient query processing and improved scalability, which has been an issue in particular fork-medoids clustering. Furthermore, we could design tailored algorithms that can take advantage of the database setting and the features of SPARQL, to further optimise the query processing.

Acknowledgements

This work was partly funded by ANID - Millennium Science Initiative Program - Code ICN17_002, by the Swedish Research Council (Vetenskapsrådet, project reg. no. 2019-05655), and the CENIIT program at Linköping University (project no. 17.05).

References

[1] A. Petrova, E. Sherkhonov, B.C. Grau and I. Horrocks, Entity Comparison in RDF Graphs, in: International Semantic Web Conference (ISWC) Springer, 2017, pp. 526–541.
[2] M.A. Sherif and A.N. Ngomo, A systematic survey of point set distance measures for link discovery, Semantic Web 9(5) (2018), 589–604.
[3] G. Giacinto, A Nearest-neighbor Approach to Relevance Feedback in Content Based Image Retrieval, in: International Conference on Image and Video Retrieval (CIVR) ACM, New York, NY, USA, 2007, pp. 456–463. ISBN 978-1-59593-733-9.
[4] H. Li, X. Zhang and S. Wang, Reduce Pruning Cost to Accelerate Multimedia kNN Search over MBRs Based Index Structures, in: Third International Conference on Multimedia Information Networking and Security, 2011, pp. 55–59. ISSN 2162-8998.
[5] R. Guerraoui, A. Kermarrec, O. Ruas and F. Taïani, Fingerprinting Big Data: The Case of KNN Graph Construction, in: International Conference on Data Engineering (ICDE) 2019, pp. 1738–1741. ISSN 2375-026X.
[6] S. Ferrada, B. Bustos and A. Hogan, IMGpedia: a linked dataset with content-based analysis of Wikimedia images, in: International Semantic Web Conference (ISWC) Springer, 2017, pp. 84–93.
[7] C. Böhm and F. Krebs, Supporting KDD applications by the k-nearest neighbor join, in: International Conference on Database and Expert Systems Applications (DEXA) Springer, 2003, pp. 504–516.
[8] A. Hogan, M. Mellotte, G. Powell and D. Stampouli, Towards Fuzzy Query-Relaxation for RDF, in: Extended Semantic Web Conference (ESWC) 2012, pp. 687–702.
[9] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault and J. Morissette, Bio2RDF: towards a mashup to build bioinformatics knowledge systems, Journal of Biomedical Informatics 41(5) (2008), 706–716.
[10] P. Zezula, G. Amato, V. Dohnal and M. Batko, Similarity search: the metric space approach, Vol. 32, Springer Science & Business Media, 2006.
[11] P.N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: Symposium on Discrete Algorithms (SODA) Vol. 93, 1993, pp. 311–321.
[12] E.H. Jacox and H. Samet, Metric space similarity joins, ACM TODS 33(2) (2008), 7.
[13] V. Dohnal, C. Gennaro, P. Savino and P. Zezula, D-index: Distance searching index for metric data sets, Multimedia Tools and Applications 21(1) (2003), 9–33.
[14] R. Paredes and N. Reyes, Solving similarity joins and range queries in metric spaces with the list of twin clusters, Journal of Discrete Algorithms 7(1) (2009), 18–35.
[15] C. Kiefer, A. Bernstein and M. Stocker, The fundamentals of iSPARQL: a virtual triple approach for similarity-based Semantic Web tasks, in: International Semantic Web Conference (ISWC) Springer, 2007, pp. 295–309.
[16] J. Volz, C. Bizer, M. Gaedke and G. Kobilarov, Discovering and Maintaining Links on the Web of Data, in: International Semantic Web Conference (ISWC) Springer, 2009, pp. 650–665.
[17] R. Battle and D. Kolas, Enabling the geospatial Semantic Web with Parliament and GeoSPARQL, Semantic Web 3(4) (2012), 355–370.

[18] X. Zhai, L. Huang and Z. Xiao, Geo-spatial query based on extended SPARQL, in: *International Conference on Geoinformatics (GEOINFORMATICS)*, IEEE, 2010, pp. 1–4.
[19] R. Oldakowski and C. Bizer, SemMF: A Framework for Calculating Semantic Similarity of Objects Represented as RDF Graphs, *Poster at ISWC* (2005).
[20] A.N. Ngomo and S. Auer, LIMES – A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data, in: *International Joint Conference on Artificial Intelligence (IJCAI)*, 2011, pp. 2312–2317.
[21] A. Ławrynowicz, Query results clustering by extending SPARQL with CLUSTER BY, in: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, Springer, 2009, pp. 826–835.
[22] Y.N. Silva, S.S. Pearson and J.A. Cheney, Database similarity join for metric spaces, in: *International Conference on Similarity Search and Applications (SISAP)*, Springer, 2013, pp. 266–279.
[23] L. Qi, H.T. Lin and V. Honavar, Clustering remote RDF data using SPARQL update queries, in: *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*, IEEE, 2013, pp. 236–242.
[24] S. Ferrada, B. Bustos and A. Hogan, Extending SPARQL with Similarity Joins, in: *The Semantic Web – ISWC 2020*, J.Z. Pan, V. Tamma, C. d'Amato, K. Janowicz, B. Fu, A. Polleres, O. Seneviratne and L. Kagal, eds, Springer International Publishing, Cham, 2020, pp. 201–217. ISBN 978-3-030-62419-4.
[25] S. Harris, A. Seaborne and E. Prud'hommeaux, SPARQL 1.1 Query Language, 2013, https://www.w3.org/TR/sparql11-query/.
[26] J. Pérez, M. Arenas and C. Gutiérrez, Semantics and complexity of SPARQL, *ACM TODS* **34**(3) (2009), 16:1–16:45.
[27] B.C. Ooi, Spatial kd-tree: A data structure for geographic database, in: *Datenbanksysteme in Büro, Technik und Wissenschaft*, Springer, 1987, pp. 247–258.
[28] A. Guttman, *R-trees: A dynamic index structure for spatial searching*, Vol. 14, ACM, 1984.
[29] K. Shim, R. Srikant and R. Agrawal, High-dimensional similarity joins, in: *International Conference on Data Engineering (ICDE)*, IEEE, 1997, pp. 301–311.
[30] G. Navarro, Analyzing Metric Space Indexes: What For?, in: *International Conference on Similarity Search and Applications (SISAP)*, IEEE Computer Society, Washington, DC, USA, 2009, pp. 3–10. ISBN 978-0-7695-3765-8.
[31] C. Böhm, B. Braunmüller, F. Krebs and H.-P. Kriegel, Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data, in: *SIGMOD Record*, Vol. 30, ACM, 2001, pp. 379–388.
[32] J.-P. Dittrich and B. Seeger, 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)*, ACM, 2001, pp. 47–56.
[33] M. Muja and D.G. Lowe, Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration, in: *International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISSAPP)*, INSTICC Press, 2009, pp. 331–340.
[34] M. Galkin, M. Vidal and S. Auer, Towards a Multi-way Similarity Join Operator, in: *New Trends in Databases and Information Systems (ADBIS)*, Springer, 2017, pp. 267–274.
[35] J. MacQueen et al., Some methods for classification and analysis of multivariate observations, in: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1, Oakland, CA, USA, 1967, pp. 281–297.
[36] L. Kaufman and P.J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*, John Wiley & Sons, 2009.
[37] M. Ester, H.-P. Kriegel, J. Sander, X. Xu et al., A density-based algorithm for discovering clusters in large spatial databases with noise., in: *Proceedings of the 2nd ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, Vol. 96, 1996, pp. 226–231.
[38] C. Ordonez and E. Omiecinski, Efficient disk-based K-means clustering for relational databases, *IEEE Transactions on Knowledge and Data Engineering* **16**(8) (2004), 909–921.
[39] C. Zhang and Y. Huang, Cluster by: a new SQL extension for spatial data aggregation, in: *Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, 2007, pp. 1–4.
[40] C. Li, M. Wang, L. Lim, H. Wang and K.C.-C. Chang, Supporting ranking and clustering as generalized order-by and group-by, in: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of data*, 2007, pp. 127–138.
[41] Y.N. Silva, W.G. Aref and M.H. Ali, Similarity Group-By, in: *2009 IEEE 25th International Conference on Data Engineering*, 2009, pp. 904–915. doi:10.1109/ICDE.2009.113.
[42] D. Vrandečić and M. Krötzsch, Wikidata: A Free Collaborative Knowledgebase, *Comm. ACM* **57** (2014), 78–85.
[43] E. Prud'hommeaux, SPARQL query language for RDF, 2008, http://www. w3. org/TR/rdf-sparql-query/.
[44] E. Schubert and P.J. Rousseeuw, Fast and eager k-medoids clustering: O (k) runtime improvement of the PAM, CLARA, and CLARANS algorithms, *Information Systems* **101** (2021), 101804.
[45] T. Neumann and G. Moerkotte, Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins, in: *International Conference on Data Engineering (ICDE)*, 2011, pp. 984–994.
[46] E. Schubert, J. Sander, M. Ester, H.P. Kriegel and X. Xu, DBSCAN revisited, revisited: why and how you should (still) use DBSCAN, *ACM Transactions on Database Systems (TODS)* **42**(3) (2017), 1–21.

Appendix A. Proofs of Algebraic Properties

This section presents proofs for the algebraic properties of the similarity join operator introduced in Section 4.2.

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

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

$$\begin{aligned} X \bowtie_{\mathfrak{r}} Y &\coloneqq \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Y \bowtie \{[v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})]\} \mid \mu_v(v) \ \mathsf{6} \ r\} \\ &\equiv \{[\mu_2, \mu_1, \mu_v] \in Y \bowtie X \bowtie \{[v/\delta([\![\mathcal{V}]\!]^{\mu_2}_{\mu_1})]\} \mid \mu_v(v) \ \mathsf{6} \ r\} \\ &\equiv Y \bowtie_{\mathfrak{r}} X \end{aligned}$$

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

$$\begin{aligned} X \bowtie_{\mathfrak{r}} (Y \cup Z) &\coloneqq \{[\mu_1, \mu_2, \mu_v] \in X \bowtie (Y \cup Z) \bowtie \{[v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})]\} \mid \mu_v(v) \ \mathsf{6} \ r\} \\ &\equiv \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Y \bowtie \{[v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})]\} \mid \mu_v(v) \ \mathsf{6} \ r\} \cup \\ &\quad \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Z \bowtie \{[v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})]\} \mid \mu_v(v) \ \mathsf{6} \ r\} \\ &\equiv (X \bowtie_{\mathfrak{r}} Y) \cup (X \bowtie_{\mathfrak{r}} Z) \end{aligned}$$

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

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

Proof. As counter examples for commutativity and distributivity, note that there exist sets of mappings X, Y, Z with $|X| = n$, $|Y| = |Z| = 2n$, $n > k > 0$ 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 A.3. $\bowtie_{\mathfrak{n}}$ *is right-distributive over* $\cup$.

Proof. Assume $\mathfrak{n} = (\mathcal{V}, \delta, v, \mathsf{nn}_k)$, with threshold distance r^0. We see that:

$$\begin{aligned} (X \cup Y) \bowtie_{\mathfrak{n}} Z &\coloneqq \{[\mu_1, \mu_2, \mu_v] \in (X \cup Y) \bowtie Z \bowtie \{[v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})]\} \mid \mu_v(v) \ \mathsf{6} \ r^0\} \\ &\equiv \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Z \bowtie \{[v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})]\} \mid \mu_v(v) \ \mathsf{6} \ r^0\} \cup \\ &\quad \{[\mu_1, \mu_2, \mu_v] \in Y \bowtie Z \bowtie \{[v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})]\} \mid \mu_v(v) \ \mathsf{6} \ r^0\} \\ &\equiv (X \bowtie_{\mathfrak{n}} Z) \cup (Y \bowtie_{\mathfrak{n}} Z) \end{aligned}$$

□

Proposition A.4. $(X \bowtie_{\mathfrak{s}} Y) \bowtie_{\mathfrak{s}^\theta} Z \not\equiv X \bowtie_{\mathfrak{s}} (Y \bowtie_{\mathfrak{s}^\theta} Z)$ *holds, i.e.,* $\bowtie_{\mathfrak{s}}$ *is not associative.*

Proof. As a counter example, consider that $\mathfrak{s}$ and $\mathfrak{s}^0$ bind distance variables v and v^0 respectively such that $v^0 \in \mathrm{udom}(X)$, $v^0 \notin \mathrm{udom}(Y) \cup \mathrm{udom}(Z)$ and $v \notin \mathrm{udom}(X) \cup \mathrm{udom}(Y) \cup \mathrm{udom}(Z)$. Now $(X \bowtie_{\mathfrak{s}} Y) \bowtie_{\mathfrak{s}^\theta} Z$ returns an error as the left operand of $\bowtie_{\mathfrak{s}^\theta}$ assigns v but $X \bowtie_{\mathfrak{s}} (Y \bowtie_{\mathfrak{s}^\theta} Z)$ will not. □

Proposition A.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. $(X \bowtie_{\mathfrak{s}} Y) \between Z \equiv (X \between Z) \bowtie_{\mathfrak{s}} Y$ *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* $\text{idom}(X)$.

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).

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

2. For a mapping $\mu = [\mu_1, \mu_2]$ such that $\text{udom}(Z) \cap (\text{dom}(\mu_2) - \text{dom}(\mu_1)) = \emptyset$, there does not exist $\mu^0 \in Z$ such that $\mu \sim \mu^0$ if and only if there does not exist $\mu^0 \in Z$ such that $\mu_1 \sim \mu^0$. Taking $\mu_1 \in X$ and $\mu_2 = [\mu_2^{\emptyset}, \mu_2^{\emptyset\emptyset}] \in Y \bowtie \{v/\delta([\![\mathcal{V}]\!]^{\mu_1}_{\mu_2})$ 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 $\bowtie_{\mathfrak{t}}$ and $\bowtie_{\mathfrak{n}}$ over $\cup$ proven in previous propositions.

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

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

□