SIMILARITY-BASED WEB QUERIES

TESIS PARA OPTAR AL GRADO DE DOCTOR EN COMPUTACIÓN

SEBASTIÁN CAMILO FERRADA ALIAGA

PROFESORES GUÍA:
BENJAMÍN BUSTOS CÁRDERNAS
AIDAN HOGAN

MIEMBROS DE LA COMISIÓN:
MARCELO MENDOZA ROCHA
DIEGO ARROYUELO BILLIARDI
GUILLAUME GRAVIER

Este trabajo ha sido parcialmente financiado por el Instituto Milenio Fundamentos de los Datos y CONICYT-PFCHA 2017-21170616

SANTIAGO DE CHILE
2021
Resumen

La Web de Datos apunta a que todos los datos en la Web estén estructurados, interconectados y sean entendibles por los computadores. Usualmente los esfuerzos del área están enfocados en datos textuales y enciclopédicos. Sin embargo, la Web es una rica colección de datos multimodales con múltiples formatos, codificaciones e idiomas con varios niveles de utilidad y veracidad. En este trabajo proponemos y desarrollamos varias modalidades para incluir esta diversidad en la Web de Datos, además de nuevos métodos para acceder y navegar a través de ellos.

En este trabajo estudiamos los diferentes problemas que surgen al considerar datos heterogéneos y cómo algoritmos, consultas y agrupamientos basados en similitud pueden ayudar a resolverlos. De esta manera, innovamos en tres áreas: (1) proponemos una nueva heurística para resolver reuniones por similitud aproximadas, (2) presentamos e implementamos una extensión a SPARQL (el lenguaje de consulta estándar para la Web de Datos) que incluye un operador de reunión por similitud, y (3) hacemos uso de estas técnicas para mejorar una base de datos enlazadas de imágenes y para implementar una extensión de clústering para SPARQL.

Nuestra contribución se compone de lo siguiente: primero, reconocemos que las reuniones basadas en similitud a través de vecinos más cercanos son algorítmicamente más complejas de resolver que aquellas basadas en rangos, por lo tanto, proponemos una heurística aproximada con peor caso subcuadrático; segundo, acercamos la Web de Datos a las bases de datos multimedia definiendo detalladamente un nuevo operador algebraico para SPARQL con su respectiva sintaxis y semántica, proveyendo una implementación hecha sobre Apache Jena; tercero, basado en lo aprendido durante el trabajo, mejoramos IMGpedia, una base de datos enlazadas que contiene relaciones de similitud calculadas estáticamente sobre las imágenes de Wikimedia Commons, de forma que los usuarios puedan realizar consultas más expresivas y calculen relaciones de similitud de forma dinámica; finalmente, implementamos una propuesta anterior que incluye un modificador de consultas SPARQL basado en clústering, actualizando su definición para que funcione con el estándar actual.

El valor de este trabajo reside en que, además de simplemente soportar nuevas operaciones y tipos de consulta, presenta el potencial para impactar a otras áreas dentro de la Web de Datos, como lo son la Integración de Datos, Enlazamiento de Entidades y Relajación de Consultas. Además prevemos que para que la Web de Datos se vuelva convencional, es necesario que se considere completamente la diversidad y complejidad de los datos en la Web, donde esta tesis puede ser el cimiento.
Abstract

The Web of Data aims that all data on the Web is well-structured, interlinked and machine-readable. Oftentimes the efforts are focused on textual and encyclopaedic data. However, the Web is a rich collection of multi-modal data with various formats, encodings and even languages at various levels of usefulness and trustworthiness. In this work we develop and propose different means to include more diverse data on the Web, as well as novel methods to access and navigate it.

In this work we study different problems that arise when considering heterogeneous data and how similarity-based algorithms, queries and grouping can help solve them. Along these lines, we innovate in three areas: (1) we propose a novel heuristic to solve approximated similarity joins, (2) we introduce and implement an extension to SPARQL (the standard query language for the Web of Data) that includes a similarity join operator, and (3) we make use of these techniques to improve an existing linked dataset of images and to implement a clustering extension for SPARQL.

Our contribution is composed of the following: first, we recognise that similarity joins based on nearest neighbours are algorithmically harder to solve than those based on threshold distances, and thus propose an approximated heuristic with subquadratic worst case; second, we bring together the Web of Data with Multimedia Databases, by thoroughly defining a new algebraic operator for SPARQL, along with detailed syntax and semantics, and providing an implementation built on top of Apache Jena; third, we leverage what was learnt to improve IMGpedia, a linked dataset with static similarity relations among Wikimedia images, in a way that users can perform more expressive queries and dynamically compute similarity relations that are not currently present in the data; finally, we implement an old proposal to include clustering of query solutions natively in SPARQL, updating this definition so that it works with SPARQL 1.1.

The value of this work lies in that, besides just supporting new operations and kinds of queries, it presents the potential to impact other areas within the Web of Data such as Data Integration, Entity Linking and Query Relaxation. We also foresee that for the Web of Data to become mainstream it must take into account the diversity and complexity of the data on the Web, where this thesis can be a cornerstone.
Para Marisol, Álvaro, Alejandro, Pedrito y Tomás.
Acknowledgements

Well, this was hard. I jumped right from my masters thesis into this one, believing that the success of the former assured the success of the latter. I couldn’t be more wrong. It was thanks to many people that I was able to pull this off, and I’d love to thank everyone here, in no particular order.

First off, thanks to my family for always having my back, especially during the pandemic. I love you all. To my mom, I will never be able to repay everything you do for me; I just hope that this work makes you proud. To my boyfriend Tomás, for loving me as much as you do, for always being encouraging and supporting, for always reminding me how brilliant I can be, despite me thinking otherwise. Thanks to my friends who always listened to me, to whom I could ask the most random questions and never made me feel like a weirdo. Thanks for the beers and the coffees and the cults and the nights at the disco. All of you have been a lighthouse during this journey. Special thanks to Jorge and Ivana who pushed me to take care of my mental health; without your encouragement I would’ve never got better.

I’d like to thank my advisers, Aidan and Benjamin, for all the reviews and the ideas and the discussions and the friendship. I have learnt so much from you. Both of you are my role models on how to be a great researcher, an excellent teacher and a good person. Thanks to all the professors that have worked with me and have encouraged me to follow this dream: Claudio Gutierrez, Alex Bergel, Sergio Ochoa and Gonzalo Navarro. Thanks to Christian Bridevaux for giving me many chances to teach. Thanks to the Institute for Foundational Research on Data for providing me with a great scientific community and for the various fundings you have supported me with. Thanks to ANID for the scholarship that funded my PhD.

Thanks to Sandra and Angélica, for being always eager to help. To Sergio Aguilera, for all the support with the server and for never judging me even when I asked the dumbest questions. Thanks to Nicolás Bravo for the work described in Section 6.1.1. Thanks to Yasin Silva for giving me pointers on how to install and run DBSimJoin.

I’m sure I forgot to mention many people, but I’m running out of space and out of time. Thanks to everyone who has given me a word of encouragement. Thanks to everyone who has ever given me feedback and reviews. Thanks to everyone that sent me a meme when I most desperately needed one. Thanks to everyone who provided me with a hug when I was anxious. Thanks to everyone who worked on the front lines containing the pandemic, you are the truer heroes.
## Contents

1. **Introduction** .......................................................... 1  
   1.1. Hypothesis and Objectives ........................................ 3  
      1.1.1. Hypothesis .................................................. 4  
      1.1.2. Objectives .................................................. 4  
   1.2. Methodology ..................................................... 5  
   1.3. Contribution of this Work ....................................... 6  
   1.4. Structure of This Work ......................................... 7  

2. **Preliminaries** ..................................................... 8  
   2.1. The Semantic Web ................................................ 8  
       2.1.1. RDF .......................................................... 8  
       2.1.2. RDFS and OWL ............................................ 9  
       2.1.3. SPARQL ...................................................... 11  
   2.2. Similarity Search ................................................ 13  
       2.2.1. Vector Spaces ............................................ 13  
       2.2.2. Metric Spaces ............................................ 14  
       2.2.3. Similarity Queries ..................................... 14  
       2.2.4. Similarity Joins ........................................ 15  

3. **Related Work** ..................................................... 16  
   3.1. Similarity Join Evaluation ................................. 16  
   3.2. Similarity in Databases ....................................... 19  

4. **An Efficient Algorithm for Approximated Similarity Joins** 24  
   4.1. Preliminary Work ............................................... 24  
   4.2. The Algorithm .................................................. 25  
       4.2.1. Complexity Analysis ................................... 27  
       4.2.2. Implementation .......................................... 28  
   4.3. Evaluation ...................................................... 29  
       4.3.1. Experimental Settings .................................. 29  
       4.3.2. Average Precision ...................................... 29  
       4.3.3. Precision Distribution .................................. 31  
       4.3.4. Execution Time and Distance Calculations ............ 32  
       4.3.5. Centre Selection Strategies ............................. 33  
       4.3.6. Comparison with Other Similarity Join Algorithms .... 35  

5. **Similarity Joins in SPARQL Queries** .......................... 37
5.1. Desiderata ................................................................. 37
5.2. Challenges .............................................................. 38
  5.2.1. Size of the Data ................................................. 38
  5.2.2. Dimension-Agnostic Indexing .................. 38
  5.2.3. Multi-valued Attributes ...................................... 39
  5.2.4. Incomplete Data .............................................. 40
  5.2.5. Value Distribution ........................................... 40
  5.2.6. Abstract Distances ........................................... 41
5.3. Syntax ................................................................. 41
5.4. Semantics .............................................................. 42
5.5. Algebraic Properties .............................................. 44
5.6. Use-Case Queries ................................................... 46
5.7. Implementation ....................................................... 48
  5.7.1. Implementation Details ..................................... 49
5.8. Query Planning and Optimisation .............................. 51
5.9. Evaluation .............................................................. 53
  5.9.1. Wikidata: k-nn Self-Similarity Queries .......... 54
  5.9.2. Corel Colour Moments: Range Similarity Queries 56
6. Applications ............................................................. 58
  6.1. IMGpedia ........................................................... 58
    6.1.1. Browsing the Data ........................................ 59
    6.1.2. Querying the Data ......................................... 61
  6.2. Dynamic Clustering of SPARQL Query Results .......... 66
    6.2.1. Syntax and Semantics ..................................... 67
    6.2.2. Implementation ............................................ 68
    6.2.3. Use-Cases .................................................. 70
7. Conclusions and Future Directions ................................. 72
  7.1. Conclusions ........................................................ 72
  7.2. Assessment of the Objectives of the Thesis ............ 73
  7.3. Future Work ........................................................ 74
Bibliography ........................................................................ 76
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.</td>
<td>Histogram of the actual ranking of the results of the algorithm</td>
<td>25</td>
</tr>
<tr>
<td>4.2.</td>
<td>Average Precision of the Algorithm for the different Datasets, values of $c$ and $k$</td>
<td>30</td>
</tr>
<tr>
<td>4.3.</td>
<td>Average precision of the algorithm using vp-tree-like centre selection</td>
<td>35</td>
</tr>
<tr>
<td>4.4.</td>
<td>Summary of the comparison of Root-Join v. other algorithms</td>
<td>36</td>
</tr>
<tr>
<td>5.1.</td>
<td>10 most frequent characteristic sets of numeric properties in Wikidata</td>
<td>54</td>
</tr>
<tr>
<td>5.2.</td>
<td>Execution times in seconds for range similarity joins over the CCM Dataset</td>
<td>57</td>
</tr>
<tr>
<td>6.1.</td>
<td>High-level statistics for IMGpedia</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Figures

2.1. RDF graph written in Turtle ............................................ 9
2.2. Visual representation of the RDF in Figure 2.1 ....................... 10
2.3. Example SPARQL query requesting the names of authors born before 1990 .................................................. 11

3.1. DBSimJoin query with a range-based self-similarity join [54] ....... 20
3.3. Visuo-semantic query for IMGpedia [21] ............................. 21
3.4. iSPARQL Query that obtains similar books [35] ................. 22

4.1. Example results of the group partition algorithm for varying values of $c$ and using Manhattan distance .............................................. 27
4.2. Average precision of the algorithm per value of $c$ when $k = 16$ ...... 29
4.3. Distribution of the number of correctly found 16-nn using DECAF dataset .................................................. 31
4.4. Execution time for each dataset, $c$ and $k$ ............................. 32
4.5. Execution time for each value of $c$ and $k = 16$ .................... 33
4.6. Computed distances per $c$ value and per dataset .................... 33
4.7. Partition of random 2D points using the centre selection strategy of Yianilos [66] .................................................. 34

5.1. Proposed syntactic extension to SPARQL 1.1 Grammar [30] ....... 41
5.2. Example extended SPARQL query computing a range similarity join .................................................. 42
5.3. Query for European countries most similar to Latin American countries in terms of a variety of economic indicators, with sample results ....................... 46
5.4. Query for non-metal chemical elements similar to metallic elements in terms of atomic properties .................................................. 47
5.5. Query for elections similar to the 2017 German Federal Election in terms of number of candidates, parties and ideologies participating, with results .................................................. 48
5.6. Parsing of a SPARQL query with Similarity Joins to Query Elements .................................................. 50
5.7. Tree of algebraic operations .................................................. 51
5.8. Average execution time by number of input solutions (entities) for $k = 1, 4, 8$ .................................................. 55
5.9. Value distribution of query execution time by number of properties and per algorithm for $k = 1$ and $k = 8$ .................................................. 56
5.10. Trend curve predicting execution times for DBSimJoin for $r \geq 0.4$ .................................................. 57

6.1. Images of the new interface of IMGpedia ............................ 60
6.2. Government palaces in Latin America (sample) ...................... 61
6.3. Educational institutions similar to Latin American government palaces .................................................. 63
6.4. People Similar to Paintings in the Louvre ................................ 64
Chapter 1

Introduction

The Semantic Web is a global effort to maintain human knowledge in structured, machine-readable formats on the Web in such a manner that data provide links to other data, potentially residing at remote locations elsewhere on the Web. Notable examples of datasets and ecosystems that are part of the Semantic Web include Wikidata [63] and DBpedia [3], which provide machine-readable access to the data relating to Wikipedia. In this work we identify and propose two areas in which the Semantic Web can be improved: a) its relation with multimedia and b) the types of queries users are allowed to write.

Amongst the available datasets describing multimedia in the Semantic Web, such as DBpedia Commons [61], the emphasis has been on capturing the high-level metadata of the multimedia files (e.g., author, date created, file size, width, duration) rather than audio or visual features of the multimedia content itself. However, as mentioned in previous works (e.g., [1, 36, 60]), merging structured metadata with multimedia content-based descriptors could lead to a variety of applications, such as semantically-enhanced multimedia publishing, retrieval, preservation, etc. These works have proposed methods to describe the audio or visual content of multimedia files in Semantic Web formats, but none of them incorporate content-based descriptors of multimedia files. For example, DBpedia Commons [61] does not extract any audio/visual features directly from the multimedia files of Wikimedia Commons, but rather only captures metadata from the documents describing the files. Along these lines we have created IMGpedia [22], a linked dataset that provides visual descriptor and similarity relations between images from Wikimedia Commons, as well as links to related resources in DBpedia and Wikidata in order to allow visuo-semantic queries over the data (queries combining semantic facts with visual relations). We mainly use IMGpedia as a test-bed for the work presented here.

RDF datasets, the Semantic Web standard for storing information, 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, a SPARQL query may request the names of Nobel laureates that have fought in a war. However, there are times when an exact match is not what users need; instead they wish to write a similarity query, for which there is no declarative support in the SPARQL standard (nor in SQL). Hence, a query to obtain the Latin American country with the most similar population and GDP to Italy cannot be
written declaratively in SPARQL; even if the query were expressed with distances defined using low-level numeric operators, applying ORDER BY in a sub-query, etc., the SPARQL engine is unlikely to know how to optimise such a query, resorting to brute-force evaluation. On the other hand, the potential applications for similarity queries in SPARQL are numerous, including: entity comparison and linking \cite{50, 53}, multimedia retrieval \cite{27, 37}, similarity graph management \cite{29, 22}, pattern recognition \cite{8}, query relaxation \cite{33}, as well as domain use-cases, such as protein similarity \cite{2, 6}.

Given the scenario that web content is increasingly requiring the support of content-based similarity queries and that there is not a formal framework and implementation for doing so, we propose to extend the standard query language of the Semantic Web: SPARQL. In this work we present a research proposal regarding the development of such an extension that supports similarity joins over RDF datasets based on vector spaces. The contribution of the work has multiple aspects: a) the development and benchmarking of algorithms to solve similarity queries; b) the proposal of a framework that defines the steps required to satisfy a similarity-based SPARQL query and retrieves the relevant answers; c) the proposal and evaluation of an extended syntax that integrates with the SPARQL standard; d) the implementation of a database engine that supports the proposed extension, implements the framework, plans and optimises the execution of the queries, and allows users to write similarity-based SPARQL queries.

Supporting similarity queries in SPARQL requires the definition and implementation of an algebraic operator, namely a similarity join $X \bowtie_{s} Y$, which is defined as obtaining all pairs $(x, y)$ such that $x \in X$, $y \in Y$ and $x$ is similar to $y$ according to similarity criteria $s$. A variety of algorithms and indexes have been proposed to evaluate similarity joins in a more efficient manner than processing the Cartesian product $X \times Y$ and applying the similarity criteria $s$ to each pair, where the available optimisations depend on the precise definition of $s$, the nature of the objects in the sets, among other factors.

Similarities are often measured in terms of distance functions between pairs of objects in a $d$-dimensional vector space, with two objects being as similar as they are close 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, symmetric, satisfies the triangle inequality, and states that elements of distance 0 to each other are identical. There are two main types of similarity criteria $s$ considered in practice: a) in a range-based similarity join, $s$ 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, $s$ 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 $x$.

We can use the same similarity notions to implement and develop a system that can dynamically cluster the results of a SPARQL query, through a query modifier keyword and parameters. An initial proposal was made to conceive a CLUSTER BY modifier that brings together similar bindings, thus giving a more flexible option to GROUP BY which matches bindings exactly \cite{69}. In this thesis we review the concepts proposed and present an implementation of the query modifier on top of Apache Jena.

Considering how costly similarity queries in high-dimensional spaces can be, we present a simple algorithm that computes an approximation of the $k$ nearest neighbour self-similarity.
join problem in general metric spaces. This algorithm is a generalisation of a previous work that presented an heuristic for solving the 1NN problem [24]. The algorithm presented here computes $\Theta(n^{3/2})$ distances even in the worst case and gives an approximation with up to 46% of precision for 1NN self-similarity queries and 39% for 16NN. We show how the algorithm performs with three different real datasets: the English dictionary, very high-dimensional deep-feature vectors, and a massive collection of classic visual descriptors.

Though evaluating similarity joins in vector spaces has 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.). A user may even query for dimensions, for example, asking for similar countries based on the available economic indicators present in the data. Being dimension-agnostic introduces various complications; for example, indexing on all combinations of $d$ dimensions would naively result in $O(2^d)$ different index structures, and would not support dynamically generated dimensions. Such challenges, and many others also reviewed in this work, 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 (based on latitude and longitude) [5, 68], etc.

The concrete contribution of this work can be summarised in the following points:

1. The proposal, development and evaluation of a new algorithm that solves approximated $k$ nearest neighbour self-similarity joins in general metric spaces that computes $O(n^{3/2})$ distances.

2. The proposal of a declarative extension for SPARQL Syntax, allowing users to express similarity queries.

3. The proposal and evaluation of a framework that allows the execution of similarity-based queries over RDF datasets.

4. The implementation of an RDF/SPARQL database engine that parses, plans, optimises and executes similarity-based queries using the previously defined syntax and following the proposed framework.

5. The evaluation of the system using real-world massive datasets such as Wikidata [63] and IMGpedia [22], and a comparison with the state-of-the-art.

6. The application of the concepts and systems developed in the thesis on real world problems, particularly regarding clustering of SPARQL query results.

1.1. Hypothesis and Objectives

In this section we present the thesis proposal: the hypothesis to be tested, the main and specific objectives, and the proposed solution that tackles the problem stated in the previous paragraphs.
1.1.1. Hypothesis

The hypothesis of this work is that supporting similarity-based queries within the Semantic Web will allow for the expression of novel queries that enable a new form of web retrieval that can be satisfied in a scalable, efficient, usable manner. By scalable, we expect a system that handles massive datasets, such as Wikidata or IMGpedia: both with billions of RDF triples. As for efficiency, we expect to avoid a quadratic amount of distance computations in the similarity joins. Finally, by usability, we expect that users already familiar with SPARQL are able to write concise SPARQL queries with similarity operators.

This hypothesis leads to the following research questions:

1. How can the quadratic worst case of the similarity join evaluation be avoided? At what cost of precision?

2. How can we compare different algorithms for similarity join evaluation in an RDF/S-PARQL environment?

3. How can SPARQL queries involving similarity search be written succinctly, but in a way that does not compromise expressiveness? How can these queries be optimised?

4. What are the applications for SPARQL queries with similarity joins?

1.1.2. Objectives

The main objective of the thesis is to research and develop methods by which the content on the Web can be accessed, queried and browsed by users with respect to the similarity between entities such that the process is friendly and efficient, the results are satisfying visually and semantically and the queries support a given level of expressiveness. This main objective leads to the following specific objectives:

- Research and develop techniques to efficiently obtain similarity relations from large datasets, especially regarding the evaluation of similarity joins.

- Develop querying algorithms or indexes for SPARQL related to similarity-based queries, potentially using dimensions independently.

- Research and develop algorithms for query plan and optimisation, taking into consideration the possible combination of content-based and semantic features.

- Propose, develop and evaluate semantic and similarity-based retrieval systems, based on the previously developed Semantic Web technologies, that can scale with the data.

- Apply the proposed techniques and systems to current open problems, particularly in terms of being able to manage and query multimedia data in a Semantic Web context.
1.2. Methodology

In this section we present the steps that are followed in this work. The main resources used in this thesis are Wikidata [63] and IMGpedia [22] datasets. The first provides a massive amount of facts related to several different domains, such as Astronomy, Demography, etc. The second is an RDF dataset that contains visual descriptors of 15 million images and similarity relations among them. Both datasets are relevant testbeds for similarity search algorithms and systems and propose interesting applications: e.g., a query-by-example search engine for IMGpedia.

The methodology is based on the classic understand, study, design, build, evaluate framework. Concretely, the work consists of the following steps:

- **Literature review.** Acquire a deep understanding about the algorithms, data structures and heuristics that are used to efficiently evaluate similarity joins between sets of objects of diverse domains. Research how these techniques are incorporated into DBMSs or standalone search engines: the kinds of data that are handled, the performance of similarity search, how expressive the systems are, and how much usage the system receives.

- **Algorithm Benchmark.** Compare the most relevant techniques previously found in order to understand their favourable cases, their worst cases, how they scale, how correct/satisfying are the results retrieved. Build a benchmark using multiple scenarios with different numbers of objects, different numbers of dimensions, and different domains. At this stage, it is possible to propose a new algorithm or an improvement to an existing method.

- **Framework Design.** A pipeline for the evaluation of queries with similarity joins has to be defined, proposing a set of clearly defined steps from the writing of the query to the retrieval of results, stating for each step the inputs required and the outputs produced.

- **Syntax, Parsing, Planning and Optimisation.** Devise and design a declarative language extension that includes similarity joins. Propose an algebra parsing that is built on top of the existing one for queries without similarity joins. Discuss and evaluate different strategies for the planning and optimisation of similarity join queries.

- **System Development.** Implement the devised framework within or alongside an RDF database management system. Such an implementation has to be transparent to the user and to other interfaces of the system: users cannot lose any expressiveness, and input and output data should follow the current format of the system.

- **Evaluation.** Compare the performance of the implemented system to other similar ones (whenever possible), and to different versions of itself. Performance can be measured in terms of execution time, scalability, precision of the output, user adoption, among others.

- **Improvement.** Taking into consideration the results of the evaluation phase, discuss where the language extension, the framework, and/or the implementation can be im-
proved, fixed, redesigned or remade.

1.3. Contribution of this Work

Firstly, we present and evaluate a simple but effective algorithm that computes approximated $k$ nearest neighbour self-similarity joins while computing $\Theta(n^2)$ distances. Secondly, we discuss a proposed extension for SPARQL that allows the computation of both range and $k$NN similarity joins over vector spaces, using arbitrary dimensions. Finally, we present applications where both parts of this work are used including the implementation of a query modifier for clustering.

To the best of our knowledge, the two proposals most closely related to our extension are DBSimJoin [54] and iSPARQL [35]. Unlike DBSimJoin, our goal is to introduce similarity joins to the RDF/SPARQL setting. Unlike both systems, we support kNN 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 [35] and the Quickjoin of DBSimJoin [54]. 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.

In terms of publications we provide a list of all the accepted papers related to this thesis since the beginning of the program:


1.4. Structure of This Work

This work is organised as follows:

1. In Chapters 2 and 3, we describe the theoretical framework of this thesis. First, we address the definitions regarding RDF, SPARQL and similarity joins that are used throughout the text. Second, we discuss the related work in terms of the evaluation of similarity joins, and of how similarity search in general has been included in database systems.

2. Next, in Chapter 4, we present Root-Join, a simple and efficient algorithm for the evaluation of approximated \( k \)-nn self-similarity joins in metric spaces.

3. In Chapter 5 we talk about the design, development and evaluation of a database system that supports similarity join SPARQL queries on RDF datasets.

4. In Chapter 6 we discuss further applications that make use of the notions and learning from previous chapters. In particular, we delve into the implementation of a dynamic clustering query modifier for SPARQL, and an extension to the querying and browsing of images in IMGpedia.

5. Finally, in Chapter 7, we summarise the conclusions of this work and discuss some of the future research lines for the project.
Chapter 2

Preliminaries

In this chapter we briefly define the RDF data model and its query language SPARQL, as well as, the concepts, definitions and principles regarding similarity search and similarity joins that are necessary to follow the thesis.

2.1. The Semantic Web

The Web is strongly document-centric and most of its content makes sense only to humans, not machines. That becomes a problem when attempting to ask more “sophisticated” queries over those documents, or requiring some type of reasoning over the semantics of the data instead of matching the query string to the documents. To address these situations and other deeper problems (as seen in the seminal paper of Berners-Lee et al. [7] about their vision for the Semantic Web), the Semantic Web standards are being developed. According to Hogan [32], the Semantic Web can be conceptualised as an extension of the current Web as it enables the creation, sharing and intelligent re-use of machine-readable content on the Web.

In this section, the core standards and technologies for developing a Semantic Web are presented and formalised.

2.1.1. RDF

The Resource Definition Framework (RDF) is the W3C standard graph-based data model for the semantic web [40]. RDF defines the way data is structured in the Semantic Web for the sake of sharing such information between independent sources. RDF consists fundamentally of Terms, Triples and Graphs.

RDF terms can be either IRIs (I), literal values (L) or blank nodes (B). IRIs are global, international identifiers for resources among the Web, such as http://dbpedia.org/resource/Chile to identify the country of Chile. Literals are primitive values, such as integer or string; datatypes are often borrowed from XML Schema and used as a postfix: “9”^^xsd:integer is the representation of the number 9. Finally, blank nodes are entities
that can confirm the existence of a resource without the need of naming it; for example, we know that Chile has a capital city, but if it is unknown which city it is, we can use a blank node as a placeholder to denote its existence.

A triple \((s, p, o) \in I_B \times I \times I_B I\) is called an RDF triple, where \(s\) is referred to as the subject, \(p\) as the predicate, and \(o\) as the object. For example, a triple containing the number of inhabitants of Santiago would be \((:Santiago,:population,5.6M)\).

An RDF Graph is a set of RDF triples that may share Terms. Given this graph nature of RDF datasets, IRI labeled nodes can be reached from outside in such a way that multiple datasets can be linked together and share information between them. As an example, Figure 2.1 presents an RDF Graph written in Turtle syntax for RDF data where we present triples describing resources for Quentin Tarantino and Uma Thurman. In Figure 2.2 we present a visual representation of the RDF graph of Figure 2.1, where subjects and objects are shown in ellipses, properties are depicted as arrows that go from subject to object, and literal values are highlighted in green. Here, and for the rest of the document, we will use different prefixes in order to abbreviate the IRIs of the examples. The main prefixes we use are rdf: for RDF, rdfs: for RDF Schema, owl: for OWL and just : for a default namespace. A collection of prefixes can be found at [http://prefix.cc](http://prefix.cc).

### 2.1.2. RDFS and OWL

RDF graphs are not enough for describing the complexity of the information. To enrich the expressiveness, define schemata and introduce the ability to reason over the data, RDF Schema [10] and The Web Ontology Language, OWL [41], were developed. An ontology is a set of rules and concepts that show the properties and relationships of the data. We will address some of the use cases of these standards:

\[\text{We define that } I_B = I \cup B\]

[http://rdfplayground.dcc.uchile.cl/](http://rdfplayground.dcc.uchile.cl/)
Figure 2.2: Visual representation of the RDF in Figure 2.1. Figure created with RDF Playground.

- **Subclasses.** In order to specialize the type of a resource the property `rdfs:subClassOf` can be used. As an example, if a resource is a dog, then the resource is an animal, so the triple (:Dog, rdfs:subClassOf, :Animal) can state that fact.

- **Range and Domains.** To automatically infer the type of the subject and object of a triple, we can define the domain and range of a property using `rdfs:domain` and `rdfs:range` properties. For example, if only persons can have a depiction image, the next triples must be written to state that: (:depiction, rdfs:domain, :Person) and (:depiction, rdfs:range, imo:Image).

- **Relation properties.** Relational properties such as symmetry or transitivity can be stated using `owl:TransitiveProperty` or `owl:SymmetricProperty`. For example, ancestry is a transitive relation; so if (:ancestor a owl:TransitiveProperty) and (:A, :ancestor, :B) and (:B, :ancestor, :C) the following triple is also a fact: (:A, :ancestor, :C).

- **Property chain axioms.** To extract conclusions about the data, chain rules can be stated. Such rules have the form `if X then Y`. For example if (:A :brotherOf :B) and (:B :fatherOf :C) it becomes clear that (:A :uncleOf :C), but that triple might not be in the original data. In order to be able to make such entailment, the rule: :uncleOf owl:propertyChainAxiom (:brotherOf :fatherOf) has to be included in the graph. Where (:brotherOf :fatherOf) represents a list in RDF.

- **Unions, intersections and more.** Classes can be unions of other classes, intersec-
Figure 2.3: Example SPARQL query requesting the names of authors born before 1990.

2.1.3. SPARQL

SPARQL – referring to the SPARQL Protocol and RDF Query Language – is the standard query language for RDF datasets [30]. In this section we introduce preliminaries for the core of SPARQL, as defined by Pérez et al. [49], and as needed to discuss later rewriting strategies; however, our proposal supports SPARQL 1.1 [30].

SPARQL Syntax

Let \( V \) denote a set of query variables disjoint with \( IBL \). We define the abstract syntax of a SPARQL query as follows:

1. A triple pattern \( t \) is a member of the set \( VIB \times VI \times VIBL \) (i.e., an RDF triple allowing variables in any position). A triple pattern is a query pattern.

2. If \( Q_1 \) and \( Q_2 \) are query patterns, then \([Q_1 \text{ AND } Q_2], [Q_1 \text{ UNION } Q_2] \text{ and } [Q_1 \text{ OPTIONAL } Q_2]\) are query patterns.

3. Let \( f \) be a built-in expression that takes elements of \( VIBL \) as arguments and applies some operations and comparisons (e.g., \( =, \neq, <, \text{ etc.} \) on those elements. A Boolean combination of such expressions (\( \neg, \land, \lor \)) is itself an expression. If \( Q \) is a query pattern and \( f \) is a built-in expression of this form, then \( \text{filter}_f(Q) \) is also a query pattern.

4. If \( Q \) is a query pattern and \( V \) is a set of variables such that for all \( v \in V \), \( v \) appears in some triple pattern contained in \( Q \), then \( \text{select}_V(Q) \) is a query.

Blank nodes in SPARQL queries are non-distinguished query variables where we will assume they have been replaced with fresh query variables. Per the final definition, we assume without loss of generality, that all queries have a projection \( \text{select}_V(Q) \).

In Figure 2.3 an example SPARQL query is displayed. The query requests for the given names of the actors born before 1990.

Whenever we use \( \text{filter}_f(Q) \) we assume that the variables present in \( f \) are a subset of the variables present in \( Q \).

Note that \( \text{SELECT } * \) is equivalent to returning all variables (or omitting the feature).
SPARQL Algebra

We will now define the algebra for SPARQL queries. A mapping $\mu$ is a partial function from variables in $V$ appearing in the query to constants from IBL appearing in the data. Let $\text{dom}(\mu)$, the domain of the mapping, denote the variables for which $\mu$ is defined. We say that two mappings $\mu_1$, $\mu_2$ are compatible, denoted $\mu_1 \sim \mu_2$, if and only if for all $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it holds that $\mu_1(v) = \mu_2(v)$. If two mappings $\mu_1$ and $\mu_2$ are compatible, we have that $\mu_1 \cap \mu_2$ is also a mapping. Letting $X$, $Y$ denote sets of mappings, we now define the algebraic operations for join, union, minus, optional, filter and projection:

\[
X \bowtie Y := \{ \mu_1 \cup \mu_2 \mid \mu_1 \in X, \mu_2 \in Y, \mu_1 \sim \mu_2 \}
\]
\[
X \cup Y := \{ \mu \mid \mu \in X \cup \mu \in Y \}
\]
\[
X \setminus Y := \{ \mu \in X \mid \exists \mu' \in Y : \mu \sim \mu' \}
\]
\[
X \bowtie^* Y := (X \bowtie Y) \cup (X \setminus Y)
\]
\[
\sigma_f(X) := \{ \mu \in X \mid f(\mu) = \text{true} \}
\]
\[
\pi_V(X) := \{ \mu' \mid \exists \mu \in X : \mu' \subseteq \mu, \text{dom}(\mu') = V \cap \text{dom}(\mu) \}
\]

SPARQL Semantics

Pérez et al \[49\] define the semantics of SPARQL operators in terms of their evaluation over an RDF graph, which results in a set of solution mappings. We denote the evaluation of a query $Q$ over an RDF graph $G$ as $Q(G)$. Before defining $Q(G)$, first let $t$ denote a triple pattern; then by $V(t)$ we denote the set of variables appearing in $t$ and by $\mu(t)$ we denote the image of $t$ under a solution $\mu$. Finally, we can define $Q(G)$ recursively as follows:

\[
t(t(G) := \{ \mu \mid \mu(t) \in G, \text{dom}(\mu) = V(t) \}
\]
\[
[Q_1 \text{ AND } Q_2](G) := Q_1(G) \bowtie Q_2(G)
\]
\[
[Q_1 \text{ UNION } Q_2](G) := Q_1(G) \cup Q_2(G)
\]
\[
[Q_1 \text{ OPTIONAL } Q_2](G) := Q_1(G) \bowtie^* Q_2(G)
\]
\[
\text{FILTER}_f(Q)(G) := \sigma_f(Q(G))
\]
\[
\text{SELECT}_V(Q)(G) := \pi_V(Q(G))
\]

SPARQL 1.1

In the SPARQL 1.1 standard \[30\], the following features were introduced:

- Negation: The operators NOT EXISTS and MINUS were introduced.

- Property Paths: Properties can now be operated and recursively defined within queries, e.g., given a property :p, the operation :p* refers to a path of zero or more :p.

- Variable Assignment: For storing values into named variables, the operators BIND and VALUES were introduced.
• Aggregation Functions: SPARQL 1.1 supports aggregation functions such as COUNT, AVG and SUM along with solution modifiers GROUP BY and HAVING.

• Sub-queries: A query Q can now contain a nested query Q'.

• Miscellaneous Functions: Several new functions were included to the standard, such as COALESCE, ISNUMERIC, ABS, etc.

2.2. Similarity Search

Similarity search, as opposed to exact search, refers to the task of finding objects that are similar to each other instead of finding identical matches. Similarities among objects can be defined in several ways and depend on various factors such as the nature of the objects, the goals of the application, human appreciation, etc. In this section we will formalise the different elements and tasks involved in similarity search.

Let \( U \) be the universe of objects that are being considered, and \( D \subseteq U \) a dataset. Similarity search is defined, upon a query object \( q \in U \), as obtaining all elements \( x \in D \) such that \( x \) and \( q \) are similar. 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, considering the distance as inversely proportional to the similarity. In this work we focus on the similarity problem defined in metric and vector spaces.

2.2.1. Vector Spaces

A vector space defined over a field \( F \) (such as \( \mathbb{R} \) or \( \mathbb{C} \)) is a triple \((V,+,\cdot)\) where \( V \) is a set of vectors, \(+ : V \times V \rightarrow V\) is called vector addition, and \(\cdot : F \times V \rightarrow V\) is the scalar multiplication. Objects in \( V \) are called vectors and values from \( F \) are called scalars. Vectors have the form \( \vec{v} = \langle v_1, \ldots, v_d \rangle \), where \( d \) is the dimension of the space, and all \( v_i \) are scalar values taken from the field \( F \). + and \( \cdot \) should satisfy various conditions over \( V \):

- + is commutative and associative.

- + presents a neutral element \( 0 \in V \) such that \( 0 + v = v \), \( \forall v \in V \), as well as inverse elements for every element in \( V \setminus \{0\} \).

- \( \cdot \) presents a neutral element \( 1 \in V \), \( 1 \neq 0 \), such that \( 1 \cdot v = v \), \( \forall v \in V \).

- \( \cdot \) is compatible with the scalar multiplication \( \times \), i.e., \( \forall a,b \in F, \forall v \in V, a \times (b \cdot v) = (a \times b) \cdot v \).

- \( \cdot \) is distributive over vector and field addition.

The topological intuition that can be grasped in 2 or 3-dimensional vector spaces over \( \mathbb{R} \) does not necessarily hold in high-dimensional vector spaces, particularly the notions of closeness and distance are affected by what is called the curse of dimensionality seen as changes in the distance distribution of the space (generally the mean distance among vectors.
increases with dimension as variance diminishes). The curse of dimensionality is caused by the exponential increase on the volume of the (vector) space as the dimension increases.

2.2.2. Metric Spaces

A metric space is a pair $(\mathbb{U}, \delta)$, where $\mathbb{U}$ is a set of arbitrary objects and $\delta : \mathbb{U} \times \mathbb{U} \rightarrow \mathbb{R}$ a metric distance function. A distance function is a metric if it satisfies the following $\forall x, y, z \in \mathbb{U}$:

1. The distance is reflexive: $\delta(x, y) = 0 \iff x = y$
2. The distance is symmetric: $\delta(x, y) = \delta(y, x)$
3. The distance holds the triangle inequality: $\delta(x, z) + \delta(z, y) \geq \delta(x, y)$

It is worth noting that the previous properties imply that $\delta$ is a non-negative function. There are studies looking at spaces where $\delta$ does not hold all of the metric properties; $\delta$ is then called a pseudometric if it is not reflexive, a quasimetric if it is non-symmetric or a semimetric if it does not satisfy the triangle inequality. For a comprehensive survey on non-metric spaces, please refer to [57].

Common metric spaces that are used in this work are the following:

- $L_p$ spaces $(\mathbb{R}^d, l_p)$, where $l_p$ is a Minkowski distance (see Eq. 2.1). When $p = 1$, the distance is also called Manhattan or taxicab distance; when $p = 2$ the distance is referred to as Euclidean distance; and if $p = \infty$ it is called max-distance.

$$l_p(x, y) = \left( \sum_{i=1}^{d} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

- The space of all the strings and the Levenshtein distance, that measures the minimum number of operations (character addition, subtraction and substitution) required to transform one string into another.

2.2.3. Similarity Queries

There are two main types of similarity queries in metric spaces $(\mathbb{U}, \delta)$, given a a dataset $\mathbb{D} \subseteq \mathbb{U}$ and a query object $q \in \mathbb{U}$: range queries, and $k$-nearest neighbours queries.

Range queries aim to find all objects in the dataset that are at a distance of at most $\varepsilon$ from the query point:

$$Q(\mathbb{D}, q, \varepsilon) = \{ x \in \mathbb{D} \mid \delta(x, q) \leq \varepsilon \}$$

Nearest neighbour queries aim to find a subset of $k$ objects that have the lowest distance to $q$, i.e.:
\[ Q(\mathbb{D}, q, k) = \{ x \in \mathbb{D} \mid \kappa_\delta(q, x) \leq k \} \] (2.3)

where \( \kappa_\delta(x, y) \) returns the rank of \( y \) with respect to its distance from \( x \).

The naive approach to solve these queries is to compare the query object against all the objects in the dataset \( \mathbb{D} \), which is called a *sequential scan* and computes \( \Theta(|\mathbb{D}|) \) distances. When \( \mathbb{D} \) and \( \delta \) form a metric space, several methods can be applied to reduce the total number of computed distances thus reducing the runtime of the query evaluation. These methods include the construction of data structures that allow the pruning of objects from comparison, and approximated algorithms that trade-off the precision of the response for a lower runtime. A review of the available techniques is presented in Chapter 3.

### 2.2.4. Similarity Joins

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. Let \( X \) and \( Y \) be subsets of a universe \( \mathbb{U} \), their similarity join is defined as follows:

\[ X \bowtie_\mathfrak{s} Y = \{(x, y) \mid x \in X \land y \in Y \land x \text{ is similar to } y \text{ according to criteria } \mathfrak{s}\} \] (2.4)

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

\[ X \bowtie_\delta Y := \{(x, y) \mid x \in X, y \in Y, \delta(x, y) \leq \varepsilon\} \] (2.5)

\[ X \bowtie_\kappa Y := \{(x, y) \mid x \in X, y \in Y, \kappa_\delta(x, y) \leq k\} \] (2.6)

where \( \delta \) is a distance function: \( \mathbb{U} \times \mathbb{U} \to [0, \infty) \) and \( \kappa_\delta(x, y) \) works as previously defined.

A similarity join can be evaluated using a brute-force approach. Namely, check if every pair \( (x, y) \in X \times Y \) satisfies the corresponding condition. Such an approach is called a nested-loop and requires \( O(|X| \cdot |Y|) \) distance computations. Several optimisation techniques are available depending on the nature of the sets and on the similarity conditions, some of which are presented in Chapter 3.

Throughout this work, we distinguish two special types of similarity joins: a) the query-by-example, where \( Y = \{y\} \) and b) the self-similarity join, where \( X = Y \).
Chapter 3

Related Work

In this chapter we discuss the relevant literature for this work. It is divided into two sections, the first reviewing the different methods for efficient evaluation of similarity joins in metric spaces; and the second, where we address the ways similarity has been adopted in database systems, particularly in RDF/SPARQL settings.

3.1. Similarity Join Evaluation

During the past years, several algorithms, heuristics, and data structures for solving the various types of similarity joins have been proposed. In this section, we review the related literature. The naive algorithm to compute a similarity join is the nested loop, which checks every pair of objects to check if they are similar or not. If the datasets $X, Y$ are such that $O(|X|) = O(|Y|) = O(n)$, the nested loop takes $O(n^2)$ distance computations. In Algorithm 1 we show pseudocode for the nested loop. In the case of $k$-nn similarity joins, the results are kept in max-heaps of size $k$.

The first approach to evaluate similarity joins is to use metric or multidimensional indices that provide sublinear time similarity search, thus having subquadratic time when performing $n$ searches to complete the join. 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) [66], 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 [44], thus having an upper bound of $O(n^{2\alpha})$ for a similarity join. In Algorithm 2 we present pseudocode for the creation of the vp-Tree, where $select_{vp}$ chooses the vantage point according to the even distribution criteria [66]. Other tree indexes, such as the D-Index [18] and the List of Twin Clusters [47], propose to use clustering techniques over the data.

Gionis et al. [28] proposed Locality-Sensitive Hashing (LSH). This method hashes all objects using functions that ensure that objects that are closer in the metric space have a
Algorithm 1: Pseudocode for a nested loop computing a range similarity join.

**Data:** X, Y sets of objects, r a distance threshold, δ a distance function

**Result:** result a set of pairs of objects

1. \( \text{result} \leftarrow \emptyset \);  
2. \( \text{for } x \in X \text{ do} \)
3. \( \quad \text{for } y \in Y \text{ do} \)
4. \( \quad \quad \text{dist} \leftarrow \delta(x, y) ; \)
5. \( \quad \quad \text{if } \text{dist} \leq r \text{ then} \)
6. \( \quad \quad \quad \text{result} \leftarrow \text{result} \cup \{(x, y)\} \)
7. \( \quad \quad \text{end} \)
8. \( \text{end} \)
9. \( \text{return } \text{result} \)

higher collision probability than those further apart. A kNN query \((x, k)\) is processed by hashing \(x\) and retrieving the elements that collide with it. For the purposes of LSH, Datar et al. [17] propose to use random hash functions taken from a \(p\)-stable probability distribution in order to mimic the behaviour of an \(L_p\) distance. LSH can be used to implement a kNN self-similarity by using all objects in the dataset as a query object for the kNN search.

Böhm and Krebs [9] identified the kNN join as an important database primitive for implementing data mining methods. They proposed an algorithm for computing the kNN join using the so-called MuX index. Their algorithm and methods are focused on solving the kNN join on vector spaces, aiming at minimizing CPU and I/O costs.

Jacox and Samet [34] proposed the Quickjoin algorithm for similarity joins. It divides the space in ball cuts using data objects as pivots. Given two pivots, it uses one as a center and the distance between them to define a radius. It splits the data into the objects inside and outside the ball, and then proceeds recursively until the groups are small enough to perform a nested loop. It keeps window partitions in the frontier of the ball in case there are pairs relevant for the result where each vector ends in different partitions. In Algorithm 3 we present pseudocode for the Quickjoin algorithm. Quickjoin is shown to have a complexity of \(O(n(1 + w)^{\lceil \log n \rceil})\), where \(w\) is the average of the fraction of elements lying within the window partitions. Hence, it has a quadratic worst case. Quickjoin was intended for range-based similarity joins; however, Fredriksson and Braithwaite propose several extensions to Quickjoin [25], including an algorithm for \(k\)-Distance joins: it is intended to compute the \(k\) pairs of objects in the entire dataset that are the closest in the defined space.

Yu et al. [67] propose a dynamic method called \(k\text{NNJoin+}\) for kNN similarity join. This method has the ability to deal with data updates, and it focuses mainly on high-dimensional data. It also allows for computing reverse kNN queries.

Yao et al. [65] proposed \(k\text{NN}\) and \(k\text{NN-join}\) algorithms that can be implemented using SQL primitives. Their algorithms do not depend on the distance function defined by the user. This allows them to take advantage of the query optimizer that can produce an efficient query plan. Their proposal supports exact and approximate similarity searches and \(k\text{NN-join}\)
Algorithm 2: Pseudocode for the recursive function for the creation of a vp-Tree [66].

Data: $S$ the set of objects, $\delta$ a distance function

Function make_vptree($S$):
1:    if $S = \emptyset$ then
2:        return $\emptyset$
3:    end
4:    node ← new(node);
5:    node.point ← select_vp($S$);
6:    node.mu ← Median$_{s \in S}(\delta(node.point, s))$;
7:    L ← $\{s \in S - \{node.point\} \mid \delta(node.point, s) < node.mu\}$;
8:    R ← $\{s \in S - \{node.point\} \mid \delta(node.point, s) \geq node.mu\}$;
9:    node.left ← make_vptree(L);
10:   node.right ← make_vptree(R);
11:   return node;

Function select_vp($S$):
12:   $P ← random_selection(S)$;
13:   best_spread ← 0;
14:   best_p ← $\emptyset$;
15:   for $p \in P$ do
16:        $D ← random_selection(S)$;
17:        $\mu ← Median_{d \in D}(\delta(p, d))$;
18:        spread ← 2$^{nd}$ Moment$_{d \in D}(\delta(p, d) - \mu)$;
19:        if spread $> best$ _spread then
20:            best_spread ← spread;
21:            best_p ← $p$;
22:        end
23:   end
24:   return best_p
joins. Their work only considers vector spaces with $L_p$ norms.

Lu et al. [38] propose an algorithm for computing a $k$NN join using MapReduce. It divides the objects into groups, choosing the centres from one dataset and using those centres to partition the other dataset. The division into groups is done using a Voronoi partition. At each partition, the algorithm keeps track of the closest objects to the centre of the partition and it stores the maximum and minimum distance from the objects to their respective partition centre. Each of these groups is processed by a reducer for computing the $k$NN join. The algorithm may replicate objects in several groups for obtaining the exact answer. Other methods for computing similarity joins based on MapReduce have been proposed by Silva and Reed [55], Wang et al. [64], Song et al. [59], Rong et al. [52], Chen et al. [13], Cech et al. [12], and Moutafis et al. [42].

Pearson and Silva [48] propose an algorithm based on similarity searches for the case of similarity joins. The algorithm is an extension of the eD-Index [19], and it supports a join operation over two individually indexed datasets. The main idea of the algorithm is to index each of the datasets independently with a D-index. Both indexes share the same structure, use the same number of levels, and the same pivots for each level. Then, they use the similarity join algorithm proposed for the self-join over the D-index. The original algorithm is modified so that each pair given as part of the result contains an object from a different dataset.

Chen et al. [14] propose index structures for range joins in uncertain metric data. Given two sets $U$ and $V$ of uncertain objects, a distance function $d$, a parameter $r$ and a probability threshold $\theta$, a probabilistic range join returns all pairs of uncertain objects $(u, v) \in U \times V$ such that $Pr(d(u, v) \leq r) \geq \theta$. For computing the probabilistic range joins, they define two index structures for secondary memory, the so-called “uncertain pivot B+ -tree” (UPB-tree) and the “uncertain pivot B+ -forest” (UPBforest). Both indexes are based on the B+ -tree.

### 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 by Silva et al. [54] implements a range-based similarity join operator as an extension for PostgreSQL. This implementation claims to handle any metric space, thus supporting various metric distances; it is based on the aforementioned index-free Quickjoin (QJ) algorithm which was extended to work in secondary memory. In Figure 3.1 we present a query that uses the extension to obtain similar images, based on 9-dimensional visual descriptors. In the figure it can be appreciated that the user needs to define a threshold distance for the range-based similarity join, as well as the dimensions with respect to which to compute the distance. Despite Silva et al.’s, claim that any metric distance is supported, the described keyword for indicating the distance function is not supported in the PostgreSQL extension.

A number of works have proposed online computation of similarity joins in the context of domain-specific measures and the Semantic Web. Zhai et. al [68] 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 [5] implements
an OGC standard called GeoSPARQL, which aside from various geometric operators, also includes geospatial distance and topological relationships among objects. Query performance is improved by using geospatial indexes and bounding box queries to estimate the size of the results. In Figure 3.2 we present a query written for the Parliament engine, requesting parks less than three kilometres away from the Washington Monument. RecSPARQL [41] proposes to extend SPARQL with features that allow for recommendations; while the focus is on using graph patterns to generate recommendations, similarity measures are considered, e.g., to find similar users. Works on link discovery may also consider specific forms of similarity measures [33], often string similarity measures over labels and descriptions [62].

Other approaches pre-materialise distance values that can then be incorporated into (standard) SPARQL queries. IMGpedia [22] pre-computes a k-nn self similarity join offline over images and stores the results as part of the graph so that users are able to query for similar images. IMGpedia includes relations between images and related entities from DBpedia and Wikidata, further enabling visuo-semantic queries. In Figure 3.3 we present a query performed over IMGpedia that requests images of educational facilities similar to Latin American government palaces, along with the resulting images. 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 [46, 33].

One proposal that supports general similarity joins in SPARQL queries is iSPARQL [35]: an extension to the standard that introduces 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, generating a virtual triple for each pair of compared values. Furthermore, no
SELECT DISTINCT ?img ?img2 ?name ?label WHERE{
  SERVICE <https://query.wikidata.org/bigdata/namespace/wdq/sparql> {
    ?palace wdt:P31 wd:Q16831714 ;
    wdt:P17 ?country .
    ?country wdt:P361 wd:Q12585 .
    OPTIONAL {
      ?country rdfs:label ?cname
      FILTER(LANG(?name)='en' && LANG(?cname)='en')
    }
  }
  ?img imo:associatedWith ?palace ;
  imo:similar ?img2 .
  FILTER(CONTAINS(STR(?wiki),'wikidata.org'))
  SERVICE <https://query.wikidata.org/bigdata/namespace/wdq/sparql> {
    ?wiki wdt:P31/wdt:P279* wd:Q2385804;  
    rdfs:label ?label .
    FILTER(LANG(?label)='en')
  }
}

Lecumberri’s Palace
Haynesville High School
Casa Rosada

Lecumberri’s Palace
Hyogo University
Casa Rosada
TCNJ

Figure 3.3: Visuo-semantic query for IMGpedia [21]
SELECT ?pub1 ?pub2 ?sim WHERE {
    ?pub1 rdfs:label ?title1 ;
    ?pub2 swrc:label ?title2 ;
    IMPRECISE {
        ?sim1 isparql:jac (?title1 ?title2).
        FILTER (?sim1 >= 0.5) .
    }
    FILTER (?sim2 >= 0.5) .
    ?sim isparql:score (?sim1 ?sim2 0.6 0.4).
    FILTER (?sim >= 0.5)
} ORDER BY DESC(?sim)

Figure 3.4: iSPARQL Query that obtains similar books [35].

optimisations are considered for the multidimensional case, thus requiring a nested loop to be performed with $|X| \cdot |Y|$ distance computations. In Figure 3.4 we present a query written in iSPARQL requesting for books with similar titles. In the IMPRECISE block of the query it can be appreciated that after each distance computation it is necessary to filter the virtual triples that do not meet the desired threshold so that those triples do not stack up. The function isparql:score performs a weighted sum of the different similarities.

Galkin et al. [26] propose a multi-way (semantic based) similarity join to determine if two entities obtained from different sources through federated queries are similar. It is based on semantic similarity functions that indicate whether or not two entities are similar using an auxiliary ontology that provides information and facts useful for a similarity score computation, such as logical axioms and class hierarchies.

This limitation is explicitly discussed, with optimisations left for future work [35].
Algorithm 3: Pseudocode for the Quickjoin algorithm for self-similarity joins.

Data: \( X \) a set of objects, \( r \) a threshold distance, \( \delta \) a distance function, \( c \) an integer

Result: \( \text{result} \) a set of pairs of objects, initially \( \emptyset \)

1. **Function Quickjoin**\((X, r, \delta, c)\):
   1. if \( |X| \leq r \) then
   2. return \text{NestedLoop}(X, r, \delta)
   3. end
   4. \( p_1 \leftarrow \text{random\_object}(X) \);
   5. \( p_2 \leftarrow \text{random\_object}(X - \{p_1\}) \);
   6. \((\text{partL}, \text{partG}, \text{winL}, \text{winG}) \leftarrow \text{partition}(X, r, \delta, p_1, p_2) \);
   7. \( \text{result} \leftarrow \text{result} \cup \text{Quickjoin}(\text{partL}, r, \delta, c) \);
   8. \( \text{result} \leftarrow \text{result} \cup \text{Quickjoin}(\text{partG}, r, \delta, c) \);
   9. \( \text{result} \leftarrow \text{result} \cup \text{QuickjoinWindow}(\text{winL}, \text{winG}, r, \delta, c) \);
   10. return \text{result} \;

2. **Function partition**\((X, r, \delta, p_1, p_2)\):
   1. \( \text{partL}, \text{partG}, \text{winL}, \text{winG} \leftarrow \emptyset \);
   2. \( \text{radius} \leftarrow \delta(p_1, p_2) \);
   3. for \( x \in X \) do
   4. \( \text{dist} \leftarrow \delta(x, p_1) \);
   5. if \( \text{dist} < \text{radius} \) then
   6. \( \text{partL} \leftarrow \text{partL} \cup \{x\} \)
   7. else
   8. \( \text{partG} \leftarrow \text{partG} \cup \{x\} \)
   9. end
   10. if \( \text{radius} < \text{dist} \leq \text{radius} + r \) then
   11. \( \text{winG} \leftarrow \text{winG} \cup \{x\} \)
   12. else if \( \text{dist} \geq \text{radius} - r \) then
   13. \( \text{winL} \leftarrow \text{winL} \cup \{x\} \)
   14. end
   15. return \( \text{partL}, \text{partG}, \text{winL}, \text{winG} \)

3. **Function QuickjoinWindow**\((\text{winL}, \text{winG}, r, \delta, c)\):
   1. if \( |\text{winL}| + |\text{winG}| < c \) then
   2. return \text{NestedLoop}(\text{winL} \cup \text{winG}, r, \delta)
   3. end
   4. \( p_1 \leftarrow \text{random\_object}(\text{winL} \cup \text{winG}) \);
   5. \( p_2 \leftarrow \text{random\_object}((\text{winL} \cup \text{winG}) - \{p_1\}) \);
   6. \((\text{partL}_1, \text{partG}_1, \text{winL}_1, \text{winG}_1) \leftarrow \text{partition}(\text{winL}, r, \delta, p_1, p_2) \);
   7. \((\text{partL}_2, \text{partG}_2, \text{winL}_2, \text{winG}_2) \leftarrow \text{partition}(\text{winG}, r, \delta, p_1, p_2) \);
   8. \( \text{result} \leftarrow \text{result} \cup \text{QuickjoinWindow}(\text{winL}_1, \text{winG}_2, r, \delta, c) \);
   9. \( \text{result} \leftarrow \text{result} \cup \text{QuickjoinWindow}(\text{winG}_1, \text{winL}_2, r, \delta, c) \);
   10. \( \text{result} \leftarrow \text{result} \cup \text{QuickjoinWindow}(\text{partL}_1, \text{partL}_2, r, \delta, c) \);
   11. \( \text{result} \leftarrow \text{result} \cup \text{QuickjoinWindow}(\text{partG}_1, \text{partG}_2, r, \delta, c) \);
   12. return \text{result}
Chapter 4

An Efficient Algorithm for Approximated Similarity Joins

In this chapter we present a simple algorithm that computes an approximation of the $k$ nearest neighbour self-similarity join problem in metric spaces. It is grounded in the intuition that the nearest neighbours of an object are likely to be nearest neighbours among themselves, thus forming groups within the datasets. Contrary to LSH [28], the groups created are not created by hyperplane cuts, but by choosing centres and assigning elements to the group of the closest centre. Our algorithm is a generalisation of a previous work that presented an heuristic for solving the 1-nn problem [24]. The algorithm presented here computes $\Theta(n^{3/2})$ distances even in the worst case and gives an approximation with up to 46% of precision for 1-nn self-similarity queries and 39% for 16-nn. We show how the algorithm performs with three different real datasets: the English dictionary, very high-dimensional deep-feature vectors, and a massive collection of classic visual descriptors.

4.1. Preliminary Work

We propose an heuristic that solves approximately the 1-nn self-similarity join [24]. Solving the problem using a simple brute-force algorithm requires $\binom{n}{2}$ distance calculations, since it requires to compare every element against all others. We propose a simple divide-and-conquer algorithm that gives an approximated solution for the self-similarity join that computes only $\Theta(n^{3/2})$ distances.

The algorithm receives a set of points $D$ as input, with size $n$ and outputs a set of pairs $S \subseteq D \times D$ where every element of the set is paired with some other element expected to be its first nearest neighbour or a good approximation otherwise.

The algorithm works as follows: first we select a set of $\sqrt{n}$ centres among the elements of the input; second, the rest of the elements are partitioned in different groups, one group per centre, such that each object falls within the group of the closest centre, where each group has a maximum capacity of $\sqrt{n}$ elements; third, within each group we compute the brute-force algorithm to obtain the nearest neighbour for each of the elements (a nested loop); and
finally, all partial results are aggregated and returned. It is trivial to note that the algorithm computes $\Theta(n^{3/2})$ distances.

A preliminary evaluation of the algorithm was performed, comparing the algorithm with another version of itself that uses a greater group capacity: $\sqrt{n}$ v. $2\sqrt{n}$. The algorithm runs over a dataset of 900 thousand 72-dimensional feature vectors taken from IMGpedia [22]. The ground truth was computed using a nested loop. The version of the algorithm with capacity for $2\sqrt{n}$ elements performed best in terms of precision. In Table 4.1 we present the results of the algorithm with group size of $2\sqrt{n}$, where we see that 30% of the objects found the correct nearest neighbour, and around 78% of them were paired with an object within their 10 nearest neighbours, according to the cumulative percentage.

### 4.2. The Algorithm

The algorithm here presented is a generalisation of the one presented in the previous section, extending self-similarity joins from 1-nn to $k$-nn and improving its performance. Given a dataset $D \subset \mathbb{U}$, with $|D| = n$, and a metric function $\delta$ defined over $\mathbb{U}$, the algorithm selects $\sqrt{n}$ objects as centres, where each centre defines a group. Then, the algorithm distributes the remaining objects into the groups such that each object is in the group with the closest centre, with respect to distance $\delta$. Groups have a maximum size of $c\sqrt{n}$ objects, where $c$ is a parameter. Given the size restriction, if the group where an object should be sorted into is already full, then it is sorted into the next closest group. The criteria for selecting the centres and the algorithm to form the groups can be re-defined by the users. When grouping is finished, the algorithm searches for the $k$ nearest neighbours of each object from a fixed set of candidates: the group of the current object and another group that is the closest to the object.

Algorithm 4 presents the pseudo-code for the proposed algorithm. Routines `select_centres` and `partition` choose the $\sqrt{n}$ centres and assign the remaining objects into the groups respectively. For each element in a group, the target set is formed to contain the suitable $k$-nn candidates. The `next_closest_group` subroutine finds a group, different from the current one, that is closest to the element $e$. This means, obtain the group $G$ such that $\delta(G.centre, e) - G.radius$ is minimum. The `get_k-nn` routine computes the distance between an object $e$ and all elements in `target`, keeping the approximated $k$ nearest neighbours of $e$.

For the purposes of this work, centres are chosen at random. There are techniques for choosing evenly distributed objects in metric spaces [11] that can be applied at this point;
Algorithm 4: Algorithm for approximated $k$-nn self-similarity join.

**Data:** Data, a set of objects; $c$ an integer; $k$ the number of neighbours to find

**Result:** result, a set of pairs of objects

1. centres ← select_centres(Data);
2. groups ← partition(Data, centres, $c$);
3. result ← $\emptyset$;
4. for group ∈ groups do
   5. for $e$ ∈ group do
      6. do
      7.      target ← group $\cup$ next_closest_group($e$);
      8.      while $|\text{target}| < k$;
      9.      partial ← get_k-nn($k$, $e$, target);
     10.      result ← result $\cup$ partial;
    11.  end
5. end
6. return result

however, the maximum distance of the metric space must be known beforehand, a parameter must be fine-tuned, and there is no way to select the number of objects that the algorithm chooses as it relies in the intrinsic dimensionality of the space. Yianilos, along with the definition of the VP-trees, describes a way to choose elements as far apart as possible considering the mean and standard deviation of the distance distribution of a subset of the elements [66]; however it requires further parameters to select a subsample of the objects to compute the mean of the distances among them and maximising the standard deviation.

We propose that the partition process, given a set of centres, tries to assign each object into the group of the closest centre if it has space; if not, the object will be assigned to the second-closest group; if it is also full the algorithm will attempt with the next closest, and so on until the object is assigned. The maximum size of the groups is $c\sqrt{n}$, where $c$ is a constant parameter, usually lower than 10. Figure 4.1 shows a 2D example of how the partition of the elements can change as the value of $c$ increases: 5 thousand randomly-generated points are distributed into $\lceil \sqrt{5000} \rceil = 71$ groups using Manhattan distance; groups are shown with different colours. Above each graph, the respective value of $c$ is presented. It can be seen that when $c = 1$ the groups do not have clear boundaries separating each other, and as $c$ increases to 2 and 3, the groups seem better-defined and well-separated from the others. Using higher values of $c$ does not show much change in the partition results; however, in higher dimensions it might have an impact. Other partition criteria can be used, as long as the number of centres and maximum size of groups remain as $O(\sqrt{n})$. We attempted to assign each object to the group that needed to grow the least in order to include it; however we found that the first group to accept an object is filled until its maximum capacity before allowing other groups to grow; therefore, it causes groups to become less compact, which is a consequence of the curse of dimensionality (variance of the distances diminishes with dimension increasing). Algorithm 5 presents the pseudo code for the partition process that uses the first described criteria. We improve the efficacy of the algorithm by computing exactly the $k$-nn of each centre, since the distances from all objects to all centres are known after this step.
4.2.1. Complexity Analysis

In this section, we discuss the complexity of the algorithm in terms of the number of distances it computes. We propose that given a reasonable constraint on the number $k$ of nearest neighbours, our algorithm computes $\Theta\left(\frac{n^3}{2}\right)$ distances.

**Theorem 4.1** The proposed algorithm for approximated $k$-nn self-similarity join computes $\Theta\left(\frac{n^3}{2}\right)$ distances, where $k$ is a constant.

**Proof.** The random selection of $\sqrt{n}$ centres does not involve distance computations. In the partition phase, we compute the distance between all the centres and all the remaining objects, being in total $(n - \sqrt{n}) \cdot \sqrt{n} = n^{3/2} - n = \Theta(n^{3/2})$ distances. In the next phase we compute the distance between each object $q$ and all other objects in its own group and the next groups closest to $q$ which are called the target set of $q$; note that as many groups are added to the target until it has a size of at least $k$. The worst case occurs when the target set has $k - 1$ elements, thus another group needs to be added and it happens to contain the maximum number of objects, $c\sqrt{n}$. Therefore, the target set has a size of $c\sqrt{n} + k - 1 < c\sqrt{n} + k = \Theta(\sqrt{n})$. Finally, if every element falls in the worst case, we would compute $n$ sequential scans over sets of size $\Theta(\sqrt{n})$ which requires $n \cdot \Theta(\sqrt{n}) = \Theta(n^{3/2})$ distance computations and proves the result.
Algorithm 5: Proposed partition strategy

**Data:** data, the set of elements; centres, the list of centres; c the constant for max size of groups

**Result:** groups, a list of sets that partition the data

1. \( \text{groups}[i] \leftarrow \{ \text{centres}[i], \forall i, 0 \leq i < |\text{centres}| \} \);
2. \( \text{maxSize} \leftarrow c\sqrt{|\text{data}|} \);
3. for \( \text{obj}_i \in \text{data} \) do
   4. \( D \leftarrow \emptyset \);
   5. for \( \text{centre}_j \in \text{centres} \) do
      6. \( D \leftarrow D \cup \delta(\text{obj}_i, \text{centre}_j) \);
   7. end
   8. sort\((D[i])\);
   9. bestGroup \( \leftarrow \) the group of the centre in \( D[0] \);
10. repeat
    11. if not isFull(bestGroup) then
    12. bestGroup \( \leftarrow \) bestGroup \( \cup \) \{\text{obj}_i\};
    13. else
    14. bestGroup \( \leftarrow \) the group of the centre in next\((D)\);
    15. end
16. until \( \text{obj}_i \) is added to a group;
17. end

4.2.2. Implementation

The algorithm is implemented in Python and is publicly available[^1]. The module contains a main function called self._sim_join that receives a row-major numpy matrix with the data, in the case of vector objects, or a filename where to read in the case of strings. The method also receives the parameters \( c \) and \( k \). The function first calls a sub-routine for the random selection of the \( \sqrt{n} \) centres which are then separated from the rest of the data. Later, the groups are formed using the aforementioned criteria, thus computing the distances between the centres and all other objects. For optimisation and precision purposes, it stores relevant similarity relations among the centres and the rest of the data. Nearest neighbour candidates are stored in priority queues, one per each object, so whenever a closer object is found it replaces the current candidate at the furthest distance. In the final stage, for each group and for each element in the group a sequential search is performed: the element is compared against all others in its own group and all elements in the one other group that is closer; in the rare case that both groups combined do not have \( k \) objects, further groups are added until there are at least \( k \). As per before, every time a better candidate for nearest neighbour is found in the sequential scan the object is added to the priority queue, as the further candidate is removed. Finally, the set of priority queues is returned as the result of the self-similarity join.

[^1]: https://github.com/scferrada/self-sim-join
4.3. Evaluation

In this section we describe the experimental design used to test the algorithm. We present and discuss the results obtained. We finish by comparing the algorithm to others.

4.3.1. Experimental Settings

We tested our algorithm on three real-world datasets:

- **STRINGS**: 46,801 words from the English dictionary, using edit distance, and considering only words with 4 or more letters.

- **HOG**: 928,276 visual feature vectors of 72 dimensions, taken from the IMGpedia dataset [22]. We use Manhattan distance to compare.

- **DECAF**: 39,327 deep-visual features DeCAF7 [20] of 4092 dimensions compared with Manhattan distance.

We run the algorithm 100 times on each dataset, computing 1, 2, 8 and 16 nearest neighbours self-similarity join. We use six values of \( c \): 1, 2, 3, 10, 100 and 1,000. All experiments were executed on a machine with Debian 4.1.1, a 2.2 GHz 24-core Intel® Xeon® processor, and 120GB of RAM. The ground truth is computed using a nested loop.

4.3.2. Average Precision

Table 4.2 shows the average precision of the algorithm for each dataset and value of \( k \) and \( c \). The average precision of the algorithm increases as the value of \( c \) increases. The biggest increment occurs between \( c = 1 \) and \( c = 2 \) where the precision increases about 14 percentual points (pp) in the vector spaces and about 8 pp in strings. When \( c = 3 \) the average precision increases only around 5.5 pp versus \( c = 2 \) in the vector spaces, and only 1 pp in strings. For higher values of \( c \), average precision also increases but variance increases as well. This increase in variance explains why in some cases the average precision using \( c = 100 \) or 1000 is not much better than with \( c = 10 \). Thus, we propose that an optimal value for \( c \) is between 2 and 10. In Figure 4.2 we present how the average precision changes with different values of \( c \) when \( k = 16 \), and the previously mentioned behaviour is evident.
Table 4.2: Average Precision of the Algorithm for the different Datasets, values of $c$ and values of $k$

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$c$</th>
<th>$k = 1$</th>
<th>$k = 4$</th>
<th>$k = 8$</th>
<th>$k = 16$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECAF</td>
<td>1</td>
<td>18.15%</td>
<td>17.22%</td>
<td>16.32%</td>
<td>15.48%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>31.75%</td>
<td>30.26%</td>
<td>28.65%</td>
<td>27.25%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>37.94%</td>
<td>35.69%</td>
<td>34.26%</td>
<td>32.38%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>44.31%</td>
<td>41.41%</td>
<td>39.14%</td>
<td>38.19%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>44.80%</td>
<td>41.93%</td>
<td>40.31%</td>
<td>37.79%</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>45.16%</td>
<td>41.92%</td>
<td>41.21%</td>
<td>38.49%</td>
</tr>
<tr>
<td>HOG</td>
<td>1</td>
<td>20.18%</td>
<td>18.90%</td>
<td>18.06%</td>
<td>17.16%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>36.80%</td>
<td>34.32%</td>
<td>32.80%</td>
<td>31.10%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>42.18%</td>
<td>39.53%</td>
<td>37.89%</td>
<td>35.29%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>45.77%</td>
<td>42.77%</td>
<td>41.29%</td>
<td>38.94%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>45.74%</td>
<td>42.68%</td>
<td>40.97%</td>
<td>38.81%</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>46.00%</td>
<td>42.84%</td>
<td>41.06%</td>
<td>39.03%</td>
</tr>
<tr>
<td>STRINGS</td>
<td>1</td>
<td>15.54%</td>
<td>15.86%</td>
<td>15.08%</td>
<td>13.24%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>23.66%</td>
<td>24.55%</td>
<td>23.08%</td>
<td>20.11%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24.68%</td>
<td>26.29%</td>
<td>25.46%</td>
<td>22.56%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>25.07%</td>
<td>25.87%</td>
<td>25.06%</td>
<td>22.92%</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>25.82%</td>
<td>26.71%</td>
<td>25.25%</td>
<td>22.45%</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>24.64%</td>
<td>26.92%</td>
<td>24.05%</td>
<td>22.16%</td>
</tr>
</tbody>
</table>

The average precision also decreases as the number of requested nearest neighbours increases. This is because having more elements to find increases the chances for the algorithm of finding wrong ones. For $c = 3$ we report a difference of almost 7 pp of precision in the HOG dataset between 1-nn and 16-nn self-similarity joins; this difference is of 5.5 pp in the DECAF dataset, and 2 pp in the STRINGS dataset. For higher values of $c$ this trend is also present with the precision dropping between 1 and 2 percent when $k$ increases from 8 to 16.

The difference between the precision of the algorithm in the vector spaces can be explained by the dimensionality of the data. As HOG is a 72-dimensional space and DECAF a 4098-dimensional space, the distribution of distances is greatly different between both datasets, presenting a low mean and high variance in the former and a high mean and a low variance in the latter. This behaviour is known as the curse of dimensionality. As the dimension grows the quality of the results might become less satisfactory. The issue of the quality of a $k$-nn query results in very high-dimensional spaces is discussed by Hinneburg et. al [31] where they define a quality metric and compare different metric distances, concluding that Manhattan distance usually retrieves better quality results than other $L_p$ distances.

The average precision of the algorithm using the STRINGS dataset with edit distance is worse than when using the other two datasets. This is mostly because when employing discrete distances it becomes odd to talk about nearest neighbours since more than $k$ elements can be at an exact given distance $d$. In such a case, which elements should be retrieved? Are those elements chosen by the brute-force algorithm aligned with those chosen for our
algorithm? The answers to these questions depend on assumptions that must be made beforehand, redefining the nearest neighbour search. Hence, the results over the vector spaces with continuous distances are better than in string spaces with discrete distances. This kind of behaviour also explains that in the STRINGS dataset the precision increases from $k = 1$ to $k = 4$.

### 4.3.3. Precision Distribution

We present the distribution of the number of correctly found nearest neighbours per object. To obtain this, we used 100 runs of the algorithm over the DECAF dataset and compute the number of correct matches per object in the computation of the 16 nearest neighbours self-similarity join. In Figure 4.3 we present six histograms that display the percentage of elements that have $i$ correctly found neighbours, being $i$ a bin of the histogram. A histogram of the ground truth would show a single bar on the sixteenth bin with length 1. We see that the distributions are single-peaked and skewed right. The figure also presents an outlier peak in the last bin (the elements for which the algorithm finds all correct 16-nn). When $c = 1$ the results tend to be poor, being most of the elements matched with none of its actual 16 nearest neighbours. When $c$ increases, the results improve remarkably: the elements with 0 matches decrease considerably and the distribution becomes more uniform. With higher values of $c$ we see further improvement; however the marginal increase in precision is not substantial when increasing the group size given that the distributions for $c = 10, 100$ and $1000$ are similar. In each case, around 7% of the objects fail to match any of their 16-nn.

Figure 4.3: Distribution of the number of correctly found 16-nn using DECAF dataset.
4.3.4. Execution Time and Distance Calculations

In Figure 4.4 the execution time for the algorithm in the different experimental settings is displayed. Times are provided as an average and a variance, since multiple runs of each experiment setting are considered. It can be seen that the variance of the execution time greatly increases with \( c \) due to the potential formation of a few massive groups and many small groups, where the search in big groups dominates the time. Since computing distances is more expensive as the dimension of the space increases, DECAF average times are higher than HOG times for higher values of \( c \). When considering low values of \( c \), time depends more on the size of the dataset than on the dimensionality of the objects; however, as \( c \) increases the dependency of execution time on the number of objects becomes less clear. This behaviour can be appreciated in Figure 4.5 where \( k = 16 \); it can be seen that the number of required nearest neighbours does not seem to correlate with the final execution time when \( k \) is much less than \( \sqrt{n} \), because the candidate results are stored in priority queues and their management does not require extra distance computations.

As per the number of computed distances, Figure 4.6 shows the values in logarithmic scale for each dataset and value of \( c \). In all datasets, the average number of computed distances increases with \( c \), as well as the variance. The last set of bars of the graph present the distances computed by a brute force approach. In the DECAF dataset, the increment is more notorious due to the higher dimension of the vector space. The average number of distances computed in the DECAF space is even greater than the one of STRINGS, despite being a smaller set.
When comparing the number of computed distances with the execution time, we can conclude that distance computations are more time consuming when the dimension of the space gets higher, thus dimension has a greater impact on computation time even than the size of the dataset: the algorithms takes much more time in DECAF than in HOG, when DECAF is much smaller than HOG. Finally, when comparing with a brute force approach we can see that our algorithm takes up to 3 fewer orders of magnitude.

4.3.5. Centre Selection Strategies

Results in previous sections consider a random selection of centres. Here we present results using a more sophisticated technique. In Section 4.2 two approaches were discussed: the automatic pivot selection of Brisaboa et al. [11] and the sampling technique used by Yianilos [66] when building vp-trees. The former technique cannot be applied to this algorithm since it does not contemplate a way to define the number of pivots to be returned. The latter
approach requires to set a sample size, since the algorithm selects those objects in a random sample that have the greatest spread w.r.t. all other objects.

Algorithm 6 presents the pseudo-code of the strategy. It selects $c\sqrt{n}$ random objects to be candidate centres (the original work proposes constant size samples). Each of the candidates is compared with another random sample of the same size as the previous sample. The median and standard deviation of the distances is computed. Using a max-heap, we keep the objects that have the maximum standard deviation and return them as centres. In Figure 4.7 we display how 5 thousand random 2-dimensional points distribute into 71 groups using this strategy for the different values of $c$; it can be seen that well-formed groups start to appear with $c = 10$ and are mostly unchanged with higher values.

We tested our algorithm selecting $\sqrt{n}$ objects with the technique from Yianilos. We run the algorithm several times using the HOG dataset, $k = 4$ and $c = 1, 2, 3, 10, 100, 1000$. Results are presented in Table 4.3, where it can be seen that the achieved average precision of the algorithm is consistently worse using vp-tree-like centre selection than with a random selection, even with high values of $c$ where, as can be seen in Figure 4.7 when groups are well-delimited, the average precision drops to $\sim 12\%$. Further experimentation can be done: using increasingly larger samples of the data could yield better results. Nonetheless, we explain the poor performance of Yianilos technique by the reasoning that is given in their work: vantage points are meant to be close to the corner of the space so those points are more useful to divide the space into balanced partitions, which is a different task than to group objects closer together.

Figure 4.7: Partition of random 2D points using the centre selection strategy of Yianilos [66].
Algorithm 6: centre Selection Strategy based on Yianilos

**Data:** data, the set of elements; c the constant for max size of groups

**Result:** centres, a list of the objects chosen as centres

1. centres ← φ;
2. candidates ← Heap(\{(⊥, 0) \forall i \leq c\sqrt{|data|}\});
3. \(N \leftarrow c\sqrt{|data|}\);
4. sample ← choose_random_noreplace(data, N);
5. for \(obj_i \in sample\) do
   6. \(D \leftarrow choose_random_noreplace(data, N)\);
   7. \(dist \leftarrow distance(obj_i, D)\);
   8. \(\mu \leftarrow median(dist)\);
   9. \(\sigma \leftarrow stdv(dist - \mu)\);
   10. if \(\sigma > candidates.peek\) then
       11. \(candidates.pop \leftarrow push((obj_i, \sigma))\);
   end
13. end
14. centres ← candidates;
15. return centres;

Table 4.3: Average precision of the algorithm using vp-tree-like centre selection.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>(k)</th>
<th>(c = 1)</th>
<th>(c = 2)</th>
<th>(c = 3)</th>
<th>(c = 10)</th>
<th>(c = 100)</th>
<th>(c = 1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOG</td>
<td>4</td>
<td>18.98%</td>
<td>31.29%</td>
<td>31.25%</td>
<td>11.94%</td>
<td>12.04%</td>
<td>12.08%</td>
</tr>
</tbody>
</table>

4.3.6. Comparison with Other Similarity Join Algorithms

We found that, for range-based similarity joins, Quickjoin is widely used \[54, 56\]. Quickjoin was extended to support \(k\)-distance joins \[25\], i.e., to obtain the \(k\) minimum distances between any pair of elements of a dataset. We propose another extension that builds upon the \(k\)-distance join and computes a \(k\)-nn self-similarity join. As defined in \[25\], we run Quickjoin twice. The first time, we perform a range-based similarity join with \(\varepsilon = 0\) where we collect the \(k\) nearest neighbours found in that setting, which is an approximation of the real answer. Using a threshold distance 0 means that, in practice, all the window partitions of Quickjoin are always empty. The second run uses the maximum distance found for a candidate nearest neighbour in the previous run as the value for \(\varepsilon\); the value can be decreased over time, whenever the maximum candidate distance decreases. Using such an algorithm brings with it two problems regarding efficiency and efficacy. The first run of Quickjoin does not necessarily compute at least \(k\) distances per element, which can affect the precision by not getting a more fit boundary. Since the maximum distance found in the first run is usually near the average distance of the space (especially in very high-dimensional spaces due to the curse of dimensionality), the window partitions of the second run of Quickjoin tend to contain a higher fraction of the objects, which implies that the time it would take to complete the second run will be closer to quadratic.

We use this extension to Quickjoin to compute \(k\)-nn self-similarity joins. Quickjoin requires
a parameter $c$ that represents the minimum number of objects in a partition so it can be processed with a nested loop. We run the algorithm 100 times using the DECAF dataset, $k = 4$ and $c$ as 0.1% of the data. We use only this scenario due to time and memory constraints: The execution of this scenario takes 7 days using multiple threads, and with higher values of $k$ the algorithm crashes due to lack of memory (this happens mostly because of the window partitions over-replicating the data). This modified version of Quickjoin reaches a 29.49% average precision, compared with 41.41% of our algorithm in the same scenario. Our algorithm is also faster, it takes 47.5 minutes on average to complete one run over the DECAF dataset, whereas modified Quickjoin takes around 19 hours (again, this is due the size of the window partitions, which greatly increases the number of required recursive steps).

LSH requires several parameters: The number of hash tables to be used, the number of hash functions of each table, and the parameters required for computing the hash functions. We followed the suggestion of Datar et al. [17], and use 30 tables with 10 hash functions, and for each function we draw a standard Cauchy distribution parameter and a uniformly distributed parameter. Using this setting with the DECAF dataset gives a poor performance for the algorithm in terms of average precision, reaching only 8.15% for $k = 1$, 7.44% for $k = 4$, and 5.62% for $k = 16$. We argue that this poor precision has to do not only with the parameter selection, which needs to be fine-tuned, but it is mostly due to the high dimension of the space, which implies that the locality is not well captured by the proposed hash functions. In terms of time, LSH takes up to 8 minutes to complete.

In Table 4.4 we summarise the results of the comparison of Root-Join with the modified Quickjoin and LSH with the settings described previously: DECAF dataset, $k = 4$. The summary contains both precision and running time.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (min.)</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root-Join</td>
<td>47.5</td>
<td>41.41%</td>
</tr>
<tr>
<td>Quickjoin</td>
<td>10,222.0</td>
<td>29.49%</td>
</tr>
<tr>
<td>LSH</td>
<td>8.4</td>
<td>7.44%</td>
</tr>
</tbody>
</table>
Chapter 5

Similarity Joins in SPARQL Queries

In this chapter, we motivate similarity-based queries for SPARQL and present a system for executing such queries over RDF graphs and integrating them with a SPARQL Engine [23]. Given that this is the first work to address this topic, we outline the unique challenges that arise when considering similarity queries in an RDF/SPARQL setting. We propose the syntax and semantics of a SPARQL extension that supports similarity queries. We further introduce and implement various low-level physical operators for evaluating similarity joins within a SPARQL engine, and discuss their trade-offs. We propose a benchmark based on Wikidata [63] that includes atomic similarity queries composed of a single self-similarity join, as well as complex similarity queries that combine similarity joins with other SPARQL operators. Finally, we provide a comparison to the work that is closest to ours: DBSimJoin [54], which is a SQL extension for range similarity joins built upon Quickjoin [34].

5.1. 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 [35]; 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 \texttt{WHERE} clause of a SPARQL query; furthermore, we allow the computed distance measure to be bound to a variable, allowing 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 and defined 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 distance thresholds for ranges can be unintuitive in abstract, high-dimensional metric spaces.

5.2. Challenges

To the best of our knowledge, we are the first work to consider extending SPARQL with similarity joins. During this work we encountered a number of conceptual challenges arising from the particular characteristics of SPARQL/RDF, some of which do not have unique solutions. In this section, we thus enumerate these challenges, discussing not only the solution we currently propose, but also alternative solutions that could be explored in future work, and in so doing, we outline a possible roadmap for further research.

In this section we discuss such conceptual challenges, which need to be overcome in order to provide support for similarity joins in SPARQL queries. There are no unique solutions for these challenges; we propose initial solutions that can be further developed in the future.

5.2.1. Size of the Data

As it was stated in Chapter 2, performing a similarity join between two sets of size \( O(n) \) requires \( O(n^2) \) distance computations in the worst case: as the data involved grows larger, the cost of such joins increases quadratically. This challenge can be dealt with through different techniques. The first one is to implement similarity joins using efficient matrix operations that compute multiple distances at once, thus outperforming a naive, single-vector-at-a-time nested-loop implementation. Indexing techniques also arise as a possible solution, but since we propose a dimension-agnostic framework, only partial and/or online indexing may be achievable (discussed next). Another option may be to perform space-partitioning in query evaluation time to try to “chunk” similar entities into buckets. Approximation may also be useful in cases where the user is willing to trade precision/recall for time.

Solution proposed: The main contribution of this work is to investigate all four strategies for similarity join implementation: nested-loop, (online) tree indexing, space-partitioning algorithms and approximations, designing a benchmark for their comparison in the RDF/S-PARQL setting. Furthermore, given that a similarity join will be embedded as part of a larger query, there may be possibilities for optimisations based on query rewriting; we will discuss such optimisations and rewritings later in Section 5.8.

5.2.2. Dimension-Agnostic Indexing

If we consider using indexing techniques to evaluate similarity queries while allowing users to specify the dimensions on which similarity should be computed, we will run into technical challenges. The most straightforward approach is to build an offline index over the RDF graph, where the naive approach could be to index every possible subset of dimensions in the data, potentially leading to \( 2^d - 1 \) indexes for \( d \) different dimensions; added to this,
datasets like Wikidata have thousands of properties, each of which can form a dimension. One approach might be to build base indexes on all combinations of attributes appearing on common subjects in the data, but this again is still exponential (with respect to the maximum number of relevant attributes/properties on a single subject); maintaining such indexes would also increase the costs of updates. In any case, when applying similarity joins in SPARQL queries, users can dynamically generate new attributes in a query that do not appear in the data, for example, based on the result of an aggregation function (count, sum, etc.); these “dynamic attributes” preclude the possibility of having a complete offline index to cover all possible similarity joins.

**Solution proposed:** Rather than forego offline indexing entirely, in this work we investigate online indexes that are applied on the results of relations generated by the query; though such a strategy incurs the possible overhead of indexing at query time, it may still provide performance benefits by avoiding brute-force comparisons in favourable cases, especially if the construction of the index takes subquadratic time (we will return to address this question in our evaluation in Section 4.3).

**Alternatives:** Another option that may be explored in future is to adopt a lazy indexing approach: to build an index only when a query requires it and then keep it in an index cache, removing the indexes used least recently to make space for new indexes; work has shown this to be competitive with an online (index-free) strategy [58]. On the other hand we chose not to explore this model as it was later shown that under a more realistic cost model, no online strategy could be \( c \)-competitive for any constant \( c \) [39]. In summary, offline or lazy indexing strategies are not easily applied to the RDF/SPARQL setting, but with adaptations, it may be possible to use such techniques for partial evaluation of a similarity join.

5.2.3. Multi-valued Attributes

Triple patterns can have multiple objects for the same pair of subject and predicate \((s, p)\) in different points of time, units of measurement, decimal precision, languages, calendars and many others. In the context of RDF and SPARQL, assuming that a given pair \((s, p)\) only appears once in a relation would lose generality; for example, although in our experimentation we use the “truthy” version of Wikidata that picks a preferred value for properties, in later experiments we still find triples with multiple object values, such as physical properties in multiple measurement units, etc. On the one hand, the user writing the query should take care to ensure that attributes on both sides of the similarity join are comparable and – if desired – that the subjects form keys, aggregating multiple values (e.g., with \( \text{MAX} \), \( \text{MIN} \), \( \text{AVG} \), etc.) as appropriate. On the other hand, to avoid loss of generality, similarity joins should be applicable in the case that descriptors are not unique but have multiple combinations of joined values.

**Solution proposed:** We consider \text{ANY} semantics: given a dimension \(?o\) involved in the similarity join, we will consider all the solution mappings of \(?o\) of a same entity independently.

**Alternatives:** A stricter semantics in the case of attributes for multiple values may be \text{ALL}, where in order to include an entity in the result, all its bindings of dimension \(?o\) must...
satisfy the similarity condition.

5.2.4. Incomplete Data

The Open World Assumption at the heart of RDF acknowledges that data are not complete. Likewise, SPARQL features like `OPTIONAL` generate missing (unbound) values. Furthermore, RDF allows for the use of blank nodes to represent values known to exist, but whose value is not known; this is also used in Wikidata, where, for example, the date of death of Jack the Ripper (Q43963) is stated to be a blank node, representing an unknown value. How should missing values be treated by distance measures?

**Solution proposed:** We adopt a conservative approach, which is to consider unbounds and blank nodes to be incomparable values (including amongst each other; e.g., an unbound is incomparable with another unbound), filtering such results from the similarity query. In other words, we assume complete data in the solutions over which the similarity join is applied, filtering incomplete tuples. Researching ways in which unknown values can be considered during the computation of distances is left as future work.

**Alternatives:** There may be techniques to overcome the incompleteness issue. For example, if a maximum and minimum value can be estimated for the missing value, then a conservative distance can be computed as the maximum distance possible in the given range; this may help preserve useful results where a pair of entities are very similar for \( n - 1 \) dimensions but the \( n \)th dimension is missing on one of the entities. A more sophisticated method would be to estimate a probability density function to model missing values, identifying similar entities with a probability guarantee [15].

5.2.5. Value Distribution

Given that we use sets of attributes not known **a priori**, the distributions of these attribute values are also unknown and may present several orders of magnitude difference across attributes; for example, a generic distance measure applied over countries with respect to population and number of cities may put undue weight on the much higher absolute values of population. In this case, a small difference in the higher values would have a bigger impact in the similarity score than a difference in the lowest value.

**Solution proposed:** To prevent some dimensions dominating others, normalisation and/or weighted distance functions can be applied. We currently apply linear normalisation of dimensions for each attribute into the interval \([0, 1]\), where 0 is the minimum and 1 is the maximum of all values for that attribute on either side of the similarity join. Such an approach provides a general way to equalise the importance of attributes that, by their nature, happen to have different magnitudes in terms of absolute values. Ideally, this behaviour can be turned on and off by the final user depending on their needs.

**Alternatives:** Another approach may be to use the maximum and minimum values of an attribute’s property in the graph (not just those returned by the query) to perform normalisation. We do not choose this approach as it would not be directly applicable in the
case of “dynamic attributes” – defined through aggregations, functions, etc. – that compute novel values not appearing in the data. A limitation of linear normalisation is that it is sensitive to extreme maximum or minimum values that will lead to typical values being squeezed into a small interval; other forms of normalisation based on ranks, logarithms, etc., could also be explored.

### 5.2.6. Abstract Distances

When working with distances in domains such as geospatial data, distances have a physical intuition (e.g., 100 metres). However, when working with arbitrary combinations of attributes, such as population and number of cities of a country, the resulting distances are far from intuitive. This is particularly an issue for range queries, where users may not be able to intuit a suitable distance to add as a range parameter in such queries.

**Solution proposed:** As aforementioned, we propose to support not only range queries, but $k$-nearest neighbour queries, which do not require specifying a distance value in the query, but rather a $k$ value, which should be more intuitive for abstract spaces.

### 5.3. 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 [30], 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 set of 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).

In Figure 5.1, we propose our extension to the SPARQL 1.1 EBNF Grammar [30], adding one new production rule (for `SimilarityGraphPattern`) and extending one existing production rule (`GraphPatternNotTriples`). All non-terminals aside from `SimilarityGraphPattern` are interpreted as defined in the standard EBNF Grammar [30]. The keyword `ON` is used to define the variables in both graph patterns with respect to which the distance is computed; the keywords `TOP` and `WITHIN` denote a $k$-nn query and an $r$-distance 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
SELECT ?country1 ?country2 ?distance WHERE {
  { ?country1 a :Country;
    :population ?pop1 . }
  SIMILARITY JOIN ON (?pop1) (?pop2)
  WITHIN 0.2 DISTANCE :euclidean AS ?distance
  { ?country2 a :Country;
    :population ?pop2 . }
}

Figure 5.2: Example extended SPARQL query computing a range similarity join

be fresh, i.e., to not appear elsewhere in the SimilarityGraphPattern (similar restriction to BIND operator). The syntax may be extended in future to provide further customisation, such as supporting different normalisation functions, or to define default parameters or default values for unknown dimensions. We remark that although other join operators in SPARQL are defined based on natural equi-joins (joining variables with the same name using equality), similarity joins are not equi-joins, and thus cannot be written as a natural join; otherwise a variable in a single solution may take two values. In Figure 5.2 we present an example query, requesting for countries with similar populations where the proposed syntax can be further appreciated.

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 in SPARQL, with dedicated syntax and physical operators offering sub-quadratic performance.

5.4. Semantics

In Chapter 2, we addressed the semantics of the core operators of the SPARQL algebra. In this section we define the semantics of the introduced similarity join operator, its algebraic properties and its relation with other operators.

The similarity join expression parsed from the aforementioned syntax is defined as \( s := (\mathcal{V}, \delta, v, \phi) \), where \( \mathcal{V} \subseteq \mathcal{V} \times \mathcal{V} \) contains pairs of variables to be compared; \( \delta \) 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 \(\infty\)) for incomparable inputs; \( v \in \mathcal{V} \) is a fresh variable to which distances will be bound; and \( \phi \in \{rg_r, mn_k\} \) is a filter expression based on range or k-nn.
In the following, we present several definitions that build up the specification of our similarity join operators.

**Definition 5.4.1** Given two solution mappings \( \mu_1 \sim \mu_2 \), we define the set of mapping pairs to be compared in the similarity join as:

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

We note that: \([V]_{\mu_1}^{\mu_2} = [V]_{\mu_2}^{\mu_1}\).

**Definition 5.4.2** We define the result of a natural equi-join as \([\mu_1, \ldots, \mu_n] \in X_1 \bowtie \ldots \bowtie X_n\) 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 also define \(\mu = [\mu_1, \ldots, \mu_n]\) as the mapping \(\bigcup_{i=1}^{n} \mu_i\).

**Definition 5.4.3** Given a solution mapping \(\mu\) and a set of solution mappings \(X\), the solution mappings of \(X\) compatible with \(\mu\) is denoted by \(X_{\sim \mu}\) as:

\[
X_{\sim \mu} := \{\mu' \in X \mid \mu \sim \mu'\}.
\]

**Definition 5.4.4** Given a set of solution mappings \(X\) we define the union and intersection domains as:

\[
udom(X) := \bigcup_{\mu \in X} \text{dom}(\mu)
\]

\[
idom(X) := \bigcap_{\mu \in X} \text{dom}(\mu)
\]

**Definition 5.4.5** The expression \(v/d\) denotes a mapping \(\mu\) such that \(\text{dom}(\mu) = \{v\}\) and \(\mu(v) = d\).

**Definition 5.4.6** Given two sets of solution mappings \(X\) and \(Y\), we define the evaluation of range and \(k\)-nn similarity joins, respectively, as:

\[
X \bowtie Y := \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Y \bowtie \{v/d([\mu_1])\} \mid \mu_v(v) \leq r\}
\]

\[
X \bowtie Y := \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Y \bowtie \{v/d([\mu_1])\} \mid \mu_v(v) \leq \kappa_{\mu_1,Y}^{\delta,k}\}
\]

when \(v \not\in \text{udom}(X) \cup \text{udom}(Y)\) or error otherwise, where:

\[
\kappa_{\mu_1,Y}^{\delta,k} := \min\{\delta([\mu_1]) \mid \mu_2 \in Y_{\sim \mu_1} \land |\{\mu'_2 \in Y_{\sim \mu_1} \mid \delta([\mu'_2]) < \delta([\mu_2])\}| < k\}.
\]

An error is returned when \(v \not\in \text{udom}(X) \cup \text{udom}(Y)\), i.e. \(v\) is not fresh, to emulate a similar behaviour to \textbf{BIND} in SPARQL. Per the definition of \(\kappa_{\mu_1,Y}^{\delta,k}\), more than \(k\) results can be returned for \(\mu_1\) in the case of ties in distance, which keeps the semantics deterministic.
We define bag semantics for similarity joins in the natural way, where the multiplicity of \( \mu \in X \bowtie s Y \) is defined to be the product of the multiplicities of the solutions \( \mu_1 \in X \) and \( \mu_2 \in Y \) that produce it.

5.5. Algebraic Properties

We now state some algebraic properties of the similarity join operators, regarding commutativity, associativity, distributivity and how it relates to other SPARQL operators. We use \( \bowtie_r, \bowtie_n, \bowtie_s \in \{\bowtie_r, \bowtie_n\} \) to denote range, k-nn and similarity joins, respectively.

**Proposition 5.1** \( \bowtie_r \) is commutative and distributive over \( \cup \).

**Proof.** Assume \( r = (V, \delta, v, rg_r) \). For any pair of sets of mappings \( X \) and \( Y \):

\[
X \bowtie_r Y := \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Y \bowtie \{[v/\delta([V]_{\mu_1}^{\mu_2})] \mid \mu_v(v) \leq r\}\}
\]

\[
\equiv \{[\mu_2, \mu_1, \mu_v] \in Y \bowtie X \bowtie \{[v/\delta([V]_{\mu_1}^{\mu_2})] \mid \mu_v(v) \leq r\}\}
\]

\[
\equiv Y \bowtie_r X
\]

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

\[
X \bowtie_r (Y \cup Z) := \{[\mu_1, \mu_2, \mu_v] \in X \bowtie (Y \cup Z) \bowtie \{[v/\delta([V]_{\mu_1}^{\mu_2})] \mid \mu_v(v) \leq r\}\}
\]

\[
\equiv \{[\mu_1, \mu_2, \mu_v] \in X \bowtie Y \bowtie \{[v/\delta([V]_{\mu_1}^{\mu_2})] \mid \mu_v(v) \leq r\}\} \cup
\]

\[
\{[\mu_1, \mu_2, \mu_v] \in X \bowtie Z \bowtie \{[v/\delta([V]_{\mu_1}^{\mu_2})] \mid \mu_v(v) \leq r\}\}
\]

\[
\equiv (X \bowtie_r Y) \cup (X \bowtie_r Z)
\]

Commutativity and left-distributivity over \( \cup \) imply distributivity over \( \cup \). \( \square \)

**Proposition 5.2** \( \bowtie_n \) 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 \( |X| = n, |Y| = |Z| = 2n \), \( n \geq k \) such that:

- Commutativity: \( |X \bowtie_n Y| = nk \) and \( |Y \bowtie_n X| = 2nk \).

- Distributivity: \( |X \bowtie_n (Y \cup Z)| = nk \) and \( |(X \bowtie_n Y) \cup (X \bowtie_n Z)| = 2nk \). \( \square \)

**Proposition 5.3** \( \bowtie_n \) is right-distributive over \( \cup \).
Proof. Assume $n = (\mathcal{V}, \delta, v, n_k)$. We see that:

$$(X \cup Y) \Join_n Z := \{ [\mu_1, \mu_2, \mu_v] \in (X \cup Y) \Join Z \Join \{v/\delta([\mathcal{V}]^{\mu_v}_{\mu_2})\} \mid \mu_v(v) \leq \kappa^{\delta,k}_{\mu_1,Z} \}$$

$$\equiv \{ [\mu_1, \mu_2, \mu_v] \in X \Join Z \Join \{v/\delta([\mathcal{V}]^{\mu_v}_{\mu_2})\} \mid \mu_v(v) \leq \kappa^{\delta,k}_{\mu_1,Z} \} \cup$$

$$\{ [\mu_1, \mu_2, \mu_v] \in Y \Join Z \Join \{v/\delta([\mathcal{V}]^{\mu_v}_{\mu_2})\} \mid \mu_v(v) \leq \kappa^{\delta,k}_{\mu_1,Z} \}$$

$$\equiv (X \Join_n Z) \cup (Y \Join_n Z) \qed$$

**Proposition 5.4** Similarity joins are not associative, i.e., $(X \Join_s Y) \Join_{s'} Z \neq X \Join_s (Y \Join_{s'} Z)$ holds.

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

Finally, we discuss how the defined operators relate to other key SPARQL operators. The condition in claim 3 is analogous to well-designed queries [49].

**Proposition 5.5** Let $s = (\mathcal{V}, \delta, v, \phi)$. If each mapping in $X \Join Y$ binds all variables in $\mathcal{V}$ and $v \notin \text{udom}(X) \cup \text{udom}(Y) \cup \text{udom}(Z)$, then the following hold:

1. $(X \Join_s Y) \Join Z \equiv (X \Join Z) \Join_s Y$
2. $(X \Join_s Y) \setminus Z \equiv (X \setminus Z) \Join_s Y$ if $\text{udom}(Z) \cap (\text{udom}(Y) - \text{idom}(X)) = \emptyset$
3. $(X \Join_s Y) \wedge Z \equiv (X \wedge Z) \Join_s Y$ if $\text{udom}(Z) \cap (\text{udom}(Y) - \text{idom}(X)) = \emptyset$
4. $\sigma_f(X \Join_s Y) \equiv \sigma_f(X) \Join_s Y$ if $f$ is scoped to idom($X$).

Proof. We prove each claim in the following:

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

$$\begin{align*}
(X \Join_s Y) \Join Z := & \{ [\mu_1, \mu_2, \mu_v] \in X \Join Y \Join \{v/\delta([\mathcal{V}]^{\mu_v}_{\mu_2})\} \mid \phi(\mu_v) \} \Join Z \\
& \equiv \{ [\mu_1, \mu_1', \mu_2, \mu_v] \in X \Join Z \Join Y \Join \{v/\delta([\mathcal{V}]^{\mu_v}_{\mu_2})\} \mid \phi(\mu_v) \} \\
& \equiv \{ [\mu_1, \mu_2, \mu_v] \in (X \Join Z) \Join Y \Join \{v/\delta([\mathcal{V}]^{\mu_v}_{\mu_2})\} \mid \phi(\mu_v) \} \\
& \equiv (X \Join Z) \Join_s Y
\end{align*}$$

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 $\mu' \in Z$ 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 $\mu_2 = [\mu_2', \mu_2''] \in Y \Join \{v/\delta([\mathcal{V}]^{\mu_v}_{\mu_2})\}$ from $X \Join_s Y$, 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
  \n  SIMILARITY JOIN
  ON (\?lifex1 \?ngdp1 \?gdp1 \?growth1 \?hdi1)
  (\?lifex2 \?ngdp2 \?gdp2 \?growth2 \?hdi2)
  \n  TOP 1 USING '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
}

<table>
<thead>
<tr>
<th>?c1</th>
<th>?c2</th>
<th>?d</th>
</tr>
</thead>
<tbody>
<tr>
<td>wd:Q419</td>
<td>[Peru]</td>
<td>0.129</td>
</tr>
<tr>
<td>wd:Q298</td>
<td>[Chile]</td>
<td>0.134</td>
</tr>
<tr>
<td>wd:Q96</td>
<td>[Mexico]</td>
<td>0.195</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: Query for European countries most similar to Latin American countries in terms of a variety of economic indicators, with sample results

3. The second step here uses the previous two results. The third step uses the right-distributivity of $\bowtie_s$ and $\bowtie_g$ over $\cup$ proven in previous propositions.

$$(X \bowtie_g Y) \bowtie Z := ((X \bowtie_g Y) \bowtie Z) \cup ((X \bowtie_g Y) \setminus Z)$$

$$\equiv ((X \bowtie Z) \bowtie_s Y) \cup ((X \setminus Z) \bowtie_g Y)$$

$$\equiv ((X \bowtie Z) \cup (X \setminus Z)) \bowtie_g Y$$

$$\equiv (X \bowtie Z) \bowtie_g Y$$

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 $\mu_2 = [\mu_2', \mu_2''] \in Y \bowtie Y / \delta([\nu \mu_1])$ from $X \bowtie_s Y$, the result then holds per the given assumptions.

5.6. 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 [63].

**Similar Countries:** In Figure 5.3 we present a similarity query for Wikidata [63] that, for each Latin American country, will return the European country with the most similar...
Figure 5.4: Query for non-metal chemical elements similar to metallic elements in terms of atomic properties.

welfare indicators to it, considering life expectancy, Gross Domestic Product (GDP), nominal GDP, GDP growth rate and Human Development Index (HDI). 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 Chemicals** Chemical elements contain several numeric/ordinal properties, such as boiling and melting point, mass, electronegativity and so on. In Figure 5.4 we ask for metals similar to non-metals in terms of those properties. along with sample results of the similar metal/non-metal query.

**Similar Elections:** In Figure 5.5 we present a more complex similarity query over Wiki-data 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 use of aggregates and paths in the operand graph patterns of the similarity join. The figure also presents the results of the query.

\[^{1}\text{We use prefixes as defined in \url{http://prefix.cc}}\]
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
    }
}

SIMILARITY JOIN ON (?c1 ?p1 ?i1) (?c2 ?p2 ?i2)
TOP 4 USING '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
}
GROUP BY ?e2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>wd:Q15062956</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>0.000</td>
</tr>
<tr>
<td>wd:Q1348890</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>0.220</td>
</tr>
<tr>
<td>wd:Q1505420</td>
<td>10</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>0.240</td>
</tr>
<tr>
<td>wd:Q19818995</td>
<td>10</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td>0.293</td>
</tr>
</tbody>
</table>

Figure 5.5: Query for elections similar to the 2017 German Federal Election in terms of number of candidates, parties and ideologies participating, with results

5.7. Implementation

The implementation of the system extends ARQ – the SPARQL engine of Apache Jena – which indexes an RDF dataset and receives as input a (similarity) query in the syntax discussed in Section 5.3. 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 publicly available at https://github.com/scferrada/jenasj. The project can be imported as a dependency, and then queries can be executed through code. An example for a similarity join query execution in Java can be found in Listing 5.1.
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 Jena’s Parser through JavaCC\(^3\) wherein the new keywords and syntax rules are defined. The result of the Parsing stage is a tree of algebraic operators, e.g., `knnsimjoin(leftjoin(...), triplepattern(...)).` The ALGEBRA OPTIMISATION then applies static rewriting rules (discussed in Section \(^5.8\)) over the query, further turning logical operators (e.g., `knnsimjoin`) into physical 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.

### 5.7.1. Implementation Details

Jena is written in Java and provides several objects for all the different stages of query evaluation. In this section we discuss in depth about these objects, the execution flow and how we modify them to support similarity joins.

Upon receiving a query string, Jena creates a Query object based on a given syntax, through a Query Factory object. The Query object contains all the defined parameters, such as those of LIMIT and OFFSET, as well as an Element object that represents the WHERE clause. We added our syntax extension presented in Section \(^5.3\) and compiled a custom parser using

[https://javacc.github.io/javacc/](https://javacc.github.io/javacc/)
JavaCC. Parsers transform the string into a linked list of query elements, where we introduced the class `ElementSimJoin`. The parameters of the similarity join are directly introduced into the query object. In Figure 5.6, we present a Similarity Join query written in SPARQL, and the elements of the parsed query are represented as boxes: The green box is the root element that represents the entire `WHERE` clause; it is called an Element Group and it contains a collection of groups; next, within that collection are the two elements presented in yellow, an Element Path Block and an Element SimJoin for the BGP and the similarity join respectively; finally, the Element SimJoin presents a subelement which is another Element Path Block, depicted in blue. The latter structure follows suit from other binary operators, such as `MINUS` and `OPTIONAL` that encapsulate the second operator into an inner element.

After the Query object is created, it is compiled into a tree of algebraic operations (Op objects in Jena). The compile method goes through the Elements of the query and translates them into unary or binary operations, along with any other required parameters. For example, an `OPTIONAL` element is compiled to an `OpLeftJoin` operator with two operands. At this level, we included two operators (Op objects): `OpKNNSimJoin` and `OpRangeSimJoin` for nearest neighbours and range similarity joins respectively. In Figure 5.7, we display the tree of operations extracted from the query presented in Figure 5.6. In the figure, it can be seen that each triple pattern is parsed to a BGP, which are then joined before being fed to the similarity join operator and finally to the projection operator. The operator tree can be further optimised with respect to certain heuristics and rewriting rules in order to make its evaluation more efficient [16].

Once the algebraic operator tree is compiled, static optimisation can be performed. Using the Visitor Design Pattern, Jena can traverse the tree and transform it using query rewriting rules, such as executing filters before joins, delaying certain operations, etc. Our approach in this regard is further discussed in Section 5.8.
After the previous stages are completed, the algebraic tree is executed from the bottom-up. Each node in the tree produces a Query Iterator object which retrieves the solution mappings resulting from the evaluation of the operator over the dataset. Query Iterators can be consumed by Query Iterators of parent nodes and by the final user. Query Iterators operate on a fetch-as-required basis, so that the memory can be used more efficiently; they implement a typical iterator interface with methods `hasNext` and `next`.

We provide two Query Iterator objects, one for each kind of similarity join. The iterators make use of a SimJoin Solver object, for each kind of join and each available algorithm. The Solver objects prepare the data, and evaluate the similarity join in batches. The iterator requests a new batch of results when the current batch runs out of mappings. If the Solver retrieves an empty batch, that means that there are no further results, thus the `hasNext` method returns `false`. Since we do not propose a custom syntax for the users to select the desired similarity join algorithm, we tend to choose the solvers with the algorithms that perform best in the experimentation (see Section 5.9). An interesting research direction is to estimate which algorithm would perform best given a query and a dataset.

Before the similarity join computation, the operands need to be normalised. The normalisation process requires the complete evaluation of the operands, meaning that both Query Iterator objects need to be read. Usually, once an iterator is consumed, it cannot be rewinded to be consumed again. To manage this problem, we proposed a `BufferedQueryIteratorFactory` that, given a Query Iterator, produces copies of it (in the form of `BufferedQueryIterator` objects) on demand by keeping an in-memory buffer of the iterator results.

### 5.8. Query Planning and Optimisation

The Parsing phase will output either a `OpKNNSimJoin` or `OpRangeSimJoin` 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 $X \bowtie_s Y$, the naive strategy of computing a nested-loop join will require $|X| \cdot |Y|$ distance computations. Furthermore, algorithms for range queries may not efficiently support $k$-nn queries, and vice versa. Hence the algorithms we include with our implementation of
similarity joins are:

- **Range queries:**
  - Nested Loops: Checks every pair of objects and determines if they are similar. Uses $\Theta(n^2)$ distance computations (see Alg. 1).
  - Vp-Trees\(^4\) Builds an index with one of the sets and uses the elements of the other to query the index. Uses $\Theta(n^{2\alpha})$ distance computations on average to build the index and to compute the similarity join (see Alg. 2), where $0 \leq \alpha \leq 1$.
  - Quickjoin: Recursively divides the space in balls using pivots and checks similarity as it goes. Uses $O(n(1 + w)^{\lceil \log(n) \rceil})$ distance computations, which in the worst case is quadratic (see Alg. 3).

- **k-nn queries:**
  - Nested Loops: Checks every pair of objects and determines if they are similar, keeping a max heap for each element. Uses $\Theta(n^2)$ distance computations (see Alg. 1).
  - Vp-Trees: In this case, we use a different search algorithm that also requires $O(n^\alpha)$ distance computations per element.
  - FLANN\(^5\) Internally uses a random kd-forest that builds up to 20 randomised kd-trees \(^{43}\). This approach takes $O(\log(n))$ distance computations per query on average, and $O(n \log(n))$ distance computations to build the index.

Nested loops constitute a baseline for evaluating similarity joins without optimisation, as would be applied for similarity queries written in vanilla SPARQL syntax or in iSPARQL \(^{35}\). On the other hand, Quickjoin is used in DBSimJoin \(^{54}\), 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. It is important to remark that, since there is not an especial syntax for self-similarity joins, Root-Join is not used in this implementation.

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 5.3 for example, while the values for GDP tends to be in the order of millions, billions or trillions, the values for HDI fall between zero and one; 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 – hopefully more efficient – plan through query rew-

\(^4\)We use the library provided by Chambers at [https://github.com/jchambers/jvptree](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](https://gitlab.com/jadro-ai-public/flann-java-port.git).
riting 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 query evaluation times, yielding a negative result, as we now discuss.

Regarding rewriting rules, in fact there is little opportunity for such rules to optimise queries with similarity joins for two main reasons. The first reason is that since similarity joins are a relatively expensive operation (when compared with operators based on term equality, for example), optimally the results returned from the sub-operands should be as small as possible at the moment of evaluation; this often negates (and indeed reverses) the benefit of trying to delay certain operations until after the similarity join. The second reason is that when the similarity join is based on $k$-nn, and as discussed in Section 5.5, the operator is not commutative, nor associative, nor even monotonic (adding more data may cause some solutions to be dropped) and as a result, when evaluating a $k$-nn similarity join, in almost all cases, the operands must be evaluated as originally specified by the user. This behaviour becomes clear in Proposition 5.5, where we see that there is little room for impactful rewritings.

Since the operands of similarity joins often have significant overlap in terms of triple patterns (modulo variable naming), another avenue for optimisation lies in evaluating this overlap once, reusing the results across both sub-operands. As an example, for the query in Figure 5.3, both operands have the same graph pattern except the last triple pattern, which distinguishes European and Latin American countries. We implemented techniques to compute this overlap only once and reuse the results in both sub-operands, which we expected would reduce the runtime by avoiding redundant computation. However, experiments contradicted this expectation, which we attribute to two main reasons. First, the overlap may be (far) less selective than the non-overlapping patterns, making it disadvantageous to evaluate first; in fact, this tends to be the case as the non-overlapping patterns will often indicate distinguishing features of both sides of the similarity join. Conversely, if we conservatively identify the maximal overlap maintaining the original selectivity-based ordering for each operand, the overlap will often be negligible for the same reason. Second, even in favourable examples where a significant and selective overlap was present, no effect on performance was seen, due perhaps to lower-level caches; even if the query planner executes the overlap twice, for example, blocks of data from the disk will already have been cached.

In summary, our experience has been that the most efficient way to evaluate queries with similarity joins is “bottom-up”: optimising and evaluating the individual operands, optimising the similarity join itself, and then optimising the outer query. Other techniques that we explored – rewriting rules involving similarity joins, and caching overlapping patterns – were not advantageous according to our experiments, often giving the same or worse performance. We thus chose to not maintain these techniques in the final system.

5.9. Evaluation

We now present our evaluation, focusing on a performance-based comparison of different physical operators for similarity joins in the context of increasingly complex SPARQL queries,
Table 5.1: 10 most frequent characteristic sets of numeric properties in Wikidata

<table>
<thead>
<tr>
<th>Ordinal Characteristic Set</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>wdt:P644-(genomic start), wdt:P645-(genomic end)</td>
<td>624,494</td>
</tr>
<tr>
<td>wdt:P1539-(female population), wdt:P1540-(male population)</td>
<td>99,056</td>
</tr>
<tr>
<td>wdt:P1538-(households), wdt:P1540-(male population)</td>
<td>87,230</td>
</tr>
<tr>
<td>wdt:P1538-(households), wdt:P1539-(female population)</td>
<td>87,222</td>
</tr>
<tr>
<td>wdt:P1538-(households), wdt:P1539-(female pop.), wdt:P1540-(male pop.)</td>
<td>87,039</td>
</tr>
<tr>
<td>wdt:P1457-(absolute magnitude), wdt:P1096-(eccentricity)</td>
<td>26,333</td>
</tr>
<tr>
<td>wdt:P1096-(eccentricity), wdt:P4501-(albedo)</td>
<td>21,744</td>
</tr>
</tbody>
</table>

as well as a comparison with the baseline system: DBSimJoin. We conduct experiments with respect to two benchmarks: the first is a novel benchmark we propose for Wikidata, while the second is an existing benchmark used by DBSimJoin based on visual feature vectors of images.

5.9.1. 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. To arrive at these queries, we begin with some data exploration. Specifically, from the dump, we compute the ordinal/numeric characteristic sets 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 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. In Table 5.1 we display the most common such characteristic sets found.

The values of $k$ used in the experiment 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, over which we report average runtimes. Experiments were run on a Windows 10 machine with a 4-core Intel i7-7700 processor @2.80GHz and 16GB of RAM.

Figure 5.8 presents the average execution time for differing numbers of entities, defined to be the cardinality of the solutions $|M|$ input to the self-similarity join $M \bowtie s M$. Highlighting that the $y$-axis is presented in log scale, we initially remark that the value of $k$ appears to have a linear 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

---

6 We use a truthy dump available in February, 2020.
increases in time as the number of input entities increases. More specifically, FLANN is faster in 54.7% of the query scenarios; however, we remark that, unlike vp-Trees, FLANN computes an approximation of the real result where, in these experiments, it gave 98% precision overall compared with the ground truth extracted from the nested loop results.

In terms of absolute times, we find that both FLANN and vp-Trees can compute self-similarity joins $M \bowtie M$ where $|M| < 20000$ 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 ($|M| < 100000$) within 10 seconds for $k = 1$, 40 seconds for $k = 4$, and 80 seconds for $k = 8$.

In Figure 5.9, we present box-plots to explore the effect of differing numbers of properties (i.e., dimensions) on the performance of the similarity join algorithms; we highlight that the $y$-axis is again in log-scale. We note that there is a slightly increasing trend in query runtimes as the number of properties increases. Comparing systems again, we see much more efficient results for vp-trees and FLANN than for nested loops; comparing vp-trees and FLANN, they are difficult to distinguish: for 4 and 5 properties, vp-trees presents a slightly more favourable distribution than FLANN; however, from 6 properties and more, the distribution of time values of FLANN becomes narrower and has a slightly lower median. Regarding the latter observation, note, however, that the narrower distributions for queries with 10, 11 and 12 properties are also attributable to there being fewer queries with this many properties.

In summary, Figures 5.8 and 5.9 both show that the nested loop strategy is consistently worse than the other two approaches in terms of average execution time, being up to two orders of magnitude more expensive than its counterparts. When comparing FLANN with vp-trees, in terms of the 3,644 cases tested – for varying $k$, entity number and property number – we obtain that in a case-by-case pairwise comparison, FLANN performs best in 1,528 cases (42%), whilst vp-trees performs best in the remaining 2,116 cases (58%). Finally, given that FLANN is an approximation-based approach, we compute the average precision of FLANN over the query runs to be 98%, showing that it provides accurate approximations. However, we argue that the most useful strategy is to use vp-trees since they perform best in most scenarios, compute exact results and can solve both range and $k$-nn similarity joins.
5.9.2. Corel Colour Moments: Range Similarity Queries

We compare our system with the closest found in literature, DBSimJoin [54], a PostgreSQL extension that supports range-based similarity joins in metric spaces implementing the Quickjoin algorithm. As DBSimJoin only supports range queries, we compare it with the vp-Tree implementation in Jena (Jena-vp), which supports both range and $k$-nn queries (the Java port of FLANN only supports $k$-nn queries). 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 numeric values representing the mean, standard deviation and skewness for the Hue, Saturation and Lightness (HSV) of pixels in the image. For this dataset, the DBSimJoin paper only reports the effect of the number of Quickjoin pivots on the execution time when $r = 1\%$ of the maximum possible distance in the space [54]. Hence we obtained the DBSimJoin implementation and the CCM dataset in order to compare it with our system for more general performance metrics. We converted the CCM dataset to an RDF graph using a direct mapping: each record is an object with 9 properties valued as the respective vector dimension. We run the experiments on the same machine as per the previous evaluation.

Table 5.2 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 Quickjoin algorithm, which is a key variable in Quickjoin complexity. DBSimJoin actually crashes when $r \geq 0.4$, where we include additional results between 0.1 and 0.4 to illustrate the exponential increase in time and subsequent crash. Execution time in Jena-vp increases slowly as $r$ increases, where more tree branches need to be visited as the result-set increases up to three hundred million results. Our implementation does not crash with massive results because it fetches the bindings lazily: it obtains all pairs within distance $r$ 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,
Table 5.2: Execution times in seconds for range similarity joins over the CCM Dataset.

<table>
<thead>
<tr>
<th>Search radius</th>
<th>Results</th>
<th>DBSimJoin (s)</th>
<th>Jena-vp (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>68,462</td>
<td>47.22</td>
<td>6.92</td>
</tr>
<tr>
<td>0.10</td>
<td>68,498</td>
<td>84.00</td>
<td>7.45</td>
</tr>
<tr>
<td>0.30</td>
<td>72,920</td>
<td>775.96</td>
<td>9.63</td>
</tr>
<tr>
<td>0.39</td>
<td>92,444</td>
<td>1,341.86</td>
<td>11.17</td>
</tr>
<tr>
<td>0.40</td>
<td>121,826</td>
<td>--</td>
<td>11.30</td>
</tr>
<tr>
<td>1.00</td>
<td>4,233,806</td>
<td>--</td>
<td>35.01</td>
</tr>
<tr>
<td>5.90</td>
<td>395,543,225</td>
<td>--</td>
<td>1,867.86</td>
</tr>
</tbody>
</table>

Figure 5.10: Trend curve predicting execution times for DBSimJoin for $r \geq 0.4$.

Quickjoin starts piling up window partitions that replicate the data at a high rate, thus causing it to crash as distance increases. We compute a trend line for the remaining $r$ values for DBSimJoin to better estimate how it compares to our system. In Figure 5.10, we present the linear trend predicted for DBSimJoin, where it can be seen that for $r = 5.9$ DBSimJoin would take up to 15 thousand seconds, whereas vp-jena takes less than 2 thousand seconds. The R-squared value of the trend line is 0.939.

Even when considering range-based similarity joins, our system outperforms the PostgreSQL-based DBSimJoin implementation in terms of efficiency (our system is faster) and scalability (our system can handle massive amounts of results).
Chapter 6

Applications

In this chapter we present two main applications and derivative works for similarity-based queries on the Web. Firstly, we discuss IMGpedia and how it has been extended from its original version [22] and how it benefits from the novel similarity join operator for SPARQL. Secondly, we present the query modifier CLUSTER BY for SPARQL Queries for dynamic clustering of query results. We first show the syntax and semantics of the modifier, as introduced by Ławrynowicz [69], and discuss how it can be adapted for SPARQL 1.1. Later, we present our implementation of the modifier in Apache Jena, and how the definitions are interpreted in practice. Finally, we showcase use-case queries that make use of dynamic clustering over their results.

6.1. IMGpedia

IMGpedia [22] is a linked dataset that provides information about 15 million images from the Wikimedia Commons dataset. The dataset of IMGpedia includes different visual descriptors that capture various features from the images, such as color distribution, shape patterns and grayscale intensities. It also provides static similarity relations among the images and links to related entities on DBpedia [3] and (now) to Wikidata [63]. With IMGpedia it is then possible to answer visuo-semantic queries through a public SPARQL endpoint. These queries combine the similarity relations with the semantic facts that can be extracted from the linked sources; an example of such a query might be “retrieve images of museums that look similar to European cathedrals.”

In our previous work [22] we described the creation of the dataset of IMGpedia and illustrated some preliminary queries that it can respond to using the links provided to DBpedia. In this paper we report on some new visuo-semantic queries that are enabled by newly added links to Wikidata, which provides a more flexible way to request for external entities. We also present a new user interface for IMGpedia that allows people to browse the results of the SPARQL queries featuring the images involved, and to explore details about the images, such as their related web resources and their similar images. Finally we present some future directions in which we foresee the IMGpedia project developing and possible ways in which it might contribute to existing Wikimedia projects.
Before we present novel aspects of IMGpedia – the user interface and the new queries enabled by links to Wikidata – we first provide an overview of IMGpedia, the images from which it is built, the visual descriptors used, the types of relations considered, and so forth. Here our goal is to provide an overview of the knowledge-base; for more details we refer the reader to our previous paper [22].

IMGpedia contains information about 14.7 million images taken from the Wikimedia Commons multimedia dataset. We only consider the images with extensions JPG and PNG (92% of the dataset) in order to perform a standardised analysis of them. We store the images locally and characterise them by extracting their visual descriptors, which are high-dimensional vectors that capture different features of the images. The descriptors used were the Histogram of Oriented Gradients (HOG), the Gray Histogram Descriptor (GHD), and the Color Layout Descriptor (CLD). These descriptors extract features related to the borders of the image, to the brightness of the grayscale image, and to the distribution of the color in the image, respectively. Using these sets of vectors we computed static similarity relations among the images: for each image and descriptor, we find its 10 nearest neighbours and we store these relations along with the computed distances and the type of descriptor being used (to scale to the full image-set, we use an approximate nearest neighbour algorithm). Finally, we take all this information and represent it as an RDF Graph, upload it to a Virtuoso Server and provide a public SPARQL endpoint for clients to issue queries. In Table 6.1 we present some statistics to give an idea of the size of the dataset and to show the main entities and properties provided by IMGpedia.

6.1.1. Browsing the Data

The initial release of IMGpedia used the default interfaces from Virtuoso Server[^1] for writing queries, browsing the results and exploring the resources. Though these interfaces provided the necessary mechanisms for agents to access the resources – a SPARQL endpoint with Linked Data dereferencing – the interfaces for human users were mainly plain HTML tables for displaying SPARQL query results and almost illegible HTML documents for displaying the details of individual resources. We foresaw that the lack of more human-friendly interfaces would prevent people from using and querying the dataset; conversely, being a knowledge-base centered around images, we foresaw much potential for creating a visually-appealing querying and browsing interface over IMGpedia.

Along these lines, to initially address the usability of IMGpedia, we set up a front-end

[^1]: http://imgpedia.dcc.uchile.cl/sparql
The application that makes the process of querying and browsing our data a more friendly experience. The application was developed with the AngularJS 5 framework and uses the Virtuoso Server as a back-end. The application consists of three main components: a SPARQL query editor, a browser for the results of SPARQL queries that shows the images in-place, and an interface for exploring the details of individual visual entities.

The SPARQL query editor provides a text area for writing the queries and a button for executing them. The interface then displays the results of the SPARQL queries below the query editor; if the result contains any visual entities, rather than simply display the URL of the image, it automatically displays the corresponding image by making a request to Wikimedia Commons. In Figure 6.1a we present a screenshot of the interface showing a simple query for three pairs of similar images within a particular visual similarity distance, with the results displayed below.

A user may click on an image displayed in such a result, which will lead them to a detailed focus view of the information available for that visual entity. This view shows the focus image that is being described, along with its name in Wikimedia, and links to related Wikipedia, Wikidata and DBpedia resources (based on links in those knowledge-bases, appearances in the Wikipedia article associated with those entities, etc.). We also display images similar to the focus image based on precomputed similarity relations present in the IMGpedia knowledge-base; similar images are grouped by visual descriptor and are sorted by distance; the user can hover over each such image to see the distance or can click to go to its focus page. In

---

2The source code of the interface application can be found at https://gitlab.com/nbravo/imgpediavis.
3http://imgpedia.dcc.uchile.cl/query
SELECT ?imgu ?name ?cname WHERE {
    SERVICE <https://query.wikidata.org/bigdata/namespace/wdq/sparql>{
        ?palace wdt:P31 wd:Q16831714 ; # type government palace
            wdt:P17 ?country .
        ?country wdt:P361 wd:Q12585 . # countries in Latin America
        OPTIONAL{
            ?country rdfs:label ?cname
            FILTER(LANG(?name)='en' && LANG(?cname)='en')
        }
    }?

    ?img imo:associatedWith ?palace ;
        imo:fileURL ?imgu .
}

Figure 6.2: Government palaces in Latin America (sample)

Figure 6.1b a screenshot of the interface can be found showing a drawing of Queen Mary I of England by Lucas Horenbout; below this the interface displays similar images found through the gray intensity descriptor in ascending order of distance.

6.1.2. Querying the Data

In the first version of IMGpedia, the supported visuo-semantic queries relied heavily on the existence of categories for the resources of DBpedia; for the example query in Chapter 3, we use the DBpedia category `dbc:Roman_Catholic_cathedrals_in_Europe` for obtaining European Cathedrals (see [22, Figure 3]). To address this issue and in order to support more diverse queries, we have added links to other complementary context sources; in particular, IMGpedia now provides 6,614,244 links to Wikidata, where a visual entity in IMGpedia is linked to an entity from Wikidata if the image appears on the English Wikipedia article corresponding to that Wikidata entity. The match between Wikipedia articles and Wikidata entities was made by querying a dump of Wikidata.4

Using these new links, we can ask novel queries that were not possible before. For example, now we can request images of governmental palaces in Latin America. This was not possible before, since there is no category referring to such palaces in DBpedia; if we now leverage

4The Wikidata dump used was that from 2017-07-25
these new links to Wikidata, we can combine different predicates in order to achieve our goal. In Figure 6.2 we show the SPARQL query that satisfies our requirements; it first requests from Wikidata all entities referring to governmental palaces in Latin America through SPARQL federation, further retrieving the URLs of related images, along with the label of the palace and the name of the country. Figure 6.2 then shows a sample of the results returned.

The previous query is what we would refer to as a purely “semantic query”, meaning that it does not rely on any of the visual information present in IMGpedia computed from the content of the images. On the other hand, with links to Wikidata, we can perform new visuo-semantic queries combining Wikidata facts and IMGpedia visual similarity; for example, we can ask for educational institutions that are similar to the images of government palaces obtained before. In Figure 6.3 we show the SPARQL query required to do so. First we use the same service clause requesting for the palaces as in the query of Figure 6.2 so we omit its body here. Later we obtain the similar images and the entities they are associated with and finally we keep those images that are related with educational institutions using SPARQL property paths to consider any subclasses. In Figure 6.3 we show the result of the query.

In Figure 6.4 we show another example of a visuo-semantic query where we look for people with images similar to those associated with paintings in the Louvre. Below the query in Figure 6.4 we show its result, where each painting has two similar people. It is worth noting that the first painting is not on display at the Louvre; however the image is related to the Louvre since it is a self-portrait of Jean Auguste Ingres, the painter of Napoleon I on His Imperial Throne which is on display at the Louvre, where the portrait of Ingres then appears in the Wikipedia article of the painting.

The SPARQL query of Figure 6.4 can be modified to obtain other paintings similar to those exhibited at the Louvre by changing the type requested on the second SERVICE clause from wd:Q5 (human) to, for example, wd:Q3305213 (painting). An interesting result of this query variation can be seen in Figure 6.5, where the two images depict the same painting. In such cases we can say that the images are near_copy of each other. Such relations among images could allow reasoning over the entities related to the images; in this case one image corresponds to the painting entity and the other is associated to the painter, and hence it is probable that the painter is the author of the artwork.

Finally, with the SPARQL extension that includes similarity joins, queries do not need to rely on pre-existing similarity relations among images since those can be computed at query time. An example of such a query can be found in Figure 6.6.

The migration of the back-end of the system, from Virtuoso to SJ-Jena, has been left for future work. However, we loaded the IMGpedia Dataset [22] into SJ-Jena to test how a similarity join query over the visual descriptors would work. Figure 6.6 presents a similarity query over IMGpedia. 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 the Histogram of Oriented Gradients feature descriptor.
SELECT ?imgurl ?imgurl2 ?name WHERE{
  SERVICE <https://query.wikidata.org/bigdata/namespace/wdq/sparql>{
    ... # obtain government palaces as in previous query
  }
  ?img imo:associatedWith ?palace ;
      imo:similar ?img2 ;
      imo:fileURL ?imgurl .
  ?img2 imo:associatedWith ?wiki;
  FILTER(CONTAINS(STR(?wiki), 'wikidata.org'))
  SERVICE <https://query.wikidata.org/bigdata/namespace/wdq/sparql>{
    ?wiki wdt:P31/wdt:P279* wd:Q2385804; # subclass of edu. institute
        rdfs:label ?label .
    FILTER(LANG(?label)='en') }
}

Figure 6.3: Educational institutions similar to Latin American government palaces
SELECT ?painting ?people WHERE {
  SERVICE <https://query.wikidata.org/bigdata/namespace/wdq/sparql>{
    ?paintingw wdt:P31 wd:Q3305213 ;
    wdt:P276 wd:Q19675 ;
    rdfs:label ?label .
    FILTER(LANG(?label)='en')
  }
  ?painting imo:associatedWith ?paintingw ;
  imo:similar ?img2 ;
  imo:fileURL ?imgu .
  ?people imo:associatedWith ?peoplew ;
  imo:fileURL ?img2u .
  FILTER(CONTAINS(STR(?wiki), 'wikidata.org'))
  SERVICE <https://query.wikidata.org/bigdata/namespace/wdq/sparql>{
    ?peoplew wdt:P31 wd:Q5 . }
}

<table>
<thead>
<tr>
<th>?painting</th>
<th>?people</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jean Auguste Ingres self-portrait</td>
<td>Michael Willmann</td>
</tr>
<tr>
<td>The Fifer by Manet</td>
<td>Grigoriy Myasoyedov</td>
</tr>
<tr>
<td>Michel Alcan</td>
<td>Jay Bothroyd</td>
</tr>
</tbody>
</table>

Figure 6.4: People Similar to Paintings in the Louvre
Figure 6.5: Paintings similar to other paintings at the Louvre. Image on the right appears in the Olympia artwork article while the image on the left appears in the article of its painter, Édouard Manet.

```
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 USING 'manhattan' AS ?d # 3-nn using Manhattan
  { ?cathedral wdt:P31 wd:Q2977 . # Cathedrals
    ?vector2 a imo:HOG ; imo:describes ?img2 ; imo:value ?hog2 . # HOG descriptor
  }
}
```

Figure 6.6: Query for the three images of cathedrals most similar to each image of the US Capitol Building in terms of the HOG visual descriptor
6.2. Dynamic Clustering of SPARQL Query Results

In this section we analyse, develop and evaluate a CLUSTER BY solution modifier for SPARQL queries initially proposed by Ławrynowicz [69]. First, we discuss how it was initially conceived, how it can be formalised syntactically and semantically and how it interacts with the rest of the SPARQL modifiers. Later, we propose an implementation of the solution modifier built on top of Apache Jena. Finally, use-case queries are presented.

As we have argued before in this thesis, users sometimes require non-exact matches over the data and instead they wish for similar matches. This is also the case when using the solution modifier 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 [30] the algebra of GROUP BY is defined as follows:

**Definition 6.2.1** A group evaluates a list of expressions against a solution sequence, producing a set of partial functions from keys to solution sequences, as follows:

\[
\text{Group}(\text{expList}, X) := \{ \text{ListEval}(\text{expList}, \mu) \mid \mu' \in X, \text{ListEval}(\text{expList}, \mu) = \text{ListEval}(\text{expList}, \mu') \} \mid \mu \in X
\]

where \( X \) is a multiset of mappings, \( \rightarrow \) represents a partial function and \( \text{ListEval}(\text{expList}, \mu) \) returns a list \([e_1, \ldots, e_n]\) where \( e_i = \text{expr}_i(\mu) \) or error, \( \text{expr}_i \in \text{expList} \).

However, this definition can become cumbersome when trying to extend it, as in our case. We make use of the BIND operator that takes an expression and binds it to a variable to omit the expression list given to the group by and thus alleviate the complexity of the notation.

**Definition 6.2.2** Given a multiset of mappings \( X \) and a set of variables bound in the query \( V \subseteq V \), the Group operator is defined as:

\[
\text{Group}(X, V) := \{ (\mu', X') \mid \mu' \in \pi_V(X) \land X' = \{ \mu \in X \mid \pi_V(\mu) = \mu' \} \}
\]

where \( \pi \) is the projection defined in Chapter 2.

As presented, the GROUP BY modifier partitions the bindings into equivalence classes (the \( X' \) multisets) based on the different key values (the values of \( \mu' \)). The similarity-based alternative to GROUP BY is to perform clustering over the bindings, so result mappings that are similar are assigned to the same cluster. We see that the methods and decisions made in Chapter 5 can be applied in the development of a cluster-based query modifier.
6.2.1. Syntax and Semantics

In order to support clustering in SPARQL it is necessary to define its syntax and semantics in a way that a user can express the cluster variables, algorithms and their required parameters. Ławrynowicz [69] proposes the following extension for SPARQL 1.0 [51], where the cluster identifier is bound AS a fresh variable:

\[
\text{SolutionModifier ::= OrderClause? LimitOffsetClauses? ClusterByClause?}
\]
\[
\text{ClusterByClause ::= 'CLUSTER BY' Var+ 'AS' Cluster UsingClause?}
\]
\[
\text{UsingClause ::= 'USING' Method ('('Params')')?}
\]
\[
\text{Cluster ::= VAR1}
\]
\[
\text{Method ::= IRIRef}
\]
\[
\text{Params ::= TriplesBlock}
\]

Figure 6.7: Syntactic extension to the SPARQL 1.0 grammar for clustering [69]

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 \text{GROUP BY} modifier. The definition of a clustering modifier for SPARQL 1.1 requires to take into account the existence of \text{GROUP BY}, as it has to be decided how it interacts with \text{CLUSTER BY}.

Since clustering is the similarity-based version of grouping (which is based on exact matching), we could propose that the modifiers \text{CLUSTER BY} and \text{GROUP BY} should be mutually exclusive in a query. We lean towards the compatibility of the two modifiers, thus defining the clustering modifier to bind the cluster identifier of each mapping to a fresh variable and possibly allowing users to later group the results by the identifier value. We make this decision because any new operation included into a language should be as non-disruptive as possible, and to negate the usage of \text{GROUP BY} when clustering is indeed disruptive. We keep the syntax for indicating the clustering algorithm to be used as well as further parameters it might require, such as the number of clusters.

Since the \text{CLUSTER BY} modifier is not mutually exclusive with \text{GROUP BY}, \text{CLUSTER BY} does not interact with the \text{HAVING} clause and the different aggregators, which are computed after the clustering takes place. This effectively means that clustering cannot be applied with respect to values computed through aggregation functions, unless the aggregation is computed in a subquery.

The clustering modifier called without further parameters makes use of the default technique implemented in the engine. Ławrynowycz makes use of the SPARQL Query Results Clustering Ontology (SQRCO) to define the methods and parameters of the clustering algorithm to be applied. However, at the time of writing, the ontology is not found on the Web and was not registered in [prefix.cc](https://fno.io/spec/). Therefore, we make use of the Function Ontology [5] to introduce the different algorithms and parameters required. We defined the Similarity Namespace (\text{sim:}) that contains the specification of the clustering algorithms along with their parameters. The complete RDF can be found in Figure 6.9. Users must specify the algorithm to be used and their parameters in the query.
Therefore, the proposed extension for SPARQL 1.1 is:

\[
\text{SolutionModifier} ::= \text{OrderClause}? \quad \text{LimitOffsetClauses}? \quad \text{ClusterByClause}? \\
\quad \text{GroupClause}? \quad \text{HavingClause}?
\]

\[
\text{ClusterByClause} ::= \text{‘CLUSTER BY’} \text{Var+ ‘AS’ Var} \text{kClause WithClause}?
\]

\[
\text{WithClause} ::= \text{‘WITH’} \text{‘{‘Params’}’)?}
\]

\[
\text{Params} ::= \text{TriplesBlock}
\]

Figure 6.8: Syntactic extension to the SPARQL 1.1 grammar for clustering

An example of a SPARQL query written using the extension of Figure 6.8 can be appreciated in Figure 6.10 where we cluster the countries of Wikidata based on their life expectancy and economic growth. The clustering should be solved using \(k\)-means returning 4 clusters.

Given the SPARQL semantics of Chapter 2, the discussed syntax, and Definition 6.2.2 for \textsc{GROUP BY}, we build the semantics for the \textsc{CLUSTER BY} solution modifier.

We define the \textsc{CLUSTER BY} modifier as follows:

\textbf{Definition 6.2.3} Given a multiset of mappings \(X\), the clustering algorithm \(C\), the variables with respect to which the clustering is computed \(V \subseteq V\) and the fresh variable \(c \in V\) that stores the cluster identifier, the clustering modifier works as follows:

\[
\text{Cluster}(X, V, C, c) := \{C(\pi_V(X), \pi_V(\mu), c) \mid \mu \in X\}
\]

where \(C\) is a black box clustering function that, given a multiset of solution mappings \(X'\), a mapping \(\mu'\) and a variable \(c\) returns a mapping \(\mu' \cup c/i\) where \(i\) is the identifier of the cluster selected for \(\mu\).

\textbf{6.2.2. Implementation}

We implemented the solution modifier on top of Apache Jena, using a similar process as the one described in Chapter 5. We extended the SPARQL 1.1 grammar to include the rules of Figure 6.8. As a default configuration, we selected the \(k\)-medoids clustering algorithm with 4 clusters. We also provide implementations for \(k\)-means and DBSCAN.

All the algorithms consume the Query Iterators generated by the rest of the query and for each mapping, they add the cluster identifier bound to the fresh variable specified in the query. The algorithms produce a Query Iterator object with the computed clusters added to the mapping, using the fresh variable given in the query.

Since \textsc{CLUSTER BY} is a query modifier, it does not require defining all the objects that the similarity join operator has to provide, such as the Op and Query objects, but instead it is needed to store the cluster variables and parameters in the Query object.

The implementation of the modifier in Apache Jena is publicly available in the following repository: \url{https://github.com/scferrada/sparql-cluster}.

68
Figure 6.9: Clustering algorithms defined in the sim namespace using the Function Ontology

```
sim:kmeans
  a fno:Function ;
  dcterms:description "The k-means clustering algorithm" ;
  fno:expects sim:intParameterK .

sim:kmedoids
  a fno:Function ;
  dcterms:description "The k-medoids clustering algorithm" ;
  fno:expects sim:intParameterK .

sim:dbscan
  a fno:Function ;
  dcterms:description "The DBSCAN clustering algorithm" ;
  fno:expects (sim:doubleParameterEpsilon sim:intParameterMinPoints)

sim:intParameterK
  a fno:Parameter ;
  fno:predicate sim:numberOfClusters ;
  fno:type xsd:integer ; fno:required "true"^^xsd:boolean .

sim:doubleParameterEpsilon
  a fno:Parameter ;
  fno:predicate sim:minDistance ;
  fno:type xsd:double ; fno:required "true"^^xsd:boolean .

sim:intParameterMinPoints
  a fno:Parameter ;
  fno:predicate sim:minPoints ;
  fno:type xsd:integer ; fno:required "true"^^xsd:boolean .
```

Figure 6.10: Example SPARQL query with the CLUSTER BY solution modifier

```
SELECT ?lifex ?growth ?c WHERE {
  ?c1 wdt:P31 wd:Q6256 ; #countries
    wdt:P2250 ?lifex ;
    wdt:P2219 ?growth }
CLUSTER BY ?lifex ?growth AS ?c
WITH {
  fno:executes sim:kmeans ;
  sim:numberOfClusters "4"^^xsd:integer .
}
```

Figure 6.10: Example SPARQL query with the CLUSTER BY solution modifier
6.2.3. Use-Cases

First we graphically show in Figure 6.11 the result of the query presented in Figure 6.10 that clusters the countries in Wikidata with respect to their life expectancy and economic growth. Each point has a different colour depending on their assigned cluster.

Next, we take the Iris Dataset in RDF⁶ and use it to run a clustering query on petal length and width, and then to compare the results of the clustering with the three classes of Iris. In Figure 6.12 we present the extended SPARQL query that divides the objects into three clusters using a k-medoids algorithm. In Figures 6.13a and 6.13b we present the results of the clustering, and the data divided in classes respectively, where it can be seen that only five objects were clustered with the wrong class.

  ?iris a ?type;
} CLUSTER BY ?plength ?pwidth WITH {
  fno:executes sim:kmedoids ;
  sim:numberOfClusters "3"^^xsd:integer .
}

Figure 6.12: Extended SPARQL query to obtain three clusters of iris
Chapter 7

Conclusions and Future Directions

In this Chapter we present the main conclusions of the work here presented. We discuss and assess the hypothesis, objectives and research questions presented in Chapter 1. The chapter ends outlining future research lines that can continue this work and/or can be derived from it.

7.1. Conclusions

In Chapter 4 we presented an heuristic to solve the approximated $k$-nn self-similarity join in metric spaces. We proved it requires $\Theta(n^{3/2})$ distance computations. We provided an open implementation and tested it using three real world datasets, where we found that it can reach up to 46% precision. We present performance metrics, namely the execution time and the number of computed distances, and we discuss how the size and dimension of the datasets affects the performance of the algorithm. We discuss how the algorithm can be compared to a modified version of Quickjoin, that supports $k$-nn self-similarity joins, and to Locality Sensitive Hashing; we provide evidence where our algorithm outperforms them in terms of execution time and average precision. The algorithm is also tested using a different centre selection technique, which was found to worsen its efficacy. The root-join algorithm can prove useful for cases where building and storing an index is purposeless and/or there are time restrictions to get a fast answer. An example of such a case is multimedia similarity search on the Web, where queries tend to be unique and datasets tend to be very large.

In Chapter 5, 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 motivated and proposed an extension of the SPARQL query language that supports 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 novel syntax and semantics for multidimensional similarity joins in SPARQL, as well as an implementation based on Apache Jena that uses a selection of optimised physical operators for such joins. We also present use-case queries to illustrate the expressiveness and usefulness of the proposed extension.
We evaluate the implemented system using 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 (though FLANN did outperform it in some queries using approximations). 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.

We finally presented two applications of the study of similarity in the context of the Web of Data. We first present an extension for IMGpedia that allows users to easily browse through the images of Wikimedia and their similarity relations, as well as to the Wikipedia articles that relate to them; the extension also includes image results within the result tables of SPARQL queries, so they are more readily accessible by the users. Secondly, we introduce another extension to SPARQL, in this case a `{CLUSTER BY}` query solution modifier that instead of grouping result mappings by a given key, brings together similar keys into the same cluster; we present novel syntax and semantics compatible with SPARQL 1.1, as well as use case queries that showcase its usefulness. The clustering modifier can be used alongside other modifiers such as `{GROUP BY}`, `{ORDER BY}`, etc., since it binds a cluster identifier to a fresh variable for each solution mapping of the query. The extension provides three clustering algorithms: $k$-means, $k$-medoids and DBSCAN, where the required parameters can be given as part of the query using the Function Ontology.

7.2. Assessment of the Objectives of the Thesis

The hypothesis of this thesis is to verify that “supporting similarity-based queries within the Semantic Web will allow for the expression of novel queries that enable a new form of web retrieval that can be satisfied in a scalable, efficient, usable manner.” Given the work described in this thesis we are able to validate the hypothesis: Jena-SJ allows users to perform similarity queries over RDF datasets by using an extension of SPARQL, which is in fact a novel form of web information retrieval. Furthermore, we prove that Jena-SJ offers scalability, since when compared to the state of the art (DBSimJoin) it is able to handle hundreds of millions of results successfully. Jena-SJ is also proven to be efficient, by avoiding brute-force algorithms in favour of sub-quadratic strategies; moreover we propose a novel approximated algorithm to solve $k$-nn similarity joins. As for the usability of the system, we were able to validate the proposed syntax with the community and receive feedback that was incorporated into the proposal hereby presented; however, in the future, we hope to conduct more validation measuring the usability of the proposal through a more in-depth user study.

In the following, we review the objectives of the work and assess the level of accomplishment achieved for each item after the completion of this thesis.
Research and develop techniques to efficiently obtain similarity relations from large datasets. A comprehensive review of the many techniques to solve similarity queries is presented in Chapter 3, including both range and nearest neighbours, both sequential and parallel, and both specific domains and general metric spaces. In Chapter 4 we introduced root-join, an algorithm for approximate $k$ nearest neighbours similarity joins that computes only $\Theta(n^{\frac{3}{2}})$ distances.

Develop querying algorithms or indexes for SPARQL related to similarity-based queries, potentially using dimensions independently. We propose a complete extension to SPARQL, defining syntax and semantics to perform range and $k$nn similarity joins over solution mappings using any combination of dimensions, including computed dimensions through variable binding or aggregation functions. A custom index for similarity-based queries for SPARQL was discarded since it was not possible to maintain $O(2^d)$ indices, where $d$ is the total number of ordinal attributes in the data, and to preemptively plan for computed dimensions. Cache-based solutions and/or priority queues of indices fell out of the scope of the thesis and is proposed as future work.

Research and develop algorithms for query plan and optimisation. We present algebraic properties of the novel similarity join operator for SPARQL and prove that classic join query rewritings are not feasible with similarity joins. We also show that using a cache for overlapping triple patterns does not improve the performance of the queries. We thus leave as future work to test and find other ways to further improve the performance of the similarity join.

Propose, develop and evaluate semantic and similarity-based retrieval systems. We implement the proposed extension to SPARQL by extending Apache Jena, its grammar, parser and execution processes. We present several use-case queries along with the results provided by the system. We compare our implementation with the work deemed the closest: DBSimJoin, and we found that our system is faster and more scalable.

Apply the proposed techniques and systems to current open problems. We present two applications of the work presented. Firstly, we present an update to IMGpedia that allows users to query and browse the images more easily. Second, we design and develop another extension to SPARQL that supports a CLUSTER BY query modifier that makes use of the Function Ontology to take in the different clustering algorithms and their parameters.

7.3. Future Work

The work presented in this thesis involves a novel query paradigm that has been proposed to change the way that databases on the Web usually interact with structured data, recognising that similarity is a key feature when searching content on the Web and as such cannot continue to be omitted from modern database systems. As the work was being concluded, we identified several points of future research and improvements that we enumerate in this section.
Root-join can be further improved by using a centre selection algorithm that allows the formation of groups with less overlap. We could also allow objects to be associated to multiple groups at a time, under a set of conditions that respect the complexity of the algorithm. The definition of a better \( k \)-nn self-similarity join Quickjoin-based algorithm can be helpful to have a more fair comparison to our algorithm or even to further improve the state-of-the-art in terms of precision and execution time. Finally, the algorithm can be further generalised in order to handle similarity joins between different sets of objects.

In terms of the SPARQL extension, we propose several research branches and possible improvements. The first is to test how users can define their own distance functions in order to customise their queries whenever the provided \( L_p \) distances are not suitable; a first step in this direction is to explore how it is implemented in iSPARQL \cite{35}. Next, an important improvement of the extension and in general of how similarity joins are evaluated is to further explore planning and optimisation techniques; in this work we mostly ruled out the possibility of performing query rewrites due to the algebraic properties of the nearest neighbours similarity joins; however, it may still be possible that rewrites are a viable option for range similarity joins. As for optimisations, there are many options to explore; we unsuccessfully tried to cache the results of common triple patterns found on both similarity join operators, however there might be a possibility to change the way the operator is evaluated when there is a significant overlap of the left- and right-hand operators so that it is evaluated only once and their result is immediately fed into the similarity join by keeping deltas or filtering on-the-fly.

The main future contribution to be derived from this work is to make use of the novel similarity join operator (and the clustering query modifier) to help to solve other data-related problems, such as ontology alignment, data integration, entity linking among others. We believe that both the implemented system and the theoretical definitions can be used in those areas and effectively solve some of their problems.

Another idea that arose when conducting this work is to maintain certain indices, such as vp-trees, to improve the performance of the queries. We abandoned this branch since a naive solution would be to make indices for every combination of ordinal dimensions present in the data which would be impossible given the time and resources available, and also because such an index collection would not include “artificially” generated dimensions such as expressions and aggregated functions. Another problem with the static index approach is that the data involved in the query is not always the same, so the question of what data should be stored and how to make use of the index when some of its content is not useful remains open.

There are still open challenges regarding the management of multivalued properties and missing data in the similarity join. We currently dismiss any mappings with missing data, and consider \textsc{ANY} semantics for multiple values, but there are other options and techniques left unexplored that can be further pursued.

As a technical improvement, the IMGpedia online version can be migrated to our Apache Jena system that supports similarity joins so that users can request dynamically generated similarity relationships among images that are not currently found in the data directly from the IMGpedia webpage.
Bibliography


[51] Eric Prud’hommeaux. SPARQL query language for RDF. W3C Recommendation,


[65] Bin Yao, Feifei Li, and Piyush Kumar. K nearest neighbor queries and kNN-joins in


