Pontifícia Universidade Católica
do Rio de Janeiro

Yenier Torres Izquierdo

Contributions to the Problem of Keyword Search over Datasets and Semantic Trajectories Based on the Resource Description Framework

Tese de Doutorado

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática.

Advisor: Prof. Marco Antonio Casanova

Rio de Janeiro
January 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Yenier Torres Izquierdo

Contributions to the Problem of Keyword Search over Datasets and Semantic Trajectories Based on the Resource Description Framework

Thesis presented to the Programa de Pós–Graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Informática. Approved by the Examination Committee:

Prof. Marco Antonio Casanova
Advisor
Departamento de Informática – PUC-Rio

Prof. Antonio Luz Furtado
Departamento de Informática – PUC-Rio

Prof^{a}. Melissa Lemos Cavaliére
Departamento de Informática – PUC-Rio

Prof. Luiz André Portes Paes Leme
UFF

Prof. Alberto Henrique Frade Laender
UFMG

Rio de Janeiro, January 29th, 2021

Yenier Torres Izquierdo

The author holds a bachelor's degree in Computer Science from the University of Havana (UH), Havana–Cuba, in 2012. He joined the Masters in Informatics at the Pontifical Catholic University of Rio de Janeiro (PUC–Rio) in 2015 and obtained his M.Sc. in 2017. In March 2017, he also started his Ph.D. in Informatics at PUC–Rio. His main research topics are Linked Data, Semantic Web, and Information Retrieval.

Bibliographic data

Torres Izquierdo, Yenier

Contributions to the Problem of Keyword Search over Datasets and Semantic Trajectories Based on the Resource Description Framework / Yenier Torres Izquierdo; advisor: Marco Antonio Casanova. – 2021.

144 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2021.

Inclui bibliografia

1. Informática – Teses.
2. Pesquisa por palavras-chave. 3. Grafo RDF. 4. SPARQL. 5. Sinopses KMV. 6. Trajetórias semânticas. 7. Expressões de Sequência. I. Casanova, Marco Antonio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

I would like to give a special thanks to my parents, Katiuska and Alberto, for their support and encouragement during all these years of study. To my family and true friends which contributed to the accomplishment of this challenge.

Thank you so much to Professor Marco Antonio Casanova, the best advisor I could ever have. I have a deep admiration for his professionalism, patience and dedication to his students.

To thank also PUC-Rio, Tecgraf Institute, CNPq, and FAPERJ for funding my research.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

To all my classmates, professors and staff from the Department of Informatics.

Thanks to all for your help and for always being so accommodating.

Thank you so much to all of you!

Abstract

Torres Izquierdo, Yenier; Casanova, Marco Antonio (Advisor). **Contributions to the Problem of Keyword Search over Datasets and Semantic Trajectories Based on the Resource Description Framework**. Rio de Janeiro, 2021. 144p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Keyword search provides an easy-to-use interface for retrieving information. This thesis contributes to the problems of keyword search over schema-less datasets and semantic trajectories based on RDF.

To address the keyword search over schema-less RDF datasets problem, this thesis introduces an algorithm to automatically translate a user-specified keyword-based query K into a SPARQL query Q so that the answers Q returns are also answers for K. The algorithm does not rely on an RDF schema, but it synthesizes SPARQL queries by exploring the similarity between the property domains and ranges, and the class instance sets observed in the RDF dataset. It estimates set similarity based on set synopses, which can be efficiently precomputed in a single pass over the RDF dataset. The thesis includes two sets of experiments with an implementation of the algorithm. The first set of experiments shows that the implementation outperforms a baseline RDF keyword search tool that explores the RDF schema, while the second set of experiments indicate that the implementation performs better than the state-of-the-art TSA+BM25 and TSA+VDP keyword search systems over RDF datasets based on the "*virtual documents*" approach. Finally, the thesis also computes the effectiveness of the proposed algorithm using a metric based on the concept of graph relevance.

The second problem addressed in this thesis is the keyword search over RDF semantic trajectories problem. Stop-and-move semantic trajectories are segmented trajectories where the stops and moves are semantically enriched with additional data. A query language for semantic trajectory datasets has to include selectors for stops or moves based on their enrichments, and sequence expressions that define how to match the results of selectors with the sequence the semantic trajectory defines. The thesis first proposes a formal framework to define semantic trajectories and introduces stop and move sequence expressions, with well-defined syntax and semantics, which act as an expressive query language for semantic trajectories. Then, it describes a concrete semantic trajectory model in RDF, defines SPARQL stop-and-move sequence expressions, and discusses strategies to compile such expressions into SPARQL queries. Next, the thesis specifies user-friendly keyword search

expressions over semantic trajectories based on the use of keywords to specify stop and move queries, and the adoption of terms with predefined semantics to compose sequence expressions. It then shows how to compile such keyword search expressions into SPARQL queries. Finally, it provides a proof-of-concept experiment over a semantic trajectory dataset constructed with user-generated content from Flickr, combined with Wikipedia data.

Keywords

Keyword search; RDF graph; SPARQL; KMV-Synopses; Semantic trajectories; Sequence expressions.

Resumo

Torres Izquierdo, Yenier; Casanova, Marco Antonio. **Contribuições ao Problema de Busca por Palavras-Chave em Conjuntos de Dados e Trajetórias Semânticas Baseados no Resource Description Framework**. Rio de Janeiro, 2021. 144p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Busca por palavras-chave fornece uma interface fácil de usar para recuperar informação. Esta tese contribui para os problemas de busca por palavras-chave em conjuntos de dados sem esquema e trajetórias semânticas baseados no *Resource Description Framework*.

Para endereçar o problema da busca por palavras-chave em conjuntos de dados RDF sem esquema, a tese introduz um algoritmo para traduzir automaticamente uma consulta K baseada em palavras-chave especificadas pelo usuário em uma consulta SPARQL Q de tal forma que as respostas que Q retorna também são respostas para K. O algoritmo não depende de um esquema RDF, mas sintetiza as consultas SPARQL explorando a semelhança entre os domínios e contradomínios das propriedades e os conjuntos de instâncias de classe observados no grafo RDF. O algoritmo estima a similaridade entre conjuntos com base em sinopses, que podem ser precalculadas, com eficiência, em uma única passagem sobre o conjunto de dados RDF. O trabalho inclui dois conjuntos de experimentos com uma implementação do algoritmo. O primeiro conjunto de experimentos mostra que a implementação supera uma ferramenta de pesquisa por palavras-chave sobre grafos RDF que explora o esquema RDF para sintetizar as consultas SPARQL, enquanto o segundo conjunto indica que a implementação tem um desempenho melhor do que sistemas de pesquisa por palavras-chave em conjuntos de dados RDF baseados na abordagem de "*documentos virtuais*" denominados TSA+BM25 e TSA+VDP. Finalmente, a tese também computa a eficácia do algoritmo proposto usando uma métrica baseada no conceito de relevância do grafo resposta.

O segundo problema abordado nesta tese é o problema da busca por palavras-chave sobre trajetórias semânticas baseadas em RDF. Trajetórias semânticas são trajetórias segmentadas em que as paradas e os deslocamentos de um objeto móvel são semanticamente enriquecidos com dados adicionais. Uma linguagem de consulta para conjuntos de trajetórias semânticas deve incluir seletores para paradas ou deslocamentos com base em seus enriquecimentos e expressões de sequência que definem como combinar os resultados dos seletores com a sequência que a trajetória semântica define. A tese inicialmente propõe um framework formal para definir trajetórias semânticas e introduz

expressões de sequências de paradas-e-deslocamentos (*stop-and-move sequences*), com sintaxe e semântica bem definidas, que atuam como uma linguagem de consulta expressiva para trajetórias semânticas. A tese descreve um modelo concreto de trajetória semântica em RDF, define expressões de sequências de paradas-e-deslocamentos em SPARQL e discute estratégias para compilar tais expressões em consultas SPARQL. A tese define consultas sobre trajetórias semânticas com base no uso de palavras-chave para especificar paradas e deslocamentos e a adoção de termos com semântica predefinida para compor expressões de sequência. Em seguida, descreve como compilar tais expressões em consultas SPARQL, mediante o uso de padrões predefinidos. Finalmente, a tese apresenta uma prova de conceito usando um conjunto de trajetórias semânticas construído com conteúdo gerado pelos usuários do Flickr, combinado com dados da Wikipedia.

Palavras-chave

Pesquisa por palavras-chave; Grafo RDF; SPARQL; Sinopses KMV; Trajetórias semânticas; Expressões de Sequência.

Table of contents

List of figures

List of tables

1
Introduction

1.1
Context and Motivation

Keyword search is a very popular information discovery method because it allows naive users to retrieve information without any knowledge about schema details or query languages. The user specifies a few terms, called *keywords*, and it is up to the system to retrieve the documents, such as Web pages, that best match the keywords.

Traditional Information Retrieval (IR) systems allow users to search unstructured documents using keywords. They retrieve the documents that best match the keywords and rank the retrieved documents so that the top ones are the most relevant, according to some relevance criteria.

Keyword queries also offer a convenient alternative to query structured datasets. In general, keyword queries avoid the use of complex query languages, but they require tools that face the challenging task of automatically determining, from a set of keywords, what pieces of information to retrieve, and how these pieces can be combined to provide a relevant answer to the user. Note that traditional IR systems do not have to combine pieces of information since they match keywords to one document at a time.

Systems that process keyword queries over relational databases are commonly called *relational keyword search systems* or *R-KwS systems* [1, 2, 9, 30, 31]. R-KwS systems consider the relational database as a network of tuples interconnected by foreign keys. Given a keyword query, they detect those tuples that contain the keywords, generate connected components based on how these tuples are associated, and return these connected tuples as an answer to the query, as proposed in [44].

In the last decade, the Resource Description Framework (RDF) emerged as a data model that represents data as a set of triples, which in turn induces a graph. Keyword search systems over RDF datasets (or RDF graphs), or *RDF-KwS systems*, are similar to R-KwS systems. They operate over the RDF graph and, given a keyword query, retrieve nodes of the RDF graph that match the keywords, and discover how the nodes are interrelated (by paths in the RDF

graph) to compose complete answers [41]. Hence, an answer to a keyword query over an RDF graph is not just a set of nodes, but a set of nodes and paths between them.

The keyword search systems proposed in the literature for relational and RDF environments have points in common. However, RDF datasets pose an additional challenge when no schema is defined, which is never the case for relational databases. The first problem addressed in this thesis then is the *keyword search over schema-less RDF datasets problem*, precisely defined in Section 1.2.1.

Going further, the expressiveness of keyword queries can be expanded by considering terms with a predefined semantics. For example, QUIOW [33] uses reserved terms to express comparison operators, such as "*between*", "*less than*", "*greater than*", etc. As a more complex example, one may consider terms with predefined semantics that help express aggregations [70]. Along these lines, this thesis explores how to expand keyword queries to semantic trajectories.

In this case, keyword search expressions over a dataset of semantic trajectories uses keywords to specify stop and move queries and adopts terms with predefined semantics, such as "*begin*", "*end*", "*then*", and "*later on*", to define sequence expressions that match the stop and move queries with the sequence of actions defined in the semantic trajectory.

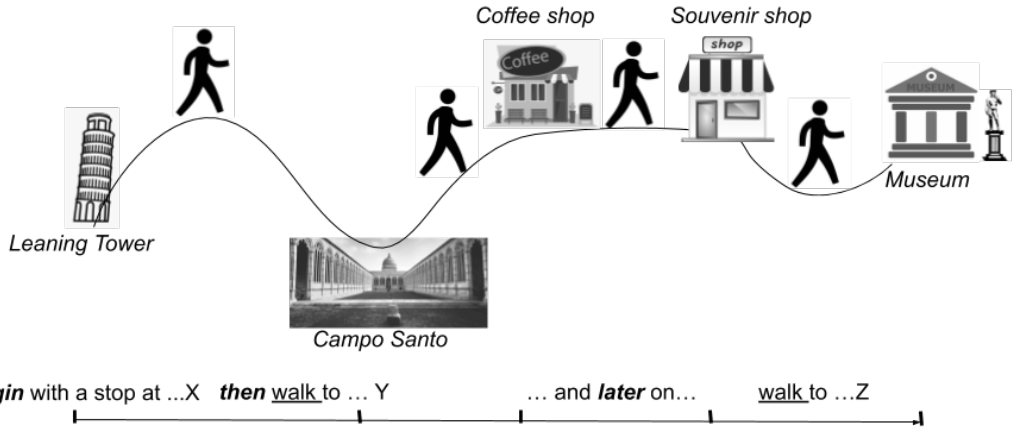


Figure 1: A semantic trajectory and a query that searches for it.
X, Y, Z match stop attributes, underlined words match move attributes, words in bold are part of the query notation

For example, consider a trajectory dataset containing tourist trips in the historic city of Pisa, Italy. One may want to submit the query "*Find walking trajectories that begin at the* ***Leaning Tower****, then stop at* ***Campo Santo****, and end at a* ***museum***" that retrieves semantic trajectories such as that in Figure 1. The intended interpretation of "*then*" is that the first two stops are consecutive, but "*later on*" indicates that there might be several stops between

Campo Santo and the **museum**, as long as all moves are by **walking**, as transportation means. Note that the terms in boldface are keywords that select points-of-interest, based on their enrichments, and transportation means, based on their characteristics.

Thus, the second problem addressed in this thesis is the *keyword search over RDF semantic trajectories problem*, precisely defined in Section 1.2.2.

1.2 Problems Addressed

This section defines the problems the thesis addresses.

1.2.1 The Keyword Search over Schema-less RDF Datasets Problem

A *keyword-based query* is a set K of *literals*, or *keywords*. An *answer* for K over an RDF dataset T is a subset A of T such that: (i) A has triples that match keywords in K; (ii) A induces a connected RDF graph. Note that we can then compare answers based on the number of keywords they match and on their number of triples, as defined in detail in [24].

Let G_A be the RDF graph induced by an answer A. If G_A is a Steiner tree of T that covers the nodes that match keywords, then G_A is connected and does not have unnecessary edges. Therefore, a high-level strategy to solve the RDF-KwS Problem would be to construct an algorithm that:

(i) find as many keyword matches as possible with nodes in the RDF graph;
(ii) find Steiner trees of T that cover the matching nodes;
(iii) rank them by relevance to the user information need.

However, this solution is challenging due to the complex and heterogeneous structure of RDF graphs, that, unlike relational databases, do not necessarily have a schema.

The first problem this thesis addresses is the *keyword search over schema-less RDF datasets problem*:

"*Given an RDF dataset T that does not follow an RDF-Schema, and a keyword-based query K, find an answer A for K over T, preferably with as many keyword matches as possible and with the smallest set of triples as possible*".

1.2.2 The Keyword Search over RDF Semantic Trajectories Problem

A *raw trajectory* consists of spatio-temporal positions extracted from a raw movement track [46]. A *segmented trajectory* is a partition of the points of a raw

trajectory into homogeneous segments, where a given set of properties holds. For example, according to the *stop-and-move model* [54], a raw trajectory can be split into segments of two kinds: *stop*, where the speed of the object is lower than a certain threshold; and *move*, where the speed is greater than such threshold. Raw trajectories are useful for applications that require only the movement track of the objects, but most applications require additional data (for instance, the city information, traffic conditions, weather data, among others). The process of adding data to the raw trajectories from external repositories is known as the *semantic enrichment process* [46].

A *semantic trajectory* is a trajectory that is segmented, using various segmentation criteria (e.g., stops, turns, etc.), into sub-trajectories and is enriched with additional data [49] that describe the segmentation points and the resulting segments. More specifically, in this thesis, we focus on *stop-and-move semantic trajectories* of humans [50], where the segmentation points are stops, and the sub-trajectories are the way humans moved from one stop to the other. Consequently, the stop-and-move semantic trajectory is enriched by data that describe the type of stops and moves. For example, a stop can be enriched with the points-of-interest (POIs) at the stop, and a move with the transportation means, duration, and distance traveled. The trajectory is usually connected to a moving object, which also has its semantic properties that can also be of interest (e.g., for a traveler's trajectory, the person's health status, age, occupation, etc.).

The second problem this thesis addresses is the *keyword search over RDF semantic trajectories problem*:

"*Given a set T of semantic stop-and-move trajectories, represented as an RDF dataset, define a keyword search language to retrieve trajectories in T. The language must include:*

(i) stop and move keyword queries that select sets of stops or moves based on their enrichments; and

(ii) sequence expressions that define how to match the stop and move queries with the sequence of actions defined in the semantic trajectory".

1.3 Goal and Contributions

The contributions of this thesis are:

- For the keyword search over schema-less RDF datasets problem:
  1. A novel algorithm to address the keyword search over schema-less RDF datasets problem by automatically translating a keyword

query K into a SPARQL query Φ so that the answers Φ returns are also answers for K. The algorithm neither relies on an RDF schema, nor accesses the RDF graph during the compilation process.

2. Two sets of comprehensive experiments with an implementation of the algorithm. The first set of experiments shows that the implementation outperforms, in all metrics adopted, a baseline RDF keyword search tool that explores the RDF schema. The second set of experiments indicate that the implementation performs better than the TSA+BM25 and TSA+VDP keyword search systems over RDF datasets based on the "virtual documents" approach, using the metrics and the benchmarks proposed originally to assess these systems.
3. A metric named *Graph Relevance Ratio* (GRR) to establish when an answer graph is relevant w.r.t. a ground truth graph. It is based on the number of relevant and non-relevant triples in the RDF graph, but it punishes the presence of non-relevant triples, and does not memorize the relevant triples in previous rank positions.

- For the keyword search over RDF semantic trajectories problem:
  1. A formal framework, based on Description Logic, to define semantic trajectories.
  2. The definition of the syntax and semantics of stop and move sequence expressions.
  3. A concrete framework that represents a semantic trajectory model in RDF.
  4. SPARQL stop and move sequence expressions templates that help compile such expressions into SPARQL queries.
  5. User-friendly keyword search expressions to specify stop and move queries, which adopt terms with predefined semantics, such as "*then*" and "*later on*", to define sequence expressions.
  6. A strategy to compile keyword search expressions into SPARQL stop and move sequence expressions, which are then compiled into SPARQL queries, taking advantage of the concrete framework.
  7. Finally, a proof-of-concept experiment to validate the proposed solution.

1.4 Thesis Structure

This thesis is organized as follows:

- Chapter 1 states the motivation, problems addressed, and contributions.
- Chapter 2 provides the necessary background.
- Chapter 3 reviews related work.
- Chapter 4 describes the proposed algorithm to compile keyword queries into SPARQL queries, which neither relies on an RDF schema, nor accesses the RDF graph during the compilation process.
- Chapter 5 evaluates the performance of an implementation of the proposed algorithm by comparing it with state-of-art systems, adopted as baselines. Also, it introduces an alternative measure to establish the relevance of an answer graph.
- Chapter 6 defines a formal framework for querying semantic trajectories, formalized in Description Logic.
- Chapter 7 introduces a concrete RDF framework for querying semantic trajectories, based on the formal framework. Also, it presents an algorithm for translating keyword query expressions over semantic trajectories in RDF to SPARQL queries. Finally, it describes a proof-of-concept experiment to validate the proposed approach.
- Finally, Chapter 8 contains the conclusions and suggests directions for future.

2
Background

This chapter presents an overview of the main concepts involved in this thesis. Section 2.1 summarizes basic topics about RDF. Section 2.2 details the main features of the SPARQL query language. Section 2.3 introduces the more significant definitions related to keyword-based queries over RDF graphs. Section 2.4 summarizes the key definitions about set similarity measures and KMV–Synopses. Finally, Section 2.5 briefly reviews some basic concepts about Description Logic.

2.1
Resource Description Framework (RDF)

The Resource Description Framework (*RDF*) is a family of specifications developed and supported by the W3C[1] to represent information about resources on the Web. RDF resources are classified into *IRIs* (*Internationalized Resource Identifier*), *literals* and *blank nodes*. An IRI[2] is a string used to globally identify a resource on the Web (it is a generalization of URIs which can also contain UNICODE characters). A *literal* is a basic value associated with a data type, e.g. String, Boolean, Integer, and Date. When a data type is not specified, the default is "String". Any IRI or *literal* denotes something in the world (the "universe of discourse"). The resource denoted by an *IRI* is called its $referent$, and the resource denoted by a *literal* is called its *literal value*. A blank node is a resource without a global identifier. It acts as a local identifier and can always be replaced by a new, globally unique IRI (a *Skolem IRI*[3]). An *RDF term* is either an IRI, a *blank node* or a *literal*. The sets of IRIs, blank nodes and literals are disjoint and, unlike IRIs and literals, blank nodes do not identify specific resources.

RDF models data as *triples* of the form *(s, p, o)*, where s is the *subject*, p is the *predicate* and o is the *object* of the triple. An RDF triple *(s, p, o)* says that some relationship, indicated by p, holds between the subject s and object o. The subject of a triple is an IRI or a blank node, the predicate is an IRI, and the object is an IRI, a literal or a blank node. As Figure 2 shows, a

[1] https://www.w3.org/TR/rdf11-primer
[2] https://tools.ietf.org/html/rfc3987
[3] https://www.w3.org/2011/rdf-wg/wiki/Skolemisation

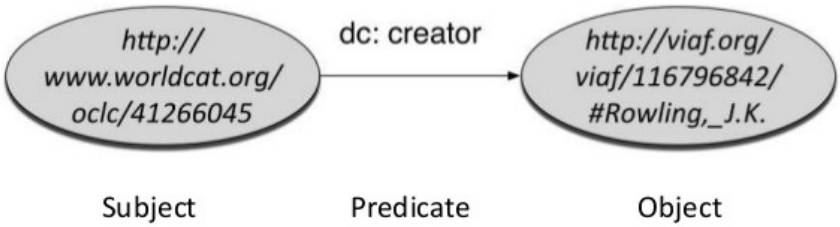


Figure 2: Example of an RDF triple

triple is also seen as an edge in a directed, labeled graph where a directed edge (labeled predicate) connects the subject node to the object node.

A set T of RDF triples, or an *RDF dataset*, is equivalent to an edge-labeled direct graph G_T, such that the set of nodes of G is the set of RDF terms that occur as subject or object of the triples in T and there is an edge *(s, o)* in G labeled with p iff the triple *(s, p, o)* occurs in T. Hence, an RDF dataset can also be called RDF graph. Note that a predicate IRI can also occur as a node in the same graph.

RDF offers enormous flexibility but, apart from the `rdf:type` property, which has a predefined semantics, it provides no means for defining application-specific classes and properties. Instead, such classes and properties, and hierarchies thereof, are described using extensions to RDF provided by the RDF Schema 1.1[4] (RDF Schema or RDF-S). In RDF-S, a *class* is any resource having an `rdf:type` property whose value is the qualified name `rdfs:Class` of the RDF Schema vocabulary. A *property* is any instance of the class `rdfs:Property`. The `rdfs:domain` property is used to indicate that a particular property applies to a designated class, and the `rdfs:range` property is used to indicate that the values of a particular property are instances of a designated class or, alternatively, are instances (i.e., literals) of an XML Schema datatype. Finally, RDF-S offers a property, `rdfs:comment`, used to associate a comment with an IRI, and a property, `rdfs:label`, used to assign a different name to a resource.

Figure 3 depicts a simple RDF graph derived from the IMDb dataset[5]. The main classes are *film* (`F`), *director* (`D`), *actor* (`A`), *profession* (`P`), and *genre* (`G`). The class instances are identified by a synthetic IRI consisting of the letter that identifies the belonging class followed by a number (e.g.; `A1` for the actor "Samuel L. Jackson", `D2` for the director "Robert Rodriguez", and `F1` for the film "Pulp Fiction"). A directed and labeled edge connects each pair of nodes.

[4]https://www.w3.org/TR/rdf-schema/
[5]https://www.imdb.com/interfaces/

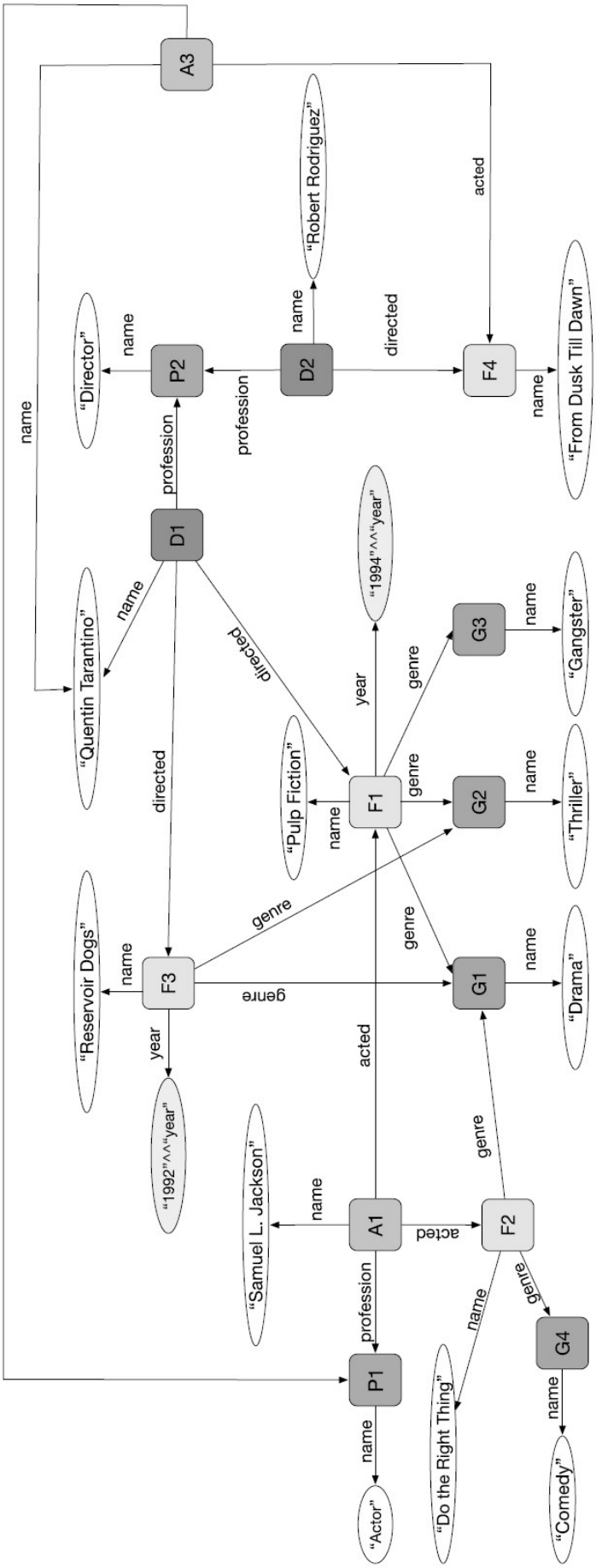


Figure 3: A sample RDF graph derived from IMDb dataset

2.2 SPARQL Query Language

SPARQL[6] is a structured language for querying RDF datasets that allows the retrieval and processing of triples. The body of a SPARQL query is a graph pattern P composed of *triple patterns*, defined like RDF triples, except that the subject, predicate or object can be a variable. The evaluation of a SPARQL query binds values to the variables using a *solution mapping* M. The application of M to a graph pattern P uniformly replaces each variable in P by the RDF term.

SPARQL offers four types of query as output: (i) a `SELECT` query that returns projections of mapping from M as a tabular data; (ii) a `CONSTRUCT` query that returns a new set of triples based on the mapping in M; (iii) an `ASK` query that returns `true` if the pattern P is matched in the input dataset or `false` otherwise; and, (iv) a `DESCRIBE` query that returns a set of triples that represent the IRIs and blank nodes found in M.

Example 1 Based on the RDF graph in Figure 3, assume that we desire to retrieve "*The title of the films directed by Quentin Tarantino*". Then, we can write the SPARQL `SELECT` query below to answer our information need.

```
SELECT ?film_name
WHERE {
  ?director_id <name> "Quentin Tarantino".
  ?director_id <directed> ?film_id.
  ?film_id <name> ?film_name }
```

The `WHERE` clause contains the graph pattern P that is matched with the RDF graph. In this example, P matches five triples (see Figure 4), identifying the director "Quentin Tarantino" with its IRI (`D1` bound to the variable `?director_id`) and the two films (`F1` and `F3` bounded to the variable `?film_id`). The `SELECT` clause projects the variable(s) in P that will appear as column(s) in the resulting table (in this case, `?film_name`). Hence, the variable `?film_name` is bound with the values "Reservoir Dogs" and "Pulp Fiction", and returned it.

[6]https://www.w3.org/TR/sparql11-query/

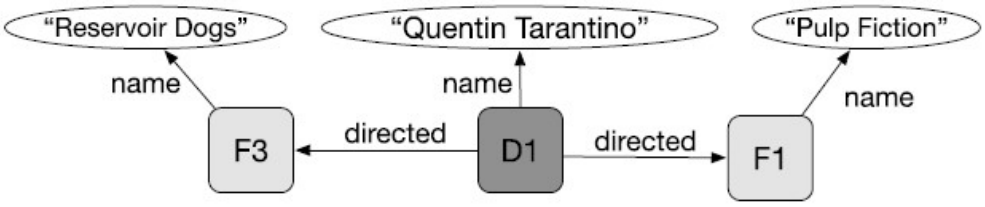


Figure 4: Graph answer to the SPARQL query reported in Example 1

2.3 RDF Keyword-based Queries

Let T be an RDF dataset and $\mathcal{L}$ be the set of all literals. A *keyword-based query* is a finite set $K = \{k_1, ..., k_n\}$ of literals, or *keywords*.

Let $\mathcal{L}$ be the set of all literals. A *match function* $\mu : \mathcal{L} \times \mathcal{L} \rightarrow Bool$ maps each pair of literals into a Boolean value is such that $\mu(L_1, L_1) = True$ and $\mu(L_1, L_2) = \mu(L_2, L_1)$, for any $L_1, L_2 \in \mathcal{L}$. We say that L_1 and L_2 *match* iff $\mu(L_1, L_2) = True$. We say that a triple $(s, p, o) \in T$ *matches* a literal L iff o is a literal and L and o match, and we also say that o is a *matching node* of G_T.

Note that a keyword may match the label of a class or property, which could alter the interpretation of the keyword-based query. For example, if `Actor` is declared as a class with label "*actor*", then the keyword query $K = \{actor, Washington\}$ may be interpreted as requesting instances of the class `Actor` that have property values that match the keyword "*Washington*".

An *answer* for K over T is a subset A of T such that there is $K_A \subseteq K$, the set of *matched keywords*, and $A_K \subseteq A$, the set of *triple matches*, such that:

- for each $k \in K_A$, there is $(s, p, o) \in A_K$ that matches k;
- for each $(s, p, o) \in A_K$, there is $k \in K_A$ matched by o;
- the RDF graph G_A induced by A is connected.

Recall that the *Schema-less RDF-KwS Problem* is defined as: "*Given an RDF dataset T, where T is a large dataset (millions of triples) and T does not (strictly) follow an RDF-Schema, and a keyword-based query K, find an answer A for K over T, preferably with as many keyword matches as possible and with the smallest set of triples as possible*".

An *answer* for K over an RDF dataset T is a subset A of T such that: (i) A has triples that match keywords in K; (ii) A induces a connected RDF graph.

We can compare answers based on the number of keywords they match and on their number of triples. Furthermore, if G_A is a Steiner tree of T that covers the matching nodes, then G_A is connected and does not have

unnecessary edges. So, if the algorithm designed to solve the *Schema-less RDF KwS–Problem* cannot find one such Steiner tree, it must abandon some of the keyword matches and restart the search for a new one.

2.4 Set Similarity Measures and KMV-Synopses

As mentioned in the Introduction, the novelty of the proposed algorithm lies in that it synthesizes SPARQL queries by exploring the similarity between the property domains and ranges and the class instance sets observed in the RDF dataset. To achieve good performance, the algorithm estimates set similarity based on KMV-synopses [10]. This chapter summarizes the essentials of these two aspects: set similarity and KMV–synopses.

Let $\mathcal{D}$ be the universe and let $A_1, ..., A_n \subseteq \mathcal{D}$.

The *Jaccard similarity measure* is a well-known way to estimate the similarity of two or more sets, $A_1, ..., A_n$, based on what elements they have in common, without giving preference to any of the sets; the measure is normalized by the number of elements in the union of the sets. The *set containment similarity measure* is adopted when one wants to find, given a set A_i which other sets A_j are similar to A_i, based on the number of elements that A_i and A_j have in common; the measure is normalized by the number of elements in A_i.

More precisely, *the n-way Jaccard similarity measure* of $A_1, ..., A_n$ as:

$$\begin{aligned} J(A_1, ..., A_n) &= \frac{|A_1 \cap ... \cap A_n|}{|A_1 \cup ... \cup A_n|} \quad \text{if} \quad A_1 \cup ... \cup A_n \neq \emptyset \\ J(A_1, ..., A_n) &= 1 \qquad\qquad\qquad\quad \text{otherwise} \end{aligned} \tag{1}$$

and *the set containment similarity measure* of A_i and A_j as:

$$\begin{aligned} C(A_i, A_j) &= \frac{|A_i \cap A_j|}{|A_i|} \qquad \text{if} \quad A_i \neq \emptyset \\ C(A_i, A_j) &= 1 \qquad\qquad\quad \text{otherwise} \end{aligned} \tag{2}$$

A generalized form of the Jaccard similarity measure can be found in [66].

Beyer et. al. [10] proposed a simple and yet powerful summarization technique, called KMV–synopses, for multiset operations. KMV stands for ***k*-*M*inimum hash *V*alue**.

Let k be a positive integer. Intuitively, a KMV–synopsis of a set $S \subseteq \mathcal{D}$ defines a random sample of S of size k, with the help of a hash function. Using the KMV-synopsis one can then estimate the cardinality of S.

More precisely, let h be a hash function from $\mathcal{D}$ to {0,...,M} (with $M \sim \mathbf{O}(|\mathcal{D}|^2)$). The *KMV–synopsis* of a set $S \subseteq \mathcal{D}$ is the set V of the k smallest values of the set $\{\ v \in \{0, ..., M\} \mid v{=}h(s)$ and $s \in S\ \}$. If h is a perfect hash function, then V induces a random sample $W = \{\ s \in S \mid h(s) \in V\ \}$ of S of size k.

The expected cost to construct a KMV–synopsis of size k from a partition S comprising N data items having D distinct values is $\mathbf{O}(N + k \cdot logk \cdot logD)$ [11].

The following example helps to clarify the KMV–synopsis. Assume that we have the feature:

$$x = \{\text{'}aa\text{'}, \text{'}bb\text{'}, \text{'}cc\text{'}, \text{'}dd\text{'}, \text{'}ee\text{'}, \text{'}aa\text{'}\}.$$

Now, suppose that the selected hash function gives the following values:

$h(\text{'}aa\text{'}) = 1$, $h(\text{'}bb\text{'}) = 3$, $h(\text{'}cc\text{'}) = 7$, $h(\text{'}dd\text{'}) = 5$, $h(\text{'}ee\text{'}) = 1$.

If we want the KMV–synopsis of this multiset with parameter $k = 2$, i.e., by keeping the two minimum hash values, then we have the following KMV–synopsis for x: {1,3}. Observe that the KMV–synopsis is a set even though the value 1 appears three times in the hash values (two for the items 'aa' and one for item 'ee').

Venetis et. al. [58] define two variations of the KMV–synopsis: (i) *incomplete*, when the corresponding feature x has more than k distinct items; (ii) *complete*, when the corresponding feature x has at most k distinct items. In this research, KMV–synopsis will be handled without distinguishing between the types.

Beyer et. al. [10] define strict and probabilistic bounds for distinct counts, unions, and intersections for incomplete synopsis.

Let $U_{(k)}$ denote the k^{th} smallest value of the KMV-synopsis V, divided by M. Then, an estimation $\overline{|S|}$ of $|S|$ is defined as

$$\overline{|S|} = \frac{(k-1)}{U_{(k)}} \tag{3}$$

with absolute ratio error given by Beyer et. al. [10]:

$$E\left[\frac{abs(\overline{|S|} - |S|)}{|S|}\right] \approx \sqrt{\frac{2}{\pi(k-2)}} \tag{4}$$

Let $V_1, ..., V_n$ be the KMV-synopses of $A_1, ..., A_n$, $k_1, ..., k_n$ be the sizes of $V_1, ..., V_n$, and $k = \texttt{min}(k_1, ..., k_n)$. Then, we define $V = V_1 \oplus ... \oplus V_n$ as the set of the k smallest values in $V_1 \cup ... \cup V_n$. Let $U_{(k)}$ denote the k^{th} smallest value of $V = V_1 \oplus ... \oplus V_n$. Finally, let $K_\cap = |V_1 \cap ... \cap V_n|$. Then, the following estimations hold [10]:

$$\overline{|A_1 \cup ... \cup A_n|} = \frac{(k-1)}{U_{(k)}} \tag{5}$$

$$\overline{J(A_1, ..., A_n)} = \frac{K_\cap}{k} \tag{6}$$

$$\overline{|A_1 \cap ... \cap A_n|} = \frac{K_\cap}{k} \cdot \frac{(k-1)}{U_{(k)}} \tag{7}$$

$$\overline{C(A_i, A_j)} = \frac{\overline{|A_i \cap A_j|}}{\overline{|A_i|}} \tag{8}$$

A detailed analysis of the accuracy of KMV–synopses to estimate unions, intersections, and Jaccard distance is given by Beyer et. al. [10]. Some works have successfully adopted KMV–synopses for a variety of query optimization problems [28, 36, 58, 48].

To illustrate this point, consider the IMDb version adopted in [18]. Table 1 shows the Jaccard values, as directly computed from the dataset, and the estimations obtained using KMV–synopses, with K=8,192 and K=32,768. Note that the accuracy of the Jaccard estimations obtained using KMV–synopses, with K=8,192, are quite reasonable.

Table 1: Approximating the Jaccard similarity measure using KMV–synopses of different sizes

Set A	Set B	J(A,B)	K=8,192			K=32,768		
			$\overline{J(A,B)}$	Abs diff.	Error vs Exact	$\overline{J(A,B)}$	Abs diff.	Error vs Exact
imdb:name	D_imdb:know_for_titles	0.8237	0.8253	0.0016	0.19%	0.8243	0.0006	0.07%
imdb:title	R_imdb:know_for_titles	0.2205	0.2185	0.0020	0.91%	0.2184	0.0021	0.95%
imdb:name	R_imdb:principal_cast	0.1768	0.1860	0.0092	5.20%	0.1856	0.0088	4.98%
imdb:title	D_imdb:principal_cast	0.1095	0.1057	0.0038	3.47%	0.1061	0.0034	3.11%
imdb:name	R_imdb:directors	0.0614	0.0593	0.0020	3.30%	0.0599	0.00145	2.36%
imdb:title	D_imdb:directors	0.5838	0.5844	0.0005	0.09%	0.5839	9.2E-05	0.02%
imdb:name	R_imdb:writers	0.0775	0.0747	0.0027	3.54%	0.0755	0.001952	2.52%
imdb:title	D_imdb:writers	0.5114	0.5067	0.0046	0.91%	0.5088	0.00255	0.50%

Notes:
1. imdb:title and imdb:name represent the sets of synopses of classes `Title` and `Name`, respectively.
2. D_p indicates the domain of property p and R_p indicates the range.
3. $J(A, B)$ indicates the Jaccard similarity measure between sets A and B.

2.5 A Brief Review of Description Logic Basic Concepts

This section summarizes some basic concepts pertaining to Description Logic (DL) [5].

Very briefly, let $\mathcal{A}$ be a DL alphabet, whose *atomic concepts* and *atomic roles* capture the classes and properties of the domain of discourse. By

definition, the *universal concept* $\top$ and the *bottom concept* $\bot$ are atomic concepts of $\mathcal{A}$, and the *identity relation* $\mathbb{I}$ is an atomic role of $\mathcal{A}$. As an abuse of notation, $\bot$ is also considered an atomic role (since $\bot$ denotes the empty set).

We assume that $\mathcal{A}$ has a set of *constants* to denote individuals of the domain. We use $c_1, c_2, \ldots$ to denote the atomic concepts of $\mathcal{A}$, $r_1, r_2, \ldots$ to denote the atomic roles of $\mathcal{A}$, and $a_1, a_2, \ldots$ to denote the constants of $\mathcal{A}$, and R_j, $j = 1, \ldots, s$, to denote *role expressions* over $\mathcal{A}$.

The variant of DL adopted determines the set of *concept expressions* over $\mathcal{A}$ and the set of *role expressions* over $\mathcal{A}$. The atomic concepts are the simplest concept expressions and the atomic roles are the simplest role expressions. We use $C_1, C_2, \ldots$ to denote concept expressions and $R_1, R_2, \ldots$ to denote role expressions. We assume that the DL variant adopted allows the *negation* of a concept expression "$\neg C_i$", *full existential quantifications* of the form "$\exists r_j.C_i$", the *union* of two concept expressions "$C_i \sqcup C_j$", the *intersection* of two concept expressions "$C_i \sqcap C_j$", the *product* of two concept expressions "$C_i \times C_j$", the *inverse* of a role expression "P_j^-", the *transitive closure* of a role expression "P_i^+", the *intersection* of two role expressions "$P_i \sqcap P_j$", and the *composition* of two role expressions "$P_i \circ P_j$". As an abuse of notation, we write "$\exists r_j.\top$" simply as "$\exists r_j$".

An *axiom* is an expression of one of the forms "$C_i \sqsubseteq C_j$", "$C_i \equiv C_j$", "$P_i \sqsubseteq P_j$", or "$P_i \equiv P_j$". An *assertion* is an expression of the form "$C_i(a_k)$" or of the form "$P_j(a_k, a_l)$".

Recall that an *interpretation* I for $\mathcal{A}$ has a set Δ^I of *individuals*, called the *domain* of I, and assigns to each atomic concept c_i of $\mathcal{A}$ a set of individuals $c_i^I \subseteq \Delta^I$, to each atomic role r_j of $\mathcal{A}$ a binary relation $r_j^I \subseteq \Delta^I \times \Delta^I$, and to each constant a_k of $\mathcal{A}$ an individual $a_k^I \in \Delta^I$. By definition, the universal concept $\top$ is interpreted as the set of all individuals, that is, $\top^I = \Delta^I$, the bottom concept $\bot$ is interpreted as the empty set, that is, $\bot^I = \emptyset$, and the identity relation $\mathbb{I}$ is interpreted as $\mathbb{I}^I = \{(x, y) \in \Delta^I \times \Delta^I | x = y\}$.

The interpretation is then recursively extended to assign a set of individuals to each concept expression of $\mathcal{A}$ and a binary relation to each role expression of $\mathcal{A}$. In particular, recall that $(\exists r_j.C_i)^I$ is the set of all individuals that r_j^I maps to some individual in C_i^I, $\{a_k\}^I$ denotes the singleton $\{a_k^I\}$, $(\neg C_i)^I = \Delta^I - C_i^I$, $(C_i \sqcup C_j)^I = C_i^I \cup C_j^I$, $(C_i \sqcap C_j)^I = C_i^I \cap C_j^I$, $(C_i \times C_j)^I = (C_i^I \times C_j^I)$, $(P_j^-)^I = (P_j^I)^-$, $(P_i^+)^I = (P_i^I)^+$, $(P_i \sqcap P_j)^I = (P_i^I \cap P_j^I)$, and $(P_i \circ P_j)^I = (P_i^I \circ P_j^I)$.

The interpretation I *satisfies* "$C_i \sqsubseteq C_j$" iff $C_i^I \subseteq C_j^I$ and "$C_i \equiv C_j$" iff $C_i^I = C_j^I$. The interpretation I *satisfies* "$C_i(a_k)$" iff $a_k \in C_i(a_k)$ and "$r_j(a_k, a_l)$"

iff $(a_k^I, a_l^I) \in r_j^I$.

3
Related Work

This chapter provides an overview of work related to RDF keyword search systems, the use of synopses to estimate similarity measures, and lists some benchmarks used to validate the performance of state-of-the-art keyword search systems. Section 3.1 summarizes the state-of-art of keyword search systems close to the proposed RDF keyword-based approach. Section 3.2 lists research that use KMV–synopses as estimators of set similarity measures. Section 3.3 briefly describes some benchmarks used to evaluate the state-of-the-art keyword search systems. Finally, Section 3.4 offers an insight into related work about semantic trajectories: approaches for representing semantic trajectories based on ontologies and researches that model semantic trajectories as sequences of stops and moves.

3.1
Keyword Search Systems

There are many R-KwS systems, however in recent years the keyword query search over RDF graphs has also attracted considerable attention. A survey of keyword-based query processing tools over relational and RDF datasets is given by Bast et. al. [8].

Early relational keyword-based query processing tools [1, 2, 30, 31, 44] explored the foreign/primary keys declared in the relational schema to compile a keyword-based query into an SQL query with a minimal set of join clauses –and this is a key idea– based on the notion of candidate networks (CNs). This approach was also adopted in recent tools [44, 9]. In particular, QUEST [9] explores the structure of the conceptual schema to synthesize an SQL query based on a Steiner tree that induces a minimum set of joins. A recent article [59] specifically investigated CN scoring functions and empirically demonstrated that the proposed function outperforms earlier scoring functions.

A recent article [48] also explored relational schema information to compile keywords into SQL queries over databases exposed on the Web. The system, called SQUIRREL, selects and ranks relational databases on the Web, based on the metadata the databases expose, pre-processes the keywords, which includes identifying aggregation functions the keywords might express,

and compiles the pre-processed keywords into SQL queries. The authors compared favorably SQUIRREL with an earlier system [9] on a relational database with nine keyword queries. This comparison was extended in [45].

The algorithm proposed in this research compiles a keyword-based query into a SPARQL query that includes restriction clauses that represent keyword matches and join clauses that connect the restriction clauses. Each answer of the SPARQL query then corresponds to a subgraph of the RDF graph that contains literal nodes that match the keywords and paths that connect the literal nodes. Without such join clauses, an answer would be a disconnected set of nodes of the RDF graph, which hardly makes sense. The generation of the join clauses builds upon the idea of candidate networks.

An RDF keyword-based query processing tool can be *schema–based*, when it exploits the RDF schema to compile a keyword-based query into a SPARQL query, or *graph–based*, when it directly explores the RDF dataset or summaries thereof. The algorithm described in this thesis falls into this last category.

QUIOW [33] is a fully automatic, schema-based tool that supports keyword-based query processing for both the relational and RDF environments. It is an extension of the algorithm proposed in [24]. The tool constructs a Steiner tree that covers a set of nodes (relation schemes or RDF classes) whose instances match the largest set of keywords and incorporates a backtracking step to further expand the keyword-query results by generating alternative (SQL or SPARQL) queries. Experiments reported in this work adopt this tool as a baseline.

QUICK [65] is another example of an RDF schema-based tool. It translates keyword-based queries to SPARQL queries with the help of the user, who chooses a set of intermediate queries, which the tool ranks and executes.

As for graph-based tools, SPARK [71] uses techniques, such as synonyms from WordNet and string metrics, to map keywords to knowledge base elements. The matched elements in the knowledge base are then connected by minimum spanning trees from which SPARQL queries are generated. A recent paper [52] also explores WordNet and proposes a ranking method to implement keyword search over RDF graphs. Elbassuoni & Blanco [20] described a technique to retrieve a set of subgraphs that match the keywords and to rank them based on statistical language models. Ranking answers of a keyword-based query is addressed, for example, in [25]. Keyword expansion, as reported in [48, 52, 71], and the processing of reserved terms, as is in [33, 48], are complementary to the discussion in this research.

Han et. al. [29] described an algorithm that uses the keywords to first

obtain elementary query graph building blocks, such as entity/class vertices and predicate edges, and then applies a bipartite graph matching-based best-first search method to assemble the final query. Tran et. al. [55] introduced the idea of generating summary graphs for the RDF graph, using the class hierarchy, to generate and rank candidate SPARQL queries. Le et. al. [37] also proposed to process keyword queries using another RDF graph summarization algorithm. Zheng et. al. [69] adopted a pattern-based approach. Lin et. al. [38] summarized all the inter-entity relationships from RDF data to translate keywords to SPARQL queries. Finally, Wen et. al. [61] introduced another graph summarization technique that amounts to recovering an RDF schema from the RDF graph. We abandoned a similar strategy early on, in preliminary experiments, since the algorithm proposed in this thesis outperformed it.

Wang et. al. [60] proposed a clustered-graph structure that summarizes the original ontology. This reduced data space is then used to compute the top-k queries, which are ranked by query length, the relevance of ontology elements w.r.t. the query and the importance of ontology elements. The authors use TAP, DBLP and LUBM for the experiments, and introduce a new metric, called Target Query Position (TQP). Section 4.4 explain how we rank the query results, which also depends on the importance of the RDF nodes, classes and properties. LUBM is one of the datasets we use in Section 5.2.2.

Gkirtzou et. al. [26] presented an approach for keyword search for temporal RDF graphs that automatically compiles keyword queries into a set of candidate SPARQL queries. To support temporal exploration, the method is enriched with temporal operators allowing the user to explore data within predefined time ranges. The novelty of the approach lies exactly on this enrichment, which we do not explore in this article. In particular, the authors use the reciprocal rank metric in the experiments, as we do in Section 5.1.2.

Yoghourdjian et. al. [64] developed a retrieval model for keyword queries over RDF knowledge graphs that only retrieves the top-k scored subgraphs for the given query based on a scoring function. The authors adopted YAGO a large-scale general-purpose RDF knowledge graph derived from Wikipedia and WordNet, and the average NDCG as metric for the experiments. The query compilation strategy described in Section 4.3 also includes a ranking score, described in Section 4.4, that limits the query results to the top-k. Sections 5.1.2 and 5.2.2 show that the combination of the query compilation approach based on KMV–synopses and the ranking function outperforms the adopted state-of-the-art baselines approaches.

Ma et. al. [39] described a keywords-to-SPARQL translation process that circumvents the lack of underlying schema information. They compute, from

the RDF data graph, an inter-entity relationship summary with complete schema information, and adopt a search prioritization scheme that combines the degree of a vertex with the distance from the original keyword element. Finally, the approach finds the top-k subgraphs, which are relevant to the conjunction of the entering keywords. The approach we propose in this thesis uses KMV–synopses to capture graph information, as explained in Section 4.3, and ranks the SPARQL query results based on a more sophisticated node importance measure, as discussed in Section 4.4.

Recently, Dosso & Silvello [18] proposed the TSA+BM25 and the TSA+VDP keyword search systems over RDF datasets based on the "virtual documents" approach. These systems move most of the computational complexity off-line and then exploits highly efficient text retrieval techniques and data structures to carry out the on-line phase. The authors show that these approaches are more efficient and effective, when compared with state-of-the-art systems.

Contrasting with these approaches, the algorithm described in this work adopts KMV–synopses [10] to concisely represent the property domains and ranges, and class instance sets. The algorithm then uses the KMV–synopses to estimate set similarity measures that in turn drive the process of compiling a keyword-based query into a SPARQL query (see Section 4.3). The algorithm also incorporates RDF resource ranking [41] to improve the query compilation process and to rank answers (see Section 4.4).

3.2 Synopses as Estimators for Set Similarity

KMV–synopsis can be viewed as originating in [7], but they did not discuss implementation, construction, or combination of such synopses.

According to Beyer et. al. [10], KMV–synopses permit estimating the cardinality of multiset expressions and several set similarity measures, including a generalized Jaccard similarity measure for more than two sets. Beyer et. al. [11] gives continuity to [10] providing a unified view of prior synopses and DV estimators. They also introduced the Augmented KMV (AKMV) synopsis concept, showed how to estimate the number of DVs for a compound partition using the partition's AKMV synopsis, and then generalized the unbiased distinct values estimator. Other solutions, as in [28], used set synopses to estimate the size of the result of set similarity queries providing a robust estimation with minimal computational cost and storage overhead.

Yang et. al. [63] introduced a KMV sketch technique to address the problem of approximating containment similarity search, outperforming an

advanced LSH method in terms of the space-accuracy trade-off, time-accuracy trade-off, and the sketch construction time. Venetis et. al. [58] proposed a similarity index for set-valued features based on KMV–synopses. The index methods proposed in these last two references could be useful to filter the candidate property domains and ranges and class instance sets to be included in a SPARQL query during the query compilation process.

Finally, Le et. al. [36] chose to use KMV–synopses precisely because they allow estimating the size of the intersection of multiple sets (not just binary intersection) in the context of rewriting queries on SPARQL views. Their motivation for adopting KMV–synopses is, therefore, quite similar to ours. The authors also briefly commented on the problem of KMV–synopses maintenance, which is outside the scope of this work. The present implementation recomputes the KMV–synopses when necessary, much in the same way that database systems recompute statistics.

3.3 Benchmarks for Evaluating Keyword Search Systems

A crucial aspect of keyword search is the evaluation of the systems. In the last years, the research community has concentrated the efforts on the evaluation of keyword search over relational databases [8] and an extensive evaluation has been conducted on the subjects [6]. Unfortunately, benchmarks to assess keyword search systems on graph data are scarce [18].

To remedy this situation, some authors, as García et. al. [24], adapted the Coffman's benchmark [16] originally developed for relational databases. This approach, however, depends on the triplification of relational databases, and does not easily induce sets of relevant query answers [33].

In fact, the state-of-the-art RDF keyword search systems use different benchmarks, which are not always available, as shown in Table 2. As a consequence, comparing such systems turns out to be a difficult task.

To circumvent this issue, Neves et. al. [42] proposes an offline method that helps build RDF keyword search benchmarks. They introduce the concept *solution generator* to produce a set of answers given a keyword-based query. In order to avoid the manual analysis of query results, they adapted Coffman's benchmark in three aspects: (1) it uses triplified versions of Mondial and IMDb; (2) it includes only keyword-based queries that explore the structure of the RDF graph; (3) for each keyword-based query, it contains a ranked list of *solution generators*. Then, the benchmark produced by [42] was used to compare the proposed approach in this thesis with a baseline based on RDF schema [33], see details in Section 5.1.

Table 2: Summary of the benchmarks used in some state-of-the-arts keyword search systems

Tool	Ref.	Year	Description of Benchmark Used
SPARK	[71]*	2007	Database and keyword queries from Mooney Natural Language Learning Data
QUICK	[65]*	2009	An initial set of queries was extracted from a query log of the AOL search engine. Then, the queries were pruned based on the visited URLs, obtaining 3,000 sample keyword queries for IMDb and Lyrics Web pages. This process yielded 100 queries for IMDb, and 75 queries for Lyrics, consisting of 2–5 keywords.
	[55]*	2009	DBLP, TAP (`http://tap.stanford.edu`) and LUBM; 30 queries for DBLP, and 9 for TAP
	[16]†	2010	Samples of the Mondial, IMDb, and Wikipedia datasets; 50 queries for each dataset (not real user queries extracted from a search engine log).
	[20]*	2011	Datasets derived from the LibraryThing community and IMDb; and 15 queries for each dataset.
	[37]*	2014	Datasets: LUBM, Wordnet, BSBM, Barton and DBpedia Infobox. 12 Queries: 4 for LUBM, 2 for Wordnet, 2 for BSBM, 2 for Barton, 2 for DBpedia Infobox.
	[69]	2016	DBpedia and Yago; queries derived from QALD-4
	[29]	2017	DBpedia+QALD-6 and Freebase* + Free917: an open QA benchmark which consists of NL question and answer pairs over Freebase.
QUIOW	[33]	2018	Full versions of the Mondial and IMDb datasets, and queries from Coffman's benchmark.
	[38]	2018	LUBM, Wordnet, BSBM, Barton and DBpedia Infobox; 4 queries for LUBM, and 10 queries for the other datasets.
KAT	[61]	2018	YAGO, DBLP and LUBM; 9 queries for YAGO, 3 queries for DBLP, and 6 queries for LUBM.
	[52]*	2018	AIFB and DBpedia; 10 queries for each dataset (the sizes of the queries were between 2 and 8 keywords).
QUIRA	[41]	2019	Full versions of IMDb and MusicBrainz; 50 queries from Coffman's benchmark for IMDb, and 25 queries from QALD-2 for MusicBrainz. Details available at `https://sites.google.com/view/quira/`
TSA+BM25 and TSA+VDP	[18]	2020	**Real datasets:** LinkedMDB (`https://data.world/linked-data/linkedmdb`), IMDb, and a subset of DBPedia; 50 queries of Coffman's benchmark for each dataset. **Synthetic datasets:** LUBM and BSBM; 14 queries for LUBM, and 13 queries for BSBM.

* Datasets have no public link or are not available for download.
† Benchmark for evaluating keyword search systems over relational databases.

A recent benchmark for relational keyword-based systems also proposed to use Mondial and IMDb, as well as DBLP and Northwind [45].

Finally, Dosso & Silvelo [18] also described a benchmark that contains three real datasets: LinkedMDB, IMDb, and a subset of DBpedia, as defined in [6]; and two synthetic databases: the Lehigh University Benchmark (LUBM) [27] and the Berlin SPARQL Benchmark (BSBM) [13]. For IMDb, they designed 50 keyword queries, together with their correct translations to SPARQL queries, which were built manually. For DBpedia, the authors considered 50 topics from the classes `QALD2_te` and `QALD2_tr`. LUBM benchmark provides 14 SPARQL test queries in `SELECT` form. Then, Dosso & Silvello converted these queries to `CONSTRUCT` queries and produced their equivalent keyword query. BSBM benchmark has 13 different `SELECT` SPARQL queries, which they also converted to `CONSTRUCT` SPARQL queries and keyword queries, as for LUBM.

For the experiments described in Section 5.2, we adopt, in part, the

benchmark proposed by [18]. For the purposes of our comparison, we used the 10 million triples versions of LUBM and BSBM, the full triplified version of IMDb, and the mentioned subset of DBpedia. We considered it unnecessary to repeat the experiments for LinkedMDB, since this dataset is smaller than the adopted version of the synthetic databases.

3.4 State-of-Art Semantic Trajectories Approaches

3.4.1 Semantic Trajectories and Ontologies

The formalization of the trajectory semantic enrichment process has been firstly outlined in [21], where the *Baquara* framework has been proposed as a general, all-inclusive ontology, representing both the trajectory and its enriching concepts. This pioneering approach set the way to the use of ontologies to support the enrichment process with Linked Open Data. Despite its broadness, which allows the Baquara ontology, to cover a broad range of applications, it also has some limitations. For example, it follows a "monolithic" approach that is complex and difficult to personalize to different needs. Furthermore, it employs a predefined set of Linked Open Data sources to enrich the trajectory.

In Baquara[2], Fileto et. al. [22] introduced the concepts of movement segments (similar to the trajectory segments) and events, such as stops and moves. Baquara[2] enables queries referring to concepts, which can be expressed in SPARQL and its extensions. It also defines hierarchies and annotations for movement objects that can capture the events of a trajectory at different granularities and assign properties to them. The demonstrated implementation mostly focuses on the creation of the semantically enriched dataset, and only provides queries on stop events, or in consecutive stop events. The datAcron ontology [53] assumes different conceptualization of trajectories, including that of temporal sequences of meaningful trajectory segments (each revealing specific behavior, event, goal, activity, etc.), which is closer to our representation of trajectories as sequences of stops and moves. From an implementation point of view, datAcron employs an iterative procedure that repeats the execution of a parameterized SPARQL query, with different parameters in each iteration, until all the necessary information is collected. This technique has similarities to the SPARQL templates technique that we employ for synthesizing SPARQL queries of higher complexity. The Geo-Ontology presented in [32], relies on the spatio-temporal features of

the trajectory points, and introduces some interesting relations, such as the 'isTraversedBy', which allows to retrieve trajectories that cross a bounded area. It also identifies the begin and end points of trajectories and uses concepts such as *hasNext*, *hasSuccessor*, *hasPrevious*, and *hasPredecessor* to identify the ordering of points in the trajectory. However, it does neither support intercalated stop and move sequence queries, nor provides any SPARQL implementation examples. Spaccapietra et. al. [54] define `Stops` and `Moves` as parts of the trajectory and *Begin* and *End* stops as specializations of `Stop`. They focus only on the various facets of stops and moves, ignoring their ordering.

One of the first approaches that tried to conceptualize movement data as a trajectory ontology was proposed in [62]. The conceptual framework was aimed at combining in a unique top-level ontology the different aspects of the movement embedded into three main ontologies representing: i) application-related information, ii) the spatio-temporal details of the trajectory, and iii) geographic information. Despite the ability it offers to query the Semantic Trajectory Ontology for the features of a trajectory (its begin and end points, its stops and moves), the proposed framework offers a conceptual and top-level vision of trajectories, without explicitly dealing with the problem of the enrichment process or adopting the Linked Open Data formalism.

Another approach for representing semantic trajectories based on ontologies was proposed in [4], and employed Ontology Engineering techniques to connect Generic Places Ontologies with POI instances. The approach focused only on the enrichment of POIs with the proper Ontologies terms, while our approach faces all steps involved in the trajectory enrichment process and analysis.

A few years later, Renso et. al. [50] made a step towards employing the *Athena* ontology into a reasoning process based on OWL. The ontology comprised an application part that defined the application domain analysis concepts and a core part that represented the segmented trajectories. The objective was to support meaningful pattern interpretations of human behavior by combining inductive and deductive reasoning.

Hu et. al. [32] introduced a geo-ontology design pattern for semantic trajectories that is very similar to our Segmented Trajectory Ontology. A formal encoding of the classes, together with their properties, is obtained by using OWL. The authors also define several interfaces to integrate related geographic information, domain knowledge, and device data. This work goes a step beyond the approach proposed by [32], since it also faces the issues of how to implement the enrichment step by using Linked Data Mashups.

There are many works in the literature [62, 32, 50] proposing trajectory ontologies that can be easily adapted to specific application needs by adding specializations of classes and properties. It is worth observing that apart of a name or other features, a stop is also characterized by a spatial location. Naturally, the specific ontology can be tailored to the application needs, and other specializations are possible, such as the transport mode segmentation [68] or the activity segmentation [67].

3.4.2 Semantic Trajectories as Sequences of *Stops* and *Moves*

Researchers in the past have extended query languages with operations, which allow retrieving trajectories that satisfy temporal (e.g., TQML [15]), or spatial criteria (e.g., Spatial SQL [19]), or the combination of spatial and temporal features concerning the stops and moves of trajectories (e.g., ST-DMQL [12, 57]). When it comes to semantic trajectories the information that needs to be stored and retrieved by queries is much richer and the query requirements can be more complicated, both in the criteria that have to match (i.e., selection) and on the information aspects that have to be retrieved for the trajectories (i.e., projection).

When dealing with semantic trajectories, it is important to be able to identify interesting trajectories, or sub-trajectories, using exact or approximate matching on the semantics. It is also important to support partial matching, as well as specific operations that capture the semantic properties of a trajectory [3, 62, 46]. As it was recently shown, it is also important to be able to match trajectories that contain stops (or moves) in a specific order (i.e., ordered sequence) or in any order (i.e., set) [47, 23]. In several scenarios (e.g., in the maritime scenario), the concept of 'turns' is also introduced, thus adding more complexity to the management of semantic trajectories. Finally, the *datAcron* ontology [53] is used for describing semantic trajectories, mostly associated with the aircraft domain, as a succession of sub-trajectories associated with points or regions.

In the following, this work focus on a scenario that aims in retrieving trajectories of interest using approximate criteria and semantic matching operators of increased flexibility compared to exact matching. The scenario assumes human trajectories in an urban environment, which are composed of stops and moves, and each one refers to a single human. The semantics of the trajectory refers to various aspects of stops and moves. The task, in this case, is to retrieve trajectories that match (partially or to a certain extent) the user requirements so that they can be used as input to data mining tasks such as

trajectory clustering, classification or trajectory recommendation.

4
Keyword Search Algorithm using KMV–Synopses

This chapter presents the keyword search algorithm that uses KMV–synopses which is the main contribution of Part I of this thesis. Section 4.1 presents a complete, explanatory example of how the proposed algorithm uses the similarity between property domains and ranges. Section 4.2 defines the notion of query graph Section 4.3 details the proposed algorithm to compile a SPARQL query from a keyword-based query over an RDF dataset, using KMV–synopses. Finally, Section 4.4 explains how matches between keywords and class and property labels are handled, and the use of resources ranking in the compilation query process.

4.1
An Explanatory, Motivational Example

As already mentioned, the proposed algorithm synthesizes SPARQL queries by exploring the similarity between the property domains and ranges, and the class instance sets observed in the RDF dataset. Section 4.4 will clarify the use of class instance sets. The following example illustrates how the proposed algorithm uses the similarity between property domains and ranges.

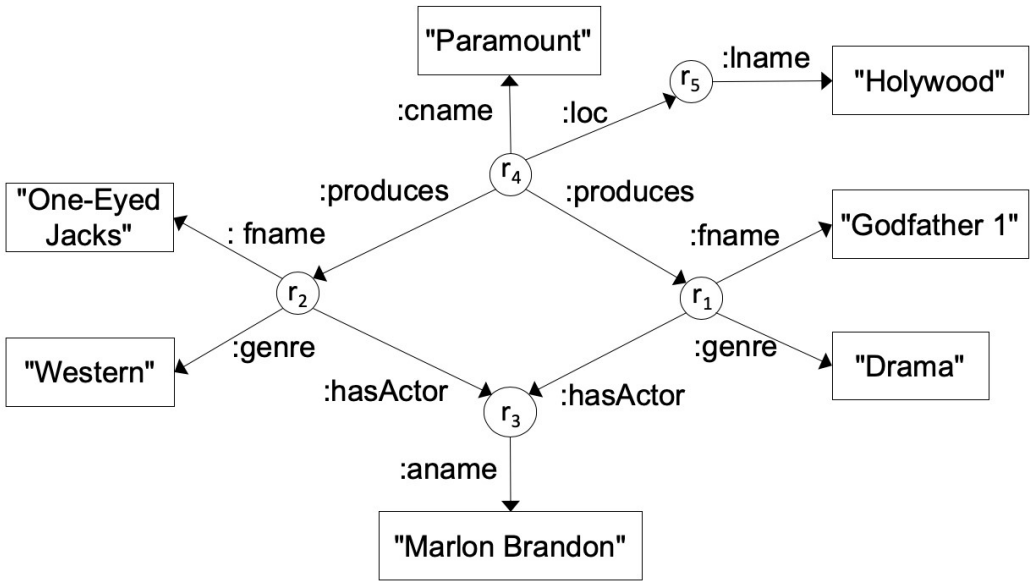


Figure 5: The graph G of an RDF dataset T

Example 2 Let T be the RDF dataset whose graph is depicted in Figure 5. In what follows, let D_p and R_p denote the observed domain and the observed range of a property p observed in T.

• *Pre-condition.* Before processing any keyword-based query, the algorithm executes a single scan of T that simultaneously pre-computes KMV–synopses for the observed property domains, non-literal ranges, and class instance sets. The KMV–synopses are stored with the RDF dataset to be later used to estimate set similarity. During the scan, the algorithm also pre-computes indexes for T that, given a keyword k, return the names of the properties that have values that match k.

Consider the keyword-based query:

$$K = \{\ \textit{One-Eyed, Western, Brandon, Hollywood}\ \}.$$

The algorithm uses a data structure, called a *query forest*, as in Figures 6 to 10, where a rectangular node is labeled with a keyword and an oval node with a list of property domains and ranges, to be interpreted as indicating their intersection. The algorithm starts with a query forest as in Figure 6 and then gradually tries to reduce the forest to a single tree by using three operations: *node fusion*, *edge addition*, and *tree expansion*. Lastly, it compiles the final forest into a SPARQL query that returns answers for K.

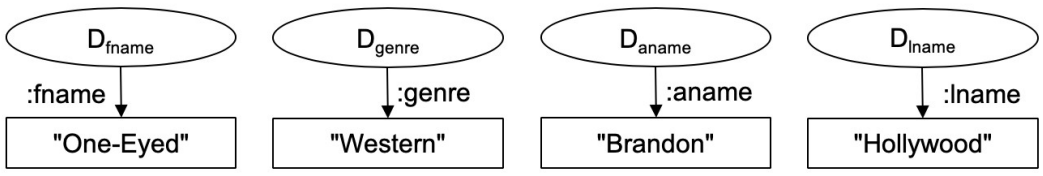


Figure 6: Initial query forest for the keyword-based query $K = \{$*One-Eyed, Western, Brandon, Hollywood*$\}$

• *Keyword matching.* First observe that the keyword query K identifies entity sets by listing keywords that should match property values of the entities. So, the first step is to match the keywords with property values and identify which are these properties. We admit partial matches so that, for example, the keyword "*One-Eyed*" matches the literal "*One-Eyed Jack*". Assume that the indexes return the following matches for K:

$Western$	matches some value of	`:genre`
$One-Eyed$	matches some value of	`:fname`
$Brandon$	matches some value of	`:aname`
$Hollywood$	matches some value of	`:lname`

This information is represented as an initial query forest, as shown in Figure 6.

Consider the leftmost tree. The oval node is labeled with D_{fname} and the rectangular node with "*One-Eyed*" to indicate that there is at least one entity e in the domain of `:fname` such the value of `:fname` for e is a string that matches "*One-Eyed*". The last step of the process synthesizes a SPARQL query that locates the set of all such entities. The other trees in Figure 6 should be likewise interpreted.

If a keyword matches values of several properties, one such match is chosen, using a combination of a literal matching score and a node ranking score. A brief description of the disambiguation strategy adopted is described in Section 4.4.

• *Node fusion.* The entities in a set might be identified by more than one property value, that is, by more than one keyword. In general, the algorithm handles this situation inspecting only the KMV–synopses of the property domains, through an operation called *node fusion*. In terms of a query forest, node fusion combines two trees by finding two nodes, one from each tree, that can be profitably replaced by a single node. By profitable we mean that the sets that label the nodes to be fused have a high Jaccard similarity value. Note that the Jaccard similarity is computed over a list of 2 or more sets, as in Equation (1). This indicates that, with a high probability, one may find entities s such that the properties that label the nodes are all defined for s (see Prop. 1a). Again, the last step of the process synthesizes a SPARQL query that takes this situation into account.

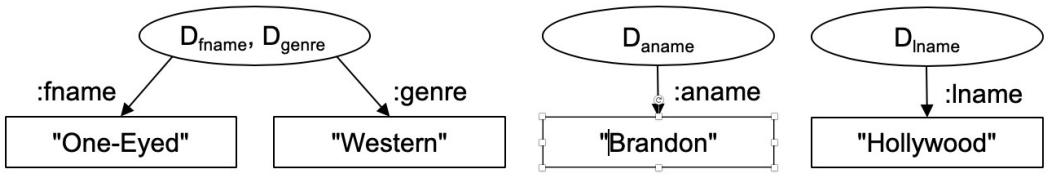


Figure 7: The query forest after node fusion

Figure 7 shows the query forest after the fusion of the roots of the two leftmost trees, respectively labeled with D_{fname} and D_{genre}. This decision is justified by estimating the Jaccard similarity between each pair of sets that label the roots of the trees in Figure 6.

For this very simple example, we computed the Jaccard similarities by just observing the RDF graph in Figure 5, that is, we in fact ignore the KMV–synopses since all sets involved in this example have low cardinality.

Since $D_{fname} = D_{genre} = \{r_1, r_2\}$, we have:

$$J(D_{fname}, D_{genre}) = 1$$

which implies that any resource s drawn from $D_{fname} \cup D_{genre}$ is in $D_{fname} \cap D_{genre}$. Furthermore, any other pair of sets that label the roots of the trees in Figure 6 are disjoint. Hence, their Jaccard similarity is 0 (zero). Thus, any other pair of roots are not good candidates for node fusion.

A new node, labeled with $\{D_{fname}, D_{genre}\}$, replaces the original nodes to signal that now we want to find resources that are in the intersection $D_{fname} \cap D_{genre}$ (each such resource has values for both properties `:fname` and `:genre` that may match two keywords, "*One-Eyed*" and "*Western*").

The entity sets identified by the keywords do not constitute answers, though, since an answer to a keyword query has to indicate how the entities are related. The algorithm then tries to identify what paths might exist in the RDF graph that connect the entities, using only the KMV–synopses, without actually traversing the graph. This is purpose of two other operations, called *edge addition* and *tree expansion*.

• *Edge addition.* Edge addition tries to find an object property p that might directly connect two entities. In this example, the key point is that the domain of property `:title` and the domain of property `:genre` might have elements in common with the domain of the object property `:hasActor`; simultaneously the range of `:hasActor` might have elements in common with the domain of `:aname` (*actor name*). This is detected again using set similarity, estimated using the pre-computed KMV–synopses. The last step of the process synthesizes a SPARQL query that has a join clause that takes this situation into account.

In terms of a query forest, edge addition tries to combine two trees by finding two nodes, a_1 and a_2, one from each tree, that can be profitably connected by a new edge, labeled with a property p, in the following sense. Let S_i be the intersection of the sets that label a_i. One should select nodes a_1, $_2$, and a property p so that, with a high probability, there is a triple (s, p, o) in T such that s is in the intersection of S_1 and the domain of p, given that we know that s is in S_1, and o is in the intersection of S_2 and the range of p, given that we know that o is in S_2 (see Prop. 1d). We use set containment for this purpose, as explained below.

Figure 8 shows the query forest after combining the two leftmost trees in Figure 7 by adding an edge, labeled with "*:hasActor*". This decision is based

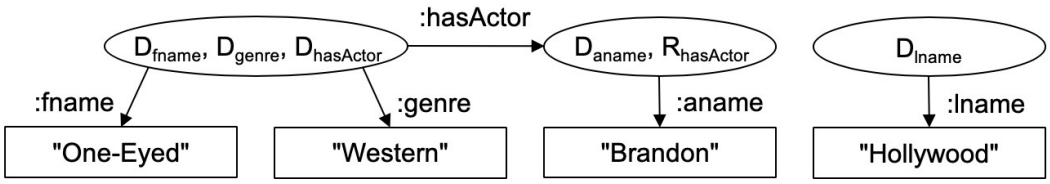


Figure 8: The query forest after adding an edge labeled with "*:hasActor*"

on the fact that the set containment of $D_{fname} \cap D_{genre}$ and $D_{hasActor}$ is

$$C(D_{fname} \cap D_{genre}, D_{hasActor}) = 1$$

which implies that any resource s in $D_{fname} \cap D_{genre}$ is also in $(D_{fname} \cap D_{genre}) \cap D_{hasActor}$. Note that we switched from Jaccard to set containment since we now know that s is in $D_{fname} \cap D_{genre}$ and we want to find a property p that maximizes the chances that s is also in D_p.

But this is not enough since the new edge should connect the two nodes. By a similar argument, the set containment similarity of D_{aname} and $R_{hasActor}$ is

$$C(D_{aname}, R_{hasActor}) = 1$$

which implies that any resource o in D_{aname} is also in $D_{aname} \cap R_{hasActor}$. Putting the two arguments together, we found a property, `:hasActor`, that maximizes the chances that s is in its domain and o is in its range. That is, the choice of `:hasActor` maximizes the product (see Proposition 1d):

$$C(D_{fname} \cap D_{genre}, D_{hasActor}) \times C(D_{aname}, R_{hasActor})$$

• *Tree expansion.* Tree expansion is a relaxation of edge addition in the sense that it does not require that both the domain and range of an object property be similar to other sets already under consideration; it suffices to have just the domain or just the range. The repeated application of tree expansion, combined with edge addition, tries to find longer paths to connect entities in the sets of already identified. For example, the range of the object property `:loc` (*location*) is similar to the domain of the property `:lname` (*location name*), so it might be profitable to combine the two properties into a path of length 2. Once again, this is detected using set similarity, estimated using the pre-computed KMV–synopses, and the last step of the process synthesizes a SPARQL query that has a join clause that takes this situation into account.

In terms of a query forest, tree expansion adds a node and an edge to create a new forest that might be transformed in a later step by node fusion or edge addition. The choice of which edge to include is similar to edge addition,

except that one of the nodes is new, that is, added together with the edge. Tree expansion is used when node fusion and edge addition cannot be applied. This point is better explained at the end of Section 4.3, when trees expansion is covered in detail.

For example, it follows from Figure 5 that D_{lname} is disjoint from the domain or range of any property, except for the range of `:loc`. Hence, we have:

$$J(D_{fname}, D_{genre}, D_{hasActor}, D_{lname}) = 0$$
$$J(D_{aname}, R_{hasAuthor}, D_{lname}) = 0$$
$$C(D_{fname} \cap D_{genre} \cap D_{hasActor}, D_{lname}) = 0$$
$$C(D_{aname} \cap R_{hasAuthor}, D_{lname}) = 0$$

which implies that we cannot use node fusion or edge addition, as in the previous steps, to create a single tree out of the forest in Figure 8. Tree expansion, therefore, adds a new node, labeled with D_{loc}, and a new edge, labeled with "*:loc*", creating the forest shown in Figure 9.

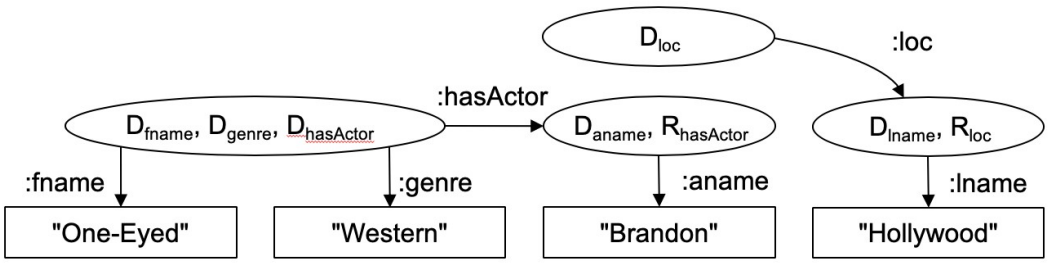


Figure 9: The query forest after expanding the rightmost tree by adding an edge labeled with "*:loc*"

• *Edge addition.* To conclude the construction of the query forest for the running example, it follows from Figure 5 that

$$C(D_{fname} \cap D_{genre} \cap D_{hasActor}, R_{produces}) = 1$$
$$C(D_{loc}, D_{produces}) = 1$$

As before, this suggests that we can add an edge, labeled with "*:produces*", to combine the two trees in Figure 9, creating the final query tree, shown in Figure 10.

• *SPARQL query compilation.* The last step is to generate a SPARQL query Φ. In this explanatory example, the final forest consists of a single tree. When this is not the case, the tree with the largest number of keyword matches is kept.

In terms of a query forest, for each node of the tree, the `WHERE` clause of Φ has a variable and, for each edge, a join clause or a `FILTER` clause. The

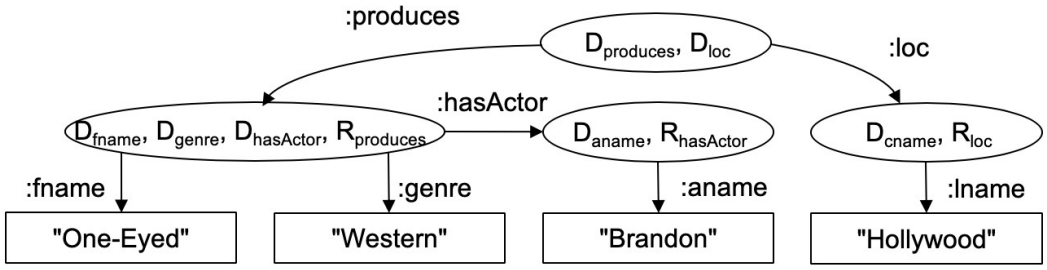


Figure 10: The final query tree after adding an edge labeled with "*:produces*"

construction of the forest and the generation of Φ also guarantees that any answer A that Φ returns is such that G_A is a Steiner tree.

The `WHERE` clause of the SPARQL query generated from the final query tree in Figure 10 is:

```
?v1     :fname      ?v2  FILTER ( match(?v2, "One-Eyed") ) .
?v1     :genre      ?v3  FILTER ( match(?v3, "Western") ) .
?v4     :aname      ?v5  FILTER ( match(?v5, "Brandon") ) .
?v6     :lname      ?v7  FILTER ( match(?v7, "Hollywood") ) .
?v1     :hasActor   ?v4 .
?v8     :loc        ?v6 .
?v8     :produces   ?v1
```

Observing Figure 10, the leftmost oval node, a_1, corresponds to the variable "`?v1`"; the two edges from a_1 to rectangular nodes correspond to two `FILTER` clauses (Lines 1 and 2), and the two other edges incident to a_1 correspond to two join clauses (Lines 5 and 7) involving "`?v1`". This implies that "`?v1`" will bind to a resource s that must have a value of the property `:fname` that matches the keyword "*One-Eyed*" (Line 1) and a value of the property `:genre` that matches the keyword "*Western*" (Line 2); also, variable "`?v4`" must bind to a resource o such that there is a triple (s,`:hasActor`,o) in T (Line 5), and likewise for the other join clause (Line 7).

For simplicity, Lines 1-4 adopt a non-standard user-defined predicate match that expresses the matches between keywords and literals, and which an implementation will map to a specific technology. For example, in Apache Jena for RDF[7] with Lucene[8], Line 1 would be rewritten as

```
(?v1 ?score ?v2) <http://jena.apache.org/text#query>
                                                ("(One-Eyed)").
 ?v1 :fname ?v2
```

that matches the values of variable "`?v2`" with "*One-Eyed*".

When executed over the graph in Figure 5, such query will return the following triples:

[7]https://jena.apache.org
[8]https://lucene.apache.org

1. (r_2, `:fname`, "One-Eyed Jack")
2. (r_2, `:genre`, "Western")
3. (r_3, `:aname`, "Marlon Brandon")
4. (r_5, `:lname`, "Hollywood")
5. (r_2, `:hasActor`, r_3)
6. (r_4, `:loc`, r_5)
7. (r_4, `:produces`, r_2)

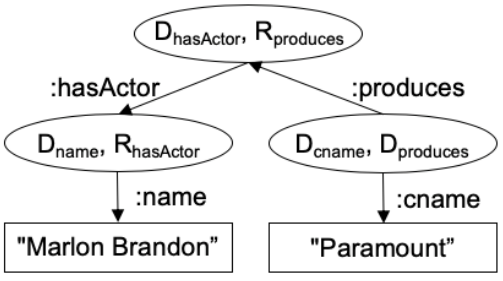
which is an answer for the keyword-based query K since these triples match all keywords in K and induce a Steiner tree of the RDF graph in Figure 5 that covers all matching nodes.

Finally, if the query returns more than one answer, they are ranked, as briefly discussed in Section 4.4. For more details, we refer the readers to [41]. □

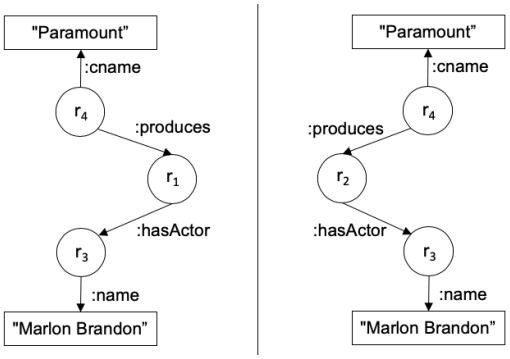
The key steps therefore are how to select trees to combine by node fusion or by edge addition and how to expand trees by adding new nodes and edges. The previous example illustrates these operations, but it leaves open an important point –how to choose trees, nodes, and edges– which is detailed in Section 4.3.

Example 3 Based on the Example 2, consider the keyword-based query K'={$Brandon$, $Paramount$}. We have the following matches between the keywords in K' and triples in T:

$Brandon$	with	(r_3	`:aname`	"Marlon Brandon")
$Paramount$	with	(r_4	`:cname`	"Paramount")



11(a) The final query graph for K'



11(b) Two answers of the SPARQL query synthesized from the query graph in 11(a)

Figure 11: Answers for the keyword-based query K'={$Brandon$, $Paramount$}

Initially, the query forest has two trees, one for each match. However, these trees cannot be directly combined since their nodes are dissimilar. The

final query graph, shown in Figure 1111(a), is obtained by adding two edges, labeled with `:hasActor` and `:produces`, and a new node, labeled with $D_{hasActor}$ and $R_{produces}$. These additions are required to avoid that the SPARQL query returns answers which are not connected graphs. Indeed, the resulting query has two answers, shown in Figure 1111(b), each of which features a node (r_1 or r_2) that connects the match nodes.

4.2 Query Graph Notion

This section defines the notion of query graph. The discussion in what follows ignores matches between keywords and class and property labels, which are discussed in Section 4.4.

Recall, let T be an RDF dataset, $\mathcal{L}$ be the set of all literals and K=$\{k_1, ..., k_n\}$ be a keyword-based query.

Definição 4.1 A ***query graph*** for K over T is a node and edge-labeled graph $Q = (N, E, \nu, \lambda)$, where ν labels nodes and λ labels edges, such that:

- N is a set of *match nodes* or *join nodes*.
- E is a set of edges such that all match nodes have in-degree 1 and out-degree 0.
- λ is such that each edge (a, b) in E is labeled with a property p that occurs in T. For simplicity, we denote an edge (a, b) in E, labeled with p, as a triple (a, p, b).
- ν is such that
  - a *match node* in N is labeled with one keyword in K;
  - a *join node* a in N is labeled with $\nu(a) = \{D_1, ..., D_m, R_1, ..., R_n\}$, where: $(a, p_1, b_1)...(a, p_m, b_m)$ are all the edges from a; $(c_1, q_1, a)...(c_n, q_n, a)$ are all the edges into a; D_i is the domain of p_i, for $1 \leqslant i \leqslant m$; and R_j is the range of q_j, for $1 \leqslant j \leqslant n$.

□

Example 2, in Section 4.1, illustrated the concept of query graph. Note that the label of a join node is determined by the domains and ranges of the properties that label the edges incident to the node. Also note that Definition 4.1 assumes that each match node is labeled with a single keyword, an assumption adopted just to reduce the complexity of the notation and the definitions that follow, and to facilitate understanding the proposed algorithm. However, we note that the implementation considers match nodes labeled with multiple keywords.

Definição 4.2 A ***query tree*** (or ***forest***) for K over T is a query graph which is a tree (or a forest). □

Definição 4.3 Let $Q = (N, E, \nu, \lambda)$ be a query graph for K over T. A SPARQL group graph pattern P_Q ***is induced by*** Q iff

- for each node a in N, there is a variable $\texttt{?v}_a$; and
- for each edge (a, q, b) of Q, there is a triple pattern in P_Q of the form "$\texttt{?v}_a$ q $\texttt{?v}_b$", if a and b are join nodes, or of the form "$\texttt{?v}_a$ q $\texttt{?v}_b$ `FILTER (match(?v`$_b$`, "M"))`", if b is a match node labeled with "M". □

As explained in Example 2, for simplicity, the second condition includes a user-defined predicate match that expresses the matches between keywords and literals. Note that, apart from the order of the triple patterns, P_Q is unique. Also note that the same variable `?va` will be used in the patterns corresponding to the edges $(a, p_1, b_1)...(a, p_m, b_m)$ from a and to the edges $(c_1, q_1, a)...(c_n, q_n, a)$ into a.

We leave open the definition of the `TARGET` clause induced by a query graph Q. It could return all, or a subset of the variables bound in the query pattern match (as in the SPARQL `SELECT` query form), or it could return an RDF graph constructed by substituting variables in a set of triple templates (as in the SPARQL `CONSTRUCT` query form). The first form induces tabular answers, which users preferred in an earlier implementation [24]. The following notion of an answer for a query graph Q factors out this discussion since it depends only on the group graph pattern P_Q induced by Q.

Definição 4.4 Let $Q = (N, E, \nu, \lambda)$ be a query graph for K over T. An ***answer for*** Q is a minimal set A_Q of triples in T that satisfies the SPARQL group graph pattern P_Q induced by Q. □

Note that A_Q induces a subgraph of the RDF graph of T and that Q may have more than one answer.

4.3 A Greedy Algorithm to Translate Keyword-based Queries to SPARQL

Let T be an RDF dataset and $K = \{k_1, ..., k_n\}$ be a keyword query. This section details the proposed algorithm to compile a SPARQL query for K over T whose answers are answers for K.

Definition 4.5 formalizes the *node fusion*, *edge addition*, and *tree expansion* operations. To avoid an awkward notation, in what follows, tree expansion is broken into two operations that depend on the direction of the edge added.

Definição 4.5 Let $Q = (N, E, \nu, \lambda)$ be a query graph for K over T. Let a_i be a node labeled with $A_i = \{A_{i,1}, ..., A_{i,n_i}\}$, for $i = 1, 2$, and p be a property with domain D_p and range R_p. Assume that a_1 and a_2 belong to different trees t_1 and t_2.

a) The $fusion$ of nodes a_1 and a_2 replaces a_1 and a_2 by a new join node c, labeled with $\{A_{1,1}, ..., A_{1,n_1}, A_{2,1}, ..., A_{2,n_2}\}$.
b) The *addition of a join edge, labeled with p, from* a_1 *to* a_2 relabels a_1 with $\{A_{1,1}, ..., A_{1,n_1}, D_p\}$ and a_2 with $\{A_{2,1}, ..., A_{2,n_2}, R_p\}$, and adds the join edge (a_1, p, a_2) to the query graph.
c) The *expansion* of tree t_1 by the addition of a join edge, labeled with p, from a_1 relabels a_1 with $\{A_{1,1}, ..., A_{1,n_1}, D_p\}$, adds a new node c, labeled with $\{R_p\}$, and adds the join edge (a_1, p, c) to the query graph.
d) The *expansion* of tree t_2 by the addition of a join edge, labeled with p, into a_2 relabels a_2 with $\{A_{2,1}, ..., A_{2,n_2}, R_p\}$, adds a new node c, labeled with $\{D_p\}$, and adds the join edge (c, p, a_2) to the query graph. □

Algorithm 1 summarizes the basic steps of the strategy, illustrated in Section 4.1. In the algorithm, a *property match* for K over T is a pair (p, k_i) such that there is at least one triple in T of the form (s, p, o) such that o matches $k_i \in K$. Step 1 is described in Section 4.4. Step 2 is simple and was already illustrated in Section 4.1. In what follows, we cover in detail Step 3, the core of Algorithm 1, and conclude with Step 4.

Step 3 selects operations based on scores defined as follows.

Definição 4.6 Let $Q = (N, E, \nu, \lambda)$ be a query graph for K over T. Let a_i be a node labeled with $\{A_{i,1}, ..., A_{i,n_i}\}$, for $i = 1, 2$, and p be a property with domain D_p and range R_p.

a) *Node Fusion Score:* assesses when to combine a_1 and a_2 into a single node:

$$node\_fusion\_score(a_1, a_2) \quad = \quad J(A_{1,1}, ..., A_{1,n_1}, A_{2,1}, ..., A_{2,n_2})$$

Algorithm 1: TRANSLATEKEYWORDQUERY

Input: T – an RDF dataset
K- a keyword-based query K over T
Output: Φ – a query for K over T that outputs answers for K
Step 1: Match the keywords in K with literals in T, creating a set $\boldsymbol{S}$ of property matches (p, k_i) such that p is a property that occurs in T, and $k_i \in K$.
Step 2: Use the set $\boldsymbol{S}$ of matches found in Step 1 to construct an *initial query forest* as follows: for each match (p, k_i) in $\boldsymbol{S}$, where D is the domain of p, the forest has a join node a, labeled with $\{D\}$, a match node b, labeled with "k_i", and an edge (a, p, b).
Step 3: Reduce the number of trees of the query forest by using *node fusion*, *edge addition*, and *tree expansion*.
Step 4: Select the tree with the largest number of keyword matches, construct a SPARQL query Φ from the selected tree, whose WHERE clause is as in Def. 4.3, and output Φ.

b) *Edge Addition Score:* assesses when to add a join edge, labeled with p, outgoing from a_1 and incoming into a_2:

$$edge\_addition\_score(a_1, p, a_2) = C(A_{1,1} \cap ... \cap A_{1,n_1}, D_p) \times C(A_{2,1} \cap ... \cap A_{2,n_2}, R_p)$$

c) *Outgoing Tree Expansion Score:* assesses when to add a join edge, labeled with p, outgoing from a_1:

$$outgoing\_tree\_expansion\_score(a_1, p) = C(A_{1,1} \cap ... \cap A_{1,n_1}, D_p)$$

d) *Incoming Tree Expansion Score:* assesses when to add a join edge, labeled with p, incoming into a_2:

$$ingoing\_tree\_expansion\_score(a_2, p) = C(A_{2,1} \cap ... \cap A_{2,n_2}, R_p).$$

□

All these scores can then be estimated using KMV–synopses for such sets –and this was the reason for adopting KMV–synopses.

In more detail, the node fusion operation uses the node fusion score to decide when to combine two nodes, a_1 and a_2, based on how similar all sets that label the nodes are; note that the node fusion score depends on the n-way Jaccard similarity measure, since all sets should be considered equally relevant.

The edge addition operation uses the edge addition score to decide when to add an edge between two nodes, a_1 and a_2, labeled with a property p, based

on how the intersection $A_{1,1} \cap ... \cap A_{1,n_1}$ of all sets that label a_1 are similar to the domain of p and, simultaneously, how the intersection $A_{2,1} \cap ... \cap A_{2,n_2}$ of all sets that label a_2 are similar to the range of p; edge addition depends on set containment, since edge addition is similar to a query that locates sets similar to $A_{1,1} \cap ... \cap A_{1,n_1}$, and likewise for $A_{2,1} \cap ... \cap A_{2,n_2}$.

Finally, the tree expansion operation uses the outgoing/incoming tree expansion scores much in the same way that the edge addition operation uses the edge addition score; the difference lies in that the added edge, labeled with property p, is such that the domain of p is similar to the intersection $A_{1,1} \cap ... \cap A_{1,n_1}$ of all sets that label a_1, in the case of the outgoing tree expansion score, and likewise the range of p is similar to the intersection $A_{2,1} \cap ... \cap A_{2,n_2}$ of all sets that label a_2, in the case of the outgoing tree expansion score.

More precisely, the following proposition lists properties of such scores ($Pr[S]$ denotes the probability of S).

Proposition 1: Let T be an RDF dataset. Let $A_{i,1}, ..., A_{i,n_i}$, for $i = 1, 2$, and B_1, B_2 be sets of IRIs, and p be a property with domain D_p and range R_p.

a) Randomly draw an element s from $A_{1,1} \cup ... \cup A_{1,n_1} \cup A_{2,1} \cup ... \cup A_{2,n_2}$. Then, the probability that the element s is in $A_{1,1} \cap ... \cap A_{1,n_1} \cap A_{2,1} \cap ... \cap A_{2,n_2}$ is given by $J(A_{1,1}, ..., A_{1,n_1}, A_{2,1}, ..., A_{2,n_2})$.

b) Randomly draw s from $B_1 \cup D_p$. Then, the probability that s is in $B_1 \cap D_p$, knowing that s is in B_1, is given by $C(B_1, D_p)$.

c) Randomly draw o from $B_2 \cup R_p$. Then, the probability that o is in $B_2 \cap R_p$, knowing that o is in B_2, is given by $C(B_2, R_p)$.

d) Randomly draw s from $B_1 \cup D_p$ and o from $B_2 \cup R_p$. Then, the probability that s is in $B_1 \cap D_p$ and o is in $B_2 \cap R_p$, knowing that s is in B_1 and o is in B_2, is given by $C(B_1, D_p) \times C(B_2, R_p)$.

Proof.

a) Randomly draw s from $A_{1,1} \cup ... \cup A_{1,n_1} \cup A_{2,1} \cup ... \cup A_{2,n_2}$. Then, by definition of the Jaccard similarity, we have

$$Pr\left[s \in \bigcap_{i=1,2}^{j=1,...,n_1} A_{i,j}\right] = \frac{|\bigcap_{i=1,2}^{j=1,...,n_1} A_{i,j}|}{|\bigcup_{i=1,2}^{j=1,...,n_1} A_{i,j}|} = J(A_{1,1}, ..., A_{1,n_1}, A_{2,1}, ..., A_{2,n_2})$$

b) Randomly draw s from $B_1 \cup D_p$. The probability σ we want to compute is:

$$\sigma = Pr\left[s \in B_1 \cap D_p | s \in B_1\right]$$

By definition of conditional probability, we have

$$\sigma = \frac{Pr\left[s \in B_1 \cap D_p\right]}{Pr\left[s \in B_1\right]}$$

Since s is drawn from $B_1 \cup D_p$, we have

$$Pr\left[s \in B_1 \cap D_p\right] = \frac{|B_1 \cap D_p|}{|B_1 \cup D_p|}$$

$$Pr\left[s \in B_1\right] = \frac{|B_1|}{|B_1 \cup D_p|}$$

From the above equalities and by definition of set containment, we have

$$\sigma = \frac{|B_1 \cap D_p|}{|B_1 \cup D_p|} \times \frac{|B_1 \cup D_p|}{|B_1|} = \frac{|B_1 \cap D_p|}{|B_1|} = C(B_1, D_p)$$

c) Follows likewise.

d) The probability ρ we want to compute is

$$\rho = Pr\left[s \in B_1 \cap D_p, o \in B_2 \cap R_p \mid s \in B_1, o \in B_2\right]$$

By definition of conditional probability, we have

$$\rho = \frac{Pr\left[s \in B_1 \cap D_p, o \in B_2 \cap D_p\right]}{Pr\left[s \in B_1, o \in B_2\right]}$$

By the independence of the drawings, we have

$$\rho = \frac{Pr\left[s \in B_1 \cap D_p\right] \times Pr\left[o \in B_2 \cap D_p\right]}{Pr\left[s \in B_1\right] \times Pr\left[o \in B_2\right]}$$

Then, as in (b), we immediately have that

$$\rho = C(B_1, D_p) \times C(B_2, R_p)$$ □

Returning to Step 3 of Algorithm 1, its implementation is limited by the following result, denominated as *MQF Problem*.

Definição 4.7 ***MQF Problem****:* Given a query forest $Q = (N, E, \nu, \lambda)$ for K over T, and minimal bounds for the scores, find a minimal forest obtained by repeatedly applying the *node fusion*, *edge addition* and *tree expansion*

operations to Q, provided that the scores of these operations are above minimal bounds.

Proposition 2: The MQF Problem is NP-complete.

Proof Sketch. By a reduction from the minimal Steiner tree problem, as demonstrated in [24].

In the face of Proposition 2, the rest of this section introduces a heuristic to implement Step 3, based on the scores introduced in Definition 4.6, estimated using KMV–synopses.

Algorithm 2: ReduceQuery

Input: T – an RDF dataset
$p_1, ..., p_n$ – the list of properties that occur in T
Q – an initial query forest
δ – a minimum threshold
η – the max number of reduction cycles allowed
Output: Q – a modified query forest, possibly with fewer trees

```
1  begin
2    count = 0;
3    while Q has more than 1 tree and count ≤ μ do
4    begin
5      CombineTreesByNode(T, Q, δ; Q);
6      if Q has a single tree then return Q;
7        CombineTreesUsingEdges(T, (p1, ..., pn), Q, η; Q);
8      if Q has a single tree then return Q;
9        ExpandTree(T, (p1, ..., pn), Q, η; Q);
10     count = count + 1;
11   end
12   CleanQuery(Q);
13   return Q;
14 end
```

Algorithm 2 – ReduceQuery implements Step 3 of Algorithm 1 by combining distinct trees by node fusion (Line 5) or edge addition (Line 7), and by tree expansion (Line 9). Note that a tree expansion operation (Line 9) may add a new edge and a new node that may end up not being used in later cycles to combine trees. This can be detected, when the loop (in Lines 3-11) finishes, by checking if there is a node, with only one incident edge, which is not a matching node. A cleaning operation (Line 12) will then eliminate such nodes and their incident edges. Finally, the number of cycles is limited to η to avoid adding too many new edges, which might lead to less meaningful queries. The constant η was empirically determined during the experiments described in Sections 5.1 and 5.2. Indirectly, η places an upper bound on the length of the paths between two nodes in the query forest. This is justified since, as

argued in [43], long paths may express unusual relationships, which might be misinterpreted by users.

The four procedures used in Algorithm 2 (lines 5, 7, 9, and 12) are described in Algorithms 3, 4, 5, and 6, respectively.

Algorithm 3: CombineTreesByNodeFusion

```
   Input: T – an RDF dataset
          Q – a query forest
          δ – a minimum threshold
   Output: Q – a reduced query forest
 1 begin
 2   create a list L of pairs of join nodes, each from a different tree,
 3         with node fusion scores above the threshold δ;
 4   order L in decreasing order of node fusion score;
 5   while L is not empty do
 6   begin
 7     apply node fusion to (a1, a2), creating a new node a3,
 8           and modifying the forest Q accordingly;
 9       remove from L any pair involving a1 and a2
10           and any pair of nodes that are now in the same tree;
11       add to L all new pairs involving the new node a3,
12           with node fusion scores above the threshold δ;
13       reorder L in decreasing order of node fusion score;
14     end
15     return Q;
16 end
```

Algorithm 3 – CombineTreesByNodeFusion implements the node fusion operation and uses the node fusion score to decide when to combine two nodes. It selects two nodes to apply node fusion in decreasing order of node fusion scores (Lines 4 and 13). Lines 11-13 are necessary to accommodate the new node obtained by node fusion. Finally, the minimum score for the node fusion score (Lines 3 and 12) tries to reduce the number of pairs added to the list L and, consequently, the number of cycles of the algorithm.

Algorithm 4 — CombineTreesByEdgeAddition implements the edge fusion operation and uses the edge addition score to decide when to add an edge between two nodes. Line 5 avoids considering the combination of two trees that have a score which is too low, which would possibly lead to a query with too few answers, or no answer at all. Line 14 is necessary because, later on, the new tree $C_{1,2}$ might in turn be combined with other trees.

Algorithm 5 — ExpandTree implements the tree expansion operation and uses the outgoing/incoming tree expansion scores to decide when to add an outgoing/incoming edge to a node.

Algorithm 4: CombineTreesByEdgeAddition

Input: T – an RDF dataset
$p_1, ..., p_n$ – the list of properties that occur in T
Q – a query forest
δ – a minimum threshold
Output: Q – a reduced query forest

```
1  begin
2    mark all trees of Q as unprocessed;
3    while Q has more than one tree and
4          there is a pair (C1, C2) of unprocessed trees of Q
5          such as tree_combination_score(C1, C2) ≥ δ do
6    begin
7      select the pair (C1, C2) of unprocessed trees of Q
8        with the highest tree_edge_combination_score(C1, C2);
9      select ai ∈ Ci, ak ∈ Ck, for 1 ≤ i ≠ k ≤ 2,
10       and a property pj such that (ai,pj,ak)
11         has the highest edge_combination_score(ai,pj,ak);
12     let ai and ak be labeled with Ai and Ak, respectively;
13     add (ai, pj, ak) to Q, combining C1 and C2 into a single tree
     C1,2;
14     relabel ai with Ai ∪ {Dj} and ak with Ak ∪ {Rj},
15       where Dj and Rj are the domain and range of pj,
     respectively;
16     mark C1,2 as unprocessed;
17   end
18   return Q;
19 end
```

Algorithm 6 — CleanQuery receives a query forest and eliminates unnecessary edges so that all terminal nodes of the modified query forest are match nodes.

By induction on the number of node fusions, edge additions and tree expansions applied, and by Definitions 4.3 and 4.4, we can prove the correctness of the Algorithm 1.

Proposition 3: Let $Q = (N, E, \nu, \lambda)$ be the tree selected in Step 4 of Algorithm 1. Let A be an answer for Q over T. Then, A is an answer for K over T. □

As for the overall complexity, Algorithm 2 executes at most η cycles. In each cycle, node fusions are tried, then edge additions and, if the forest has not been reduced to a single tree, a tree expansion. Let $|K|$ be the number of keywords of the query. Let N_P be the number of properties that occur in the RDF graph (i.e., the number of IRIs that denote properties). In the worst case, one tree expansion is executed per cycle, which adds a new join node. The initial number of join nodes is at most $|K|$. Hence, in the i^{th} cycle (starting

Algorithm 5: ExpandTree

Input: T – an RDF dataset
$p_1, ..., p_n$ – the list of properties that occur in T
Q – a query forest
δ – a minimum threshold
Output: Q – a reduced query forest

```
1  begin
2    mark all trees of Q as unprocessed;
3    while Q has more than one tree and
4          there is an unprocessed trees C of Q do
5    begin
6      select the join node a in C and the predicate p_j in T
7        with the highest scores:
8          out_edge_score(a, p_j) or in_edge_score(a, p_j);
9      add a new node b to C;
10     add the join edge (a, p_j, b) (or (b, p_j, a)) to C;
11     let a be labeled with A;
12     let D_j and R_j be domain and range of p_j, respectively;
13     relabel a with A ∪ {D_j} and b with {R_j}
14       (or a with A ∪ {R_j} and b with {D_j});
15     mark C as unprocessed;
16   end
17   return Q;
18 end
```

Algorithm 6: CleanQuery

Input: Q – a query forest, possibly with join nodes as terminals
Output: Q – a modified query forest whose terminals are match nodes

```
1  begin
2    while there is a terminal node a which is not a match node do
3    begin
4      /* since Q is a forest and a is terminal,
5         there is just one edge incident to a */
6      delete the edge incident to a;
7      delete the join node a;
8    end
9    return Q;
10 end
```

with $i = 0$), there are at most $(|K| + i)$ join nodes. Then, there are at most $(|K| + i)^2$ possible node fusions, $(|K| + i)^2 \times 2 \cdot N_P$ possible edge additions (we have to multiply by 2 since we also have to try the inverse of each property), and $(|K| + i) \times 2 \cdot N_P$ possible tree expansions. Hence, since η is a constant, in the worst case, the time complexity of Algorithms 2 and 3 is $O(|K|^2 \times N_P)$.

This concludes the discussion of Step 3 of Algorithm 1.

We conclude this section by returning to Step 4 of Algorithm 1. It suffices to recall that this step selects the tree T with the largest number of keyword matches and constructs the final SPARQL query Φ from T. Definition 3 explained how to construct the `WHERE` clause of Φ from T, as already illustrated towards the end of Example 2 (in Section 4.1). The discussion after Definition 4.3 indicated how to synthesize the target clause of Φ. In the SPARQL `SELECT` query format, Φ could return all, or a subset of the variables bound in the `WHERE` clause of Φ.

Algorithm 7 -– CompileQuery summarizes these observations, for the `SELECT` query format with all variables bound in the `WHERE` clause.

Algorithm 7: CompileQuery

Input: $Q = (N, E, \nu, \lambda)$ – a query forest
Output: Φ – a SPARQL query

```
1  begin
2    select the tree T = (N, E, ν, λ) in Q with
3          the largest number of keyword matches;
4      construct the WHERE clause of Φ as follows:
5        for each node a in N:
6          create a variable ?v_a;
7        for each edge (a, q, b) of T:
8          if a and b are join nodes:
9            create a triple pattern in P_Q of the form "?v_a q ?v_b";
10         else if b is a match node labeled with "M":
11           create a triple pattern of the form
12           "?v_a q ?v_b FILTER (match(?v_b, "M"))";
13   build the TARGET of Φ with all variables used in the WHERE clause;
14   return Φ;
15 end
```


4.4 Additional Remarks on the Translation Approach

This section deals with the extensions built into the proposed algorithm described in the previous section.

4.4.1 Treatment of Class and Property Labels

First, observe that a keyword may match the label of a class or property, which alters the interpretation of the keyword-based query. For example, if `Actor` is declared as a class with label "*actor*", then the keyword-based query K={*actor*, *Washington*} is interpreted as requesting instances of the class `Actor` that have properties matching "*Washington*".

Assume that a keyword matches the label of class c, declared in the RDF dataset T (labels are identified when T is scanned in the KMV–synopses computation). Briefly, this is captured by modifying the definition of query graph to accommodate classes in the query graphs and the SPARQL group graph pattern induced by a query graph.

The modifications to account for keywords that match property labels are entirely similar. The use of other terms of the RDF Schema vocabulary is outside the scope of this work and therefore will not be discussed.

4.4.2 Use of Ranking

We now briefly discuss two questions: (1) how to define ranking measures specifically for RDF graphs?; (2) how to use these measures to help compute and rank answers of keyword queries over RDF graphs?

To address the first problem, [41] proposed a family of importance measures for RDF graphs, collectively called $InfoRank$, that combines three intuitions: (I) "*important things have lots of information about them*"; (II) "*important things are surrounded by other important things*"; (III) "*few important relations (e.g. friends) are better than many unimportant relations (e.g. acquaintances)*". *InfoRank* requires neither the manual assignment of weights to object properties nor a training dataset to use as input to a learning algorithm.

Let T be a set of RDF triples. Recall from Section 2.1 that it is possible to identify the set C of classes observed in T, the set P of object properties observed in T, the set L of literals observed in T, and the set R of (class) instances observed in T.

The *informativeness* of an instance $r \in R$, denoted $IW(r)$, is defined as the number of triples of the form $(r, p, v) \in T$, where $v \in L$, that is, the number of property values that describe instance r. Based on instance informativeness, we say that "*important classes usually have informative instances*" and "*important properties are usually those connecting informative instances*". More precisely, the *InfoRank* of a class $c \in C$, denoted $IR(c)$, is defined as the maximum value of $IW(r)$ of all instances of class c. Likewise, the *InfoRank* of an object property $p \in P$, denoted $IR(p)$, is defined as the maximum value of $IW(r) + IW(s)$ of all triples of the form $(r, p, s) \in T$.

Note that we used only *Intuition I* to rank classes and object properties. However, we propose a combination of the three intuitions to rank class instances.

Let $r, s \in R$ and $p \in P$. Assume that $(r, p, s) \in T$ or $(s, p, r) \in T$, that is, ignore the direction of the object property p. The normalized weight of (r, p),

denoted $W(r, p)$, is defined as:

$$W(r,p) = \frac{IR(p)}{\sum_{q \in P \text{ and } ((r,q,t) \in T \text{ or } (t,q,r) \in T)} IR(q)} \quad (9)$$

Then, the weighted *PageRank* score of an instance r, denoted $PR_W(r, i)$, is recursively defined as:

$$PR_W(r, 0) = 1/N \quad (10)$$

$$PR_W(r,i) = \frac{1-\alpha}{N} + \alpha \sum_{(r,p,s) \in T \text{ or } (s,p,r) \in T} PR_W(s, i-1) * W(r,p) \quad (11)$$

where N is the total number of nodes in T and α is a dumping factor (usually set to 0.85).

The *InfoRank* score of an instance r, denoted $IR(r)$, is the *PageRank* score of r after a fixed number x of iterations, $PRW(r, x)$, weighted by the informativeness of r, $IW(r)$:

$$IR(r) = PR_W(r, x) * IW(r) \quad (12)$$

We conclude with a brief discussion about how to use *InfoRank* in the context of the process described in Section 4.3 to compute answers to RDF keyword queries. Recall that Step 1 of Algorithm 1 matches keywords with property values, and also with class (or object property) labels or descriptions. First, class and property labels or descriptions matches have priority over property value matches. Whenever a keyword matches more than one class (or object property) label or description, Step 1 of Algorithm 1 ranks all such classes (and object properties) by descending order of a linear combination of the match score values with the pre-computed InfoRank score values for the classes (or object properties) and considers only the topmost class (or object property). Likewise, Step 1 of Algorithm 1 ranks the property value matches in decreasing order of their combined scores and considers only the topmost match.

Finally, Step 4 of Algorithm 1 ranks the answers of a query using again a linear combination of match score values with pre-computed *InfoRank* score values. This is implemented by modifying the final SPARQL query to also retrieve the pre-computed *InfoRank* scores for the instances observed in an answer and to include an order by clause that ranks the answers accordingly.

4.4.3 Beyond Synopses and Ranking

There are special cases of keyword queries where synopses and ranking measures do not need to be used in the keyword query translation process into the SPARQL query. The first one consists of queries with a single keyword

that matches a class label. The second case includes queries with two keywords, where a keyword matches the label of a resource r, such that the triple (r,`rdf:type`,c) is in T, and the other keyword matches the label of an object property p, where the domain and range of p is c. Note that the triple (r,`rdf:type`,c) is in T implies that c is an observed class in T.

We now use two examples to illustrate these cases.

Let us consider that the keyword-based query system is running on top of DBpedia.

I. Suppose that the system receives the keyword query K={*World Heritage Site*} as input. Hence, the matching process finds that this keyword matches the label of class `dbo:WorldHeritageSite`, since the triple (`dbo:WorldHeritageSite`, `rdfs:label`, `"World Heritage Site"`) is in DBpedia. Next, the system directly synthesizes a SPARQL query whose `WHERE` clause corresponds to the triple pattern (?r,`rdf:type`,`dbo:WorldHeritageSite`), where the variable ?r binds the class resources.

II. Suppose now that the system receives the keyword query K={*goofy,creator*}. So, the matching process finds that the keyword "*goofy*" matches the label of the resource `dbr:Goofy`[9], and the keyword "*creator*" matches the label of the object property `dbo:creator`[10]. Next, the system detects that `dbr:Goofy` is a resource of class `dbo:Person`, and `dbo:creator` has resources of `dbo:Person` both in the domain and range. Thus, the system is unable to resolve this ambiguity, i.e., it cannot decide if `dbr:Goofy` belongs to the domain or to the range of `dbo:creator`. To deal with this issue, the system compiles a `WHERE` clause with the following triple pattern:

```
{ ?r rdfs:label "Goofy"@en .
  ?x dbo:creator ?y
  FILTER (sameTerm(?r, ?x) || sameTerm (?r, ?y)
}
```

Finally, the system runs the compiled SPARQL query and ranks the answers.

[9]http://dbpedia.org/page/Goofy
[10]http://dbpedia.org/ontology/creator

5
KMV–Synopses RDF Keyword Search System Evaluation

This chapter is composed by three parts to evaluate the schema-less approach proposed in this thesis by comparing it with state-of-the-art systems, adopted as the baseline. Firstly, Section 5.1 describes the set of experiments that compares the schema-less approach proposed in this thesis with a state-of-the-art schema-based RDF keyword search tool. Section 5.2 describes the set of experiments that compares the schema-less approach proposed in this thesis with the state-of-the-art TSA+BM25 and TSA+VDP keyword search systems over RDF datasets based on the "virtual documents" approach, adopted as baselines. Finally, Section 5.3 introduces an alternative measure for graph relevance.

5.1
Comparison with a Schema-based RDF Keyword Search Tool

This section compares the schema-less approach proposed in this thesis with a state-of-the-art schema-based RDF keyword search tool. Section 5.1.1 describes the benchmark adopted in this comparison. Section 5.1.2 describes the environment setup used to run this experiment. Finally, Section 5.1.3 shows the effectiveness and efficiency of the developed RDF keyword search approach based on KMV–synopses.

5.1.1
Benchmark Adopted

The benchmark adopted in this experiment was inspired by Coffman's benchmark [16], created to evaluate keyword search tools over relational databases. Coffman's benchmark is based on data and the relational schemes for IMDb, Mondial, and Wikipedia; each dataset has 50 keyword-based queries and their expected answers.

However, the expected answers in Coffman's benchmark do not always cover all possibilities and are somewhat arbitrary. For example, the keyword-based query niger over Mondial had as the expected answer only the instance of the class River labeled as "*Niger*". However, the instances of the classes `Country` and `Province` labeled as "*Niger*" should also be considered valid

answers. In fact, the user can refine the query as {*River, niger*}, if s/he is indeed interested in the *Niger River*. Furthermore, Coffman's benchmark group queries by topics, so that all queries within the same topic are quite similar and redundant with respect to testing the capability of the keyword search tools.

Hence, the adopted benchmark differs from Coffman's in three aspects: (1) it uses triplified versions of Mondial[11] and IMDb[12]; (2) it includes only keyword-based queries that have answers that explore the structure of the RDF graph; (3) for each keyword-based query, it contains a ranked list of answers, created with the help of the graph-based algorithm described in [42].

In more detail, Table 3 summarizes the characteristics of the triplified versions of the Mondial and IMDb databases included in the benchmark. We note that the Mondial RDF graph is much more complex than that of IMDb. However, the size of IMDb is significantly larger than the size of Mondial, in terms of the number of triples. For both datasets, we included the InfoRank property values computed in [41]. The schema and data of Mondial dataset are available at `https://www.dbis.informatik.uni-goettingen.de/Mondial`. Regarding IMDb, the RDF Schema and data are available at the QUIRA Official Page[13].

Table 3: Statistics – Mondial and IMDb Datasets

Characteristics	Mondial	IMDb
N-Triples File Size	27.6 MB	18.1 GB
Triple Types		
Class instances	37,468	32,349,586
`rdfs:Class` declarations	–	25
Classes	27	25
subClassOf axioms	–	16
Object property	32	35
Datatype property	27	89
Metadata labels declarations	86	147
Ranking datatype properties	70,611	25,968,919
Distinct indexed property values	45,325	10,571,370
Total number of triples	**266,985**	**201,622,903**
Jena Properties		
Database size	0.99 GB	42 GB
Lucene index size	4.11 MB	4.1 GB
Saving data time (upload + index)	16.8 sec	$\sim$ 4 h

[11]http://www.dbis.informatik.unigoettingen.de/Mondial
[12]http://www.imdb.com
[13]https://sites.google.com/view/quira/

Following [16], the benchmark keyword-based queries are not real user queries extracted from search engine logs, yet they reflect distinct information needs. The benchmark has 24 keyword-based queries for Mondial and 40 for IMDb, grouped according to the expected graph patterns in their answers, as shown in Tables A.1 and A.2. The average number of terms per keyword-based query is 3.42 for Mondial and 4.38 for IMDb.

By construction, all keyword-based queries have non-empty answers. For each keyword-based query K, the benchmark has a ranked list $S_{K,1}, ..., S_{K,m}$ of sets of triples of the underlying dataset, called the *solution generators* for K. Each solution generator $S_{K,p}$ has a set of literal nodes, called *seeds*, that match the keywords in K. The solution generators computed foreach keyword query are available at `https://figshare.com/s/ef9aed9657255a01c008` in the path `src/main/resources/benchmarks/ER2020`.

To test an RDF keyword search algorithm $\mathcal{A}$, one would use solution generators as follows. For each keyword-based query K of the benchmark, one would submit K to $\mathcal{A}$ and obtain one or more answers $A_{K,1}, ..., A_{K,n}$. Each answer $A_{K,q}$ should be considered relevant iff $A_{K,q}$ induces a connected subgraph of one of the solution generators $S_{K,p}$ of K and $A_{K,q}$ includes all seeds of $S_{K,p}$ as nodes. One might assign a score to $A_{K,q}$ based on the number of keywords of K that $A_{K,q}$ matches (i.e., the number of seeds of $S_{K,p}$), the number of triples of $A_{K,q}$ and the position of $S_{K,p}$ in the ranked list of solution generators for K. The exact score function is user-defined and outside the scope of the benchmark.

5.1.2 Experimental Setup

• **KMV–synopses RDF Keyword Search Tool.** We implemented an RDF keyword search tool based on the proposed algorithm, which we refer to as the KMV-synopses RDF keyword search tool, or simply the *KMV–synopses tool*, to differentiate it from the baseline tool described below. We used Java 14 to implement the tool and Lucene, which is hosted with Jena, to index the datatype property values, including `rdfs:label` values. This feature permitted combining SPARQL queries and full-text search.

The tool precomputes the KMV–synopses and stores them together with the dataset. When the dataset is opened, the tool loads the KMV-synopses into main memory to speed up the compilation of keyword-based queries. If necessary, the tool recomputes the KMV-synopses from time-to-time, much in the same way that database systems recompute statistics, as already pointed out in Section 3.2.

To compute KMV–synopses, we adopted as hash function $h(x)$=`MD5(x)`$\%M^2$, where M is the number of class instances in the dataset. `MD5(x)` is computed by calling the static method `MessageDigest.getInstance("MD5")`[14]. We used $k = 16,384$ for IMDb, and $k = 8,192$, for Mondial.

Table 4 shows the time and space the KMV–synopses tool required to compute the KMV-synopses for the Mondial and IMDb datasets, which are consistent with the fact that IMDb is 3 orders of magnitude larger than Mondial. Note that, even for IMDb, it would indeed be feasible to recompute the KMV-synopses from time-to-time, if IMDb is updated.

Table 4: Space and time required to construct and store KMV–synopses

Dataset	Time	Space
Mondial	9 sec	439.0 KB
IMDb	152 min	30.6 MB

For each keyword-based query, the KMV-synopses tool returns a ranked list of answers.

• **Baseline.** As baseline, we adopted the schema-based RDF keyword search tool described in [24], which we had full control and could test the performance in a carefully controlled environment. This baseline tool is fast and had good precision in earlier experiments [24, 33]. For simplicity, we refer to it as the baseline tool. We extended the original implementation, which used Oracle RDF, to Jena to be able to run the benchmark queries and have a fair comparison baseline.

For each keyword-based query, the baseline tool returns an ordered list of answers, but it does not adopt any particular ranking strategy.

• **Metrics.** To measure the effectiveness of the tools, we used the same metrics considered in [16]: *Mean Average Precision* (*MAP*), *Top-1*, and *Mean Reciprocal Rank* (*MRR*). The *average precision* for a query is the average of the precision values computed after each relevant answer is retrieved (and assigning a precision of 0.0 to any relevant answer not retrieved). If the average precision value of a query is 0, then we consider it a *failed query*. *MAP* is the average of precision across all queries and provides a single-figure measure of quality across recall levels. It has been shown that, among the IR measures, *MAP* has especially good discrimination and stability. The number of *top-1* relevant answers is the number of queries for which the first answer belongs to the highest-ranked relevant answer retrieved by the system. The *reciprocal rank* is

[14]https://docs.oracle.com/javase/7/docs/api/ java/security/MessageDigest.html

the reciprocal of the highest-ranked relevant answers for a given query. *MRR* is a statistical measure for evaluating any system that produces a ranked list of answers for every query. It considers the highest-ranked relevant answer retrieved for each query. The *Reciprocal Rank* (*RR*) of a query is calculated by reversing the rank of the highest-ranked relevant answer retrieved by the system. *MRR* is computed by averaging *RR* values over all queries. *Top-1* and *MRR* metrics are known to be poorly stable, but they indicate the quality of the top-ranked answers.

Let K be a keyword-based query, with a ranked list $S_{K,1}, ..., S_{K,m}$ of solution generators, defined in the benchmark. Recall that each of the tools returns an ordered list $\mathcal{A} = (A_{K,1}, ..., A_{K,n})$ of answers for K. Then, we tested each answer $A_{K,q}$ in the ordered list for relevancy, as explained at the end of Section 5.1.3, finding the $S_{K,p}$ (if it exists) for which $A_{K,q}$ is relevant, and applied the metrics described earlier. In particular, we used the ranked positions q and p to compute the *MRR* metric, and we only tested if the first answer $A_{K,1}$ is relevant with respect to the first solution generator $S_{K,1}$ to compute the *top-1* metric.

We computed the Top-1 and MRR metrics only for the proposed tool since the baseline tool returns an unordered list of answers. For this reason, we also did not use the *normalized discounted cumulative gain* (NDCG) in these experiments. Indeed, it would be unfair to compare the approach proposed in this article with the baseline tool, using ranking metrics such as Top-1, MRR, and NDCG.

• **Hardware and Software Setup.** All tests were executed on a desktop machine with OS Windows 10 Pro, a quad-core processor Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz, 16GB of RAM. To store and manage the RDF dataset, we used the component TDB2 of Apache Jena for RDF[15]. Apache Jena Fuseki (a SPARQL server) ran on a server machine with OS GNU/Linux Ubuntu 16.04.6 LTS, a quad-core processor Intel(R) Core(TM) i7-5820K CPU@3.30GHz, 64 GB of RAM and SSD 1TB.

The critical drawbacks of executing queries in Jena are the size of the heap memory and the timeout. For our tests, we configured the value of heap memory as `JVM_ARGS: -Xmx60G` and set the query timeout to 2 hours.

5.1.3 Experimental Evaluation

We examined the performance of the KMV–synopses tool and compare it with the baseline described in Section 5.1.2. Firstly, we will analyze the *effectiveness*

[15]https://jena.apache.org

of the proposed algorithm. Later, we will focus on its *efficiency*.

• **Effectiveness.** Table 5 summarizes the results for Mondial. Reading the table from left to right, the baseline tool failed in 10 queries (42% of the queries), while the KMV-synopses tool only failed in 1 of 24 queries (4% of the queries) w.r.t. the benchmark. Indeed, in Query 22 {*Atacama, Province, Argentina*}, the KMV-synopses tool failed because it chose the instance *Atacama* of class `Province` and tried to connect it with the instance *Argentina* of class `Country`, which resulted in an empty set of answers. For all query groups, the KMV-synopses tool achieved better average precision than the baseline tool. For instance, the baseline tool failed for query groups *C* and *D*, while the KMV-synopses tool processed all such queries correctly.

Table 5: Experiments with Mondial

Benchmark		Failed Queries		AP		Top-1	RR
Query Groups	**#Queries**	**BL**	**KMV**	**BL**	**KMV**	**KMV**	**KMV**
A - Retrieve resources using Metadata and Values Matches	4	–	–	1.00	1.00	1.00	1.00
B - Join of instances of different classes using values and metadata matches	4	–	–	1.00	1.00	1.00	1.00
C - Join resources of the same class using values matches	4	4	–	0.00	1.00	1.00	0.75
D - Join two same class resources to resources of another class	4	4	–	0.00	1.00	1.00	1.00
E - Join a pair of resources from different classes to elements of another class through intermediary nodes	4	–	–	1.00	1.00	1.00	0.55
F - Join resources of various classes	4	2	1	0.50	0.75	0.75	0.43
OVERALL	24	*10*	**1**	*0.58*	**0.96**	**0.96**	**0.79**

BL – baseline tool **KMV** – KMV–synopses tool **AP** – Average Precision **RR** – Reciprocal Rank

Table 6: Experiments with IMDb

Benchmark		Failed Queries		AP		Top-1	RR
Query Groups	**#Queries**	**BL**	**KMV**	**BL**	**KMV**	**KMV**	**KMV**
A - Retrieve resources using Metadata and Values Matches	7	1	–	0.86	1.00	1.00	0.92
B - Specify Instances filtering by Property Values Matches	4	1	–	0.75	1.00	1.00	0.86
C - Join of instances of different classes using values and metadata matches	15	15	6	0.00	0.60	0.60	0.60
D - Join a pair of instances of different classes using values matches	3	3	2	0.00	0.33	0.33	0.33
E - Join two same class resources to resources of another class	6	6	1	0.00	0.83	0.83	0.83
F - Join resources of various classes	5	5	1	0.00	0.80	0.80	0.80
OVERALL	40	*31*	**10**	*0.27*	**0.76**	**0.76**	**0.72**

BL – baseline tool **KMV** – KMV–synopses tool **AP** – Average Precision **RR** – Reciprocal Rank

Table 6 summarizes the results for IMDb. Again, reading the table from left to right, the baseline tool failed in 31 queries (77.5% of the queries), while the KMV-synopses tool only failed in 10 of 40 queries (25% of the queries) w.r.t. the benchmark. The baseline tool failed for all IMDb benchmark queries in groups *C* to *F* since the use of the RDF schema for compiling the queries in these groups is not sufficient. Thus, the KMV–synopses tool reached a much higher mean average precision than the baseline tool. However, in Query 17 {*rick, blaine, movie*}, it failed since the keywords "*rick*" and "*blaine*", referring to instances of class `Person`, instead of instances of class `Character`, joined with class Movie, generated a query that returned an empty set of answers. Indeed, *Rick Blaine* did not work in movies, but rather in video movies.

Moreover, in Queries Q28={*tom, hanks, 2004*} and Q29={*audrey, hepburn, 1951*}, the KMV–synopses tool matched the numbers "*2004*" and

"*1951*" with values of properties `:death_date` and `:birth_date`, respectively; the generated query then returned an empty set of answers, since these values do not simultaneously occur in instances that refer to "*tom hanks*" and "*audrey hepburn*".

Tables 5 and 6 also show the computed values of *Top-1* and *MRR* metrics for the proposed algorithm. In both datasets, the obtained values are considerably high. This means that the proposed algorithm returned relevant top-1 answers for the non-failing queries. Note that each line of these tables indicates the results and averages for a specific query group; only the last line indicates the overall averages.

To summarize, the experiments show that the KMV–synopses tool outperforms the baseline tool in all metrics.

• **Efficiency.** The *total elapsed time* depends on the *translation time*, that is, the time to compile the keyword query into a SPARQL query, and the *execution time*, that is, the time the RDF Search Engine takes to execute the SPARQL query. The total elapsed time naturally depends on the RDF Search Engine chosen (Jena, in this case). Hence, we concentrate on the translation time.

In the KMV–synopses tool, the translation time can be broken in two components: the *match time*, that is, the time it takes to match keywords with literals; and the *assembly time*, that is, the time it takes to select the best matches and to discover how to join the match results. The experiments indicated that, on average, the assembly time is 80% of the translation time.

The min time, max time, and average time, in seconds, for Mondial were 0.3s, 0.9s, and 0.6s, respectively, and for IMDb were 1.2s, 59.3s, and 11.4s, respectively. The total elapsed time was much higher for IMDb than for Mondial, since IMDb is 3 orders of magnitude larger than Mondial, and since the keyword queries for IMDb were somewhat more complex.

The last columns of Tables A.1 and A.2 (in Appendix), labeled with τ, show the total elapsed time the KMV–synopses tool took to execute each keyword-based query in the benchmark. Overall, the elapsed times of the KMV–synopses tool and the baseline tool were similar. From the experiments, for the KMV–synopses tool, we also observed that the translation time for Mondial was 45–52% of the total elapsed time, on average. For IMDb, it raised to 62–70% on average. This behavior can be explained because keyword matching is a costly process, which is heavily affected by the dataset size and the ambiguity of data, such as IMDb.

Finally, we note that, for the KMV-synopses tool, when the keyword-based query had few matches, but the compiled SPARQL query had many

joins, then the execution time represented most of the total elapsed time, such as for the Mondial benchmark keyword queries in Group *D*.

5.2 Comparison with Keyword Search Systems based on the "Virtual Documents" Approach

This section compares the schema-less approach proposed in this thesis with the state-of-the-art TSA+BM25 and TSA+VDP keyword search systems over RDF datasets based on the "virtual documents" approach, adopted as baselines. Section 5.2.1 describes the benchmark adopted to perform the comparison. Section 5.2.2 discusses the configurations to perform the experiment. Finally, Section 5.2.3 reports the experiment result according effectiveness and efficiency.

5.2.1 Benchmark Adopted

This section summarizes the benchmark described in [18], for which we refer the reader. The benchmark and the code required to run the experiments are available at `https://bitbucket.org/account/user/keywordsearchrdfproject/projects/TSAC`.

The original benchmark contains three real datasets: LinkedMDB, IMDb, and a subset of DBpedia; and two synthetic databases: the Lehigh University Benchmark (LUBM) and the Berlin SPARQL Benchmark (BSBM). However, we decided not to use the LinkedMDB dataset since the number of triples is not significantly large. So, we selected the LUBM[16], BSBM[17], IMDb[18], and DBpedia[19] datasets.

LUBM is a database about universities, professors, and students developed by Lehigh University to facilitate the evaluation of Semantic Web Repositories. BSBM is a database built on an e-commerce use case, where different vendors with posted reviews offer a set of products. The LUBM benchmark has 14 SPARQL test queries[20], whereas the BSBM Explore use case[21] has 13 different SPARQL queries. For each synthetic dataset, a version of about 10M triples was used.

[16] http://swat.cse.lehigh.edu/projects/lubm/
[17] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/
[18] https://datasets.imdbws.com/
[19] https://wiki.DBPedia.org/data-set-37
[20] http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt
[21] http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/ExploreUseCase/index.html

IMDb is a relational dataset that describes has movies, series, and artists are their relationships. We convert it into an RDF dataset with 256M triples. Finally, for DBpedia, we built an RDF graph composed by the triples in `https://wiki.DBPedia.org/data-set-37` corresponding to the DBpedia Ontology, the Ontology Infobox Types, the Titles subset, the Short Abstract subset, and the Raw Infobox Properties subset.

For IMDb, Dosso & Silvello designed 100 topics, where half were used for training and a half for testing. We used the 50 queries that they designed for testing. As for DBpedia, Dosso & Silvello considered 50 topics from `QALD2_te` and `QALD2_tr`, used by [6], manually mapped into SPARQL `CONSTRUCT` queries and the corresponding keyword queries. A topic is composed of three fields: the title, the dsc (description), and the SPARQL query. The SPARQL query contains the SPARQL query that returns the "correct" answer.

We computed the InfoRank values for resources, properties, and classes for all datasets and added them to the datasets. Table 7 summarizes the statistics about the used datasets. For comparison purposes, the benchmark is openly available at `https://figshare.com/s/d65d6a4ec70f169b4c50`.

Table 7: Statistics — LUBM, BSBM, IMDb, and DBpedia datasets

Dataset	Type	#Triples	#InfoRank Triples	#Queries
BSBM	synthetic	12M	1.6M	13
LUBM	synthetic	12M	1.7M	14
IMDb	real	256M	40.2M	50
DBpedia	real	72M	3.2M	50

5.2.2 Experimental Setup

• **KMV–synopses RDF Keyword Search Tool.** The KMV–synopses tool was already described in Section 5.1.2.

We computed the synopses for all datasets varying the parameter k. Table 8 shows the total space (in MB) required to store the KMV-synopses and the creation time in minutes. For the running experiment, we used $k = 8,192$ for all datasets and tests.

For DBpedia, we computed the KMV–synopses for 44,809 properties, considering their domains and ranges. Nevertheless, the sets of property domains and object property ranges are very skewed.

For instance, we had:

– 607 domain synopses with k = 8,192

Table 8: KMV–synopses sizes (in MB) and creation time consumption (minutes)

Dataset	k = **4,096**	k = **8,192**	k = **32,768**	**Time**
BSBM	2.83	4.66	13.3	~ 30
LUBM	1.26	2.47	9.12	~ 30
IMDb	0.89	1.96	7.21	~ 1000
DBpedia	109	143	232	~ 200

– 37 range synopses with k = 8,192

– 28,036 domain synopses with $k < 10$

– 8,138 range synopses with $k < 10$

For example, the object property `dbp:teamDirector` links a unique pair of resources: `dbr:Germany_national_handball_team` (in the domain) and `dbr:Tom_Schneider` (in the range).

• **Baseline.** As baseline for these experiments, we adopted the TSA+BM25 and the TSA+VDP keyword search systems over RDF datasets based on the "virtual documents" approach, described in [18]. These systems move most of the computational complexity off-line and then exploit highly efficient text retrieval techniques and data structures to carry out the on-line phase. Dosso and Silvello showed that these approaches are more efficient and effective, when compared with state-of-the-art systems.

• **Metrics.** For comparison purpose, the metrics adopted to evaluate the proposed approach in these tests were introduced in [18].

Recall that a *Cranfield framework* is a triple $C = (D, T, GT)$, where D is a dataset, T is a set of topics, and GT defines the ground truths. In our contexts, D is an RDF dataset, T is a set of keyword queries, and the ground truth for a topic $t_k \in T$, denoted G_{t_k}, is an RDF graph defined by the result of applying the SPARQL `CONSTRUCT` query Q_{t_k} over D. The relevant triples for topic t_k are the triples that correspond to G_{t_k}; all other triples are not relevant for topic t_k.

Let t_k be a topic, Q_k be the corresponding keyword query, and G_{t_k} be the ground truth defined for t_k.

We assume that the keyword query system returns a *ranked answer list* $R_k = [ap_1, ap_2, ..., ap_n]$, where each $ap_i = (G_i, sim_i)$ is an *answer pair*, where G_i is the *answer graph* at position i and sim_i is the *similarity* between G_i and G_{t_k}. As an abuse of notation, we write $G_i \in R_k$, when there is an answer pair of the form (G_i, sim_i) in R_k.

The *Signal-to-Noise Ratio* (*SNR*) of G_i is defined as

$$SNR(G_i) = \frac{|(G_i \cap G_{t_k}) - S|}{G_i} \tag{13}$$

where S is the union of all relevant triples for topic t_k that are also in the answer graph at position j, for all $j \in [1, i)$.

Note that SNR represents a kind of precision score, where the numerator is equal to the number of relevant triples in G_i found for the first, that is, excluding the triples in answer graphs that preceded G_i in the ranking R_k, and the denominator is the total number of triples in G_i. The example described in Appendix B.1, taken of [18], illustrates the definition of *SNR*.

Let λ be a relevance parameter such that an answer graph $G_i \in R_k$ is considered relevant iff $SNR(G_i) > \lambda$.

The *recall* of a ranked answer list R_k for topic t_k and ground truth G_{t_k} is defined as

$$recall(R_k) = \frac{|\bigcup_{G_i \in R_k | SNR(G_i) \geq \lambda} (G_i \cap G_{t_k})|}{|G_{t_k}|} \tag{14}$$

Intuitively, the recall of R_k is the ratio between the set of relevant triples that appear in some relevant answer graph in R_k and the cardinality of the ground truth. Appendix B.2 shows a quite example of how to compute the recall of a ranked answer list R_k.

The *precision* of a ranked answer list R_k for topic t_k ground truth G_{t_k} is defined as

$$precision(R_k) = \frac{|\bigcup_{G_i \in R_k | SNR(G_i) \geq \lambda} (G_i \cap G_{t_k})|}{|\bigcup_{G_i \in R_k} G_i|} \tag{15}$$

Intuitively, the precision of R_k is the ratio between the set of relevant triples that appear in some relevant answer graph in R_k and the cardinality of the set of triples that appear in some answer graph in R_k.

The *precision at c*, denoted *prec@c*, is the precision computed considering the first c elements of the ranking, and is defined as

$$prec@c(R_k) = \frac{|\bigcup_{G_i \in R_k | SNR(G_i) \geq \lambda \wedge i \in [1,c]} (G_i \cap G_{t_k})|}{|\bigcup_{G_i \in R_k \wedge i \in [1,c]} G_i|} \tag{16}$$

Appendix B.3 exemplifies how to compute the *precision* and *precision at 1* values using Equations (15) and (16).

The *Graph Relevance Weight* (GRW) measures the relevance of the answer graph at i of ranking R_k:

$$GRW(G_i) = \frac{|(G_i \cap G_{t_k}) - S|}{|G_{t_k}|} \tag{17}$$

where S again is the union of all relevant triples for topic t_k that are also in the answer graph at position j, for all $j \in [1, i)$.

The example described in Appendix B.4 shows the computation of GRW.

The *Relevance Gain* (RG) computes the relevance gain of an answer G_i in ranking R_k of size n ($n > 0$), for a given topic t_k, considering $\lambda \in [0, 0.1, 0.2, ..., 1]$ and a position b ($b > 0$).

$$RG_b(G_i) = \begin{cases} GRW(G_i) & if \quad i \leq b \ \wedge \ SNR(G_i) > \lambda \\ \frac{GRW(G_i)}{\log_b i} & if \quad i > b \ \wedge \ SNR(G_i) > \lambda \\ 0 & if \quad SNR(G_i) \leq \lambda \end{cases} \tag{18}$$

Finally, the *triple-based Discounted Cumulative Gain* ($tb - DCG$) of a ranking R_k is defined as:

$$tb - DCG_b(R_k) = \sum_{i=1}^{n} RG_b(G_i) \tag{19}$$

where b represents the highest position in relevance to the ranked answers graph.

This metric measures the overall utility of a subgraph ranking for the end-users. It weighs the top-heaviness (best answers are ranked first) and essentialness (absence of redundancy) in the ranking. Appendix B.5 exemplifies the computation of RG and $tb - DCG$ values.

• **Hardware and Software Setup.** All tests were performed under the same conditions described in Section 5.1.2.

5.2.3 Experimental Evaluation

This section describes the results obtained for the KMV–synopses tool and compares them with the baseline results from [18], using the metrics summarized in Section 5.2.2, also from [18].

• **Effectiveness.** Table 9 shows the values of the metrics computed for the KMV–synopses tool, along with the values for the baselines from [18] over the four datasets. To compare with the baselines, we used $\lambda = 0$ to estimate the relevance of the answer graphs in the rankings returned by the KMV–synopses tool, as in [18]. Recall that, by choosing $\lambda = 0$, every answer containing at least one relevant triple is considered relevant.

For the BSBM, LUBM, and IMDb datasets, the KMV-synopses tool obtained higher metrics values than the TSA systems based on the "virtual document" approaches. As for precision, this means that the KMV–synopses tool finds a larger number of relevant triples w.r.t the ground truth and ranks them adequately, contrasting with the TSA systems, that return a high number of noisy triples in the answers.

Focusing on DBpedia, the "virtual document" approach based on the BM25 function had a higher recall value than the other systems but obtains

Table 9: Results obtained with the experiments using SRR and $\lambda = 0$

Dataset	System	Prec@1	Prec@5	Recall	tb-DCG
BSBM	TSA+BM25	0.039 ± 0.01	0.010 ± 0.00	0.227 ± 0.07	0.139 ± 0.05
	TSA+VDP	0.071 ± 0.03	0.071 ± 0.03	0.047 ± 0.03	0.074 ± 0.03
	KMV–Synopses	***0.815***	***0.823***	***0.852***	***0.720***
LUBM	TSA+BM25	0.082 ± 0.04	0.111 ± 0.05	0.505 ± 0.07	0.281 ± 0.07
	TSA+VDP	0.145 ± 0.03	0.226 ± 0.05	0.384 ± 0.03	0.234 ± 0.06
	KMV–Synopses	***0.905***	***0.885***	***0.684***	***0.539***
IMDb	TSA+BM25	0.011 ± 0.00	0.009 ± 0.00	0.273 ± 0.36	0.067 ± 0.01
	TSA+VDP	0.006 ± 0.00	0.006 ± 0.00	0.363 ± 0.04	0.308 ± 0.04
	KMV–Synopses	***0.810***	***0.761***	***0.648***	***0.681***
DBpedia	TSA+BM25	0.000 ± 0.00	0.000 ± 0.00	***0.851 ± 0.03***	0.135 ± 0.01
	TSA+VDP	0.002 ± 0.00	0.002 ± 0.00	0.129 ± 0.03	0.118 ± 0.03
	KMV–Synopses	***0.233***	***0.217***	0.191	***0.363***

0 for precision values. This means that the system returns many relevant triples but cannot rank them effectively. Moving to precision and *tb-DCG*, the KMV–synopses tool obtained better values, even though the tool failed in 19 of the 50 benchmark queries (the synthesized SPARQL queries returned empty answers). This fact also influenced the low recall value. Another factor that affects the effectiveness of the KMV–synopses tool is the high degree of ambiguity of DBpedia. For example, the keyword "*governor*" exactly matches the `rdfs:label` properties of class `dbo:Governor` and the object property `dbo:governor`. Thus, deciding which element should be used to synthesize the SPARQL query is critical for the KMV–synopses tool. The current implementation prioritizes the class match found, as explained in Section 4.4.1. In the future, it is advisable to improve this heuristic by analyzing the keyword-based query context (for example, the sequence of keywords) to enhance precision and to return relevant answer graphs.

Concerning IMDb, the dataset also has a high degree of ambiguity but, in this case, the ambiguity has to do with the resource property values. For instance, the keyword "*Will Smith*" matches more than a hundred property values. However, using the ranking heuristic described in Section 4.4.2, the KMV–synopses tool selected and included in the synthesized SPARQL query the resource with the highest *InfoRank* value, in this case, the resource expected in the ground truth. This fact again raises the discussion if a manually defined ground truth covers all possible answers for a keyword-based query.

As for LUBM, the KMV–synopses tool synthesized SPARQL queries with non-empty answers for all topics in the benchmark. However, there are non-relevant graphs in the answers, since the precision value is not 1. For example, the system does not achieve perfect precision for the benchmark query Q2={*GraduateStudent, University, Department, memberOf, subOrganizationOf, undergraduateDegreeFrom*}, where the ground truth is the graph

resulting from the SPARQL `CONSTRUCT` query (query details were omitted by brevity):

```
1  CONSTRUCT WHERE{
2   ?X rdf:type swat:GraduateStudent .
3   ?Y rdf:type swat:University .
4   ?Z rdf:type swat:Department .
5   ?X swat:memberOf ?Z .
6   ?Z swat:subOrganizationOf ?Y .
7   ?X swat:undergraduateDegreeFrom ?Y }
```

Note that the variable `?Y` binds resources of class `swat:University` (line 3) and then `?Y` simultaneously appears in the object of the triple patterns of the object properties `swat:subOrganizationOf` and `swat:undergraduateDegreeFrom` (lines 6 and 7). The `WHERE` clause indicates a triangular pattern of relationships between the objects involved, which is hard to indicate through keywords. Thus, the KMV–synopses tool compiles a SPARQL query similar to above, replacing the variable `?Y` by `?W` in line 7.

A similar situation was observed for some queries in the IMDb and DBpedia benchmarks. For example, the IMDb benchmark queries Q31 to Q40 and the DBpedia benchmark queries Q1 and Q21 find films where a person (identified by her/his name) is an actress/actor and, at the same time, the film is directed/written/produced by herself/himself, such as the benchmark query Q1={*Clint Eastwood, starring, director*} in DBpedia.

To summarize, the experiments showed that the KMV–synopses tool obtained good precision and recall values, and also produced reasonable rankings. Indeed, the KMV–synopses tool compared favorably with the baseline, state-of-art systems in terms of effectiveness.

Finally, we observe that, unlike [18], the synthetic databases posed no problems for the KMV-synopses tool, whereas IMDb and DBpedia were harder to handle, due to their ambiguity. Furthermore, recall from Section 5.2.1 that [18] defined the ground truth for each keyword query as a single RDF graph that is the result of a manually specified SPARQL `CONSTRUCT` query. This decision raises at least two questions when assessing the effectiveness of an RDF keyword-based query system. First, the SPARQL query does not necessarily cover all possible answers for the keyword query. Second, the answers are not individualized, so computing effectiveness required redefining the notions of precision and recall, as described in Section 5.2.2.

• **Efficiency.** Recall the translation time corresponds to the time to compile the keyword query into a SPARQL query and the total elapsed time is the time consumed by the system since it receives the keyword query until it returns the corresponding answer. We consider that comparing the times of the KMV–

synopses system against the on-line times reported for the baselines in [18] is not reasonable since the experimental environments and RDF engines used in both experiments were different. However, we included, in Table 10, the translation and the total elapsed times of our proposed system for all four datasets, which are comparable to those reported in in [18] for the baselines.

Table 10: Minimum, Maximum, and Average for Translation and Total Elapsed Times (in sec)

Dataset	Translation Time			Total Elapsed Time		
	Min	Max	Ave	Min	Max	Ave
BSBM	2.468	9.768	6.195	4.529	11.280	7.322
LUBM	4.213	8.776	6.205	5.188	9.120	7.609
IMDb	1.542	102.792	60.609	2.896	1200	254.524
DBpedia	0.896	560.541	95.227	1.654	1200	441.875

Concerning the execution times, as expected, the times consumed to run the queries on top of the synthetic datasets (BSBM and LUBM) were considerably faster than the times in the real datasets (IMDb and DBpedia).

We focus here on the translation time of the KMV-synopses tool. We observed that, on average, the translation times for the benchmark queries of the BSBM and LUBM datasets were very similar, which can be explained by the low degree of ambiguity, which in turn implies a few matches for a keyword in the query.

Moving to IMDb, the translation time for the benchmark queries was also reasonably low. The dataset structure explains this behavior since the IMDb dataset essentially consists of instances of two different classes (`Person` and `Film`) with their datatype properties and few object properties linking them. So, the assembly process does not consume much time connecting the resources resulting from the matching process. On average, the execution time of a query represented 80% of the total elapsed time. In some cases, this proportion even exceeded 98%.

Regarding DBpedia, the performance of the KMV–Synopses tool was much lower in terms of translation times. This fact was expected because, as mentioned, the degree of ambiguity and the graph structure, regarding the number of object properties available to connect the resources in the graph, influences the matching and assembly processes, respectively. On average, the translation time consumed 80% of the total time in the syntactic databases. Concerning the ratio between the matching and the assembly processes times, we observed that, on average, the assembly process consumed more time,

except for the typical cases described in Section 4.4.3, since the assembly process directly compiles the described triple pattern.

5.3 Effectiveness Using an Alternative Measure for Graph Relevance

As mentioned in Section 5.2.2, [18] defined SNR to determine when an answer graph is relevant. However, this measure strongly considers the relevant triples observed at the top-ranked graphs. By contrast, the KMV–synopses tool returns answer graphs with relevant triples in any ranking position. Thus, this section proposes a different measure to establish when an RDF graph is relevant. It is based on the number of relevant and non-relevant triples in the RDF graph, but it punishes the presence of non-relevant triples, and does not memorize the relevant triples in previous rank positions.

Given a topic $t_k \in T$, a ranking R_k, and the ground truth graph G_{t_k}, we define the *Graph Relevance Ratio* (GRR) of $G_i \in R_k$ to establish when a graph is relevant. Note that this measure is asymmetric, as we are only interested in comparing G_i against G_{t_k}. Hence, we use a variant of the Tversky index [56].

$$S(X,Y) = \frac{|X \cap Y|}{|X \cap Y| + \alpha|X - Y| + \beta|Y - X|} \tag{20}$$

From Eq. (20), if we consider S as GRR, $X = G_i$, $Y = G_{t_k}$, $\alpha = 1$, and $\beta = 0$, then the $GRR(G_i, G_{t_k})$, or simply $GRR(G_i)$, is defined as

$$GRR(G_i) = \frac{|G_i \cap G_{t_k}|}{|G_i \cap G_{t_k}| + |G_i - G_{t_k}|} \tag{21}$$

Intuitively, by taking $\alpha = 1$, and $\beta = 0$, we consider the triples in the answer graph G_i which are not in the ground truth G_{t_k}, and ignore the triples which are in the ground truth G_{t_k}, but not in the answer graph G_i. Note that Eq. (21) is then equivalent to set containment, since $|G_i \cap G_{t_k}| + |G_i - G_{t_k}| = |G_i|$. Indeed, we can redefine GRR as

$$GRR(G_i) = \frac{|G_i \cap G_{t_k}|}{|G_i|} \tag{22}$$

Also, note that the GRR of an answer graph G_i, as SNR, rewards precise and essential graphs as it decreases whenever the graph contains non-relevant triples.

Now, inspired by [18], we redefine $precision(R_k)$ and $precision@c(R_k)$, using GRR, as

$$precision(R_k) = \frac{|\{G_i \in R_k | GRR(G_i) \geq \lambda\}|}{|R_k|} \tag{23}$$

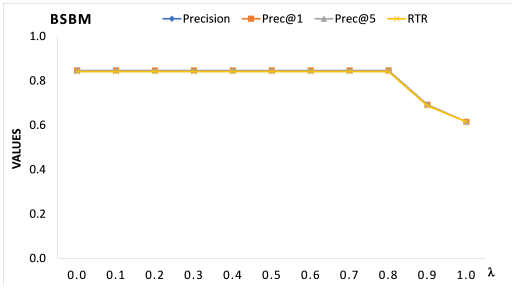
$$prec@c(R_k) = \frac{|\{G_i \in R_k | GRR(G_i) \geq \lambda \ \ \wedge \ i \in [1,c]\}|}{c} \tag{24}$$

By requiring that $GRR(G_i) \geq \lambda$, we discard those answer graphs G_i that have fewer relevant triples in the ground truth, as compared with the total number of triples in G_i.

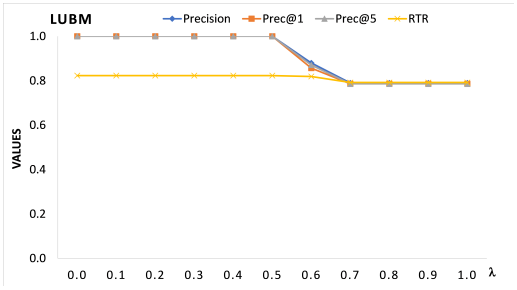
Here, precision is the ratio between the total number of relevant graphs and the total number of graphs in the ranking.

However, we decided not to name *recall* the metric equivalent to that proposed in [18], since the ground truth for a topic t_k, G_{t_k}, is a compact graph that does not individualize the answers. Therefore, we redefine the recall metric of Section 5.2.2 under the name *Relevant Triples Ratio* of R_k ($RTR(R_k)$, for short) as

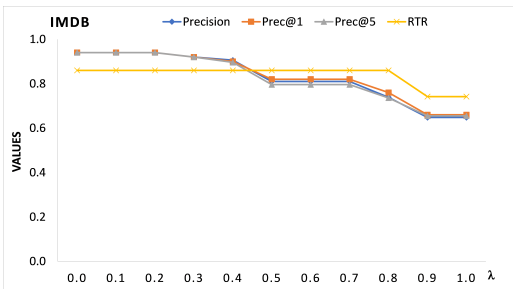
$$RTR(R_k) = \frac{|\bigcup_{G_i \in R_k \mid GRR(G_i) \geq \lambda} (G_i \cap G_{t_k})|}{|G_{t_k}|} \tag{25}$$



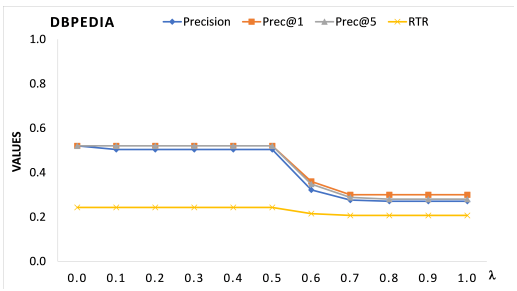
12(a) Results for BSBM dataset



12(b) Results for LUBM dataset



12(c) Results for IMDb dataset



12(d) Results for DBpedia dataset

Figure 12: Metrics values computed for the four datasets

We then computed the *precision*, *prec@1*, *prec@5*, and *RTR* values for the results of the experiments with the four datasets, using all values of λ in the set $\{0.0, 0.1, 0.2, \ldots, 1.0\}$, and the Equations 23, 24, and 25; see the results in the graphics depicted in Figure 12. Figure 12(a) shows the values for BSBM, Figure 12(b) for LUBM, Figure 12(c) for IMDb, and Figure 12(d) for DBpedia.

Note that, as expected, when the value of λ increases, the metrics values decrease. It means that the more restrictive it is to consider that an answer graph is relevant, fewer of the answers found are regarded as "*correct*" w.r.t G_{t_k}. As mentioned, the KMV–synopses tool returns non-empty answers for all benchmark queries in the LUBM dataset, and for relevance parameter $\lambda \leq 0.5$,

the precision value is perfect. It means that, for each topic, all returned graphs are relevant. We observed that the RTR values are quite stable for all values of λ. It means that the answer graphs considered as non-relevant contain few relevant triples. We note that, for $\lambda = 0.5$ or larger, the metrics values decreased. For the BSBM dataset, they begin to decrease at $\lambda = 0.8$.

Therefore, Table 11 shows only the metrics values obtained with the relevance parameter $\lambda = 0.8$ for all four datasets.

Table 11: Metrics values computed using the proposed GRR and $\lambda = 0.8$

Dataset	**Precision**	**Prec@1**	**Prec@5**	**RTR**
BSBM	0.846	0.846	0.846	0.841
LUBM	0.790	0.786	0.786	0.792
IMDb	0.739	0.760	0.736	0.860
DBpedia	0.271	0.300	0.280	0.207

Again, computing measures as DCG or NDCG is not viable since, as already mentioned, the ground truth is not a ranked list of individualized answers but a single RDF graph.

6
A Formal Framework for Querying Semantic Trajectories

This chapter defines a formal framework for querying semantic trajectories. Firstly, Section 6.1 provides a brief description of a semantic trajectory dataset, called *TripBuilder*, and introduces some sample queries that help illustrate the notation used in the keyword-based queries. Section 6.2 describes the concepts in the proposed framework by divided them into *core* and *extended* models. Section 6.3 formalizes the core and extended models in Description Logic and defines the notion of semantic trajectory induced by a individual trajectory. Section 6.4 defines a query language for semantic trajectory datasets. Finally, Section 6.5 presents some extensions to the proposed framework to explore spatio-temporal aspects.

6.1
A Keyword Search over Semantic Trajectories Use-Case

This section provides a brief description of a semantic trajectory dataset, called *TripBuilder*, and introduces some sample queries that help illustrate the notation used in the keyword-based queries.

6.1.1
Informal Description of the *TripBuilder* Trajectory Dataset

TripBuilder is a semantic trajectory dataset constructed from user-generated content obtained from Flickr, combined with data from Wikipedia [14]. The dataset contains user trajectories in 3 different Italian cities (Pisa, Rome, and Firenze). For example, for the city of Pisa, it contains 3,430 trajectories by 1,825 distinct users, from which only 389 trajectories (approximately 11%) have a length between 4 and 20. Figure 13 shows the distribution of trajectory lengths in terms of the number of stops.

To construct TripBuilder, users' photos collected by Flickr were clustered in the spatial dimension and relate to points-of-interest (POIs). The clusters generated from the photos of a user indicate her trajectory, assuming that she moves around the city taking many photos in various POIs. Each trajectory is characterized by: *cluster_id*; number of photos; time of the first photo; and time of the last photo. *Cluster_id* is, in essence, the key to get more

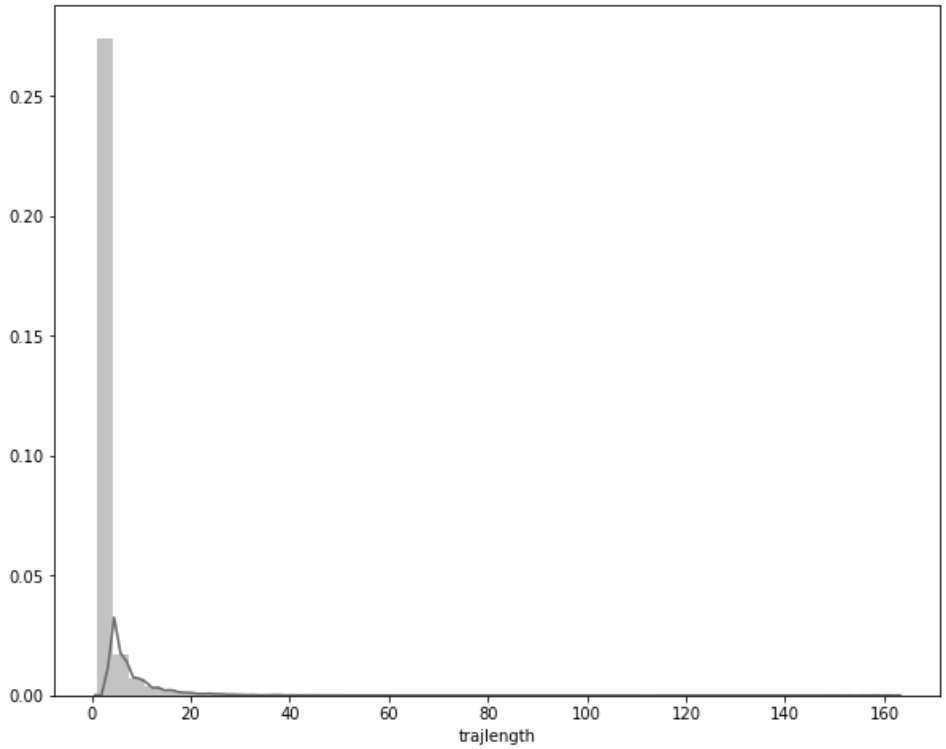


Figure 13: Distribution of trajectory lengths in the *TripBuilder* dataset (in all cities).

information about the name and category of a POI. Table 12 illustrates two trajectories, both of which have 6 POIs.

Table 12: Two illustrative trajectories, based on the *TripBuilder* dataset

Trajectory 1	Trajectory 2
1: 'Porta_Nuova_(Pisa)',	1: 'Torre_del_Leone',
2: 'Museo_delle_sinopie',	2: 'Torre_pendente_di_Pisa',
3: 'Cappella_Dal_Pozzo',	3: 'Camposanto_monumentale',
4: 'Museo_delle_sinopie',	4: 'Torre_del_Leone',
5: 'Chiesa_di_San_Giorgio_ai_Tedeschi',	5: 'Torre_pendente_di_Pisa',
6: 'Museo_delle_sinopie'	6: 'Camposanto_monumentale'

This work uses a triplified version[22] of the original TripBuilder dataset. The triplification follows the RDF schema described in Section 7.1 and is fully described in Section 7.5.1.

6.1.2 Notation and Query Examples

For our research, 10 sample queries were defined considering the TripBuilder RDF dataset. These queries can be expressed using:

[22]https://doi.org/10.6084/m9.figshare.11559090

- a *symbolic notation*, similar to that of regular expressions, that defines sequences of stop and move queries
- a *reserved terms-based notation*, which combines query terms and reserved terms that define the properties and interrelations of stops and moves.

Appendix C lists the sample queries (Q1 to Q10) defined for this research, using data from the city of *Pisa*. The first group of queries ignores moves and focuses only on the semantics of stops and their sequences. The queries related to this group (Q1 to Q7) are listed in Appendix C.1. The second group of examples seeks for semantic trajectories that combine stops and moves in a specific sequence. The sample queries related to this case (Q8 to Q10) are listed in Appendix C.2. Queries are first expressed in natural language, and then written using the symbolic and the reserved terms-based notation.

Table 13 summarizes the symbols and reserved terms of the proposed notation, with the list of symbols in the first column, their equivalent terms in the second column, and a description of their meanings in the last column.

Table 13: Alphabet for keyword queries over semantic trajectories

Symbol	Reserved Term	Description
Stop	"`any stop`"	the set of stops
Move	"`any move`"	the set of moves
Begin	`Begin`	the set of all beginning stops of trajectories
End	`End`	the set of all end stops of trajectories
$E \sqcup F$	`or`	the union of the results of queries E and F
$E \sqcap F$	`and`	the intersection of the results of queries E and F
E^+	"`at least once`"	repeat query E at least once
E^*	"`zero or more times`"	repeat query E zero or more times
$E \mid F$	`or`	execute query E or query F (but not both)
$E; F$	"`and then`"	execute query E and then query F (on consecutive stops or consecutive moves)
<M>	"`by...to`"	move from one stop to the next, where M is a query on moves

Note: E and F are queries that define a set of stops or a set of moves, depending on the context.

Table 14 shows sample query terms used in the TripBuilder RDF dataset[23], their free-text equivalents, and their meaning. The syntax will be gradually introduced along with the examples.

Section 6.4 provides a formal definition of how queries are evaluated against trajectories. Based on it, Chapter 7 introduces a concrete RDF framework for querying semantic trajectories. We assume that a query Q is evaluated against some segment of a trajectory τ. The segment is required neither to start at the beginning of τ nor to terminate at the end of τ. If Q

[23]The Pisa related terms of the TripBuilder dataset correspond to the respective Wikipedia and DBpedia entries (e.g. `https://it.wikipedia.org/wiki/Categoria:Musei_di_Pisa` or `https://it.wikipedia.org/wiki/Categoria:Chiese_di_Pisa`).

Table 14: Sample terms used on the TripBuilder dataset

TripBuilder term	Reserved Terms	Description
Museidipisa	`Musei Pisa`	the set of museums located in the city of Pisa
Cappelledipisa	`Cappele Pisa`	the set of chapels located in the city of Pisa
Chiesedipisa	`Chiese Pisa`	the set of churches located in the city of Pisa
Torredipisa	`Torre Pisa`	the set of towers located in the city of Pisa
transportation	`transportation`	indicates the transportation means of moves
Torre_pendente_di_pisa	`Torre pendente Pisa`	the Leaning Tower located in the city of Pisa
Torre_del_Leone	`Torre del Leone Pisa`	the Lion Tower located in the city of Pisa
Walk	`Walk`	the transportation means is 'by walking'
Taxi	`Taxi`	the transportation means is 'by taxi'
Bus	`Bus`	the transportation means is 'by Bus'
Subway	`Subway`	the transportation means is 'by subway'

must be evaluated against the complete trajectory, then the user must resort to the reserved symbols *Begin* and *End*, as in queries Q5, Q7, and Q10. Chapter 7.4 shows how to write the example queries in SPARQL.

6.2 Framework Overview

The formal framework for semantic trajectories provides the concepts needed to define the syntax and semantics of the query expressions and to define an RDF model and a SPARQL implementation of such expressions. To facilitate formalization, we divided the concepts in the framework into two groups:

i) the *core model* contains a minimum set of concepts whose properties can be concisely written in Description Logic, and that suffice to formalize semantic trajectories.

ii) the *extended model* that includes additional concepts that facilitate writing query expressions over semantic trajectories, as well as their SPARQL counterparts.

The core model has three classes: $Trajectory$, *Stop*, and $Move$; and a set of other classes collectively called *enrichment classes*. The individuals in $Trajectory$ are called *trajectory individuals*, those in *Stop* are called *stop individuals*, those in $Move$ are called *move individuals*, and those in the enrichment classes are called *enrichment individuals*. Whenever possible, we omit the term '*individual*' and refer simply to *trajectories*, *stops*, *moves*, and *enrichments*.

Thus, we assume that:

A1. $Trajectory$, *Stop*, and $Move$ are disjoint;

A2. $Trajectory$, *Stop*, and $Move$ are disjoint from the enrichment classes.

The core model also has four binary relationships, *enrichedBy*, *begins*, $from$, and *to*. We assume that:

A3. *enrichedBy* relates a stop or move to one or more enrichments;

A4. *begins* relates a trajectory to a single stop, called the *begin stop* of the trajectory, and a begin stop is related to a single trajectory by *begins*;

A5. $from$ relates a move to a single stop, and a stop is related to a single move by $from$;

A6. *to* relates a move to a single stop, and a stop is related to a single move by *to*;

A7. $from$ is defined for a move m_j iff *to* is also defined for m_j;

A8. *to* does not map a move to the begin stop of a trajectory.

Figure 14 schematically depicts a trajectory individual t in the core model. A formalization of the core model therefore reduces to capturing assumptions **A1**–**A8**, which is the focus of Section 6.3.

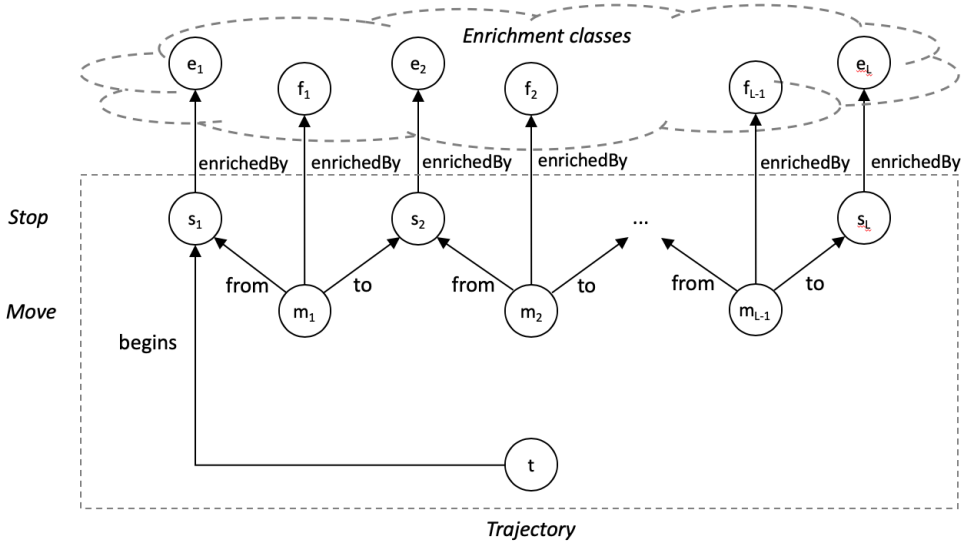


Figure 14: Schematic trajectory in the core model

Informally, examples of pairs in *enrichedBy* are:

- "*enrichedBy* relates stop s_1 to *torre_di_pisa*", where the interpretation of *torre_di_pisa* is the Leaning Tower of Pisa;
- "*enrichedBy* relates move m_j to *bus*", where *bus* denotes an individual of the class $Transportation$, indicating that the transportation mean used in move m_j is a *bus*.

Let t be a trajectory individual and s_1 be the begin stop of t. By **A4**–**A8**, we can define a unique sequence of stops, $\sigma = (s_1, \ldots, s_L)$, called the *stop sequence* of t, by traversing from s_1 to the other stops, using the $from$ and *to* relationships. Likewise, we can define a unique sequence of moves,

$\mu = (m_1, \ldots, m_{L-1})$, called the *move sequence* of t, by traversing from stop s_j to stop s_{j+1} moving by move m_j, using the $from$ and to relationships, for each $j \in [1, L-1]$.

Using **A4**–**A8**, we can prove the following properties:

P1. A stop or move belongs to at most one trajectory.

P2. A stop or move is not repeated in σ or μ.

Finally, using **A3**, we can construct a sequence of sets of stop enrichments, $\theta = (e_1, \ldots, e_L)$, and a sequence of sets of move enrichments, $\phi = (f_1, \ldots, f_{L-1})$. Therefore, a trajectory individual t induces a pair $\Sigma = ((\sigma, \mu), (\theta, \phi))$, which we call the *semantic trajectory* induced by t. A formalization of the notion of semantic trajectory is given in Section 6.3.

In addition to the classes and binary relationships of the core model, the extended model has two classes, *Begin* and *End*, and four binary relationships, *has*, *nextS*, *nextM* and *ends*. These classes and binary relationships are introduced by definition as follows:

D1. *Begin* is the set of begin stops of the trajectories.

D2. *End* is the set of the last stops in the stop sequences of the trajectories, called the *end stops*.

D3. *nextS* relates each pair of consecutive stops of the stop sequence of each trajectory,that is, *nextS* is the composition of the inverse of $from$ with *to*.

D4. *nextM* relates each pair of consecutive moves of the move sequence of each trajectory, that is, *nextM* is the composition of *to* with the inverse of $from$.

D5. *has* relates each trajectory to each stop of the stop sequence of the trajectory, and to each move of the move sequence of the trajectory.

D6. *ends* relates each trajectory to its end stop.

Figure 15 schematically illustrates a trajectory individual t in the extended model. A formalization of the extended model reduces to expressing definitions **D1**–**D6**, which is also included in Section 6.3. Finally, we stress that the additional classes and relationships of the extended model do not increase the expressiveness of the model, from the formal point of view, but they facilitate writing query expressions over semantic trajectories, as well as their SPARQL counterparts.

6.3 A Description Logic Formalization of Semantic Trajectories

This section details how to formalize the core and the extended trajectory models in Description Logic (DL). Also, it formally defines the notion of

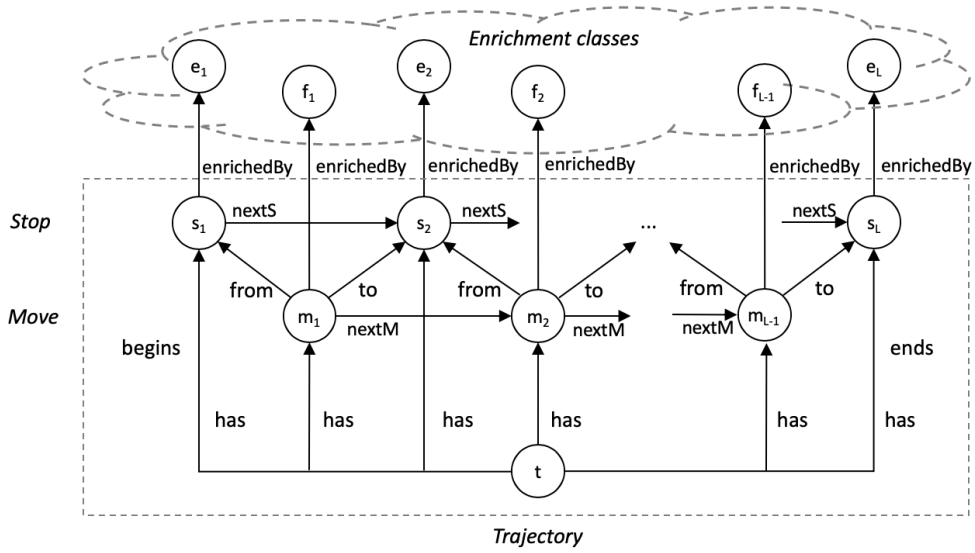


Figure 15: Schematic trajectory in the extended model

semantic trajectory induced by a trajectory individual.

In view of the definitions and concepts described in the previous section, the classes and binary relationships of the core and the extended models are accommodated by considering that alphabet $\mathcal{A}$ has five special atomic concepts: $Trajectory$, $Stop$, $Move$, $Begin$, and End; and eight special atomic roles: $enrichedBy$, has, $nextS$, $nextM$, $from$, to, $begins$, and $ends$. The special symbols are called the *trajectory symbols* of $\mathcal{A}$, and the other symbols the *enrichment symbols* of $\mathcal{A}$.

Table 15 lists the axioms of the core model, which correspond to assumptions **A1**–**A8**, and the definitions of the extended model, which correspond to definitions **D1**-**D6**. It also lists three other axioms, which are logical consequences of the axioms and definitions, and correspond to properties **P1** and **P2**. For simplicity, we assume that there is a super-class of the enrichment classes in $\mathcal{A}$, denoted $ENRICH$.

Let $\mathcal{A}$ be an alphabet and I an interpretation for $\mathcal{A}$, as in the previous section. Let $t \in Trajectory^I$. The *stop and move sequences* over I induced by t is the pair $\tau = (\sigma, \mu)$ such that:

- $\sigma = (s_1, \ldots, s_L)$ is the sequence of stops of I such that $(t, s_1) \in begins^I$, $(t, s_L) \in ends^I$, and $(s_i, s_{i+1}) \in nextS^I$, for each $i \in [1, L-1]$.
- $\mu = (m_1, \ldots, m_{L-1})$ is the sequence of moves of I such that $(m_j, s_j) \in from^I$, for each $j \in [1, L-1]$.

We say that L is the *length* of τ. Note that an empty trajectory is allowed, as well as a trajectory with just one stop, in which case μ is the empty sequence.

The *enrichment sets sequences* induced by t or simply *the enrichments of*

Table 15: List of axioms and definitions of the core and extended models

#	Formal description	Informal description
	Axioms of the Core Model	
1	$Trajectory \sqcap Stop \sqsubseteq \bot$	(**A1**) *Trajectory* and *Stop* are disjoint
2	$Trajectory \sqcap Move \sqsubseteq \bot$	(**A1**) *Trajectory* and *Move* are disjoint
3	$Stop \sqcap Move \sqsubseteq \bot$	(**A1**) *Stop* and *Move* are disjoint
4	$Trajectory \sqcup Stop \sqcup Move \sqsubseteq \neg ENRICH$	(**A2**) *Trajectory*, *Stop* and *Move* are disjoint from *ENRICH*, the super-class of all enrichments
5	$\exists enrichedBy \sqsubseteq Stop \sqcup Move$	(**A3**) The domain of *enrichedBy* is *Stop* union *Move*
6	$\exists enrichedBy^- \sqsubseteq ENRICH$	(**A3**) The range of *enrichedBy* is *ENRICH*
7	$\exists begins \sqsubseteq Trajectory$	(**A4**) The domain of *begins* is *Trajectory*
8	$\exists begins^- \sqsubseteq Stop$	(**A4**) The range of *begins* is *Stop*
9	$(> 1\ begins) \sqsubseteq \bot$	(**A4**) The cardinality of *begins* is at most 1
10	$(> 1\ begins^-) \sqsubseteq \bot$	(**A4**) The cardinality of $begins^-$ is at most 1
11	$\exists from \sqsubseteq Move$	(**A5**) The domain of *from* is *Move*
12	$\exists from^- \sqsubseteq Stop$	(**A5**) The range of *from* is *Stop*
13	$Move \sqsubseteq \exists from$	(**A5**) The cardinality of *from* is at least 1
14	$(> 1\ from) \sqsubseteq \bot$	(**A5**) The cardinality of *from* is at most 1
15	$(> 1\ from^-) \sqsubseteq \bot$	(**A5**) The cardinality of $from^-$ is at most 1
16	$\exists to \sqsubseteq Move$	(**A6**) The domain of *to* is *Move*
17	$\exists to^- \sqsubseteq Stop$	(**A6**) The range of *to* is *Stop*
18	$Move \sqsubseteq \exists to$	(**A6**) The cardinality of *to* is at least 1
19	$(> 1\ to) \sqsubseteq \bot$	(**A6**) The cardinality of *to* is at most 1
20	$(> 1\ to^-) \sqsubseteq \bot$	(**A6**) The cardinality of to^- is at most 1
21	$\exists from \equiv \exists to$	(**A7**) *from* is defined for a move iff *to* is also defined for that move
22	$\exists to^- \sqcap \exists begins^- \sqsubseteq \bot$	(**A8**) *to* does not map a move to the begin stop of a trajectory
	Definitions of the Extended Model	
23	$nextS \equiv (from^- \circ to)$	(**D3**) *nextS* is the composition of $from^-$ with *to*
24	$nextM \equiv (to \circ from^-)$	(**D4**) *nextM* is the composition of *to* with $from^-$
25	$has \equiv (begins \circ nextS^*) \sqcup (begins \circ from^- \circ nextM^*)$	(**D5**) *has* relates a trajectory to its stops and moves
26	$Begin \equiv \exists begins^-$	(**D1**) *Begin* is the set of begin stops of the trajectories
27	$End \equiv Stop \sqcap \exists has^- \sqcap \neg \exists from^-$	(**D2**) *End* is the set of end stops of the trajectories
28	$ends \equiv (begins \circ nextS^*) \sqcap \top \times End$	(**D6**) *ends* relates each trajectory to its end stop
	Logical consequences of the Extended Model	
29	$(> 1; has^-) \sqsubseteq \bot$	(**P1**) a stop or move belongs to at most one trajectory
30	$nextS^+ \sqcap \mathbb{I} \sqsubseteq \bot$	(**P2**) A trajectory has no repeated stops
31	$nextM^+ \sqcap \mathbb{I} \sqsubseteq \bot$	(**P2**) A trajectory has no repeated moves

t, are defined by the pair $\varepsilon = (\theta, \phi)$, where $\theta = (e_1, \ldots, e_L)$ is the sequence such that e_i is the set of pairs in $enrichedBy^I$ whose first element is s_i, called the *enrichments* of s_i in I, for $i = 1, \ldots, L$, and $\phi = (f_1, \ldots, f_{L-1})$ is the sequence such that f_j is the set of pairs in $enrichedBy^I$ whose first element is m_j, called the *enrichments* of m_j in I, for $j = 1, \ldots, L-1$.

Finally, the *semantic trajectory* over I induced by t is a pair $\Sigma = (\tau, \varepsilon)$ such that τ is the *stop and move sequences* over I induced by t and ε is the enrichment sets sequences induced by I. Note that ε is entirely determined by τ and the interpretation that I assigns to *enrichedBy*.

6.4 Query Expressions over Semantic Trajectories

This section defines a query language for semantic trajectory datasets that includes:

(1) stop and move queries that select a stop or a move based on its enrichments; and

(2) sequence expressions that define how to match the stop and move queries with the sequence of actions (i.e., stops or moves) defined in the semantic trajectory.

It first treats stop and move expressions as separated sequences, which is convenient from the formal point of view. Then, it introduces expressions that intercalate stop and move queries.

In what follows, let $\mathcal{A}$ be a DL alphabet and I be an interpretation for $\mathcal{A}$, satisfying the assumptions introduced in Section 6.2 and formalized in Section 6.3.

6.4.1 Enrichment, Stop, and Move Queries

An *enrichment query* is simply a concept expression C_i over the enrichment symbols of $\mathcal{A}$. A *stop query* over $\mathcal{A}$ is either one of the atomic concepts $Stop$, $Begin$, End, or a concept expression of the form

$$Stop \sqcap \exists enrichedBy.C_i \tag{26}$$

where C_i is an enrichment query. A stop query then defines the set of stops that have at least one enrichment that satisfies C_i. Indeed, the interpretation of $Stop^I$ is the set of stops of I, and the interpretation of $\exists enrichedBy.C_i$ in I is the set of individuals that $enrichedBy^I$ maps to some individual in ${C_i}^I$. Therefore, the interpretation of $Stop \sqcap \exists enrichedBy.C_i$ is the set of stops with an enrichment in ${C_i}^I$.

We recursively expand the set of stop queries to include *stop concept expressions* of the forms "$Q_i \sqcap Q_j$" and "$Q_i \sqcup Q_j$", where Q_i and Q_j are stop queries or stop concept expressions.

Likewise, a *move query* over $\mathcal{A}$ is either the atomic concept $Move$ or a concept expression of the form

$$Move \sqcap \exists enrichedBy.C_i \tag{27}$$

where C_i is an enrichment query. Again, we recursively expand the set of move queries to include *move concept expressions* defined as above.

The semantics of enrichment, stop, and move queries need not be explicitly defined, since it follows from the standard semantics of DL concept expressions, briefly summarized in Section 6.2.

6.4.2 Stop and Move Sequence Expressions

A *stop sequence expression* is a regular expression of stop queries, a *move sequence expression* is a regular expression of move queries, and a *stop/move sequence expression* is a pair (S_i, M_j), where S_i is a stop sequence expression and M_j is a move sequence expression.

More precisely, the set of *stop sequence expressions* over $\mathcal{A}$ is recursively defined as:

(1) The empty sequence λ is a stop sequence expression over $\mathcal{A}$.
(2) Any stop query of over $\mathcal{A}$ is a stop sequence expression over $\mathcal{A}$.
(3) If S_i is a stop sequence expression over $\mathcal{A}$, then (S_i), $S_i?$, S_i^*, and S_i^+ are also stop sequence expressions over $\mathcal{A}$.
(4) If S_i and S_j are stop sequence expressions over $\mathcal{A}$, then $(S_i \mid S_j)$ and $(S_i; S_j)$ are also stop sequence expressions over $\mathcal{A}$.

Parentheses may be omitted, if no ambiguity arises. The set of *move sequence expressions* over $\mathcal{A}$ is likewise defined, using move queries at the basis step.

The definition of the semantics of stop (or move) sequence expressions is simplified if we treat such expressions as specifying a set of sequences of stop (or move) queries. Recall that *Stop* is also a stop query that returns the set of all stops.

Let λ denote the empty sequence. Let $s_i; s_j$ denote the concatenation of two sequences s_i and s_j, with $s_i; s_j = s_i$, if $s_j = \lambda$, and $s_i; s_j = s_j$, if $s_i = \lambda$. Let s^n denote the n-fold concatenation $s; \ldots; s$ of s with itself, with $s^0 = \lambda$.

The *expansion of a stop* (or *move*) *sequence expression* S_i, denoted $expand(S_i)$, is a set of sequences of stop (or move) queries defined as follows:

(1) $expand(\lambda) = \emptyset$
(2) If S_i is a stop (or move) query Q_i, then $expand(S_i) = \{Q_i\}$
(3) If S_i is an expression of the form (S_j), then $expand(S_i) = expand(S_j)$
(4) If S_i is an expression of the form $S_j?$, then

$$expand(S_i) = \{\lambda\} \cup expand(S_j)$$

(5) If S_i is an expression of the form $(S_j \mid S_k)$, then

$$expand(S_i) = expand(S_j) \cup expand(S_k)$$

(6) If S_i is an expression of the form $(S_j; S_k)$, then

$$expand(S_i) = \{s_j; s_k \,/\, s_j \in expand(S_j) \wedge s_k \in expand(S_k)\}$$

(7) If S_i is an expression of the form ${S_j}^*$, then

$$expand(S_i) = \bigcup_{m \geq 0} expand(S_j)^m$$

(8) If S_i is an expression of the form ${S_j}^+$, then

$$expand(S_i) = \bigcup_{m > 0} expand(S_j)^m$$

Let $s = (s_1, \ldots, s_k)$ be a sequence with k elements, and i, j be two positive integers. Then, $segment(s, i, j) = (s_i, \ldots, s_j)$ is the segment of s starting at the i^{th} element and ending at the j^{th} element of s:

- if $1 \leq i \leq j \leq k$ or $1 \leq i \leq k < j \implies segment(s, i, j) = segment(s, i, k)$,
- if $k < i$ or $j < i \implies segment(s, i, j) = \lambda$

We extend the notion of segment to a trajectory $\tau = (\sigma, \mu)$, with length L, so that $segment(\tau, i, j) = (segment(\sigma, i, j), segment(\mu, i, j-1))$, for any two positive integers i and j. We say that a trajectory τ' is a *segment* of τ iff there are positive integers i and j such that $\tau' = segment(\tau, i, j)$.

Let I be an interpretation for $\mathcal{A}$ and $\tau = (\sigma, \mu)$ be a trajectory over I, with length L, where $\sigma = (s_1, \ldots, s_L)$ is a sequence of stops of I and $\mu = (m_1, \ldots, m_{L-1})$ is a sequence of moves of I. Let S_i be a stop (or move) sequence expression.

There are at least two options for the semantics of stop (or move) sequence expressions, depending on how S_i is evaluated against τ:

- *Strong semantics*, denoted $\tau \models_s S_i$, when S_i is evaluated from the beginning to the end of τ.
- *Weak semantics*, denoted $\tau \models_w S_i$, when S_i is evaluated against a segment of τ, which is required neither to start at the beginning of τ, nor to terminate at the end of τ.

This work adopts the weak version as the default semantics. However, note that one can force an expression S_i to be evaluated from the beginning to the end of the trajectories by using *Begin* and *End* at the beginning and at the end of S_i.

Strong satisfiability is defined as follows. If τ is a non-empty trajectory, then τ *strongly satisfies* S_i iff

- $S_i = \lambda$, or
- there is a sequence $Q_1; \ldots; Q_k$ in $expand(S_i)$ such that, for each $i \in [1, k]$, $s_i \in {Q_i}^I$ (or $m_i \in {Q_i}^I$, for move sequence expressions), and $k = L$, where

L is the length of the trajectory; in this case, we say that k is the *effective length* of S_i induced by its evaluation in τ.

If τ is the empty trajectory, then τ *strongly satisfy* S_i iff $S_i = \lambda$.

We say that τ *strongly satisfies* a stop/move sequence expression (S_i, M_j) iff τ strongly satisfies S_i, τ strongly satisfies M_j, and $k = l + 1$, where k is the effective length of S_i induced by its evaluation in τ and l is the effective length of M_j induced by its evaluation in τ.

We say that τ *weakly satisfies* S_i iff there is a segment τ' of τ such that $\tau' \models_s S_i$.

To conclude, the following examples illustrates the differences between strong and weak satisfiability. Under the notion of weak satisfiability, the queries in Section 6.1.2, with their intended interpretation, are expressions over the alphabet A_{Pisa}, where

- The terms `Museidipisa`, `Cappelledipisa`, `Torridipisa` and `Chiesedipisa` are atomic concepts of A_{Pisa}.
- The terms `Torre_pendente_di_pisa` and `Torre_del_Leone` are constants of A_{Pisa}.

Furthermore, the queries in Appendix C.1 must be rewritten as follows:

- Each constant a is replaced by the concept expression $\{a\}$.
- Each stop query E_i is replaced by the concept expression $Stop \sqcap \exists enrichedBy.E_i$, as in the example below, to conform with Eq. 26.
- Likewise, each move query E_i is replaced by the concept expression $Move \sqcap \exists enrichedBy.E_i$, to conform with Eq. 27.

For example, Queries Q1 and Q5 are formally rewritten as:

Q1: Find trajectories that stop at a museum and then at a chapel.

$$\texttt{Stop} \sqcap \exists\texttt{enrichedBy.Museidipisa};$$
$$\texttt{Stop} \sqcap \exists\texttt{enrichedBy.Cappelledipisa}$$

Q5: Find trajectories that begin at a museum and then end at a chapel.

$$\texttt{Begin} \sqcap \texttt{Stop} \sqcap \exists\texttt{enrichedBy.Museidipisa};$$
$$\texttt{End} \sqcap \texttt{Stop} \sqcap \exists\texttt{enrichedBy.Cappelledipisa}$$

Since *Begin* and *End* are specializations of *Stop*, the above expression can be simplified to:

$$\texttt{Begin} \sqcap \exists\texttt{enrichedBy.Museidipisa};$$
$$\texttt{End} \sqcap \exists\texttt{enrichedBy.Cappelledipisa}$$

By contrast, under the notion of strong satisfiability, the introduction of *Begin* and *End* becomes superfluous. For example, Q5 is formally rewritten as:

$$\texttt{Stop} \sqcap \exists \texttt{enrichedBy.Museidipisa};$$
$$\texttt{Stop} \sqcap \exists \texttt{enrichedBy.Cappelledipisa}$$

Note that the first formalization of Q1 and this second formalization of Q5 are syntactically identical, but they are interpreted under different semantics.

6.4.3 Intercalated Stop and Move Sequence Expressions

The definition of a stop/move sequence expression as a pair (S_i, M_j), where S_i is a stop sequence expression and M_j is a move sequence expression, is attractive from a formal point of view, but it may hide some complexities. Indeed, if a semantic trajectory τ satisfies (S_i, M_j), then the effective length of M_j must be one less than the effective length of S_i, by definition, to be able to intercalate the two sequences. But this requirement cannot be verified by a syntactical inspection of S_i and M_j, and is introduced only in the semantic notion of satisfiability.

For example, consider a stop/move sequence expression (F_1, G_1), where:

$$F_1 = p; Stop; r; s$$
$$G_1 = v^+; w$$

Since G_1 uses the "+" operator, it denotes sequences of move queries of arbitrary lengths, but only the sequence "$v; v; w$", which has a length equal to 3, could be properly intercalated with F_1, which has a length equal to 4.

We therefore define an *intercalated stop and move sequence expression* as an expression N_k of the form

$$N_k = S_0 < M_1 > S_1 < M_2 > S_2 \ldots S_{n-1} < M_n > S_n$$

where S_i is a stop sequence expression and M_j is a move sequence expression, for $i \in [0, n]$ and $j \in [1, n]$.

To define the semantics of intercalated stop and move sequence expression, we proceed as in Section 6.4.2, attaining only to weak satisfiability. The *expansion* of N_k, denoted $expand(N_k)$, is defined as for stop sequence expressions, but respecting the intercalation of stop and move expressions.

Let $E \in expand(N_k)$. Assume, without loss of generality, that E is of the form:

$$E = P_0 < Q_1 > P_1 < Q_2 > P_2 \ldots P_{n-1} < Q_n > P_n$$

where P_i is a sequence of stop queries and Q_j is a sequence of move queries, for $i \in [0, n]$ and $j \in [1, n]$.

Let $Stop^0 = \lambda$ and $Stop^n = Stop^{n-1}; Stop$, for $n \geq 1$, and likewise for $Move^n$. The *stop projection* of E is the sequence of stop queries F and the *move projection* of E is the sequence of move queries G such that:

$$F = P_0; Stop^{m_1}; P_1; Stop^{m_2}; P_2 \ldots P_{n-1}; Stop^{m_n}; P_n$$
$$G = Move^{s_0}; Q_1; Move^{s_1}; Q_2; Move^{s_2} \ldots Move^{s_{n-1}}; Q_n; Move^{s_n}$$

where m_j is the length of Q_j and s_i is the length of P_i, for $i \in [0, n]$ and $j \in [1, n]$.

An example of an intercalated stop and move sequence expression would be:

$$N_1 = (p \,|\, q) < v^* > r^+ < v^*; w > s$$

The following sequences pertain to $expand(N_1)$:

$$E_1 = p < v; v > r < w > s$$
$$E_2 = q < v > r; r < v; w > s$$

The stop projection of E_1 is F_1 and the move projection of E_1 is G_1, where:

$$\begin{aligned} F_1 &= p; Stop^1; r; Stop^0; s \\ &= p; Stop; r; s \\ G_1 &= Move^0; v; Move^0; v; Move^0; w \\ &= v; v; w \end{aligned}$$

The equalities follow if we observe that $Stop^0 = Move^0 = \lambda$. Note that if we intercalate F_1 and G_1 we obtain E_1 again:

$$p < v > Stop < v > r < w > s = p < v; v > r < w > s = E_1$$

Finally, let τ be a trajectory. If τ is the empty trajectory, then τ *does not weakly satisfy* N_k. If τ is a non-empty trajectory then τ *weakly satisfies* N_k iff there is $E \in expand(N_k)$ such that τ weakly satisfies (F, G), where F is the stop projection of E and where G is the move projection of E.

6.5 Extensions to Deal with Spatio-temporal Aspects

The concepts introduced in previous sections focus on the syntax and semantics of sequences of stop and move queries, exploring their enrichments. However, we may modify the formal framework to explore spatio-temporal aspects to:

(1) extend the concept of stop and move query;
(2) relate stops (or moves) with each-other;
(3) restrict trajectories;
(4) trajectories with each-other.

To extend stop/move queries, we first introduce a specific alphabet with atomic concepts and roles that capture spatio-temporal concepts and properties. Then, we extend the definitions of stop/move queries in Eqs. 26 and 27 by adding new concept expressions over this alphabet. The definitions of (intercalated) stop and move sequence expressions remain unchanged, however.

Informally, spatio-temporal restrictions that relate stops (or moves) with each-other would be, for example,"*consecutive stops that are less than 1.0 km apart*" or "*a sequence of three consecutive moves that take less than 30 minutes*". Since such restrictions involve more than one stop or move, they cannot be accommodated in the definition of stop (or move) query, which refers to individual stops (or moves). They can, however, be treated as restrictions on trajectories, that is, combined with the third type of restriction.

A spatio-temporal trajectory restriction may refer to the stops or moves of a trajectory, as just illustrated, or it may impose a restriction on the trajectory as a whole, without mentioning stops or moves. For example, one might require that a trajectory lies entirely within a given region. To accommodate such restrictions, we have to resort to new expressions that directly involve the class $Trajectory$, and new atomic concepts and roles that capture spatio-temporal trajectory properties.

The last type of restriction involves two or more trajectories, such as "*two trajectories that are never more than 1.0 km apart*". As in the previous case, we would have to resort to new atomic concepts and roles that capture spatio-temporal trajectory properties, relationships between trajectories based on their spatio-temporal properties, as well as other trajectory properties, such as who or what generated the trajectories.

Finally, spatio-temporal queries need no further comments, since they have been exhaustively discussed in the literature [17, 40, 51] and, in fact, are

part of the ISO SQL Standard[24] and the OGC GeoSPARQL standard[25]. Also, as mentioned before, the focus of this work is on the syntax and semantics of sequences of stop and move queries, exploring their enrichments.

[24]ISO/IEC 13249 - Part 3: Spatial, available at `https://www.iso.org/standard/60343.html`

[25]GeoSPARQL - A Geographic Query Language for RDF Data, available at `https://www.opengeospatial.org/standards/geosparql`

7
An RDF Framework for Querying Semantic Trajectories

Based on the formal framework of Chapter 6, this chapter introduces a concrete RDF framework for representing and querying semantic trajectories. Section 7.1 defines an RDF model for representing semantic trajectories. Section 7.2 introduces several classes of SPARQL query expressions over semantic trajectories. Section 7.3 discusses how to process SPARQL query expressions over semantic trajectories. Section 7.4 presents an algorithm to process keyword query expressions over semantic trajectories in RDF as a user-friendly alternative to SPARQL intercalated stop and move sequence expressions and, with the help of an example, discusses how to process such keyword query expressions. Finally, Section 7.5 provides a proof-of-concept to evaluate the algorithm proposed in Section 7.4.

7.1
An RDF Model for Semantic Trajectories

The proposed RDF model for semantic trajectories implements the formal model proposed in Section 6.3. The model is expressed as an RDF schema with the following classes and properties (see Figure 15):

- classes: `Trajectory`, `Stop`, `Move`, `Begin`, and `End`
- properties: `enrichedBy`, `from`, `to`, `nextS`, `nextM`, `has`, `begins`, and `ends`

and with declarations that capture Axioms (**1**-**28**) listed in Table 15.

The enrichment classes and properties are not part of the proposed RDF model for semantic trajectories, as highlighted in Section 6.2. We assume that they are defined in an external knowledge base.

7.2
SPARQL Query Expressions over Semantic Trajectories

This section first describes SPARQL stop (or move) queries and then SPARQL stop (or move) sequence expressions.

7.2.1 SPARQL Enrichment, Stop, and Move Queries

A *SPARQL enrichment query* is a SPARQL `SELECT` query over the enrichments knowledge base whose `TARGET` clause has a single variable and, thus, returns a set of IRIs that identify enrichments.

Figure 16 illustrates two SPARQL enrichment queries. The property function <`http://jena.apache.org/text#query`>, in the query of Figure 16(a), combines SPARQL and full text search via Lucene in Apache Jena. SPARQL enrichment queries such as these may, in fact, be automatically generated from keyword queries, as discussed in Section 7.4. We stress that a SPARQL enrichment query is not restricted to queries of the forms shown in Figure 16, but they can be any SPARQL query over the enrichments knowledge base, whose `TARGET` clause has a single variable.

```
1 select ?poi1
2 where {
3   ?poi1 text:query "torre pendente di pisa".
4 }
```

16(a) SPARQL query that returns the IRI of the Leaning Tower in Pisa

```
1 select ?poi2
2 where {
3   ?s1 rdfs:label "lb1".
4   ?s2 rdfs:label "pisa".
5   filter regex(?lbl,"museo").
6   ?poi2 :category ?s1.
7   ?poi2 :locatedIn ?s2.
8 }
```

16(b) SPARQL query that returns the IRIs of the museums in Pisa

Figure 16: Examples of two SPARQL enrichment queries

The rest of this section introduces the notion of SPARQL stop queries. The definition of SPARQL move queries is an exact parallel and is omitted.

Recall that a stop query is either *Stop*, *Begin*, *End*, or an expression of the form given in Eq. 26. In an exact parallel, a *SPARQL stop query* is a SPARQL select query of one of the forms:

- *Stop*, *Begin*, *End*

```
select ?v
 where{?v rdf:type C}
```

  where `C` is one of the class names `Stop`, `Begin`, or `End`.

- The equivalent of an expression of the form given in Eq. 26

```
select ?v
 where{?v rdf:type :Stop.
       ?v :enrichedBy ?p.
       { E[?p] }}
```

  where `E[?p]` denotes a SPARQL enrichment query `E` with the only variable in the `TARGET` clause replaced by `?p`.

Also, recall that we recursively expand the set of stop queries to include stop concept expressions of the forms "$Q_i \sqcap Q_j$" and "$Q_i \sqcup Q_j$", where Q_i and Q_j are stop queries or stop concept expressions. The equivalent SPARQL stop queries are of one of the forms:

- *Intersection*:

```
select ?v
 where{{ Q1[?v] }.
       { Q2[?v] } }
```

- *Union*:

```
select ?v
 where{ { Q1[?v] }
        UNION
        { Q2[?v] } }
```

where `Q1[?v]` and `Q2[?v]` denote the SPARQL stop queries `Q1` and `Q2` with the only variable in the `TARGET` clause replaced by `?v`.

7.2.2 SPARQL Stop and Move Sequence Expressions and SPARQL Intercalated Stop and Move Sequence Expressions

As in Section 6.4.2, a *SPARQL stop sequence expression* is a regular expression of SPARQL stop queries, a *SPARQL move sequence expression* is a regular expression of SPARQL move queries, and a *SPARQL stop/move sequence expression* is a pair (S_i, M_j), where S_i is a SPARQL stop sequence expression and M_j is a SPARQL move sequence expression. The notion of *SPARQL intercalated stop and move sequence expressions* is defined as in Section 6.4.3.

Finally, we introduce the notion of a *restricted SPARQL stop sequence expression*, defined exactly as a SPARQL stop sequence expression, except that expressions of the forms S_i^* and S_i^+ are allowed only when S_i is a SPARQL stop query (and not recursively a SPARQL stop sequence expression). The same holds for $< M_i^* >$ and $< M_i^+ >$. Likewise, a *restricted SPARQL intercalated*

stop and move sequence expression allows expressions of the forms S_i^* and S_i^+ only when S_i is a SPARQL stop query, and expressions of the forms $< M_i^* >$ and $< M_i^+ >$ only when M_i is a SPARQL move query.

7.3 Processing SPARQL Query Expressions over Semantic Trajectories

This section outlines how to process SPARQL query expressions over semantic trajectories. Section 7.3.1 discusses how to compile restricted SPARQL stop sequence expressions into SPARQL queries, whereas Section 7.3.2 outlines how to process unrestricted SPARQL stop sequence expressions. The processing of SPARQL move sequence expressions is entirely similar. Finally, Section 7.3.3 indicates how to extend the discussion of previous sections to cover the processing of SPARQL intercalated stop and move sequence expressions.

7.3.1 Compiling Restricted SPARQL Stop Sequence Expressions to SPARQL Queries

The compilation process recursively parses a restricted SPARQL stop sequence expression *Expr* and replaces each sub-expression of *Expr* by a SPARQL graph pattern that depends on the syntax of the sub-expression. The result is a SPARQL graph pattern, which is further post-processed to eliminate redundant triple patterns. The final SPARQL graph pattern is used to construct the `WHERE` clause of the SPARQL query **Q** that corresponds to *Expr*. The `TARGET` clause of **Q** is a list of three variables, `?t`, `?begin`, and `?end`. When executed, **Q** binds `?t` to a trajectory τ, and `?begin` and `?end` to stops s_B and s_E of τsuch that the segment of τ from s_B to s_E strongly satisfies *Expr* and, hence, τ weakly satisfies *Expr*. Section 7.4 contains a complete example of the compilation process.

The compilation process uses templates, which are expressions of the form:

`Template(`*Expr*`; ?t, ?begin, ?end)`

where *Expr* is a restricted SPARQL stop sequence expression, and `?t`, `?begin`, and `?end` are SPARQL variables. When called, the template expands to a schematic SPARQL graph pattern G, in the same way that a macro expands in traditional programming languages. The expansion process replaces the formal parameters by the concrete parameter values passed in the call and renames the other variables used in G to avoid conflicts. The schematic SPARQL graph pattern G may contain calls to other templates. When fully expanded, G results in a SPARQL graph pattern that:

- If *Expr* is not the empty stop sequence expression, G binds `?t` to a trajectory τ, and `?begin` and `?end` to a stops s_B and s_E of τ such that the sequence of consecutive stops of τ from s_B to s_E satisfies *Expr*.
- If *Expr* is the empty stop sequence expression, G binds `?begin` and `?end` to the stop before s_B, where s_B is the stop originally bound to `?begin`, since the empty stop sequence expression matches only the empty trajectory.

Recall that a stop SPARQL query `Q` has a single variable `?v` in the `TARGET` clause and let `Q[?u]` denote `Q` with `?v` replaced by `?u`. The templates are as follows:

(1) `Template("Lambda";?t,?begin,?end)`, where `"Lambda"` is the empty stop sequence expression

```
1 ?stop ^:nextS ?begin
2 bind(?stop as ?begin)
3 bind(?Stop as ?end)
```

Note: "`^:p`" denotes the inverse of property "`:p`" in SPARQL.

This graph pattern binds variables `?begin` and `?end` to the stop before s_B, where s_B is the stop originally bound to `?begin`, as already mentioned. This graph pattern is again used in the definition of the templates for the expressions `"S?"` and `"Q*"`.

(2) `Template(Q;?t,?begin,?end)`, where `Q` is a stop query

```
1 ?t :has ?begin.
2 { Q[?begin] }.
3 bind(?begin as ?end)
```

This graph pattern binds variable `?t` to a trajectory τ and variable `?begin` to a stop s_B of τ, tests if s_B satisfies `Q`, and binds `?end` to s_B.

Note: The actual implementation of the template inverts Line 1 with Line 2, for efficiency reasons.

(3) `Template("S1|S2";?t,?begin,?end)`

```
1 { Template("S1";?t,?begin;?end) }
2 UNION
3 { Template("S2";?t,?begin;?end) }
```

The recursive template calls in Lines 1 and 3 bind variable `?t` to a trajectory τ and variables `?begin` and `?end` to stops s_B and s_E of τ, respectively, such that the sequence of consecutive stops of τ from s_B to s_E satisfies `S1` or `S2`.

(4) `Template("S1;S2";?t,?begin,?end)`

```
1 ?endS1 :nextS ?beginS2.
2 { Template("S1";?t,?begin;?endS1) }.
3 { Template("S2";?t,?beginS2;?end) }
```

The recursive template calls in Lines 2 and 3 bind variable `?t` to a trajectory τ and variables `?begin`, `?endS1`, `?beginS2`, and `?end` to stops s_B, s_{E1}, s_{B2}, and s_E of τ, respectively, such that:
(i) the sequence of consecutive stops in τ from s_B to s_{E1} satisfies `S1`;
(ii) the sequence of consecutive stops in τ from s_{B2} to s_E satisfies `S2`;
(iii) s_{E1} and s_{B2} are consecutive stops of τ (by Line 1).

(5) `Template("S?";?t,?begin;?end)`

```
1 OPTIONAL
2 { Template("S";?t,?begin,?end) }
```

This graph pattern captures the fact that the expression `"S?"` is equivalent to the expression `"(Lambda|S)"`, where `"Lambda"` is the empty stop sequence expression.

(6) `Template("Q+";?t,?begin,?end)`

```
1 ?t :has ?begin.
2 ?begin :nextS* ?end.
3 { Q[?begin] }.
4 { Q[?end] }.
5 filter not exists {
6   ?begin :nextS* ?stopM.
7   ?stopM :nextS* ?end.
8   filter not exists { Q[?stopM] } }
```

This graph pattern binds variable `?t` to a trajectory τ and variables `?begin` and `?end` to stops s_B of s_E of τ such that all consecutive stops from s_B to s_E in τ satisfy `Q`, including s_B and s_E.
Notes:

(a) The actual implementation of the template places Lines 3 and 4 before Lines 1 and 2, for efficiency reasons.
(b) This template applies only when `Q` is a SPARQL stop query, and not a SPARQL stop sequence expression, in view of the use of the SPARQL path expression "`:nextS*`".
(c) Lines 5 to 7 explore the fact that $\forall x(Q)$ is equivalent to $\neg\exists x\neg(Q)$. Thus, the sentence "*for any stop s between s_1 and s_2, s satisfies Q*" is equivalent to "*there is no stop s between s_1 and s_2 such that s does not satisfy Q.*"

(7) `Template("Q*";?t,?begin,?end)`

```
1 OPTIONAL
2 { Template("Q+";?t,?begin,?end) }
```

This graph pattern captures the fact that the expression `"Q*"` is equivalent to the expression `"(Lambda | Q+)"`, where again `"Lambda"` is the empty stop sequence expression.

The final SPARQL graph pattern is subjected to a simplification process, where some triple patterns are eliminated, based on the axioms listed in Table 15. Section 7.4 contains an example that that illustrates how to compile a restricted SPARQL stop sequence expression to an equivalent SPARQL query, and how to simplify the query.

7.3.2 Processing Unrestricted SPARQL Stop Sequence Expressions

In view of the complexity of extending the templates to cover expressions of the forms $\mathtt{S}_i^+$ and $\mathtt{S}_i^*$, when $\mathtt{S}_i$ is not a SPARQL stop query, we adopt a different strategy and outline an interpreter for unrestricted SPARQL stop sequence expressions.

Recall that the semantics of a stop sequence expression S_i, defined in Section 6.4.2, is based on the expansion of S_i into a set $expand(S_i)$ of sequences of stop queries. We may likewise define the *expansion* of a SPARQL stop sequence expression $\mathtt{S}_i$ into a set $expand(\mathtt{S}_i)$ of sequences of SPARQL stop queries.

Very briefly, given a SPARQL stop sequence expression $\mathtt{S}_i$ and an RDF dataset $\mathcal{R}$ containing a set of trajectories and their enrichments, as in Section 7.1, the interpreter proceeds as follows:

1. Create $expand(\mathtt{S}_i)$, the expansion of $\mathtt{S}_i$;
2. Order the sequences in $expand(\mathtt{S}_i)$ by increasing length, creating a list L;
3. For each SPARQL stop query sequence $\mathtt{S}_j$ in L, up to a certain maximum length μ, do:
   (a) Translate $\mathtt{S}_j$ into a SPARQL query $\mathtt{P}_j$, using the corresponding templates in Section 7.3.1;
   (b) Execute $\mathtt{P}_j$ against $\mathcal{R}$;
   (c) If the result is non-empty, return the trajectories retrieved by $\mathtt{P}_j$;

Note that Step (3) limits the length of the SPARQL stop query sequences to μ, for practical reasons. The choice of μ is application-dependent. Further-

more note that Step 3(a) is indeed possible, since $\texttt{S}_j$ is a sequence of SPARQL stop queries, which is covered by the templates in Section 7.3.1.

7.3.3 Processing SPARQL Intercalated Stop and Move Sequence Expressions

Recall that a restricted SPARQL intercalated stop and move sequence expression allows expressions of the forms S_i^* and S_i^+ only when S_i is a SPARQL stop query, and expressions of the forms $<M_i^*>$ and $<M_i^+>$ only when M_i is a SPARQL move query.

Let `S1` and `S2` be restricted SPARQL stop sequence expressions and hence have associated templates, as in Section 7.3.1. Let `M`, `M1`, and `M2` be SPARQL move queries. Recall that `M` has a single variable in the `TARGET` clause, and let `M[?u]` denote `M` with this single variable replaced by `?u` (and likewise for `M1` and `M2`). The templates that follow are not an exhaustive list, but illustrate the extension process.

(8) `Template("S1<M>S2",?t,?begin,?end)`

```
1 Template("S1",?t,?begin,?endS1) .
2 Template("S2",?t,?beginS2,?end) .
3 ?move :from ?endS1; :to ?beginS2 .
4 { M[?move] }
```

The recursive template calls in Lines 1 and 2 bind variable `?t` to a trajectory τ and variables `?begin`, `?endS1`, `?beginS2`, and `?end` to stops s_B, s_{E1}, s_{B2}, and s_E of τ, respectively, and Line 3 binds variable `?move` to a move m of τ, such that:
(1) the sequence of consecutive stops in τ from s_B to s_{E1} satisfies `S1`;
(2) the sequence of consecutive stops in τ from s_{B2} to s_E satisfies `S2`;
(3) m is from s_{E1} to s_{B2} (by Line 3);
(4) m satisfies `M` (by Line 4).

(9) `Template("S1<M1|M2>S2",?t,?begin,?end)`

```
1 Template("S1";?t,?begin,?endS1) .
2 Template("S2";?t,?beginS2,?end) .
3 ?move :from ?endS1; :to ?beginS2 .
4 {{ M1[?move] }
5  UNION
6  { M2[?move] }}
```

The recursive template calls in Lines 1 and 2 bind variable `?t` to a trajectory τ and variables `?begin`, `?endS1`, `?beginS2`, and `?end` to stops s_B, s_{E1}, s_{B2}, and s_E of τ, respectively, and Line 3 binds variable `?move`

to a move m of τ, such that:
(1) the sequence of consecutive stops in τ from s_B to s_{E1} satisfies `S1`;
(2) the sequence of consecutive stops in τ from s_{B2} to s_E satisfies `S2`;
(3) m is from s_{E1} to s_{B2} (by Line 3);
(4) m satisfies either `M1` or `M2` (by Lines 4-6).

(10) `Template("S1<M1;M2>S2",?t,?begin,?end)`

```
1 Template("S1",?t,?begin,?endS1) .
2 Template("S2",?t,?beginS2,?end) .
3 ?move1 :from ?endS1; :to ?stop2 .
4 ?move2 :from ?stop2; :to ?beginS2 .
5 { M1[?move1] } .
6 { M2[?move2] }
```

The recursive template calls in Lines 1 and 2 bind variable `?t` to a trajectory τ and variables `?begin`, `?endS1`, `?beginS2`, and `?end` to stops s_B, s_{E1}, s_{B2}, and s_E of τ, respectively, and Lines 3 and 4 bind variables `?move1` and `?move2` to moves m_1 and m_2 of τ, respectively, such that:
(1) the sequence of consecutive stops in τ from s_B to s_{E1} satisfies `S1`;
(2) the sequence of consecutive stops in τ from s_{B2} to s_E satisfies `S2`;
(3) m_1 is from s_{E1} to a stop s_2 and m_2 from s_2 to s_{B2} (by Lines 3 and 4);
(4) m_1 satisfies `M1` and m_2 satisfies `M2` (by Lines 5 and 6).

(11) `Template("S1<M+>S2";?t,?begin,?end)`

```
1  Template("S1";?t,?begin,?endS1) .
2  Template("S2";?t,?beginS2,?end) .
3  ?moveB :from endS1 .
4  ?moveE :to   beginS2 .
5  ?moveB :nextM* ?moveE .
6  { M[?moveB] } .
7  { M[?moveE] } .
8  filter not exists {
9    ?moveB :nextM* ?moveM .
10   ?moveM :nextM* ?moveE .
11   filter not exists { M[?moveM] } }
```

The recursive template calls in Lines 1 and 2 bind variable `?t` to a trajectory τ and variables `?begin`, `?endS1`, `?beginS2`, and `?end` to stops s_B, s_{E1}, s_{B2}, and s_E of τ, respectively, and Lines 3 and 4 bind variables `?moveB` and `?moveE` to moves m_B and m_E of τ, respectively, such that:
(1) the sequence of consecutive stops in τ from s_B to s_{E1} satisfies `S1`;
(2) the sequence of consecutive stops in τ from s_{B2} to s_E satisfies `S2`;
(3) m_B is from s_{E1} and m_E is to s_{B2} (by Lines 3 and 4);
(4) all moves in the sequence of moves of τ from m_B to m_E satisfy `M` (by Lines 5-11).

Finally, the processing of unrestricted SPARQL intercalated stop and move sequence expressions is covered via a modification of the interpreter defined in Section 7.3.2.

7.4 Keyword Query Expressions over Semantic Trajectories

The definitions that follow are similar to those in Sections 6.4 and 7.2. A *keyword stop query* is a finite set $K = \{k_1, \ldots, k_n\}$ of literals, called *keywords*, that defines a set of stops based on their enrichments. These queries would then be applied to the RDF knowledge base to select a set R of enrichments, which are then used to select the set of stops that are related to the enrichments in R by the *enrichedBy* property. A *keyword move query* is likewise defined.

Schema-based algorithms, as [24, 33], can then be used to translate keyword stop (or move) queries into SPARQL queries that retrieve resources by their name, such as "*Torre Pendente di Pisa*", or by their attributes, such as "*Musei di Pisa*". In the first case, the query would retrieve a single POI id that corresponds to the *Leaning Tower of Pisa* (see Figure 16(a)), while in the second case it would return a list of ids that correspond to the museums in Pisa (see Figure 16(b)). One can relax the scope of the query by informing only some keywords, such as "*Torre di Pisa*", in which case the keyword query could return the ids of the "*Torre Pendente di Pisa*" and "*Ristorante La Torre Pisa*".

Keyword stop sequence expressions, *keyword move sequence expressions*, *keyword stop/move sequence expressions*, and *keyword intercalated stop and move sequence expressions* are defined as in Section 7.2, except that they are based on keyword stop queries and keyword move queries. The regular expression symbols may be replaced by the reserved terms listed in Table 13.

The processing of a keyword intercalated stop and move sequence expression N to SPARQL has three basic steps:

1. Translate N into a SPARQL intercalated stop and move sequence expression S:
   (a) If necessary, replace the reserved terms "`Stop`", "`Move`", "`Begin`" and "`End`" by SPARQL queries, as discussed in Section 7.2.1.
   (b) Also, if necessary, replace the reserved terms that denote regular expressions by equivalent symbols, using Table 13.
   (c) Translate each keyword stop (or move) query into a SPARQL query (Section 7.3.1).
2. Process S as discussed in Section 7.3.3.

3. Build the `TARGET` clause of S with variable ?t that binds the queried trajectories.

We exemplify how to process the query Q9 in Appendix C.2 (repeated below for ease of reference):

> "*Find the trajectories that begin at a chapel or a church, always move by bus between stops, and end at the Leaning Tower*"

using the algorithm just described.

Let N be the corresponding keyword intercalated stop and move expression and assume that N uses the symbolic notation and the *TripBuilder* terms of Tables 13 and 14:

```
(Begin ⊓ (Cappelledipisa | Chiesedipisa) ) <Bus+>
          (Torre_pendente_di_pisa ⊓ End)
```

In this example, we adopt Jena ARQ[26] as the SPARQL query engine. Then, the processing of N goes as follow:

- **Step 1**: Translate the keyword queries "`Cappelledipisa`", "`Chiesedipisa`", and "`Torre_pendente_di_pisa`" to SPARQL enrichment queries, as shown in Figure 17, to retrieve the resources associated with these POIs.

```
select ?v1
where { ?v1 text:query "(Cappelledipisa)" }
```

17(a) SPARQL enrichment query for "`Cappelledipisa`"

```
select ?v2
where { ?v2 text:query "(Chiesedipisa)" }
```

17(b) SPARQL enrichment query for "`Chiesedipisa`"

```
select ?v3
where { ?v3 text:query "(Torre_pendente_di_pisa)" }
```

17(c) SPARQL enrichment query for "`Torre_pendente_di_pisa`"

Figure 17: The SPARQL enrichment queries of the keyword queries in S

Note that variables `?v1`, `?v2`, and `?v3` bind the IRIs of POIs resources associated with "`Cappelledipisa`", "`Chiesedipisa`", and "`Torre_pendente_di_pisa`", respectively.

[26]https://jena.apache.org/documentation/query/index.html

- **Step 2**: Process the resulting SPARQL intercalated stop and move sequence expression S.

  ◇ The process recognizes that S satisfies Template (11):

  `Template("S1<M+>S2";?t,?begin,?end)`

  where:

  * `S1` is the SPARQL query corresponding to `"Begin ⊓ S3"`;
  * `S3` is the SPARQL query corresponding to `"(Cappelledipisa | Chiesedipisa)"`;
  * `S2` is the SPARQL query corresponding to `"S4 ⊓ End"`;
  * `S4` is the SPARQL query corresponding to `"Torre_pendente_di_pisa"`;
  * `M` is the SPARQL query corresponding to `<Bus>`.

  Note that `S3` satisfies `Template` (3).

  ◇ By combining the different templates, the final template for S is (edited for legibility):

```
### Stop query S1
  ?t :has ?begin .
  ### Stop query for "Begin"
  ?begin rdf:type Begin .
  ### Stop query for "Cappelledipisa or Chiesedipisa"
  {
    {
      ### Stop query for "Cappelledipisa"
      ?begin rdf:type Stop .
      ?begin :enrichedBy ?v1 .
      ?v1 text:query "Cappelledipisa"
    }
   UNION
    {
      ### Stop query for "Chiesedipisa"
      ?begin rdf:type Stop .
      ?begin :enrichedBy ?v2 .
      ?v2 text:query "Chiesedipisa"
    }
  }

```

```
### Stop query S2
  ?t :has ?end .
  ### Stop query for "Torre Pendente di Pisa"
  {
    ?end rdf:type Stop .
    ?end :enrichedBy ?v3 .
    ?v3 text:query "Torre_pendente_di_pisa"
  }
  ### Stop query for "End"
  ?end rdf:type End .

### Template (11)
  ?moveB :from ?begin .
  ?moveE :to   ?end .
  ?moveB :nextM* ?moveE .
  ### Move query corresponding to "by bus" for ?moveB
    ?moveB :type Move .
    ?moveB :enrichedBy ?transpB
      filter (?transpB = :Bus)
  ### Move query corresponding to "by bus" for ?moveE
    ?moveE :type Move .
    ?moveE :enrichedBy ?transpE
      filter (?transpE = :Bus)
  ### "by bus" one or more times
    filter not exists {
    ?moveB :nextM* ?moveM .
    ?moveM :nextM* ?moveE .
        filter not exists {
              ?moveM :type Move .
              ?moveM :enrichedBy ?transpM
               filter (?transpM = :Bus)
        }
    }
```

- **Step 3**: The compilation process ends by setting the `TARGET` clause (as a `SELECT` form) with the variables `?t`, which binds the queried trajectories, `?begin` and `?end`, as stated in Section 7.3.1, and applying some simplifications (indicated after the query).

  The final synthesized SPARQL query is:

```
select ?t, ?begin, ?end
where{
  ?t :begins ?begin .
  {
    {
      ?begin :enrichedBy ?v1 .
      ?v1 text:query "Cappelledipisa"
```

```
8       }
9      UNION
10      {
11        ?begin :enrichedBy ?v2 .
12        ?v2 text:query "Chiesedipisa"
13      }
14    }
15    ?t :ends ?end .
16    {
17      ?end :enrichedBy ?v3 .
18      ?v3 text:query "Torre_pendente_di_pisa"
19    }
20    ?moveB :from ?begin .
21    ?moveE :to   ?end .
22    ?moveB :nextM* ?moveE .
23    ?moveB :enrichedBy ?transpB
24        filter (?transpB = :Bus)
25    ?moveE :enrichedBy ?transpE
26        filter (?transpE = :Bus)
27    filter not exists {
28      ?moveB :nextM* ?moveM .
29      ?moveM :nextM* ?moveE .
30        filter not exists {
31                              ?moveM :enrichedBy ?transpM
32                                filter (?transpM = :Bus)
33                             }
34                    }
35  }
```

The simplifications applied and the axioms that justify them were:

- Replace `"?t :has ?begin"` and `"?begin rdf:type Begin"` by `"?t :begins ?begin"` - Axiom (26).
- Replace `"?t :has ?end"` and `"?end rdf:type End"` by `"?t :ends ?end"` - Axioms (27-28).
- Drop `"?moveB :type Move"`, since `"?moveB :from stop1"` occurs - Axiom (11).
- Drop `"?moveE :type Move"`, since `"?moveE :to stop2"` occurs - Axiom (16).
- Drop `"?moveM :type Move"`, since `"?moveB :nextM* ?moveM"` occurs - Axioms (11, 16, 24).

7.5 A Proof-of-Concept Experiment

This section first describes the construction of the *TripBuilder RDF dataset* and then reports experiments that explore how to compile a set of keyword stop and move sequence expressions into SPARQL.

7.5.1 The Use-Case Trajectory Dataset in RDF

As stated in Section 6.1.1, we selected the *TripBuilder* dataset to run the proof-of-concept experiment. The original dataset is openly available at `https://github.com/igobrilhante/TripBuilder` and contains trajectory data of three Italian cities: Pisa, Florence and Rome. The basic idea behind *TripBuilder* is to recognize that a tourist moves around a city taking many photos of various POIs, and thereby his photos are a good indication of his trajectory. The construction of TripBuilder then started by clustering users' photos collected by Flickr, using the spatial dimension, and then relating each cluster to places-of-interest (POIs). For each city, the data are organized in four files in a directory identified by the name of the city.

For the construction of the *TripBuilder RDF dataset*, we wrote a Java program that parses the TripBuilder city data files, extracts data for POIs, stops, and trajectories and triplifies the resulting data in RDF, following the model specified in Section 7.1. In special, the triplification is such that it results in an RDF dataset that satisfies the axioms listed in Table 15. We also included two new properties and a new class:

(1) The property `lenght`, added to the instances of the `Trajectory` class, indicates the length of a trajectory.

(2) The property `move_number`, added to the instances of the `Move` class, indicates the sequential position of a move in a trajectory.

(3) The class `Transportation` whose resources were automatically generated, labeled with a value in the set {"Walk", "Taxi", "Bus", "Subway"}, and randomly linked to resources of the `Move` class using the property `enrichedBy`.

This new class helps exploit the capabilities of the translation algorithm, given that the original *TripBuilder dataset* does not contain information about moves.

Table 16 shows the main statistics of the *TripBuilder RDF dataset*. The resulting RDF dataset contains a total of 1,617,582 triples, which break down to 5 `rdfs:Class declarations`, 255,018 class instances (47% of them corresponding to stops, 30% to moves and 21.5% to trajectories) and 1,973 indexed

property values; Data are available for download at `https://figshare.com/s/92332f0cbb8d591021ed`.

Table 16: Statistics about the *TripBuilder RDF Dataset*

Triple Types	# of Triples
`rdfs:Class` declarations	5
Class instances	255,018
- For class `:POI`	1,603
- For class `:Stop`	120,322
- For class `:Trajectory`	55,474
- For class `:Transportation`	4
- For class `:Move`	77,615
- Moves `enrichedBy :Walk`	19,577
- Moves `enrichedBy :Bus`	19,441
- Moves `enrichedBy :Taxi`	19,436
- Moves `enrichedBy :Subway`	19,161
Datatype properties	7
String (Indexable) datatype properties	3
Distinct indexed property values	1,973
Total of triples	**1,617,582**

We stored the RDF dataset on a Jena ARQ SPARQL server (running on a quad-core processor Intel(R) Core(TM) i7-5820K CPU@3.30GHz, 64 GB of RAM, and SSD 1TB, with GNU/Linux Ubuntu 16.04.6 LTS OS). The string property values, including `rdfs:label` values, were indexed using Lucene[27], which is hosted with Jena. This feature allows combining SPARQL queries and full-text search.

7.5.2 Experiments with a Sample Set of Keyword Query Expressions

The experiments consisted of applying the approach described in Section 7.4 using the set of keyword query expressions listed in Appendix C. Recall that Appendix C lists a sample set of queries for stop-and-moves sequence expressions: Queries Q1 to Q7 are stop sequence expressions, and Queries Q8 to Q10 are examples of intercalated stop and move sequence expressions.

We analyzed the results of the proposed translation algorithm with respect to two aspects: (1) the templates that the SPARQL stop and move sequence expressions satisfy; and (2) the average execution time of 10 repetitions of each synthesized SPARQL query. For each keyword query expressions

[27] http://lucene.apache.org/

of the test suite, Table D.1 (in Appendix D) shows the compiled SPARQL query (edited for readability) and Figure 18 indicates the average runtime.

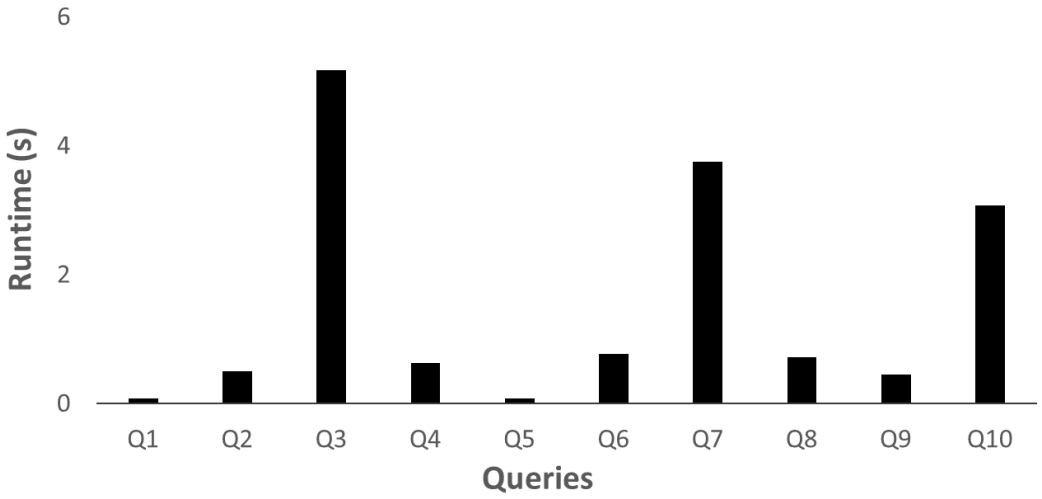


Figure 18: The runtime of the compiled SPARQL queries

The results can be summarized as:

- Q1 is a simple stop sequence that satisfies `Template` (4). As Table D.1 shows, the runtime of the synthesized SPARQL query is fast.
- Q2 combines `Template` (4) and `Template` (3). As Table D.1 shows, the runtime of the synthesized SPARQL query is reasonable.
- Q3 combines `Template` (4) and `Template`(6). The runtime of the synthesized SPARQL query is considerably higher than that for Q2. The sharp increase in runtime is due to the presence of nested `filter not exists` group patterns and the use of the SPARQL Property Path[28] operator "*" inside the patterns to solve the stop query $\texttt{Torridipisa}^{+}$.
- Q4 combines `Template` (4) into `Template` (3). The runtime of the synthesized SPARQL query is fast since the query has few joins.
- Q5 adds the *Begin* and *End* restrictions to Q1. The SPARQL query synthesized for Q5 then replaces the triple patterns `{?t :has ?stop1; :has ?stop2}` in Q1 by `{?t :begins ?stop1; :ends ?stop2}`. Then, the runtime of the synthesized SPARQL query is similar to Q1.
- Q6 uses the `OPTIONAL` and `UNION` patterns to capture "`(Cappelledipisa | Chiesedipisa)?`". The runtime of the synthesized SPARQL query is reasonable; approximately 1 second on average.
- Q7 has *Begin* and *End* restrictions and can be rewritten as

[28]https://www.w3.org/TR/sparql11-property-paths/

`Begin` ⊓ `Cappelledipisa`$^{+}$; `Cappelledipisa` ⊓ `End`.

Hence, this stop sequence expression is similar to Q3, combining `Template` (4) and `Template` (6). Also, as Q3, the runtime of the synthesized SPARQL query is significantly high, but still reasonable.

– Q8 directly follows `Template` (8), which the algorithm applies to synthesize the SPARQL query in a straightforward way. The runtime of the synthesized SPARQL query, about 1 second on average, is reasonable.

- As stated in Section 7.4, Q9 combines `Template` (11) and `Template` (3). Unexpectedly, the runtime of the synthesized SPARQL query is pretty fast.
- Q10 has a complex graph pattern. It combines `Template` (10) and `Template` (8). Note that the stop and move sequence "`Cappelledipisa <Move> Torredipisa`" is equivalent to the stop sequence "`Cappelledipisa ; Torredipisa`" since there is no restriction on the transportation means in the move sequence. Hence, the triple pattern "`?move3 :from ?stop2; :to ?stop3`" (see row for Q10 of Table D.1) can be replaced by "`?stop2 :nextS ?stop3`". The runtime of the synthesized SPARQL query, about 3 seconds, is high, but still reasonable.

To conclude, we observe that the complexity of the SPARQL query synthesized in each case naturally reflects the complexity of the keyword query expressions. As expected, queries with triple patterns with the property path operator "*" inside nested "`filter not exists`" group patterns (Q3, Q7), or with complex graph patterns (Q10) had a high runtime. However, unexpectedly, even with a complex graph pattern, Q9 had a small runtime. Queries with less complex graph patterns (Q1, Q2, Q4, Q5, Q8), or with just `UNION` or `OPTIONAL` patterns (Q6) had acceptable runtime. Overall, all queries were executed within the specified timeout of 1 minute. The average runtime of the test suite queries, about 1.5s, was reasonable. Hence, the experiment suggests that the proposed approach, based on keyword query expressions and RDF, is feasible.

8
Conclusions and Future Work

8.1
About the Keyword Search over Schema-less RDF Datasets Problem

To address the keyword search over schema-less RDF datasets problem, this thesis introduced a novel algorithm to automatically translate a user-specified keyword-based query into a SPARQL query that returns answers with respect to the keywords. The algorithm synthesizes the SPARQL query by exploring the Jaccard and set containment similarity measures between the property domains and ranges and class instance sets, observed in the RDF dataset. The algorithm estimates these similarity measures using KMV-synopses, which can be pre-computed in a single pass over the RDF dataset.

The thesis then described two sets of experiments with an implementation of the proposed algorithm, which we called the KMV–synopses RDF keyword search tool, or simply the KMV-synopses tool. The first set of experiments compared the KMV–synopses tool with a baseline schema-based tool [24, 33] over a benchmark. We were interested in testing if schema information could be replaced by KMV–synopses, without impacting performance. The experiments showed that the KMV–synopses tool outperformed the baseline tool in all metrics adopted, which shows that the lack of schema information can indeed be replaced by pre-computed, concise KMV–synopses for the property domains and ranges and class instance sets. Also, the average elapsed times of the baseline tool and the KMV–synopses tool were similar, which indicates that estimating set similarity based on KMV–synopses does not introduce significant overhead, even for large RDF datasets such as IMDb, if the KMV-synopses are pre-computed. The second set of experiments indicated that the KMV-synopses tool performed better than the state-of-the-art TSA+BM25 and TSA+VDP keyword search systems over RDF datasets based on the "virtual documents" approach, using the metrics and the benchmarks proposed originally to assess these systems [18].

Finally, the thesis proposed the *Graph Relevance Ratio* (*GRR*) to establish when an answer graph is relevant w.r.t. a ground truth graph. It is based on the number of relevant and non-relevant triples in the RDF graph, but

it punishes the presence of non-relevant triples, and does not memorize the relevant triples in previous rank positions.

As future work, we suggest to investigate indexes for the KMV–synopses to optimize the KMV–synopses tool, and methods to maintain KMV–synopses incrementally [34]. Also, we suggest to combine the strategy described with schema information to drive the query compilation process. The KMV–synopses will be used to help the query compilation process in much the same way as the usual database statistics help the query optimization process. Finally, we also plan to adapt the proposed strategy to operate over relational and RDF datasets, such as our earlier keyword search tool [33] that is schema-based.

8.2 About the Keyword Search over RDF Semantic Trajectories Problem

To address the keyword search over RDF semantic trajectories problem, this thesis defined a query language that includes: (1) stop and move queries that select sets of stops or moves based on their enrichments; and (2) sequence expressions that define how to match the stop and move queries with the sequence of actions defined in the semantic trajectory.

Based on Description Logic, the thesis first introduced a formal model for semantic trajectories and defined stop and move sequence expressions, with well-defined syntax and semantics, which act as an expressive query language for semantic trajectories. Then, it moved to a concrete semantic trajectory model in RDF and described how to process SPARQL stop and move sequence expressions, using state-of-the-art, efficient SPARQL query processors. The adoption of RDF is a natural choice, given that one can take advantage of several open-access knowledge bases in RDF to enrich trajectories. Next, the thesis defined a user-friendly way to express stop and move sequence expressions, based on the use of keywords to capture stop and move queries, and the adoption of terms with predefined semantics to define sequence expressions. It briefly indicated that such keyword stop and move sequence expressions can be compiled into SPARQL queries. This final approach aims at hiding the complexities of writing SPARQL queries, and yet it permits the use of SPARQL query processors. Finally, the thesis described a proof-of-concept experiment using the TripBuilder dataset, a semantic trajectory dataset constructed from user-generated content obtained from Flickr, combined with data from Wikipedia. These contributions were reported in [35].

As future work, we plan to run large-scale experiments with real-world

semantic trajectory datasets, backed up by a robust implementation of the keyword stop and move sequence expression SPARQL compiler. A target application would be related to investigations of cargo vessel incidents. The stop and move sequence expressions introduced in thesis work would help, for example, locate vessel trajectories that match disallowed movement patterns, such as "trajectories of oil tankers that sailed from any oil rig port in country A, sailed through a high risk region (e.g. a piracy prone area) and arrived at a port P in another country B". Such trajectories are sometimes followed by captains at the risk of not been covered by insurance companies in the case of an attack.

Stop and move sequence expressions may also be used to query different types of trajectories, such as play lists in a musics database, or navigation patterns of interest in an e-shop: "customers who visited the product page of a tv-set A (an analogy of stop), then followed a link (a type of move) to another tv-set B (another stop of the same type) and later on their session searched (i.e., a different type of "move" within a web site) for a console C".

Finally, we will invest in extending the approach to a question-and-answer (Q&A) scenario.

Bibliography

[1] ADITYA, B.; BHALOTIA, G.; CHAKRABARTI, S.; HULGERI, A.; NAKHE, C.; SUDARSHANXE, S. ; OTHERS. **Banks: Browsing and keyword searching in relational databases**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES (VLDB'02), p. 1083–1086, Hong Kong SAR, China, 2002. Elsevier.

[2] AGRAWAL, S.; CHAUDHURI, S. ; DAS, G.. **Dbxplorer: A system for keyword-based search over relational databases**. In: PROCEEDINGS OF THE 18TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE '02), p. 5–16, 2002.

[3] ALVARES, L. O.; BOGORNY, V.; KUIJPERS, B.; DE MACEDO, J. A. F.; MOELANS, B. ; VAISMAN, A.. **A model for enriching trajectories with semantic geographical information**. In: PROCEEDINGS OF THE 15TH ANNUAL ACM INTERNATIONAL SYMPOSIUM ON ADVANCES IN GEOGRAPHIC INFORMATION SYSTEMS, p. 22. ACM, 2007.

[4] ALVES, A.; ANTUNES, B.; PEREIRA, F. C. ; BENTO, C.. **Semantic enrichment of places: Ontology learning from web**. International Journal of Knowledge-based and Intelligent Engineering Systems, 13(1):19–30, 2009.

[5] Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D. ; Patel-Schneider, P. F., editors. **The Description Logic Handbook: Theory, Implementation, and Applications**. Cambridge University Press, USA, 2003.

[6] BALOG, K.; NEUMAYER, R.. **A test collection for entity search in dbpedia**. In: PROCEEDINGS OF THE 36TH INTERNATIONAL ACM SIGIR CONFERENCE ON RESEARCH AND DEVELOPMENT IN INFORMATION RETRIEVAL, p. 737–740, 2013.

[7] BAR-YOSSEF, Z.; JAYRAM, T.; KUMAR, R.; SIVAKUMAR, D. ; TREVISAN, L.. **Counting distinct elements in a data stream**. In: INTERNATIONAL WORKSHOP ON RANDOMIZATION AND APPROXIMATION TECHNIQUES IN COMPUTER SCIENCE, p. 1–10. Springer, Berlin, Heidelberg, 2002.

[8] BAST, H.; BUCHHOLD, B. ; HAUSSMANN, E.. **Semantic search on text and knowledge bases**. Foundations and Trends® in Information Retrieval, 10(2-3):119–271, 2016.

[9] BERGAMASCHI, S.; GUERRA, F.; INTERLANDI, M.; TRILLO-LADO, R. ; VELEGRAKIS, Y.. **Combining user and database perspective for solving keyword queries over relational databases**. Information Systems, 55:1–19, 2016.

[10] BEYER, K.; HAAS, P. J.; REINWALD, B.; SISMANIS, Y. ; GEMULLA, R.. **On synopses for distinct-value estimation under multiset operations**. In: PROCEEDINGS OF THE 2007 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, p. 199–210. ACM, 2007.

[11] BEYER, K.; GEMULLA, R.; HAAS, P. J.; REINWALD, B. ; SISMANIS, Y.. **Distinct-value synopses for multiset operations**. Commun. ACM, 52(10):87–95, Oct. 2009.

[12] BOGORNY, V.; KUIJPERS, B. ; ALVARES, L. O.. **St-dmql: a semantic trajectory data mining query language**. International Journal of Geographical Information Science, 23(10):1245–1276, 2009.

[13] BIZER, C.; SCHULTZ, A.. **The berlin sparql benchmark**. International Journal on Semantic Web and Information Systems (IJSWIS), 5(2):1–24, 2009.

[14] BRILHANTE, I.; MACEDO, J. A.; NARDINI, F. M.; PEREGO, R. ; RENSO, C.. **TripBuilder: A Tool for Recommending Sightseeing Tours**. In: PROCEEDINGS OF THE 36TH EUROPEAN CONFERENCE ON INFORMATION RETRIEVAL (ECIR'14), p. 771–774. Springer, Cham, 2014.

[15] CHEN, X.; PETROUNIAS, I.. **Language support for temporal data mining**. In: EUROPEAN SYMPOSIUM ON PRINCIPLES OF DATA MINING AND KNOWLEDGE DISCOVERY, p. 282–290. Springer, 1998.

[16] COFFMAN, J.; WEAVER, A. C.. **A framework for evaluating database keyword search strategies**. In: PROCEEDINGS OF THE 19TH ACM INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, p. 729–738, Toronto, ON, Canada, 2010.

[17] DE SMITH, M.; GOODCHILD, M. ; LONGLEY, P.. **Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools**. Drumlin Security, 2018.

[18] DOSSO, D.; SILVELLO, G.. **Search text to retrieve graphs: A scalable rdf keyword-based search system**. IEEE Access, 8:14089–14111, 2020.

[19] EGENHOFER, M. J.. **Spatial sql: A query and presentation language**. IEEE Transactions on knowledge and data engineering, 6(1):86–95, 1994.

[20] ELBASSUONI, S.; BLANCO, R.. **Keyword search over rdf graphs**. In: PROCEEDINGS OF THE 20TH ACM INTERNATIONAL CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, p. 237–242, New York, NY, USA, 2011. ACM.

[21] FILETO, R.; KRÜGER, M.; PELEKIS, N.; THEODORIDIS, Y. ; RENSO, C.. **Baquara: A holistic ontological framework for movement analysis using linked data**. In: INTERNATIONAL CONFERENCE ON CONCEPTUAL MODELING, p. 342–355. Springer, 2013.

[22] FILETO, R.; MAY, C.; RENSO, C.; PELEKIS, N.; KLEIN, D. ; THEODORIDIS, Y.. **The baquara2 knowledge-based framework for semantic enrichment and analysis of movement data**. Data & Knowledge Engineering, 98:104–122, 2015.

[23] FURTADO, A. S.; KOPANAKI, D.; ALVARES, L. O. ; BOGORNY, V.. **Multidimensional similarity measuring for semantic trajectories**. Transactions in GIS, 20(2):280–298, 2016.

[24] GARCÍA, G. M.; IZQUIERDO, Y. T.; MENENDEZ, E. S.; DARTAYRE, F. ; CASANOVA, M. A.. **Rdf keyword-based query technology meets a real-world dataset.** In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY (EDBT '17), p. 656–667, 2017.

[25] GHANBARPOUR, A.; NADERI, H.. **A model-based keyword search approach for detecting top-k effective answers**. The Computer Journal, 62(3):377–393, 2019.

[26] GKIRTZOU, K.; KAROZOS, K.; VASSALOS, V. ; DALAMAGAS, T.. **Keywords-to-sparql translation for rdf data search and exploration**. In: INTERNATIONAL CONFERENCE ON THEORY AND PRACTICE OF DIGITAL LIBRARIES, p. 111–123. Springer, 2015.

[27] GUO, Y.; PAN, Z. ; HEFLIN, J.. **Lubm: A benchmark for owl nowledge base systems**. Journal of Web Semantics, 3(2-3):158–182, 2005.

[28] HADJIELEFTHERIOU, M.; YU, X.; KOUDAS, N. ; SRIVASTAVA, D.. **Hashed samples: selectivity estimators for set similarity selection queries**. Proceedings of the VLDB Endowment, 1(1):201–212, 2008.

[29] HAN, S.; ZOU, L.; YU, J. X. ; ZHAO, D.. **Keyword search on rdf graphs – a query graph assembly approach**. In: PROCEEDINGS OF THE 2017 ACM ON CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT, CIKM'17, p. 227–236, Singapore, Singapore, 2017. ACM.

[30] HE, H.; WANG, H.; YANG, J. ; YU, P. S.. **Blinks: ranked keyword searches on graphs**. In: PROCEEDINGS OF THE 2007 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, p. 305–316, 2007.

[31] HRISTIDIS, V.; PAPAKONSTANTINOU, Y.. **Discover: Keyword search in relational databases**. In: PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES (VLDB '02), p. 670–681. Elsevier, 2002.

[32] HU, Y.; JANOWICZ, K.; CARRAL, D.; SCHEIDER, S.; KUHN, W.; BERG-CROSS, G.; HITZLER, P.; DEAN, M. ; KOLAS, D.. **A geo-ontology design pattern for semantic trajectories**. In: INTERNATIONAL CONFERENCE ON SPATIAL INFORMATION THEORY, p. 438–456. Springer, 2013.

[33] IZQUIERDO, Y. T.; GARCÍA, G. M.; MENENDEZ, E. S.; CASANOVA, M. A.; DARTAYRE, F. ; LEVY, C. H.. **Quiow: a keyword-based query processing tool for rdf datasets and relational databases**. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS (DEXA 2018), volumen 11030, p. 259–269. Springer, 2018.

[34] IZQUIERDO, Y. T.. **Keyword search algorithm over large rdf datasets**. In: PROCEEDINGS OF THE 38TH INTERNATIONAL CONFERENCE ON CONCEPTUAL MODELING – DOCTORAL SYMPOSYUM, volumen 11787, p. 230–238, Salvador, Bahia, Brazil, 2019. Springer.

[35] IZQUIERDO, Y. T.; MONTEAGUDO GARCÍA, G.; CASANOVA, M. A.; PAES LEME, L. A. P.; SARDIANOS, C.; TSERPES, K.; VARLAMIS, I. ; RUBACK RODRIGUES, L. C.. **Stop-and-move sequence expressions over semantic trajectories**. International Journal of Geographical Information Science (IJGIS), 0(0):1–26, 2020.

[36] LE, W.; DUAN, S.; KEMENTSIETSIDIS, A.; LI, F. ; WANG, M.. **Rewriting queries on sparql views**. In: PROCEEDINGS OF THE 20TH INTERNATIONAL CONFERENCE ON WORLD WIDE WEB, p. 655–664, Hyderabad, India, 2011. ACM.

[37] LE, W.; LI, F.; KEMENTSIETSIDIS, A. ; DUAN, S.. **Scalable keyword search on large rdf data**. IEEE Transactions on knowledge and data engineering, 26(11):2774–2788, 2014.

[38] LIN, X.-Q.; MA, Z.-M. ; YAN, L.. **Rdf keyword search using a type-based summary.** J. Inf. Sci. Eng., 34(2):489–504, 2018.

[39] MA, Z.; LIN, X.; YAN, L. ; ZHAO, Z.. **Rdf keyword search by query computation**. Journal of Database Management (JDM), 29(4):1–27, 2018.

[40] MAMOULIS, N.. **Spatial Data Management**. Morgan & Claypool Publishers, 2012.

[41] MENENDEZ, E. S.; CASANOVA, M. A.; LEME, L. A. P. ; BOUGHANEM, M.. **Novel node importance measures to improve keyword search over rdf graphs**. In: PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS (DEXA 2019), volumen 11707, p. 143–158, Linz, Austria, 2019. Springer.

[42] NEVES, A. B.; LEME, L. A. P. P.; IZQUIERDO, Y. T.; GARCÍA, G. M.; CASANOVA, M. A. ; MENENDEZ, E. S.. **Computing benchmarks for rdf keyword search**. In: (SUBMITTED FOR PUBLICATION), 2021.

[43] NUNES, B. P.; HERRERA, J.; TAIBI, D.; LOPES, G. R.; CASANOVA, M. A. ; DIETZE, S.. **Scs connector-quantifying and visualising semantic paths between entity pairs**. In: EUROPEAN SEMANTIC WEB CONFERENCE, p. 461–466. Springer, Cham, 2014.

[44] OLIVEIRA, P.; SILVA, A. ; MOURA, E.. **Ranking candidate networks of relations to improve keyword search over relational databases**. In: 2015 IEEE 31ST INTERNATIONAL CONFERENCE ON DATA ENGINEERING, p. 399–410. IEEE, 2015.

[45] OLIVEIRA FILHO, A. D. C.. **Benchmark para métodos de consultas por palavras-chave a bancos de dados relacionais**. Dissertação apresentada ao Programa de Pós-Graduação do Instituto de Informática da Universidade Federal de Goiás, 2018.

[46] PARENT, C.; SPACCAPIETRA, S.; RENSO, C.; ANDRIENKO, G.; ANDRIENKO, N.; BOGORNY, V.; DAMIANI, M. L.; GKOULALAS-DIVANIS, A.; MACEDO, J.; PELEKIS, N. ; OTHERS. **Semantic trajectories modeling and analysis**. ACM Computing Surveys (CSUR), 45(4):1–32, 2013.

[47] PETRY, L. M.; FERRERO, C. A.; ALVARES, L. O.; RENSO, C. ; BOGORNY, V.. **Towards semantic-aware multiple-aspect trajectory similarity measuring**. Transactions in GIS, 2019.

[48] RAMADA, M. S.; DA SILVA, J. C. ; DE SÁ LEITÃO-JÚNIOR, P.. **From keywords to relational database content: A semantic mapping method**. Information Systems, 88:101460, 2020.

[49] RENSO, C.; SPACCAPIETRA, S. ; ZIMÁNYI, E.. **Mobility Data**. Cambridge University Press, 2013.

[50] RENSO, C.; BAGLIONI, M.; DE MACEDO, J. A. F.; TRASARTI, R. ; WACHOWICZ, M.. **How you move reveals who you are: understanding human behavior by analyzing trajectory data**. Knowledge and information systems, 37(2):331–362, 2013.

[51] RIGAUX, P.; SCHOLL, M. ; VOISARD, A.. **Spatial Databases: with Application to GIS**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[52] RIHANY, M.; KEDAD, Z. ; LOPES, S.. **Keyword search over rdf graphs using wordnet**. In: PROCEEDINGS OF THE 1ST INT'L. CONF. ON BIG DATA AND CYBER-SECURITY INTELLIGENCE, p. 75–82, Hadath, Lebanon, 2018.

[53] SANTIPANTAKIS, G. M.; VOUROS, G. A.; DOULKERIDIS, C.; VLACHOU, A.; ANDRIENKO, G.; ANDRIENKO, N.; FUCHS, G.; GARCIA, J. M. C. ; MARTINEZ, M. G.. **Specification of semantic trajectories supporting data transformations for analytics: The datacron ontology**. In: PROCEEDINGS OF THE 13TH INTERNATIONAL CONFERENCE ON SEMANTIC SYSTEMS, p. 17–24. ACM, 2017.

[54] SPACCAPIETRA, S.; PARENT, C.; DAMIANI, M. L.; DE MACEDO, J. A.; PORTO, F. ; VANGENOT, C.. **A conceptual view on trajectories**. Data & knowledge engineering, 65(1):126–146, 2008.

[55] TRAN, T.; WANG, H.; RUDOLPH, S. ; CIMIANO, P.. **Top-k exploration of query candidates for efficient keyword search on graph-shaped**

(rdf) data. In: 2009 IEEE 25TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING, p. 405–416. IEEE, 2009.

[56] TVERSKY, A.. **Features of similarity.** Psychological review, 84(4):327, 1977.

[57] VAN HAGE, W. R.; MALAISÉ, V.; DE VRIES, G.; SCHREIBER, G. ; VAN SOMEREN, M.. **Combining ship trajectories and semantics with the simple event model (sem)**. In: PROCEEDINGS OF THE 1ST ACM INTERNATIONAL WORKSHOP ON EVENTS IN MULTIMEDIA, p. 73–80. ACM, 2009.

[58] VENETIS, P.; SISMANIS, Y. ; REINWALD, B.. **Crsi: a compact randomized similarity index for set-valued features**. In: PROCEEDINGS OF THE 15TH INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, p. 384–395, New York, NY, USA, 2012. ACM.

[59] M.S., V.; HARITSA, J. R.. **Operator implementation of result set dependent kws scoring functions**. Information Systems, 89:101465, 2020.

[60] WANG, H.; ZHANG, K.; LIU, Q.; TRAN, T. ; YU, Y.. **Q2semantic: A lightweight keyword interface to semantic search**. In: EUROPEAN SEMANTIC WEB CONFERENCE, p. 584–598. Springer, 2008.

[61] WEN, Y.; JIN, Y. ; YUAN, X.. **Kat: Keywords-to-sparql translation over rdf graphs**. In: INTERNATIONAL CONFERENCE ON DATABASE SYSTEMS FOR ADVANCED APPLICATIONS, p. 802–810. Springer, 2018.

[62] YAN, Z.; MACEDO, J.; PARENT, C. ; SPACCAPIETRA, S.. **Trajectory ontologies and queries**. Transactions in GIS, 12:75–91, 2008.

[63] YANG, Y.; ZHANG, Y.; ZHANG, W. ; HUANG, Z.. **Gb-kmv: An augmented kmv sketch for approximate containment similarity search**. In: 2019 IEEE 35TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE), p. 458–469. IEEE, 2019.

[64] YOGHOURDJIAN, H.; ELBASSUONI, S.; JABER, M. ; ARNAOUT, H.. **Top-k keyword search over wikipedia-based rdf knowledge graphs**. In: KDIR, p. 17–26, 2017.

[65] ZENZ, G.; ZHOU, X.; MINACK, E.; SIBERSKI, W. ; NEJDL, W.. **From keywords to semantic queries—incremental query construction on the semantic web**. Journal of Web Semantics, 7(3):166–176, 2009.

[66] ZHAI, E.; CHEN, R.; WOLINSKY, D. I. ; FORD, B.. **Heading off correlated failures through independence-as-a-service**. In: 11TH SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION, p. 317–334, 2014.

[67] ZHENG, V. W.; ZHENG, Y.; XIE, X. ; YANG, Q.. **Collaborative location and activity recommendations with GPS history data**. In: PROCEEDINGS OF THE 19TH INTERNATIONAL CONFERENCE ON WORLD WIDE WEB (WWW'10), p. 1029–1038. ACM Press, 2010.

[68] ZHENG, Y.; CHEN, Y.; LI, Q.; XIE, X. ; MA, W. Y.. **Understanding transportation modes based on GPS data for web applications**. ACM Transactions on the Web, 4(1), 2010.

[69] ZHENG, W.; ZOU, L.; PENG, W.; YAN, X.; SONG, S. ; ZHAO, D.. **Semantic sparql similarity search over rdf knowledge graphs**. Proceedings of the VLDB Endowment, 9(11):840–851, 2016.

[70] ZHOU, B.; PEI, J.. **Answering aggregate keyword queries on relational databases using minimal group-bys**. In: PROCEEDINGS OF THE 12TH INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY: ADVANCES IN DATABASE TECHNOLOGY, EDBT '09, p. 108–119, New York, NY, USA, 2009. Association for Computing Machinery.

[71] ZHOU, Q.; WANG, C.; XIONG, M.; WANG, H. ; YU, Y.. **Spark: adapting keyword query to semantic search**. In: PROCEEDINGS OF THE 6TH INTERNATIONAL SEMANTIC WEB CONFERENCE AND 2ND ASIAN SEMANTIC WEB CONFERENCE (ISWC'07/ASWC'07), p. 694–707. Springer, 2007.

A
Query Workloads for Mondial and IMDb

Table A.1: Query workload for Mondial

Groups	Keyword Queries	τ
A) Retrieve instances of classes (using metadata and value matches)	1. niger country	0.6
	2. atacama desert	0.5
	3. mongolia parliamentary	0.5
	4. everest elevation	0.4
B) Retrieve joined instances of different classes (using metadata and value matches)	5. spain galician	0.5
	6. poland language	0.3
	7. haiti religion	0.4
	8. brazil brasília	0.4
C) Retrieve joined instances of the same class (using value matches)	9. mongolia china	0.7
	10. lebanon syria	0.7
	11. mali france	0.8
	12. brazil portugal	0.5
D) Retrieve two instances of the same class joined by instances of another class	13. poland cape verde organization	0.8
	14. iceland mali organization	0.7
	15. mauritius india organization	0.9
	16. vanuatu afghanistan organization	0.6
E) Retrieve two instances of different classes joined by elements of another class through intermediary nodes	17. hutu country africa	0.8
	18. country asia uzbek	0.8
	19. country america catholic	0.8
	20. country european jewish	0.9
F) Retrieve joined instances from various classes	21. paranaiba province brazil	0.6
	22. atacama province argentina	0.8
	23. everest province china	0.7
	24. rhein germany province	0.7

Table A.2: Query workload for IMDb

Groups	Keyword Queries	τ
A) Retrieve instances of classes (using metadata and value matches)	1. denzel washington person	3.5
	2. johnny depp actor	5.5
	3. forrest gump work	2.6
	4. star wars movie	13.7
	5. angelina jolie gender	5.2
	6. the sound of music length	5.2
	7. lord of the rings novel	7.5
B) Retrieve instances filtering by property value matches	8. will smith male	59.4
	9. tom hanks “9 july 1956"	33.4
	10. gone with the wind “august 1991"	20.7
	11. casablanca “they had a date with fate in casablanca"	46.4
C) Retrieve joined instances of different classes (using metadata and value matches)	12. johnny depp work	5.9
	13. morgan freeman work	4.2
	14. atticus finch movie	1.7
	15. indiana jones movie	8.5
	16. james bond movie	18.6
	17. rick blaine movie	1.8
	18. will kane movie	2.1
	19. dr. hannibal lecter movie	1.2
	20. norman bates movie	1.2
	21. darth vader movie	4.4
	22. the wicked witch of the west movie	2.1
	23. nurse ratched movie	1.5
	24. jacques clouseau actor	1.3
	25. jack ryan actor	2.6
	26. terminator actor	9.3
D) Retrieve two joined instances of different classes (using value matches)	27. clint eastwood frank horrigan	10.2
	28. tom hanks 2004	5.7
	29. audrey hepburn 1951	32.5
E) Retrieve two instances of the same class joined by elements of another class	30. julia roberts richard gere work	12.6
	31. harrison ford george lucas work	30.2
	32. sean connery ian fleming work	20.7
	33. keanu reeves lana wachowski work	6.3
	34. dean jones herbie	15.6

Table A.2 – continued from previous page

Groups	**Keyword Queries**	τ
	35. Indiana Jones and the last crusade raiders of the lost ark person	5.4
F) Retrieve joined instances from various classes	36. nathan algren tom cruise Work	7.8
	37. rocky balboa sylvester stallone Work	11.9
	38. henry jaynes fonda work yours mine and ours character	3.5
	39. russell crowe work gladiator character	22.3
	40. brent spiner work star trek the next generation character	1.3

B
Examples of Computing Dosso's Metrics

Consider the ground truth graph (GT) and the sequence of answer graphs $GA1$, $GA2$, and $GA3$, in Figure B.1.

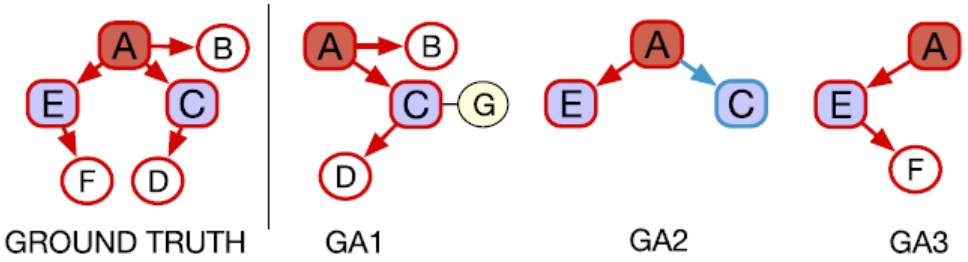


Figure B.1: An example of a ground truth graph and a sequence of answer graphs

B.1
Computing the Signal-to-Noise Ratio (SNR)

Table B.1 shows the SNR values for each answer graph. Recall SNR is defined following Equation (13):

$$SNR(G_i) = \frac{|(G_i \cap G_{t_k}) - S|}{G_i}$$

B.2
Computing Recall

Recalling the example described in Appendix B.1, and considering a relevance parameter $\lambda = 0.7$, the *recall* of $R_k = \{GA1, GA2, GA3\}$, as shown in Figure B.1, is:

$$recall(R_k) = \frac{|(GA1 \cap GT)|}{|GT|} = \frac{3}{5} = \mathbf{0.6}$$

since, by Table B.1 and given that $\lambda = 0.7$, the only relevant answer graph is $GA1$ and $|GA1 \cap GT| = |\{A \rightarrow B, A \rightarrow C, C \rightarrow D\}|$=3.

Note that if we set the relevance parameter $\lambda = 0.5$, then the recall value of R_k is 1.

Table B.1: Computation of SNR(G_i) for the answer graphs in Figure B.1

i	$SNR(G_i)$
1	$S = \emptyset$ $SNR(GA1) = \frac{\lvert(GA1 \cap GT) - S\rvert}{\lvert GA1\rvert} = \frac{\lvert\{A \to B, A \to C, C \to D\} - S\rvert}{\lvert GA1\rvert} =$ $= \frac{\lvert\{A \to B, A \to C, C \to D\}\rvert}{\lvert GA1\rvert} = \frac{3}{4} = \mathbf{0.75}$
2	$S = \{A \to B, A \to C, C \to D\}$ $SNR(GA2) = \frac{\lvert(GA2 \cap GT) - S\rvert}{\lvert GA2\rvert} = \frac{\lvert\{A \to E, A \to C\} - S\rvert}{\lvert GA2\rvert} =$ $= \frac{\lvert\{A \to E\}\rvert}{\lvert GA2\rvert} = \frac{1}{2} = \mathbf{0.5}$
3	$S = \{A \to B, A \to C, C \to D, A \to E\}$ $SNR(GA3) = \frac{\lvert(GA3 \cap GT) - S\rvert}{\lvert GA3\rvert} = \frac{\lvert\{A \to E, E \to F\} - S\rvert}{\lvert GA3\rvert} =$ $= \frac{\lvert\{E \to F\}\rvert}{\lvert GA3\rvert} = \frac{1}{2} = \mathbf{0.5}$

B.3 Computing Precision and Precision at *c*

Returning to Appendixes B.1 and B.2. Again, by Table B.1 and since $\lambda = 0.7$, the only relevant answer graph is GA1. The *precision* of $R_k = \{GA1, GA2, GA3\}$ therefore is

$$precision(R_k) = \frac{|GA1 \cap GT|}{|GA1 \cup GA2 \cup GA3|} = \frac{3}{6} = \mathbf{0.833}$$

where $|GA1 \cup GA2 \cup GA3| =$
$$= |A \to B, A \to C, C \to G, C \to D, A \to E, E \to F| = 6.$$

Considering $c = 1$, the *precision* at 1 of R_k, *prec*@1, is

$$prec@1(R_k) = \frac{|GA1 \cap GT|}{|GA1|} = \frac{|A \to B, A \to C, C \to D|}{|GA1|} = \frac{3}{4} = \mathbf{0.75}$$

B.4 Computing Graph Relevance Weight (*GRW*)

Considering the ground truth graph and the ranked answers list depicted in Figure B.1, then Table B.2 shows the computed GRW values for the answers $GA1$, $GA2$, and $GA3$, respectively. Recall that the GRW is computed using

the Eq. (17):

$$GRW(G_i) = \frac{|(G_i \cap G_{t_k}) - S|}{|G_{t_k}|}$$

Table B.2: Computation of GRW(G_i) for the answer graphs in Figure B.1

i	$\boldsymbol{GRW}(G_i)$
1	$S = \emptyset$ $GRW(GA1) = \frac{\lvert(GA1 \cap GT) - S\rvert}{\lvert GT\rvert} = \frac{\lvert\{A \to B, A \to C, C \to D\} - S\rvert}{\lvert GT\rvert} =$ $= \frac{\lvert\{A \to B, A \to C, C \to D\}\rvert}{\lvert GT\rvert} = \frac{3}{5} = \mathbf{0.6}$
2	$S = \{A \to B, A \to C, C \to D\}$ $GRW(GA2) = \frac{\lvert(GA2 \cap GT) - S\rvert}{\lvert GT\rvert} = \frac{\lvert\{A \to E, A \to C\} - S\rvert}{\lvert GT\rvert} =$ $= \frac{\lvert\{A \to E\}\rvert}{\lvert GT\rvert} = \frac{1}{5} = \mathbf{0.2}$
3	$S = \{A \to B, A \to C, C \to D, A \to E\}$ $GRW(GA3) = \frac{\lvert(GA3 \cap GT) - S\rvert}{\lvert GT\rvert} = \frac{\lvert\{A \to E, E \to F\} - S\rvert}{\lvert GT\rvert} =$ $= \frac{\lvert\{E \to F\}\rvert}{\lvert GT\rvert} = \frac{1}{5} = \mathbf{0.2}$

B.5 Computing Relevance Gain (*RG*) and Discounted Cumulative Gain (*tb-DCG*)

Let b=2 and λ=0.7. Table B.3 shows the RG values for the sequence of answer graphs considering the ground truth graph depicted in Figure B.1. Recall that the RG value is computed using Eq. (18):

$$RG_b(G_i) = \begin{cases} GRW(G_i) & if \quad i \leq b \ \wedge \ SNR(G_i) > \lambda \\ \frac{GRW(G_i)}{\log_b i} & if \quad i > b \ \wedge \ SNR(G_i) > \lambda \\ 0 & if \quad SNR(G_i) \leq \lambda \end{cases}$$

Table B.3: Computation of RG(G_i) for the answer graphs in Figure B.1

i	$\boldsymbol{RG}(G_i)$
1	$1 \leq 2$ && $SNG(GA1) = 0.75 > 0.7 \Rightarrow RG(GA1) = 3/5 = 0.6$
2	$SNR(GA2) = 0.5 < 0.7 \Rightarrow RG(GA2) = 0$
3	$SNR(GA3) = 0.5 < 0.7 \Rightarrow RG(GA3) = 0$

Now, we can compute the $tb{-}DCG_2(R_k)$. Recall the measure is computed following the Eq. (19):

$$tb - DCG_b(R_k) = \sum_{i=1}^{n} RG_b(G_i)$$

Then, attending the computed RG values in Table B.3,

$$tb - DCG_2(R_k) = 0.6 + 0 + 0 = 0.6$$

C
Sample Queries of Stop-and-Move Sequence Expressions

C.1
Stops and Sequences of Stops

Q1: Find trajectories that stop at a museum and then at a chapel.

- Symbolic notation:

```
Museidipisa; Cappelledipisa
```

- Reserved terms-based notation:

```
Museidipisa "and then" Cappelledipisa
```

Q2: Find trajectories that stop at a tower, then stop at a chapel or church, and then at a museum.

- Symbolic notation:

```
Torridipisa; (Cappelledipisa | Chiesedipisa); Museidipisa
```

- Reserved terms-based notation:

```
Torridipisa "and then" (Cappelledipisa or Chiesedipisa)
"and then" Museidipisa
```

Q3: Find trajectories that stop at least once in a tower, and then at a museum.

- Symbolic notation:

$$\texttt{Torridipisa}^{+}\texttt{; Museidipisa}$$

- Reserved terms-based notation:

```
Torridipisa "at least once" "and then" Museidipisa
```

Q4: Find trajectories that stop at the Lion Tower and then at the Leaning Tower, or stop at the Leaning Tower and then at the Lion Tower.

- Symbolic notation:

```
(Torre_del_Leone; Torre_pendente_di_pisa) |
(Torre_pendente_di_pisa; Torre_del_Leone)
```

- Reserved terms-based notation:

```
(Torre_del_Leone "and then" Torre_pendente_di_pisa) or
(Torre_pendente_di_pisa "and then" Torre_del_Leone)
```

Q5: Find trajectories that begin at a museum and then end at a chapel.

- Symbolic notation:

```
(Begin ⊓ Museidipisa); (Cappelledipisa ⊓ End)
```

- Reserved terms-based notation:

```
(Begin and Museidipisa) "and then"
(Cappelledipisa and End)
```

Q6: Find trajectories that stop at a museum and, later on, end at a chapel or a church optionally.

- Symbolic notation:

```
Museidipisa; Stop*; (Cappelledipisa | Chiesedipisa)? ⊓ End
```

- Reserved terms-based notation:

```
Museidipisa "and then" "any stop zero or more times"
"and then"
(Cappelledipisa or Chiesedipisa) optionally and End
```

Note: The term "`Stop*`" indicates that the trajectory may have zero or more stops of any kind between the stop that satisfies "`Museidipisa`" and the end of the trajectory.

Q7: Find trajectories that begin at a chapel, stop at zero or more chapels, and end at a chapel.

- Symbolic notation:

```
Begin ⊓ Cappelledipisa; Cappelledipisa*;
Cappelledipisa ⊓ End
```

- Reserved terms-based notation:

```
Begin and Cappelledipisa "and then" Cappelledipisa "zero or
more times" "and then" Cappelledipisa and End
```

C.2
Stops and Moves in a Sequence

Q8: Find trajectories that stop at a museum and then take a bus to a chapel.

- Symbolic notation:

```
Museidipisa <Bus> Cappelledipisa
```

- Reserved terms-based notation:

```
Museidipisa "by Bus to" Cappelledipisa
```

Q9: Find the trajectories that begin at a chapel or a church, always move by bus between stops, and end at the Leaning Tower.

- Symbolic notation:

```
(Begin ⊓ (Cappelledipisa | Chiesedipisa) ) <Bus+>
(Torre_pendente_di_pisa ⊓ End)
```

- Reserved terms-based notation:

```
Begin and (Cappelledipisa or Chiesedipisa)
"by Bus at least once to" Torre_pendente_di_pisa and End
```

Q10: Find trajectories that begin at a tower, then walk to take a bus to a church, and then using any transportation means end at a palace.

- Symbolic notation:

```
Begin ⊓ Torridipisa <Walk; Bus> Chiesedipisa
<Move> Palazzidipisa ⊓ End
```

- Reserved terms-based notation:

```
Begin and Torridipisa "by walk and then Bus to"
Chiesedipisa
"by any move to" Palazzidipisa and End
```

Note: The term "`Move`" indicates that the trajectory may have any kind of move between the stop that satisfies "`Chiesedipisa`" and the end of the trajectory.

D
Compiled SPARQL Queries from the Sample of Keyword Queries Expressions

Table D.1: Keyword query expressions and their translations to SPARQL

Query	Compiled SPARQL Query	Runtime (s)	Results
Q1	SELECT DISTINCT ?t, ?stop1, ?stop2 { ?v1 text:query "museidipisa". ?stop1 ex:enrichedBy ?v1 . ?v2 text:query "cappelledipisa". ?stop2 ex:enrichedBy ?v2 . ?t ex:has ?stop1; ex:has ?stop2 . ?stop1 ex:nextS ?stop2 }	0.073	33
Q2	SELECT DISTINCT ?t, ?stop1, ?stop3 { { ?v1 text:query "torridipisa" . ?stop1 ex:enrichedBy ?v1. } { ?v2 text:query "cappelledipisa". ?stop2 ex:enrichedBy ?v2 UNION { ?v3 text:query "chiesedipisa". ?stop2 ex:enrichedBy ?v3 } } { ?v4 text:query "museidipisa" . ?stop3 ex:enrichedBy ?v4. } ?t ex:has ?stop1; ex:has ?stop2; ex:has ?stop3 . ?stop1 ex:nextS ?stop2 . ?stop2 ex:nextS ?stop3 }	0.493	15
Q3	SELECT DISTINCT ?t, ?stop1, ?stop3 { ?v1 text:query "torridipisa" . ?stop1 ex:enrichedBy ?v1. ?v2 text:query "torridipisa". ?stop2 ex:enrichedBy ?v2 ?v3 text:query "museidipisa". ?stop3 ex:enrichedBy ?v3	5.167	7

Query	Compiled SPARQL Query	Runtime (s)	Results

	`?t ex:has ?stop1; ex:has ?stop2;`
	`ex:has ?stop3`
	`filter not exists {`
	`?t ex:has ?stopM .`
	`?stop1 ex:nextS* ?stopM .`
	`?stopM ex:nextS* ?stop2 .`
	`filter not exists {`
	`?stopM ex:enrichedBy ?vM .`
	`?vM text:query "torridipisa"} }`
	`?stop2 ex:nextS ?stop3 }`		

Q4	`SELECT DISTINCT ?t, ?stop1, ?stop2 {`
	`{ ?v1 text:query "torre_del_leone" .`
	`?stop1 ex:enrichedBy ?v1.`
	`?v2 text:query "torre_pendente_di_pisa".`
	`?stop2 ex:enrichedBy ?v2 .`
	`?t ex:has ?stop1; ex:has ?stop2 .`
	`?stop1 ex:nextS ?stop2 }`
	`UNION {`
	`?v1 text:query "torre_pendente_di_pisa".`
	`?stop3 ex:enrichedBy ?v3.`
	`?v4 text:query "torre_del_leone".`
	`?stop4 ex:enrichedBy ?v4 .`
	`?t ex:has ?stop3; ex:has ?stop4 .`
	`?stop3 ex:nextS ?stop4 } }`
		0.623	3

Q5	`SELECT DISTINCT ?t, ?stop1, ?stop2 {`
	`?v1 text:query "museidipisa".`
	`?stop1 ex:enrichedBy ?v1 .`
	`?v2 text:query "cappelledipisa".`
	`?stop2 ex:enrichedBy ?v2 .`
	`?t ex:begins ?stop1; ex:ends ?stop2 .`
	`?stop1 ex:nextS ?stop2 }`
		0.075	5

Q6	`SELECT DISTINCT ?t, ?stop1, ?stop2 {`
	`?v1 text:query "museidipisa".`
	`?stop1 ex:enrichedBy ?v1 .`
	`?t ex:has ?stop1`
	`OPTIONAL {`
		0.763	607

Query	Compiled SPARQL Query	Runtime (s)	Results

	`{ ?stop2 ex:enrichedBy ?v2 .`
	`?v2 text:query "cappelledipisa"}`
	`UNION {`
	`?stop2 ex:enrichedBy ?v3 .`
	`?v3 text:query "chiesedipisa"}`
	`?stop1 ex:nextS* ?stop2 .`
	`?t ex:ends ?stop2 } }`		

Q7	`SELECT DISTINCT ?t, ?stop1, ?stop3 {`
	`?v1 text:query "cappelledipisa" .`
	`?stop1 ex:enrichedBy ?v1. }`
	`?v2 text:query "cappelledipisa".`
	`?stop2 ex:enrichedBy ?v2 .`
	`?v3 text:query "cappelledipisa" .`
	`?stop3 ex:enrichedBy ?v3 .`
	`?t ex:begins ?stop1; ex:ends ?stop3 .`
	`filter not exists {`
	`?t ex:has ?stopM .`
	`?stop1 ex:nextS* ?stopM .`
	`?stopM ex:nextS* ?stop2 .`
	`filter exists {`
	`?stopM ex:enrichedBy ?vM .`
	`?vM text:query "cappelledipisa"} }`
	`?stop2 ex:nextS ?stop3 }`
		3.747	2

Q8	`SELECT DISTINCT ?t, ?stop1, ?stop2 {`
	`?v1 text:query "museidipisa".`
	`?stop1 ex:enrichedBy ?v1 .`
	`?v2 text:query "cappelledipisa".`
	`?stop2 ex:enrichedBy ?v2 .`
	`?move ex:from ?stop1; ex:to ?stop2 ;`
	`ex:enrichedBy ?transp .`
	`filter (?transp = res:Bus)`
	`?t ex:has ?stop1; ex:has ?stop2;`
	`ex:has move }`
		0.709	10

Q9	see example in **Section 7.4**	0.443	27
Q10	`SELECT DISTINCT ?t, ?stop1, ?stop3 {` `?v1 text:query "torridipisa" .`	3.066	2

Query	Compiled SPARQL Query	Runtime (s)	Results

```
?stop1 ex:enrichedBy ?v1 .
?v2 text:query "chiesedipisa" .
?stop2 ex:enrichedBy ?v2 .
?v3 text:query "palazzidipisa" .
?stop3 ex:enrichedBy ?v3 .
?t ex:begins ?stop1; ex:has ?stop2;
  ex:ends ?stop3
?move1 ex:from ?stop1; ex:to ?stopM .
?move1 ex:enrichedBy res:Walk .
?move2 ex:from ?stopM; ex:to ?stop2 .
?move2 ex:enrichedBy res:Bus .
?move3 ex:from ?stop2; ex:to ?stop3 .
?t ex:has ?move1; ex:has ?move2;
  ex:has ?stopM; ex:has ?move3 }
```

```
prefix text:<http://jena.apache.org/text#>
prefix ex: <http://localhost:8080/vocab/>
prefix res: <http://localhost:8080/resource/>
```

E
Articles Related to the Thesis

This appendix lists articles related to the topics covered in the thesis that were already published, or that are under review.

Published:

- García G.M., Izquierdo Y.T., Menendez E.S., Dartayre F., Casanova M.A. (2017) **RDF Keyword-based Query Technology Meets a Real-World Dataset.** Proc. 20th International Conference on Extending Database Technology (EDBT), March 21-24, Venice, Italy: ISBN 978-3-89318-073-8, on OpenProceedings.org, pp. 656-667. doi: 10.5441/002/edbt.2017.86

- Izquierdo Y.T., Casanova M.A., García, G.M., Dartayre F., Levy C.H. (2017) **Keyword Search over Federated RDF Datasets.** Proc. ER Forum 2017 and ER Demo track co-located with the 36th International Conference on Conceptual Modelling (ER 2017), Valencia, Spain, November 6th-9th, 2017, CEUR Workshop Proceedings, Vol. 1979, CEUR-WS.org, pp. 86–99.

- Izquierdo Y.T., García G.M., Menendez E.S., Casanova M.A., Dartayre F., Levy C.H. (2018) **QUIOW: A Keyword-Based Query Processing Tool for RDF Datasets and Relational Databases.** In: Hartmann S., Ma H., Hameurlain A., Pernul G., Wagner R. (eds) Database and Expert Systems Applications. DEXA 2018. DEXA 2018. Lecture Notes in Computer Science, vol. 11030. Springer, Cham, pp. 259–269.

- Izquierdo Y.T. (2019) **Keyword Search Algorithm over Large RDF Datasets.** In: Guizzardi G., Gailly F., Suzana Pitangueira Maciel R. (eds) Advances in Conceptual Modeling. ER 2019. Lecture Notes in Computer Science, vol 11787. Springer, Cham, pp. 230–238. DOI: https://doi.org/10.1007/978-3-030-34146-6_21.

- Izquierdo, Y.T., García, G.M., Casanova, M.A., Leme, L.A.P.P., Sardianos, C., Tserpes, K.,Varlamis, I., Ruback Rodrigues, L. (2020) **Stop-and-move sequence expressions over semantic trajectories.** International Journal of Geographical Information Science, Vol. 0, No. 0, Taylor & Francis, pp. 1-26 . DOI: https://doi.org/10.1080/13658816.2020.1793157.

- Izquierdo Y.T., García, G.M., Lemos M., Novello A., Novelli B., Damasceno C., Leme, L.A.P.P., Casanova, M.A. (2020) **Keyword Search over COVID-19 Data.** 35th Brazilian Symposium on Databases (SBBD 2020).

- Izquierdo, Y.T., García, G.M., Novelli, B., Lemos-Cavaliere, M., Lima, M.J.D., Casanova, M.A., Roehl, D. (2020) **Integrating a Geomechanical Collaborative Research Portal with a Data & Knowledge Retrieval Platform**. Proceedings of the Rio Oil & Gas Expo and Conference, Rio de Janeiro, RJ, Brazil. DOI: https://doi.org/10.48072/2525-7579.rog.2020.421.

Under review:

- Izquierdo Y.T., García G.M., Casanova M.A., Menendez E.S., Leme L.A.P.P., Neves A.B., Lemos M., Finamore A.C., Oliveira C.M.S. **Keyword Search over Schema-less RDF Datasets by SPARQL Query Compilation.** (under review).

- Neves A.B, Leme L.A.P.P., Izquierdo Y.T., García G.M., Casanova, M.A., Menendez E.S. **Computing Benchmarks for RDF Keyword Search.** (under review).