Grettel Monteagudo García

**A Keyword-based Query Processing Method
for Datasets with Schemas**

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
March 2020

Grettel Monteagudo García

Grettel Monteagudo García holds a Master Degree in Computer Science from the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) since 2016, and a Bachelor Degree in Computer Science from the University of Havana (UH), since 2012. Her main research topics are Semantic Web and Information Retrieval.

Acknowledgments

I would like to thank to all my family, a special thank you to my parents, Sonia and Camilo, for their support and encouragement during all these years of study and to my little sisters, all I do is with the hope to be an example to them.

The most special thank you to my husband Alejandro, my loyal companion in this adventure so far from my country.

I would like to thank my advisor Marco Antonio Casanova, the best advisor I could ever ask for. I admire him for their professionalism and unconditional support to his students. For sure, his fully support and wisdom were key contributors in my academic achievements.

I can't forget to thank the team from Tecgraf/K2, which were a key part in the years of this project. Best regards Elisa, Fred, Kaka, Bruno and Melissa.

Of course, I would like to extend my appreciation and gratitude to classmates, professors and staff from the Department of Informatics. Thanks to all for your help and for always being so accommodating.

Last, but not least important, my deep gratitude to the Cuban troops in Rio de Janeiro, specially to my old friend and research partner Yenier and to my Cuban/Brazilian godchild Liam and his parents for that gift.

Abstract

Users currently expect to query data in a *Google-like* style, by simply typing some terms, called *keywords*, and leaving it to the system to retrieve the data that best match the set of keywords. The scenario is quite different in database management systems, where users need to know sophisticated query languages to retrieve data, and in database applications, where the user interfaces are designed as a stack of pages with numerous "boxes" that the user must fill with his search parameters. This thesis describes an algorithm and a framework designed to support keyword-based queries for datasets with schema, specifically RDF datasets and relational databases. The algorithm first translates a keyword-based query into an abstract query, and then compiles the abstract query into a SPARQL or a SQL query such that each result of the SPARQL (resp. SQL) query is an answer for the keyword-based query. It explores the schema to avoid user intervention during the translation process and offers a feedback mechanism to generate new answers. The thesis concludes with experiments over the Mondial, IMDb, and Musicbrainz databases. The proposed translation algorithm achieves satisfactory results and good performance for the benchmarks. The experiments also compare the RDF and the relational alternatives.

Keywords

Resumo

Monteagudo García, Grettel; Casanova, Marco Antonio (orientador). **Método para o Processamento de Consultas por Palavras-Chaves para Bases de Dados com Esquemas.** Rio de Janeiro, 2020. 101p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Usuários atualmente esperam consultar dados de maneira semelhante ao Google, digitando alguns termos, chamados palavras-chave, e deixando para o sistema recuperar os dados que melhor correspondem ao conjunto de palavras-chave. O cenário é bem diferente em sistemas de gerenciamento de banco de dados em que os usuários precisam conhecer linguagens de consulta sofisticadas para recuperar dados, ou em aplicações de banco de dados em que as interfaces de usuário são projetadas como inúmeras "caixas" que o usuário deve preencher com seus parâmetros de pesquisa. Esta tese descreve um algoritmo e um framework projetados para processar consultas baseadas em palavras-chave para bases de dados com esquema, especificamente bancos relacionais e bases de dados em RDF. O algoritmo primeiro converte uma consulta baseada em palavras-chave em uma consulta abstrata e, em seguida, compila a consulta abstrata em uma consulta SPARQL ou SQL, de modo que cada resultado da consulta SPARQL (resp. SQL) seja uma resposta para a consulta baseada em palavras-chave. O algoritmo explora o esquema para evitar a intervenção do usuário durante o processo de busca e oferece um mecanismo de feedback para gerar novas respostas. A tese termina com experimentos nas bases de dados Mondial, IMDb e Musicbrainz. O algoritmo proposto obtém resultados satisfatórios para os benchmarks. Como parte dos experimentos, a tese também compara os resultados e o desempenho obtidos com bases de dados em RDF e bancos de dados relacionais.

Palavras-chave

Árvores de Steiner; Busca por palavras-chave; RDF; SPARQL; SQL.

Table of Contents

List of Figures

List of Tables

1
Introduction

1.1   Context and Motivation

Users currently expect to query data in a *Google-like* style, by simply typing some terms, called *keywords*, and leaving it to the system to retrieve the data that best match the set of keywords. These systems usually offer an advanced search interface, which the user may take advantage to specify Boolean functions involving the keywords or to limit the search. We call *advanced keyword queries* (*KwQ+*) keyword-based queries that allow specifying Boolean functions involving the keywords.

Keyword search mechanisms were mostly used by search engines for Web pages, but the use of keyword search was extended to retrieve images, videos, publications, and others. The success of such systems may, therefore, be credited to (1) a very simple user interface; (2) an efficient retrieval mechanism; and (3) a ranking algorithm that meets user expectations, that is, the user will find the most interesting items at the top of the result list.

In database management systems and database applications, the scenario is quite different. Usually, to retrieve data, users need to know sophisticated query languages and how the data is structured. Database applications create user interfaces that hide the complexity of the query language. These interfaces are often designed as a stack of pages with numerous "boxes" that the user must fill with his search parameters. Hitting the middle ground, we find database applications that offer keyword-based query interfaces (in short *KwS database applications*). KwS database applications should reach a performance similar to that of the Information Retrieval applications for the Web, although the underlying data is stored in a conventional database. Furthermore, they should free the user from filling "boxes" with exact data by compiling keyword-based queries into meaningful queries, from the user point of view, written in the supported language.

Unquestionably, relational databases are widely used, but with the emergence of the concepts of Linked Data, the use of RDF datasets became an

interesting alternative. The adoption of RDF as the underlying data model has some attractive advantages, the most obvious is the flexibility RDF offers by modeling data as RDF triples of the form (s,p,o), which asserts that resource s has property p with value o. A collection of RDF triples intrinsically represents a labeled, directed multi-graph. Conceptually a relational database can also be viewed as a graph, where tuples in different tables are treated as nodes connected via foreign key relationships. Both relational databases and RDF datasets can therefore be viewed as a graph.

In Web Information Retrieval, there are two main tasks: (1) matching keywords with indexed documents; (2) ranking the retrieved documents by order of relevance. KwS database applications present a further challenge, compared to the Web, since the data that a user needs may not be in one single place, but rather it is distributed over the database. An answer for a keyword-based query over a graph database is a substructure of the graph containing all keywords. Summarizing, the three main tasks in *KwS database applications* over graph databases are: (1) finding pieces of information in the database; (2) assembling the retrieved pieces of information to compose answers; (3) ranking the answers.

KwS database applications over relational databases have been studied for quite some time (Aditya, 2002; Agrawal et al., 2002; Hristidis & Papakonstantinou, 2002). Considering that RDF datasets are interesting sources of knowledge that are also queried with non-friendly SPARQL queries, *KwS database applications over RDF datasets* became a relevant research topic (Gkirtzou et al., 2015; Han et al., 2017; Zhou et al., 2007). In what follows, we refer to these alternatives as the *relational environment* and the *RDF environment*, when the underlying data are respectively stored in a relational database using the SQL query language or in an RDF dataset using the SPARQL query language.

The main motivation of this work is how to construct a KwS database application for graph databases with schema. We focus on the problem for both the relational environment and the RDF environment.

1.2   Goal and Contributions

The problem addressed in this work is how to find answers for *KwQ+* over graph databases under the assumption that the dataset or database has a schema. The solution described a method to uniformly solves both versions of the problem (RDF datasets or relational databases).

To create this tool, we identified two points in a KwS database application that are environment-dependent:

1. How the database schema is defined, and
2. The query language of the database.

Based on this observation, we designed DANKE as a flexible tool, which easily extended for new environments. The translation algorithm has three functionalities that should be implemented for each environment that we want to extend. The functionalities are:

1. Mapping the database schema into an abstract schema;
2. Finding in the database the elements that cover the keywords;
3. Mapping an abstract query into a query in the environment query language.

The *abstract schema* and the *abstract query* are respectively general representations of a schema and a query for graph databases with schema, and do not depend on the environment.

The first and key contribution of this thesis is an algorithm that translates *KwQ+* into a query in the environment query language. The algorithm can be easily extended for different environments. It explores the schema to dispense with user intervention during the translation process and to minimize the number of joins in a query. The problem of minimizing the number of joins to assemble the query is equivalent to the problem of finding a minimum Steiner Tree, an NP-Complete problem, and this is why we use approximate solutions.

The second contribution is the framework that allows extending the search algorithm for new environments. The framework is fully implemented as a tool called *DANKE* (*Da*ta and *K*nowledge R*e*trieval). The implementation is engineered to work with different RDF stores and relational DBMSs. The current implementation supports ORACLE 12c and JENA TDB, for the RDF environment, and ORACLE 12c and POSTGRES, for the relational environment. We also discuss

the tasks required to prepare the database to make the search process faster. The tasks are executed only once and depend on the environment.

Finally, the third contribution of this thesis is an extensive set of experiments to assess the correctness and the performance of the algorithm over the RDF and the relational environment. The experiments use RDF and relational versions of IMDb, which includes descriptions of artists, movies, documentaries, TV series, and even computer games, and the Mondial database, compiled from geographical Web data sources. For the experiments, we also use a relational version of the MusicBrainz database, compiled from music metadata.

1.3   Structure of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 introduces background concepts. It also discusses related work for *KwS database applications* in both environments. Chapter 3 presents the keyword search problem. Chapter 4 features the translation algorithm. Chapter 5 specifies the architecture and the implementation of the framework for each environment. Chapter 6 covers experiments to assess the algorithm. Finally, Chapter 7 contains the conclusions and indicates directions for future work.

2
Background and Related Work

This chapter provides an overview of the main concepts related to this thesis. Section 2.1 covers graph concepts. Section 2.2 defines the main tasks of an information retrieval system. Finally, Section 2.3 presents related work.

2.1   Graph Concepts

Let $G=(V,E)$ be a weighted graph, where V and E denote the set of nodes and edges respectively and, for each edge $(u,v) \in E$, $w(u,v)$ denotes the specific cost to connect u and v.

The *minimum spanning tree problem* (*MST problem*) refers to the problem of finding an acyclic subset $T \subseteq E$ that connects all of the nodes of G and whose total weight is minimum. There exist two well-known algorithms to solve the MST problem: Kruskal's algorithm and Prim's algorithm. Generally, each of them runs in time $O(|E|.log(|V|))$ using ordinary binary heaps. By using Fibonacci heaps, Prim's algorithm runs in time $O(|E|+|V|.\log(|V|))$, which improves the binary-heap implementation, if $|V|$ is much smaller than $|E|$ (Cormen et al., 2009). Figure 1 contains an example of an MST of a graph.
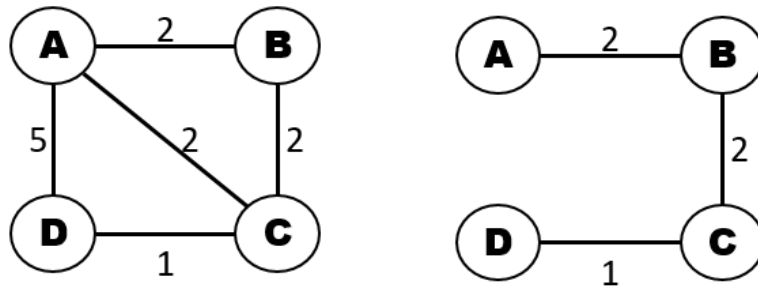
Figure 1 MST Example

The *shortest path problem* refers to the problem of finding a path $p=(v_0, v_1, ..., v_k)$ between two given nodes, v_0 and v_k, such that the sum of the weights of the edges in p is minimum.

The *all-pairs shortest-paths problem* refers to the problem of finding the shortest path for every pair of nodes. Usually, to solve this problem, the shortest path from one node to all the others is computed $|V|$ times, using Dijkstra's algorithm. Alternatively, Floyd-Warshall's algorithm, which has complexity $O(|V|^3)$, or Johnson's algorithm may be used. For dense graphs, Floyd-Warshall's algorithm is a better option than running Dijkstra's algorithm $|V|$ times (Cormen et al., 2009). Figure 2 contains an example of an all shortest-path distance table.

	A	B	C	D
A	0	2	2	3
B	2	0	2	3
C	2	2	0	1
D	3	3	1	0
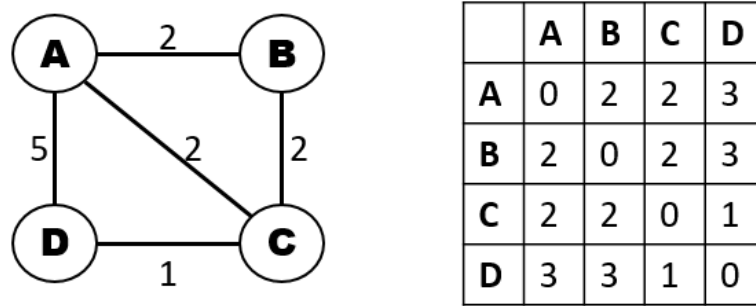
Figure 2 Example of all shortest-path distance table

The *minimum Steiner tree problem* (*MST problem*) refers to the problem of finding a minimum weight tree in G that spans a set of nodes X, with $X \subseteq V$. This problem is known to be NP-complete. In fact, it is one of the Karp's 21 NP-complete problems (Chopra & Rao, 1994).

The Steiner tree problem can be seen as a generalization of two other famous combinatorial optimization problems: the (non-negative) shortest path problem and the minimum spanning tree problem. If a Steiner tree problem contains exactly two terminals, it reduces to finding the shortest path. If, on the other hand, $X=V$, the Steiner tree problem is equivalent to the minimum spanning tree. However, while both the non-negative shortest path and the minimum spanning tree problem are solvable in polynomial time, the Steiner tree problem is NP-complete. Figure 3 contains an example of the minimum Steiner tree of a graph.
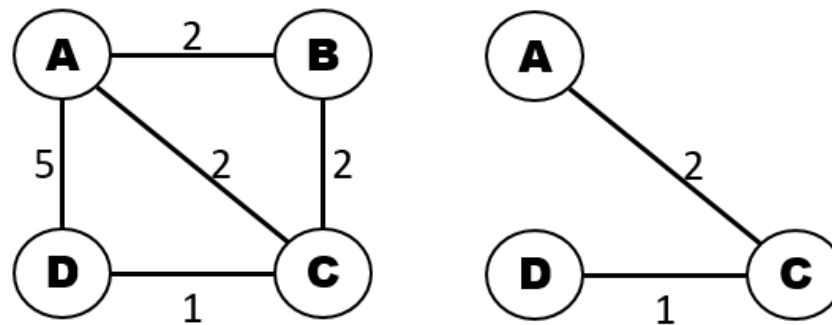
Figure 3 Minimum Steiner Tree for the nodes set {A,D}

2.2 Information Retrieval

2.2.1. Definition

An information retrieval (IR) system is a software program that manages data and helps users find the information they need. There are three basic processes an IR system has to support: the representation of the content, the representation of the user information needs, and the comparison of the two representations (Hiemstra, 2009).

In a classical IR system, the content is a database of documents, and the user needs are expressed through a keyword-based query. The document representation process is usually called the *indexing process,* and it takes place offline. The representation process of the user information needs is often referred as the *query formulation process*. In a broad sense, query formulation might denote the complete interactive dialogue between the system and the user, leading not only to a suitable query but, possibly, also to the user better understanding his information needs. The comparison of the query against the document representations is called the *matching process.* This process usually results in a ranked list of documents. The documents that satisfy the user information needs are called *relevant documents,* and in the IR system result will hopefully put the relevant documents at the top of the ranked list, minimizing the time the user has to invest in reading the documents. Figure 4 details the complete process.

A *perfect* retrieval system would retrieve only relevant documents (that is, it would have 100% *precision*) and would retrieve all such documents (that is, it would have 100% *recall*). However, perfect retrieval systems do not exist since

search statements are incomplete, and relevance depends on the subjective opinion of users.
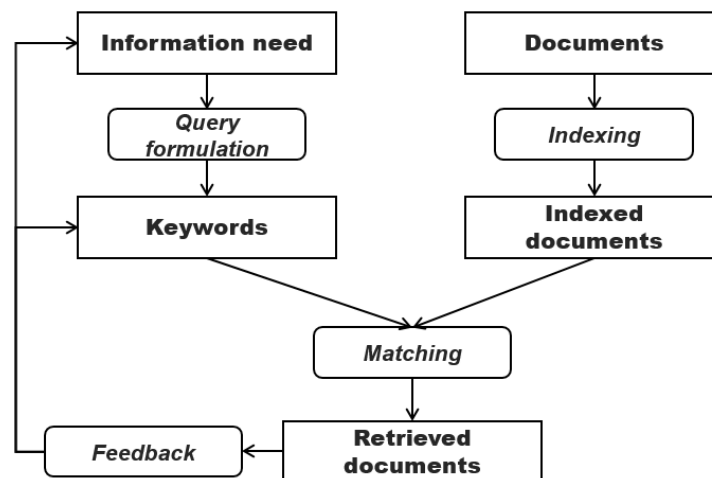
Figure 4 Classical Information Retrieval Process

An IR system, where the content is a database, may be seen as an extension of a classical system, where the database values are "documents". However, additionally, the system needs to connect the relevant documents to relate the information and compose answers.

The database representation is also called the *indexing process*; the values and connections may be indexed. The comparison of the query against the database representations wraps the *matching process* and the *connecting process*. These processes result in a ranked list of answers. Figure 5 outlines de whole process.
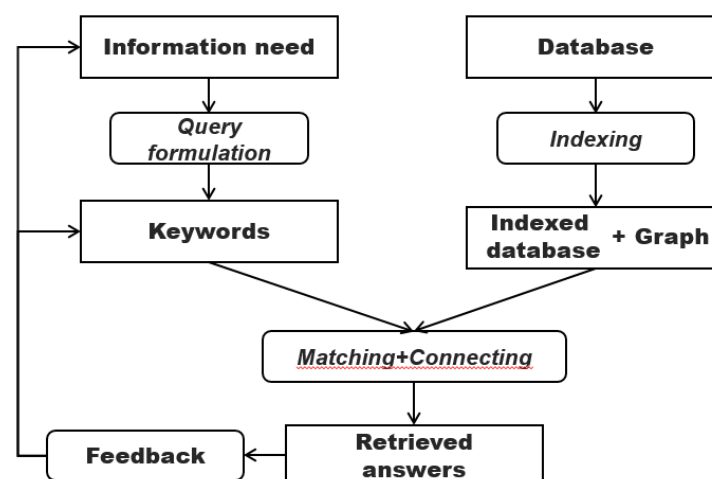
Figure 5 Information Retrieval Process for Databases

2.2.2. The matching process

As defined before, the comparison of the query against the document representations is called the *matching process*.

There are several types of matches:

1. Fuzzy match: the search query is similar to a substring of the content (find matches even when users mispell words or enter only partial words for the search).

2. Contain match: This is a type of fuzzy match, when the search query is a substring of the content.

3. Exact match: This is a type of contain match, when the search query is equal to the content.

In general, the type of the match affects the order of the ranked list of answers.

2.2.3. Disambiguation Problem

Ambiguity in natural language has long been recognized as having a detrimental effect on the performance of text-based information retrieval (IR) systems. Sometimes called the polysemy problem, the problem that a word may have more than one meaning is entirely discounted in most traditional IR strategies (Stokoe et al., 2003). If ambiguous words can be correctly disambiguated, IR performance will increase.

2.2.4. Evaluation

This section recalls the definitions of precision and average precision for a ranked list of answers, which we will use to compare and evaluate an IR system.

Let S be a list of answers, considered as the *golden standard*. An answer d is *relevant* iff $d \in S$. Let L be a list of answers.

The *precision at position k* of L for S is defined as:

$$P(k) = \frac{|\, S \cap retrieved\,(k)\,|}{|\, retrieved\,(k)\,|}$$

where *retrieved(k)* is the set of all answers in L until position k.

The *average precision* of L, concerning S, is defined as:

$$AP_L = \frac{1}{|S|} \sum_{k=1}^{n} relevance(k) * P(k)$$

where *relevance(k)* is an indicator function that returns *1*, if the answer at position k is relevant, and *0*, otherwise. Notice that the average precision of the golden standard S is $AP_S=1$, which is the target performance of a centrality measure.

2.3 Related Work

In this section, we discuss different tools or algorithms that implement keyword search over relational databases or RDF datasets. We may distinguish a tool as *schema-based*, *graph-based, or pattern-based. Schema-based* tools model the database to be queried as a graph that represents the conceptual schema. G*raph-based* tools operate directly on the data, for relational databases the nodes are tuples and an edge between two tuples denotes that they are connected by a foreign key constraint. *Pattern-based* tools hit the middle ground, in the sense that they mine patterns from the RDF dataset to be used instead of the conceptual schema. It is also useful to distinguish between *fully automatic* tools from tools that resort to user intervention during the search process. An early survey of keyword search in databases can be found in (Qin et al., 2009).

2.3.1. Schema-based Tools for Keyword Search

Usually, *schema-based* applications for relational environments explore 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) (Aditya, 2002; Agrawal et al., 2002; Bergamaschi et al., 2016; Hristidis & Papakonstantinou, 2002). DBXplorer (Agrawal et al., 2002) does not consider solutions in which keywords hit different tuples from the same relation. Furthermore, they only consider exact matches, where a keyword must match exactly an attribute value. DISCOVER (Hristidis & Papakonstantinou, 2002) does not consider that the keywords may match the metadata of the database. QUEST (Bergamaschi et al., 2016) proposes a graph structure that includes the tables, the attributes and domains of the attribute.

SPARK (Zhou et al., 2007), an *RDF environment* tool, uses the Wordnet ontology to discover the relations between the keywords and the dataset. Then, it generates all possible term mapping subsets and, finally, uses a minimal spanning tree to create a query graph for each subset. The query graph is an abstract definition of a semantic query that, using conversion rules, is translated into SPARQL. The limitations of SPARK are the high numbers of subsets that may be created for ambiguous datasets, and that it did not take into account that there may be more than one graph to connect a specific subset of term mappings.

Other examples of RDF schema-based tools are QUICK (Zenz et al., 2009), Hermes (Tran et al., 2009) and Gkirtzou et al. (2015). QUICK 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. Algorithms based on user feedback reduce the number of CNs generates and improve the precision of the results. In QUICK, the user should be familiar with RDF graphs and should know the schema of the dataset; also, he may have to select intermediate queries in many steps. In Hermes, the data graph is preprocessed to obtain a keyword index and a graph index, which is basically a summary of the original graph containing structural (schema) elements only. To compute the top-k queries, graph elements are augmented with scores, associated with structure elements and computed off-line; but scores of keyword elements are specific to the query. They also enrich every element label with semantically similar terms extracted from the Wordnet ontology. Similar to SPARK, Gkirtzou et al. (2015) find all possible subsets of the matched elements and generate a candidate SPARQL query for each combination using the notion of shortest path. They include a module to translate SPARQL queries to natural language, which avoids the user having to understand SPARQL syntax to decide the query that will be executed. Among these tools, only QUICK is not *fully automatic*.

2.3.2. Graph-based and Pattern-based Tools for Keyword Search

The challenge of *graph-based* tools is to handle the large and complex graphs induced by the database instance, which may lead to an intractable problem. Furthermore, different interpretations (with different structures) that arise due to inherent keyword ambiguities appear all mixed up in the result sets. BANKS (Aditya, 2002) is an example; the tuple graph is created based on database schema; then their algorithms work on huge data graphs ,ignoring the important structural information provided by the database schema. BANKS is not fully automatic - if multiple nodes match a keyword, the user needs to disambiguate.

The work reported in He et al. (2007) proposed pruning and accelerating the construction of efficient ranked keyword searches on schema-less node-labeled graphs, without focusing on a particular environment. Following the standard approach taken by other systems, they also restrict answers to those connected

substructures that are minimal and construct an index that is a selectively precomputation and materialization of some shortest-path information. They also propose a technique to reduce the index disk space, partitioning the data graph into blocks.

Most of the graph-based algorithms proposed in the literature work for the RDF environment (Elbassuoni & Blanco, 2011; Han et al., 2017; Le et al., 2014; Lin et al., 2018; Rihany et al., 2018; Virgilio, De et al., 2013). Elbassuoni & Blanco (2011) described a technique for retrieving a set of subgraphs that match the keywords and for ranking them based on statistical language models. Virgilio, De et al. (2013) proposed a solution that adopts the algorithm proposed in Virgilio, De (2012) to discover the connections between nodes implementing an index for RDF graphs based on the principles of tensor calculus. Le et al. (2014) and Lin et al. (2018) proposed a type-based summarization approach for the RDF data that prunes large portions of the graph that are irrelevant to the query. Han et al. (2017) proposed a two-phase framework to interpret keyword queries. In the first phase, they address the keyword disambiguation problem; a keyword query generates a set of annotated queries (entity, class or predicate edges) wherein two annotated queries do not have overlapping sets of keywords. In the second phase, they assemble a valid graph with the minimum assembly cost for each annotated query. Rihany et al. (2018) also explored Wordnet to solve the gap between the keywords of the query and the terms used in the dataset, and proposed a ranking method based on the semantic relations which have been used during the matching process.

Zheng et al. (Zheng et al., 2016) adopted a *pattern-based* approach, and proposed a systematic method to mine semantically equivalent structure patterns to summarize the knowledge graph and, thereby, circumvent the lack of an RDF schema. Yang et al. (2014) proposed to mine tree patterns that will then connect together the keywords specified by the user; the tree patterns are ordered by relevance using their size, the PageRank of the nodes, and the quality of keyword match.

2.3.3. Improved Tools for Keyword Search

The algorithms proposed by Oliveira, De et al. (2015), Zhang et al. (2014), and Wang et al. (2017) focused on improving the existing tools for keyword search.

Oliveira, De et al. (2015) discussed the problem of ranking CNs and showed that processing only the top-4 CNs, and not all CNs, improves not only the time it takes to return answers, but also the quality of the answers retrieved. Wang et al. (2017) and Zhang et al. (2014) described algorithms for keyword query rewriting and concluded that specifying the exact keywords that describe the user intention is easier to find the adequate results through keyword query.

2.3.4. The Proposed Tool

The tool described in this thesis takes advantage of the ideas proposed in the state-of-art tools summarized in the previous section to: i) generate queries with a minimal set of joins; ii) improve the efficiency, by using schema information to generate few (but good) queries (Oliveira, De et al., 2015) and creating indexes with schema information (Tran et al., 2009); and iii) handle the keyword disambiguation problem, by augmenting the elements of the schema with scores (Zenz et al., 2009), using information about the proximity of the keywords (Kumar & Tomkins, 2010) and user feedback (Zenz et al., 2009).

Differently from the other tools, we implemented a translation algorithm that: i) uses schema and query abstractions that capture what is common to all graph databases with schema, thereby allowing to extend it for any graph database; ii) match keywords with metadata and values; iii) allows matches that may be exact, contain, or fuzzy; iv) considers *KwQ+*; and v) incorporates a heuristics based on that users prefer answers that induce minimal connected graphs, that is, not only the joins are minimized but also the "size" of the answers.

The approach is fully automatic because the algorithm always produces answers without user intervention; only when the algorithm fails, that is, none of the generated queries are relevant to the user, the tool produces new answers based on the user feedback.

The tool has no mechanism to rewrite queries or to enrich the keywords with an ontology, but it has an autocomplete mechanism to help the user select terms that occur in the database.

3
The Keyword Search Problem

3.1 Advanced Keyword Query

This section presents the concepts involved in the definition of an *advanced keyword query* (*KwQ+*).

A *keyword* is a literal. A *simple keyword-based query* is a set K such that each $k \in K$ is a literal. A *boolean function* given a literal, returns a Boolean value. An *advanced keyword query* is a set K such that each $m \in K$ is either a pair $m=(k,f)$, where k is a literal and f is a *Boolean function* , or m is a literal. If m is a pair $m=(k,f)$, we denote $k_m=k$ and $f_m=f$, and if m is a literal, we denote $k_m=m$ and $f_m=\emptyset$.

We say that a keyword k and a literal v *match* iff k and v are similar according to a given similarity function *sim* and a given threshold μ, that is, $sim(k,v)>\mu$.

3.2 Answers for an Advanced Keyword Query

3.2.1. RDF Environment

This section presents a formal definition for the notion of an answer for a keyword-based query over an RDF dataset.

The RDF environment assumes that each RDF dataset T follows an RDF schema S, with $S \subseteq T$, that is, the RDF schema is indeed defined and is part of the RDF dataset.

Let K be a *KwQ+* query. We say $m \in K$ has a *metadata match* with a triple $(r,p,v) \in S$ iff r is a class or property defined in S and k_m and v match. We say that $m \in K$ has a *data match* with a triple $(r,p,v) \in T{-}S$ iff k_m and v match (note that $(r,p,v) \in T{-}S$ and, hence, the triple is not part of the schema).

An *answer* for K over T is defined as a set A of triples in T, partitioned into three sets, A_{CM}, $A_{PM,}$ and A_{DM}, such that there are three possibly empty subsets of K,

denoted K/A_{CM}, K/A_{PM} and K/A_{DM}, the members of K that are *matched* by A, such that:

(1) For each $m \in K/A_{CM}$, there is $(r,p,v) \in A_{CM}$ such that m has a metadata match with (r,p,v), and r is declared as a class in S.

(2) For each $m \in K/A_{PM}$, there is $(r,p,v) \in A_{PM}$ such that m has a metadata match with (r,p,v), and r is declared as a property in S.

(3) For each $m \in K/A_{DM}$, there is $(r,p,v) \in A_{DM}$ such that m has a data match with (r,p,v).

(4) For each $(r,p,v) \in A_{CM}$, there is $(s,\mathsf{rdf:type},t) \in A_{DM}$ such that $t=r$ or t is a subclass of r in S.

(5) For each $(r,p,v) \in A_{PM}$, there is $(s,t,l) \in A_{DM}$ such that $t=r$ or t is a subproperty of r in S, and $f_m = \emptyset$ or $f_m(l)=true$, where $m \in K/A_{PM}$ and m has a metadata match with (r,p,v).

(6) G_{DM}, the graph induced by A_{DM}, is connected.

(7) There is no other answer B for K over T such that B matches more keywords in K than A.

As expected, Conditions (1), (2) and (3) say that a keyword k may have a metadata match or a data value match with a triple (r,p,v) of the answer A. Conditions (4) and (5) are not so obvious, though. They capture the interpretation that, if the user selects a class or a property (via a keyword), he actually wants an instance (and not all instances) of that class or property (other instances may be returned upon request). Specifically, Condition (5) assures that values of the properties that match with keywords that have a Boolean function associated will satisfy it. Condition (6) avoids disconnected answers. Condition (7) requires that an answer must match as many keywords in K as possible. Also, Conditions (1), (2), and (3) do not require that all keywords in K be matched in an answer.

3.2.2. Relational Environment

This section presents a formal definition for the notion of an answer for a keyword-based query over a relational database. We indicate how to adjust the definitions in Section 3.2 for the relational environment.

As usual, a *relation scheme* is denoted as $U[P_1,...,P_n]$, where U is the *name* and $P_1,...,P_n$ are the *attributes* of the scheme. A foreign key is an expression of the

form $F(U:L,V:M)$, where F is the *name* of the foreign key, U and V are names of relation schemes, and L and M are lists of attributes of U and V, respectively, with the same length. We say that $F(U:L,V:M)$ *connects* U to V.

A *relational schema* is a pair $S=(\Sigma,\Phi)$ such that Σ is a set of relation schemes and Φ is a set of constraints. A schema $S=(\Sigma,\Phi)$ induces a labeled multigraph $G_S = (N_S, E_S, EL_S)$ such that: $N_S=\Sigma$ and there is an arc $(U,V)\in E_S$, which EL_S labels with F, iff there is a foreign key $F(U:L,V:M)$ in Φ. Note that G_S is a multigraph since there might be more than one foreign key between the same pair of schemes.

A *consistent database state* σ of $S=(\Sigma,\Phi)$, or simply a *database with schema S*, is defined as usual and assigns a relation $\sigma[U]$ to each relation scheme $U\in\Sigma$ so that all constraints in Φ are satisfied. A set T of tuples from the relations in σ induces a labeled multigraph $G_A=(N_T,E_T,EL_T)$ such that $N_T=T$ and there is an arc (u,v) in E_T, which EL_T labels with F, iff $u\in\sigma[U]$, $v\in\sigma[V]$, with $u[L]=v[M]$, and there is a foreign key $F(U:L,V:M)$ in Φ.

Let K be a $KwQ+$. An $m\in K$ has a *metadata match* with a relation scheme or an attribute P in S with description v iff k_m and v match. A keyword $m\in K$ has a *data match* with $t[P]$, where U is a relation scheme in S, P is an attribute of U, and $t\in\sigma[U]$, iff k_m and v match.

An *answer* for K over a database σ, with relation schema S, is a triple $A=(A_{SM},A_{AM},A_{TM})$, where

- A_{SM} is a set of relation scheme of S
- A_{AM} is a set of pairs (U,P), where U is a relation scheme of S and P is an attribute of U
- A_{TM} is a set of triples (U,P,t), where U is a schema name of S, P is an attribute of U, and $t\in\sigma[U]$

such that there are three possibly empty subsets of K, denoted K/A_{SM}, $K/A_{AM,}$ and K/A_{TM}, the members of K that *matched by A*, such that:

(1) For each $m\in K/A_{SM}$, there is U in A_{SM} such that there a metadata match between m and the description of U.

(2) For each $m\in K/A_{AM}$, there is (U,A) in A_{AM} such that there is a metadata match between m and the description of A in U.

(3) For each $m \in K/A_{TM}$, there is (U, A, t) in A_{TM} such that there a data match between m and $t[A]$.

(4) For each U in A_{SM}, there is $(U,A,t) \in A_{TM}$, and $f_m = \emptyset$ or $f_m(t) = true$, where $m \in K/A_{AM}$ has a metadata match with the description of A in U.

(5) G_{DM}, the graph induced by A_{SM}, is connected.

(6) G_{TM}, the multigraph induced by the tuples in A_{TM}, is connected.

(7) There is no other answer B for K over T such that B matches more keywords in K than A.

3.3 User Intentions

As we mention in Section 2.2, in general, search statements are incomplete, and relevance depends on the subjective opinion of users. Given a set of possible answer for a *KwQ+*, which would be the most interesting for the user?

Generally, there are two obstacles to a *KwS database system*. First, it is the ambiguity of keywords. Given a keyword, we may have multiple ways to interpret the keyword. A system should figure out which interpretation is correct, given the context of the keywords. The second obstacle is the ambiguity of query structures. Even if each keyword has been correctly interpreted, how to represent the complete query intention is also a challenging task.

Example: Ambiguous query.

Given $K=\{$*"Panama"*, *"City"*, *"Population"*$\}$, the user intentions may be interpreted as:

Intention 1. the *population* of *"Panama City"*,

Intention 2. the *cities* of *Panama* and the *population* of each one,

Intention 3. the *population* of *Panama* and the *cities* of *Panama*.

3.3.1. Assumptions

To deal with this problem, we consider the following assumptions about user intentions:

U_1. The user selects resources by specifying keywords that match the resources' property values.

U_2. The user prefers resources that, individually, match as many keywords as possible.

U_3. The user prefers to observe as few resources as possible.

U_4. The user prefers to observe resources that are interrelated.

Summarizing, based on assumptions U_1, U_2, U_3, and U_4, we consider that users prefer minimal answers, those that induce minimal, connected graphs and that match as many keywords as possible. The next section introduces the formal definition of the minimal answer.

3.3.2. Minimal Answer

The answer definitions given in Section 3.2 do not force an answer A to be minimal but based on the assumptions users prefer minimal answers. To define minimal answers, we introduce a total order between answers for both environments.

Minimal answer in the RDF environment: A total order between answers, denoted "$\leq$", such that $A \leq B$ iff $|A| \leq |B|$, where $|\alpha|$ denotes the cardinality of a set α. An answer A for K over T is *minimal* iff there is no other answer B for K over T such that $B \leq A$.

Example: Comparing answers in RDF dataset.

Consider:

- $A_1 = \{A_{CM}, A_{PM}, A_{DM}\}$, with $A_{CM} = \{\}$, $A_{PM} = \{(:population, rdfs:label, "Population")\}$ and $A_{DM} = \{(:panama\_city, :name, "Panama City");$ $(:panama\_city, :population, "880 691")\}$.

- $A_2 = \{A_{CM}, A_{PM}, A_{DM}\}$, with $A_{CM} = \{(:city, rdfs:label, "City")\}$, $A_{PM} = \{(:population, rdfs:label, "Population")\}$ and $A_{DM} = \{(:panama, :name, "Panama");$ $(:colon, rdf:type, :city);$ $(:colon, :population, "253 366");$ $(:colon, :of\_country, :panama)\}$

- $A_3 = \{A_{CM}, A_{PM}, A_{DM}\}$ with $A_{CM} = \{(:city, rdfs:label, "City")\}$, $A_{PM} = \{(:population, rdfs:label, "Population")\}$ and $A_{DM} = \{(:panama, :name, "Panama");$ $(:colon, rdf:type, :city);$ $(:panama, :population, "4 162 618");$ $(:colon, :of\_country, :panama)\}$

Hence, we have, by definition of total order, that $|A_1| \leq |A_2| \leq ||A_3|$.

Minimal answer in Relational environment: A total order between answers "≤" such that $A \leq B$ iff $|A_{SM} \cup A_{AM} \cup A_{TM}| \leq |B_{SM} \cup B_{AM} \cup B_{TM}|$, where $A=(A_{SM}, A_{AM}, A_{TM})$ and $B=(B_{SM}, B_{AM}, B_{TM})$. An answer A for K over a database σ, with relation schema S, is minimal iff there is no other answer B for K over σ such that $B \leq A$.

Example: Comparing answers in relational database.

Consider:

- $A_1=\{A_{SM}, A_{AM}, A_{TM}\}$ with $A_{SM}=\{\}$, $A_{AM}=\{(population, city)\}$ and $A_{TM}=\{(city, name, "Panama City",); (city, population, "880 691"), \}$

- $A_2=\{A_{SM}, A_{AM}, A_{TM}\}$ with $A_{SM}=\{(city)\}$, $A_{AM}=\{(city, population)\}$ and $A_{TM}=\{(country, name, "Panama"); (city, name, "Colon"); (city, population, "253 366"); (city, of\_country, "Panama")\}$

- $A_3=\{A_{SM}, A_{AM}, A_{TM}\}$ with $A_{SM}=\{(city)\}$, $A_{AM}=\{(country, population)\}$ and $A_{TM}=\{(country, name, "Panama"), (city, name, "Colon"); (country, population, "4 162 618"); (city, of\_country, "Panama")\}$

    Hence, we have, by definition of total order, that $|A_1| \leq |A_2| \leq ||A_3|$.

3.3.3. Matches Filtering

Additionally, to help assist in the keyword disambiguation problem, we also assume that the order of the keywords and the type of Boolean function may also contain hints about the user intentions.

Example: Keyword Order.

Consider the query $K=\{("country"), ("city"), ("population")\}$. Intuitively, it makes more sense to interpret K as requesting the population of the cities, since "*city*" and "*population*" appear next to each-other, than to interpret K as requesting the population of the countries. Thus, we may discard metadata matches with the keyword "*population*" that do not represent the population of the cities.

As another example, consider the query $K=\{(\text{*"city"*}),\ (\text{*"population"*}),\ (\text{*"country"*})\}$. The user probably requires the population of a city or country but not of a province or something else. Again. we may discard metadata matches with the keyword *"population"* that do not represent the population of cities or countries.

Finally, for the query $K=\{\ (\text{*"population"*})\}$, we may discard any metadata match.

Example: Type of the Boolean function.

Given the query $K=\{(\text{*"census"*},\ f_1)\}$, where f_1 is a function that returns *True* iff a literal is equal to the date "*January 18, 2020*". For the metadata matches with the keyword *"census"*, we may therefore discard those properties or attributes for which do not have values that are dates.

In what follows, we use *FilterByCloseEntity* and *FilterByDataType* when we discard matches due to the order of the keywords and to the type of the Boolean function, respectively.

3.4   Chapter Conclusion

In this chapter, we presented the formal definition of the Keyword Search problem, for RDF dataset and relational databases. We also discussed some assumptions about the user intentions, such as users prefer minimal answer, and the order of the keywords may have information about the user intentions. Based on the definition of the problem, as well as our assumptions about the user intentions, we proposed the translation algorithm presented in the next section.

4
The Keyword Search Algorithm

This chapter presents the algorithm that translates a *KwQ+* query to a SPARQL or SQL. Section 4.1 presents the general definitions used by the translation algorithm. Section 4.2 details the translation algorithm. Finally, Section 4.3 presents the feedback algorithm.

4.1   General definitions

This section introduces some general definitions that are used in what follow.

4.1.1. Graph Database Schema

In this section, we define the notion of an *abstract schema* that is independent of the environment and argue that we can map any database schema into an abstract schema.

An *entity* is a pair $e=(r,L)$, where $r \in (0,1]$, and L is a set of literals; we denote *ranking*$(e)=r$ and *labels*$(e)=L$.

Example: $e_1=(0.9,\{$*"Country"*$\})$.

A *data property* is a tuple $p=(r,e,L,D)$, where $r \in (0,1]$, e is an entity, L is a set of literals, and D is a set of datatypes; we denote *ranking*$(p)=r$, *domain*$(p)=e$, *labels*$(p)=L$ and *dtypes*$(p)=D$.

Example: $p_1=(0.5, (0.9,\{$*"Country"*$\}), \{$*"Population"*$\}, \{$*xsd:numeric*$\})$

A *join* is a triple $j=(r, e_1, e_2)$, where $r \in (0,1]$ and e_1 and e_2 are entities; we denote *ranking*$(j)=r$, *domain*$(j)=e_1$ and *range*$(j)=e_2$.

Example: $j_1 = (1.0, (0.9, \{\text{``Country''}\}), (0.8, \{\text{``City''}\}))$.

An *abstract schema*, or simply a *schema*, is a tuple $S=(E,P,J,\eta)$, where:

- E is an entity set
- P is a property set
- J is a join set
- η: $E \cup P \cup J \rightarrow DATABASE\_ELEMENT$ maps a schema element to a database schema element.

We denote $E(S)=E$, $P(S)=P$, $J(S)=J$ and $\eta(S)=\eta$.

A schema S induces an undirected multi-graph $G(S)=(V_s,E_s)$, where $V_s=E(S)$ and $E_s=J(S)$.

Table 1 maps the concepts of the *schema* with the concepts of Relational Databases and RDF Datasets.

Table 1 Mapping of Schema Concepts

Schema Concept	Relational Databases	RDF Datasets
entity	Table	rdfs:Class
property	Attribute	owl:DatatypeProperty
join	Foreing key	owl:ObjectProperty

A *resource* of the schema is an *entity*, a *property* or a *join*.

The rest of this section presents two examples of how to construct the schema of a database. The first example shows how to construct a schema for an RDF dataset, and the second shows how to construct a schema for a relational database. In Chapter 5, we explain more formally how to map the database schema or an RDF schema to a schema.

Example: An abstract schema for an RDF dataset.

For RDF datasets, the entities correspond to the classes declared in the RDF schema using *rdfs:Class*, the properties to the RDF datatype properties declared in the RDF schema using *owl:DatatypeProperty*, the joins to the RDF object properties declared in the RDF schema using *owl:ObjectProperty*, and η relates an element to the URI of the corresponding resource declared in the RDF schema. For the RDF schema in Figure 6, the schema would be the tuple (E,P,J,η), where:

- $E=\{e_1,e_2\}$, where $e_1=(0.9, \{\text{"Country"}\})$ and $e_2=(0.8, \{\text{"City"}\})$.

- $P=\{p_1,p_2,p_3,p_4\}$, where

  - $p_1=(0.9, e_1, \{\text{"Population"}\}, \{xsd{:}numeric\})$

  - $p_2=(0.8, e_2, \{\text{"Population"}\}, \{xsd{:}numeric\})$

  - $p_3= (0.9, e_1, \{\text{"Name"}\}, \{xsd{:}string\})$

  - $p_4=(0.8, e_2, \{\text{"Name"}\}, \{xsd{:}string\})$

- $J=\{j_1\}$, where $j_1=(0.9, e_1, e_2)$

- $ɱ=\{(e_1, :country), (e_2, :city), (p_1, :population{+}:country), (p_2, :population{+}:city),$

  $(p_3, :name{+}:country), (p_4, :name{+}:city), (j_1, :of\_country)\}$
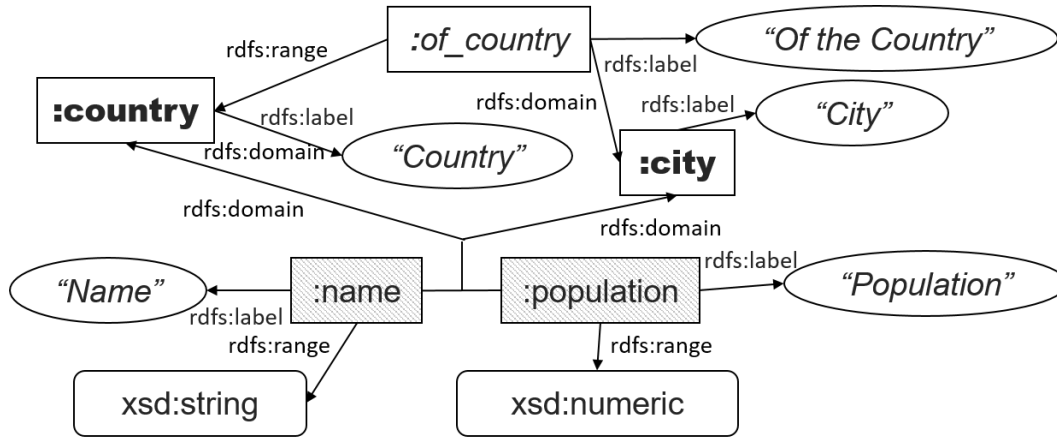
Note: Bold square elements are *rdfs:Class*, dashed square elements are *owl:DatatypeProperty* and italic square elements are *owl:ObjectProperty*.

Figure 6. RDF Dataset.

Example: An abstract schema for a relational database.

For relational datasets the entities are the tables, the properties the attributes of the tables, the joins the foreign key constraints, and $ɱ$ relates an element with its correspondent name in the database. For the database in Figure 7, the schema is the tuple $(E,P,J,ɱ)$, where:

- $E=\{e_1,e_2\}$, where $e_1=(0.9, \{\text{"Country"}\})$ and $e_2=(0.8, \{\text{"City"}\})$

- $P=\{p_1,p_2,p_3,p_4\}$, where

  - $p_1=(0.9, e_1, \{\text{"Population"}\}, \{NUMBER\})$

  - $p_2=(0.8, e_2, \{\text{"Population"}\}, \{NUMBER\})$

  - $p_3=(0.9, e_1, \{\text{"Name"}\}, \{VARCHAR\})$

○ $p_4= (0.8, e_2, \{"Name"\}, \{VARCHAR\})$

- $J=\{j_1\}$, where $j_1=(0.9, e_1, e_2)$

- m=$\{(e_1, country), (e_2, city), (p_1, population+country), (p_2, population+city), (p_3,$ $name+country),\ (p_4, name+city), (j_1, city\_country)\}$

TABLE_NAME	COMMENTS	
country	"Country"	
COLUMNS		
COLUMN_NAME	DATA_TYPE	COMMENTS
name	VARCHAR	"Name"
population	NUMBER	"Population"

CONSTRAINTS
CONSTRAINT 'city_country' FOREIGN KEY city (of_country) REFERENCES country (name)

TABLE_NAME	COMMENTS	
city	"City"	
COLUMNS		
COLUMN_NAME	DATA_TYPE	COMMENTS
name	VARCHAR	"Name"
population	NUMBER	"Population"
of_country	VARCHAR	"Of the country"

Figure 7. Relational Database.

4.1.2. Boolean Functions and Buckets

In what follows, we extend the definition of a Boolean function f, saying that f may be expecting a literal l of a specific datatype, denoted *datatype(f)*.

Example: Let f be a function that returns *True* iff a literal is equal to the date "*January 18, 2020*". Then, we have that *datatype(f)= "date"*.

A *bucket* associates an element of the schema (entity or property) with a set of keywords.

An *entity bucket b_E* is a pair $b_E = (e, K)$, where e is an entity, and K is a set of keyword; we denote *entity(b_E) = e* and *keywords(b_E)=K*.

A *property bucket b_P* is a triple $b_P=(p,K,f)$, where p is a property, K is a set of keywords, and f is a Boolean function; we denote *property(b_P)=p*, *keywords(b_P)=K* and *bexpr(b_P)=f*.

A *value bucket b_v* is a pair $b_V=(p, K)$, where p is a property and K is a set of keyword; we denote *property(b_V)=p* and *keywords(b_V)=K*.

For a bucket set B, we define that *keywords(B)=$\bigcup_{b\in B}$ keywords(b)*, $B_E(B)$ is the bucket set with the entity buckets in B, $B_P(B)$ is the bucket set with the property buckets in B, and $B_V(B)$ is the bucket set with the value buckets in B.

4.1.3. Nucleus and Abstract Query

A *nucleus* associates an entity bucket with property buckets belonging to the same entity. More precisely, a *nucleus n* is a triple $n=(b_e,B_P,B_V)$, where b_e is an entity bucket, B_P is a set of property buckets, and B_V is a set of value buckets such that, for each bucket b in B_P or B_V, $domain(p) = entity(b_e)$, where $p = property(b)$. We denote $entity(n) = entity(b_e)$ and $keywords(n) = keywords(b_e) \cup keywords(B_P) \cup keywords(B_V)$.

Given a set *N* of nucleuses, the entity set *E induced* by *N*, denoted *entities(N)*, is defined as $\bigcup_{n \in N} (entity(n))$.

An *abstract query a_q* is a pair $a_q=(N,J)$, where *N* is a nucleus set and *J* is a join set. An abstract query induces a graph, where the nodes are the nucleuses and the edges are the joins of the query. An edge *j* joins the nucleuses n_1, n_2 iff $domain(j)=entity(n_1)$ and $range(j)=entity(n_2)$ or $domain(j)=entity(n_2)$ and $range(j)=entity(n_1)$.

4.2 Translation Algorithm

A naïve algorithm that finds all matches, generates all possible answers with the matches, and then selects the minimal answers is not feasible for a large and ambiguous dataset. For instance, if we search *IMDb RDF* dataset looking for the keyword *rocky* in some value, we find 9,600 triples, for the keyword *sylvester*, we find 4,237 triples, and for keyword *stallone*, we find 1,242 triples. This would generate billions of possible answers to be analyzed.

Therefore, we looked for strategies to minimize these problems. The first strategy is that the translation algorithm proposed should be *schema-based*. Among the advantages of adopting this strategy, we have: (i) the algorithm does not analyze the triples or tuples themselves, but rather it groups the matches at the level of entities and properties; (ii) the algorithm does not generate the connection between every collection of triples or tuples, but rather it generates a graph at the schema level; iii) the algorithm synthesizes SPARQL or SQL query, leaving the responsibility for finding the actual instances, and paths between instances, to the database management systems, which were designed for this purpose. The second strategy is to generate a few potential queries that, possibly, induce minimal

answers. To find the potential queries, the translation algorithm implements heuristics that try to capture the user intentions, addressing the problems of which interpretation of the keywords is correct and which query structure should be produced.

Figure 8 presents an overview of the main steps of the translation algorithm. Given a set of matches, the first heuristic, called the *match heuristic*, discards some matches and builds the buckets from the remaining matches. The heuristic discards matches according to a threshold, the order of the keywords and the Boolean function present in the keyword query. The Build Buckets algorithm implements the *match heuristic*. The second heuristic, called the *nucleuses heuristic*, implements a greedy algorithm that, given the nucleuses built from the buckets, constructs the small nucleus set that best covers the keywords. The Build Nucleuses and Select Nucleuses algorithms implement the *nucleuses heuristic*. The last heuristic, called the *connection heuristic*, find the best, minimal way of connecting the entities in the nucleuses. The Connect Entities algorithm implements the *connection heuristic*. The abstract query is created from the nucleus resulting from Step3 and the joins resulting from Step 4. Then, the abstracts query is compiled into a structured query in SQL or SPARQL, and executed in the database to compute the results of the keyword query.
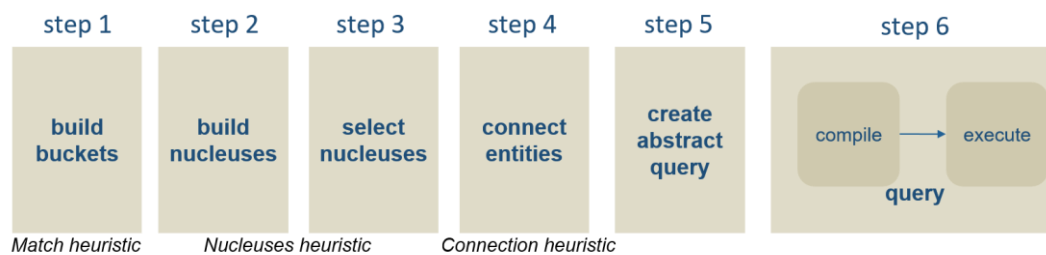
Figure 8 Algorithm Overview

Section 4.2.1 describes the pre-processing algorithm that builds an index with information about the shortest paths between the entities of the schema; this index is used by the heuristics. Section 4.2.2 corresponds to the *match heuristic*. Sections 4.2.3 and 4.2.4 describe the *nucleuses heuristic*. Section 4.2.5 details the connection heuristic. Finally, Section 4.2.6 outlines the entire algorithm.

4.2.1. Shortest Path Index

This section presents a pre-processing algorithm that basically computes an index containing the information about the shortest paths between each pair of entities in the schema. The shortest path index construction does not depend on the user query but on the schema, so it is built once by database, but if the database schema changes, it needs to be updated. The algorithms that we will introduce in Sections 4.2.4 and 4.2.5 use this index to find the minimal join set to build connected answers.

The shortest path index is defined as two functions: δ, or the shortest-path distance values, and π, or the shortest-path building information.

As we mentioned in Section 2.1, the Floyd-Warshall algorithm is an alternative to compute the all-pairs shortest-paths distances in a graph (Cormen et al., 2009). The performance and space complexity of the classical Floyd-Warshall algorithm (in short *FW-C*) depends on the number of nodes in the graph, in our case the number of entities in the schema, that is $O(|E(S)|^3)$ and $O(|E(S)|^2)$, respectively. *FW-C* outputs:

- δ: $E(S) \times E(S) \rightarrow \mathbb{R}$, where $\delta(e_1,e_2)=n$ indicates that n is the weight of the shortest path from entity e_1 to entity e_2

- π: $E(S) \times E(S) \rightarrow E(S)$, where $\pi(e_1, e_2)=e_3$ indicates that e_3 is the successor of entity e_1 in the shortest path to entity e_2.

With δ we know the length of the shortest paths and with π we can build the shortest path itself. Note that with π we can build a unique shortest path p between e_1 and e_2, but other shortest paths may exist. Figure 9 shows a graph with two shortest paths between nodes *B* and *D*.
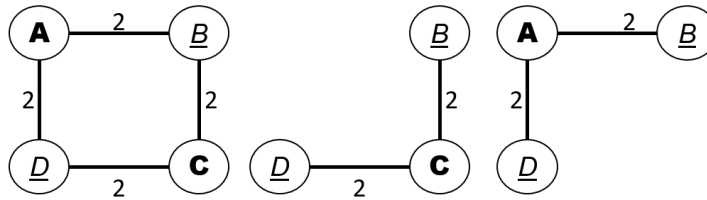
Figure 9. Example of two shortest path between the nodes B and D.

Following the idea that users prefer minimal answers, we are interested in answers with fewer joins. We also consider that the *FW-C* is limited, and only finds one shortest path between two entities, although there might be others.

We use a modification of *FW-C,* that we call *FW-2,* where the computation of π is adjusted to find more than one shortest path between two nodes. In *FW-2,* $\pi(e_1,e_2)=P$, where P is the set that contains the pairs (e,j) such that there exists a shortest path p between e_1 and e_2, where e is the successor of e_1 on the shortest path p based on the join j. With π, we recursively build all shortest paths between two entities (for more details, see Section 4.2.5). As in *FW-C*, $\delta(e_1,e_2)=n$ indicates that n is the weight of the shortest path from entity e_1 to entity e_2.

Given the graph G induced by the *schema S*, Algorithm 1 computes the shortest path index, which is δ and π. Comparing with the implementation of *FW-C* in Cormen et al. (2009), the differences are in lines from 16 to 18 that build π. It is easy to realize that the performance complexity of *FW-2* is $\Omega(|E(S)|^3)$, in the best case, when there is a unique shortest path for each pair of vertex, and is $O(|E(S)|^4)$, for the worst cases. Similarly, the space complexity is $\Omega(|E(S)|^2)$ and $O(|E(S)|^3)$.

Algorithm 1 FW-2
Input: *G*=(*E*(*S*),*J*(*S*))
Output: δ, π

1. δ, $\pi \leftarrow$ {}, {}
2. **for each** entity e_1 in $E(S)$
3.   **for each** entity e_2 in $E(S)$
4. $\delta(e_1,e_2) \leftarrow$ **if** $e_1=e_2$ **then** 0 **else then** ∞
5. $\pi(e_1,e_2) \leftarrow$ {}
6. **for each** join j in J(S)
7. $\delta(domain(j),range(j)) \leftarrow 1$
8. $\pi(domain(j),range(j)) \leftarrow (range(j),j)$
9. $\delta(range(j),domain(j)) \leftarrow 1$
10. $\pi(range(j),domain(j)) \leftarrow (domain(j),j)$
11. **for each** entity e_K in $E(S)$
12.   **for each** entity e_1 in E(S)
13.   **for each** entity e_2 in $E(S)$
14.   **if** $\delta(e_1, e_2) > \delta(e_1, e_K) + \delta(e_k, e_2)$ **then**
15. $\delta(e_1, e_2) \leftarrow \delta(e_1, e_K) + \delta(e_k, e_2)$
16. $\pi(e_1, e_2) \leftarrow \pi(e_1, e_k)$
17.   **elseif** $\delta(e_1, e_2) = \delta(e_1, e_K) + \delta(e_k, e_2)$ **then**

```
18.              π(e₁, e₂) ← π(e₁, e₂) ∪ π(e₁, eₖ)

19. return δ, π
```

Example: Pre-processing algorithm.

Given the graph $G=(E,J)$, Figure 10 shows G, $E=\{e_1,e_2,e_3,e_4\}$, where $e_1=(1,\{''e_1''\})$, $e_2=(0.9,\{''e_2''\})$, $e_3=(0.8, \{''e_3''\})$ and $e_4=(0.7, \{''e_4''\})$ and $J=\{j_1, j_2, j_3, j_4, , j_5\}$, where $j_1=(1.0,e_1,e_2)$, $j_2=(0.9,e_1,e_4)$, $j_3=(0.9,e_1,e_3)$, $j_4=(0.9,e_2,e_4)$, and $j_5=(0.9,e_3,e_4)$.
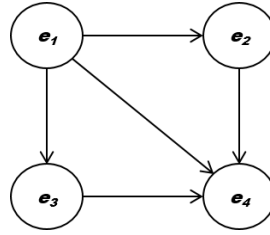
Figure 10 Graph Example

The output of the Algorithm 1 for G is:

Table 2. Example of δ.

	e_1	e_2	e_3	e_4
e_1	0	1	1	1
e_2	1	0	2	1
e_3	1	2	0	1
e_4	1	1	1	0

Table 3. Example of π.

	e_1	e_2	e_3	e_4
e_1	[]	$[(e_2, j_1)]$	$[(e_3, j_3)])$	$[(e_4, j_2)]$
e_2	$[(e_1, j_1)]$	[]	$[(e_1, j_1), (e_4, j_4)]$	$[(e_4, j_4)]$
e_3	$[(e_1, j_3)]$	$[(e_1, j_3), (e_4, j_5)]$	[]	$[(e_4, j_5)]$
e_4	$[(e_1, j_2)]$	$[(e_2, j_4)]$	$[(e_3, j_5)]$	[]

Note that, given π, we know that there are at least two shortest paths between e_2 and e_3.

4.2.2. Building Buckets

This section outlines *Build Buckets* algorithm, which is the first step of the translation algorithm, and also corresponds to the match heuristic. The concept of bucket is related to assumption U2 (see Section 3.3.1), which says that users prefer resources that, individually, match as many keywords as possible; a bucket

therefore contains the information about how many and which keywords an element of the schema covers. This algorithm associates an element e of the schema S with a set of keywords K such that, for each $k \in K$, k matches e. Some of the matches are discarded according to a threshold, to the order of the keywords, and to the Boolean function presented in the query. Algorithm 2 presents the pseudo-code of this process.

Algorithm 2 has as inputs a $KwQ+$ K, the schema S, the functions f_{ME}, f_{MP} and f_{MV}, and a similarity threshold μ. The algorithm finds the matches between a keyword and an element of the schema, using the functions f_{ME}, f_{MP}, and f_{MV}, in lines from 5 to 7. Those functions need to be implemented for each environment. The full details about the implementation of the functions for each environment are presented in Section 5.5.1.

In more detail, f_{ME} is a function that maps a schema, a keyword and a number $\mu \in (0,1]$ to a subset of $E(S)$ and is such that $f_{ME}(S, k, \mu) = E'$ indicates that each entity in E' has at least a label that matches the keyword k with similarity greater than the threshold μ.

f_{MP} is a function that maps a schema, a keyword and a number $\mu \in (0,1]$ to a subset of $P(S)$ and is such that $f_{MP}(S,k,\mu) = P'$ indicates that properties P' has at least a label that matches the keyword k with similarity greater than the threshold μ.

f_{MV} is a function that maps a schema, a keyword and a number $\mu \in (0,1]$ to a subset of $P(S)$ and is such that $f_{MV}(S, k, \mu) = P'$ indicates that the properties in P' have at least one value that matches the keyword k with similarity greater than the threshold μ.

Given the elements of the schema that match a keyword, the algorithm builds bucket sets B_o and B_f, where B_o contains the buckets built from all matches found, in line 8, and B_f contains the buckets built from the *relevant* matches.

Section 3.3.3 introduces the notion that some matches may be *irrelevant*. To build B_f, we exclude irrelevant matches using the functions *FilterByCloseEntity* and *FilterByDatatype,* in line 9. *FilterByCloseEntity* excludes those property matches that do not have as domain any of previous entity matches or the next entity match. *FilterByDatatype* excludes those that cannot be applied to the properties that match the keyword because of the Boolean function associated with the keyword.

More formally, we have:

FilterByCloseEntity: Two keywords *k* and *k'* are close in a *KwQ+ K*, if *k* occurs immediately after or before *k'* in *K*. If a keyword *k* matches an entity *e*, and another keyword *k'*, close to *k*, matches a set of properties *P*, the properties $p \in P$ such that *domain(p)≠e* have a better chance of being *irrelevant* elements for the user. Given a property set *P* and an entity set *E*, *FilterByCloseEntity* is then defined as:

$$FilterByCloseEntity\ (P, E)$$
$$= \begin{cases} \{p \in P | domain(p) \in E\}, \text{if } (\exists p \in P)(domain(p) \in E) \\ P \qquad\qquad\qquad\qquad\qquad\qquad otherwise \end{cases}$$

FilterByDatatype: Given a property *p* and a Boolean function *f*, we say that *f can be applied* to *p* iff the *datatype(f)∈dtypes(p)*. If $m \in K$, where *K* is a *KwQ+* query and k_m matches a set of properties *P*, the properties $p \in P$ such that *f* cannot be applied to *p* have a better chance of being *irrelevant* elements for the user. Given a property set *P* and a Boolean function *f*, *FilterByDatatype* is then defined as:

$$FilterByDatatype(P, f)$$
$$= \begin{cases} \{p \in P| f \text{ can be applied to } p\} \ \exists_{p \in P} f \text{ can apply to } p \\ P \qquad\qquad\qquad\qquad\qquad otherwise \end{cases}$$

B_o does not involve in any other step of the translation algorithm, but the Feedback algorithm uses it. How to use user feedback to generate different answers is discussed in section 4.3.

Algorithm 2 Build Buckets
Input: *K, S, f_{ME}, f_{MP}, f_{MV}, μ*
Output: B_o, B_f

1. Initialize B_o as an empty set
2. Initialize B_f as an empty set
3. Initialize me_{-1}, mp_{-1}, k_{m-1}, f_{-1}, mv_{-1} for save the information about de previous iteration
4. **for each** $m \in K$
5.   Create the set *me* that contains the entities in *S* that have metadata match with *m*, that is the return of $f_{ME}(S, k_m, μ)$
6.   Create the set *mp* that contains the properties in *S* that has metadata match with *m*, that is the return of $f_{MP}(S, k_m, μ)$
7.   Create the set *mv* that contains the properties in S that has data match with *m*, that is the return of $f_{MV}(S, k_m, μ)$

```
8.    Update the buckets in Bₒ with the information that the elements in me,mp and mv
      match with the keyword kₘ.
9.    Consider the sets
```

- $mp'=FilterByCloseEntity(mp, me_{-1}) \cap FilterByDatatype(mp, f_m)$, that is, the properties in mp, filtered by the previous entities me_{-1} and the Boolean function f_m;

- the set $mp_{-1}'= FilterByCloseEntity(mp_{-1}, me) \cap FilterByDatatype(mp_{-1}, f_{-1})$, that is, the properties in mp_{-1}, filtered by the current entities in me and the Boolean function f_{-1};

- the set $mv'=FilterByCloseEntity(mv, me_{-1})$, that is, the properties in mv filtered by the previous entities me_{-1}; and

- the set $mv_{-1}'= FilterByCloseEntity(mv_{-1}, me)$, that is, the properties in vp_{-1} filtered by the current entities me.

```
10. Update the buckets in B_f with the information that the elements in me, mp' and mv'
    match the keyword kₘ, and mp₋₁', mv₋₁' match the keyword kₘ₋₁.
11. Update me₋₁, mp₋₁, kₘ₋₁, f₋₁ and mv₋₁ with me, mp, mv, kₘ and f₋₁, respectively.
12. Return Bₒ, B_f
```

If we analyze the bucket set B_f that Algorithm 2 outputs and the definition of answer for *RDF* (*relational*) environment, we can say that: $B_E(B_f)$ represents the *metadata matches* with classes (relation schemes) or A_{CM} (A_{SM}); $B_P(B_f)$ represents the *metadata matches* with the properties (attributes) or A_{PM} (A_{AM}); and $B_v(B_f)$ represents the *data matches* or A_{DM} (A_{TM}). The Build Nucleuses and Select Nucleuses algorithms have as goal to reduce the sets B_E, B_P, B_v to obtain minimal answer.

Example: Build Buckets algorithm.

Consider the schema of Section 4.1.1 and let $K=\{$ *"Country"*, (*"Population"*, $f)\}$, where f is a function that returns *True* iff a literal is greater than 10,000 and datatype(f)=NUMBER.

The matches with the keyword *"Country"* are $me_1=\{e_1\}$ $mp_1=\{\}$ and $mv_1=\{\}$

The matches with the keyword *"Population"* are $me_2=\{e_1\}$ $mp_2=\{p_1, p_2\}$ and $mv_2=\{p_2\}$

Then, $B_o=\{b_1, b_2, b_3\}$ and $B_f=\{b_1, b_2\}$, where $b_1= (e_1, \{$*"Country"*$\})$, $b_2=(p_1, \{$*"Population"*$\}, f)$ and $b_3= (p_2, \{$*" Population"*$\}, f)$. Note that the set B_f does not

include the bucket b_3, since line 9 excludes the property p_2 of mp_2, because the domain of p_2 does not belong to any of the entities in me_1.

4.2.3. Build Nucleuses

This section outlines the *Build Nucleuses* algorithm, which is the second step of the translation algorithm, and the first part of the *nucleuses heuristic*. This algorithm builds the nucleus set derived from a bucket set. The concept of nucleus, as the concept of bucket, are also related to the assumption *U2* (see Section 3.3.1). A nucleus groups the buckets that are related to the same entity. A nucleus expresses the set of keywords that a resource of a class (RDF) or a tuple (relational) would cover.

Given a bucket b, we define the function *related_entity* as:

$$related\_entity(b) = \begin{cases} entity(b) & \text{if } b \text{ is an entity bucket} \\ domain(property(b)) & otherwise \end{cases}$$

Algorithm 3 details the Build Nucleuses algorithm. The input of the algorithm is a bucket set B; each bucket b in B is assigned to a nucleus according to the *related_entity(b)*. The output is the nucleus set N.

Algorithm 3 Build Nucleuses
Input: B
Output: nucleus set N

1. $N \leftarrow \{\}$
2. **for each** bucket b_e **in** $B_E(B)$
3.   *Create a new nucleus with the bucket b_e and*
       add the nucleus to N
4. **for each** bucket b_p **in** $B_p(B)$
5.   if exists a nucleus in N such that
       $entity(n)=related\_entity(b_p)$
       then add b_p to N,
       else create a new nucleus with b_p and add n to N.
6. **for each** b_v in $B_v(B)$
7.   if exists a nucleus in N such that
       $entity(n)=related\_entity(b_v)$
       then add b_v to n,
       else create a new nucleus n with b_p and add n to N.
8. **return** N

Example: Build Nucleuses Algorithm.

Consider again the schema of Section 4.1.1 and let $B_o = \{b_1, b_2, b_3\}$, where

- $b_1 = (e_1, \{\text{"Country"}\})$
- $b_2 = (p_1, \{\text{"Population"}\}, \emptyset)$
- $b_3 = (p_2, \{\text{" Population"}\}, \emptyset)$

Then, $N = \{n_1, n_2\}$, where $n_1 = (b_1, \{b_2\}, \{\})$ and $n_2 = ((e_2, \{\}), \{b_3\}, \{\})$.

The example reveals that resources of the entity e_1, represented by n_1, cover the keywords *"Country"* and *"Population"*. However, resources of the entity e_2 only cover the keyword *"Population"*.

4.2.4. Select Nucleuses

This section outlines the Select Nucleuses algorithm, the third step of the translation algorithm, and the second and last part of the *nucleuses heuristic*. Given a nucleus set, the algorithm returns another nucleus set, built from the original, that contains the best nucleuses. Based in Section 3.3, the *best nucleus set* is the smallest nucleus set that covers the largest set of keywords, and that has the best similarity between the keywords and the elements of the nucleuses. However, the computation of the best nucleus set is an NP-complete problem (by a reduction to the bin packing problem). Then, the *nucleuses heuristic* tries to generate an approximate solution to the problem. The main elements of this heuristic are: (i) the best nucleus algorithm; (ii) the greedy algorithm that, given a *KwQ+* query *K* and a schema *S*, builds the *best nucleus set*, using the best nucleus algorithm.

To compute the best nucleus, the algorithm uses the function *score*. Given a schema *S*, a *KwQ+* query *K* and a nucleus (or bucket), the function *score* expresses, quantitatively, the relevance of the nucleus (or bucket) for the query. The *score* function depends on the environment and is detailed in Section 5.5.2. Note that we can generate different heuristics for finding the best nucleus, using different *score* functions.

4.2.4.1. Best Nucleus Algorithm

Given a nucleus set, which is the *best nucleus*? The answer of this question depends on the user intention that, as already discussed, is ambiguous. Then, we propose a heuristic that tries to guess the user intention.

Obvious ideas about the best nucleus explore the number of keywords that the nucleus cover, the similarity of the metadata matches, or the values matches with the keywords, etc. These ideas are grouped, quantitatively, in the function *score*.

What is not so obvious is the idea that the best nucleus is not necessarily a nucleus in *N*, but it would be a nucleus built from one of the nucleuses in *N.* Following the idea that users prefer minimal answers, we assume that two buckets in a nucleus should not cover the same keywords, because we can remove one of the buckets without affecting the number of keywords that are covered by the nucleuses, that is equivalent to reduce the answer. The following examples analyze the importance of reducing the nucleuses.

Example. Reduce nucleuses.

Given the query $K=\{$*"Country*, *("Population", f)*$\}$, where

- *f* is a function that returns *True* iff a literal is greater than one million
- $e_1=(0.9, \{$*"Country"*$\})$
- $p_1=(0.5, e_1, \{$*"Population"*$\}, \{xsd:numeric\})$
- $p_2=(0.45, e_1, \{$*"Population Density"*$\}, \{xsd:numeric\})$
- $n = (b_1, \{ b_2, b_3\},\{\})$, where

  ○ $b_1= (e_1, \{$*"Country"*$\})$

  ○ $b_2=(p_1, \{$*"Population"*$\}, \emptyset)$

  ○ $b_3= (p_2, \{$*"Population"*$\}, \emptyset)$.

The nucleus *n* indicates to retrieve the countries that have population and population density greater than one million, that is, no country. However, if we reduce *n* to $n'= (b_1, \{ b_2 \},\{\})$, we build a nucleus that covers the same keyword set and has several answers, since the nucleus *n'* indicates to retrieve the countries with population greater than one million (probably the real user intention).

As another example, given $K = \{$*"City"*$\}$, with

- $e_1=(0.8,\{$*"City"*$\})$

- $p_1=(0.8, e_1, \{"Name"\}, \{xsd:string\})$

- the nucleus $n=\{b_1,\{\},\{b_2\}\}$, where $b_1=(e_1, \{" City"\})$ and $b_2=(p_1, \{"City"\})$

     The nucleus n indicates to retrieve the cities with the word *"City"* in the name. A nucleus only with the bucket b_1 indicates to retrieve all cities in the database. Again, the second case would probably be the real user intention.

     Algorithm 4 outlines the *best nucleus* algorithm. It has as input the set of nucleuses *N*, the function *score* and the keyword set *K*. It has as output the best nucleus n_s. If two nucleuses have the same *score,* we select as the best nucleus that with the entity with higher ranking.

     Given a nucleus set, the first step of the *best nucleus* algorithm is to reduce each nucleus by keeping buckets that do not share keywords. Section 4.2.4.2 handles the problem of reducing a nucleus. Then, the algorithm finds and returns the nucleus with the greatest *score*.

Algorithm 4 Best Nucleus
Input: *N, score, K*
Output: n_s

1. Create the reduced nucleus set N_r from *N* using Algorithm 5, *score* and *K*
2. Find the nucleus n_s in N_r with the greatest *score* for the query *K*
3. return n_s

4.2.4.2. Reduce Nucleus

In the previous section, we discussed why we need to reduce a nucleus when it has at least two buckets sharing a keyword. Ideally, the bucket set that will remain in the nucleus is the smallest bucket subset that covers the same keywords that the original set and that better covers the keywords, based on some *score*. Again, the reduce nucleus problem is an NP-complete problem, and again we address the problem using a greedy algorithm to find an approximate solution. The greedy algorithm prioritizes buckets with the greatest *score*.

     Algorithm 5 details the greedy method that reduces a nucleus. It has as input the nucleus *n*, the function *score*, and the keyword set *K*. It has as output the reduced nucleus *n'*, which is a nucleus derived from *n* and is such that both nucleuses are related to the same entity and cover the same keywords. The algorithm starts by

creating the nucleus *n'* with the entity bucket of *n*. Then, it iteratively: (i) drops from *n* the keyword set that *n'* covers; (ii) finds the property bucket or value bucket *b* in *n* with the greatest *score*; and (iii) adds *b* to *n'*. The iterations stop when there are no buckets in *n*. If two buckets have the same *score,* we select as the best bucket that with the property with higher ranking.

Disregarding a keyword set K_c from *n* implies to remove from the buckets in *n* the keywords in K_c. After that, the disregard operation removes from *n* the buckets *b* that do not cover keywords, that is, such that *keywords*(*b*)=∅. Note that the buckets remaining in *n* in each iteration cover at least a keyword not covered by any of the buckets previously selected.

Algorithm 5 Reduce Nucleus
Input: *n*, *K*, *score*
Output: *n´*

1. Initialize *n´* with the entity bucket in *n*
2. **while** exists bucket in *n*
3.   Disregard from *n* the keywords *n´* covers
4.   Find the bucket *b* in *n* with the greatest *score* for the query *K*
9.   Add *b* to *n´*
10. **return** *n´*

4.2.4.3. Greedy Algorithm

The greedy algorithm generates a set N_s of nucleuses such that:

(1) N_s covers a large subset of keywords in *K*.

(2) All entities in N_s are in the same connected component of the graph induced by the schema.

Algorithm 6 corresponds to a pseudo-code of the Select Nucleuses algorithm; it has as input the *KwQ+* query *K*, the nucleus set *N*, the shortest-path distance values *δ* and the function *score*; and it has as output the set of nucleuses N_s.

First, it initializes the set N_s as the empty set. Next, it finds the best nucleus in N_s, using Algorithm 4. Then, using *δ*, it removes from *N* those nucleuses that not maintain connectivity with the *best nucleus*; that is $n \in N$ such that *δ*(*entity*(*n*),*entity*(n_s))=∞, where n_s is the best nucleus.

Iteratively, Algorithm 6 continues by: (i) adding the best nucleus to *N*; (ii) disregarding from *N* the keyword set that the best nucleus covers; (iii) finding the

best nucleus among the nucleus remaining in N. The iterations stop when there is no nucleus in N.

Disregard a keyword set K_c from N implies to remove from all the buckets in the nucleuses in N the keywords in K_c. After that, the disregard operation removes from N the nucleuses n that do not cover keywords, that is, such that *keywords*$(n)=\emptyset$. Note that the nucleuses remaining in N in each iteration cover at least one keyword not covered by any of the nucleuses previously selected.

Algorithm 6 Select Nucleuses
Input: *K*, *N*, *δ*, *score*
Output: N_s

1. Initialize N_s as empty set
2. Find the best nucleus, using Algorithm 4 with *N*, *score,* and *K*
3. Preserve the connectivity in N_s, that is, remove from *N* the nucleus *n* such that
    δ(*entity*(*n*), *entity*(n_s))=∞, where n_s is the best nucleus
4. **while** there is a best nucleus
5.   Add the best nucleus to N_s
6.   Disregard from *N* the keywords that the best nucleus cover
7.   Find the best nucleus, using Algorithm 4 with *N*,
        score and *K*
8. **return** N_s

4.2.5. Connect Entities

This section outlines the Connect Entities algorithm, which is the fourth step of the translation algorithm, and which corresponds to the *connection heuristic*. As we remarked in Section 3.3, the user prefers to observe resources that are interrelated and prefers to observe as few resources as possible. In the previous section, we introduced a heuristic to minimize the nucleus set, but we still need to interrelate the nucleuses and keep the number of resources to the minimum.

The schema describes, through the joins, the valid ways to interrelate the nucleuses. Two nucleuses n_1 and n_2 are *related* in a schema S iff there exists a path between *entity(n_1)* and *entity(n_2)* in the graph induced by S. To handle both problems, interrelate and minimize, we need to find the shortest join set that connects the entities set induced by the nucleus, which is equivalent to finding the minimum Steiner tree for the entity set in the graph induced by the schema S.

However, the minimal Steiner tree problem is NP-Complete. There is a heuristic proposed in Chopra & Rao (1994), called *SteinerH*, which produces a Steiner tree whose weight is within a $2 - 2/t$ factor of the weight of the minimal Steiner tree.

Given a graph *G=(V, E)* for the set $V´\subseteq V$, the *SteinerH* algorithm computes the minimal Steiner tree approximation in four main steps. The first step builds the *metric closure G´* of *V´* in *G*. The metric closure for a set *V´* in a graph *G* is the complete graph *G´=(V´,D)*, where each edge in $(e_1, e_2) \in D$ is weighted by the shortest path value $\delta(e_1, e_2)$ in *G*. The second step is to compute the minimum spanning tree *T* of *G´*. The third step is to generate the set $E´\subseteq E$ such that, for each edge (v_1, v_2) in *T*, computing the shortest path *p* between each pair of vertices, v_1 and v_2, in *G*, and then adding the edges of *p* to *E´*. The last step is to create the Steiner tree approximation *S=(V´,E´)*. Figure 11 and Figure 12 show examples of how the *SteinerH* works. In the example of Figure 11, *SteinerH* finds a minimal Steiner tree, but in the example of Figure 12, the heuristic fails. *SteinerH* runs in polynomial time; finding the shortest path and the shortest path length can be solved in polynomial time using Floyd-Warshall; building the metric closure is also polynomial, and finding the minimum spanning tree of a graph is polynomial, using Prim´s algorithm.
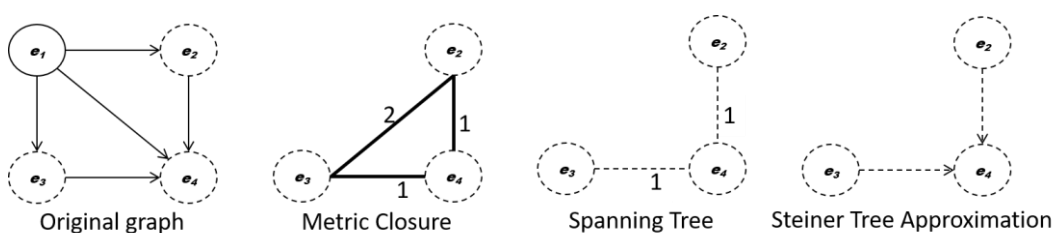
Figure 11. Example of the heuristic to find a minimum Steiner tree.

SteinerH solves the problem of finding a minimal Steiner tree approximation efficiently. Returning to our original problem, to interrelate the nucleus in a minimal way, *SteinerH* seems to solve the problem. It is easy to perceive that, given a graph, multiple minimal Steiner tree may exist, as illustrated in Figure 13, but the result returned by *SteinerH* may not satisfy the user. The concept of the best way for the user, according to the assumptions of Section 3.3.1, is basically the minimal way. All minimal Steiner trees for an entity set are good candidates for the user. The

number of minimal Steiner trees for a graph would be exponential, since the number is bounded by the number of spanning-trees of a complete graph of *n* vertices, which is n^{n-2} (Smith, 2015). Considering that a single minimal Steiner tree seems limited and all minimal Steiner trees would be computationally expensive, we propose a modification to the *SteinerH* algorithm*, called *MultipleSteinerH* in what follows*, to find at most *m* Steiner trees satisfying the approximation factor of *SteinerH*.
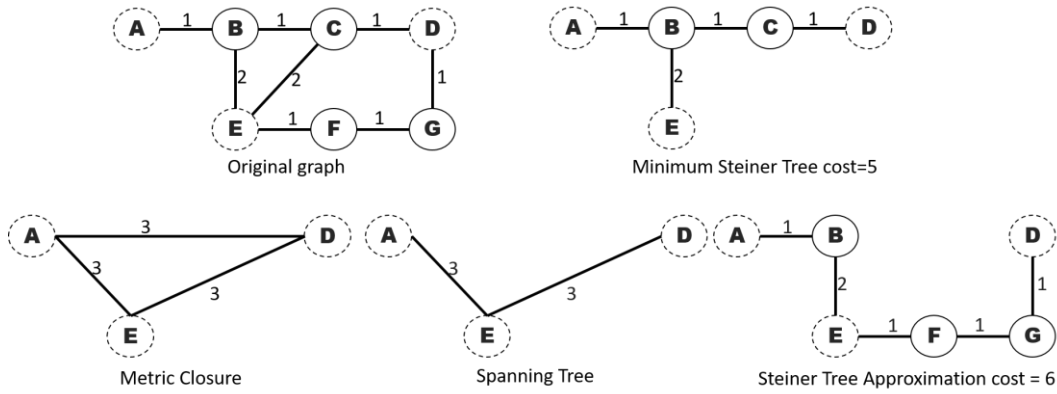
Figure 12. Example of a wrong result of the heuristic.
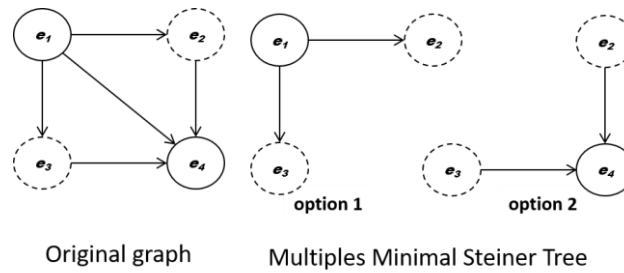
Figure 13. Example of multiple minimal Steiner Trees.

There are two points in *SteinerH* that should be changed to generate several Steiner trees, preserving the conditions that validate the approximation factor. The first one is step 2 of the algorithm that generates the minimum spanning tree (MST). If there exists another *MST*, then we may generate different results, as *MST* 1 and *MST* 2 in Figure 14. The second one is step 3 that generates the Steiner tree from the shortest path between the nodes that have edges in the *MST*. If there is another shortest path between two nodes that has an edge in the *MST*, then we may generate different results too. The Steiner trees (a) and (b) in Figure 14 are generated from *MST* 1, because there are two shortest paths between nodes *E* and *D*.

Fundamentally, *MultipleSteinerH* only finds different *MST* and other shortest paths, and not only one, as in *SteinerH*.
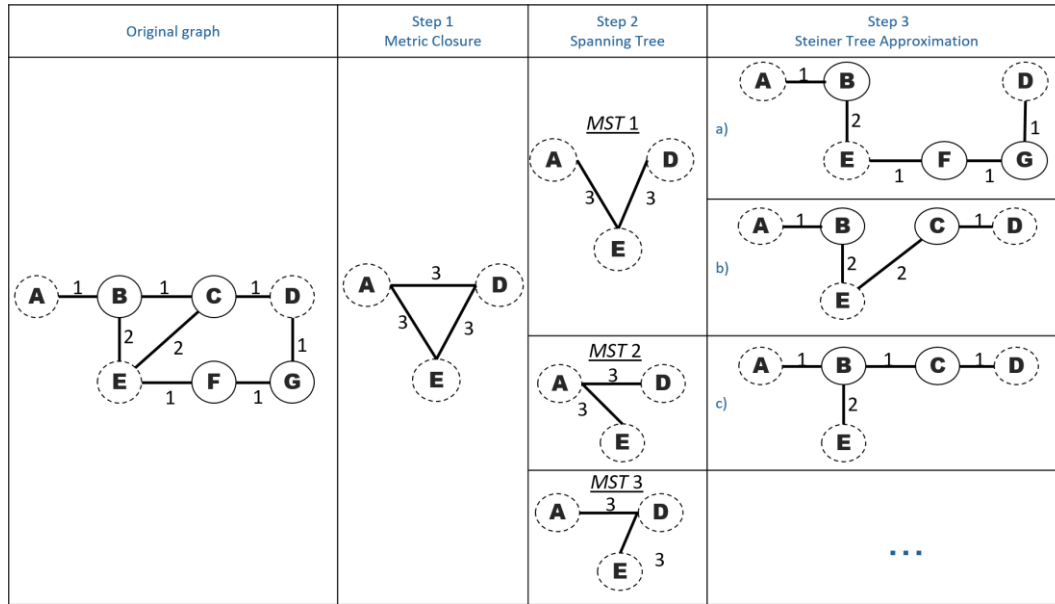
Figure 14. Example of multiple spanning trees and shortest paths.

Summarizing, the *MultipleSteinerH* heuristic starts as *SteinerH*, producing the metric closure of a graph. Then, it iteratively produces an *MST* (different from those produced in other iterations). From the *MST*, it produces different Steiner trees, when there are different shortest paths. The iterations continue until the number of Steiner trees produced exceeds *m*, or the number of iterations exceeds *m*, or there are no different *MSTs* to produce. Algorithm 7 details the *MultipleSteinerH* algorithm.

Algorithm 7 has as input an entity set *E*, and functions δ and π. Section 4.2.1 introduced functions δ and π, computed from the *FW-2* algorithm. Function π maps two entities, e_1 and e_2, into a set *P*, where *P* is the set that contains the pairs (e,j) such that exists a shortest path *p* between e_1 and e_2 where *e* is the successor entity of e_1 on the shortest path *p* based on join *j*. Note that, for $\pi(e_1,e_2)=P$, if $|P|=c$ then there are at least *c* different shortest paths between e_1 and e_2. Function δ maps two entities, e_1 and e_2, into an integer *n* such that *n* is the weight of the shortest path from entity e_1 to entity e_2.

Algorithm 7 MultipleSteinerH
Input: *E*, π, δ, *m*
Output: *R*

```
1. Compute the metric closure G´ using δ as a function of the graph that has as nodes
   the entities in E
2. Initialize the set Tp as empty; Tp will contain the MSTs produced in the iterations of
   the algorithm
3. Initialize i=0; i will count the number of iterations
4. while |Sp|<m (the number of Steiner trees produced does not exceed m) and i<m
   (the number of iterations does not exceed m)
5.    Compute the MST T from G´ such that T is different from any MST in Tp
6.    If T does not exist, return m, there is no different MST to produce
7.    Add T into Tp
8.    Compute the set Si with at most m-|Sp| Steiner trees from T using π, that is, using
      the different shortest paths between the entities that have an edge in T
9.    Add to R those trees such that R is not contained in Si
10. return R
```

If we analyze step 5 of Algorithm 7, we note that a classical *MST* algorithm does not exactly solve that problem. We also note that the classical path reconstruction algorithm, given π (or path reconstruction algorithm for Floyd-Wharsall algorithm) also does not solve the problem of step 8. Finally, step 9 has as a challenge how to compare tree sets efficiently.

To solve the problem of step 5, we propose an adaptation to *Prim´s* algorithm, that we refer to as *Prim-2*. Cormen, T. et al. (2009) details the original *Prim´s* algorithm. Given a graph $G=(V, J)$ and a tree set T_p, *Prim-2* builds the spanning tree T similarly to the original algorithm; the difference is that, when there are edges with the same weight in a cut. In this case, *Prim-2* tests if any of those edges does not belong to a tree in T_p. If there is an edge that satisfies the test, *Prim-2* adds that edge to the spanning tree T. If, in any iteration of *Prim-2*, any edge satisfies the test, *Prim-2* returns that T does not exist, otherwise, *Prim-2* returns T.

To handle the problem of step 9, we use a hash for a graph, where the nodes are entities, and the edges are joins. Algorithm 8 outlines the pseudo-code of how to calculate the hash. Basically, we propose the hash of a string as a hash of the graph. The string is the concatenation of: the string *"E:"*; the string formed by η of the entities in the graph in *S*, comma-separated and sorted in ascending order; string *"J:"*; and the string formed by η of the joins in the graph in *S*, comma-separated and sorted in ascending order.

```
Algorithm 8. Hash
Input: G=(E,J), S
Output: h
1. Initialize s as a string that contains "E:"
2. Sort the entities in E in increasing order by ɱ(S)(e), with e∈ E
3. Concatenate to s the string composed by the comma-separated ɱ(S)(e) values, with
     e∈ E
4. Concatenate to s the string "J:"
5. Sort the joins in J in increasing order by ɱ(S)(j), with j∈ J
6. Concatenate to s the string composed by the comma-separated ɱ(S)(j) values, with
     j∈ J
7. Set h as the hash of s
8. return h
```

For example, if we have an *MST* with the edges (e_1, e_2) and (e_2, e_3), where there are 2 shortest paths between e_1 and e_2 and 3 shortest paths between e_2 and e_3, the number of Steiner trees from the *MST* of the example will be 6. We can combine each shortest path that connects e_1 and e_2 with each shortest path that connects e_2 and e_3. In brief, the solution that we implemented to solve Step 8 of Algorithm 7 explores those combinations, that is, for each edge in *T*, calculate the shortest path set between the entities of the edge (using π) and combine the sets (cartesian product) to create different Steiner trees. In fact, Step 8 of Algorithm 7 requests producing a limited number of combinations (m-$|S_p|$); then, the cardinality of the shortest path set is also limited to produce no more combinations than requested. Algorithm 9 details the algorithm to solve Step 8 of Algorithm 7. It has as input the tree *T,* the number m and the function π, and has as output the tree set S_i, where the $|S_i|{\leq}m$.

```
Algorithm 9. Find Trees
Input: T=(V,E), π, m
Output: Sᵢ
1. Initialize Pₛ as an empty set; Pₛ will contain the shortest path set, for each edge in E
2. for each edge (e₁, e₂) in E
3.    Compute the set P, using π, where P has at most m shortest path set between
        entities e₁ and e₂
4.    Add P to Pₛ
5.    Update the bound m to produce the correct number of combinations, that is,
        m=m/|P|
```

```
6. Js is the cartesian product of Ps
7. Initialize set Si as an empty set
8. for each set J in Js do
9.    Create the tree S=(V,J)
10.   Add S to Si
11. return Si
```

Algorithm 10 details how to resolve Step 3 of Algorithm 9, that is, to find at most m shortest paths between two entities using function π. It has as output the set P, where $|P| \le m$, and each set $J_p \in P$ is a minimal join set that connects the entities.

Algorithm 10 is a recursive algorithm that iterates for the pairs (e, j) in $\pi(e_1, e_2)$, and recursively finds P' that contains at most m shortest paths between (e, e_2). Each iteration of the algorithm finds $|P'|$ shortest paths; then, in each iteration, the value of m is decreased by $|P'|$.

```
Algorithm 10. Shortest Paths
Input: e1, e2, π, m
Output: P
1. Initialize P as empty set
2. if e1 = e2 or π(e1, e2) is empty return P
3. Iterate for the elements (e,j) ∈ π(e1, e2) in increasing order of the ranking of e
4.    Compute the set P´ by recursively calling Shortest Paths using π with at most m
      shortest paths between the entities
      e and e2
5.    Decrease m with the value of |P´|, that is m=m-|P´|
6.    Create the set Pc, that is, the Cartesian product between the set {j} and P´
7.    Add to P the set Pc
8. return P
```

Note that for $m=1$ the outputs of *MultipleSteinerH* and *SteinerH* are equivalent.

4.2.6. Translation Algorithm

Algorithm 11 outlines the entire translation process, whose goal is to find minimal answers for a *KwQ+* query. The algorithm addresses this problem and finds solutions that are probably minimal, because we use heuristics, instead of an exact method. Sections from 4.2.1 to 4.2.5 detail the heuristics. Algorithm 11 has as input

a *KwQ+ K* query, the schema *S*, functions δ, π, f_{ME}, f_{MP}, f_{MV}, *score* and *query*; and the numbers μ and *m*.

Recall that Section 4.2.1 defines functions δ and π, and explains how to compute them; Section 4.2.2 defines functions f_{ME}, f_{MP} and f_{MV}, whose computation depends on the environment and μ; and Section 4.2.4 defines the function *score*, which also depends on the environment.

Function *query* maps an abstract query to answers. Given an abstract query *Q*, function *query* compiles a structured query in a specific environment query language, executes the query in the environment, and returns as answers the result of the execution. Obviously, how to compile and execute the query is a process that depends on the environment, as detailed in Section 5.6.

Algorithm 11 executes in sequence steps from 1 to 4, which are Algorithm 2, Algorithm 3, Algorithm 6 and Algorithm 7, respectively. Step 5 of the translation algorithm is to create, for each tree *T=(V,E)* output by Algorithm 7, an abstract query *Q=(N,E)*, where *N* is the nucleus set output of Algorithm 6. Finally, it finds answers from the abstract queries created using function *query*.

The main output of the Algorithm 11 is the answer set *A,* but the algorithm also returns the bucket set B_o, output by Algorithm 2, and Q_A, the abstract query set created in step 5. When the set *A* does not satisfy the user expectations, we use B_o and Q_A, as we explain in Section 4.3, to find more answers.

Algorithm 11. Translate Algorithm

Input: *K, S, δ, π, f_{ME}, f_{MP}, f_{MV}, score, query, μ, m*

Output: A*, Q_A, B_o*

1. Build the buckets set B_o, B_f from *K* and *S* by running Algorithm 2 with input *K, S, f_{ME}, f_{MP}, f_{MV}, μ*

2. Build the nucleus set *N* from B_f by running Algorithm 3 with input B_f

3. Select the best nucleus N_s from *N* by running Algorithm 6 with input *K, N, δ, score*

4. Create the tree set *R* from the entities in N_s by running Algorithm 7 with input entities(N_s), π, δ and *m*

5. Create the set Q_A by adding the abstract query *Q=(N_s,J)* to Q_A,

   for each tree *T=(V,J)* in *R*

6. Create the set *A*, initially empty

7. Iterate for each abstract query A_q in Q_A computing A=*query(A_q,S)* until |A|>0

8. **return** *A, B_o, Q_A*

As we use heuristics, we cannot claim that, given a *KwQ+* query *K*, the output *A* of Algorithm 11 contains minimal answers for *K*. But we may expect that Algorithm 11 finds minimal answer in most of the cases. The general idea to prove this claim is that the answers *A* are derived from an abstract query $Q=(N_s,J)$ such that: the construction *Q* leads to a connected graph (Step 3 and 4); by construction, N_s is an approximation of the minimal bucket set that covers as many keywords as possible (Step 1, 2 and 3); and, by construction, *J* is an approximation of the minimal join set that connects the nucleuses in N_s.

4.3 User Feedback

In this section, we discuss what to do when the answers in *A*, that Algorithm 11 returns, do not meet the user intention. Most of the papers summarized in Section 2.3 propose to use a backtracking algorithm to generate alternative queries and alternative results. Backtracking strategies can be computationally exhaustive and confusing, and do not guarantee success. Then, we propose a mechanism for the user to give feedbacks that enable building a set of answers *A´* with better success guarantees.

We can raise two reasons for a user not to be satisfied with an answer *A*: (i) the user expected the resources that appear in the answers in *A* but interrelated in different ways; or (ii) the resources in *A* are not what the user expected. Section 4.3.1 and 4.3.2 explain the feedback that users can give for the first and second reasons, respectively.

4.3.1. Computing Alternatives to Interrelate the Resources

Suppose that the user expected the resources that appear in the answers in *A* but interrelated in different ways. Or, in other words, the user agrees with the nucleuses that Algorithm 11 selected, but not with the joins. As Algorithm 11 has as output the set Q_A with different forms to interrelate the nucleuses selected, the user can select from Q_A the expected joins and force the generation of an alternative query *Q* and, hence, a different answer set *A´=query(Q)*. If none of the queries *Q* in Q_A satisfies the user, we compute more ways to interrelate the joins, using Algorithm 7.

4.3.2. Computing Alternative Resources

Suppose that the resources in *A* are not what the user expected. Or, in other words, the user does not agree with the nucleuses. Algorithm 11 has as output the set B_o with all resources that have matches with keywords. As feedback, the user can select a subset $B' \subseteq B_o$. From *B*, we can execute an algorithm similar to Algorithm 11, but guaranteeing that the buckets in *B* belong to nucleuses selected.

Algorithm 12. Feedback Algorithm
Input: *B, K, S, δ, π, f_{ME}, f_{MP}, f_{MV}, score,query,μ,m*
Output: A*, Q_A, B_o*

1. Remove *keywords(B)* from *K*
2. Build the buckets set B_o, B_f from *K* and *S* by running Algorithm 2 with input *K, S, f_{ME}, f_{MP}, f_{MV}, μ*
3. Build the nucleus set *N* from B_f by running Algorithm 3 with input B_f
4. Select the best nucleus N_s from *N* by running Algorithm 6 with input *K, N, δ, score*
5. Adds the buckets in *B* to N_s
6. Create the tree set *R* from the entities in N_s by running Algorithm 7 with input entities(N_s), *π, δ* and *m*
9. Create the set Q_A by adding the abstract query *Q=(N_s,J)* to Q_A,
7.   for each tree *T=(V,J)* in *R*
8. Create the set *A* empty
9. Iterate for each abstract query A_q in Q_A computing A=*query(A_q,S)* until |*A*|>0
10. **Return** *A, B_o, Q_A*

4.4 Chapter Conclusion

In this chapter, we presented a solution for the Keyword Search problem over graphs databases with schema. As we explained, the problem is NP-complete. Hence, we introduced heuristics to find approximate solutions efficiently. The first heuristic considers the order of appearance of the keywords, instead of treating the keyword-based query as a "bag of words", to select the relevant matches. To select the relevant pieces of information, the strategy groups the matches at the level of entities and properties, in buckets. Then, it groups the buckets, based on the entities, in nucleuses. Using greedy heuristics, the translation algorithm selects the best set of nucleuses that cover the keywords, are in the same connected component in the graph induced by the schema and contain the minimal set of elements of the schema.

We also proposed a heuristic that, given the entities derived from the nucleus set selected by the algorithm and the joins, finds at most *m* minimal Steiner trees. The heuristic uses a variation of Floyd-Warshall algorithm to find several shortest-paths between two nodes. Finally, the translation algorithm generates abstract queries, using the nucleus set and the Steiner Trees. The abstract query is then compiled into a structured query and executed in the specific database to generate answers. Since the translation algorithm finds answers based on heuristics, we also proposed feedback strategies to generate new answers, if the algorithm fails.

5
Implementation

This chapter presents the technical aspects of DANKE, a keyword search tool and framework that implements the strategy proposed in Chapter 4. Section 5.1 illustrates the process of parsing a user text into a *KwQ+* query. Section 5.2 summarizes the components and the architecture of the framework. Section 5.3 introduces the use of auxiliary tables to help finding the matches and building the schema. Section 5.4 shows an algorithm to map the database schema into an abstract schema. Section 5.5 details the function to find keyword matches and calculate the *score* of a bucket or nucleus. Section 5.6 describes how to map an abstract query into an SQL query or SPARQL query. Finally, Section 5.7 presents the interface.

5.1 User Query Parser

A user can specify the query through a text that follows a particular grammar. The grammar is an *LL(*)* grammar (Parr, 2013), built using ANTLR[1]. Annex 9.1 details the grammar.

The Boolean functions are fragments of texts that follow some specific rules imposed by the grammar, where each rule defines a function. Table 4 shows some examples of parsing text into Boolean functions.

Table 4. Examples of parsing text to Boolean functions.

Text	Filter
greater than 100	Function that returns True if a literal is a number greater than 100
< 100	Function that returns True if a literal is a number less than 100

---

[1] https://www.antlr.org/

equal 18/07/2019	Function that returns True if a literal is a date equal to *July 18, 2019*
> 1 and <5	Function that returns True if a literal is a number between 1and 5
equal car or train	Function that returns True if a literal is a string equal to the string *"car"* or the string *"train"*

The Parser algorithm translates a user text into a *KwQ+* query. The fragments of text that are not recognized as Boolean function are considered keywords. A keyword can be a word or phrase; a phrase is a word set between quotes. With the *ANTLR* tool, we generated the grammar parser. The Parser first classifies the original text as a sequence of keywords and Boolean functions, using the grammar. Given the order of the elements in the sequence, it builds the *KwQ+* query. Table 5 presents some examples about how the Parser algorithm works.

Table 5. Examples of parsing text into a *KwQ+* query.

Text	Sequence	*KwQ+*
panama city	"Panama" is a keyword; "City" is a keyword	{"panama"; "city"}
"panama city"	"panama city" is a keyword	{"panama city"}
name equal Colón	"name" is a keyword; "equal Colón" is the Boolean function *f*, where *f* returns True if a literal is a string equal to the string *"Colón"*	{("name", *f*)}
population > 1 and <5	"population" is a keyword; > 1 and <5 is the Boolean function *f*, where *f* returns True if a literal is a number between 1 and 5	{("population", *f*)}
country population >10000	"country" is a keyword "population" is a keyword >10000 is the Boolean function *f*, where *f* returns True if a literal is a number greater than 10000	{"country"; ("population", *f*)}

5.2 Architecture

Figure 15 summarizes the component diagram of DANKE, whose main components are Pre-processing, Database, Functions, Translate Algorithm, FeedBack Algorithm and Query parser. The Pre-processing component executes the algorithms that map the database schema into an abstract schema (Section 5.4) and compute the Shortest Path Index (Section 4.2.1). The Database component executes structured queries over the database. The Functions component contains the functions that are required by the Translation Algorithm (Sections 5.5 and 5.6). The Translation Algorithm component finds answer for a *KwQ+* query (Section 4.2.6). The Feedback Algorithm component has two possibilities: the first one is a new abstract query, and the second one is a bucket set (Section 4.3 explains both cases). The Query Parser component parses a text into a *KwQ+* query using a specific grammar (Section 5.1).
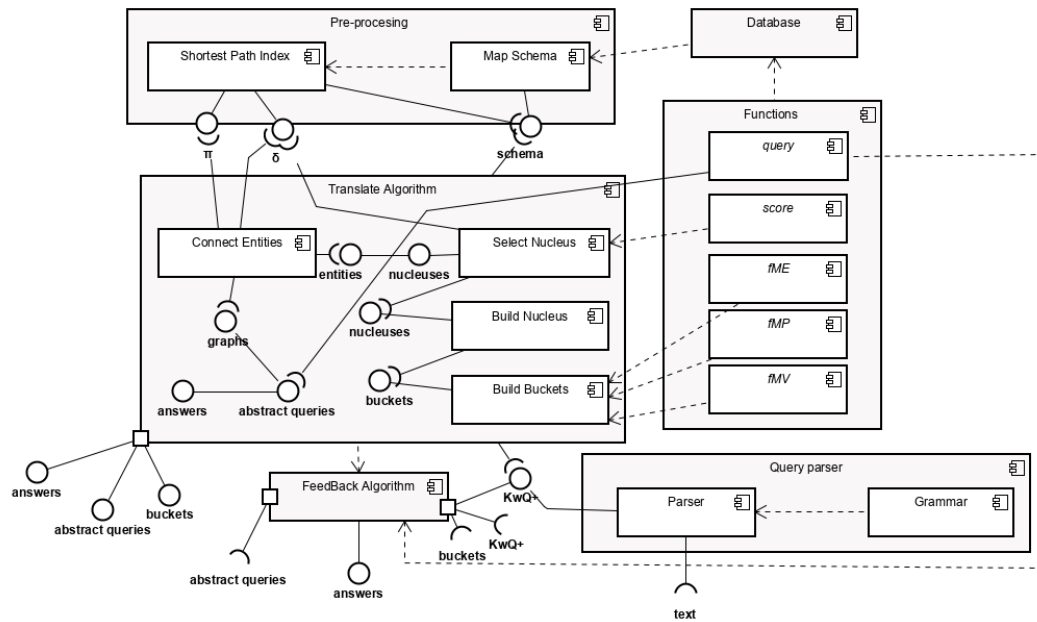
Figure 15. DANKE Component Diagram.

The execution flow of DANKE goes as follows:

1. The pre-processing algorithms are executed.
2. The user submits a text or a feedback.
3. If the submit is a text:
   a. The Query Parser parses the text into a *KwQ+* query.

b. Given the functions, the pre-processing results and the *KwQ+* query produced by the Query Parser, the Translation Algorithm finds answers, abstract queries and buckets for the *KwQ+* query.

4. If the user submitted a feedback, the Feedback algorithm finds new answers from the feedback:

a. If the feedback is an abstract query, the *query* function is used to find new answers.

b. If the feedback is a bucket set, a version of the translation algorithm is used to find new answers.

Figure 16 outlines the architecture of DANKE search tool. Danke search tool has an implementation of functions f_{ME}, f_{MP}, f_{MV}, *score*, *query* and the function that maps the database schema into an abstract schema, for each environment. The next sections present the implementation of these functions for each environment.
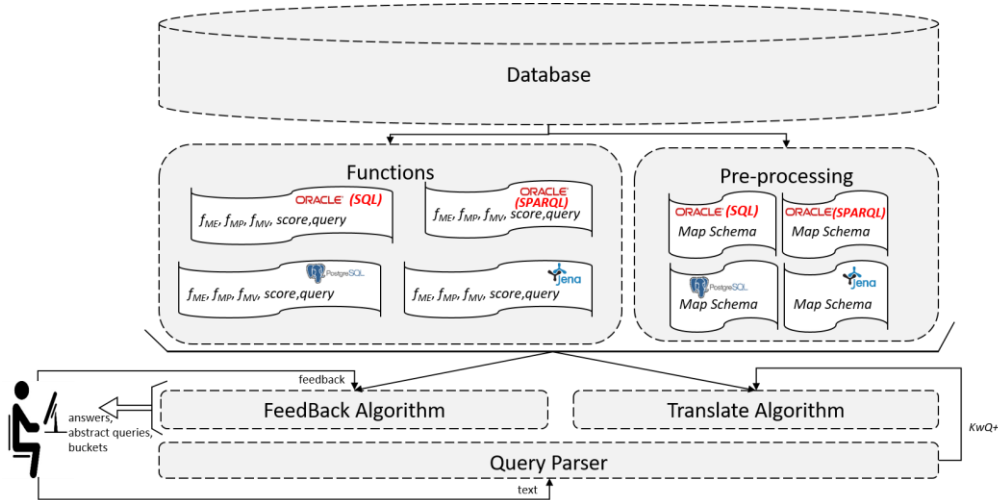
Figure 16. DANKE Architecture.

5.3 Auxiliary Tables

For efficiency purposes, we build auxiliary tables, with metadata and data from the database. The auxiliary tables help finding the matches and building the schema faster; these tables are computed only once. Section 5.3.1 detail how to populate the auxiliary tables for the relational environment and the RDF environment respectively.

For Oracle SQL, Postgres, and Oracle RDF, we compute all tables, but for the Jena TDB, we do not compute the VALUES table. For Oracle SQL, Postgres and Oracle RDF, we materialize the auxiliary tables in the database only once. For the Jena TDB, the auxiliary tables are built, in memory, every time the tool is initialized.

The auxiliary tables are: *ENTITIES* table, *PROPERTIES* table, *JOINS* table, and *VALUES* table. The *ENTITIES* table has three columns: *name*, *labels*, and *ranking*. The *PROPERTIES* table has five columns: *name, domain*, *labels*, *ranking*, and *datatypes*. The *JOINS* table has six columns: *name*, *domain_entity, domain_properties*, *range_entity*, *range_properties,* and *ranking*. The *VALUES* table has tree columns: *value, property,* and *domain_entity*.

5.3.1. Populating the Auxiliary Tables in the Relational Environment

The *ENTITIES* table contains data about all the tables in the database. The *name* column is filled with the table names; and the *labels* and *ranking* columns are filled with values assigned by someone with context knowledge.

The *PROPERTIES* table contains data about the properties in the database. The *name* column is filled with the attributes' names, the domain is filled with the table name of the table to which the attribute belongs, the *labels* and *ranking* columns are filled with values assigned by someone with context knowledge, and the *datatypes* column is filled with the *DATA_TYPE* of the attribute.

The *JOINS* table contains the data about the foreign keys in the database. For each foreign key definition

```
[CONSTRAINT [fk_name]] FOREIGN KEY [tbl_name] (col_name₁,...,col_nameₖ)
REFERENCES rtbl_name (rcol_name₁,...,col_nameₖ)
```

the *JOINS* table is filled with *name*, *domain_entity, domain_properties*, *range_entity* and *range_properties* equal to *fk_name*, rtbl_name, *"rcol_name₁,...,col_nameₖ"*, tbl_name and *"col_name₁, ...,col_nameₖ"*, respectively. The *ranking* column is filled with values assigned by someone with context knowledge.

The *VALUES* table contains the values of each attribute, with datatype VARCHAR, in the database. The table is filled with the different values of each

attribute, the *property* contains the attribute name, and *domain_entity* column contains the table name of the table to which the attribute belongs.

Example: Auxiliary Tables in the Relational Environment.

Consider a relational database with the metadata shown in Figure 7 and the data in Table 6 and Table 7. Table 8, Table 9, Table 10 and Table 11 show examples of the tables *ENTITIES, PROPERTIES*, *JOINS* and *VALUES*, respectively. Note that, if we apply the algorithm to create the schema described in Section 5.4 to the auxiliary tables, the result is the schema *S* for relational databases of Section 2.3.1.

Table 6. Country Table Data.

name	population
Panama	4 162 618
Vatican City	1 000

Table 7. City Table Data.

name	population	of_country
Panama City	880 691	Panama
Colon	253 366	Panama

Table 8. Example of the *ENTITIES* Table for the Relational Environment.

name	labels	ranking
country	Country	0.9
city	City	0.8

Table 9. Example of the *PROPERTIES* Table for the Relational Environment.

name	domain	labels	ranking	datatypes
name	country	Name	0.9	VARCHAR
population	country	Population	0.9	NUMBER
name	city	Name	0.8	VARCHAR
population	city	Population	0.8	NUMBER

Table 10. Example of the *JOINS* Table for the Relational Environment.

name	domain_entity	domain_properties	range_entity	range_properties	ranking
city_country	country	name	city	of_country	0.9

Table 11 Example of the *VALUES* Table for the Relational Environment

value	domain_entity	property
Panama	country	name
Vatican City	country	name
Panama City	city	name
Colon	city	name

5.3.2. Populating the Auxiliary Tables in the RDF Environment

The auxiliary tables are filled using SPARQL queries, where the *ENTITIES* table contains the classes, the *PROPERTIES* table contains the datatype properties, the *JOINS* table contains the object properties, and the *VALUES* Table contains the distinct literals of type STRING.

ENTITIES Table

```
INSERT into ENTITIES (name, labels, ranking)
  select ?class ?literal ?ranking
  where
  { ?class rdf:type rdfs:Class .
    ?class rdfs:label ?literal.
    ?class danke:ranking ?ranking.
  }
```

PROPERTIES Table

```
INSERT into PROPERTIES (name, domain, labels, datatype, ranking)
  select distinct ?property ?class ?literal ?datatype ?ranking
  where
  {
    ?property rdf:type owl:DatatypeProperty.
    ?property rdfs:label ?literal.
    ?property rdfs:domain ?class.
    ?property rdfs:range ?datatype.
    ?property danke:ranking ?datatype.
  }
```

JOINS Table

```
INSERT into JOINS (name, domain_entiy, range_entity, ranking)
  select distinct ?o_property ?d_class ?r_class ?ranking
  where
  {
    ?o_property rdf:type owl:ObjectProperty.
    ?o_property rdfs:domain ?d_class.
    ?o_property rdfs:range ? r_class.
    ?property danke:ranking ?datatype.
  }
```

VALUES Table

```
INSERT into VALUES (value, domain_entity, domain_property)
  select distinct ?literal ?property ?class
  where
  {
    ?property rdf:type owl:DatatypeProperty.
    ?property rdfs:domain ?class
    ?r ?property ?literal.
    filter( isLiteral(?literal) AND xsd:String(?literal))
  }
  group by ?class ?property ?literal
```

Example: Auxiliary Tables in the RDF Environment.

Consider an RDF dataset with the RDF schema shown in Figure 6and the triples shown in Figure 17. Table 12, Table 13, Table 14 and Table 15 show examples of the tables *ENTITIES, PROPERTIES*, *JOINS* and *VALUES*, respectively. Note that, if we apply the algorithm to create the schema of Section 5.4 to the auxiliary tables, the result is the schema *S* for RDF dataset of Section 2.3.1.

```
:panama rdf:type :country;
   :name "Panama";
   :population "4 162 618".
:vatican rdf:type :country;
   :name "Vatican City";
   :population "1 000".
:panama_city rdf:type :city;
   :name "Panama City";
   :population "880 691";
   :of_country :panama.
:colon rdf:type :city;
   :name "Colon";
   :population "253 366";
   :of_country :panama.
```

Figure 17. RDF Data.

Table 12. Example of the *ENTITIES* Table for the RDF Environment.

name	labels	ranking
:country	Country	0.9
:city	City	0.8

Table 13 Example of the *PROPERTIES* Table for the RDF Environment.

name	domain	labels	ranking	datatypes
:name	:country	Name	0.9	VARCHAR
:population	:country	Population	0.9	NUMBER
:name	:city	Name	0.8	VARCHAR
:population	:city	Population	0.8	NUMBER

Table 14. Example of the *JOINS* Table for the RDF Environment.

name	domain_entity	domain_properties	range_entity	range_properties	ranking
:of_country	:country	null	:city	null	0.9

Table 15. Example of the *VALUES* Table for the RDF Environment.

value	domain_entity	property

Panama	:country	:name
Vatican City	:country	:name
Panama City	:city	:name
Colon	:city	:name

5.4 Map Schema

The algorithm that maps a database schema into an abstract schema depends on the auxiliary tables. Algorithm 13 outlines that process. It has as input the auxiliary tables *ENTITIES*, *PROPERTIES* and *JOINS* and as output the abstract schema *S*.

The algorithm creates an entity for each tuple in the *ENTITIES* table, a property for each tuple in the *PROPERTIES* table and a join for each tuple in the *JOINS* table. The function m maps the elements of the abstract schema with the column *name* in the auxiliary tables.

Algorithm 13. Map Schema
Input: *ENTITIES, PROPERTIES, JOINS*
Output: S

1. Create the entity set *E* as empty
2. Create the property set P as empty
3. Create the join set *J* as empty
4. Create the map m as empty
5. **for each** tuple *t* in *ENTITIES*
6.   Create the entity e=(*ranking*(t), *labels*(t))
7.   Add *e* to *E*
8.   Add (e, *name*(t)) to m
9. **for each** tuple *t* in P
10.   Find the tuple (*e,s*) in m such that *s* is equal to *domain*(t)
11.   Create the property p=(*ranking(t),e*, *labels*(t),*datatypes(t)*)
12.   Add *p* to P
13.   Add (*p, n*) to m, where n is concat(*name*(t),'+', *domain*(t))
14. **for each** tuple *t* in *J*
15.   Find the tuple (e_d,s_d) in m such that s_d is equal to *entity_domain*(t)
16.   Find the tuple (e_r,s_r) in m such that s_r is equal to *entity_range*(t)
17.   Create the join j=(*ranking*(t), e_d, e_e)
18.   Add *j* to *J*
19.   Add (*j, name(t)*) to m
20. Create S as the tuple (*E, P, J*, m)
21. Return S

Give a property p and the abstract schema S, the *property*($\eta(S)(p)$) is defined as *split*($\eta(S)(p)$,'+').*first()*, and *domain*($\eta(S)(p)$) is defined as *split*($\eta(S)(p)$,'+').*last()*, where the *split* function splits a string into an array of strings using the specified separator, and the *first* and *last* functions return the first and the last elements of the array, respectively.

5.5 Matches and Score

We index the auxiliary tables to search the matches and to calculate the score faster. We use Oracle Text[2] to index the auxiliary tables for Oracle environments. We use pg_trgm[3] module to index the auxiliary tables for the Postgres environment. Finally we use Lucene[4] to create an index over the rdf:label and the *owl:DatatypeProperty* in Jena TDB environment.

5.5.1. Find Matches

This section outlines the implementations of the functions f_{ME}, f_{MP} and f_{MV}.

Given a keyword k, a number μ, and a schema S, f_{ME} produces an entity set *me*. The first step is to create the query f_{ME_query}, which depends on the environment, for k and μ. Then, f_{ME_query} is executed. Finally, f_{ME} fills the set *me* with all e such that $(e, s) \in \eta(S)$ and $s \in R$.

Given a keyword k, a number μ, and a schema S, f_{MP} produces a property set *mp*. The first step is to create the query f_{MP_query}, which depends on the environment, for k and μ. Then, f_{MP_query} is executed. Finally, f_{MP} fill the set *mp* with all p such that $(p, s) \in \eta(S)$ and $s \in R$.

Given a keyword k, a number μ, and a schema S, f_{MV} produces a property set *mv*. The first step is to create the query f_{MV_query}, which depends on the environment, for k and μ. Then, f_{MV_query} is executed. Finally , f_{MV} fills the set *mv* with all p such that $(p, s) \in \eta(S)$ and $s \in R$.

Oracle SQL and Oracle RDF

---

[2] https://docs.oracle.com/cd/B28359_01/text.111/b28303/quicktour.htm#g1011793
[3] https://www.postgresql.org/docs/9.6/pgtrgm.html
[4] https://jena.apache.org/documentation/query/text-query.html

As in Oracle, RDF databases and SQL databases coexist, the queries f_{ME_query}, f_{MP_query}, f_{MV_query} are the same.

The f_{ME_query} for a keyword k and μ is:

```
SELECT DISTINCT name FROM ENTITIES WHERE CONTAINS(labels, fuzzy(k), 1)>μ
```

The f_{MP_query} for a keyword k and μ is:

```
SELECT DISTINCT name||'+'||domain FROM PROPERTIES WHERE
CONTAINS(labels, fuzzy(k), 1)>μ
```

The f_{MV_query} for a keyword k and μ is:

```
SELECT DISTINCT property||'+'||domain_entity FROM VALUES WHERE
CONTAINS(value, fuzzy(k), 1)>μ
```

The function *fuzzy* it is used to find fuzzy matches.

Postgres

The f_{ME_query} for a keyword k and μ is:

```
SELECT DISTINCT name FROM ENTITIES WHERE similarity(labels,k)>μ
```

The f_{MP_query} for a keyword k and μ is:

```
SELECT DISTINCT name||'+'||domain FROM PROPERTIES WHERE WHERE
similarity(labels,k)>μ
```

The f_{MV_query} for a keyword k and μ is:

```
SELECT DISTINCT property||'+'||domain_entity FROM VALUES WHERE
similarity(value,k)>μ
```

Jena TDB

The f_{ME_query} for a keyword k and μ is:

```
SELECT DISTINCT ?class WHERE{
   ?class rdf:type rdfs:Class.
   (?class ?score ?v) text:query (rdf:label 'k~').
   filter (?score>μ)
}
```

The f_{MP_query} for a keyword k and μ is:

```
SELECT DISTINCT ((concat(?property,"+",?class) as ?s)
WHERE{
   ?property rdf:type owl:DatatypeProperty.
   ?property rdfs:domain ?class.
   (?class ?score ?v) text:query (rdf:label 'k~').
   filter (?score>μ)
}
```

The f_{MV_query} for a keyword k and μ is:

```
SELECT DISTINCT ((concat(?property,"+",?class) as ?s)
WHERE{
```

```
    ?property rdf:type owl:DatatypeProperty.
    ?property rdfs:domain ?class.
    (?class ?score ?v) text:query ('k~').
    ?i ?property ?v.
    filter (?score>μ)
}
```

5.5.2. Compute Score

This section outlines the implementations of function *score*.

The *score* of an entity bucket b_e is defined by the query b_e_score, which depends on the environment.

The *score* of a property bucket b_p is defined by the query b_p_score, which depends on the environment.

The *score* of a value bucket b_v is defined by the query b_v_score, which depends on the environment.

The score of a nucleus $n=(b_e,B_P,B_V)$ given a *KwQ+* query K is:

$$score(n, K) = [\alpha \times score(b_e)] + \left[\beta \times \frac{\Sigma_{b\in B_P}\ score(b)}{|B_p|}\right] +$$

$$\left[\gamma \times \frac{\Sigma_{b\in B_V}\ score(b,)}{|B_v|}\right] + \left[(1 - \alpha - \beta - \gamma) \times \frac{|KEYWORDS(n)|}{|K|}\right]$$

where α, β, $\gamma \in (0,1]$ are such that $0 < \alpha + \beta + \gamma \leq 1$, and weight between the score of the entity bucket, the score of the property buckets, the score of the value bucket set and the numbers of keywords that the nucleus covers. These coefficients are experimentally set.

Oracle SQL and Oracle RDF

Since, in Oracle databases, RDF databases and SQL databases coexist, the queries b_e_squery, b_p_squery, b_v_squery are the same.

The b_e_squery for an entity bucket $b_e=(e, K)$ in the schema S, where $K=\{k_1,.. ,k_m\}$, is:

```
SELECT score(1) as score FROM ENTITIES WHERE
name=η(S)(e) AND CONTAINS(labels, fuzzy(k₁ accum k₂ … accum kₘ), 1)>0
ORDER BY score DESC
FETCH FIRST 1 ROWS ONLY
```

The b_p_squery for a property bucket $b_p=(p,K,f)$ in the schema S, where $K=\{k_1,.. ,k_m\}$, is:

```
SELECT score(1) as score FROM PROPERTIES WHERE
```

```
  name=property(ɱ(S)(p)) AND
  domain=domain(ɱ(S)(p)) AND
  CONTAINS(labels, fuzzy(k₁ accum k₂ … accum kₘ), 1)>0
ORDER BY score DESC
FETCH FIRST 1 ROWS ONLY
```

The b_v_squery for a value bucket $b_v=(p, K)$ in the schema S, where $K=\{k_1,..$
$,k_m\}$, is:

```
SELECT score(1) as score FROM VALUES WHERE
  property=property(ɱ(S)(p)) AND
  domain_entity=domain(ɱ(S)(p)) AND
  CONTAINS(value, fuzzy(k₁ accum k₂ … accum kₘ), 1)>0
ORDER BY score DESC
FETCH FIRST 1 ROWS ONLY
```

The ACCUM operator gives a cumulative score based on how many query terms are found (and how frequently).

Postgres

The b_e_squery for an entity bucket $b_e=(e, K)$ in the schema S, where $K=\{k_1,.. ,k_m\}$, is:

```
SELECT similarity(labels, 'k₁ k₂ … kₘ') as score FROM ENTITIES WHERE
name=ɱ(S)(e) AND labels % 'k₁ k₂ … kₘ'
ORDER BY score DESC
FETCH FIRST 1 ROWS ONLY
```

The b_p_squery for a property bucket $b_p=(p,K,f)$ in the schema S, where $K=\{k_1,.. ,k_m\}$, is:

```
SELECT similarity(labels, 'k₁ k₂ … kₘ')  as score FROM PROPERTIES WHERE
  name=property(ɱ(S)(p)) AND
  domain=domain(ɱ(S)(p)) AND
  labels % 'k₁ k₂ … kₘ'
ORDER BY score DESC
FETCH FIRST 1 ROWS ONLY
```

The b_v_squery for a value bucket $b_v=(p, K)$ in the schema S, where $K=\{k_1,..$
$,k_m\}$, is:

```
SELECT similarity(labels, 'k₁ k₂ … kₘ')  as score FROM VALUES WHERE
  porperty=property(ɱ(S)(p)) AND
  domain_entity=domain(ɱ(S)(p)) AND
  values % 'k₁ k₂ … kₘ'
ORDER BY score DESC
FETCH FIRST 1 ROWS ONLY
```

Jena TDB

The b_e_squery for an entity bucket $b_e=(e, K)$ in the schema S, where $K=\{k_1,.. ,k_m\}$, is:

```
SELECT  ?score WHERE{
   (ɱ(S)(e) ?score ?v) text:query (rdf:label 'k₁~ … kₘ~').
   filter (?score>0)
}
ORDER BY ?score DESC LIMIT 1
```

The b_p_squery for a property bucket $b_p=(p,K,f)$ in the schema S, where $K=\{k_1,.. ,k_m\}$, is:

```
SELECT  ?score WHERE{
   (property(ɱ(S)(p)) ?score ?v) text:query (rdf:label 'k₁~ … kₘ~').
   filter (?score>0)
}
ORDER BY ?score DESC LIMIT 1
```

The b_v_squery for a value bucket $b_v=(p, K)$ in the schema S, where $K=\{k_1,.. ,k_m\}$, is:

```
SELECT ?score WHERE{
   ?i property(ɱ(S)(p)) ?v.
   (?i ?score ?v) text:query ('k₁~ … kₘ~').
   filter (?score>0)
}
ORDER BY ?score DESC LIMIT 1
```

5.6 Compiling Abstract Query

5.6.1. Relational Environment

As already mentioned, the function *query*, given an abstract query $a_q=(N,J)$ and a schema *S*, compiles and executes a structured query. For the relational environment, the function *query* compiles the query *q* as follows:

1. $\forall e \in$ entities(N), the *SELECT* clause of *q* contains the primary keys of ɱ(S)(e).

2. $\forall n \in N$ $\forall b \in [B_p(n) \cup B_v(n)]$, with $p=property(b)$, the *SELECT* clause of *q* contains the *property*(ɱ(S)(p)).

3. $\forall j \in J$, the FROM clause of *q* contains ɱ(*domain*(j)) and ɱ(*range*(j)).

4. $\forall j \in J$, the WHERE clause contains the filters defined for the foreign key $\mathfrak{m}(j)$, that is, the tuple (*name,domain_entity, domain_properties, range_entity,* and *range_properties*) in JOINS table where *name*=$\mathfrak{m}(j)$.

5. $\forall n \in N \ \forall b \in B_P(n)$, with *p=property(b)* such that *b* has a filter *f*, the *WHERE* clause of *q* filters the *property*($\mathfrak{m}(p)$) to guarantee that *f* is satisfied

6. $\forall n \in N \ \forall b \in B_V(n)$, with *p=property(b)*, the *WHERE* clause of *q* filters the *property*($\mathfrak{m}(p)$) according to the *keywords(b)*.

Section 5.6.2 exemplifies these steps for Oracle SQL and Postgres.

5.6.2. Example for the Relational Environment

Consider the database and the auxiliary tables of Section 5.3.1, the schema *S* for relational databases of Section 2.3.1, and the abstract query a_q=(N, J), where

$N=\{n_1,n_2\}$

n_1=(b_{e1}, {b_{p1}}, { }), b_{e1}=(e_1,{"country"}), b_{p1}=(p_3, {"population"},f_1), where f_1 returns true if a literal is a number greater than 1 000 000

n_2=(b_{e2}, { }, {b_{v2}}), b_{e1}=(e_1,{"city"}), b_{v2}=(p_2, {"colon"})

$J=\{j_1\}$

The steps described in Section 5.6.1 produces the query *q* as follow.

Oracle SQL

1. To add to the *SELECT* clause the primary keys of tables $\mathfrak{m}(S)(e_1)$ and $\mathfrak{m}(S)(e_1)$ that is the tables *country* and *city*.

> *q* = SELECT **city.name**, **country.name** FROM WHERE

2. To add to the *SELECT* clause of *q* the *property*($\mathfrak{m}(S)(p_3)$) and *property*($\mathfrak{m}(S)(p_2)$).

> *q* = SELECT country.name, **city.name**, **country.population**
> FROM WHERE

Note that *property*($\mathfrak{m}(S)(p_2)$) is not added because it was already added in the step 1.

3. To add to the FROM clause the tables $\mathfrak{m}(domain(j_1))$ and $\mathfrak{m}(range(j_2))$, that is, the tables *country* and *city*.

> *q* = SELECT country.name, city.name, country.population
> FROM **country**, **city** WHERE

4. To add to the WHERE clause the filters defined for the foreign key ɱ(*j₁*), that is, the tuple (*city_country, country, name, city, of_country*).

> *q* = SELECT country.name, city.name, country.population
>   FROM country, city
>   WHERE **country.name=city.of_country**

5. To add to the *WHERE* clause the filter for *property*(ɱ(*p₃*)) to guarantee that *f₁* is satisfied.

> *q* = SELECT country.name, city.name, country.population
>   FROM country, city
>   WHERE country.name=city.of_country **AND**
>   **country.population > 1000000**

6. To add to the *WHERE* clause the filter for *property*(ɱ(*p₁*)) with the value "colon".

> *q* = SELECT country.name, city.name, country.population
>   FROM country, city
>   WHERE country.name=city.of_country AND
>   country.population > 1000000 **AND**
>   **contains(city.name, fuzzy('colon'),1)>0**

Postgres

Steps from 1 to 5 are equivalent to Oracle SQL:

> *q* = SELECT country.name, city.name, country.population
>   FROM country, city
>   WHERE country.name=city.of_country **AND**
>   **country.population > 1000000**

6. To add to the *WHERE* clause the filter for *property*(ɱ(*p₁*)) with the value "colon".

> *q* = SELECT country.name, city.name, country.population
>   FROM country, city
>   WHERE country.name=city.of_country AND
>   country.population > 1000000 **AND**
>   **city.name % 'colon'**

5.6.3. RDF Environment

For the RDF environment, the function *query* compiles the query *q* as follows:

1. $\forall e \in$ entities(N), the *SELECT* clause of *q* contains ?v_e and the WHERE clause contains the triple (?v_e, rdf:type, ɱ(*e*)).

2. $\forall n \in N \ \forall b \in [B_p(n) \cup B_v(n)]$, with $p=$proprety(b), the *SELECT* clause of q contains $?v_p$ and the WHERE clause contains the triple $(?v_e,$ property(ɱ(p)), $?v_p)$. Note that $?v_e$ is the variable associated with domain(p) in step 1.

3. $\forall j \in J$, the *WHERE* clause of q contains the triple $(?v_{e1},$ ɱ(j), $?v_{e2})$. The variables $?v_{e1}$ and $?v_{e2}$ are associated with domain(j) and range(j), respectively.

4. $\forall n \in N \ \forall b \in B_P(n)$, with $p=property(b)$ such that b has a filter f, the *WHERE* clause of q filters the variable $?v_p$, associated with p in step 2, guaranteeing that f is satisfied.

5. $\forall n \in N \ \forall b \in B_V(n)$, with $p=property(b)$, the *WHERE* clause of q filters the variable $?v_p$, associated with p in step 2, according to *keywords*(b).

Section 5.6.3 exemplifies these steps for Oracle RDF and Jena TDB databases.

5.6.4. Example for the RDF Environment

Consider the dataset and the auxiliary tables of Section 5.3.2, the schema S for RDF dataset of Section 2.3.1, and the abstract query $a_q=(N, J)$, where

$N=\{n_1,n_2\}$,

$n_1=(b_{e1}, \{b_{p1}\}, \{\})$, $b_{e1}=(e_1,\{\text{"country"}\})$, $b_{p1}=(p_3, \{\text{"population"}\},f_1)$,

where f_1 returns true if a literal is a number greater than 1 000 000

$n_2=(b_{e2}, \{\}, \{b_{v2}\})$, $b_{e1}=(e_1,\{\text{"city"}\})$, $b_{v2}=(p_2, \{\text{"colon"}\})$

$J=\{j_1\}$

The steps described in Section 5.6.3 produces the query q as follow.

Oracle RDF

1. To add to the *SELECT* clause the variables $?v_{country}$ and $?v_{city}$, and to add to the WHERE clause triples with $(?v_{country}$, rdf:type, :country) and $(?v_{city},$ rdf:type, :city).

```
q = SELECT ?vcountry ?vcity
    WHERE{
        ?vcountry rdf:type :city.
        ?vcity rdf:type :country.
```

```
}
```

2. To add to the *SELECT* clause ?$v_{population}$ and ?v_{name}, and to add to the WHERE clause the triples (?$v_{country}$, :population, ?$v_{population}$) and (?v_{city}, :name, ?v_{name}).

```
q = SELECT ?v_country ?v_city ?v_population ?v_name
  WHERE{
    ?v_country rdf:type :city.
        ?v_city rdf:type :country.
        ?v_country :population ?v_population.
        ?v_city :name ?v_name.
  }
```

3. To add to the *WHERE* clause of *q* the triple (?$v_{country}$, :of_country, ?v_{city}).

```
q = SELECT ?v_country ?v_city ?v_population ?v_name
  WHERE{
  ?v_country rdf:type :city.
        ?v_city rdf:type :country.
        ?v_country :population ?v_population.
        ?v_city :name ?v_name.
        ?v_country :of_country ?v_city.
  }
```

4. To add to the *WHERE* clause the filter for ?$v_{population}$ to guarantee that f_1 is satisfied.

```
q = SELECT ?v_country ?v_city ?v_population ?v_name
  WHERE{
  ?v_country rdf:type :city.
        ?v_city rdf:type :country.
        ?v_country :population ?v_population.
        ?v_city :name ?v_name.
        ?v_country :of_country ?v_city.
        filter (?v_population>1000000)
  }
```

5. To add to the *WHERE* clause the filter to the ?v_{name} with the value "colon".

```
q = SELECT ?v_country ?v_city ?v_population ?v_name
  WHERE{
  ?v_country rdf:type :city.
        ?v_city rdf:type :country.
        ?v_country :population ?v_population.
        ?v_city :name ?v_name.
        ?v_country :of_country ?v_city.
        filter (?v_population>1000000)
        filter (oratext:contains(?v_name, fuzzy('colon'),1)>0)
  }
```

Jena TDB

Steps from 1 to 4 are equivalent to Oracle RDF:

```
q = SELECT ?v_country ?v_city ?v_population ?v_name
    WHERE{
?v_country rdf:type :city.
            ?v_city rdf:type :country.
            ?v_country :population ?v_population.
            ?v_city :name ?v_name.
            ?v_country :of_country ?v_city.
            filter (?v_population>1000000)
        }
```

5. To add to the *WHERE* clause the filter for the $?v_{name}$ with the value "colon".

```
q = SELECT ?v_country ?v_city ?v_population ?v_name
    WHERE{
?v_country rdf:type :city.
            ?v_city rdf:type :country.
            ?v_country :population ?v_population.
            ?v_city :name ?v_name.
            ?v_country :of_country ?v_city.
            filter (?v_population>1000000).
            (?v_city ?score ?v_name) text:query ('colon~').
        }
```

5.7 User Interface

The main requirements of a keyword search interface are a text box, where the user types keywords, and a layout area to present the answers to the user. The user interface offers an auto-completion feature to help users formulate a keyword-based query, as in Figure 18. The interface suggests new keywords based on the previous keywords, the schema vocabulary, and the labels.
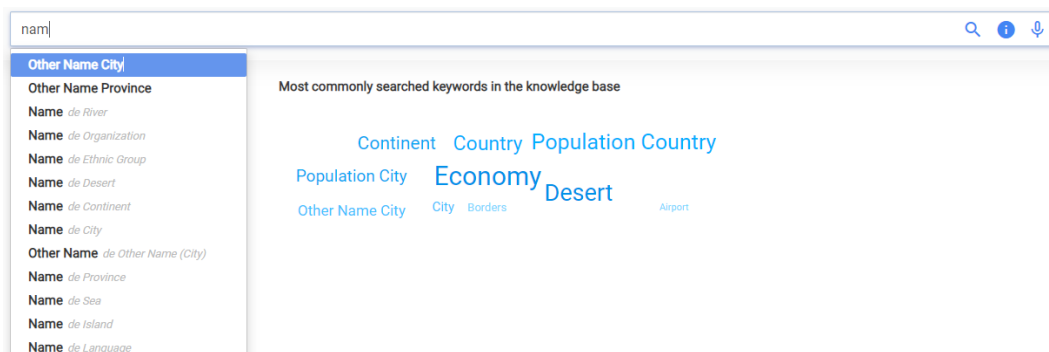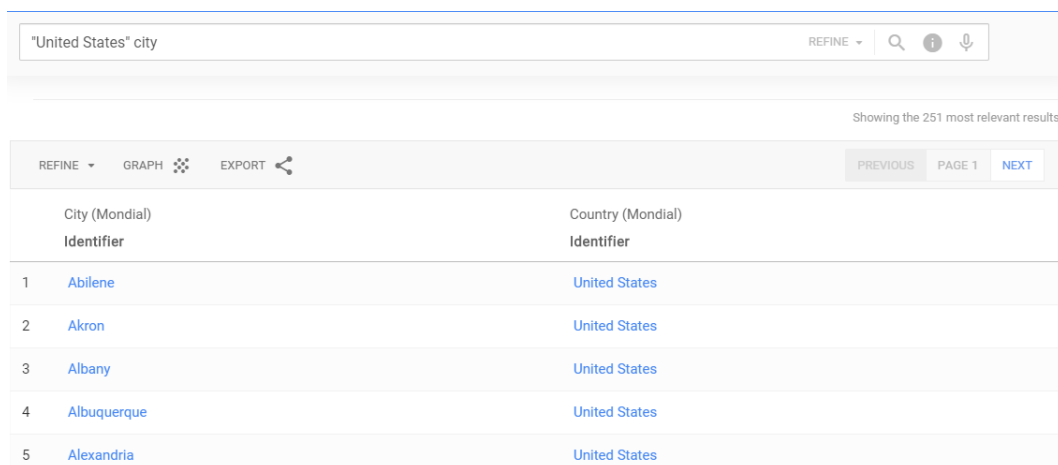
Figure 18. Example of auto-completion.

Since an answer *A* for a keyword-based query *K* over an RDF dataset *T* is formally a subset of *T*, it would be consistent to present *A* as a set of triples. However, this option proved to be inconvenient for the users, which are more familiar with tabular data, as in relational systems. We then implemented a user interface that presents the results of *K* by combining a table (Figure 19) with the Steiner tree underlying the SQL or SPARQL query, which is exhibited by clicking on the graph button, as in Figure 20. Note that there is an indication of the *Page* and a *Next* button so that the user can see all the results.

Another feature of the presentation is to show to the user the entities that compose the answers, to allow her to see the available properties of these entities, and to select those that she wants to include in the table. Figure 21 shows an example of this feature, in which a user wants to include the population of the retrieved country.

Figure 19. Example of tabular answer.
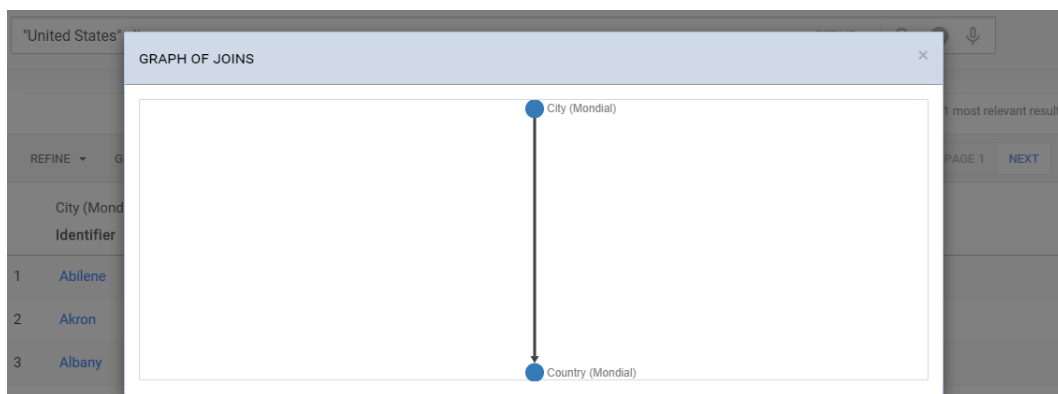
Figure 20. Example of query graph.

Figure 21. Property selection.

Besides the keyword search engine, we also allow the user to navigate to an instance by clicking on the links and see its data (Figure 22) and its relations with other instances (Figure 23). The user may continue navigating through other instances to discover more data, as shown in Figure 23.

Figure 22. Example of instance information.
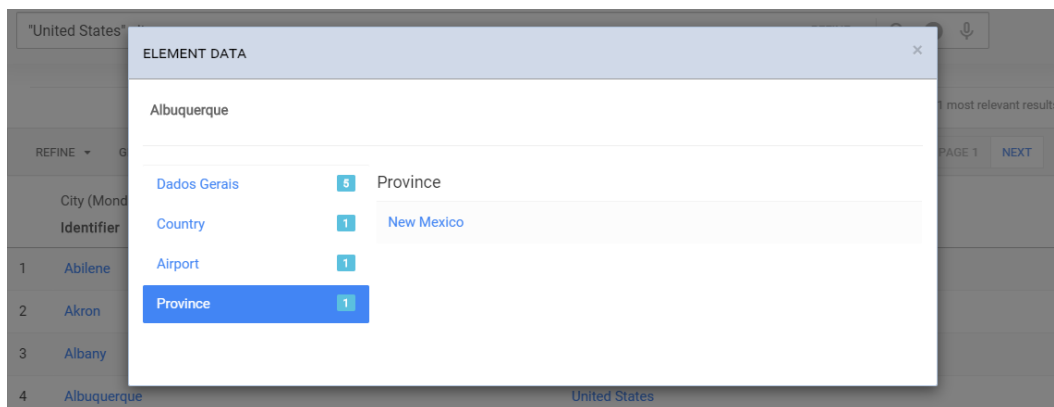
Figure 23. Example of instance relations.
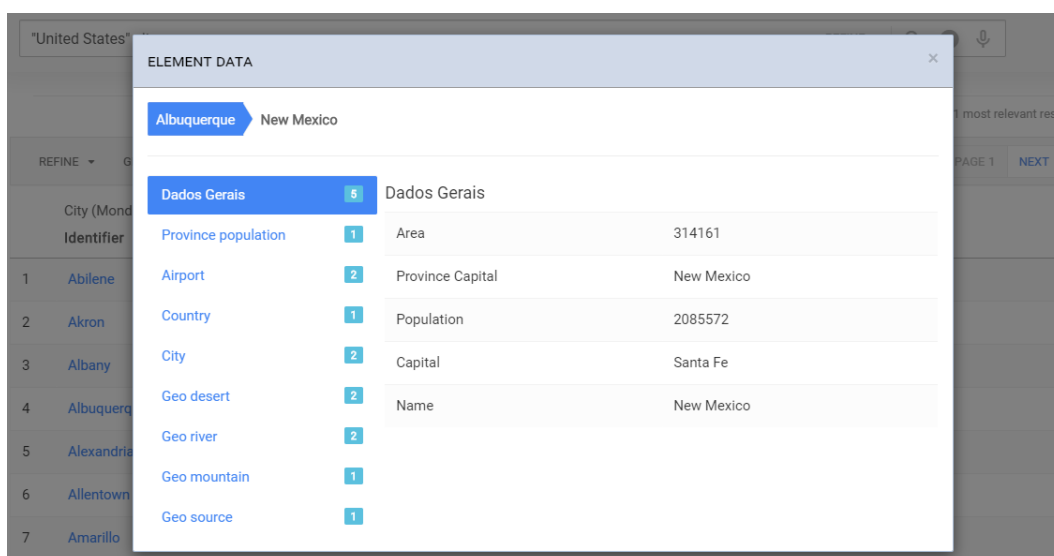
Figure 24. Example of navigation.

As we explained in Section 4.3, the user can choose other elements of the schema to find new answers, by clicking on the *refine* button, as in Figure 25, or he can choose another graph, if one exists, as in Figure 26.
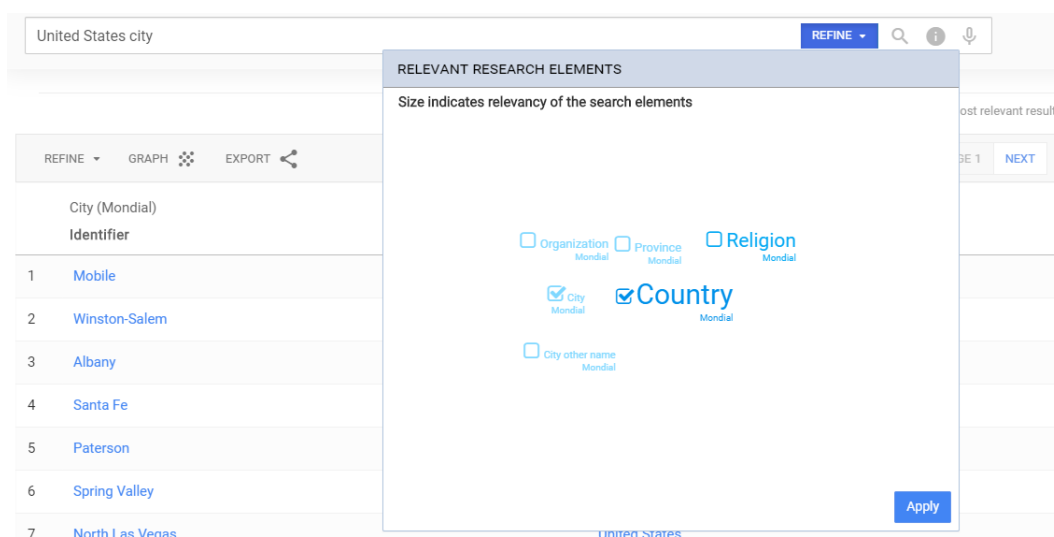
Figure 25. Example of feedback with other resources.
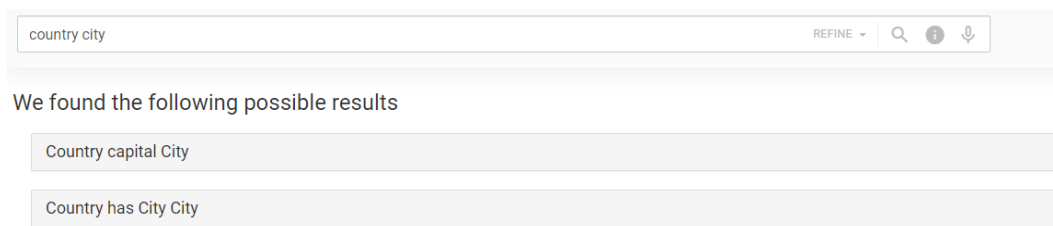
Figure 26. Example of feedback with multiple Steiner trees.

5.8 Chapter Conclusion

In this chapter, we presented the architecture, the implementation details and the interface of the DANKE tool. The architecture reflects the steps of the keyword search process, as defined in Chapter 4. Also, we discussed, for each database management system, how to improve the performance of the algorithm, how to build the schema, and how to construct the queries that will be executed. Finally, the interface permits the user to submit queries, analyze answers, give a feedback and navigate through the instances of the graph.

6
Evaluation

6.1 Setup

To evaluate the Translation algorithm, we ran Coffman's benchmark (Coffman & Weaver, 2010) for Mondial and IMDb. We compared and evaluated the algorithm using Oracle, that is, using the Oracle SQL environment and the Oracle RDF environment. The Mondial dataset is available at https://www.dbis.informatik.uni-goettingen.de/Mondial/ and the IMDb dataset is available at https://sites.google.com/site/ontopiswc13/ home/imdb-mo. The versions of IMDb and Mondial used are different from the versions used in Coffman's benchmark. Continuing our experiments, we used 25 queries from QALD-2[5] (adapted to keyword search) to evaluate the Translation algorithm over the MusicBrainz database, available at https://musicbrainz.org/doc/MusicBrainz_Database/Download.

Table 16 shows basic statistics about the RDF datasets and relational databases used in the experiments.

Table 16. Statistics – Mondial and IMDb.

RDF Dataset	#Triples	
	Mondial	IMDb
Class declarations	4	11
Object property declarations	62	11
Datatype prop. declarations	130	25
subClassOf axioms	-	-
Indexed properties	71	7
Indexed prop. instances	11.094	12.609.418
Class instances	43.869	70.520.744
Object property instances	63.652	204.917.673
Total number of triples	**235.387**	**382.295.213**

Relational Database	#Objects		
	Mondial	IMDb	MusicBrainz
Number of Relations	40	11	15
Number of Attributes	187	20	31
Number of Tuples	40,247	70.520.722	51.965.280
Size (in GB)	**0.11**	**60.63**	**16,6**

---

[5] https://github.com/ag-sc/QALD

All experiments used the RESTful Web service that the DANKE tool provides, developed in Java. The Web service ran on a desktop machine with OS Windows 10 Pro, a quad-core processor Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz, 16 GB of RAM. The relational databases and RDF datasets were stored in Oracle 12c, running on a quad-core machine with processor Intel(R) Core(TM) i5 CPU 660 @ 3.33GHz, 7GB of RAM, and 4096 KB of Cache size, and configured with a PGA size of 324 MB and an SGA size of 612 MB with 148 MB of cache size and 296 MB of buffer cache.

Table 17 shows the time (in minutes) taken by the pre-processing task, to build and index the auxiliary tables and compute the shortest path index.

Table 17 Time taken by the pre-processing tasks

Tasks		Time (in minutes)		
		Mondial	IMDb	MusicBrainz
Auxiliary tables	Relational	0.3	38	32
	RDF	0.3	40	-
Shortest path index		0.06	0.05	0.7

6.2 Coffman's benchmark

The results obtained by the Translation algorithm were exactly the same in both the RDF and the relational environments, as expected, since the construction process of the abstract query, matches, and score, was the same in both cases. For the experiments, we measured the *query build time* – the time taken by the translation algorithm until the construction of the SQL or SPARQL query, and the *total elapsed time* – the time from the submission of the query until the display of the first 75 results. We also included the individual Mean Average Precision (MAP) score for each query.

6.2.1. Experiments with Mondial

We tested the tool against relational and RDF versions of the Mondial dataset using the list of 50 keyword-based queries defined in Coffman's benchmark (Coffman & Weaver, 2010). As the dataset that we used for the experiments is not the same used in Coffman's benchmark, two of the queries, Query 7 and Query 14, contain keywords that do not occur in the database, and then the number of valid queries is 48.

Table 18 shows the test results. To summarize, the tool correctly answered 33 queries, nearly 69% of the 48 valid queries in Coffman's benchmark for the version of Mondial adopted. Both versions reached the same results. A brief analysis of the correctness of the results follows:

Queries 1-5 – countries: All queries were correctly answered.

Queries 6-10 – cities: Query 7 returned no answer since the version of Mondial adopted does not have a city called "Sonsonate". The other queries are correctly answered.

Queries 11-15 – geographical: Query 14 returned no answer since the version of Mondial adopted had no desert called "Asauad". Query 12 returned the country "Niger". With the feedback algorithm, Query 12 returns the river "Niger" that is the expected answer.

Queries 16-20 – organization: All queries returned answers, covering all keywords; some keywords were not listed in class Organization, then the results did not coincide with the expected answers of the benchmark. We considered that all queries were correctly answered.

Queries 21-25 – border between countries: The keywords matched the labels of two instances of class Country, but the keywords were not sufficient to infer that the question is about the borders between countries and, thus, were not correctly answered.

Queries 26-35 – geopolitical or demographic information: The expected answer of Query 35 was obtained using a feedback mechanism. We considered that all queries were correctly answered.

Queries 36-45 – member organizations two countries belong to: The expected answer is the list of organizations that the countries belong to; however, the tool did not identify the IS_MEMBER class when generating the nucleuses.

Queries 46-50 – Miscellaneous: The expected answer of Query 40 and Query 50 were obtained using a feedback mechanism. We considered that all queries were correctly answered.

Table 18 Mondial Results

Query Number	Keywords	SQL			SPARQL		
		Build Time	Total time	MAP	Build Time	Total time	MAP
1	Thailand	0.036	0.065	1,0	0.049	0.555	1,0
2	Netherlands	0.035	0.066	1,0	0.023	0.318	1,0
3	Georgia	0.039	0.065	1,0	0.043	0.354	1,0
4	country China	0.043	0.067	1,0	0.031	0.321	1,0
5	Bangladesh	0.024	0.051	1,0	0.032	0.326	1,0
6	Alexandria	0.056	0.201	1,0	0.030	0.580	1,0
7	sonsonate	The version of Mondial the keyword "Sonsonate" does not exist					
8	Xiaogan	0.064	0.099	1,0	0.030	0.320	1,0
9	city Glendale	0.044	0.074	1,0	0.026	0.532	1,0
10	city Granada	0.037	0.059	1,0	0.030	0.458	1,0
11	Lake Kariba	0.032	0.155	1,0	0.035	0.311	1,0
12	Niger	0.117	0.240	1,0	0.047	0.486	1,0
13	Arabian Sea	0.027	0.138	1,0	0.012	0.355	1,0
14	Asauad	The version of Mondial the keyword "Asauad" does not exist					
15	Sardegna	0.046	0.203	1,0	0.045	0.333	1,0
16	Arab Cooperation Council	0.149	0.516	1,0	0.121	1.742	1,0
17	world labor	0.109	0.279	1,0	0.083	1.354	1,0
18	Islamic Conference	0.154	0.251	1,0	0.079	1.311	1,0
19	30 group	0.021	0.065	1,0	0.014	0.328	1,0
20	Caribbean economic	0.085	0.124	1,0	0.049	1.099	1,0
21	slovakia hungary	-	-	0	-	-	0
22	mongolia china	-	-	0	-	-	0
23	niger algeria	-	-	0	-	-	0
24	kuwait saudi arabia	-	-	0	-	-	0
25	lebanon syria	-	-	0	-	-	0
26	Cameroon economy	0.075	0.106	1,0	0.042	0.872	1,0
27	Nigeria gdp	0.066	0.115	1,0	0.044	1.278	1,0
28	Mongolia Republic	0.091	0.132	1,0	0.090	0.701	1,0
29	Kiribati politics	0.043	0.068	1,0	0.042	0.831	1,0
30	Poland language	0.069	0.106	1,0	0.027	0.809	1,0
31	Spain Galician	0.103	0.219	1,0	0.055	0.974	1,0
32	Uzbekistan eastern orthodox	0.066	0.312	1,0	0.068	1.982	1,0
33	Haiti religion	0.094	0.131	1,0	0.040	0.923	1,0
34	Suriname ethnic group	0.089	0.121	1,0	0.053	1.091	1,0
35	Slovakia German	0.064	0.101	1,0	0.056	0.935	1,0
36	poland cape verde organization	-	-	0	-	-	0
37	saint kitts cambodia	-	-	0	-	-	0
38	marshall islands grenadines organization	-	-	0	-	-	0
39	czech republic cote divoire organization	-	-	0	-	-	0
40	panama oman	-	-	0	-	-	0
41	iceland mali	-	-	0	-	-	0
42	guyana sierra leone	-	-	0	-	-	0
43	mauritius india	-	-	0	-	-	0
44	vanuatu afghanistan	-	-	0	-	-	0
45	libya australia	-	-	0	-	-	0
46	Hutu Africa	0.053	0.263	1,0	0.041	1.044	1,0
47	Serb Europe	0.062	0.148	1,0	0.052	0.980	1,0
48	Uzbek Asia	0.066	0.105	1,0	0.059	0.983	1,0
49	Rheine Germany	0.040	0.093	1,0	0.046	0.948	1,0
50	Egypt Nile	0.072	0.122	1,0	0.064	1.046	1,0

Figure 27 shows, on the Y-axis, the query build time and the total elapsed time, in seconds, of each query in Coffman's benchmark, numbered 1 to 50 on the X-axis. Note that, for each keyword-based query: the SPARQL total elapsed time (shown as a dot) was always much larger than the SQL total elapsed time (shown as a cross), and the SPARQL and the SQL query build times (respectively shown

as squares and triangles) were nearly the same (most squares are on top of the triangles). Section 6.3 discusses these points for both experiments.
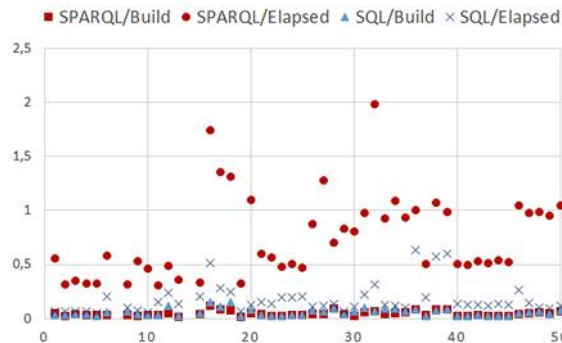

Figure 27 Mondial - Build Time and Total Elapsed Time

6.2.2. Experiments with IMDb

As we mentioned before, we tested the tool against a full and more recent version of IMDb, which we refer to as *Full IMDb* to differ it from the *Restricted IMDb* version used in Coffman's benchmark. Contrasting with the Restricted IMDb, the Full IMDb features a much more complex conceptual schema (see Table 16). Furthermore, while the Restricted IMDb has data only about movies, the Full IMDb has data about movies, series, episodes, video games, etc. We considered these differences when comparing the result of our tool with that of the benchmark.

In order to reduce ambiguity when using the Full IMDb, as compared with the Restricted IMDb, and consequently, to improve processing time, we surrounded most keywords with quotes. For instance, consider the query {denzel, washington}. If we treat the keywords separately, we find that {denzel} has 670 data matches, while {washington} has 23,720. Indeed, "washington" is a very ambiguous keyword, since it matches the name of an actor, movie, TV series, city, state, etc. Hence, if we treat the query as "denzel OR washington" we have a total of 23,851 data matches. However, if we treat the query as "denzel AND washington", we have only 539 data matches. For example, Table 19 presents the SQL queries and the total elapsed times (in seconds) to compute the value score for class movie_info and property info; we note that the total time to create an abstract query was, on average, 30 times faster with quotes.

Table 19 Data Match Scores

	Query	Elap
with quotes	SELECT score(1) as score FROM VALUES WHERE domain = 'movie_info' and property = 'info' and contains (value, 'denzel washington', 1) > 0 ORDER BY score DESC FETCH FIRST 1 ROWS ONLY	0.14
no quotes	SELECT score(1) as score FROM VALUES WHERE domain = 'movie_info' and property = 'info' and contains (value, 'denzel accum washington', 0) > 0) ORDER BY score DESC FETCH FIRST 1 ROWS ONLY	5.04

Table 20 shows the test results. To summarize, the tool correctly answered 45 queries, nearly 90% of the 50 queries in Coffman's benchmark for IMDb. We note that the use of quotes only improved the query build time and did not change the final results.

A brief analysis of the correctness of the results follows:

Queries 1-20 – <actors or movies>: relevant results contain a single tuple of the specified individual or film. For Query 13, the tool returned the actor named Casablanca, not the movie. With the feedback algorithm, Query 13 returned the movie, that is the expected answer. We considered that all queries were correctly answered.

Queries 21-30 – title+<character>: For Queries 22, 23, 28, the tool returned a wrong answer since the name of the character is also the name of some title; the expected results of those queries were obtained using a feedback mechanism. We considered that all queries were correctly answered.

Queries 31-35 – title+<quote>: All queries were correctly answered.

Queries 36, 44, 46-49 – <actor>+<character, director or writer>: Queries *36* and 44 were correctly answered. For queries from 46 to 49 the tool returned a wrong answer because the relation is between two instances of the same class.

Queries 37, 41 – <actor>+<year>. For Query 41, the tool returned a wrong answer since the name of the actress is also the name of a title; the expected results of this query were obtained using a feedback mechanism. We considered that all queries were correctly answered.

Queries 38-40 – <actor>+<film>: relevant results must denote the character that an actor plays in a film. For queries 38 and 39, the tool returned a wrong answer since the name of the actor is also the name of some character; the expected results of this query were obtained using a feedback mechanism. We considered that all queries were correctly answered.

Queries 42, 43, 45 – name+<character>: For all queries, the tool returned a wrong answer since it matched the keyword "name" with property name from class Character; the expected results of this query were obtained using a feedback mechanism. We considered that all queries were correctly answered.

Queries 50 – name+<character>: The tool returned a wrong answer since two keywords match instances from the same class.

Table 20 IMDb Results

Query Number	Keywords	SQL			SPARQL		
		Build Time	Total time	MAP	Build Time	Total time	MAP
1	"denzel washington"	2.083	2.252	1,0	2.173	4.895	1,0
2	"clint eastwood"	2.145	2.171	1,0	3.004	15.698	1,0
3	"john wayne"	4.152	4.225	1,0	4.717	44.616	1,0
4	"will smith"	2.572	2.713	1,0	3	42.369	1,0
5	"harrison ford"	2.037	2.104	1,0	2.248	42.529	1,0
6	"julia roberts"	2.34	2.431	1,0	2.363	44.548	1,0
7	"tom hanks"	2.005	2.063	1,0	3.076	14.891	1,0
8	"johnny depp"	2.43	2.526	1,0	2.994	14.001	1,0
9	"angelina jolie"	2.447	2.5	1,0	2.405	12.188	1,0
10	"morgan freeman"	2.04	2.096	1,0	2.018	13.078	1,0
11	"gone with the wind"	2	2.143	1,0	2.037	43.739	1,0
12	"star wars"	7.269	7.466	1,0	7.217	46.666	1,0
13	"casablanca"	1.376	1.414	1,0	1.514	12.998	1,0
14	"lord of the rings"	2.985	3.102	1,0	3.005	14.881	1,0
15	"the sound of music"	1.874	1.944	1,0	1.969	45.506	1,0
16	"wizard of oz"	2.818	3.146	1,0	3.119	15.166	1,0
17	"the notebook"	0.513	0.569	1,0	0.572	10.103	1,0
18	"forrest gump"	0.802	0.855	1,0	1.019	13.247	1,0
19	"the princess bride"	0.481	0.585	1,0	0.507	41.915	1,0
20	"the godfather"	2.78	2.926	1,0	3.031	14.239	1,0
21	title "atticus finch"	0.323	12.629	1,0	0.324	16.968	1,0
22	title "indiana jones"	1.519	1.862	1,0	2.008	13.141	1,0
23	title "james bond"	2.469	2.717	1,0	2.655	46.868	1,0
24	title "rick blaine"	0.545	13.057	1,0	0.522	17.864	1,0
25	title "will kane"	0.11	2.481	1,0	0.129	19.06	1,0
26	title "dr. hannibal lecter"	0.536	14.151	1,0	0.52	17.716	1,0
27	title "norman bates"	0.714	13.105	1,0	0.75	18.867	1,0
28	title "darth vader"	0.637	0.718	1,0	0.689	11.18	1,0
29	title "the wicked witch of the west"	0.601	13.14	1,0	0.641	17.57	1,0
30	title "nurse ratched"	0.248	12.869	1,0	0.246	17.233	1,0
31	title "frankly my dear i don't give a damn"	0.159	0.742	1,0	0.154	7.895	1,0
32	title "i'm going to make him an offer he can't refuse"	0.284	0.443	1,0	0.03	29.944	1,0
33	title "you don't understand i coulda had class i coulda been a contender i coulda been somebody instead of a bum which is what i am"	0.108	0.748	1,0	0.108	8.334	1,0
34	title "toto, i've a feeling we're not in kansas any more"	0.897	0.986	1,0	0.921	9.098	1,0
35	title "here's looking at you kid"	0.4	1.989	1,0	0.32	28.921	1,0
36	hamill skywalker	4.231	4.484	1,0	4.147	18.766	1,0
37	"tom hanks" year = "2004"	1.755	13.094	1,0	2.092	19.562	1,0
38	"henry fonda" "yours mine ours character"	1.137	12.452	1,0	1.237	23.098	1,0

39	"russell crowe" gladiator character	0.598	0.751	1,0	0.605	25.764	1,0
40	"brent spiner" "star trek"	7.818	8.506	1,0	7.868	30.819	1,0
41	"audrey hepburn" year = "1951"	1.216	1.305	1,0	1.31	11.792	1,0
42	name "jacques clouseau"	0.193	0.216	1,0	0.204	11.566	1,0
43	name "jack ryan"	0.064	0.093	1,0	0.032	38.878	1,0
44	"rocky" "stallone"	13.765	15.279	0	14.016	30.179	0
45	name "terminator"	0.642	0.721	1,0	0.84	15.023	1,0
46	"harrison ford" "george lucas"	3.097	3.333	0	3.51	20.146	0
47	"sean connery" "fleming"	6.73	7.084	0	6.49	25.58	0
48	"keanu reeves" "wachowski"	1.778	2.059	0	1.753	13.084	0
49	"dean jones" "herbie"	2.672	3.025	0	3.01	20.091	0
50	"indiana jones" "last crusade" "lost ark"	3.989	4.746	0	4.029	21.359	0

Figure 28 shows the query build time and the total elapsed time (in seconds, on the Y-axis) of each query in Coffman's benchmark (numbered 1 to 50, on the X-axis) for the SPARQL and SQL versions. Note that, again, the SPARQL total elapsed time was always much larger than the SQL total elapsed time, except for a few queries (crosses on top of squares), and that the SPARQL and the SQL query build times were nearly the same.
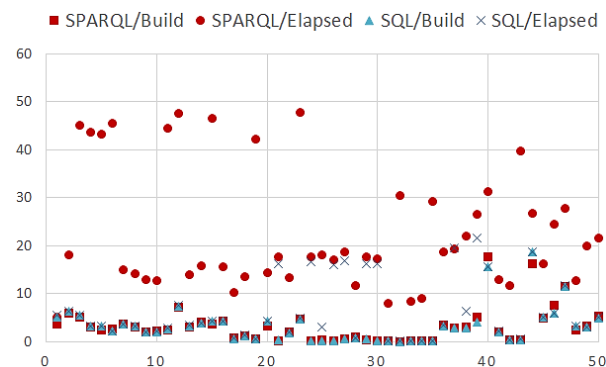

Figure 28 IMDb - Build Time and Total Elapsed Time

6.2.3. Experiments with MusicBrainz

We tested the tool against relational versions of the MusicBrainz dataset and a list of 25 KwQ+ queries adapted from the question of QALD-2[6].

Table 21 shows the questions, the KwQ+ queries for each question, and the test results. As for IMDb tests, to improve processing time, we surrounded some keywords with quotes.

---

[6] https://github.com/ag-sc/QALD

To summarize, the precision was nearly 88%, 23 of the 25 valid queries achieve 100% of precision. A brief analysis of the correctness of the results follows:

Query 14 and Query 19 failed because the database does not contain information about the track composer or album.

Query 7 achieved only 80% of precision and Query 21 failed because the database does not have information about the period of time a person belonged to a band.

Table 21 Music Brainz Results

Query Number	QALD Question	Keywords	Build Time	Total time	MAP
1	Which singles did Slayer release?	slayer group track	0,371	2,624	1,00
2	Which groups was David Bowie a member of?	David Bowie person "music group"	1,978	2,360	1,00
3	When was the band Dover founded?	artist Dover begin year	0,478	0,793	1,000
4	How many albums did Michael Jackson record?	"Michael Jackson" artist album	0,373	1,943	1,000
5	Who composed the Star Wars soundtrack?	"Star Wars" track person	0,241	1,918	0,14
6	Which artists have their 50th birthday on May 30, 1962?	person "begin date day"=30 "begin date month"=5 "begin date year"=1962	0,044	0,786	1,00
7	Give me the present members of The Cure	person group "The Cure"	0,172	0,502	0,80
8	Give me all Kraftwerk albums!	Kraftwerk group album	0,353	1,482	1,00
9	How many bands are called Nirvana?	group Nirvana	0,253	0,467	1,00
10	When did the Sex Pistols break up?	"Sex Pistols" end year	0,432	0,687	1,00
11	Was Quee MacArthur a member of Queen?	"Quee MacArthur" music group	0,105	0,404	1,00
12	When is the birthday of Tom Waits?	"Tom Waits" person "begin date day" "begin date month" "begin date year"	0,091	0,337	1,00
13	Which artists were born on the 29th of December 1960?	person "begin date day"=29 "begin date month"=12 "begin date year"=1960	0,043	0,851	1,00
14	How many bands broke up in 2010?	roup end year 2010	0,059	0,637	1,00
15	Give me all albums with the BBC Symphony Orchestra.	"BBC Symphony Orchestra" group album	0,446	1,401	1,00
16	Give me all bands that Michael Stipe is a member of.	"Michael Stipe" person group	0,066	0.431	1,00
17	How many albums did Amy Macdonald release?	"Amy Macdonald" artist album	0,224	1,397	1,00
18	Give me all live albums by Michael Jackson	"Michael Jackson" artist album live	0,727	1,966	1,00
19	Who produced the album In Utero?	"In Utero" album artist	0,247	1,5	0,08
20	How long is the song Hardcore Kids?	Hardcore Kids duration	0,403	1,388	1,00
21	When did Kurt Cobain join Nirvana?	"Kurt Cobain" person Nirvana group	0,383	0,583	0,00
22	Give me all songs by Aretha Franklin	"Aretha Franklin" artist track	0,331	2,164	1,00
23	Since when does Millencolin exist?	Millencolin group start year	0,316	0,490	1,00

24	How many members does the band Trio have?	group Trio person	0,245	0,389	1,00
25	Are there members of the Ramones that are not called Ramone?	group Ramones person name!="Ramone"	0,188	20,716	1,00

6.3 Chapter Conclusion

This section summarizes the lessons learned from the experiments. Table 22 shows the total elapsed time, from the submission of the keyword-based query until the display of the results, and the query build time, i.e., the time to process matches and construct the SQL or SPARQL queries.

Quality of the query results. The correctness of the translation process for the 50 keyword-based queries of Coffman's benchmark was satisfactory, for both Mondial and IMDb, in both environments. For each keyword search executed, the results obtained were exactly the same in both the RDF and relational environments, as expected, since the construction process of the abstract query was the same in both cases. In this aspect, the difference is in the concrete query structure (SPARQL versus SQL) and not in the query target. The correctness of the translation process for the 25 questions, adapted to *KwQ+* queries, from QALD-2 for MusicBrainz dataset was also satisfactory.

Total elapsed time. The total elapsed time was reasonable, on average, in all experiments. Even for a large database, such as IMDb, the total elapsed time was, on average, nearly 4 seconds, in the relational environment, but raised to 22 seconds, in the RDF environment. Indeed, the total elapsed time of the SQL queries was 4-6 times faster than the SPARQL queries, on average. Queries with contain filter use a text index, which is over all object values of the triples, for RDF datasets. However, for relational databases, there is a separate, smaller index for each text attribute. Thus, the total elapsed time of SQL queries with a contain filter was smaller than that of SPARQL queries.

Queries with a contain filter use a text index, which is over all object values of the triples, in RDF datasets. But, in relational databases, there is a separate, smaller index for each text attribute. Thus, the total elapsed time of SQL queries with a contain filter was smaller than that of SPARQL queries (as Queries 12-14 in Section 6.2.2).

Query build time. In all experiments, the query build time was nearly the same in both environments, since processing matches, calculating scores and constructing the abstract query were the same.

In the relational environment, for the experiments with Mondial, the query build time accounted for 40-50% of the total elapsed time, on average; for the experiments with IMDb, it raised to slightly over 70-75%, possibly due to the ambiguity of IMDb data; for the experiment with MusicBrainz, it was 30%.

By contrast, in the RDF environment, the query build time accounted for only 6-15% of the total elapsed time, on average. This behavior can be explained because matching is a costly process in both environments, but SPARQL queries took much longer to execute than SQL queries.

Table 22 Summary of the experiments

		Database		
		Mondial	IMDb	MusicBrainz
Total Elapsed Time (in seconds)				
Average	Relational	0.147	6.050	1.93
	RDF	0.802	22.679	–
Maximum	Relational	0.516	21.560	20.72
	RDF	1.982	47.680	–
Minimum	Relational	0.051	0.080	0.34
	RDF	0.311	7.950	–
Query Build Time (in seconds)				
Average	Relational	0.066	3.723	0.34
	RDF	0.047	3.219	–
Maximum	Relational	0.154	18.690	1.98
	RDF	0.121	17.680	–
Minimum	Relational	0.021	0.030	0.04
	RDF	0.012	0.030	–
Query Build Time / Total Elapsed Time (in percentage)				
Average	Relational	50.5%	77.6%	30.7&
	RDF	6.6%	15.5%	–
Maximum	Relational	73.6%	99.7%	83.8
	RDF	13.5%	60.3%	–
Minimum	Relational	19.6%	0.7%	0.9%
	RDF	3.3%	0.1%	–

7
Conclusions and Future Work

7.1 Conclusions

In the last years, database applications that offer keyword-based query interfaces became a relevant research topic with the goal of hiding from users the non-friendly queries. The main contribution of this thesis, presented in Chapter 4, is a translation algorithm for graph databases that, given an advanced keyword query, produces answers. Since to find the optimal solutions is an NP-Complete problem, we proposed heuristics and incorporated them in the translation algorithm. The translation algorithm adopts standard definitions of schema and query, and functions to find the match and calculate the score, which allows to easily extend the algorithm for different environment.

Chapter 5 detailed the architecture, implementation and interface of a tool and a framework that uses the translation algorithm. This framework is the second contribution of this thesis and supports ORACLE 12c and JENA TDB, for the RDF environment, and ORACLE 12c and POSTGRES, for the relational environment.

The third contribution, presented in Chapter 6, is the evaluation of the translation algorithm. We tested the proposed algorithm with popular keyword search benchmarks for Mondial, IMDb and MusicBrainz. The correctness of the translation process was satisfactory. The total elapsed time was reasonable, on average, in all experiments. Even for a large database, such as IMDb, the total elapsed time was, on average, nearly 4 seconds. Hence, to summarize the results, the translation algorithm achieved 69%, 90% and 88% of MAP for Mondial, IMDb and MusicBrainz, respectively.

In the experiments, we also compared the RDF and relational environments, in Oracle. The results obtained were the same, as expected, since the construction process of the abstract query was the same in both cases. The total elapsed time was reasonable, on average, in all experiments. The total elapsed time of the SQL queries was 4-6 times faster than the SPARQL queries, on average. The query build

time was nearly the same in both environments, since processing matches, calculating scores and constructing the abstract query were the same.

Partial results related to this thesis, as well as other important works, were reported in the following articles:

- García, G.M., Izquierdo, Y.T., Menendez, E.S., Dartayre, F., Casanova, M.A RDF Keyword-based Query Technology Meets a Real-World Dataset. 20th EDBT 2017, pp. 656-667.

- Izquierdo, Y.T., Casanova, M.A., García, G.M., Dartayre, F., Levy, C.H. Keyword Search over Federated RDF Datasets. Proc. ER Forum 2017, CEUR Workshop Proceedings, Vol. 1979, CEUR-WS.org

- 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. 29th DEXA 2018. pp. 259-269

- Izquierdo, Y.T., García, G.M., Casanova, M.A., Menendez, E.S., Paes Leme, L.A.P., Neves Jr., A., Lemos, M., Finamore, A.C., Oliveira, C.M. S., Keyword Search over Schema-less RDF Datasets by SPARQL Query Compilation. (submitted for publication)

- Izquierdo, Y.T., García, G.M., Casanova, M.A., Paes Leme, L.A.P., Sardianos, C., Tserpes, K., Varlamis, I., Ruback, L. Stop and Move Sequence Expressions over Semantic Trajectories in RDF. (submitted for publication)

- Neves, A.B, Paes Leme, L.A.P., Izquierdo, Y.T., Garcia, G.M, Casanova, M.A, Menendez, E.S. Computing Benchmarks for RDF Keyword Search. (submitted for publication)

7.2 Future Work

We may suggest a range of possibilities to improve our solution.

The first possibility is to use the measure proposed in (Menendez et al., 2019) to automatically compute the ranking of the elements of the schema. We could also use this measure to rank the results of a query.

The second possible path to follow is to change the step Connect Entity, and to use the paths that better capture the connectivity between a given entity pair, instead of the shortest path. To find these paths, we could use the ideas proposed in (Herrera et al., 2016).

Other more obvious possibilities to improve our keyword search system is to create a mechanism to automatically transform Natural Language sentences or questions into advanced keyword queries and use our algorithm to find the answers, equivalently to what we did in the experiments for Musicbrainz.

We also can use Machine Learning algorithms that take advantage of user's feedback to better tune the ranking of the elements and the score to produce better answers.

8
Bibliography

ADITYA, B. BANKS : Browsing and Keyword Searching in Relational Databases *. **VLDB '02: Proceedings of the 28th International Conference on Very Large Databases**, [s.l.], p. 1083–1086, 2002. DOI: 10.1016/B978-155860869-6/50114-1.

AGRAWAL, S.; CHAUDHURI, S.; DAS, G. DBXplorer: A system for keyword-based search over relational databases. **Proceedings - International Conference on Data Engineering**, [s.l.], p. 5–16, 2002. ISSN: 10844627, DOI: 10.1109/ICDE.2002.994693.

BERGAMASCHI, S. et al. Combining user and database perspective for solving keyword queries over relational databases. **Information Systems**, [s.l.], vol. 55, p. 1–19, 2016. ISSN: 03064379, DOI: 10.1016/j.is.2015.07.005.

CHOPRA, S.; RAO, M. R. The Steiner tree problem I: Formulations, compositions and extension of facets. **Mathematical Programming**, [s.l.], vol. 64, no. 1–3, p. 209–229, 1994. ISSN: 0025-5610, DOI: 10.1007/BF01582573.

COFFMAN, J.; WEAVER, A. C. A framework for evaluating database keyword search strategies. In: **dl.acm.org**. [s.l.]: [s.n.], 2010. retrieved <https://dl.acm.org/citation.cfm?id=1871531>. accessed Jun./11/19. DOI: 10.1145/1871437.1871531.

CORMEN, T. et al. Introduction to algorithms. [s.l.], 2009.

ELBASSUONI, S.; BLANCO, R. Keyword search over RDF graphs. In: **Proceedings of the 20th ACM international conference on Information and knowledge management - CIKM '11**. New York, New York, USA: ACM Press, 2011. retrieved <http://dl.acm.org/citation.cfm?doid=2063576.2063615>. accessed Jun./12/19. ISBN: 9781450307178, DOI: 10.1145/2063576.2063615.

GKIRTZOU, K.; PAPASTEFANATOS, G.; DALAMAGAS, T. RDF Keyword Search based on Keywords-To-SPARQL Translation. In: **Proceedings of the First International Workshop on Novel Web Search Interfaces and Systems - NWSearch '15**. New York, New York, USA: ACM Press, 2015. retrieved <http://dl.acm.org/citation.cfm?doid=2810355.2810357>. accessed Jun./12/19. ISBN: 9781450337892, DOI: 10.1145/2810355.2810357.

HAN, S. et al. 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**. New York, New York, USA: ACM Press, 2017. retrieved <http://dl.acm.org/citation.cfm?doid=3132847.3132957>. accessed Jun./12/19. ISBN: 9781450349185, DOI: 10.1145/3132847.3132957.

HE, H.; YANG, J.; YU, P. S. BLINKS: Ranked Keyword Searches on Graphs. **Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data**. [s.l.]: [s.n.], 2007. 305--316 p. ISBN: 9781595936868, DOI: 10.1145/1247480.1247516.

HERRERA, J. E. T. et al. DBpedia Profiler Tool : Profiling the Connectivity of Entity Pairs in DBpedia. In: **Proc. of the 5th Int'l. Workshop on Intelligent Exploration of Semantic Data (IESD16) collocated with the 15th Int'l. Semantic Web Conf. (ISWC'16)**. [s.l.]: [s.n.], 2016. retrieved <http://www.inf.puc-rio.br/~casanova/Publications/Papers/2016-Papers/2016-IESD-Herrera.pdf>. accessed Feb./12/20.

HIEMSTRA, D. Information Retrieval Models. [s.l.]: [s.n.], 2009. retrieved <https://pdfs.semanticscholar.org/7e43/8364660c2869d21947c4a44c3b448e83d8d0.pdf>. accessed Jul./05/19.

HRISTIDIS, V.; PAPAKONSTANTINOU, Y. Discover: Keyword Search in Relational Databases. **VLDB '02: Proceedings of the 28th International Conference on Very Large Databases**, [s.l.], p. 670–681, 2002. ISBN: 9781558608696, DOI: 10.1016/B978-155860869-6/50065-2.

KUMAR, R.; TOMKINS, A. A characterization of online browsing behavior. In: **pdfs.semanticscholar.org**. [s.l.]: [s.n.], 2010. retrieved <https://pdfs.semanticscholar.org/1170/7110ae2ed0924f1c633459c24d4e18f0e947.pdf#page=5>. accessed Jul./03/19. DOI: 10.1145/1772690.1772748.

LE, W. et al. Scalable keyword search on large RDF data. **IEEE Transactions on Knowledge and Data Engineering**, [s.l.], vol. 26, no. 11, p. 2774–2788, 2014. ISSN: 10414347, DOI: 10.1109/TKDE.2014.2302294.

LIN, X.-Q.; MA, Z.-M.; YAN, L. I. RDF Keyword Search Using a Type-based Summary. **Journal of Information Science & Engineering**, [s.l.], vol. 34, no. 2, p. 489–504, 2018. ISSN: 10162364.

MENENDEZ, E. S. et al. Novel Node Importance Measures to Improve Keyword Search over RDF Graphs. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [s.l.]: Springer, 2019. ISBN: 9783030276171, ISSN: 16113349, DOI: 10.1007/978-3-030-27618-8_11.

OLIVEIRA, P. DE; SILVA, A. DA; MOURA, E. DE. Ranking Candidate Networks of relations to improve keyword search over relational databases. In: **Proceedings - International Conference on Data Engineering**. [s.l.]: [s.n.], 2015. retrieved <https://ieeexplore.ieee.org/abstract/document/7113301/>. accessed Jun./10/19. ISBN: 9781479979639, ISSN: 10844627, DOI: 10.1109/ICDE.2015.7113301.

PARR, T. The Definitive ANTLR 4 Reference. **Climate Change 2013 - The Physical Science Basis**. [s.l.]: [s.n.], 2013. 322 p. ISBN: 9788578110796, ISSN: 1098-6596, DOI: 10.1088/1751-8113/44/8/085201.

QIN, L.; YU, J. X.; CHANG, L. Keyword search in databases. In: **Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09**. New York, New York, USA: ACM Press, 2009. retrieved <http://portal.acm.org/citation.cfm?doid=1559845.1559917>. accessed Jun./11/19. ISBN: 9781605585512, DOI: 10.1145/1559845.1559917.

RIHANY, M.; KEDAD, Z.; LOPES, S. Keyword Search Over RDF Graphs Using WordNet. BT - Proceedings of the 1st International Conference on Big Data and Cyber-Security Intelligence, BDCSIntell 2018, Hadath, Lebanon, December 13-15, 2018. [s.l.]: [s.n.], 2018. retrieved <http://ceur-ws.org/Vol-2343/paper15.pdf>.

SMITH, F. The Matrix-Tree Theorem and Its Applications to Complete and Complete Bipartite Graphs. [s.l.]: [s.n.], 2015. retrieved <http://www.austinmohr.com/15spring4980/paper final draft.pdf>. accessed Aug./01/19.

STOKOE, C.; OAKES, M. P.; TAIT, J. Word Sense Disambiguation in Information Retrieval Revisited. [s.l.]: [s.n.], 2003.

TRAN, T. et al. Top-k Exploration of Query Graph Candidates for Efficient Keyword Search on RDF ∗ Technical Report. **Data Engineering, 2009. ICDE '09. IEEE 25th International Conference**, [s.l.], 2009.

VIRGILIO, R. DE. RDF keyword search query processing via tensor calculus. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [s.l.]: [s.n.], 2012. retrieved <http://link.springer.com/10.1007/978-3-642-33615-7_22>. accessed Jun./11/19. ISBN: 9783642336140, ISSN: 03029743, DOI: 10.1007/978-3-642-33615-7_22.

VIRGILIO, R. DE; MACCIONI, A.; CAPPELLARI, P. A Linear and Monotonic Strategy to Keyword Search over RDF Data. [s.l.]: [s.n.], 2013. p. 338–353. DOI: 10.1007/978-3-642-39200-9_28.

WANG, Y.; WANG, N.; ZHOU, L. Keyword Query Expansion Paradigm Based on Recommendation and Interpretation in Relational Databases. **Scientific Programming**, [s.l.], vol. 2017, p. 1–12, 2017. ISSN: 1058-9244, DOI: 10.1155/2017/7613026.

YANG, M. et al. Finding Patterns in a Knowledge Base using Keywords to Compose Table Answers. **PVLDB**, [s.l.], vol. 7, no. 14, p. 1809–1820, 2014.

ZENZ, G. et al. From keywords to semantic queries-Incremental query construction on the semantic web. **Journal of Web Semantics**, [s.l.], vol. 7, no. 3, p. 166–176, 2009. ISSN: 15708268, DOI: 10.1016/j.websem.2009.07.005.

ZHANG, L.; TRAN, T.; RETTINGER, A. Probabilistic query rewriting for efficient and effective keyword search on graph data. **Proceedings of the VLDB Endowment**, [s.l.], vol. 6, no. 14, p. 1642–1653, 2014. ISSN: 21508097, DOI: 10.14778/2556549.2556550.

ZHENG, W. et al. Semantic SPARQL similarity search over RDF knowledge graphs. **Proceedings of the VLDB Endowment**, [s.l.], vol. 9, no. 11, p. 840–851, 2016. ISSN: 21508097, DOI: 10.14778/2983200.2983201.

ZHOU, Q. et al. SPARK: Adapting keyword query to semantic search. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [s.l.]: [s.n.], 2007. retrieved <https://link.springer.com/chapter/10.1007/978-3-540-76298-0_50>. accessed Jun./10/19. ISBN: 3540762973, ISSN: 03029743, DOI: 10.1007/978-3-540-76298-0_50.

9
Annex

9.1 Tokenize query grammar

```
grammar DankeGrammar_en_US_;

//the initial rule holds one or more sequences of keyword token
(keyword + filter)
start: sequence+ EOF;

//a sequence is a keyword with or without a filter
sequence: keyword filter?;

//A filter could be a simple filter or a logical filter or a range
filter
filter:   rangeFilter | simpleFilter | logicalFilter;

rangeFilter:  RANGE? DATE AND? DATE | RANGE? value AND? value;

logicalFilter: andFilter | orFilter | notFilter;

andFilter: simpleFilter AND? simpleFilter;

orFilter: simpleFilter OR simpleFilter;

simpleFilter: lessFilter | greaterFilter| equalFilter |
notEqualFilter | containsFilter;

equalFilter: EQUAL? (DATE|value) | EQUAL (STRING|ID);

lessFilter: LESS (DATE | value);

greaterFilter: GREATER (DATE|value);

containsFilter: CONTAINS (STRING|ID);

notEqualFilter: DISTINCT (DATE|value) | DISTINCT (STRING|ID);

notFilter: NOT IN? (STRING|ID);

value: FLOAT;

//a keyword is a literal with or without quotes
keyword: (ID | STRING) ;

EQUAL: '=' | [Ee][Qq][Uu][Aa][Ll] ([ ]? [Tt][Oo])?;

CONTAINS: [Cc][Oo][Nn][Tt][Aa][Ii][Nn][Ss];

GREATER: '>' | [Gg][Rr][Ee][Aa][Tt][Ee][Rr] ([ ]?
[Tt][Hh][Aa][Nn])?;
```

```
FLOAT: '-'? DIGIT+ (POINT DIGIT+)?;

DATE: (DIGIT DIGIT? WS* SEPARATOR?  WS*)?  MONTH  WS* SEPARATOR?
WS* DIGIT DIGIT (DIGIT DIGIT)? ;

MONTH: JAN
     | FEV
     | MAR
     | APR
     | MAY
     | JUN
     | JUL
     | AUG
     | SEP
     | OCT
     | NOV
     | DEC
     ;

JAN : [Jj][Aa][Nn][Uu][Aa][Rr][Yy]         | [Jj][Aa][Nn] | '01' |
'1';
FEV : [Ff][Ee][Bb][Rr][Uu][Aa][Rr][Yy]     | [Ff][Ee][Bb] | '02' |
'2';
MAR : [Mm][Aa][Rr][Cc][Hh]                 | [Mm][Ar][Rr] | '03' |
'3';
APR : [Aa][Pp][Rr][Ii][Ll]                 | [Aa][Pp][Rr] | '04' |
'4';
MAY : [Mm][Aa][Yy]                         | [Mm][Aa][Yy] | '05' |
'5';
JUN : [Jj][Uu][Nn][Ee]                     | [Jj][Uu][Nn] | '06' |
'6';
JUL : [Jj][Uu][Ll][Yy]                     | [Jj][Uu][Ll] | '07' |
'7';
AUG : [Aa][Uu][Gg][Uu][Ss][Tt]            | [Aa][Uu][Gg] | '08' |
'8';
SEP : [Ss][Ee][Pp][Tt][Ee][Mm][Bb][Ee][Rr] | [Ss][Ee][Pp] | '09' |
'9';
OCT : [Oo][Cc][Tt][Oo][Bb][Ee][Rr]         | [Oo][Cc][Tt] | '10';
NOV : [Nn][Oo][Vv][Ee][Mm][Bb][Ee][Rr]     | [Nn][Oo][Vv] | '11';
DEC : [Dd][Ee][Cc][Ee][Mm][Bb][Er][Rr]     | [Dd][Ee][Zz] | '12';

SEPARATOR: '/' | '-';

AND: [Aa][Nn][Dd];

OR: [Oo][Rr];

NOT: [Nn][Oo][Tt];

IN: [Ii][Nn];

DISTINCT: '!=' | [Dd][Ii][Ss][Tt][Ii][Nn][Cc][Tt] ([ ]?
[Oo][Ff])?;

LESS: '<' | [Ll][Ee][Ee][Ss] ([ ]? [Tt][Hh][Aa][Nn])?;

RANGE: [Bb][Ee][Tt][We][Ee][Ee][Nn];

POINT: '.' | ',';

SYMBOL: '-' | '_'|'\''|'\"'|'/' | '.';
```

```
ID: (LETTER|DIGIT+ SYMBOL) (LETTER|SYMBOL|DIGIT)*;

STRING: '"' ( ESC_SEQ | ~('\\'|'"') )* '"';

fragment
DIGIT: [0-9];

WS: (' '
   | '\t'
   | '\r'
   | '\n')+ -> skip;

fragment
LETTER: 'A'..'Z'
      | 'a'..'z'
      | [áÁ] | [éÉ] | [íÍ] | [óÓ] |[úÚ] |[Ã³]
      | '\u00C0'..'\u00D6'
      | '\u00D8'..'\u00F6'
      | '\u00F8'..'\u02FF'
      | '\u0370'..'\u037D'
      | '\u037F'..'\u1FFF'
      | '\u200C'..'\u200D'
      | '\u2070'..'\u218F'
      | '\u2C00'..'\u2FEF'
      | '\u3001'..'\uD7FF'
      | '\uF900'..'\uFDCF'
      | '\uFDF0'..'\uFFFD';

fragment
HEX_DIGIT: ('0'..'9'|'a'..'f'|'A'..'F');

fragment
ESC_SEQ: '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
       | UNICODE_ESC
       | OCTAL_ESC;

fragment
OCTAL_ESC: '\\' ('0'..'3') ('0'..'7') ('0'..'7')
         | '\\' ('0'..'7') ('0'..'7')
         | '\\' ('0'..'7');

fragment
UNICODE_ESC: '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
```