Master Thesis

Tabular Information Extraction

Tobias Matysiak

23.10.2019

Albert-Ludwigs-Universität Freiburg im Breisgau
Technische Fakultät

Writing Period
23. 04. 2019 – 23. 10. 2019

Examiner
Prof. Dr. Hannah Bast

Adviser
Niklas Schnelle

Abstract

Searching for content in knowledge bases can involve some challenges, especially for inexperienced users. On the one hand, they need to know the underlying data structure (mostly represented by an ontology), on the other hand, they need to know how to query the knowledge base. This thesis introduces a web application called *Knowledge Base Table Extractor* (*KBTE*) that is able to generate SPARQL queries for Freebase and Wikidata back-ends to retrieve information in a tabular form. For this, the thesis provides a simple tabular description format for specifying table definitions like `city | country | population` as input. KBTE tries to find a *table matching* which maps each column keyword to a class or property of the knowledge base and relates the columns to each other. Unlike keyword search and question answering approaches, it is not focused on matching entities. The algorithm for table matching exploits the combined use of a *search index* and a *column graph*. The search index is used for the mapping part, the column graph is used to find relations between two columns.

Furthermore, the thesis provides three datasets for evaluation. One is created from tables of Wikipedia, the other two contain hand-crafted table definitions. The evaluation compares the table matching algorithm used by KBTE with a simple baseline algorithm. The results can confirm the better performance of the former.

Zusammenfassung

Das Suchen nach Inhalt in Wissensdatenbanken kann einige Hürden mit sich bringen, insbesondere für unerfahrene Benutzer. Auf der einen Seite müssen sie die zugrundeliegende Datenstruktur kennen (welche meist durch eine Ontologie dargestellt wird), auf der anderen Seite müssen wie man Anfragen an die Wissensdatenbank macht. Diese Abschlussarbeit stellt eine Webanwendung namens *Knowledge Base Table Extractor* (*KBTE*) bereit, welche in der Lage ist, SPARQL Anfragen für Freebase und Wikidata Backends für das Abrufen von Informationen in tabellarischer Form zu generieren. Dazu stellt die Arbeit ein einfaches Tabellenbeschreibungsformat zur Verfügung, mit welchem Tabellendefinitionen wie `city | country | population` als Input eingegeben werden können. KBTE versucht ein *Tabellen-Matching* zu finden, welches dem Schlüsselwort jeder Spalte eine Klasse oder Eigenschaft aus der Wissensdatenbank zuweist und die Spalten zueinander in Beziehung setzt. Im Gegensatz zu Keyword Search und Question Answering Ansätzen ist KBTE nicht auf das Matchen von Entitäten fokussiert. Der Tabellen-Matching-Algorithmus nutzt den kombinierten Gebrauch eines *Suchindex* und eines *Spaltengraphs* aus. Der Suchindex wird für das Zuweisungsproblem benutzt, der Spaltengraph wird benutzt, um Beziehungen zwischen zwei Spalten zu finden.

Außerdem stellt die Arbeit drei Datensätze für die Evaluation bereit. Der eine ist aus Wikipedia-Tabellen erstellt, die anderen beiden enthalten manuell erstellte Tabellendefinitionen. Die Evaluation vergleicht den Tabellen-Matching-Algorithmus, welcher von KBTE benutzt wird, mit einem einfachen Ausgangsalgorithmus. Die Ergebnisse bestätigen die bessere Leistung des ersteren.

Contents

1 Introduction

In times of increasing amounts of information, it becomes more and more necessary to keep this information in a structured form. This structure may be a knowledge base specifying an ontology as its structure. An ontology is a model that represents data by a set of concepts within a domain and the relationships among those concepts. There are many ontological knowledge bases, e.g. the Freebase, Wikidata or DBPedia. Often it is not easy for non-expert users to extract the desired information from a knowledge base. It is necessary to know both the underlying ontology, i.e. the classes and properties, and how to query the data using the standard query language SPARQL. Many papers try to tackle this problem by providing query builders [1, 2], question answering [3, 4] or keyword search [5, 6] approaches which simplify access to the data.

This thesis focuses on the extraction of information in tabular form. It presents the web application *KBTE* (Knowledge Base Table Extractor) which, given a structured table definition in a tabular description format, generates a SPARQL query for different knowledge base back-ends. KBTE supports the two collaborative knowledge bases Freebase[1] which was already shut down in 2016 and Wikidata[2], a knowledge base of the Wikimedia Foundation. The SPARQL query then yields the content for this table from the respective knowledge base back-end. Assuming, someone is interested in a table containing global cities together with their countries and populations, the table definition may look as follows:

```
city | country | population
```

Listing 1.1 shows a SPARQL query for Wikidata to retrieve the defined table. A non-expert user would encounter the problem of finding the correct classes and properties for each column in order to formulate such a query. For Wikidata, the intended classes and properties for the particular columns would be the item *Q515* for the column *city* and the properties *P17* and *P1082* for the columns *country* and *population*. The columns *country* and *population* should be related to the column *city*, because they are properties of instances of the class *Q515*. In another formulation of the table definition, it would also be possible to relate the column *population* to the column *country*.

Listing 1.2 shows a SPARQL query that leads to the same defined table for Freebase. In this case, the column *country* is not represented by a property but by the Freebase

[1]`https://developers.google.com/freebase/`
[2]`https://www.wikidata.org`

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?city_name ?country_name ?population WHERE {
  ?city wdt:P31 wd:Q515 .
  ?city wdt:P17 ?country .
  ?city wdt:P1082 ?population .
  ?city rdfs:label ?city_name .
  ?country rdfs:label ?country_name .
  FILTER (lang(?city_name) = "en") .
  FILTER (lang(?country_name) = "en") .
}
```

Listing 1.1: SPARQL query for Wikidata

```
PREFIX fb: <http://rdf.freebase.com/ns/>
SELECT DISTINCT ?city_name ?country_name ?population WHERE {
  ?city fb:type.object.type fb:location.citytown .
  ?city fb:type.object.name ?city_name .
  ?country fb:type.object.type fb:location.country .
  ?country fb:type.object.name ?country_name .
  ?city fb:location.location.containedby ?country .
  ?city fb:location.statistical_region.population ?m_population .
  ?m_population fb:measurement_unit.dated_integer.number ?population .
}
```

Listing 1.2: SPARQL query for Freebase

type *location.country*. It is then related to the column *city* by using the property *location.location.containedby* in an additional triple.

1.1 Definitions of Terms

This thesis uses terms that can vary depending on the knowledge base.

KBTE is able to use either Freebase or Wikidata as knowledge base. To make KBTE work with different knowledge bases, this thesis focuses on the similarities of them and tries to generalize differences. This section states important definitions that are used throughout the thesis.

KBTE uses the concepts of classes and properties to describe columns of a table. In the resulting SPARQL query a column can be either represented by a class or by a property of the knowledge base. The content of the single column cells are instances of a class or values of a property, depending on the respective column.

Knowledge base data is often stored as RDF dump. The Resource Description Framework[3] (RDF) is a specification of the World Wide Web Consortium (W3C) that is used to represent data in triples. A triple (or n-tuple with $n = 3$) is a line of data with the structure `subject predicate object .`. In a triple, a *subject* is linked through a *predicate* to an *object*. In this way, facts can be expressed in a natural language sentence: Berlin (*subject*) is located in (*predicate*) Germany (*object*).

Table 1.1 gives an overview of Freebase-specific and Wikidata-specific terms for the general concepts of *entities*, *classes* and *properties*.

The terms used for Freebase, are described in [7]. In Freebase, entities are called objects. Each object has a unique machine ID called *mid*. For example, the mid of the city Berlin is *m.0156q*. The concept of a class is expressed by *Freebase types*. A *Freebase object* can have multiple types. In order to form facts, *Freebase properties* are used to link objects with to objects or values. To represent n-ary relations with $n > 2$, Freebase uses Compound Value Types (CVTs). Within the scope of this thesis, they are called *mediator classes* or *mediators*. For example, the mediator *location.geocode* is used to link geographic coordinates to a location. by the property *location.location.geolocation*.

In Wikidata, as described in [8], entities are represented as *Wikidata items*. Each item has a unique identifier called *qid*. For example, the qid of the city Berlin is *Q64*. An item A can be an instance of another item B, turning item B into a class and item A into an instance. The data model of Wikidata consists of triples in the form `item property value` representing a (claimed) fact or claim. Together with references, a claim forms a statement.

[3]`https://www.w3.org/TR/rdf-concepts/`

Concept	**Freebase**		**Wikidata**	
Instance or *Entity*	Freebase Object (used as subject with the property *type.object.type*)	*m.0156q* ("Berlin")	Wikidata Item (used as subject with the property *P31*)	*Q64* ("Berlin")
Class	Freebase Type (used as object with the property *type.object.type*)	*location.citytown* ("City/Town/Village")	Wikidata Item (used as object with the property *P31*)	*Q515* ("city")
Property	Freebase Property	*location.statistical_region.population* ("Population")	Wikidata Property	*P1082* ("population")

Table 1.1: Terms used by Freebase and Wikidata with examples

1.2 Problem Definition

An input table is represented by a structured table definition specified in a tabular description format.

The problem of translating a table definition with a set of columns into a SPARQL query can be split up into the following sub problems:

Column matching Each column definition has to be matched to a class or property of the knowledge base model. Column matching again is a sub task of *table matching*.

Relations between columns For each column, it has to be determined to which other column it should be related. There is just one column, the so called *master column*, which should not be related to another column in order to relate most (or all) other columns to this master column. A further part of this sub problem is to find the most appropriate master column.

Table matching Table matching combines the problems of column matching and finding relations between columns. The aim is to find a mapping where each column is mapped to a tuple. The first value of the tuple contains the column to which the current column relates (relations between columns). The second value is the best matching class or property of the knowledge base for the column's definition (column matching). For the introductory example, a possible table matching for Wikidata of the table definition `city | country | population` would be

```
city -> (None, Q515),
country -> (city, P17),
population -> (city, P1082)
```

Here, the column *city* is used as the master column and the columns *country* and *population* relate to *city*.

Figure 1.1 illustrates the matching of the exemplary table definition to Wikidata classes and properties.

Query generation Generation of the SPARQL query from the table matching represents the last sub problem of the table translation. According to the table matching, we have to find the connecting relations between the matched pairs of columns.

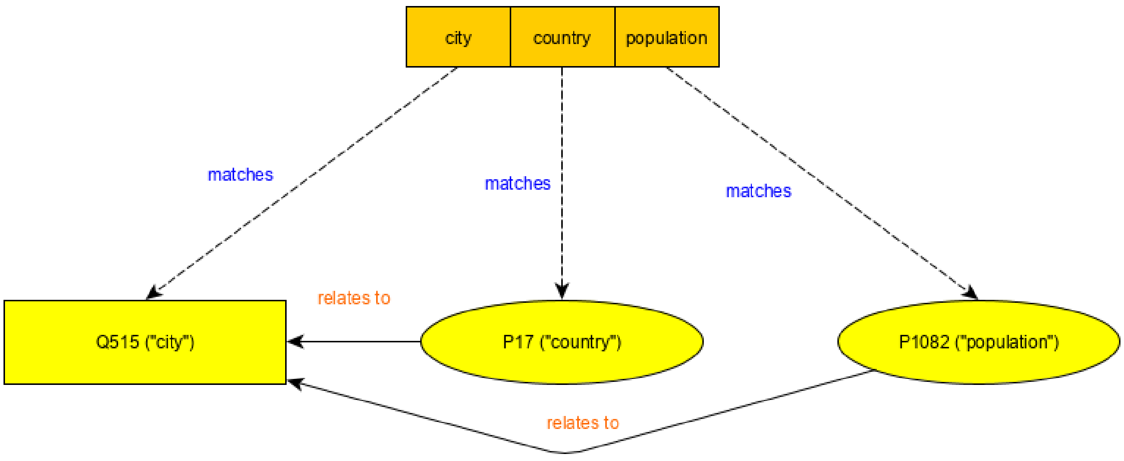

Figure 1.1: Table matching example for Wikidata

1.3 Overview of the tool

The back-end of KBTE consists of three key parts:

- A module for reading the input table definition. For this purpose the thesis specifies a **tabular description format**. In addition to defining table columns, it allows column-wise setting of a filter, an order and an explicit linking to another column.
- A **column graph** whose nodes represent the objects of the knowledge base that can be used as table columns (classes and properties). The edges of the graph represent the matchings between columns according to certain *templates*.

The column graph is used to find possible interpretations of how the defined columns are related to each other.

- A **search index** consisting of a document-term matrix with tf-idf features. For this, the documents correspond to the classes and properties of a knowledge base and the terms are created out of the trigrams of their respective names.

For the translation of a table definition, the column graph and the search index are used in combination in order to find a table matching. The implementation of KBTE is described in more detail in chapter 3.

In chapter 4, the performance of the table matching algorithm of KBTE is analyzed on different datasets comparing to a baseline algorithm. The first dataset contains table definitions which have been created from tables of Wikipedia. Another two datasets, one for Freebase, one for Wikidata, contain a small number of hand-crafted table definitions together with their intended classes and properties. They are used for a more qualitative comparison between the algorithms.

2 Related Work

This thesis is related to the field of semantic search on knowledge bases. This data searching technique does not only look for literal matches of the input words, but it also determines the intent or contextual meaning of the input. Referring to [9], textual semantic search can be divided into searching on text in natural language, searching on structured data such as knowledge bases or searching on a combination of text and structured data. These data types can be searched by keyword search, structured search or natural language search (e.g. question answering).

In keyword search on knowledge bases, the aim is to generate a SPARQL query or matching items ordered by relevance given a sequence of keywords as input. Considering the graph structure of the knowledge base, these keywords have to be matched to nodes of the graph (i.e. entities) first. It is then tried to find a minimum connected tree or subgraph that covers all matched nodes. This problem is similar to the Group Steiner Tree problem (GST), which is NP-complete.

Approaches for keyword search on knowledge bases either focus on solutions to this problem, others present query builders such as FreeQ [1] or GraFa [5]. FreeQ is an interactive query interface for incremental query construction on Freebase and GraFa is a faceted browsing interface for Wikidata. The authors of [6] present a keyword search approach which is focused on exploiting parallelism. It does not use the GST model but introduces a concept called Central Graph. It further presents a real-time search engine for Wikidata called WikiSearch.

Question answering on knowledge bases also aims for the generation of a SPARQL query. The basic approach for question answering again involves matching parts of the question to entities. In addition, it is also tried to recognize relations of the knowledge base in parts of the question that connect two entities (relation extraction). Recognizing relations is a harder problem than recognizing entities because relations can have different formulations in natural language.

In [10], the process of question answering is divided into the following tasks:

- question analysis: a syntactic analysis of the question using techniques like part-of-speech-tagging (POS-tagging) and named entity recognition (NER)
- phrase mapping: identification of resources that correspond to phrases of the question
- disambiguation: determination which of the identified resources from phrase mapping are the right ones

- query construction: construction of a SPARQL query
- querying distributed knowledge: retrieving information from several knowledge bases

There exist numerous approaches for question answering on knowledge bases like Freebase and Wikidata. Aqqu [3], an approach for Freebase, creates query candidates by matching the input question to certain templates. The candidates are then ranked according to their similarity with the question as determined by several interwoven machine learning models trained using manually generated question answer pairs. Platypus [4], a multilingual system for Wikidata, uses a grammatical analyzer and a template-based analyzer to parse questions. The analyzers convert the question into logical representations. These representations are ranked by their likelihood.

The approach presented in this thesis has certain similarities to keyword search and question answering approaches. It also tries to match parts of the input to objects of a knowledge base and tries find connecting relations. But unlike keyword search and question answering, the approach of this thesis matches columns of the input to classes and properties only and not to entities. Furthermore, it does not use a sophisticated ranking method.

3 Implementation

This chapter describes the implementation details of the tool *KBTE*. KBTE takes a user defined table definition as input, that is in a specific *tabular description format*. In order to generate a SPARQL query out of this table definition, the tool has to find a linking class or property for each defined column. This is implemented as using the *search index* of KBTE. Additionally, the columns have to be matched with each other in such a way that each column, except one *master* column, is related to another column. The relatedness between classes or properties of a knowledge base is defined by various templates and implemented as a so called *column graph*.

3.1 Tabular Description Format

```
tabledef   = columndef (separator columndef)*
columndef  = string [filter] [order] [link]
filter     = "(" comparator (string | number) ")"
order      = "[" ("asc" | "desc") ("," number) "]"
link       = "->" number

separator  = "|"
string     = CHAR+
number     = DIGIT+
comparator = "!=" | "<=" | "<" | ">=" | ">" | "="
```

Listing 3.1: Specification of the tabular description format in standard EBNF notation

The specified tabular description format allows input of the table definition in a structured way. A structured table input is advantageous for KBTE, because it contains certain constraints which simplify the generation of the SPARQL query. The most important constraint is that columns correspond to a class or property of the knowledge base ontology. Listing 3.1 shows the specification of the format. A valid table definition consists of at least one table column. Columns are separated by the vertical bar character. To each column a filter, an order and a link can optionally be added. A filter is written in round brackets, an order in square brackets. A link can be used to force the tool to relate the column to a specific column, which simplifies the matching progress and corrects or prevents an unwanted interpretation of the input. In this way, a user of KBTE could utilize links to disambiguate a table definition. For example, in the table definition `city | country | population -> 0` the *population* column is explicitly linked to the first column (*city*, column with index 0). The alternative table definition `city | country | population -> 1` would relate *population* to *country*.

The following table definition depicts a more detailed variant of the example in chapter 1:

```
city [asc, 2] | country (="Germany"@en)| population (>= 100000)[desc,
 1] -> 0
```

The table is constrained to cities of the *country* Germany with a *population* of at least 100000 inhabitants. In addition, the table is arranged by *population* in descending order first (optional second parameter in square brackets, lower numbers are prioritized), and by *city* in ascending second. An explicit link is also used here.

3.2 Search index

In order to fill a table with information of a knowledge base, it is necessary to know how to retrieve the content of single table columns. In the simplest case, this is achieved by using classes and properties of the knowledge base. For KBTE, information about the classes and properties has been collected in a data structure that allows quick access given a keyword. For example, assuming an index for Wikidata, the keyword *city* should result in a list of classes and properties containing the class key *Q515* (“city”) as first element. Alternatives with a lower popularity would be the classes *Q1549591* (“big city”) or *Q1093829* (“city of the United States”). So the index ensures, that a non-expert user doesn’t need to know the exact knowledge base object. It has to be mentioned that the current implementation does not take the class hierarchy of Wikidata into account yet. At the moment, only direct instances with the property *P31* (“instance of”) are considered as class. Subclasses with the property *P279* (“subclass of”) are not considered which leads to wrong popularity counts. For example, the class *Q1093829* (“city of the United States”) has more direct instances than *Q515* (“city”) which can be problematic for particular column definitions.

In order to create the index, the collection of all classes and properties is converted to a document-term matrix of tf-idf features using the Python library *scikit-learn*[1]. In this matrix, the classes and properties correspond to the documents. The matrix describes the frequency of terms that occur in these documents, where the terms are the single trigrams created from the particular labels of classes and properties. An n-gram is a slice of length n of a text sample and trigrams are the special case for $n = 3$. For example, the set of all trigrams of the word *mountain* would be

[1]`https://scikit-learn.org/stable/index.html`

`[mou, oun, unt, nta, tai, ain]`. Figure 3.1 shows an example table containing the term frequencies for three example Wikidata items used as documents. The document set contains the three Wikidata objects with the labels *mountain*, *mountain range* and *mountain pass*. Figure 3.2 illustrates the document-term matrix A with their L2-normalized tf-idf scores resulting from that table. The scores are calculated by the formula $\mathit{tf\text{-}idf}(t,d) = \mathit{tf}(t,d) \cdot \mathit{idf}(t)$, where the term frequency $\mathit{tf}(t,d)$ is the number of occurrences of a term t in a document d, n is the total number of documents in the document set and $\mathit{idf}(d,t) = \ln(\frac{1+n}{1+\mathit{df}(d,t)}) + 1$. The document frequency $\mathit{df}(d,t)$ is the number of documents in the document set that contain the term t.

Together with the number of triples used as counts of the classes and properties, the document-term matrix is used to find the most popular exact knowledge base object for a column name of the table definition. The search index calculates a relevance score for each document to a given input string (e.g. the column name). A column name can be represented as a vector q by the terms of the search index. The relevance score then is the result of the dot product $q \cdot A^\top$, where A is the document term matrix. Since A is stored as a sparse matrix, the calculation is efficient even with a lot of documents. In order to yield the final popularity score, the relevance score of each class and property is multiplied by its respective count (number of triples).

Assuming a table definition `mountain | range` which should result in a table of mountains and the mountain range they're belonging to, the column names *mountain* and *range* have to be linked to a document contained in the search index. Figure 3.2 gives an example for the calculation of the relevance score for the input "range". It shows the document-term matrix A with L2-normalized tf-idf scores, the query vector q_{range} (also L2-normalized) and their dot product. The dot product indicates that only the second document *P4552* ("mountain range") is relevant for the column *range* with a score of 0.63. All values are rounded off to 2 digits.

	Terms (tri-grams)															
Documents (classes and properties)	␣pa	␣ra	ain	ang	ass	in␣	mou	n␣p	n␣r	nge	nta	oun	pas	ran	tai	unt
Q8502 (“mountain”)			1				1				1	1			1	1
P4552 (“mountain range”)		1	1	1		1	1		1	1	1	1		1	1	1
Q133056 (“mountain pass”)	1		1		1	1	1	1			1	1	1		1	1

Figure 3.1: Term frequencies of three exemplary Wikidata documents

$$A = \begin{pmatrix} 0 & 0 & 0.41 & 0 & 0 & 0 & 0.41 & 0 & 0 & 0 & 0.41 & 0.41 & 0 & 0 & 0.41 & 0.41 \\ 0 & 0.36 & 0.21 & 0.36 & 0 & 0.28 & 0.21 & 0 & 0.36 & 0.36 & 0.21 & 0.21 & 0 & 0.36 & 0.21 & 0.21 \\ 0.39 & 0 & 0.23 & 0 & 0.39 & 0.29 & 0.23 & 0.39 & 0 & 0 & 0.23 & 0.23 & 0.39 & 0 & 0.23 & 0.23 \end{pmatrix}$$

$$q_{range}^{\top} = \begin{pmatrix} 0 & 0 & 0 & 0.58 & 0 & 0 & 0 & 0 & 0 & 0.58 & 0 & 0 & 0 & 0.58 & 0 & 0 \end{pmatrix}$$

$$q_{range} \cdot A^{\top} = \begin{pmatrix} 0 & 0.63 & 0 \end{pmatrix}$$

Figure 3.2: Example calculation of the relevance scores for the input “range”

3.3 Column graph

The column graph is another module which is used for finding the best matching knowledge base classes and properties for a given table definition. The nodes of the graph are the classes and properties and the edges are added according to certain templates. The edges indicate, whether two exact column definitions can be matched with each other and contain the count for this match. The Class-Property template and the Class-Class template are the two basic templates. For Freebase, it is necessary to introduce additional templates which can handle connections via mediator classes. Figure 3.3 depicts an exemplary sub graph containing some Wikidata classes and properties that are connected according to the basic templates. For example, the edge between the class *Q515* and the property *P17* has a count of 8552.

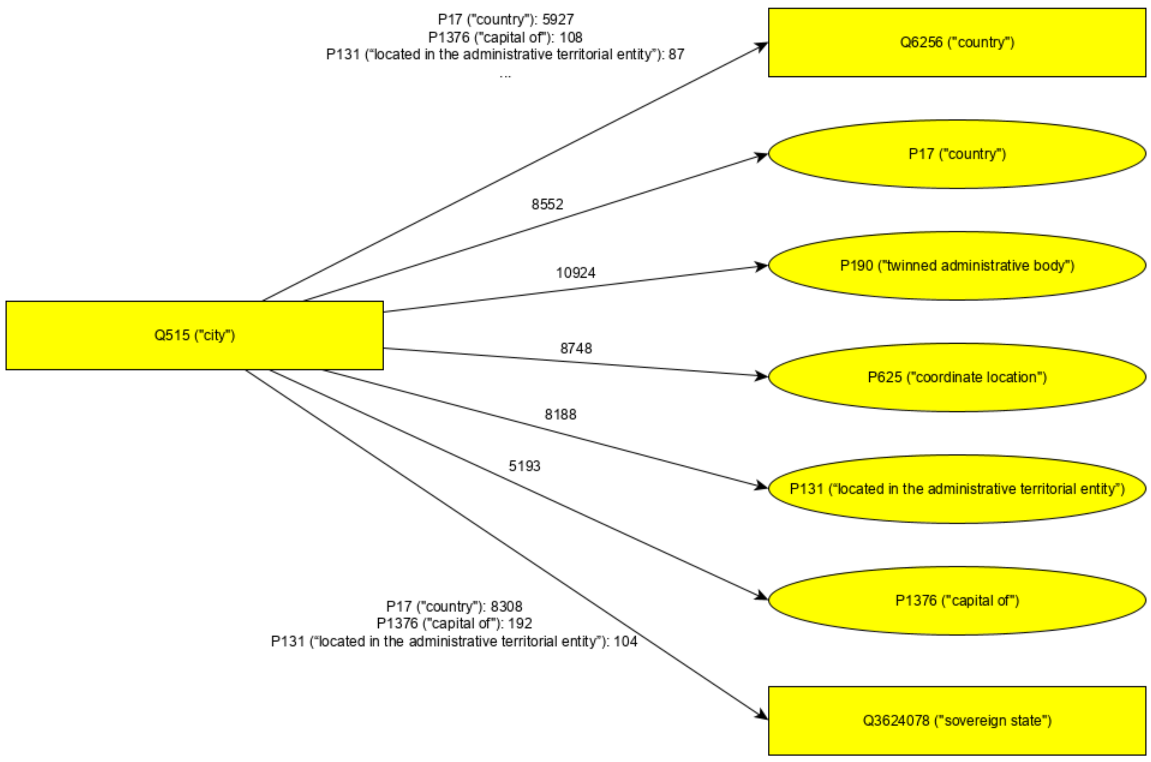
P17 ("country"): 5927
P1376 ("capital of"): 108
P131 ("located in the administrative territorial entity"): 87
...
Q6256 ("country")
P17 ("country")
8552
P190 ("twinned administrative body")
10924
Q515 ("city")
8748
P625 ("coordinate location")
8188
P131 ("located in the administrative territorial entity")
5193
P1376 ("capital of")
P17 ("country"): 8308
P1376 ("capital of"): 192
P131 ("located in the administrative territorial entity"): 104
Q3624078 ("sovereign state")

Figure 3.3: A sub graph of the column graph for Wikidata demonstrating exemplary relations between classes and properties

3.3.1 Basic templates

Class-Property template (CP)

```
SELECT ?o WHERE {
  ?s <is-a> [CLASS] .
  ?s [PROPERTY] ?o .
}
```

Listing 3.2: Class-Property template

Listing 3.2 contains the SPARQL template for the Class-Property template (CP) where the predicate `<is-a>` should be replaced by the knowledge base specific property for class instances. For Wikidata this would be the property *P31* (“instance of”), for Freebase it would be *type.object.type* (“Type”).

The CP template indicates if a property can be applied on a class. This will be the case if the corresponding SPARQL query delivers at least one result (the number of results corresponds to the count). For example, the template would imply an edge in the column graph between the Wikidata class *Q515* (“city”) and the property *P17* (“country”) because a city is part of a country. In contrast, the property *P50* (“author”) is not related to *Q515*, because a city has no author.

Class-Class template (CC)

```
SELECT ?p WHERE {
  ?s <is-a> [CLASS1] .
  ?o <is-a> [CLASS2] .
  ?s ?p ?o .
}
```

Listing 3.3: Class-Class template

The other basic template is the Class-Class template (CC). Compared to the CP template, it is not only a quantitative template because there is the additional interest in finding the property with the highest count for the variable *?p.* This property connects the two classes. For example, the top three connecting properties for the Wikidata classes *Q515* (“city”) and *Q6256* (“country”) would be *P17* (“country”) first, *P1376* (“capital of”) second and *P131* (“located in the administrative territorial entity”) third (compare with the first edge in Figure 3.3).

3.3.2 Extraction of the column graph data

For KBTE, the column graph data is extracted directly from the RDF dumps of Wikidata and Freebase. In these dumps statements are stored in triples of the form `<subject> <predicate> <object> .`. It is necessary to iterate twice over the full dump. In the first run, a large dictionary is built which maps each single entity of the knowledge base to a list of classes. For Freebase entities, the relevant triples are `<Entity> <http://rdf.freebase.com/ns/type.object.type > <Class> .` which have the property *type.object.type* as predicate. For Wikidata entities, the triples which have the property *P31* as predicate are in the form `< Entity> <http://www.wikidata.org/prop/direct/P31> <Class> .`. After full initialization, the entity dictionary occupies a large amount of RAM, around 45 GB for Freebase.

In the second run, the entity dictionary is used to create the column graph edges that match the basic templates CC and CP.

CC template

For all triples in the form `<Entity1> <Property> <Entity2> .`, the lists of all classes of *Entity1* and *Entity2* are looked up in the dictionary and an edge from every single class of *Entity1* to every single class of *Entity2* is created with the property that is used as predicate. If an edge already exists, the corresponding count number is incremented by 1.

CP template

For all triples in the form `<Entity> <Property> <Value or other Non-entity> .`, an edge is created from every single class of *Entity* to the property used as predicate. The edge counts are determined in the same way as in CC template case.

Class and property counts

Using the number of triples as count for edges is also applied on the nodes. These node counts are used for calculating the popularity scores while searching the search index described in section 3.2. For classes, two counts are created. Every triple that has an entity as subject increments the *subject count* of all entity classes by 1. Every triple that has an entity as object increments the *object count* of all entity classes by 1. The count of a property is incremented every time the property is used as predicate.

3.3.3 Freebase-specific templates

For Freebase a special case has to be handled because of mediator classes (or Compound Value Types). For this, two additional cases have to be considered.

Class-Mediator-Property template (CMP)

In the Class-Mediator-Property template (Listing 3.4), the property is used on the mediator class which can be derived from the ID of the property. For example, the mediator class of the property of *location.geocode.latitude* is *location.geocode*. It has to be found a property which connects the given class on the left side with the mediator class on the right side.

```
PREFIX fb: <http://rdf.freebase.com/ns/>
SELECT ?p WHERE {
  ?s fb:type.object.type [CLASS] .
  ?s ?p ?mediator .
  ?mediator [PROPERTY] ?o .
}
```

Listing 3.4: Class-Mediator-Property template

Class-Property-Mediator template (CPM)

In the Class-Mediator-Property template (Listing 3.5), the property is used to connect the given class to a mediator class. In this case, the most popular property of the mediator class has to be found. This can be ambiguous because often multiple mediator properties have the same count. For example, if a column is specified by the property *location.location.geolocation*, the solving property can either be *location.geocode.latitude* or *location.geocode.longitude*.

```
PREFIX fb: <http://rdf.freebase.com/ns/>
SELECT ?p WHERE {
  ?s fb:type.object.type [CLASS] .
  ?s [PROPERTY] ?mediator .
  ?mediator ?p ?o .
}
```

Listing 3.5: Class-Property-Mediator template

3.4 Table matching algorithm

This section describes the algorithms which are used to find a solution for the table matching problem as described in section 1.2. The main algorithm MatchTable (see Algorithm 3.4) determines the mapping.

The basic idea for table matching is the pairwise matching of single column pairs. The single column pair matching is done by the algorithm MatchPair. MatchPair exploits the combined use of the search index (section 3.2) and the column graph (section 3.3) in order to find the best matching classes and properties for a column pair. It is used by the main algorithm. For the main table matching algorithm this section presents two versions. One is the baseline algorithm MatchTableBaseline and the other is the improved version MatchTable. MatchTable is used for KBTE as default algorithm.

All three algorithms are described in the following.

3.4.1 Single column pair matching

Algorithm 3.1 shows the algorithm MatchPair which tries to match a single column pair and outputs a ranked list of the best matching classes and properties for the two columns. The technical details of the function SearchIndex that is called in the lines 5 and 11, has been described in section 3.2. It takes a string as main parameter and returns classes and properties ranked by a popularity score. An optional second parameter which takes a class as input restricts the search results to be neighbors of this class in the column graph. Another function ColumnGraphEdgeCount which is called in line 14 only returns the count of the best template for an edge between two classes or between a class and a property in the column graph. In this way, it ensures that the pair matching results actually do yield results when they are put in a SPARQL query with the respective basic template (CC or CP). For example, in order to match the fuzzy column pair `(city, country)`, one possible result of MatchPair for Freebase would be `(location.citytown, location.country)` because it fulfills both conditions necessary to be a pair match. The left column definition *city* matches the label “City/Town/Village” of the class *location.citytown* and the right column definition *country* matches the label “Country” of the class *location.country*, which satisfies the first condition. The second condition is satisfied as well, because *location.citytown* has an edge to *location.country* in the column graph. For instance, they can be connected with the CC template using *location.location.containedby* as connecting property. In contrast to this valid pair match, the pair `(location.citytown, film.film.country)` is not a valid result. While the first condition is satisfied, the second is not because the class *location.citytown* and the property *film.film.country* can not be applied on the CP template.

3.4.2 Baseline algorithm for table matching

The baseline algorithm MatchTableBaseline (Algorithm 3.2) describes a basic approach to the table matching problem. Given a table definition with n columns, the algorithm tries to find a valid table matching. For each column definition, it simply chooses the best matching class or property from SearchIndex (line 3) and stores it in the mapping *CM*. After that, all possible column pairs $(C_i, C_j), 0 \leq i < n, 0 \leq j < n, i \neq j$ are tested for matches with the function MatchPair using

Algorithm 3.1 Column pair matching algorithm

```
1: function MatchPair(column definitions leftcol, rightcol)
2:     if $leftcol$ is a KB class then
3:         left column candidates $LCC \leftarrow leftcol$
4:     else
5:         $LCC \leftarrow$ all results of SearchIndex($leftcol$)
6:     end if
7:     for each $leftcandidate$ in $LCC$ do
8:         if $rightcol$ is a KB class or property then
9:             right column candidates $RCC \leftarrow rightcol$
10:        else
11:            $RCC \leftarrow$ all results of SearchIndex($rightcol, leftcandidate$)
12:        end if
13:        for each $rightcandidate$ in $RCC$ do
14:            $count \leftarrow$ ColumnGraphEdgeCount($leftcandidate, rightcandidate$)
15:            if $count > 0$ then
16:                $result \leftarrow result + ((leftcandidate, rightcandidate), count)$
17:            end if
18:        end for
19:    end for
20:    return $result$ sorted by count
21: end function
```

the best match for both columns as input. The pair match results are then stored in a nested dictionary *PMR* with C_j as first key, and C_i as second key in order to preserve the mapping from column pairs to the results (line 9). For each column C_j, the best left-side column C_i has to be found. It is the column which, from all pair match results that use C_j on the right side, leads to the highest count. The resulting mapping of best pair matches *BPM* (line 13) indicates to which other column C_i a column C_j should be related. The master column is the one to which the most other columns relate (line 16). In case of a tie, the column with the lower index in the input table definition is taken. To get the final table matching, the values of the mapping *CM* and *BPM* are combined in a tuple for each column (line 21). For the master column the first value of the tuple is None, because it does not relate to any other column (line 19).

Algorithm 3.2 Baseline algorithm for table matching

```
1:  function MatchTableBaseline(column definitions $C$ with $n$ columns)
2:      for each column $C_i$ in $C$ do
3:          column match $CM[C_i] \leftarrow$ best match of SearchIndex($C_i$)
4:          if no matches then
5:              return no solution
6:          end if
7:      end for
8:      for each column pair $(C_i, C_j)$ with $0 \leq i < n, 0 \leq j < n, i \neq j$ do
9:          pair match results $PMR[C_j][C_i] \leftarrow$ MatchPair($CM[C_i], CM[C_j]$)
10:     end for
11:     for each $C_j$ in $C$ do
12:         find $C_{best}$ where count of $PMR[C_j][C_{best}]$ is maximized
13:         best pair match $BPM[C_j] \leftarrow (C_{best}, PMR[C_j][C_{best}])$
14:         column frequency $CF[C_{best}] += 1$
15:     end for
16:     master column $mc \leftarrow C_{max}$ where $CF[C_{max}]$ is maximized
17:     for each column $C_i$ in $C$ do
18:         if $C_i = mc$ then
19:             $tablematching[C_i] \leftarrow (None, CM[C_i])$
20:         else
21:             $tablematching[C_i] \leftarrow (BPM[C_i][0], CM[C_i])$
22:         end if
23:     end for
24:     return $tablematching$
25: end function
```

3.4.3 Improved table matching algorithm for usage in KBTE

It is expected that the baseline algorithm MatchTableBaseline often is not able to find related columns for a specific column. The reason for this is that each column is already matched to a class or property in the first step. To tackle this problem, the column matching in the algorithm MatchTable (see Algorithm 3.4) takes a potential relation between matched column classes and properties into account.

In the trivial case (lines 2 to 6) where the number of columns is $n = 1$, only an index search is performed for the single column. The best matching knowledge base class is chosen as a result.

In the other case, where $n > 1$, the dictionary *PMR* containing all pair match results is created (line 9) as in the baseline algorithm. The important difference is that the call of MatchPair now takes the columns as input and not their best matches. The function RankedMasterCandidates iterates through *PMR* and returns an ordered list *MACOL* of the table columns (line 11). The order is used for trying each column as *master column*. It is counted how frequently a column is used as second key C_i, meaning for how many other columns it could stand on the left side of a column pair. The columns are then sorted by the highest count, in case of tie, the one with the lower index in the table definition is prioritized.

The function RankedClassesForMasterCandidate also iterates through *PMR* and returns a dictionary *MACLA* which contains a ranking of the best matching classes for each column (line 12).

In the next step, it is tested for the first master column candidate *master* (line 13) together with its first matched class *mc* (line 15) whether all other columns of the table definition have a pair match with *mc*. To find the best matching class or property for a column based on the current values for *master* and *mc*, the function FindRightColType is used (see Algorithm 3.3). The return value is added to the column's match for the final table matching (line 21). If a column does not have pair matches with *master*, it will be added to a list of skipped columns *SC* (line 23). For each skipped column *skcol* it is tried to find the best pair match with another column that is already contained in the final table matching (lines 26 to 35). If it is impossible to find a match for at least one skipped column, the table matching so far will be discarded. The algorithm then continues with the next possible class *mc* for *master*. On further failures for all other possible classes, the algorithm continues with the next master column candidate until a final table matching is found or until the search is exhausted and stops without solution.

Algorithm 3.3 Function for finding a class or property for the right column C_j

```
1: function FindRightColType($C_i$, $C_j$, $class_i$, pair match results $PMR$)
2:     for each $pairmatch$ in $PMR[C_j][C_i]$ do
3:         if $pairmatch[0] = class_i$ then
4:             return pairmatch[1]
5:         end if
6:     end for
7:     return None
8: end function
```

Algorithm 3.4 Improved algorithm for table matching

```
function MatchTable(column definitions $C$ with $n$ columns)
    if $n = 1$ then
        $bestClasses \leftarrow$ SearchIndex($C_0$)
        $tablematching[C_0] \leftarrow (None, bestClasses_0)$
        return $tablematching$
    end if
    if $n > 1$ then
        for each column pair $(C_i, C_j)$ with $0 \leq i < n, 0 \leq j < n, i \neq j$ do
            pair match results $PMR[C_j][C_i] \leftarrow$ MatchPair($C_i, C_j$)
        end for
        $MACOL \leftarrow$ RankedMasterCandidates($PMR$)
        $MACLA \leftarrow$ RankedClassesForMasterCandidate($PMR$)
        for each master column candidate $master$ in $MACOL$ do
            master column classes $MC \leftarrow MACLA[master]$
            for each class $mc$ of $MC$ do
                clear $tablematching$
                $tablematching[master] \leftarrow (None, mc)$
                for each $C_i$ in $C$, $C_i \neq master$ do
                    $col2type \leftarrow$ FindRightColType($master, C_i, mc, PMR$)
                    if $col2type$ exists then
                        $tablematching[C_i] \leftarrow (master, col2type)$
                    else
                        add $C_i$ to skipped columns $SC$
                    end if
                end for
                for each $skcol$ in $SC$ do
                    for each $col1$ with $col1class$ in $tablematching$ do
                        $col2type \leftarrow$ FindRightColType($col1, skcol, col1class, PMR$)
                        if $col2type$ exists then
                            $tablematching[skcol] \leftarrow (col1, col2type)$
                        else
                            break
                        end if
                    end for
                end for
                if $tablematching[C_i]$ exists $\forall C_i \in C$ then
                    return $tablematching$
                end if
            end for
        end for
    end if
    return no solution
end function
```

3.4.4 Complexity analysis

Assuming a knowledge base with n_c classes and n_p properties, the result list of SEARCHINDEX would have size $n_c + n_p$. In the worst case, where MATCHPAIR (Algorithm 3.1) takes two fuzzily defined columns as input, it enters the else clauses of lines 5 and 11. It therefore iterates over two nested lists of the worst case size $n_c + n_p$ and produces a list of size $(n_c + n_p)^2$. The complexity of MATCHPAIR is $\mathcal{O}((n_c + n_p)^2)$. The input of MATCHTABLE is a table definition with n_t columns. Since n_t usually ranges between 2 and 7, it can be seen as a constant factor. The number of possible column pairs to be matched is $n_t \cdot (n_t - 1)$. So the pairwise matching part of the algorithm has a complexity of $\mathcal{O}((n_c + n_p)^2)$. In the worst case, the exhaustive search part has to iterate over each of n_t master columns and for each column over all n_c classes. For each class over $n_t - 1$ other columns and for each other column over $(n_c + n_p)^2$ pair match results. The resulting number of steps is $n_t \cdot n_c \cdot (n_t - 1) \cdot (n_c + n_p)^2 = (n_t^2 - n_t) \cdot (n_c^3 + 2n_c^2 n_p + n_p^2)$. So the worst case complexity of the exhaustive search part is $\mathcal{O}(n_c^3)$ which is also the worst case complexity of MATCHTABLE.

Compared to that, the baseline table matching algorithm MATCHTABLEBASELINE (Algorithm 3.2) has a complexity of $\mathcal{O}((n_c + n_p)^2)$.

3.4.5 Practical performance & optimizations

The complexity analysis showed that the number of classes and properties has a high influence on the runtime of the table matching algorithm. Because of this, the performance of KBTE can be improved by limiting the number of intermediate results. Without limiting the result size of SEARCHINDEX, it would always return a list of size $n_c + n_p$ that contains all classes and properties of the knowledge base whereas the majority would have a popularity score of 0. For this reason, SEARCHINDEX is adapted, in order to return only matches with a popularity score higher than 0. The number of results of MATCHPAIR is also limited to 5. Because of the limitation, the complexity of MATCHPAIR gets $\mathcal{O}(1)$. This also reduces the worst case complexity of MATCHTABLEBASELINE to $\mathcal{O}(n_t^2)$. The complexity of MatchTable does not change because the exhaustive search part still has a higher influence than the pairwise matching part of the columns.

In practice, the performance of the improved algorithm should be acceptable because the number of classes and properties is constant. Nevertheless, the table matching should be slower for knowledge bases with a higher number of classes.

3.5 Query generation

To generate a SPARQL query, a variable is associated for each column. There is a difference between column's content instances that can either have labels (e.g. the column *city*) or values such as numbers or dates (e.g. *population* or *release date*). If they have labels, an additional name variable is associated with the column. Knowledge base entities are labeled by using a specific property. In Freebase it is the property *type.object.name*, in Wikidata it is *<http://www.w3.org/2000/01/rdf-schema#label>*.

The triples for the WHERE clause of the resulting SPARQL query are created according to the templates which result from the table matching.

The result query is assembled by filling an SPARQL barebone query. The column variables are inserted in the SELECT clause, the triples in the WHERE clause along with optional filters and orders that have been declared in the table definition input.

4 Evaluation

This chapter gives a description of the methods used to evaluate the application KBTE. For evaluating the generation of SPARQL queries with tabular content there is no well-known benchmark yet. Since KBTE tries to generate tables as used in Wikipedia articles (e.g. *list of largest cities*, *list of mountains by elevation*), this thesis provides a new dataset and evaluation method based on Wikipedia tables. The dataset consists of selected Wikipedia table column headers which are extracted from an English Wikipedia dump[1] containing all articles and other pages. The aim is to obtain a set of column headers per table that form table definitions in the tabular description format described in section 3.1. These table definitions are used as input for KBTE. The following section describes the selection criteria as well as the evaluation method details and results.

4.1 Evaluation with tables from Wikipedia

4.1.1 Creation of evaluation dataset

In order to create the evaluation dataset, this thesis used the Wikipedia dump of October 10, 2018. It contains 18,665,935 pages, whereof 14,051,148 have the article namespace. The single articles were parsed by the Python tool *WikiTextParser*[2]. From each Wikipedia article, all tables found were extracted. In total 2,372,431 tables were extracted. For further processing, it was tried to remove all the parts from the parsed tables, that are not part of the visible table content. Therefore all references (`<ref>...</ref>`) and line breaks (`</br>`) in the table cells were removed. Links (`[[target]]` or `[[target|text]]`) were replaced by their text if present, or by their target otherwise. The expansion of templates (`{{name|arg1|arg2|...}}`) is not possible with WikiTextParser, so the templates were retained. Tables with templates are therefore less suited for evaluation because of the following selection criteria. Tables should have at least two rows in order that the first row can be interpreted as column headers. Additionally, only tables with column spans and row spans of 1 are suitable, because larger spans lead to the duplicate column headers in the same table. For each column header, the search index of KBTE was searched for matching classes and properties in any of the available knowledge bases. If there is

[1] `https://dumps.wikimedia.org/enwiki/`

[2] `https://pypi.org/project/wikitextparser/`

at least one matching class or property, the “inKB”-attribute of the column header will be set to TRUE. For each table it is counted how many columns have “inKB”-attribute set to TRUE. Out of the extracted Wikipedia tables, only those tables are chosen for the evaluation dataset which have a count larger than 2 and larger than the total number of columns divided by 2. In this way 1,066,292 tables left for potential KBTE input.

It has to be mentioned that many Wikipedia tables do not contain the context to its articles, which results in more general table definitions. For example, the article to the country *Guinea* contains a table about its regions and prefectures. The produced table definition `Region | Capital | Population` has no relation to *Guinea*. The context could be recovered by manually adding an additional column with a filter of the article’s entity. A possible adapted table definition could be `Country(="Guinea"@en) | Region | Capital | Population`. This is infeasible for the large number of tables.

4.1.2 Evaluation method

To measure the performance of the table matching algorithm of KBTE, a query translation is executed for each evaluation table in the dataset for both Freebase and Wikidata back-end. The input table definition is created by the concatenation of those column headers that have its “inKB”-attribute set to TRUE, separated by vertical bar characters (`|`). For comparison, the same query translation are executed using the baseline algorithm described in subsection 3.4.2. The generated SPARQL queries are executed on Freebase and Wikidata back-ends running the query engine QLever[11]. The evaluation is performed over a small subset of the first 5000 tables.

An execution can lead to the following outcomes:

1. KBTE produces an error (*kbte-error*).
2. KBTE successfully outputs a SPARQL query, but the execution of this query on an appropriate knowledge base endpoint produces an error (*sparql-error*).
3. KBTE successfully outputs a SPARQL query, but the execution produces no result rows (*sparql-empty*).
4. KBTE successfully outputs a SPARQL query and the execution produces result rows (*sparql-full*).

It has to be mentioned that the fourth outcome (*sparql-full*) for a table definition does not imply a correlation between the content of the original Wikipedia table and the table resulting from the SPARQL query.

4.2 Evaluation with hand-crafted tables

Since the dataset with tables of Wikipedia only allows a purely quantitative evaluation method, the datasets with hand-crafted tables have been created for a more qualitative analysis. The datasets are shown in Table 4.1 for Freebase and in Table 4.2 for Wikidata. Each dataset contains 15 table definitions along with the same table definitions expressed with the exact intended classes and properties. It was tried to use the same table definitions for Freebase and Wikidata. Due to differences in the ontologies of the two knowledge bases and the missing ability of KBTE to recognize synonyms, some table definitions had to be slightly adapted. Also the missing ability to use occupations with KBTE for Wikidata is handled in the tables 9 and 10 by applying filters.

4.3 Results and discussion

The results of Wikipedia dataset evaluation are summarized in Table 4.3. Out of the 5000 evaluation tables, 3915 different table definitions were created. The results show that a high percentage of the table definitions is not suitable at all for KBTE to generate SPARQL queries. Comparing the baseline algorithm with the improved algorithm (called *KBTE* in the tables), one can see that the improved algorithm is able to generate more result yielding SPARQL queries for both Freebase and Wikidata. The most working SPARQL queries which yield at least one result are generated for Freebase using the improved algorithm (12.1%). However, the majority of these generated queries still does not match the intention of the table definition. For example, the table definition `Year|Film|Role|Notes` extracted from the Wikipedia article of the actor *Amitabh Bachchan* is translated to the Freebase table matching

```
{Film: (None, 'film.actor'), Year: (Film, 'award.award_nomination.
   year'), Role: (Film, 'music.track_contribution.role'), Notes: (
   Film, 'award.award_nomination.notes_description')}
```

The table matching leads to results (*sparql-full*) but the column matchings are not correct. The reason is that a column definition like `Actor(="Amitabh␣Bachchan"@en)` is missing which would restore the context to the article. Besides this, the lacking support of using synonyms is a problem here for the columns `Year` (to the property *film.film.release_date_s*) and `Role` (to the property *film.performance.character*). Furthermore, columns such as `Notes` which not directly relate to the table's content often cause the query translation to fail.

There are still well-translated queries for both Freebase and Wikidata. For example, the table definitions `District|Population` and `Country|City` (for both), `Metro Area | Population | Area | Country`, `County | Seat | Area | Population` and `Year | Winner` (for Freebase) or `Club | City | Sport | League | Venue` (for Wikidata).

The evaluation with Wikipedia tables shows that the created dataset represents a hard challenge for KBTE. The dataset may have to be further filtered and table definitions should be manually adapted for each knowledge base separately. In this way, it could be assured that the intended table content is contained in the knowledge base.

Table 4.4 and Table 4.5 show the analysis results of the hand-crafted datasets (in Table 4.1 and Table 4.2). Since these datasets are adapted to the specific content of each knowledge base, KBTE produces better table matchings compared to the Wikipedia tables dataset. Comparing by the percentage of correctly matched single columns, the improved algorithm reaches 83% in Freebase and 90% in Wikidata (Baseline: 30% and 61%). In the Freebase dataset, the improved algorithm is able to match all columns correctly for 11 of 15 tables (9 of 15 for Wikidata). The baseline algorithm is not able to match any table completely correct in Freebase and only one table in Wikidata. The failing column matchings of the baseline algorithm confirm the expectations mentioned in subsection 3.4.3. For example, in the table definition `mountain | country | elevation | mountain range` the column `elevation` is wrongly matched to *location.geocode.elevation* instead of *geography.mountain.elevation*. In the table definition `book | author | publication date` the column `publication date` is matched to *book.book_edition.publication_date* instead of *book.written_work.date_of_first_publication*. In a case where a mediator relation has to be used for a column, both algorithms do not match the intended relation. For example, the improved algorithm matches the column `mass` of the 15th table definition to *chemistry.atomic_mass.mass* instead of *chemistry.chemical_element.atomic_mass*. In this case, both properties should be working, the first property would match the CMP template, the second would match the CPM template.

On the dataset for Wikidata, the improved algorithm also performs better than the baseline algorithm. The baseline algorithm produces similar mistaken column matches as on Freebase tables. The general problem of the performance on Wikidata is the missing support for the class hierarchy which influences the calculation of popularity scores. This leads to the case where a subclass is preferred over its superclass because of a larger number of direct instances. For example, in the table definition `city | country | population | location` the column `city` is matched to the subclass *Q1093829* (“city of the United States”) of *Q515* (“city”).

	Table definition	Intended exact table definition
1	person | birth | death | spouse	people.person | people.person.date_of_birth | people.deceased_person.date_of_death | people.person.spouse_s
2	city | country | population | latitude | longitude	location.citytown | location.country | location.statistical_region.population | location.geocode.latitude | location.geocode.longitude
3	city | country | capital	location.citytown | location.country | location.capital_of_administrative_division.capital_of
4	mountain | country | elevation | mountain range	geography.mountain | location.country | geography.mountain.elevation | geography.mountain.mountain_range
5	film | release date | genre | country | director	film.film | film.film.release_date_s | film.film.genre | film.film.country | film.director
6	film | date | genre | country | director	film.film | film.film.release_date_s | film.film.genre | film.film.country | film.director
7	book | author | publication date	book.book | book.written_work.author | book.written_work.date_of_first_publication
8	book | author | date	book.book | book.written_work.author | book.written_work.date_of_first_publication
9	mission | astronaut | start | end	spaceflight.space_mission | spaceflight.space_mission.astronauts | time.event.start_date | time.event.end_date
10	politician | gender | country | date of birth	government.politician | people.person.gender | people.person.nationality | people.person.date_of_birth
11	super bowl | date | location | champion | runner-up | result	american_football.super_bowl | time.event.start_date | location.location | sports.sports_championship_event.champion | sports.sports_championship_event.runner_up | sports.sports_championship_event.result
12	sports teams | location | sport | league | venue	sports.sports_team | sports.sports_team.location | sports.sports_team.sport | sports.sports_team.league | sports.sports_team.venue
13	building | architect | country | height | floors	architecture.building | architecture.architect | location.country | architecture.structure.height_meters | architecture.building.floors
14	airport | city | country | passengers	aviation.airport | location.citytown | location.country | aviation.airport.number_of_passengers
15	element | number | symbol | mass	chemistry.chemical_element | chemistry.chemical_element.atomic_number | chemistry.chemical_element.symbol | chemistry.chemical_element.atomic_mass

Table 4.1: Hand-crafted table definitions for Freebase

	Table definition	Intended exact table definition
1	human | birth | death | spouse	Q5 | P569 | P570 | P26
2	city | country | population | location	Q515 | P17 | P1082 | P625
3	city | country | capital	Q515 | P17 | P1376
4	mountain | country | elevation | mountain range	Q8502 | P17 | P2044 | P4552
5	film | publication date | genre | country | director	Q11424 | P577 | P136 | P495 | P57
6	film | date | genre | country | director	Q11424 | P577 | P136 | P495 | P57
7	book | author | publication date	Q571 | P50 | P577
8	book | author | date	Q571 | P50 | P577
9	mission | human | occupation(="astronaut"@en)	Q2133344 | Q5 | P106(=”astronaut”@en)
10	human | occupation(="politician"@en) | gender | country | date of birth	Q5 | P106(=”politician”@en) | P21 | P17 | P569
11	super bowl | point in time | location | winner	Q32096 | P585 | P276 | P1346
12	club | city | sport | league | venue	Q847017 | Q515 | P641 | P118 | P115
13	building | architect | country | height | floors	Q41176 | P84 | P17 | P2048 | P1101
14	airport | city | country | patronage	Q1248784 | Q515 | P17 | P3872
15	element | number | symbol | mass	Q11344 | P1086 | P246 | P2067

Table 4.2: Hand-crafted table definitions for Wikidata

Knowledge base	Freebase		Wikidata	
Algorithm	Baseline	KBTE	Baseline	KBTE
# table definitions	3915			
kbte-error	3345 (85.4%)	2938 (75.0%)	3793 (96.9%)	3379 (86.3%)
sparql-error	46 (1.2%)	66 (1.7%)	2 (0.1%)	17 (0.4%)
sparql-empty	210 (5.4%)	436 (11.1%)	26 (0.7%)	247 (6.3%)
sparql-full	314 (8.0%)	475 (12.1%)	94 (2.4%)	272 (6.9%)

Table 4.3: Comparison between baseline and improved algorithm (KBTE) by frequencies of the outcomes (percentages are rounded)

	Number of correctly matched columns	
Table number	Baseline	KBTE
1	3/4 (75%)	4/4 (100%)
2	4/5 (80%)	5/5 (100%)
3	2/3 (67%)	3/3 (100%)
4	2/4 (50%)	4/4 (100%)
5	2/5 (40%)	5/5 (100%)
6	2/5 (40%)	5/5 (100%)
7	1/3 (33%)	3/3 (100%)
8	1/3 (33%)	3/3 (100%)
9	*kbte-error*	4/4 (100%)
10	2/4 (50%)	4/4 (100%)
11	*kbte-error*	2/6 (33%)
12	*kbte-error*	1/5 (20%)
13	*kbte-error*	3/5 (60%)
14	*kbte-error*	4/4 (100%)
15	*kbte-error*	3/4 (75%)
overall	19/64 (30%)	53/64 (83%)

Table 4.4: Results for Freebase table definitions comparing baseline and improved algorithm

	Number of correctly matched columns	
Table number	Baseline	KBTE
1	4/4 (100%)	4/4 (100%)
2	3/4 (75%)	3/4 (75%)
3	1/3 (33%)	2/3 (67%)
4	2/4 (50%)	4/4 (100%)
5	4/5 (80%)	5/5 (100%)
6	4/5 (80%)	5/5 (100%)
7	2/3 (67%)	3/3 (100%)
8	2/3 (67%)	3/3 (100%)
9	2/3 (67%)	2/3 (67%)
10	3/5 (60%)	5/5 (100%)
11	*kbte-error*	4/4 (100%)
12	4/5 (80%)	4/5 (80%)
13	3/5 (60%)	4/5 (80%)
14	3/4 (75%)	3/4 (75%)
15	*kbte-error*	4/4 (100%)
overall	37/61 (61%)	55/61 (90%)

Table 4.5: Results for Wikidata table definitions comparing baseline and improved algorithm

5 Summary and Future Work

This thesis presented an approach of searching for data in tabular form on knowledge bases and introduced the tool *KBTE*. It tries to reduce the effort necessary to retrieve the intended tables by providing a simple tabular description format which is used for querying. Compared to keyword search and question answering approaches, tabular information extraction shifts focus from entities to classes and properties of a knowledge base because table columns have to be matched to them. In addition to the matching for each column, the columns have to be related to each other.

The evaluation showed that KBTE is able to generate table contents for small subset of Wikipedia tables. Too many tables of the created evaluation dataset are not suitable as input table definitions for KBTE. In addition, many errors emerge due to the missing ability of matching columns without lexical relatedness to the label of the intended class or property. This issue could be solved by adding synonyms to the search index of KBTE. The alternative labels[1] used in Wikidata could be used for this purpose. For Wikidata, additional improvements are possible because of the different data model compared to Freebase. This includes professions of a person which are not separate classes as in Freebase (e.g. *film.actor*, *film.director*, *spaceflight.astronaut*, *book.author*) but instances of the Wikidata class *Q28640* ("profession"). The problem could be tackled by adapting the RDF extraction of subsection 3.3.2. In addition to the property *P31* ("instance of"), the property *P106* ("occupation") has to be considered. According to triples in the form `< Entity> <http://www.wikidata.org/prop/direct/P106> <Profession> .`, the entity dictionary should be extended. In a similar way, Wikidata subclasses could be considered as well. At the moment, subclasses are problematic for KBTE because only direct instances of classes are used as content for a column. For example, *Q515* ("city") does not include cities of subclasses such as *Q1549591* ("big city") or *Q1093829* ("city of the United States"). This problem could be solved by extracting the complete class hierarchy of Wikidata first and then additionally adding all superclasses of a class during creation of the entity dictionary. In this way, the column graph calculates the right counts for profession classes or superclasses. Obviously, these special class constructs have to be considered afterwards during query generation from the table matching.

The evaluation results to the hand-crafted datasets confirmed that the improved algorithm yields more accurate table matchings than the baseline algorithm. Al-

[1]with the predicate <http://www.w3.org/2004/02/skos/core#"altLabel>

though the efficiency of the improved algorithm is worse, it still performs well in practice.

List of Algorithms

List of Figures

Listings

List of Tables

Bibliography

[1] E. Demidova, X. Zhou, and W. Nejdl, "FreeQ: An Interactive Query Interface for Freebase," in *Proceedings of the 21st International Conference on World Wide Web*. New York, NY, USA: ACM, 2012, pp. 325–328. [Online]. Available: http://doi.acm.org/10.1145/2187980.2188040

[2] S. Ferré, "Sparklis: An Expressive Query Builder for SPARQL Endpoints with Guidance in Natural Language," *Semantic Web: Interoperability, Usability, Applicability*, vol. 8, no. 3, pp. 405–418, 2017. [Online]. Available: http://www.semantic-web-journal.net/content/sparklis-expressive-query-builder-sparql-endpoints-guidance-natural-language-1

[3] H. Bast and E. Haussmann, "More Accurate Question Answering on Freebase," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 1431–1440. [Online]. Available: http://doi.acm.org/10.1145/2806416.2806472

[4] T. Pellissier Tanon, M. D. de Assunção, E. Caron, and F. M. Suchanek, "Demoing Platypus – A Multilingual Question Answering Platform for Wikidata," in *The Semantic Web: ESWC 2018 Satellite Events*. Cham, Switzerland: Springer International Publishing, 2018, pp. 111–116.

[5] J. Moreno-Vega and A. Hogan, "GraFa: Scalable Faceted Browsing for RDF Graphs," in *The Semantic Web – ISWC 2018*. Cham, Switzerland: Springer International Publishing, 2018, pp. 301–317.

[6] Y. Yang, D. Agrawal, H. V. Jagadish, A. K. H. Tung, and S. Wu, "An Efficient Parallel Keyword Search Engine on Knowledge Graphs," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, April 2019, pp. 338–349.

[7] N. Chah, "OK Google, What Is Your Ontology? Or: Exploring Freebase Classification to Understand Google's Knowledge Graph," *Computing Research Repository (CoRR)*, 2018. [Online]. Available: http://arxiv.org/abs/1805.03885

[8] T. Pellissier Tanon, D. Vrandečić, S. Schaffert, T. Steiner, and L. Pintscher, "From Freebase to Wikidata: The Great Migration," in *Proceedings of the 25th International Conference on World Wide Web*. Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 1419–1428. [Online]. Available: https://doi.org/10.1145/2872427.2874809

[9] H. Bast, B. Buchhold, and E. Haussmann, "Semantic Search on Text and Knowledge Bases," *Foundations and Trends® in Information Retrieval*, vol. 10, no. 2-3, pp. 119–271, 2016. [Online]. Available: http://dx.doi.org/10.1561/1500000032

[10] D. Diefenbach, V. Lopez, K. Singh, and P. Maret, "Core Techniques of Question Answering Systems over Knowledge Bases: A Survey," *Knowledge and Information Systems*, vol. 55, no. 3, pp. 529–569, Jun 2018. [Online]. Available: https://doi.org/10.1007/s10115-017-1100-y

[11] H. Bast and B. Buchhold, "QLever: A Query Engine for Efficient SPARQL+Text Search," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17. New York, NY, USA: ACM, 2017, pp. 647–656. [Online]. Available: http://doi.acm.org/10.1145/3132847.3132921

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.