INSTITUT FÜR SOFTWARE & SYSTEMS ENGINEERING
Universitätsstraße 6a 86135 Augsburg

Recognizing, Naming and Exploring Structure in RDF Data

Linnea Passing

Masterarbeit im Elitestudiengang Software Engineering

SOFTWARE ENGINEERING
Elite Graduate Program

INSTITUT FÜR SOFTWARE & SYSTEMS ENGINEERING
Universitätsstraße 6a 86135 Augsburg

Recognizing, Naming and Exploring Structure in RDF Data

Matrikelnummer:	1227301
Beginn der Arbeit:	25. November 2013
Abgabe der Arbeit:	22. Mai 2014
Erstgutachter:	Prof. Dr. Alfons Kemper, Technische Universität München
Zweitgutachter:	Prof. Dr. Alexander Knapp, Universität Augsburg
Betreuer:	Prof. Dr. Peter Boncz, Centrum Wiskunde & Informatica, Amsterdam
	Florian Funke, Technische Universität München
	Manuel Then, Technische Universität München

SOFTWARE ENGINEERING
Elite Graduate Program

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Garching, den 22. Mai 2014 Linnea Passing

Abstract

The *Resource Description Framework* (*RDF*) is the de facto standard for representing semantic data, employed e.g., in the Semantic Web or in data-intense domains such as the Life Sciences. Data in the *RDF* format can be handled efficiently using relational database systems (RDBMSs), because decades of research in RDBMSs led to mature techniques for storing and querying data. Previous work merely focused on the performance gain achieved by leveraging RDBMS techniques, but did not take other advantages, such as providing a SQL-based interface to the dataset and exposing relationships, into account. In contrast, our approach is the first to strive for a complete transformation of RDF data into the relational data model. For that purpose, inherently unstructured RDF data is structured by means of semantic information, and relationships between these structures are extracted. Moreover, names for structures, their attributes, and relationships are automatically generated. Subsequently, using the relational schema thus created, RDF data is physically stored in efficient data structures. Afterwards, it can be queried with high performance and in addition – because of the generated names – be presented to users. Our experiments show that structures exist even within Web-crawled RDF data which is considered dirty. Using our algorithms, we can represent 79% of the DBpedia dataset (machine readable part of Wikipedia) by using only 140 tables. Furthermore, our survey shows that the generated table names get an average score of 4.6 on a 5-point Likert scale (1 = bad, 5 = excellent). Our approach therefore enables users to gain a fast and simple overview over large amounts of seemingly unstructured RDF data by viewing the extracted relational model.

Zusammenfassung

Das *Resource Description Framework* (*RDF*) ist der de-facto-Standard zur Repräsentation von semantischen Daten, wie sie zum Beispiel im Semantic Web oder in datenintensiven Forschungsbereichen wie den Life Sciences verwendet werden. Daten im RDF-Format lassen sich effizient in relationalen Datenbanksystemen verarbeiten, weil diese seit Jahrzehnten entwickelten Systeme über ausgereifte Techniken zur Datenspeicherung und -abfrage verfügen. Bisherige Arbeiten verwenden relationale Datenbanksysteme lediglich zur Steigerung der Performanz von Abfragen über RDF-Daten. Weitere Vorteile dieser Systeme, etwa das Herausstellen von Beziehungen und das Anbieten einer SQL-Schnittstelle zu den Daten, wurden bislang nicht beachtet. Unser Ansatz strebt erstmals eine vollständige Transformation der RDF-Daten in das relationale Datenmodell an. Dazu werden die inhärent unstrukturierten RDF-Daten mit Hilfe semantischer Informationen strukturiert und Beziehungen zwischen den Strukturen extrahiert. Sowohl für Strukturen als auch für ihre Attribute und Beziehungen werden unter Zuhilfenahme semantischer Informationen Namen erzeugt. Mittels des so generierten relationalen Schemas werden RDF-Daten in effizienten Datenstrukturen gespeichert, können performant abgefragt werden und zusätzlich, aufgrund der vergebenen Namen, auch Nutzern präsentiert werden. Unsere Experimente zeigen, dass selbst per Webcrawler gesammelte „dreckige“ Daten, Strukturen enthalten. Mit unseren Algorithmen können 79% der DBpedia-Daten (DBpedia enthält den maschinenlesbaren Teil der Wikipedia) auf nur 140 Relationen abgebildet werden. Die automatisch generierten Tabellenamen wurden im Durchschnitt mit 4,6 auf einer 5-Punkt-Likert-Scala, bei der 1 die schechteste und 5 die beste Bewertung darstellt, bewertet. Somit ermöglicht unser Ansatz einen einfachen Überblick über große Mengen eigentlich unstrukturierter RDF-Daten.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

In the last years, both the scientific community and industry have picked up the need for efficient RDF (*Resource Description Framework*) data stores. However, an inherent problem of RDF data, missing insight into data due to the lack of schema information, has not yet been tackled. This thesis presents techniques for finding and naming structures in RDF data that can be presented to the users to overcome this disadvantage of the RDF data model. The work described in this thesis is part of the *MonetDB/RDF* project that aims to fully integrate RDF data into the relational database system *MonetDB*. The *MonetDB/RDF* project is located at the Database Architecture group at *Centrum Wiskunde & Informatica* (CWI), where the *MonetDB* product family is developed.[1]

Section 1.1 of this chapter motivates the *MonetDB/RDF* project and the research that is done as part of it. In section 1.2, we define the scope of this thesis and name our research questions and contributions. Section 1.3 briefly introduces our approach to tackle the research questions. Finally, section 1.4 introduces a running example that is used throughout this thesis.

1.1. Motivation

The *Resource Description Framework* (RDF), the de facto standard for exchanging data on the Web [29], is much used in the *Semantic Web*, for example for annotating e-commerce items or social networks as well as for DBpedia[2], the machine readable version of Wikipedia. Besides the Semantic Web, RDF is widely accepted in data-intense domains such as *geography* (e.g., for weather forecasts and spatial data), *life sciences* (e.g., for taxonomies and drug databases), and also *open government data* (e.g., economic statistics or court decisions). This implies that RDF is used for *large datasets*, which raises a couple of problems on how to efficiently store, query, and explore RDF datasets.

RDF data is stored in *triples*. Each triple consist of a *subject*, a *predicate*, and an *object*. The terms subject, predicate, and object do also define grammatical components of sentences; and RDF triples can easily be seen as sentences talking about a resource (subject) having a certain value (object) in a category (predicate). Each triple therefore contains one *fact* about its subject. From another perspective, RDF data can also be interpreted as *graph data*, where subjects and objects are nodes and properties represent directed edges.

In comparison to regularly shaped data (as in the relational database model) graphs are much less restricted to a certain form. Because of this diversity, techniques for storing and querying RDF graph data tend to be more complicated and less efficient than techniques used in RDBMS. The simplest way to store RDF data is a so-called *triple store*, consisting of a single table with three columns *subject*, *predicate*, and *object*. Even for simple queries such as "Give all information about a certain subject S", difficulties arise: To collect all facts (*rows* in the tripe store table) about a subject, many *self-joins* are necessary. Besides that, no *data locality* is ensured for information that is often asked for in common, such as facts about a common subject. In addition, the lack of an explicit schema in graph data makes it difficult to pose the right queries to an RDF dataset, because it is not possible to present simple meta-information, as a relational schema would be.

[1] My stay at CWI was partially funded by *Studienstiftung des deutschen Volkes* (German National Academic Foundation).

[2] DBpedia `http://dbpedia.org/`

However, in practice, RDF data tends to be quite *structured* [20, 23]. Therefore attempts have been made to cluster RDF data and transform it to relational tables. By using *relational databases*, one can benefit from decades of experience in building RDBMS's, the well-known query language SQL, and relational tables as a familiar data model.

Well-researched techniques from relational database systems can be leveraged to optimize query plans [5, 8, 16, 20], improve storage layout [18, 34, 35], and build fast indices [1]. However, previous research focused on performance gains only, and did not tackle the problem of missing insight into RDF datasets. From a usability point of view, the lack of inherent structure in RDF graphs makes it hard for humans to gain insight into the data, which is necessary to formulate useful queries on it. Therefore, tools to outline and visualize RDF graphs have been implemented [7, 10, 11, 22]. Instead of creating graph visualizations to support users, we propose enhancing the relational schema of transformed RDF data to make it explorable and searchable by users.

Current research on *MonetDB/RDF* aims to combine these advantages of structuring RDF data by *i)* storing the data in such a way that both SPARQL and SQL queries can handle it efficiently, *ii)* structuring the triples into a relational schema, *iii)* labeling the data and providing a keyword search over the schema, and *iv)* making it available over an SQL interface, which makes structured RDF data usable via existing SQL visualization and exploration tools. The storage layout of *MonetDB/RDF* is based on so-called *characteristic sets* (CS's) [20], sets of predicates that occur on the same subjects [23]. Hence, CS's often correspond to real-world concepts, e.g., concept *book* with predicates *title*, *author*, or *publisher*. Grouping the data in concepts leads to human understandable partitioning into the synthetically created tables.

This thesis will focus on *ii)* and *iii)*, detecting structures in RDF data, creating human readable names for tables, attributes, and relations (i.e., foreign key attributes), and transformation into a relational schema. It will also include evaluating the understandability of the resulting data representation.

1.2. Research Questions

The topics of this research can be divided into three research questions that will be answered in this thesis.

Q1: Detecting relational structures in RDF data How can RDF triples be grouped and transformed to finally be represented as relational tables? Which concepts of the relational world are found during this process? Which information can be extracted from the RDF data itself, which additional data sources can be used?

Q2: Labeling the detected structures How can meaningful labels for these structures be created? Which information can be extracted from the RDF data itself, which additional data sources can be used? Is there a way to adopt labeling techniques from other domains?

Q3: Measures and evaluation How can the quality of the found structures be evaluated? How can "good labels" be measured (both technically and asking humans)? How to the quality of structure and the label quality connect?

1.2.1. Goals

In this master's thesis we develop a prototype for detecting relational structures in RDF data and labeling the structures. This work is part of the *MonetDB/RDF* project that aims for full integration of RDF data into the relational database system *MonetDB*. The software prototype to be developed will provide comprehensibility for large amounts of RDF data. A detailed list of goals and criteria is developed as part of this thesis and can be found in section 3.4. This detailed list differentiate between goals and criteria for structuring/merging

RDF data and for labeling these structures. Furthermore, additional criteria that have to be met for *production readiness* of the *MonetDB/RDF* module. These are listed as well. To define the scope of this thesis more detailed, *non-goals* are listed in the next section.

1.2.2. Non-Goals

Besides structure detection and labeling, *MonetDB/RDF* consists of physical data transformation, optimized algorithms to answer queries and an SPARQL interface. These tasks are not part of this thesis, but are researched and implemented by Pham [23]. Within the structure detection and labeling part of *MonetDB/RDF*, the following limits apply:

- The whole idea of *MonetDB/RDF* relies on *structured* RDF. Our research therefore focuses on the majority of structured RDF data only, the small percentage of *outliers* will not be included in the relational tables.
- Our approach requires multiple interlinked structures in the RDF dataset. Datasets that consist of one structure only, e.g., a large list of persons (subjects with the attributes `name`, `birthDate`, and `birthPlace`), cannot be structured using our approach and will end up as *one large relational table*. However, we aim to create good structures and labels for varying datasets, including e.g., Web-crawled data and DBpedia.
- As our approach only relies on predicate names and triple data as basis for label creation, the generated labels will never be as good as those generated from *texts*. We therefore do not aim for label quality comparable to those generated from texts.
- We currently do not support updating the dataset or adding new data to an existing dataset.
- Graphical representation of graph data is also out of scope, we only focus on tabular representation as provided over an SQL interface.

1.2.3. Related Work

Some research has been done on all research questions and is presented in the corresponding chapters of this thesis. Research on RDF stores can be found in section 2.2.1. Research on structuring (Q1) is presented in section 3.1. Research on labeling (Q2) can be found in section 4.2. Research on measures and evaluation (Q3) is presented in section 7.1.

1.2.4. Contributions

Structure detection in RDF data and labeling is our first contribution. We advance the property table approaches as well as characteristic sets to efficiently find structures in RDF data. We do not only take *structural* information into account as the afore-mentioned approaches do, but are the first to enrich an RDF dataset with additional *semantical* information from ontology values.

Secondly, we show that techniques from *Information Retrieval* are useful for structuring RDF data to computing similarity or exploit names. We use *tf-idf* (*term frequency*, *inverse document frequency*, a score on how good a term describes a document) to compute similarity of structures in RDF data, give special attention to certain properties (corresponding to field weighting in Information Retrieval), and exploit *anchor text* information and facts encoded in URIs itself.

Our third contribution is leveraging *relationships* between RDF subjects for structuring/merging and labeling. The previous work ignored these relations because they are not relevant for their performance improvement

goals, but humans can grasp a connected set of tables better than one without explicit connection, so we include relationship extraction.

Finally, we transform RDF data into a relational schema that is fast to query, efficient to store, and human readable. We are the first to include *foreign key relationships* into shredded RDF data. We therefore present the transformed schema to the users, what has not been done before in similar approaches. The relational schema we create is similar to schemas created by humans.

1.3. Approach

In this section, we will introduce our approach to answering the three research questions posed in section 1.2.

Structure Detection and Data Transformation *Characteristic sets* are used to detect the base structures in RDF data. To achieve larger structures, the algorithms allow small derivations in structure as missing values and data errors require a certain fuzziness. We discover that this *structural* approach is substantially improved by taking *semantical* information into account. Therefore the dividing line between the structuring and labeling tasks blurs, allowing each part to benefit from information gathered in the other one.

Information Extraction and Labeling All available textual information within the database should be taken into account for creating labels. Therefore, textual information has to be extracted from URIs and – if available – schema information. In a second step, external ontologies might be considered. The extracted data has to be transformed into meaningful labels for tables, attributes, and foreign key attributes. The algorithm to be chosen has to create labels that distinguish tables from each other and can be understood by humans. The labels should consist of one headword.

Evaluation To allow for judging the quality of the detected structures, measures have to be found, spanning e.g., the percentage of RDF data covered by a given number of relational tables, or the distinctness between each pair of relational structures. As our goal is to improve ease of use and comprehensibility, an evaluation method to measure these factors has to be designed. It should measure the correctness and comprehensibility of generated structures and its labels. Furthermore, several *variants* of the algorithms should be evaluated, to reason the decisions we make during designing these algorithms. For measuring comprehensibility for humans, a questionnaire has to be designed.

1.4. Running Example

We will introduce a running example that is used throughout this thesis. In this section, we show the transformation of RDF data into relational data in several steps. The process shown here is just meant to give a first overview. The details of each steps are described in chapters 3 to 5. Section 2.3 introduces the underlying data format *Resource Description Framework* (*RDF*), and the details of the identifiers (URIs) used within RDF data.

Figure 1.1 shows a small RDF graph. We will use this example to show the process of structuring, labeling, and merging RDF data. In reality, edges of RDF graphs are labeled with *Uniform Resource Identifiers* (*URIs*) rather than with words as shown in the figure. Examples of URIs are `http://dbpedia.org/ontology/country` and `http://xmlns.com/foaf/0.1/name`. As described in section 2.3.4, URIs in the

RDF context consist of a *prefix* and an *identifier*. For example, `http://xmlns.com/foaf/0.1/` is the prefix, denoting the *namespace* "Friend of a friend ontology". `name` is the identifier within the namespace. Figure 1.1 shows the input for the structuring and labeling algorithms that are presented in this thesis.

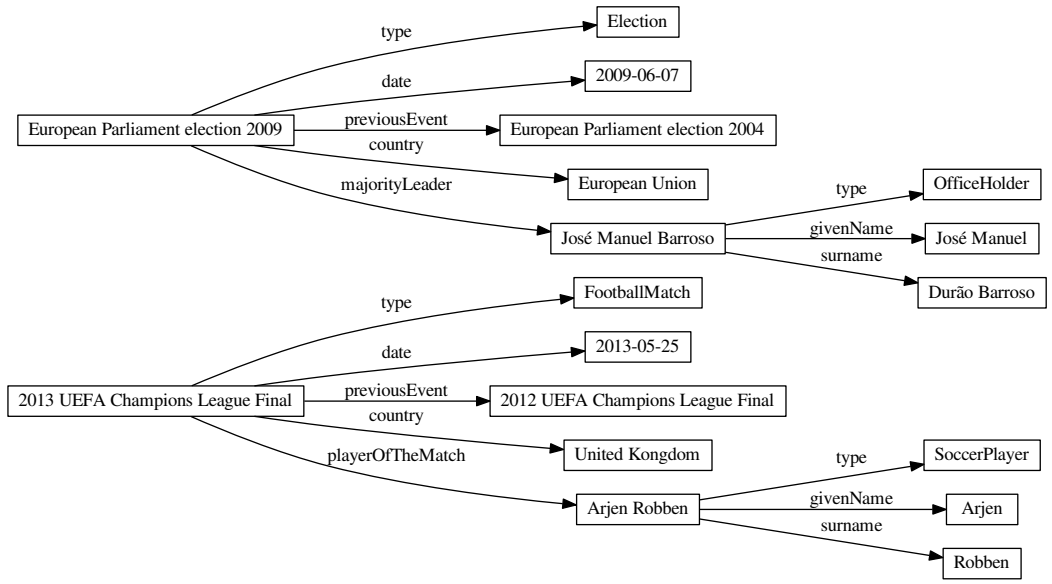


Figure 1.1.: Example: RDF graph

In a first step, the input RDF triples are grouped by subject. Each subject is uniquely identified by its URI. The result for the small example graph is shown in figure 1.2.

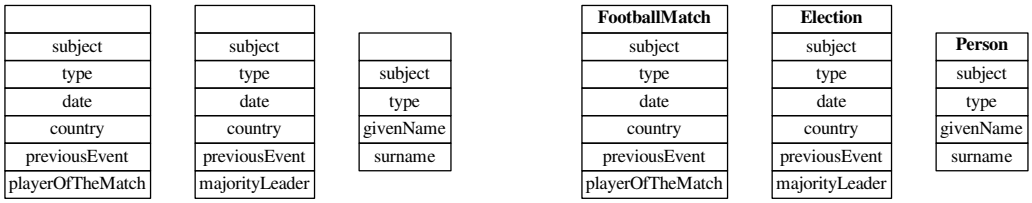
```
{2013 UEFA Champions League Final, type:FootballMatch, date:2013-05-25,
previousEvent:2012 UEFA Champions League Final, country:United Kingdom,
playerOfTheMatch:Arjen Robben}

{European Parliament election 2009, type:Election, date:2009-06-07,
previousEvent:European Parliament election 2004, country:European Union,
majorityLeader:José Manuel Barroso}

{Arjen Robben, type:SoccerPlayer, givenName:Arjen, surname:Robben}

{José Manuel Barroso, type:OfficeHolder, givenName:José Manuel,
surname:Durão Barroso}
```

Figure 1.2.: Example: RDF data grouped by subjects

For each subject, the set of its properties is computed. Subjects with the same property set (here: `Arjen Robben` and `José Manuel Barroso` share the same property set {type, givenName, surname}) can be grouped together. The result is a list of property sets with its associated subjects. These property sets are called *characteristic sets* (*CS's*) [20]. The CS's built from this RDF graph are shown in figure 1.3a. CS 1 contains the subject `2013 UEFA Champions League Final`, CS 2 contains `European Parliament`

`election 2009`, and CS 3 contains the two subjects `Arjen Robben` and `José Manuel Barroso`.

In the example, the values `FootballMatch`, `Election`, and *Person* of the predicate `type` are comprehensible choices for table names. In fact, the `type` information is available quite often (e.g., for nearly all subjects in DBpedia dataset), making it an important data source for labeling. To keep the example simple we do not consider other sources like ontology classes and foreign key relationships here. Information on these sources, as well as details about the usage of `type` properties for labeling is given in section 4.3. We assign each CS a label, as shown in figure 1.3b



(a) Characteristic sets

(b) Labeled characteristic sets

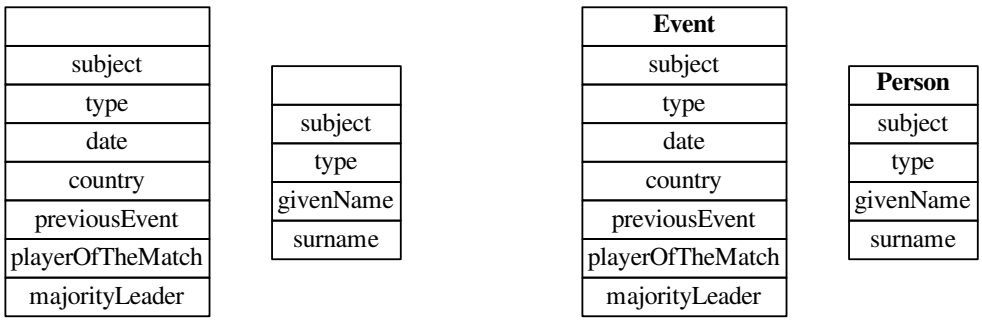
Figure 1.3.: Example: Detection of characteristic sets and initial labeling

At this point, we have a high number of similar CS's. For example, the birth date is known for most persons, but not for all. Therefore, we end up with *at least* two CS's that contain persons, one with a `birthDate` property, and one without. As our final goal is to achieve a small relational schema, we merge similar characteristic sets to reduce the amount of CS's. The merging rules are explained in section 5.1. In our example, the CS's `FootballMatch` and `Election` are quite similar. The only difference is the `playerOfTheMatch` property which is missing in Election, and the `majorityLeader` property which is missing in `FootballMatch`. The other four properties (`type`, `date`, `country`, `previousEvent`) are shared. Therefore, these two CS's are merged, as shown in figure 1.4a.

Besides the similarity of property sets, additional resources are integrated for merging. One important resource is the *category hierarchy* which is built from ontology information (cf. section 4.3). A category hierarchy allows for *semantical* comparison between two concepts. For example, the concepts `FootballMatch` and `Election` belong to the same subtree `Event`. This means they have a common set of properties that is defined by `Event` and some individual additional properties for their specific type of event. The name of the common ancestor is a good choice for a CS label of the merged CS. The resulting names are shown in figure 1.4b.

Afterwards, the intermediate UML schema can be computed. It consists of the merged CS's plus *relationships* between them. The values of the property `majorityLeader` within `Event` are usually persons, meaning they can be found in the `Person` CS. These relationships are added to the CS's to build the UML schema, as shown in figure 1.5a.

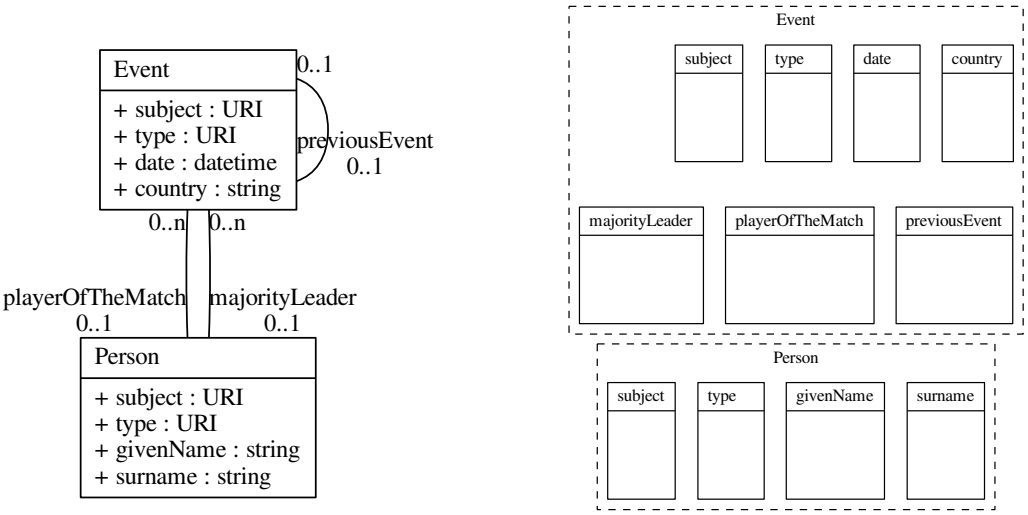
The final step is the transformation into a relational schema. A possible transformation into relational tables is shown in figure 1.5b. As *MonetDB* is a column-store, each property is stored in its own "table" with only one data column. To keep this example simple, difficulties with data types, NULL values, and multivalued properties were left out. They are explained in section 5.4.



(a) Merged characteristic sets

(b) Labeled merged characteristic sets

Figure 1.4.: Example: Merging of characteristic sets and final labeling



(a) UML diagram

(b) Relational schema

Figure 1.5.: Example: Transformation of RDF graph data to relational structures

Organization The rest of the thesis is organized as follows. Chapter 2 describes the fundamentals of *MonetDB* and *MonetDB/RDF*. Furthermore, the basics of Semantic Web Technologies such as the *Resource Description Framework* (*RDF*) are covered. Chapter 3 discusses the detection of structures in RDF data, and the implementation of the proposed algorithms in *MonetDB/RDF*. Chapter 4 introduces algorithms to label these structures by exploiting RDF metadata. In Chapter 5, we discuss how the labeled structures can be further refined and transformed into relational structures. Chapter 6 describes the integration of the developed algorithms into the *MonetDB/RDF* module. Experimental results are discussed in chapter 7. A conclusion and outlook is given in chapter 8.

2. Fundamentals

In section 2.1, the RDBMS *MonetDB* is introduced. *MonetDB* is the base system that will be extended to support RDF data by the *MonetDB/RDF* project. We describe the *MonetDB* fundamentals to make design decisions made *MonetDB/RDF* understandable. In addition, the *MonetDB* architecture is introduced, this is necessary for later descriptions of the interactions between *MonetDB* and the *MonetDB/RDF* module.

MonetDB/RDF, the module of the relational database system *MonetDB* handling RDF data, is introduced in section 2.2. This thesis contributes to *MonetDB/RDF*, therefore this introduction to the module explains the basic concepts of the module only. Section 6.3 contains further information about the architecture of *MonetDB/RDF*.

In section 2.3, this chapter explains the basics of Semantic Web Technologies, e.g., the *Resource Description Framework* (*RDF*), the query language *SPARQL*, and ontologies. Scope and content of this work are directly influenced by data formats, query languages, and standards that exist around the Semantic Web. We therefore introduce these technologies.

2.1. *MonetDB*

Being based on the relational database management system (RDBMS) *MonetDB*[3], *MonetDB/RDF* makes heavy use of features and concepts of the database system. We therefore introduce *MonetDB*, its fundamental concepts, and architecture.

The implementation of the open-source database system *MonetDB* started in 1993 at the Database Architecture group at *CWI*, the national research institute for mathematics and computer science in the Netherlands. Today, *MonetDB* is still being actively developed there.

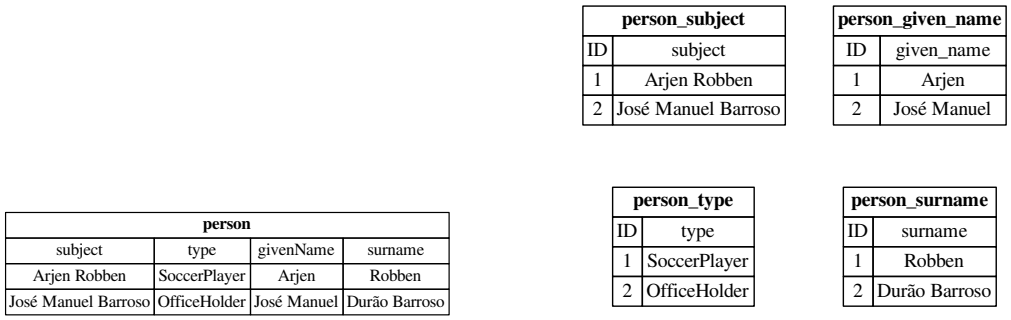
Though implementing the whole SQL 2003 standard, *MonetDB* is mainly built to support OLAP workloads on large datasets, e.g., for business intelligence and e-science. The system therefore is used for example in telecommunication companies and astronomy projects [13]. Besides the relational model, *MonetDB* supports XML data and array data. With *MonetDB/RDF*, support for RDF data is added.

Features The large amount of main memory that is available on present-day systems is exploited by *MonetDB* [17] to increase performance while keeping the advantages of storing data on disk, i.e. persistence. Performance is further improved by using *run-time query optimization*, i.e., the algorithms to be used in the execution of a query are chosen when the query is run to exploit additional information, e.g., whether the data is already sorted, available at that point in time.

The database system names itself “column-store pioneers”, as *MonetDB* was one of the first database systems using a vertical data layout instead of the classical horizontal, tuple-centered data layout. While databases with a horizontal data layout have advantages to return full tuples, column-store systems are usually faster to return aggregates over single columns. In column-stores, each relational table is split into

[3] *MonetDB* `http://www.monetdb.org/`

two-columnar tables. A relational table with n columns is split into n two-columnar tables as shown in figure 2.1.



person			
subject	type	givenName	surname
Arjen Robben	SoccerPlayer	Arjen	Robben
José Manuel Barroso	OfficeHolder	José Manuel	Durão Barroso

person_subject	
ID	subject
1	Arjen Robben
2	José Manuel Barroso

person_given_name	
ID	given_name
1	Arjen
2	José Manuel

person_type	
ID	type
1	SoccerPlayer
2	OfficeHolder

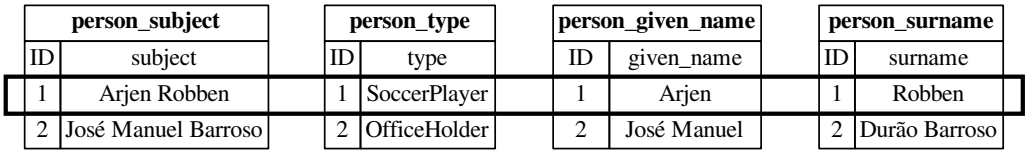
person_surname	
ID	surname
1	Robben
2	Durão Barroso

(a) Horizontal data layout (row-store)

(b) Vertical data layout (column-store)

Figure 2.1.: Horizontal and vertical data layout in RDBMSs

Every two-columnar table stores one column of the original relational table, together with object identifiers. In *MonetDB*, the two-columnar tables are called *Binary Association Tables* (*BATs*). The two columns in a BAT are called *head* and *tail*, with the head containing object identifiers (OIDs) and the tail containing the values. Values in different BATs that belong to the same tuple are associated with the same OID, thus OIDs allow *tuple reconstruction* as shown in figure 2.2.



person_subject	
ID	subject
1	Arjen Robben
2	José Manuel Barroso

person_type	
ID	type
1	SoccerPlayer
2	OfficeHolder

person_given_name	
ID	given_name
1	Arjen
2	José Manuel

person_surname	
ID	surname
1	Robben
2	Durão Barroso

Figure 2.2.: Tuple reconstruction in vertical data layout, based on same OIDs

MonetDB reconstructs tuples as late as possible during a query execution. Processing BATs instead of tuples allows for a *vector-like* execution model with few function calls and high instruction locality in the algorithms [13]. OIDs do not need to be stored explicitly because they represent *positions* in the BATs, similar to array indices that are neither stored explicitly.

Data types of variable length, e.g., strings, are stored using *dictionary encoding*. Strings that appear multiple times in the dataset are stored only once, and each usage of this string refers to an entry in the dictionary. This also allows for easier string comparison, as *pointers to the dictionary* have to be compared instead of strings. These pointers are represented as OIDs in *MonetDB* and can be compared fast. Figure 2.3 shows an example dictionary and how it is used in BATs.

Architecture *MonetDB* has a three-tier architecture [13]. The database system can be queried using SQL, JDBC, XQUERY, or – in our case – SPARQL. The *front-end* transforms these queries to – in case of SQL –

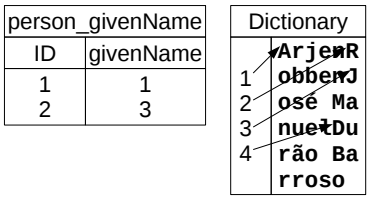


Figure 2.3.: Dictionary encoding and a BAT using dictionary OIDs

relational algebra which is then translated to *MAL*. *MAL*, the *MonetDB Assembly Language*, is a low-level language used for query execution within *MonetDB*.

The query plan in MAL is then send to the *back-end*. The back-end optimizes the plan, similar to a compiler, and adds rules for resource management.

The third tier is the *kernel* of *MonetDB*. Within the kernel, the BAT storage structure is provided. Furthermore the kernel offers relational operators, transformed to support the vertical storage layout. Several MAL implementations for each operator exist and are chosen at runtime.

One key feature of *MonetDB* is its *extensibility*. By adding new modules, the systems functionality is enhanced. For example, *MonetDB/RDF* is a module.

MonetDB/RDF makes use of some features mentioned afore-head, namely vertical storage (BATs) and dictionary encoding. An overview of *MonetDB/RDF* is given in section 2.2.

2.2. *MonetDB/RDF*

MonetDB/RDF is an enhancement of *MonetDB* to provide support for RDF data. The main idea is the usage of *characteristic sets* [23] to physically cluster data that belongs to the same subject or similar subjects. Afterwards, storing and querying techniques for relational data can be adapted and re-used for RDF data. Therefore, the *MonetDB/RDF* module is a lightweight component.

The following sections describe the core ideas of *MonetDB/RDF* storing and querying and the challenges of usability that will be dealt with in this thesis.

2.2.1. Storage

Related Work The typical implementation of RDF database systems as *triple store* led to query plans characterized by a high number of self-joins, for which RDBMS are not optimized. Therefore, two other storage possibilities were implemented: *Decomposed stores* [1] consist of one table per predicate. Hence, every table consists of two columns, subject and object. As the data is ordered by subject, linear joins can be used to improve performance compared to triple-stores. *Property table stores* [8, 34, 35] store data by subjects. A set of subjects that has the same set of predicates is grouped in a table. This layout reduces the amount of joins as the majority of RDF queries sample one *concept* only. Combinations of both layouts have been proposed by Levandoski and Mokbel [16] to further improve performance.

A similar approach to property tables has been proposed by Matono and Kojima [18]. The authors propose *paragraph tables* that store data structured by subjects, but assume that the *order* of subjects within an RDF data file implies correlation. Their approach therefore works for well-structured RDF files only, but

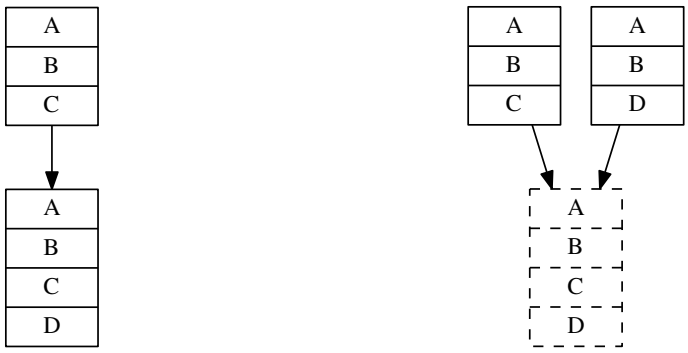
not for Web-crawled data or generated RDF files. For example, the DBpedia data files are sorted roughly alphabetical.

Recently, Bornea et al. [5] built an RDF store on top of a relational system. However, instead of exploring the underlying concepts, they use hash functions to shred RDF data into few tables regardless of the concepts the triples belong to. Although achieving notable performance, their schema does not contain relationships between concepts and is not design to be understood by humans.

Neumann and Moerkotte [20] use *characteristic sets* (CS's) to improve cardinality estimation, this idea will be adopted and used for data storage in *MonetDB/RDF* by Pham [23]. Characteristic sets group subjects with the same set of predicates, similar to property table stores. In contrast to property tables, characteristic sets are built without human effort by computing the predicates per subject and grouping subjects with the same predicates.

However, none of the previous work had the need to label the tables or make them searchable as they were not presented to users.

Extensions to *Characteristic Sets* The idea of characteristic sets is tuned to allow its usage as storage model [23]. By allowing NULL values in the CS's, many similar CS's can be joined. This leads to less CS's, therefore better storage structure and better comprehensibility by users. Figure 2.4 shows the two types of merging: Including a complete subset into its superset, and merging *similar* CS's. In opposition to what is shown in the figure, more than two CS's can be merged at a time. For the latter case, a similarity metric based on *tf-idf* is used to compute whether two CS's are similar enough to be joined. The details of CS's and CS merging in *MonetDB/RDF* are described in sections 3.2 and 5.1.



(a) A CS is merged into an existing CS which is a superset

(b) Two similar CS's are merged into the newly created CS, their least common superset

Figure 2.4.: Types of similarity between CS's

If the predicate set is the only criterion for creating CS's, values with different data types fall into the same CS. For example, values of a predicate `price` could be integers or floats. *MonetDB/RDF* creates separate CS's for each combination of data types in the attribute set [23].

CS's as described by Neumann and Moerkotte [20] had no need to keep track of links between CS's, i.e., an object in CS 1 is a subject in CS 2. In *MonetDB/RDF*, these relationships are stored as foreign keys to keep the linkage information that was found in the RDF data. Attributes with multiple values per subject are split up to a separate CS to normalize the data according to the relational model.

Data Sorting and Storing To avoid random access pattern in RDF query plans, the subjects are grouped by CS and physically clustered [23]. Within CS's, further ordering can be achieved by using predicates as additional indices. Inside these clusters, the records are ordered by object literals to simplify (range) predicate evaluation. The resulting clustered storage structure is a *triple store*.

After the structures in the RDF data are extracted, *MonetDB/RDF* explores foreign keys and possible primary keys within the CS's.

2.2.2. Querying

The CS-wise clustered storage can lead to efficient retrieval of sets of subjects with much less joins. To achieve this, new operators have been created.

If a query covers one CS only, no join is needed because all data to be received is stored aligned. The operator used in this case is called *RDFscan* [23] and provides a CPU efficient scan over the queried CS. For more complex queries, where subjects from more than one CS are covered, an operator *RDFjoin* is provided. *RDFjoin*, originally proposed as *Pivot Index Scan* by Brodt et al. [6], takes a binding for one CS and a linkage to the other CS as input and delivers a list of objects.

Example *RDFscan* is used to answer queries like "Show the names of all persons of type `SoccerPlayer`", because only one CS is involved. Unlike simple RDF index scans, the *RDFscan* operator in *MonetDB/RDF* delivers multiple objects per resulting subject. To answer "Show all football matches whose player of the match is left-footed", which queries two CS's, *RDFjoin* is used. The operator gets two inputs: the information that the person's footedness has to be left-footed and the information that the CS's are linked using the `playerOfTheMatch` predicate.

2.2.3. Usability

If RDBMS provide better storing and querying solutions for (structured) RDF data, why not using RDBMS techniques to improve the usability of RDF data? This is where the research in this thesis starts.

To overcome the lack of usability of RDF triples, some visualization tools have been created over the last years. Both graphical tools for exploring the graph [10, 11, 22] and hierarchical utilities that work on a simple table-like representation of RDF data [7] exist. However, these solutions are additional stand-alone software products that provide much less integration with querying and processing tools than the standard SQL interface that we strive for. Furthermore, our solution will enable the users to explore *similar subjects* by grouping them into tables. In addition, relational tables are a well-known and dense format to represent data. Graph representations visualize links to other parts of the data. This feature will be provided in *MonetDB/RDF* by fetching neighbor tables using foreign keys and hierarchical relationships.

Based on the concepts of relational tables and relationships between them, labels can be created to describe the contents of the generated tables. Chapter 4 discusses research on labeling and how these solutions can be applied to our task.

2.3. Semantic Web Technologies

The goal of this work is to extend *MonetDB* for supporting data in the *Resource Description Framework* format (*RDF*). RDF is a data format standard in the *Semantic Web*. We therefore briefly introduce the Semantic Web, the Resource Description Framework (RDF), and other related technologies.

2.3.1. Semantic Web

The World Wide Web Consortium [32] defines the Semantic Web as follows:

> The Semantic Web is about two things. It is about common formats for integration and combination of data drawn from diverse sources, where on the original Web mainly concentrated on the interchange of documents. It is also about language for recording how the data relates to real world objects.

Thus, the Semantic Web aims for machine-readability and standardized ways of interlinking data. These common formats and language are described in the following section.

The ideas of the Semantic Web can be found in different applications. Examples are

- *Semantic Search*, where search queries are enriched or refined using semantic information like synonyms or temporal context, e.g., a search for "bike rental" also giving results containing "bicycle rental",
- *Semantic Wikis*, where human readable wiki texts are enriched with machine readable information, ensuring – for example – machine-readability of relations without trimming the variety of expressing relations in human languages,
- and *Geotagging*, where geographical data is enriched with a standardized geo format that allows e.g., computing the distance between entities.

2.3.2. Linked Data

Linked data, also referred to as *graph data*, deals with bits of information connected to each other, hence forming a graph. Linked data can be found on the Web, as *links* are a main feature of the WWW. Furthermore, linked data representation is also applicable for all datasets about *networks*, e.g., datasets describing computer networks or social networks. Linking information enables users to interactively *browse* data, e.g., by using graphical tools as pointed out in section 1.1. Well-known examples for linked datasets are *DBpedia*, a machine readable subset of Wikipedia data, and *UniProtKB*[4], containing protein information.

It is also possible to set links to a different dataset. This is called *interlinking*. For interlinking different data sources, equivalences or relations between entities in both datasets have to be defined. This is done inside each dataset by explicitly naming *equal* entities that can be found in other datasets, or listing *hypernyms* and *hyponyms*. If this information is present, it is possible to automatically traverse both datasets as if they were one dataset. Interlinked data sources offer additional information, e.g., data from social networks can be enriched with additional geographical information. Interlinkage of many datasets, including the afore-mentioned *DBpedia* and *UniProtKB*, is provided by the LOD (Linked Open Data) community and the *LOD2* project[5].

To allow interlinking and usage of linked data, standards are needed. These standards are described in the following section. The *Resource Description Framework* (*RDF*) serves as language for linked data, providing the general concept of *resources* and *triples*. Various syntaxes exist to express linked data in RDF format. Another standard is needed for querying these datasets: the query language *SPARQL* is the standard query language for accessing RDF databases. Additional vocabulary is needed to describe *relations* between facts in the datasets. Basic vocabulary is provided by *RDF Schema* and the *Web Ontology Language* (*OWL*). Vocabularies can easily be build based on the *ontology* concept. Vocabularies for domains like e-commerce[6],

[4] UniProtKB `http://www.uniprot.org/`
[5] LOD2: Creating Knowledge out of Interlinked Data `http://lod2.eu/`
[6] Good Relations [12] `http://www.heppnetz.de/ontologies/goodrelations/v1.html`

social networks[7], or drugs[8] are already established. Equipped with these standards, it is possible to create datasets that can be extended and analyzed by others easily.

2.3.3. Resource Description Framework (RDF)

Segaran et al. [29, p. 64] define RDF as

> language for expressing data models using statements expressed as triples.

RDF has become a W3C recommendation in 1999 [15], and is therefore the de facto standard to express semantic data. For "expressing data models", the first part of the definition by Segaran et al. [29], RDF conceptualizes everything as *resource* [29, p. 65]. To avoid ambiguities, resources are identified using a *Universal Resource Identifier* (*URI*), e.g., an URL pointing to a Web resource or an ISBN identifying a book. The syntax of URIs is described in section 2.3.4. As the second part of the definition mentions, RDF data is expressed as *triples*. These triples consist of a *subject*, a *predicate*, and an *object*. The terms subject, predicate, and object do also define grammatical components of sentences; and RDF triples can easily be seen as sentences talking about a resource (subject) having a certain value (object) in a category (predicate). Each triple therefore contains one *fact* about its subject. A common serialization format for RDF triple date is *N-triples*. N-triples is a plain text format where each line contains one triple (subject, predicate, object) with the three elements separated by spaces.

Picking up the example triple used in the introduction, we are now able to extend the example to reflect the graph structure of RDF data. In `(2013 UEFA Champions League Final, country, United Kingdom)`, the resource `2013 UEFA Champions League Final` has a value for the category `country` which is `United Kingdom`. `United Kingdom` itself can serve as subject in other triples, e.g., `(United Kingdom, capital, London)`, and `2013 UEFA Champions League Final` can be the object value in other triples, respectively. To answer the question "What is the area of the capital of the United Kingdom?" we need a query language to *traverse* the graph. This language is introduced in the following section.

2.3.4. Uniform Resource Identifier (URI)

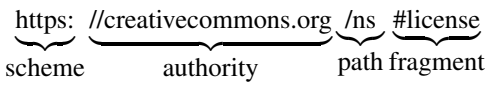


Figure 2.5.: Example of the URI syntax

Uniform Resource Identifiers are used to distinguish resources. Syntax and the process for resolving URIs are defined in the Internet Standard 66 [4]. Figure 2.5 shows an example. An URI consists of a *scheme* (e.g., `http`) followed by a colon), an optional *authority* (e.g., a domain name), an optional *path*, an optional *query* starting with a question mark, and an optional *fragment* beginning with a pound sign.

URIs take different roles in RDF triple data: If used as subjects, URIs are a unique identifier for instances. If used as predicates, URIs define a relationship between subject and object.

Objects values can be either literals (strings, dates, numbers) or URIs. Object URIs are either links to subjects within the dataset (*foreign keys*) or links to subjects in external data sources. An example for the latter case are RDF triples like the one shown in figure 2.6 where the object URI refers to the external GoodRelations ontology to classify the subject.

[7] FOAF `http://xmlns.com/foaf/spec/`
[8] DrugBank `http://www.drugbank.ca/`

```
<http://example.com/onlineshop/product/leather_armchair>,
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
<http://purl.org/goodrelations/v1#ProductOrService>
```

Figure 2.6.: Example of an object URI linking to the external GoodRelations ontology

2.3.5. SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL is the query language for RDF data. SPARQL searches for given patterns in an RDF graph as shown in figure 2.7. First, the URI prefix `http://purl.org/goodrelations/v1#` is abbreviated to `gr` to improve readability. The query searches for entities `o` that fulfill the requirements listed in the WHERE clause: The sought entity has to be of type `Offering` which is a class defined in the GoodRelations namespace *gr*. The sought entity has to have the value `gr:Cash` for the predicate `gr:acceptedPaymentMethods`. `gr:Cash` is a so-called *predefined individual*. Predefined individuals can be seen as enumeration values. They avoid ambiguities caused by, for example, spelling mistakes if string literals would have been used instead. In conclusion, the SPARQL query returns all offers that can be paid cash.

```
PREFIX gr:<http://purl.org/goodrelations/v1#>

SELECT ?o
WHERE {
    ?o a gr:Offering.
    ?o gr:acceptedPaymentMethods gr:Cash.
}
```

Figure 2.7.: SPARQL query to find all offers that can be paid cash using GoodRelations ontology

To answer the question from the previous section, we could use the query in figure 2.8 that uses the capital `?c` as internal intermediate hop but does not include it in the result.

```
SELECT ?a
WHERE {
    <United Kingdom> <capital> ?c.
    ?c <area> ?a.
}
```

Figure 2.8.: SPARQL query to find the area of the capital of the United Kingdom

2.3.6. Ontologies and the Web Ontology Language (OWL)

In information science, *ontologies* are formal descriptions of concepts and structures. Ontologies cover a limited area such as e-commerce[9], proteins[10], or business cards[11]. Ontologies use cross-references to define relationships between similar concepts (e.g., the concept of a *comment* exists in multiple ontologies). As mentioned before, these equivalences are essential to create interlinked datasets. The ontology concept is independent from its syntactical representation. For their use in the Semantic Web, ontologies are often expressed in *XML/RDF*, a common XML representation for RDF data.

[9] Good Relations [12]
[10] UniProt `http://www.uniprot.org/core/`
[11] vCard `http://www.w3.org/TR/vcard-rdf/`

Ontologies are described using the *RDF Schema*[12] (*RDFS*) and *Web Ontology Language*[13] (OWL) ontologies, both defined by the W3C. RDFS defines basic concepts like *Class*, *Property*, and *Literal* as well as basic descriptions and connections between these concepts (e.g., *domain*, *subClassOf*). OWL is an extension to RDFS [29, p. 135] and defines more specific concepts and relationships such as *unionOf* or *incompatibleWith*.

Figure 2.9 shows an example of a class definition in an ontology in RDF/XML format. The defined class `foaf:Person` is initially described by a label. Additionally, relationships with other classes are described: `foaf:Person` has two superclasses `foaf:Agent` and `contact:Person`, at which the latter superclass belongs to a different ontology. The `disjointWith` property defines that a `Person` must not be a `Project`.

```
<rdfs:Class rdf:about="http://xmlns.com/foaf/0.1/Person" rdfs:label="Person">
  <rdfs:subClassOf>
    <owl:Class rdf:about="http://xmlns.com/foaf/0.1/Agent"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class rdf:about="http://www.w3.org/2000/10/swap/pim/contact#Person"/>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="http://xmlns.com/foaf/0.1/Project"/>
</rdfs:Class>
```

Figure 2.9.: (Shortened) example of a class definition in FOAF ontology

Although the Semantic Web is a relatively new research topic and its technologies are changing constantly, some standards have been established. Standards are crucial for the Semantic Web as it depends on a large number of contributors adding data and vocabulary. Besides the Semantic Web, RDF is widely used for graph-formed data, e.g. in life sciences. In this section we have introduced the key techniques we need to build an RDF module for *MonetDB*.

[12] RDF Schema `http://www.w3.org/TR/rdf-schema/`
[13] OWL `http://www.w3.org/TR/owl-ref/`

3. Finding Structure in RDF Data

Adding structure to RDF data and transforming the RDF data into relational data leads to improvement in two dimensions: *i)* Data retrieval via SPARQL can be accelerated by using relational operators. These operators need to be adjusted to the characteristics of RDF data, but do still benefit from the decades of experience in relational databases. *ii)* Users can benefit from relational techniques and the tool chain of relational databases. Being able to use these well-known and mature techniques supports their understanding of the data.

In this chapter, we describe how structures in RDF data are identified and how these structures are leveraged to create an understandable, high-performance data representation in the *MonetDB/RDF* module. In section 3.1, we present related work on structuring RDF data. Section 3.2 introduces *characteristic sets*, a structure defined by Neumann and Moerkotte [20] and used as base for finding broader structures in RDF data in our work. In section 3.3, we describe the complete process of finding structure in RDF data. The goals and criteria of the structuring/merging and labeling task are collected in section 3.4. Sections 3.5 to 3.8 describe the steps of the process in detail.

The structuring process described in this chapter is followed by the labeling step (chapter 4), that adds ontology information to RDF data to allow for labeling the detected structures, and the merging step (chapter 5), that further simplifies the structures by merging similar ones. The algorithms described in these three chapters transform RDF triple data into relational data that is both fast to query and easy to understand by users, as shown in our evaluation (chapter 7).

(Implementations described in sections 3.5 to 3.8 have been done by Pham Minh-Duc.)

3.1. Related Work

So far, structures in RDF data were explored and exploited to optimize query plans [5, 8, 16, 20], improve storage layout [18, 34, 35], and build fast indices [1]. None of the previous work was interested in the *readability* of the structures, as they were not presented to the users. The related work can therefore only be the basis for the structuring algorithms we describe.

Property tables that group RDF triples by subject, as described by [8, 34, 35] and similar approaches (e.g., [16, 18]) form the basis for our structuring algorithms. In opposition to these previous approaches, we take *relationships* between structures into account, and *label* the structures we find (cf. chapter 4). In addition, we aim for transforming the vast majority of data in a dataset first into an UML-like representation and then into a relational schema.

Another basis for our approach are the *characteristic sets* [20] that formalize the grouping idea behind property tables and use them for cardinality estimation. Characteristic sets are therefore introduced in detail in section 3.2.

3.2. Characteristic Sets

A *characteristic set* (CS), as introduced by Neumann and Moerkotte [20], contains the properties of all RDF data triples with the same subject. For a given subject s in a dataset R, they define a characteristic set as the set of properties that are available for s:

$$CS(s) := \{p | \exists o : (s, p, o) \in R\}. \tag{3.1}$$

For two subjects s_1 and s_2, $CS(s_1)$ equals $CS(s_2)$ if the same set of properties is used to describe both s_1 and s_2.

Characteristic sets are used by Neumann and Moerkotte [20] to estimate cardinality in RDF queries only. Our approach aims to leverage the idea of characteristic sets for structuring RDF data by using the above-mentioned comparability of CS's. We define $CS(P)$ as the set of triples (s, p, o) that share the same property set. For a given set of properties P and a dataset R we define a characteristic set as follows:

$$CS(P) := \{(s, p, o) | (s, p, o) \in R \wedge CS(s) = P\}. \tag{3.2}$$

Using this definition, subjects whose property set is a *superset* of P are not included in $CS(P)$. Further mentions of the term *CS* in this thesis use definition (3.2).

Because of the common property set, all data in a CS can be represented as table, as shown in table 3.1. The properties serve as columns, and each subject adds a row to the table. All table cells are filled with objects.

subject	type	givenName	surname
Arjen Robben	SoccerPlayer	Arjen	Robben
José Manuel Barroso	OfficeHolder	José Manuel	Durão Barroso

Table 3.1.: A CS represented as table

3.3. Structure Detection and Labeling Process

Transforming RDF data into a relational schema is a two-step process. The first step is transforming the RDF data into an UML-like data model, e.g., containing class hierarchies and multivalued properties. Afterwards, the transformation into the relational model requires eliminating hierarchies and moving multivalued attributes into an extra table. The process described in this section performs the first step, creates the final set of CS's that form the basis for the targeted relational schema.

The goal is to represent as much data as possible in as few CS's as possible. Irregular data that does not fit well into the structure of the majority of the data is thrown away.

In the beginning, structuring and labeling where two independent steps that were executed after each other to produce the final set of labeled relational structures. This can be seen in figure 3.1.

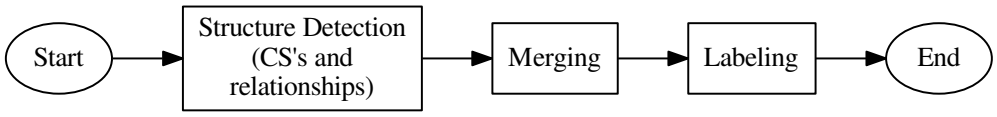


Figure 3.1.: Flow chart for structuring (old version)

We soon discovered that merging without taking semantic information into account led to bad structures:

- More general concepts are merged into more specific concepts. For example, most mammals described by the DBpedia dataset belong to the category `Mammal`. `Mammal` has sub-concepts to further specify certain species, however this list is not complete. The sub-concepts introduce new properties (e.g., `jockey` for the sub-concept `RaceHorse`). As a result, the property set of most mammals is a *subclass* of the sub-concepts' property sets. One of our merging rules was to merge subset CS's into superset CS's. Because of that rule, all mammals that did not belong to a sub-concept were nevertheless added to a sub-concept because of the subset-superset relation of their property sets. The resulting CS's contained many NULL values (e.g., in the `jockey` property). In addition, the labels assigned to the resulting CS's were chosen based on the property sets and therefore represented the sub-concepts only (e.g., label `RaceHorse` for the CS that contained the property `jockey`), ignoring the fact that the majority of mammals does not belong to a sub-concept and should therefore be labeled `Mammal`. Finally, another disadvantage was the randomness of CS merging: A subject of concept `Mammal` could have been merged into any of the sub-concept CS's, this was dependent on the order of CS detection only.

  As this example shows, merging subset CS's into superset CS's without further conditions results in CS's containing various, mixed concepts and therefore low label quality. However, merging subset CS's into superset CS's is a promising way to join similar CS's. In general, two different type of subset-superset CS relationships exist: *i)* different subjects of *one* concept differ in some properties (e.g., no property `firstWin` for motorsport racers without any won races) *ii)* instances of a concept and a sub-concept differ in some properties (e.g., the only difference between `Mammal` and `RaceHorse` is the property `jockey`) Merging should be applied in the first case, but has to be avoided in the second case. To differentiate these two cases, we add a labeling phase before applying the merging techniques. Using the CS labels, we know whether a subset CS should be merged into a superset CS, or not, as described in section 5.1.

- Some properties occurred for nearly every subject (e.g., `wikiPageID` for DBpedia subjects), hence decisions on merging should not be based on these properties only. We overcome this issue by introducing *tf-idf*-based [28] measures to evaluate the ability of each property in a CS to *discriminate* its CS from others.

- The resulting schema contained many tables with same labels, because "having same labels" was not recognized as an important indicator for merging. By changing the order of steps in the structuring process, we can use the merging phase to merge CS's based on their previously computed labels. This step is explained in section 5.1.

- Foreign keys between the resulting relational tables could not be enforced because foreign key columns where often pointing to more than one table. Relationships between CS's were not used for structuring and labeling purposes, but for the final physical data transformation only. We therefore introduced CS

relationships as important data source for labeling and merging. Besides improving structure and labels, this also leads to a schema that is compliant with the relational model. CS Relationships are introduced in section 3.7.

Because of these drawbacks, we decided to exchange information between the structuring and labeling steps. When we integrated semantic information for the labeling part, it was obvious that this information would also improve structure detection quality. Therefore, we interlinked the structuring and labeling phases to make semantic information available in a structuring phase. The resulting process overview is shown in figure 3.2. This led to a complex mixture of steps, but significantly improved label quality. This interlinking did not only help to improve structures, but also led to more discriminating labels and therefore better label quality.

Figure 3.2.: Flow chart for structuring

Results of both versions of the process are presented in section 7.3.3.

3.4. Goals and Criteria

The overall goal of *MonetDB/RDF* is to build a fast and efficient RDF store over *MonetDB*, by transforming RDF data to relational data. By doing so, the advantages of well-researched RDBMS can be used, e.g. efficient join processing and the well-known query language SQL. Furthermore, the resulting relational schema should be presentable to users to let them benefit from the induction of schema achieved by our algorithms. Hence, *MonetDB/RDF* also aims for improved ease of use and comprehensibility of RDF datasets by extracting the underlying *schema* in RDF data.

The structuring and merging phases of the *MonetDB/RDF* algorithms aim for a small, condensed relational schema. The main goal of the labeling process is the generation and assignment of *human understandable* names to tables, attributes and foreign keys. Using these names, the majority of data should be grasped fast and simple by the users.

We establish the following criteria for structuring and merging:

Small Schema As much data as possible should be represented in as few tables as possible. Within this trade-off, no factor must be overrated or ignored, as this would lead to an unbalanced result.

Quality vs. Complexity *MonetDB/RDF* follows a fuzzy *80% approach*. Outlying data is not included and summation is used to provide an overview over large RDF datasets. To act on this approach, the algorithms have to trade quality off against complexity, leading to a simple-yet-effective approach to structuring, merging and also labeling.

Level of Detail of Tables When merging similar tables into one, one has to be aware of the aspired level of detail. For example, two tables about long-distance runners and sprinters should probably be merged together, whereas merging the tables about singers and actors might not be wanted by the end users.

Table names, as the result of the labeling process, are also judged by some additional criteria: Geraci et al. [9] mention three factors for high label quality: well-formedness, descriptive power, and discriminatory power. Using these and some more quality indicators, we come up with the following list of criteria for labels:

Descriptiveness Table names have to describe the content of the table at the best possible rate. Attribute names have to characterize the content of the columns and the binary relationship between the subjects and the attribute (cf. [31]). Foreign key names have to describe the role of the table they point to.

Headwords The names will usually consist of one headword, e.g., `Country`. Lists of keywords (for example `City-states, Monarchies, Middle Eastern Countries`) have to be avoided because in that case the users would have to find a hypernym on their own to understand the contents of the table (cf. [30]). This criterion includes the *well-formedness* demanded by Geraci et al. [9].

Uniqueness Column names have to be unique within a table. If possible, there should not be multiple tables with the same name. Instead, tables with the same names should be merged (as described in section 5.1). Similarity scores describe the rareness of possible table names within the schema and should therefore be used to achieve unique table names.

Level of Detail of Table Names Table names should be as detailed as possible. For example, a table containing soccer players could be named `Person`, but `SoccerPlayer` would give a better description of the content.

These criteria ensure high quality of tables and labels. The structuring/merging and labeling functionality will be included in the product *MonetDB*. For *production readiness*, additional goals arise:

No Internet Connection The process may not rely on an Internet connection to load external data or check URIs because database servers in production environments usually have no (full) access to the Internet. Instead, an interface for loading external metadata into *MonetDB/RDF* has to be provided to the users.

Fast Transformation of RDF Data into Relational Data On the one hand, structuring, labeling, and merging is done once after data loading and therefore directly influences to *data-to-query* time, i.e., the initial startup time from data loading until the first query is answered. It should therefore be kept short. On the other hand, users do expect waiting time when loading big datasets, so instance response is not required. However, the user should be kept informed about the progress of data loading, structuring, and labeling.

Few user input To meet *MonetDB/RDF*'s goal of being *self-organizing*, the software prototype should not require additional configuration or decisions by users. Only few user interaction is acceptable for the whole process. If possible, a (basic) structuring and labeling should be possible without any actions by users.

Separation from Other Modules The *MonetDB/RDF* algorithms have to be kept separate from other parts of *MonetDB* as *MonetDB* is generally unaware of semantics. The resulting SQL schema has to work with all parts of *MonetDB* without further effort.

SQL Compatibility As the resulting SQL schema will be accessed through an SQL interface, the schema have to be SQL standard conformable, e.g., table names have to be unique within a SQL schema, and many-to-many relationships between tables must be represented using an additional table rather than using multivalued properties.

The fulfillment of these criteria is discussed in section 7.5. In the following sections, the steps of the structuring process are described in detail.

3.5. Loading RDF Data

MonetDB/RDF loads RDF data in the common *N-triples format* introduced in section 2.3.3. During loading, duplicate and malformed triples are removed.

MonetDB/RDF stores the data into a simple *SPO table*. A SPO table has three columns (subject, predicate, object), hence one triple is stored per row. The table is ordered by subject, then predicate, then object. The effort of sorting taken in this steps reduces the complexity in the following steps, as necessary information can be gathered in only one pass over the SPO table.

Strings are stored using *dictionary encoding*, as introduced in section 2.1. As RDF data contains many duplicated strings (i.e., subject and property URIs), dictionary encoding significantly reduces storage space. Processing RDF data triples requires comparisons of these strings. By using dictionary encoding, strings are represented by numbers, which simplifies checking equality of two strings.

3.6. Exploring CS's

Exploring CS's takes one pass over the SPO table only, exploiting the previously described sorted SPO table: Because of the subject order, all information for one subject is collected in one run. The predicate order leads to a consistent order of predicates within all predicate lists.

Leveraging the consistent predicate order, comparing predicate sets can be simplified. Instead of comparing all predicates of two subjects, the list of properties can be hashed, and the comparison can take place on these hash values. A hash table is used to store all CS's that have already been created. For each new subject, the hash value is computed, and the corresponding CS is found using a hash table lookup. If no CS is found, a new CS consisting of that subject is created and added to the hash table. Of course, one has to take care of hash collisions.

3.7. Exploring CS Relationships

A relationship between CS A and CS B exists if an object value (of data type *URI*) in CS A occurs as an subject in CS B or vice versa. An example of this situation is shown in figure 3.3.

```
<European Parliament election 2009>, <majorityLeader>, <José Manuel Barroso>

<José Manuel Barroso>, <givenName>, "José Manuel"
```

Figure 3.3.: Example of object URI `<José Manuel Barroso>` also being a subject URI (i.e., describing a resource)

Again, frequencies of these relationships are stored. Relationships are eliminated if they are *infrequent*, meaning that the percentage of instances in CS A referring to CS B via a specific property is below a certain percentage. Infrequent relationships would bloat the schema by adding too many relationships. Therefore they will not be used in the further process.

The relationships between CS's will be transformed to foreign key references in the final relational schema. It is still possible that one property of a CS links to multiple other CS's. As this is not possible for foreign keys, further transformations of the CS's are done in the following steps. First, it is possible that the different CS's a property links to should be joined (cf. section 5.1). If there are still multiple link destinations left, the property is split (cf. section 5.4).

3.8. Dropping Irregular Data

Only CS's with enough instances are kept for the next steps. The threshold for keeping these CS's is the only parameter defined by the user, as the algorithm runtime depends on the number of CS's used throughout the structuring process.

CS relationships help identifying *important* CS's. A CS that is referred to by many instances is considered important and is kept even if it does not have enough instances. The numbers of instances that reference a table are not only computed for *direct* references, but also for *indirect* references over one or more intermediate hops. By also taking indirect references into account one can also recognize important tables that are only referenced by other small but important tables. If this rule had not been used, small dimension tables (e.g., the NATION table in the TPC-H schema) would have been dropped in the structuring process.

Dropping infrequent CS's is the first time we are omitting *irregular data* to increase the structuredness of the remaining data.

After the CS's have been found and the relationships have been detected, labels are assigned to the CS's. This is described in chapter 4. Afterwards, the CS's are refined by merging them, as chapter 5 describes.

4. Labeling Structures in RDF Data

Assigning labels to CS's is the second main part of this thesis. Using the list of CS's, their relationships, and additional ontology information, a ranked list of possible names for each CS is created and the best candidate is chosen and assigned to each CS. This chapter describes how human readable names are assigned to *CS's*. Labeling of *properties* is done at a later point in time, this is described in section 5.2.

The rest of this chapter is organized as follows. The main ideas and terms for labeling are introduced in section 4.1. Section 4.2 discusses previous work on labeling different kinds of data. Section 4.3 describes the data sources used to generate the labels. The extraction of human readable headwords from URIs is described in section 4.4. In sections 4.5 to 4.9, the extraction of label candidates is described as well as the assignment.

4.1. Idea

Figure 4.1 shows a schema before (top) and after (bottom) labeling. The column names can directly be created by looking at the URIs (cf. section 4.4), whereas the table name generation requires more effort. The process to generate the CS's at the bottom is described in sections 4.5 to 4.9.

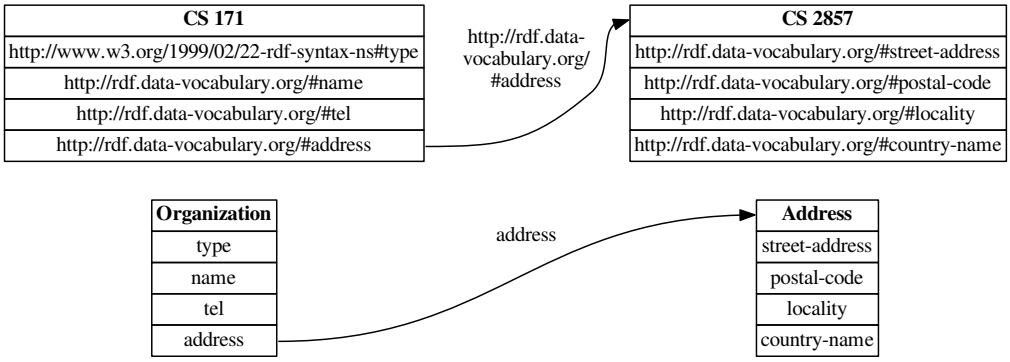


Figure 4.1.: Schema before (top) and after (bottom) labeling

Different parts of a SQL schema have to be labeled:

Tables Table names are the most important names within a SQL schema. They are the first labels a user looks at when trying to understand a schema. Table names usually contain of a headword describing all the objects within a table. There are also SQL tables not describing any objects but linking objects in a many-to-many relationship. Those tables are not considered here as CS's can contain multivalued attributes hence not raising the need for relationship tables. However, when transforming

the intermediate schema to relational tables as described in section 5.4, multivalued properties will be separated into an additional table, but is not labeled as it does not represent a *concept*.

Attributes Attribute names describe the content of data columns and are certainly necessary to understand the meaning of the values and the relation of this data to the subjects.

Foreign Keys Foreign key relationship names describe the *role* of the referenced table, e.g., within a table `book`, the relationship `author` describes the role of the linked table `Person`. In the relational schema foreign keys are modeled the same way as attributes.

The resulting names have to be conforming to the SQL standard. For example, table names have to be unique and labels must no contain spaces. Additional criteria for the labeling process are established in section 3.4.

The main data sources for labels are *Uniform Resource Identifiers* (URIs). URIs usually contain a good description of their meaning, e.g., `https://creativecommons.org/ns#license` describes a license. As predicates in RDF triples transform to columns in relational tables, predicate URIs are the main candidates for attribute names. Another useful data source are *ontologies*. Ontologies describe the structure the triples were created according to and provide names for these structure, hence being a helpful resource for naming structures.

Table labeling requires multiple techniques and data sources. However, they should contribute to *MonetDB/RDF*'s overall goal of *Data Exploration* which leads to a fuzzy approach using heuristics and ignoring outliers. The following techniques are used to create table name candidates: Many tables contain one or more attributes named `type` or the like. A type attribute refers to the *concept* instantiated by a subject. These type attributes usually contain only few different values because the set of attributes within a CS roughly defines the concepts described by the CS, though only subjects with specific type values form a table. The values of type attributes therefore make up good table name candidates. For instance, a table with attributes `birthDate` and `deathDate` might have type values `Artist`, `Politician`, and `Person`, but no values describing different concepts, like `Food` or `City`. A second source for generating table name candidates are *ontologies*. Ontologies usually contain classes that can be matched to tables by comparing the attributes of the ontology classes and the attributes of the tables. The third and last source for table name candidates are the URIs of *incident foreign keys*. Incident foreign keys are links from another table pointing to this table. For example, a table describing books has an adjacent (outgoing) foreign key named `author`. `Author` therefore is a table name candidate for the target table of the foreign key. Out of the list of table name candidates, the final table name has to be chosen. A scoring and weighting algorithm therefore has to be applied to the list of table name candidates.

4.2. Related Work

Labeling is researched on a variety of data including HTML table data, clustered data, and search queries. It is an important task within *Information Retrieval*. Because RDF data does not contain sentences or other sequences of words, most of the research on entity labeling and Information Retrieval [3, 14] cannot be applied. Instead the main source for labeling will be the URIs that are used to identify subjects and predicates.

Neumayer et al. [21] describe *Semantic search with (almost) no semantics* to improve the quality of ad-hoc search results in the Semantic Search Challenge whose data is in RDF format. To achieve better results, they introduce three simple techniques: First, the authors assign higher weighting to fields which names end with `name`, `title`, or `label` to treat title data preferentially. Preference of some predicates will not be applied to our system as all attributes are treated equal in SQL tables. Second, Neumayer et al. [21] favor data from trusted sources (DBpedia in their case). Favoring of data from specific sources is not needed in our task as the user defines the set of data he wants to explore himself by loading it into *MonetDB*. The last

technique introduced by the authors is URI preprocessing to extract additional textual information. Their simple extraction method is to *extract the last part of the URI* (after the last slash). This idea is proven by Neumayer et al. [21] to significantly improve the quality of search results over their baseline without URI preprocessing. A slightly more sophisticated version of the extraction technique will be part of the labeling process in *MonetDB/RDF* as described in section 4.4.

Venetis et al. [31] recover *semantics of Web tables* by creating attribute names from hypernyms of the columns' contents. The hypernyms are created using an external *isA database*. Their approach has to go without whole sentences and is therefore comparable to our task. They mention that their labels often do not occur in the table itself. We experienced the same and therefore decided to include external data (ontologies) for the labeling process. The authors describe that many tables consist of a *subject column* containing the subjects, and other columns that have a *binary relationship* with the subject column. RDF has a similar view as all objects are linked to subjects using a binary predicate. In the relational model, relationships are not modeled as links, but as attributes. For foreign keys, these attributes contain special values, for links within tables there is no explicit documentation of this fact. In the relational tables we created from RDF data it stands out that *verbs* occur as column names, e.g., `has_creator` or `operatedBy`, instead of *nouns* that would probably have been used if the data was modeled as relational data in the beginning (`creator`, `operator`). Venetis et al. [31] mention poor recall because of poor quality of Web tables. As an example, they refer to tables that are used for layout purposes only. As relational tables have a fixed structure, we assume better recall when the approach is applied to relational data.

Ranking label candidates in *hierarchical clusters* using a linear model has been studied by Treeratpituk and Callan [30]. Their approach takes hierarchy into account by computing term frequencies for clusters and parent clusters. The authors mention that a headword is often a more useful label than a list of terms because in the latter case a human has to infer the general term from the list. The same applies for our labeling process where the list of type values often does not describe the overall topic of a table but contains descriptions of parts of the data (e.g., list `Day_school, Independent_school, State_school` does not explicitly contain the *concept* represented by the CS, which is `School`). Like Venetis et al. [31], Treeratpituk and Callan [30] show that documents often do not contain *self-descriptive terms*. Instead, they use anchor texts (pieces of texts on and next to hyperlinks to the document) to find better labels. We use a similar idea by inferring names from *foreign key names* which are the relational equivalent to hyperlink texts. Hyperlink descriptions can contain spelling errors, synonyms, and generally all issues of free-form text whereas foreign key names are built from the set of URIs provided by the ontology. Therefore frequencies of terms used in foreign keys can easily be calculated and used for measuring descriptiveness of certain terms. A linear model is used by Treeratpituk and Callan [30] to rate the descriptiveness of label candidates which is comparable to our approach using tf-idf-like similarity scores. However, their linear model needs to be fitted with training data what is not possible in the *MonetDB/RDF* scenario that tries to avoid user interaction.

Some research has been done on finding subtopics (i.e., categories, facets, or senses of a topic) of search queries used for *Search Result Diversification*. Wang et al. [33] propose a clustering algorithm based on term frequency for extracting subtopics and automatically labeling them. The authors compute the so-called *core term* of a cluster by applying a tf-idf-like relevance measurement. They claim that the core term has to be expanded to represent the contents of the cluster better and to look more like a search query suggestion. To expand the core term into a *core phrase*, words are appended to the core term left and right alternatively if they exceed a co-occurrence threshold. The approach by Wang et al. [33] uses more *context* than available for our labeling task, their approach of extending a term to a phrase is therefore not applicable for *MonetDB/RDF*. To our mind using one headword only instead of half sentences is sufficient for our task as we label *objects* instead of *pieces of connected information*.

4.3. Data Sources

Data sources can roughly be divided into two categories, *internal* and *external* data sources. The *internal* data source for label generation are the URIs in the RDF triple data. In the RDF world, URIs are used for subjects, predicates, and (partially) objects. As predicates, URIs are candidates for attribute labels. As objects, URIs might mark relationships between CS's and are therefore candidates for table names (for the table names the relationship points to).

In general, *external* data sources should not be used because we may not rely on an Internet connection. Inclusion of the large external data sources such as dictionaries is also not an option because label generation is just a minor feature of the *MonetDB* database system and therefore should not require that many resources. Ontologies are small and a small number of ontologies covers the vast majority of RDF, as shown in section 4.3.2. Ontologies are therefore the only external data source that will be used. However, the user has to take care of integrating these *external* ontologies into *MonetDB/RDF* by loading them.

4.3.1. URIs: Type Values and Foreign Key Names

URIs (cf. section 2.3.4) are used as identifiers with RDF datasets. URIs usually contain a rough description of their meaning, they are *human readable*. This meaning can be extracted and used for labeling. As URIs are taken from ontologies, we can be sure that the same meaning will be represented by the same URI, so we do not have to take care of spelling errors, synonyms, and other difficulties in analyzing texts.

Within our labeling process, URIs are used twice: First, we exploit self-descriptiveness that is available in many datasets by analyzing the values of properties named `type` or similar (e.g., `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`). Second, we store the property URIs that values link to other subjects. These property URIs describe *relations* between subjects, or – in relational terms – name foreign keys. Using URIs is the intuitive way of analyzing RDF data because they are a central concept of the RDF idea and are already available within the dataset.

4.3.2. Ontologies

Ontologies, as described in section 2.3, contain vocabulary to describe RDF data. RDF datasets that use a certain ontology can be structured and labeled more easily if the additional information on classes and hierarchies in the ontology is exploited.

For example, most of the RDFa data (*RDF in Attributes*, small snippets of RDF data on Web pages) contained in a segment of the Common Web Crawl [14] uses only few ontologies (cf. table 4.1). Furthermore, also the statistics of the Linking Open Data community project[15], *semanticweb.org*[16], and *prefix.cc*[17] imply that a small number of ontologies covers the vast majority of RDF data. We therefore decided to include the most popular ontologies (according to the Common Web Crawl, *semanticweb.org*, *prefix.cc*, and the Linking Open Data community project) in *MonetDB/RDF* when doing our experiments. We think this approach matches user behavior best because it covers the vast majority of data with little loading effort.

[14] Common Web Crawl `http://webdatacommons.org/`
[15] LOD Community Project `http://lod-cloud.net/state/`
[16] Semantic Web `http://semanticweb.org/`
[17] prefix.cc `http://richard.cyganiak.de/blog/2011/02/top-100-most-popular-rdf-namespace-prefixes/`

Ontology	Occurrences
Facebook Open Graph Protocol	53%
Google Data Vocabulary	19%
RDF Schema	14%
DublinCore Terms	3%
GoodRelations	2%

Table 4.1.: Top 5 ontologies in the Common Web Crawl
Open Graph is used to link Web pages to Facebook, e.g., by including a Like button.
Data Vocabulary is markup code that is supported by the Google search engine.
RDF Schema is basic vocabulary such as `comment` or `label` offered by *W3C*.
DublinCore Terms describes resources, e.g., books, videos, or Web pages.
GoodRelations [12] offers e-commerce markup.

The most popular ontologies according to the afore-mentioned sources include DublinCore Terms, Facebook Open Graph Protocol, Simple Knowledge Organization System (SKOS), Friend of a Friend (FOAF), Geo, Geonames, GoodRelations, and the "meta ontologies" `owl`, `rdf`, and `rdfs`.

4.4. URI Shortening

Attribute labels are simply extracted from predicate URIs using *URI shortening*. URI shortening reduces the URIs to one headword according to the goals defined in section 3.4. Predicate URIs, e.g., `http://xmlns.com/foaf/0.1/name`, usually contain the predicate information as last part. The heuristic described by Neumayer et al. [21] uses the part after the last slash as a predicate name. However, some ontologies use a slightly different scheme, e.g., an URI part marker (hash sign #) (e.g., `https://creativecommons.org/ns#license`) or dividing the predicate information by slashes (e.g., `http://search.yahoo.com/searchmonkey/media/duration`). The *URI schemes* of the most popular ontologies are included into *MonetDB/RDF* to ensure correct and complete extraction of relevant parts for these widely-used ontologies. As a fall-back for ontologies unknown to our system, the heuristic described by Neumayer et al. [21] is used.

4.5. Labeling Process

The RDF data itself, the information derived from the CS structuring algorithm introduced in chapter 3, and the data sources described in section 4.3 form the input for the labeling process described in figure 4.2.

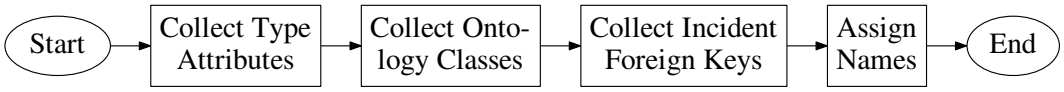


Figure 4.2.: Flow chart for labeling

Starting with no table names we include techniques for table name candidate extraction bit by bit, analyze the candidates each techniques selects, and incorporate more sophisticated techniques including foreign data

sources. Along the way we define reasonable thresholds and identify the pros and cons of each technique, resulting in the assignment algorithm described in section 4.9.

The following sections 4.6 to 4.8 describe how table name candidates are extracted from the data sources. Afterwards, a table name is chosen out of these candidates, as described in section 4.9.

4.6. Collection of Type Values

Many ontologies include a predicate called `type` or similar. The values of these attributes describe the *concept* the subject belongs to.

4.6.1. Loading of Type Property List

A list of common type predicates from the most popular ontologies is stored within *MonetDB/RDF*. These type properties are listed in appendix C.

4.6.2. Process

During the extraction phase, our algorithm loops through all subjects and extracts the values for the type predicates. Type values are either URIs linking to an ontology or string literals. String literals in RDF can contain a marker for their language (e.g., `"website"@en-us`) which is ignored for this analysis as the type values are the same for every language. Using the frequencies of the names, a histogram per CS and per type predicate within the CS is created. An example can be found in table 4.2.

Hierarchy Level	Type	# of Subjects	% of Subjects
0	Thing	1084	100
1	Species	1084	100
2	Eukaryote	1084	100
3	Animal	1084	100
4	Bird	546	50
4	Mammal	284	26
4	Fish	185	17
4	Amphibian	27	2
4	Reptile	15	1
4	Insect	11	1
4	Crustacean	10	1
4	Mollusca	3	0
4	Arachnid	1	0

Table 4.2.: Type property values in a CS about animals

Algorithm 4.1 specifies the process in detail. We noticed that type attributes tend to be *multivalued* properties. This is also the case in table 4.2. In DBpedia, the type values represent the whole class hierarchy a subject belongs to. For example, a subject describing a soccer player has type values `SoccerPlayer`, `Athlete`, `Person`, `Agent`, and `Thing`.

In a first version of the type value algorithm, we were not aware of this fact and therefore had two problems: *i)* we reached only low percentages that often did not exceed our threshold because we divided the frequencies

Algorithm 4.1: Collecting type value statistics

input : triple data, list of type attributes
output : type value frequency per CS and type attribute

```
/* compute histogram of type value frequencies */
forall the subjects do
    forall the type attributes do
        get the CS the subject belongs to
        collect all type values of that subject and that type attribute
        forall the type values do
            if type value already exists for this Cs and this type attribute then
                increase bucket counter by one
                else
                    create bucket
                    set bucket counter to one
                end
            end
        end
    end
end

/* sort type values */
forall the CS's do
    forall the type attributes do
        sort list of type values descending by frequency
    end
end

/* assign percentages */
forall the CS's do
    forall the type attributes do
        forall the type values do
            percentage of this type value is frequency of that value divided by the number of subjects in the
            CS
        end
    end
end
```

by the number of type values rather than by the number of subjects in the CS (e.g., for a table containing persons, the histogram looked like this: 33% `Person`, 33% `Agent`, and 33% `Thing`, and even worse for deeper hierarchies), *ii)* we got too generic name candidates (e.g., `Thing` is the root of the DBpedia hierarchy and therefore the most common type value).

In the second version, we therefore decided to *i)* change the computation of the percentages and *ii)* change the rules for choosing a table name candidate out of the list of type values. The percentages assigned in the algorithm do not sum up to 100% per CS and type attribute anymore, but to a higher value, e.g., 100% `Thing`, 99% `Agent`, 90% `Person`, because we denote the *number of subjects in a CS* as 100%, and not the sum of all type values found within a CS. This also implies that the value with the highest per cent value is not the best CS name candidate because it is too generic. We therefore use a high threshold (e.g., 80%) and take a closer look at all type values that exceed this threshold: By adding ontology hierarchy information, we get to know the *hierarchical level* of each remaining type value, and therefore how *specialized* each value is. We choose the most-specific value out of the remaining as table name candidate. The process of choosing the best label is described in section 4.9.

4.6.3. Discussion

On the one hand, type values tend to be too specific and describe only parts of the table contents. This might be caused by the low quality of self-description as shown by Treeratpituk and Callan [30] or because several small concepts are merged into one CS that represents a parent concept then. On the other hand, type values can also be too generic, e.g., (geological) `Feature` if villages are described, or `Thing` for DBpedia subjects. After recognizing the *multivaluedness* of type values and changing the way we choose the best table name candidate, we do achieve the wanted level of detailedness in the candidates.

If type value information is available, a candidate that exceeds the threshold is found almost every time. Therefore, CS's that mix up different concepts (e.g., `City` and `Animal`) end up with a very general candidate such as *Thing*. However, these CS's do not represent *one* concept what makes it hard to label them, so `Thing` is the best name we can achieve.

Some tables do not have type information, we therefore need additional labeling techniques. Two additional label techniques are described in sections 4.7 and 4.8

4.7. Collection of Ontology Classes

Matching attribute sets of SQL tables with attribute sets of ontology classes requires external data. The use of external data should be avoided in the labeling process because it introduces space and licensing issues when this data is shipped with *MonetDB*, or would require additional loading steps when this data is needed. However, ontologies provide very good table name candidates as RDF datasets are usually created according to ontologies. Ontologies offer a hierarchy that helps to overcome the main problem of type value candidates as described by Treeratpituk and Callan [30]: Self-descriptions tend to be on a lower hierarchy level than useful for table labeling. Because of the relatively small size of ontologies and the big help they are in labeling, we decided to use this external resource.

4.7.1. Loading of Ontology Data

MonetDB/RDF makes use of class names, attribute lists per class and the class hierarchy to find the best table name candidates using ontologies.

Ontologies can be loaded into *MonetDB/RDF* by users using a generalized format. For each class in the ontology, URI, class name and – if available – superclass URI are needed. In addition, pairings of property URIs and class URIs where they belong to are necessary. This includes *inherited* attributes, meaning that a class inheriting from another class also inherits the property sets of the superclass. We therefore use two tables to store this information, one with class and class hierarchy information, the other one with property information. Appendix B contains the SPARQL statements for transforming the well-known ontology DBpedia to this format. Other ontologies require similar code for transformation. *MonetDB/RDF* offers an interface for loading transformed ontologies before applying the structuring/merging and labeling algorithms. In the future, a more sophisticated uploading procedure might be implemented to allow ontology loading for users that are not familiar with transforming ontologies (cf. section 8.2).

4.7.2. Similarity

A measure for the *similarity* and *discriminatory power* of tables/classes is used to support the goal of highly descriptive and unique names. Similarity is needed when comparing a table to an ontology class. In an early version of the labeling process a table was considered similar to a class only if the set of columns

of the class was an exact subset of the attributes of the ontology class. However, this strict criterion led to non-intuitive results due to errors in the data and the ontologies as well as loss of precision during the CS creation algorithms. For example, the DBpedia ontology defines that railway stations do not have an address, but the DBpedia data provides address information for stations. Another examples are species in DBpedia. In the dataset, species have a property `class`, but the ontology uses the Latin word `classis` instead.

We therefore decided to use a *tf-idf*-like [28] similarity score instead of the strict subclass criterion. tf-idf (*term frequency*, *inverse document frequency*) is a technique used in Information Retrieval. The measure expresses how frequent a term is used in a document in comparison to how often it is used in other documents. tf-idf hence expresses how good a term *characterizes* a document (term is frequent in that document, but infrequent in others). For every property in the RDF dataset, we compute a tf-idf-like score. By combining these scores, we get to know how good the *set of properties* describes a CS and differentiates it from other CS's in the dataset.

The term frequency (tf) is 1 for every property, because properties occur only once per CS:

$$\text{tf}(p) = 1. \tag{4.1}$$

The inverse document frequency is defined as the logarithm of the total number of documents divided by the number of documents the term occurs in:

$$\text{idf}(p) = ln\left(\frac{\text{\# documents}}{1 + \text{\# documents with property } p}\right). \tag{4.2}$$

For tf-idf, tf and idf are multiplied, resulting in:

$$\text{tf-idf}(p) = \text{idf}(p) = ln\left(\frac{\text{\# documents}}{1 + \text{\# documents with property } p}\right). \tag{4.3}$$

There are two possible interpretations for *document* in our scenario: characteristic set (CS) and ontology class. An advantage of the first option, counting in how many *CS's* a property is used, is the independence from the ontology: If the ontology defines properties that do not occur in the dataset, these values get a tf-idf score assigned that can never be reached by the actual data. In addition, using the second option, counting in how many *ontology classes* a property is used, has the disadvantage that the form of the ontology influences the tf-idf scores: If an ontology uses many sub-classes to describe a concept such as `Person`, properties of this concept (e.g., `birthDate`) occur often within the ontology and therefore get only low tf-idf scores. However, these properties are good indicators for CS's about `Persons`, but are seen as weak indicators because they occur so often within the ontology.

But also the second option, using ontology classes as *documents* for tf-idf computation, has some advantages that cannot be denied: Properties that are not part of the loaded ontologies are ignored for tf-idf computation. Later in the process of computing ontology-based label candidates, when the tf-idf scores are summed up to compute similarity between CS's and ontology classes, only the second option (ontology classes) allows for high similarity scores because these non-ontology properties have no influence on the similarity computation. We therefore implement the second option:

$$\text{tf-idf}(p) = \text{idf}(p) = ln\left(\frac{\text{\# ontology classes}}{\text{\# ontology classes with property } p}\right). \tag{4.4}$$

tf-idf provides an intuitive approach for similarity calculation: Attributes that occur seldom within the ontology are good markers for specific ontology classes (*discriminatory power*) whereas widely used attributes (e.g., `name` or `address`) are not relevant for deciding whether a CS and an ontology class describe the same

concept. The tf-idf score of a property p, that occurs in $freq(p, o)$ ontology classes, in an ontology o with number of classes $numClasses(o)$ is defined as:

$$\text{tf-idf}(p) := ln\left(\frac{numClasses(o)}{1 + freq(p, o)}\right). \tag{4.5}$$

A property that occurs in every ontology class would therefore get tf-idf score slightly below zero, and rare properties get scores up to slightly below $ln(numClasses(o))$.

4.7.3. Process

Per CS attribute, the tf-idf score is computed. The score defines how important an attribute is for the table it occurs in by counting how often it occurs within the ontology, as described afore-head.

The list of attributes of a table has to be grouped by ontology, because multiple ontologies can be used within a table. Per CS attribute, the comparison with the ontology attributes is done and all ontology classes that contain this CS attribute are added to the list of candidates of the CS attribute.

There are three possibilities for *normalizing* this summed tf-idf score. The first option is to compute the summed tf-idf for the *ontology class*, and to divide the score by that baseline. By doing so, a high normalized score is reached if and only if most of the attributes in the ontology are available in the CS. The second option is to use the summed tf-idf for the *CS* as baseline. This emphasizes the role of the CS, and high normalized scores are reached if and only if most of the CS properties belong to *one* ontology class. The third and last option is using a mixed baseline by multiplying the square roots of both baselines. Our experiments showed that it is common that the ontologies define many more properties than available in the dataset. It is therefore best to use option number two. This also ensures that a CS only represents *one* concept, as CS's that mix different concepts would not reach a high normalized score.

For all ontology classes that are on the list of candidates of one or more CS attributes, the sum of the tf-idf scores of the corresponding CS attributes is computed. The maximum tf-idf sum is the sum of all CS attributes. The score of each ontology class divided by the maximum score is a value between 0 (no common attributes) and 1 (table is an exact subset of this ontology class, 100% match).

If the value exceeds a certain threshold value (e.g., 0.8), the ontology class and the table are considered *similar* and the ontology name is added to the list of name candidates. The list of candidates is then compared to the ontology class hierarchy: If both a class and its superclass are on the list of candidates, the child class is removed from the list because its parent is a more general fit. Algorithm 4.2 specifies the process in detail. The output of the algorithm is a list of ontology class names that could be a name for the CS. It is ordered, class names with higher tf-idf score are listed first.

4.7.4. Discussion

Not all ontologies can be included into *MonetDB/RDF*, and not all missing ontologies will be added by users. Not all tables contain enough columns for a reasonable ontology lookup, or the attributes are spread over too many ontologies.

In comparison to type values, ontology class names are on a higher hierarchy level. Leading to one headword per table, using ontology class names instead of type values reduces the accuracy of table description if subclass attributes are missing. For example, a table containing soccer players will be labeled as `Athlete` if the tuples do not contain soccer-specific attributes.

An additional issue is added by the CS merging algorithm. If the CS structuring algorithm joins tables that ontology-wise contain different data, the resulting attribute set does not match any ontology class. CS

Algorithm 4.2: Collecting ontology class names

input : list of CS's, list of ontologies
output : list of ontology class candidates per CS

```
/* compute tf-idf score per CS attribute */
forall the ontology classes do
    forall the ontology class attributes do
        if bucket for this attribute URI exists then
            increase bucket counter by one
        else
            create bucket
            set bucket counter to one
        end
    end
end
forall the buckets do
    compute tf-idf(bucket) = ln ( numClasses / (1+ bucket counter) )
end

/* collect ontology class names */
forall the CS do
    group CS attributes by ontology
    forall the ontologies do
        /* collect candidates per attribute */
        forall the ontology attributes do
            forall the CS attributes that belong to this ontology do
                if ontology attribute URI equals CS attribute URI then
                    add ontology class URI to the list of table name candidates of this CS attribute
                end
            end
        end
        /* compute similarity score for all candidates */
        forall the ontology class URIs that occur on one or more lists of candidates do
            sum tf-idf scores of each CS attribute that supports the ontology class
        end
        calculate the optimal tf-idf score (sum all CS attribute tf-idf scores that belong to this ontology) forall
        the ontology class URIs that occur on one or more lists of candidates do
            normalize score (tf-idf score divided by optimal tf-idf score)
        end
        /* remove candidates depending on similarity and hierarchy */
        remove ontology class URI from list of candidates if its tf-idf score is below a given threshold
        remove ontology class URI from list of candidates if one of its superclasses is also on the list and has
        a higher or equal tf-idf score
    end
end
```

structuring therefore has to be conservative to avoid such situations. Instead it is better to introduce a merging phase after labeling, as discussed in section 3.3, that includes the semantic information retrieved during labeling. This second structuring phase is described in chapter 5.

After matching with ontologies, some tables still have no table name candidates. We therefore need another source for candidates: foreign key names, that are introduced in section 4.8.

4.8. Collection of Incident Foreign Keys

When creating SQL schemas manually, foreign keys are often named like the table they point to, e.g., `order_id` points to the order table. We therefore suggest looking at foreign key names in RDF data as source for table name candidates.

4.8.1. Process

Unlike relational tables, relationships in CS's may point to more than one other CS. We therefore store how many links of a foreign key point to a specific table. Values below a given fraction are ignored because they are probably dirty data. Using infrequent values as labels would not support the goal of *Data Exploration*, where the majority of data should be grasped fast and simple by the users. Figure 4.3 shows different frequencies of *adjacent* foreign keys.

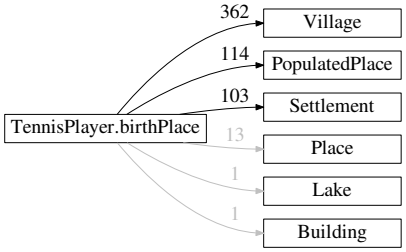


Figure 4.3.: Different frequencies of adjacent foreign keys. The gray links are infrequent and are therefore ignored when inferring labels from foreign key names

Incident foreign keys that are frequent enough are then stored per target table and foreign key name. Foreign keys with the same names and the same target table but different origin tables are stored together and their statistics are summed up. An example is shown in table 4.3.

Algorithm 4.3 specifies the process in detail. The resulting statistics per CS contain a list of incident properties, from how many CS's this property is used to link to the current CS, and the sum of subjects linking to this CS using the property.

4.8.2. Discussion

Foreign key names describe the *role* of a table rather than the table itself. In some cases role and table content are described using similar words (e.g., role `team` for table `SoccerTeam`), but sometimes the words differ (e.g., roles `author`, `chairman`, and `goldMedalist` for table `Person`). In the latter case a table is referred to using many different foreign key names, and the summarizing term of these roles is not known.

Foreign key name	Number of CS's with this FK name	Total frequency
location	30	36502
country	18	135
birthPlace	17	40538
city	15	11146
owner	14	2741
deathPlace	10	8229
headquarter	9	1322
team	8	771
region	7	28204
district	7	14917

Table 4.3.: Example of incident foreign key names for the `Village` table

4.9. Assignment of Names

Table name candidates generated by looking at type values (section 4.6), foreign key names (section 4.8) and ontologies (section 4.7) are the input for the label assignment algorithm. Furthermore, frequencies and statistics collected while generating the candidates are handed over to the assignment procedure. The result of the assignment process is a *list* of candidates, sorted descending so that the first candidate in the list is chosen as CS label. The list contains sections for the different data sources the candidates come from: ontology, type value, foreign key name. A fourth section is introduced when merging takes place in chapter 5 to store the CS label of a merged CS.

4.9.1. Process

Name candidates discovered through analyzing *type values* are considered most important because type values usually point to the ontology, have a hierarchy, and were explicitly added to the RDF dataset upon creation. There is a *semantic reason* for choosing these names. For each type attribute (e.g., `dbpedia:type` and `rdf:type`), the best values is chosen according to the rules described in section 4.6. Out of these, the most frequent value is chosen.

Ontology-based candidates are good source for naming, too. Candidates discovered through matching CS's and ontology classes rely on comparison of multiple attributes, have a hierarchy, and the RDF data was created with the ontology in mind. However, in opposition to type values, they were not added explicitly to the dataset and their exploration requires fuzzy comparisons with an external data source. There is a again a *semantic reason* for choosing these names. Ontology candidates are already ordered by similarity score (cf. section 4.7), hence the ontology-based candidates can be added to the ordered list of all candidates without further sorting.

If neither ontology names nor type values are available, names of *incident foreign keys* are considered. Foreign key names describe the *role* of a table, not the table itself and are therefore less suitable for labeling. A foreign key name is considered a good choice for a table name if multiple incident foreign key links use this name (cf. section 4.8). Therefore the first criterion to choose a table name is the number of incident foreign keys that use the link name. If several foreign key names have the highest number of occurrences as incident foreign keys, the foreign key name with the highest total frequency is chosen as table name.

If none of the mentioned data sources is available, no useful table name can be assigned to the table. A dummy value is used instead. For ontology-based datasets like DBpedia, this is a very uncommon case, but

Algorithm 4.3: Collecting incident foreign key statistics

input : list of CS's, list of edges
output : incident foreign key statistics

```
forall the CS's do
    forall the attributes per CS do
        forall the edges do
            if edge frequency > threshold then
                if bucket for this incident foreign key name exists then
                    increase bucket counter by one
                    increase bucket frequency by the edge frequency
                else
                    create bucket
                    set bucket counter to one
                    set frequency to the edge frequency
                end
            end
        end
    end
end
forall the CS's do
    sort foreign key names descending by number of occurrences and frequency
end
```

it could happen in dirty datasets that are not based on a single, sophisticated ontology, e.g., Web-crawled data.

The CS labels will be used in merging (cf. section 5.1) to decide whether two CS's should be merged or not. Besides the chosen name, a list of all label candidates is maintained, that is also used for deciding about merging. Furthermore, the origin of the chosen label (type, ontology, or foreign key) is stored for evaluation purposes. If hierarchical information is available for the chosen name (i.e. the name is ontology-based and the ontology is loaded into *MonetDB/RDF*), the hierarchy is also stored. Hierarchical information will be needed when CS's are merged. After the merging phase, some labels need to be refined, this is done in the labeling step described in section 5.2. This step also includes labeling *properties*.

The rules for label assignment are described in detail in algorithm 4.4.

4.9.2. Storing CS Labels

The name assigned to each CS is stored among each instance in the CS. This is necessary as CS's will be merged in the next step (section 5.1). Merged CS's contain instances from different CS's and will therefore be described using a more generic name. Storing CS labels ensures that the detailed labeling information gathered in the last steps is not lost.

4.9.3. Discussion

Before we added semantic information to the type value extraction as described in section 4.6, ontologies were the first data source considered for assigning names. But with the ontological and hierarchical information used in type value extraction, it is now the most-valuable data source and therefore the first data source to look at during assignment of labels.

Algorithm 4.4: Assigning table names

input : ontology lookup candidates, type values, incident foreign key names
output : table name, list of sorted table name candidates

```
forall the CS's do
    if type values exist then
        add all type values to list of candidates
        /* take the best candidate per type attribute */
        forall the type attributes do
            take the candidate with the highest ontology level that exceeds the threshold
        end
        out of these, take the candidate with the highest frequency
    else if one or more ontology candidates exist then
        add all ontology-based candidates to list of candidates
        use first as table name
    else if incident foreign keys exist then
        add all foreign key names to list of candidates
        take the foreign key name that is used by the most incident foreign keys, if there are multiple choose
        the one with the highest frequency
    else /* no table name found */
        use dummy value
    end
    remove duplicate entries in candidate list
end
```

In general, the order of assigning uses *semantics* first: type values and ontology-based candidates. These semantic candidates are for example widely available in DBpedia and other carefully constructed datasets. As a fall-back for "dirtier" data like Web-crawled data, the non-semantic data source of foreign key names is available. Our experiments (cf. section 7.3) show that "dirty" We-crawled data in fact takes advantage of foreign-key-based labels, whereas semantic labels are sufficient for labeling the well-maintained dataset DBpedia.

Algorithm 4.4 exploits that the input candidate lists are ordered by suitability, hence keeping the assignment algorithm short and simple. We maintain the list of candidates because they are used for merging: The decision whether two CS's represent the same concept gets more evidence if not only the chosen names of the two CS's, but also the list of candidates, are compared.

5. Merging Labeled Structures in RDF Data

After the semantic information has been added to the structures, the CS's can be refined to increase the structuredness of the data. We therefore introduce a set of rules to decide whether two CS's should be merged (section 5.1). Merged CS's require an additional simple labeling step to adjust the CS labels (section 5.2). The result of these steps is the intermediate schema in a UML-like representation, as introduced in section 5.3. Afterwards, the schema has to be transformed into relational data. This is the first time we care about data types, relationship cardinalities, and foreign key constraints. The transformation process is described in section 5.4.

(Implementations described in sections 5.1 and 5.4 have partly been done by Pham Minh-Duc.)

5.1. Merging CS's

The goal of this step is to reduce the number of characteristic sets. As the resulting number of CS's determines the number of relational tables in the final schema, it is important to get a low number of CS's (e.g., below 1000) to reduce the overhead caused by creating and maintaining relational tables as well as to provide easier insights in the dataset. Important small CS's, as defined in section 3.8, are not merged but directly added to the final schema.

Similarity of two CS's is measured in two dimensions. CS's can be comparable because of their labels and ontology information, we call this type of similarity *semantic similarity*. Besides that, CS's can have a similar structure in their property sets or relationships, that is what we call *structural similarity*. The decision whether two CS's are merged is based on combinations of the following semantic and structural similarity rules.

Semantically similar CS's are identified based on their *labels*:

Rule 1: Same label Two CS's have a same label in their top k label candidate list (e.g., $k = 3$). This rule is executed per data source, meaning that the same labels have to come both from type values (cf. section 4.6), or both from ontology (cf. section 4.7), or both from foreign key relationships (cf. section 4.8).

Rule 2: Labels with a common ancestor For most ontologies we analyzed (cf. appendix A), hierarchy information for class names is given. If these class names are used as labels, hierarchy information can be exploited to find out whether two CS's should be merged. CS's will be merged if their labels belong to the same hierarchy. We call the point where the two hierarchies split, i.e., the lowest-level common hierarchy element, *common ancestor*.

For example, two CS's `Mayor` and `Senator` can be merged because of their common ancestor `Politician`. However, if the dataset is about politicians only, we would end up with one big CS `Politician`. To avoid this clustering, no merging is applied when the common ancestor of two CS's is too generic (i.e., covering a large fraction of the data). We therefore compute the fraction of CS's described by a specific label for each label in the hierarchy. For example, if the dataset is summarized by ten equally-sized CS's out of which one is labeled `Senator` and another one `Mayor`, these two CS's would not be merged because their least common ancestor `Politician` covers a rather big (20%) fraction of the CS's. This fact is displayed in figure 5.1.

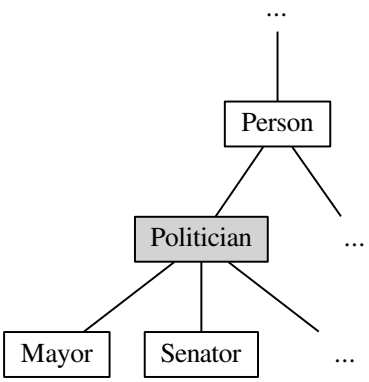


Figure 5.1.: `Politician` is the lowest-level common ancestor of `Mayor` and `Senator`

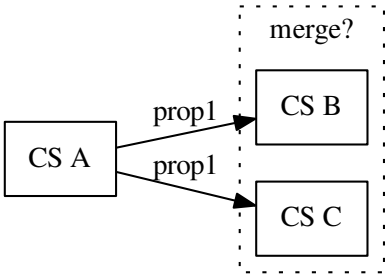


Figure 5.2.: Two CS's that are accessed from the same CS using the same property can be merged

Merging justifications that come from the ontology have a high value because a dataset is usually build according to its ontology. The ontology is therefore a good expression of the data classification the authors of ontology and dataset had in mind. However, these *semantic* rules is not enough to create the small, dense set of CS's we want, especially if a dataset uses multiple ontologies (e.g., Web-crawled data) or the ontologies do not provide a class hierarchy. We therefore also use merging rules based on *structural* similarity.

Structural similarity is used in addition to semantic similarity to decide on CS merging. Structurally similar CS's are identified using their *property sets* and *relationships*:

Rule 3: Superset If the property set of a CS is a subset of the property set of another CS, the smaller CS is merged into the bigger CS as shown in figure 5.3a. In relational terms, the value `D := NULL` is added to all subjects in the smaller CS. This is only done if the two CS's are not too different (i.e., the superset CS must not have more than a specified number of properties more than the subset CS).

Rule 4: Similar property sets The intersection of the property sets of two CS's is compared to the property sets of the two CS's. Identicalness in so-called *discriminating properties*, properties that do occur often within the two CS's but rarely in other CS's leads to a high *similarity score* of the two CS's. If the similarity score exceeds a certain threshold (e.g., 0.8 as for the tf-idf similarity between CS's and ontology classes described in section 4.7), the two CS's are merged as shown in figure 5.3b. In relational terms, the value `D := NULL` is added to all subjects in the left CS, and `C := NULL` is added to all subjects in the right CS.

Rule 5: Same incident references If two CS's is referred to from the same CS via the same property, the CS's are merged, as shown in figure 5.2.

In the first implementation of the merging step, we merged only two CS's at a time. Due to low performance we switched to *multiway merging*, i.e. merging n CS's at a time. Besides better performance, using multiway merging also avoids *chaining effects*, where a relatively big CS that is merged with multiple smaller CS's can pass its label through all merging steps, although its label does not represent the majority of data of the final merged CS. Only CS's that are matched because of the same rule(s) can be merged at a time, e.g., all CS's with the same label in top k (rule 1). Multiway merging is used for rules 1 and 5. For rule 1, multiway

(a) The smaller CS (top) is merged into the bigger CS (bottom)

(b) Two CS's with similar property sets are merged into a CS that contains the set union of the properties

Figure 5.3.: Merging CS's: Either into an existing CS or a newly created one

merging saves a lot of effort of there are many CS's with the same label. For rule 5, it is also natural to do multiway merging because all different referenced CS's have to be joined anyway and they are also detected at the same time. For rule 2, one cannot merge all CS's with the same common ancestor at a time because they are not detected at the same time and the common ancestor will get too generic if one tries to merge multiple ($n > 2$) CS's at a time. Rule 3 is not applicant for multiway merging because the superset-subset relation can be detected for only two CS's at a time. In addition, adding a subset CS to a superset CS does not require effort in merging properties, so the overhead introduced by sticking to merging only two CS's at a time is negligible. The same applies to rule 4: similarity is only defined between *two* CS's. Implementations of the five rules are shown in algorithms 5.1 to 5.5.

Algorithm 5.1: Merging Rule 1: Same label, multiway merging version

```
create a list of labels and the CS's they occur in (as top k candidate)
forall the labels do
    foreach data source do /* (ontology, type, fk) */
        collect all CS's that have this candidate from this data source
        merge them
    end
end
```

Algorithm 5.2: Merging Rule 2: Labels with a common ancestor

```
forall the pairs of CS's do
    if they have a least common ancestor then
        if least common ancestor is not too generic then /* percentage of data covered by
        this concept is below a threshold */
            merge them
        end
    end
end
```

Algorithm 5.3: Merging Rule 3: Subset-superset

forall the *pairs of CS's* **do**
  if *CS1 is a subset of CS2* **then**
    if *CS2 has at maximum 3 properties more than CS1* **then** `/* to avoid that small CS's are merged into very big CS's */`
      merge them
    end
  end
end

Algorithm 5.4: Merging Rule 4: Similar property sets

forall the *pairs of CS's* **do**
  compute tf-idf(CS1) = $\sqrt{\sum_{\text{properties } p \text{ of } CS1} \text{tf-idf}(p)^2}$
  compute tf-idf(CS2) = $\sqrt{\sum_{\text{properties } p \text{ of } CS2} \text{tf-idf}(p)^2}$
  compute similarityScore(CS1, CS2) = $\frac{\sum_{\text{properties } p \text{ that occur both in } CS1 \text{ and } CS2} \text{tf-idf}(p)^2}{\text{tf-idf}(CS1) * \text{tf-idf}(CS2)}$
  if *similarityScore(CS1,CS2) is above a threshold* **then**
    merge them
  end
end

Algorithm 5.5: Merging Rule 5: Same incident references, multiway merging version

forall the *CS's* **do**
  forall the *properties of that CS* **do**
    collect all CS's that are frequently referenced from this property
    merge them
  end
end

The merging step balances two objectives for the final relational schema: On the one hand, the schema should contain as few tables as possible, achievable by lax merging conditions. On the other hand, the tables should not contain too many NULL values which demands strict merging conditions. Setting the thresholds to reasonable values is essential for the quality of the resulting schema.

5.2. Labeling Final CS's

As shown in figures 5.3a and 5.3b, merging can either result in a newly created CS, or an update to an existing CS by adding new data to it. Newly created CS's do not have a name assigned yet, and names of updated CS's have to be checked and updated if necessary. This final labeling step assigns names to the final set of CS's, leveraging the labels of the CS's the merged CS consists of. Section 5.1 explains that we switched from merging two CS's at a time to multiway merging to increase performance. The following explanations on labeling merged CS's therefore assume an unknown number (≥ 2) of CS's that are merged together.

The labeling techniques that are applied in this step vary depending on the rules that were used to merge the CS's. If the CS's are merged according to *semantic* rules, the label for the merged CS is already known:

Either, the CS's are merged because of a common name (rule 1), in this case the common name is used as label, or they have a common ancestor in the hierarchy of their labels (rule 2), in that case the least common ancestor is used as label. If *structural* similarity rules decided about merging the CS's, the name of the merged CS cannot be semantically inferred. Instead, the *size* of the CS's decide about the final name: If the CS's to be merged form a subset-superset hierarchy (rule 3), the name of the superset CS is chosen as the name of the merged structure. If the structurally similar CS's are merged into a newly created CS because there is no subset-superset hierarchy (rules 4, 5), the name of the biggest CS (i.e., having most subjects) that is part of the merged CS is chosen as the name of the merged structure.

As described in sections 4.6 to 4.8, three data sources are used to compute label candidates, hence three lists of candidates are produced. Section 4.9 describes that these lists are concatenated, starting with ontology-based candidates, followed by type-value-based candidates, ending with relationship-based candidates. When CS's are merged, and CS names are updated, the candidate lists also have to be merged. All CS's that participate in a merged CS contribute to the merged candidate list. The candidate lists are merged *per group*, hence within ontology-based, type-value-based, and relationship-based candidate lists. Within these groups, the candidates of all participating CS's are concatenated, starting with the most important CS (i.e., the superset CS or the biggest CS). Duplicated entries are removed. As this labeling step is intersected with the merging step described in section 5.1, the merged candidate list is instantly used for further merging decisions.

Furthermore, names for each property are defined. Property naming does not need a complex labeling algorithm, but is done using URI shortening described in section 4.4. Property names are the column names in the final relational schema, will be presented to the user and should hence be human readable, no cryptic OIDs.

At this point in time, all labels are transformed from OIDs to strings. Throughout our structuring, labeling, and merging algorithms, OIDs were used to simplify label comparison and storage, but from now on, for the physical transformation into the relational data model, string names are needed. Hence, URI shortening as described in section 4.4 is applied.

5.3. UML-like Intermediate Result

As mentioned in section 3.3, structuring RDF data is a two-step process. The first step, transforming RDF triple data into an intermediate, UML-like model consisting of CS's and their relationships has been explained in chapters 3 to 5. This model does still contain elements that cannot be represented in a relational model, such as multivalued properties and undetermined foreign keys. Hence, a transformation is necessary that purges these elements. It is described in section 5.4.

5.4. Transformation to Relational Structures

To create the final relational model, the relational type system has to be taken into account. Casting between *data types* has to be avoided to speed up the processing in the targeted relational schema. Therefore properties which values belong to different data types (e.g., `length_in_meters` can be either of type integer or decimal) will be represented by multiple table columns, one for each data type. These additional columns are stored in an extra table to keep the schema of the main table short and correct. If only a small percentage belong to a different data type, these values are removed from the main table and stored in the triple store that is maintained for outlying data. NULL values are used if values are missing, e.g., because of CS merging.

In addition, properties might be split up over multiple columns if they contain *foreign keys*. Because the relational integrity requires foreign key columns to point to one other table only, splitting is necessary if a property points to multiple other tables or some values do not point to anything but contain plain values. Because merging rule 5 merges referenced CS's, the number of foreign keys pointing to multiple CS's is already low. To further reduce the number of split columns, outlying values are removed. For example, if only 1% of the values in a foreign key column point to a specific other table, these pointers are removed from the dataset. By removing these outliers, the regularity of the final schema is increased and only few data is dropped.

To represent *multivalued properties*, an additional table per multivalued property and table is added. These additional tables have a many-to-one relationship with the main table and contain the mapping between subjects and multivalued values.

All data that has been dropped during structuring and merging is not stored in the relational schema, but in a *PSO table*. A PSO table is a triple store that is ordered by predicate, then subject, then object.

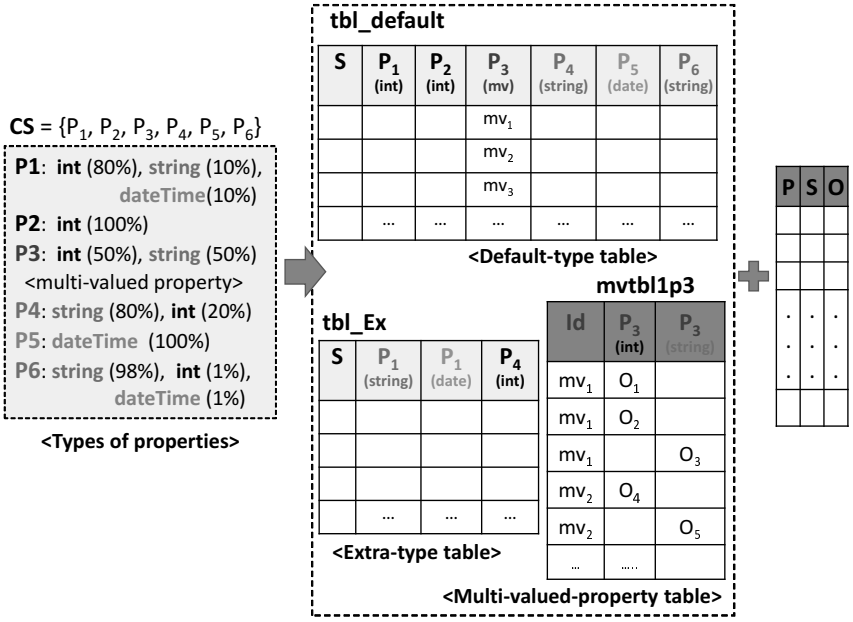


Figure 5.4.: Transformation of a CS to relational tables
(Figure by Pham Minh-Duc [24])

Figure 5.4 *(by Pham Minh-Duc [24])* shows this transformation in detail. A CS with six properties is transformed into three relational tables plus the PSO table. The first step is the analysis of the CS: the data types of its six properties are counted and the multivaluedness of the third property is recognized. The default table has exactly the schema of the CS. It contains the most data, hence the most common data type per property is chosen to be in the default table. For the properties that have multiple common data types, an extra table is created. In this extra table, values of the other data type are stored. The subject column that is available in both the default and the extra table ensures that the contents can be merged. As shown in the figure, infrequent data types such as `int` for the sixth property are not transferred to the relational model. Instead, these outliers are moved to the PSO table. For each multivalued property, another relational table is created. In the example, only one table for the third property has to be created. An artificial Id that is used in that table as well as in the default table ensures that all contents can be merged. For each data type that occurs frequently in the multivalued property, a column in the new table is created.

The schema created during transformation is used as physical data layout, not only as an abstraction layer. The layout ensures high data locality and reduces the number of joins that is necessary for typical queries against RDF data.

The union of relational schema and PSO table contains all data, ensuring correct results when querying it. If only the relational schema is queried or explored, the majority of data is shown and can easily be understood by humans. The result hence covers both needs: fast and correct querying, and a human readable, relational presentation of the contents of the data.

6. Implementation

The algorithms to structure and label RDF data described in chapters 3 to 5 are implemented as part of the database module *MonetDB/RDF*. The following sections describe the architecture of *MonetDB/RDF* and the design decisions that led to it. In addition, libraries and tools used in the module are introduced.

6.1. Requirements and Constraints

As *MonetDB/RDF* is designed to be integrated and shipped with *MonetDB*, several additional criteria arise.

Robustness Against Input Data As RDF data is often generated by Web crawls, malformed input data has to be considered. For example, wrong usage of string escape characters by the data providers can lead to corrupted object strings. The same applies to spelling errors in predicate URIs and the like. In addition, RDF data (e.g., titles and description texts) is often available in multiple languages, hence correct character encoding for many writing systems is required. Finally, duplicated RDF triples must not be added to the triple store when parsing the RDF data as they do not provide additional information. *MonetDB/RDF* therefore has to be built to cope with all mentioned malformations.

Loading RDF Metadata The structuring/merging and labeling algorithms rely on ontology class names and hierarchy information. It is therefore essential to have the ontologies that are used in the dataset available in *MonetDB/RDF*. The "moving target" RDF metadata cannot be mastered by adding ontologies before shipping *MonetDB* because of the large and quickly changing set of ontologies (cf. section 4.3.2 for the distribution of different ontologies in a Web-crawled dataset). Instead, users must be able to load new ontologies with little effort. This is not trivial as ontologies can be written in different file formats, entailing different metadata extraction algorithms for different ontologies. Leaving ontology loading to end-users also ensures compliance with ontology licenses from a *MonetDB* point of view.

Performance During loading of RDF data into *MonetDB*, several resource-expensive data transformation steps take place: First, RDF data needs to be parsed and stored into an initial triple store. Except for the afore-mentioned error cases this means that one line in the input file results in one row in the triple store. While storing, duplicates are eliminated and the data is ordered by subject, then predicate, then object. Afterwards, the structuring and labeling algorithms described in chapters 3 to 5 are called, also resulting in heavy transformations and aggregations on the whole dataset. As we do not support adding data or updating the dataset by now, these expensive transformations happen only once per dataset.

MonetDB/RDF loading performance does not affect other parts of the database system and is therefore not critical when it comes to integration with *MonetDB*. However, RDF datasets can be very large and even if users do expect waiting time when loading big datasets, loading should not take longer than necessary. Users should be kept informed about the progress and maybe estimated duration.

***MonetDB* Integration** The work in this thesis is part of *MonetDB/RDF* which is a module of *MonetDB*. Hence, integration with both the module and the whole RDBMS is necessary. To be usable as an encapsulated module, side effects have to be avoided and a call interface has to be provided. Our algorithms have to be called using the *MonetDB Assembly Language* (*MAL*).

MonetDB is written in *C*. As parts of *MonetDB/RDF*, namely the operators *RDFscan* and *RDFjoin*, should be highly integrated with the database kernel [23], using C is advantageous. Persistent storing of data (i.e., data being available after a restart of the database) has to make use of *MonetDB* data types and data structures, e.g., numerical types with certain value ranges. Integration with SQL requires sticking to the SQL table naming rules, e.g., the first character of a table name has to be alphabetic.

6.2. Tools and Libraries

As *MonetDB/RDF* introduces a *new approach* on how to analyze, structure, label, and store RDF data in an RDBMS, the main functionalities are implemented as part of the project and are not based on or relying on external libraries. However, libraries come into application for parsing RDF data and string tokenization, as these are well-researched topics where libraries – external or within *MonetDB* – do exist.

RDF Parser: *Raptor* Parsing RDF data, as described in section 3.5, is done using the RDF library *Redland librdf*. For parsing and serializing, librdf provides *Raptor*[18]. Raptor can handle many RDF data formats including N-triples and can easily be integrated with C programs. While parsing, Raptor extracts various type information:

- For subjects, Raptor states whether they are represented by a blank node or an URI,
- for predicates, Raptor determines whether they have a proper format (URI) or not, and
- for objects, Raptor distinguishes between nodes (either blank nodes or URIs) and literals.

We use this information to correctly classify strings and to remove blank nodes. Raptor hands over one triple at a time, enriched with type information, to be appended to the triple store in our module.

String Tokenizer To efficiently store URI strings in *MonetDB* data structures, the *MonetDB* string tokenizer is used. The tokenizer splits strings at a given separator character. The tokens are then stored in a *Binary Association Table* (*BAT*), the *MonetDB* data structure for storing values and their corresponding *object-identifiers* (*OIDs*) [13]. A BAT represents a single column in the column-store *MonetDB*, with the OID identifying the row and the corresponding values being the content.

The *MonetDB* string tokenizer creates an OID for every string and reconstructs strings if an OID is given. Using *MonetDBs* dictionary encoding (cf. section 2.1), the tokenizer ensures that each token is stored only once in the string storage data structure. All instances of this string get the same OID assigned. Duplicate elimination is essential for efficient storage as URIs in RDF data often have common tokens, e.g., ontology prefixes. Representing strings as OIDs also simplifies string comparison, which is heavily used in creating characteristic sets and during the labeling phase.

[18]Raptor `http://librdf.org/raptor/`

6.3. Software Architecture and Integration with *MonetDB*

The *MonetDB/RDF* module uses a three-tier architecture. The architecture is shown in figure 6.1 and explained in the following paragraphs.

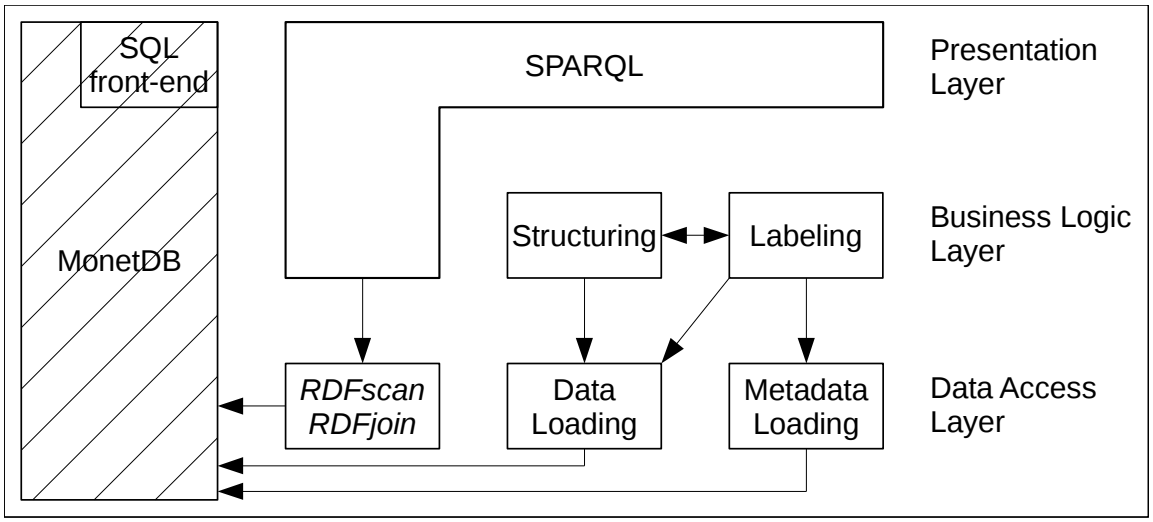


Figure 6.1.: *MonetDB/RDF* architecture
SQL front-end used to access *MonetDB/RDF* data is taken from *MonetDB* core

Data Access Layer On the data access layer, loading and parsing of both RDF data and metadata takes place as well as conversion to SPO triples and relational structures, respectively. Data and metadata (i.e., ontologies) are loaded from plain text files, parsed and sorted. Loaded metadata is used in the labeling algorithms to create better names for the structures. The structuring and merging algorithms use metadata, too, to decide which CS's are merged. The parsed results are stored in a defined database schema. By doing so, all auxiliary data is stored in one place and can be accessed efficiently by the business logic layer.

The operators *RDFscan* and *RDFjoin*, as introduced by Pham [23] and briefly described in section 2.2, access *MonetDB* internal data structures directly to provide CPU efficient data retrieval.

Business Logic Layer The labeling and structuring parts in the business logic layer of *MonetDB/RDF* are described in detail in chapters 3 to 5. As explained there, the amount of communication between these two parts grew over the time. Extensive sharing of intermediate results is used to avoid duplicated computations and therefore keep loading times short.

Both parts need access to the dataset: Structuring requires all subject and predicate information to find characteristic sets (CS's) in the triples. Labeling has to scan all triples including object values to collect information on type values as described in section 4.6. In addition, both parts also need access to all ontology metadata to compare CS's with ontology classes and to exploit ontology class hierarchies (cf. section 5.1).

As explained in section 5.3, the labeling and structuring/merging parts transform RDF data into an UML-like presentation. Afterwards, this intermediate schema is transformed into a pure relational schema, e.g., by adding and ensuring foreign key constraints. This is explained in section 5.4. Finally, the resulting relational schema is stored persistently into the database.

Presentation Layer Two front-ends provide access to the RDF data. The standard SQL front-end from *MonetDB* can be used because the final schema uses features of standard SQL only. When using the SQL interface, users can benefit from the variety of available SQL tools, e.g., for graphical schema representation (cf. section 1.1). The SPARQL front-end allows querying the data the RDF standard way. In contrast to the SQL front-end, the SPARQL front-end also incorporates the irregular triples that were put into the *irregular* part of the triple store during the structuring and merging phase.

Interface The *MonetDB/RDF* interface consists of four commands: First, using the command `rdf.shred`, RDF triple data in N-triples format is loaded into a SPO store within *MonetDB*. As part of this step, URIs and strings are stored separately in the *MonetDB* dictionary as introduced in section 2.1. After the ontologies have been loaded into the database via the `rdf.copyOntologyToDatabase` command, they have to be loaded into the *MonetDB/RDF* module using the `rdf.loadOntologies` command. This function transfers all ontologies within the database into the C code of *MonetDB/RDF*, and detects the ontology hierarchy used for labeling and merging. To start the actual transformation of RDF data into relational data, the command `rdf.reorganize` needs to be called, and the frequency threshold has to be set. Within this function, all parts of the transformation process, from basic CS detection to the final psychical reorganization, are executed. Of course, all commands keep the user updated about their current status.

7. Evaluation

Our evaluation should cover many aspects of the work presented in this thesis. We therefore decided to not only do *experiments* to evaluate the effects of e.g., different parameter choices, but also create a *survey* to ask *humans* about the structures and labels we created.

First in this chapter, in section 7.1, we introduce some related work on evaluating labels and define the metrics used throughout our evaluation. In section 7.2, we introduce the datasets we use for the experiments. The experiments are discussed in section 7.3. Section 7.4 shows the survey and its results. Finally, in section 7.5, we discuss the outcomes of the evaluations and conclude.

7.1. Related Work and Metrics

Venetis et al. [31] present some table labels to humans and ask for classification on the scale *vital*, *okay*, *incorrect*. In addition, they also ask the participants to provide additional labels. However, they use this manually created *gold standard* to rate the quality of their labels in the overall dataset, whereas we present *all* tables and *all* labels to users. The authors use their gold standard to automatically classify labels created by other algorithms, whereas we let humans rank our whole data basis and are especially interested in the *order* of labels, their *level of detail*, and the *overall quality*.

For evaluating the understandability of labels, we design a survey, where humans rank the labels suggested by our algorithms. In contrast to the afore-mentioned 3-point scale by Venetis et al. [31], we decided to use a classical 5-point Likert scale. For the experiments, we measure the number of tables in our schema, and the percentage of triples covered by our schema. These measures give a first overview how good the algorithms variants are at *compressing* the dataset into a relational schema. In addition, label sources and possible duplicated labels are shown to inform about label quality. All these measures can be compared over the different experiment we carry out. Evaluation of the consumption of time and resources is not part of this examination.

7.2. Datasets

We use two datasets for our experiments. These two different datasets represent the possible spectrum of RDF datasets. DBpedia is a carefully administered dataset that is designed according to its ontology. This dataset contains clearly distinguishable concepts (e.g., `City`, `Person`). The other dataset is a dirty, Web-crawled dataset. It contains references to many different ontologies, spelling mistakes, outdated data, and other flaws. However, we expect our algorithms to work with this kind of data, too.

7.2.1. DBpedia

The DBpedia[19] dataset consists of 99 separate data files. For the tests in this thesis, we removed several datasets: links to other datasets such as DrugBank, non-English text data, long text data such as abstracts,

[19] DBpedia `http://dbpedia.org/`

links to external Web pages, links to images, Wikipedia internal data (e.g., page redirects, article categories). We end up with nine datasets that are loaded into *MonetDB/RDF* for testing purposes: Geo Coordinates, Instance Types, Instance Types Heuristic, Labels, Mapping-based Properties Cleaned, Persondata, PND, SKOS Categories, and Specific Mapping-based Properties. These nine datasets contain mostly the information one can find in the infoboxes in Wikipedia articles. These datasets have 8.8GB of CSV data, containing 68,711,383 triples.

7.2.2. Common Web Crawl

Only a small slice of the Common Crawl[20] data is loaded for our tests. We chose a 700MB file of the August 2012 version of Common Crawl that contains 4,594,728 triples. The Common Crawl data is crawled from small snippets of RDF data on Web pages (RDFa), such as `date` information attached to online newspaper articles. In opposition to DBpedia, it contains properties from many different ontologies and usually only few facts (triples) per subject.

7.3. Experiments

The experiments carried out on the two described datasets will be used to discuss the outcomes of our algorithms, on different datasets and with different parameters. We start with a *base experiment*. This experiment shows the final result, with all features developed as part of this thesis included and active. Afterwards, we modify the experiment to show variances of the algorithms:

Data Sources for Labeling We remove and add data sources that are used for labeling. We start without using any data source, then add the internal data sources *foreign keys* and *type properties*. The latter one already uses ontologies and is therefore not a pure internal data source. Finally, we also add the external data source *ontologies* and end up with the base experiment setup again.

Semantic Information for Merging Phase As discussed in section 3.3, the first version of the whole structuring and labeling process merges CS's that are in a subset-superset relationship without taking semantic information into account. This experiments therefore disables the semantic checks in the subset-superset merging phase.

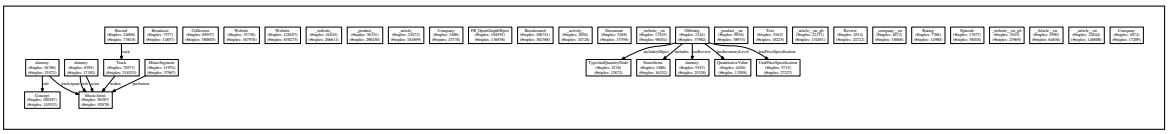
Per experiment, we note the dataset and frequency threshold that are used. We present (parts of) the final schema in tabular and graphical representation, and provide statistics about the schema and its labels as well as the irregular data in the PSO triple store.

7.3.1. Base Experiment

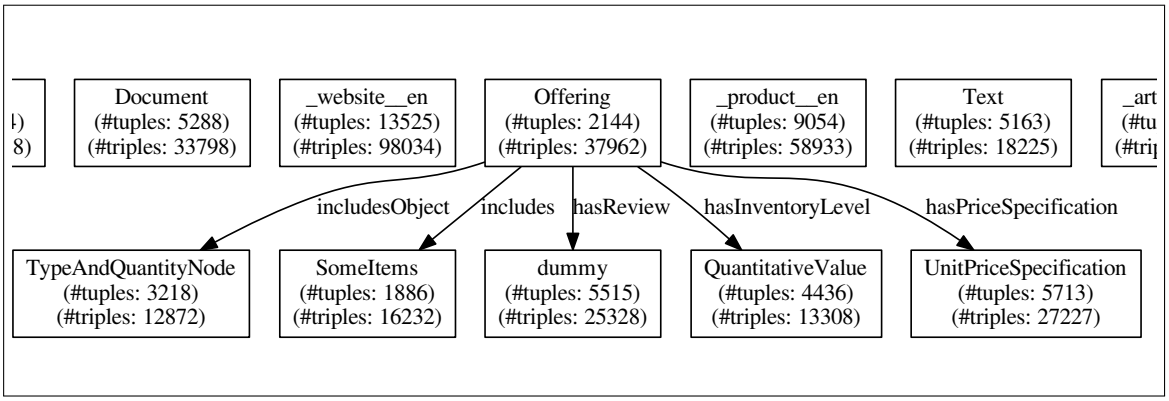
The frequency threshold is the only parameter that is set by the user. It defines the number of subjects each CS has to have at the end of the basic CS discovery (cf. section 3.8). We assume that each subject if formed of 20 triples, this includes multivalued properties. For this experiment, we want to have less than 100 CS's, this adds the factor 100. Furthermore, we assume that each final merged CS consists of 10 basic CS's. If we multiply these factors, we get 20000, hence $\frac{\#\text{ triples}}{2000}$ is a reasonable value for the frequency threshold. For the Common Crawl dataset, we therefore use 250, and 3500 is used for the DBpedia dataset.

[20] Common Web Crawl `http://webdatacommons.org/`

Schema Overview Figures 7.1a and 7.2a show the UML-like schema generated by our algorithms. As one can see, the Web-crawled data is kind of clustered in groups that have many links between each other but nearly no links to other groups.

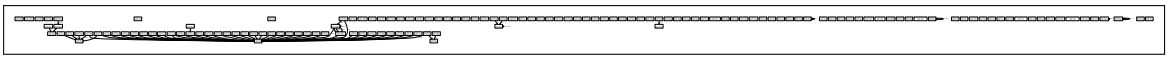


(a) Full UML-like schema

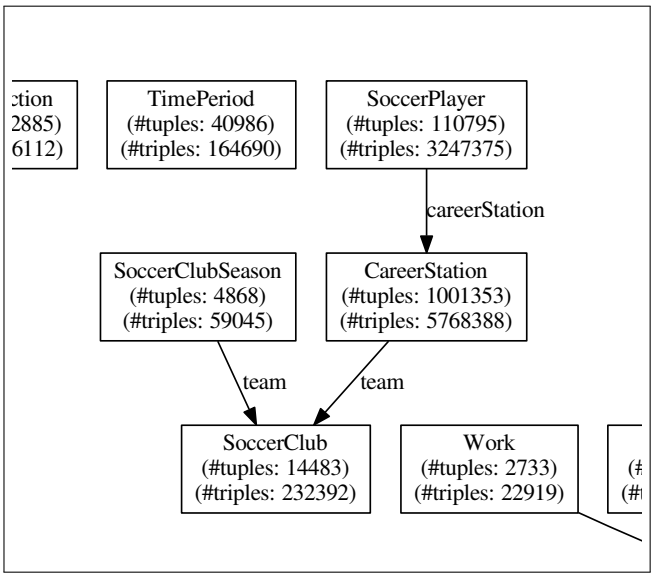


(b) Cluster on e-commerce

Figure 7.1.: Base experiment: CS's and their relationships in the Web-crawled dataset



(a) Full UML-like schema



(b) Cluster on soccer players and their clubs

Figure 7.2.: Base experiment: CS's and their relationships in the DBpedia dataset

The clusters loosely relate to ontologies. The DBpedia dataset contains more foreign keys, but still many CS's without links to other tables. There seem to be *central* tables with many foreign key links. Figure 7.1b shows one cluster within the Web-crawled dataset: The classes and the data within them mainly relates to the GoodRelations ontology that describes e-commerce. However, one CS has the name `dummy` which means that no name could be assigned to it. Figure 7.2b shows a similar cluster in the DBpedia dataset. As this data uses one ontology only, the clusters relate to *topics* rather than ontologies. This cluster shows `SoccerPlayer` and related information, such as their clubs and different career stations (at different clubs). The class names in this magnified part of the DBpedia dataset are taken from the DBpedia ontology.

On our evaluation machine (two 2GHz 8-core processors, 256GB main memory), the whole transformation process takes about 3 minutes for the Web-crawled dataset, and 35 minutes for the bigger DBpedia dataset. As table 7.1 shows, most time is spend on the initial loading of triples into the SPO table. Only 12 seconds (Web-crawled dataset) and 4 minutes (DBpedia) are spend on structuring, loading, merging and physically reorganizing the RDF dataset. Hence, transforming RDF data into relational data causes no significant overhead compared to loading RDF data into a triple store. Within our transformation, the phases structuring, labeling and reorganizing have to loop over each triple, whereas the merging phase only loops over CS's. It is therefore the fastest phase.

	Web-crawled dataset	**DBpedia dataset**
Loading	157	1878
Structuring	0.5	16.5
Labeling	0.6	31
Merging	0.1	1.4
Transforming	11	188
Total	169	2115

Table 7.1.: Base experiment: Runtimes in seconds (machine: two 2GHz 8-core processors, 256GB main memory)

The statistics gathered for the base experiments is shown in tables 7.2 and 7.3. The following paragraphs explain and interpret these numbers.

	base experiment
coverage	89%
# tables	37
# labeled	35
type	78%
ontology	16%
foreign key	0%
none	5%
duplicates (# tables)	Article (4), Company (3), Product (2), Website (5)

Table 7.2.: Base experiment: Statistics for the Web-crawled dataset

	base experiment
coverage	79%
# tables	140
# labeled	140
type	100%
ontology	0%
foreign key	0%
none	0%
duplicates (# tables)	Person (2)

Table 7.3.: Base experiment: Statistics for the DBpedia dataset

Structures and Schema The Web-crawled dataset is represented by 37 CS's. The dimension criterion applied to none of them. Out of the 344 properties, 24 are multivalued, meaning that they have multiple values. An example for a multivalued property within the Web-crawled dataset is `acceptedPaymentMethods` of the CS `Offering` that is also shown in figure 7.1b. Within the schema, 12 foreign key references exist. As shown before, this is a relatively low number of references between CS's, and results in some clusters and many CS's without any connection to other CS's. We presume that the wide range of different ontologies and data sources leads to this low number of foreign keys. Out of the 4,594,728 triples that the Web-crawled dataset consists of, 4,091352 made it into the CS'S, and 503,376 are stored in the PSO triple store. This means that 11% of the triples are considered *irregular*. Within these irregular triples, 77% were not assigned to a CS in the first place. The other 23% were removed from a CS later, for example because these triples' object types did not fit the majority of object types within a property of a CS (cf. section 5.4). By changing the frequency threshold to a lower value, one can increase the percentage of regular triples, but will also end up with a higher number of tables.

When run with frequency threshold 3500, as suggested by our calculations, the DBpedia dataset is represented by 35 CS's that do cover 55% of the triples. The reason for the bad coverage is the existence of a very big CS covering more than 10 million triples. Our previous calculation therefore have to be adjusted. We experimentally found out that a frequency threshold of 50 results in a good trade-off between number of tables and coverage. We conclude that finding a good frequency threshold is not a trivial operation and discuss other possibilities in section 8.2. With the reduced threshold of 50, the DBpedia dataset is represented by 140 CS's, out of which zero where kept because of the dimension criterion. Within the 1759 properties, the dataset has 690 multivalued properties. For example, type properties are usually multivalued because they contain a whole hierarchy of types for each subject. Another example is the property `starring` of the CS `Film`. 62 foreign key relationships exist within the dataset, hence supporting the first impression that arose from the overview figure 7.2a. The whole dataset consists of 68,711,383 triples, out of which 79% (54,222,943 triples) are represented by the CS's. The remaining 14,488,440 triples are stored in the PSO triple store. 97% of the irregular triples were never assigned to a CS, this means that the DBpedia dataset has many property sets that occur less than 50 times in the whole dataset. When looking at the large number of properties per CS, this sounds reasonable, and is another reason why we had to set the frequency threshold that low.

Labeling 35 of the 37 tables in the Web-crawled dataset are labeled, two remain unlabeled. Within the 35 labels, 29 are created from the first data source, type properties. The remaining six tables get ontology-based names. The third data-source, foreign key names, is never used to generate a label. From looking at the schema we see that the unlabeled tables do not contain type properties, are difficult to match with an ontology because the both contain two properties only, and have no incoming references. This is the reason they do not get labeled. Within the 35 labeled tables, several labels occur multiple times, for example `Article` and `Website`. The reason behind the duplicates is that the concepts of an article and a Website are defined in multiple ontologies, hence they can be represented using different sets of properties, and are therefore not chosen for merging.

The DBpedia dataset of 140 tables contains no unlabeled tables. All labels are generated using type properties, the first data source. Only one table name occurs twice: `Person`. One can imagine that the rather generic concept of a person can contain very diverse subjects, hence diverse property sets. This leads to not merging the two *Person* tables.

The use of only *one* ontology leads to very good labeling results in the DBpedia dataset. We list several possible extensions to our algorithms in section 8.2 that would help to increase the label quality in the Web-crawled dataset. For example, by taking connections between ontologies into account (e.g., exploiting `owl:sameAs` property), one could overcome the disadvantages of having multiple ontologies in one dataset.

Figures 7.3 and 7.4 add all table names to the hierarchy based on the ontologies. These figures show how much data is covered by each table. For example, the `SoccerClubSeason` table contains only 0.3% of the DBpedia dataset, but as the dataset is very large this is still a sufficient amount of data for the table to be added to the final schema.

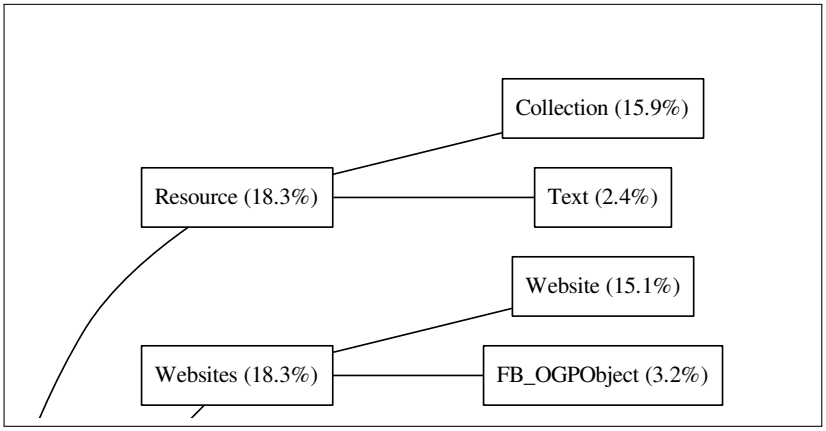


Figure 7.3.: Base experiment: Ontology classes found in the Web-crawled dataset, zoomed in

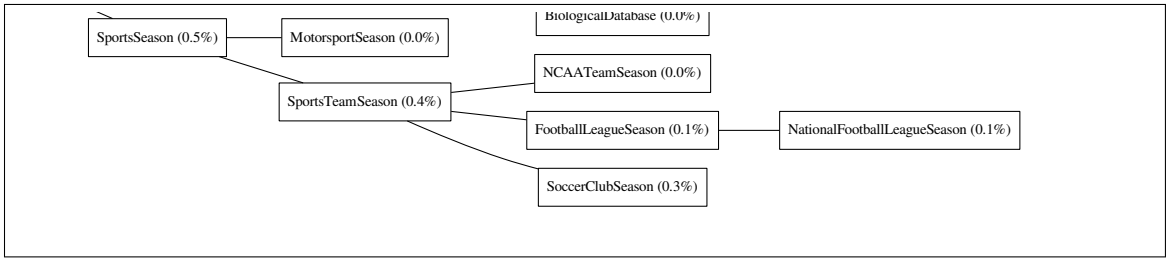


Figure 7.4.: Base experiment: Ontology classes found in the DBpedia dataset, zoomed in

7.3.2. Data Sources Experiment

Within these experiment series, we disable and enable data sources (type properties, ontologies, foreign key references). As tables 7.4 and 7.5 show, the characteristics of the resulting schemas change when we disable data sources.

If only CS relationships (FKs) are used for labeling, only 11 out of 40 tables in the Web-crawled dataset are labeled. If we add more data sources, more tables are labeled, and in the end, no tables has a relationship-based label anymore. We conclude that tables in a cluster, that have incoming relationships which names can be used as labels, usually also have label candidates from other data sources. For example, a table with the relationship-based label `page` also has a type-based label `Website`. The latter label describes the table's contents better. Our survey (cf. section 7.4) supports the fact that type- and ontology-based labels are in general better understandable than FK-based labels. Our chosen order of data sources, using FKs only if no other data sources lead to label candidates, seems therefore reasonable. Another observation extracted from the table is the changing number of tables. Although the percentage of triples that is covered by our schema slightly increases when additional data sources are added, the number of tables varies. This happens because of the label-based merging. Labels are taken into account for the semantic merging rule 1, where CS's with the same labels are merged. In addition, two CS's are merged if they both have a label from the

	none	**foreign keys**	**FKs and type**	**all**
coverage	88%	88%	89%	89%
# tables	41	40	44	37
# labeled	0	11	31	35
type	0%	0%	64%	78%
ontology	0%	0%	0%	16%
foreign key	0%	28%	7%	0%
none	100%	73%	30%	5%
duplicates (# tables)		URL (2)	Article (4), Product (2), Website (3)	Article (4), Company (3), Product (2), Website (5)

Table 7.4.: Data sources experiment: Statistics for the Web-crawled dataset

same hierarchy and their common ancestor name is not too general (rule 2). This rule has most effect when type- or ontology-based labels are available, as these do often have hierarchy information attached. These two semantic merging rules explain why the number of tables in our schema *shrinks*. But there is also a conserve effect, the number of tables can also *grow*, as shown in table 7.4 when we add the property type values as data source. Rule 3 of the merging rules (cf. section 5.1) merges two CS's in a subset-superset relationship (if their property sets are not too different), only when there is no *semantic evidence* against merging them. As explained in section 3.3, this check is necessary to avoid having concepts merged into its sub-concepts. Both effects, merging based on labels and not-merging because of labels, account for the oscillating number of tables in our schema throughout this series of experiments.

	none	**foreign keys**	**FKs and type**	**all**
coverage	78%	78%	79%	79%
# tables	187	197	136	140
# labeled	0	39	136	140
type	0%	0%	100%	100%
ontology	0%	0%	0%	0%
foreign key	0%	20%	0%	0%
none	100%	80%	0%	0%
duplicates (# tables)			Person (2)	Person (2)

Table 7.5.: Data sources experiment: Statistics for the DBpedia dataset

When looking at the DBpedia dataset, we observe similar effects. The percentage of labeled tables increases if additional data sources are enabled. As this dataset is well-equipped with property types and ontology information, adding one of these sources suffices to have all tables labeled based on these data sources. Here, the number of tables in our schema drops drastically when type-based labels are added to the algorithms, from 197 to 136 tables. Again, merging based on labels is responsible. Using the ontology-backed property type values, label- and hierarchy-based merging (cf. rules 1 and 2 from section 5.1) takes place and reduces the number of tables significantly. Only very few duplicates remain because the label-based merging merges most duplicates. In fact, only tables with not exactly the same names remain. The two `Person` tables in the DBpedia dataset come from different ontologies (DBpedia and FOAF) and are therefore not merged using rule 1. The `Person` class from both ontologies are marked as being equivalent (`owl:equivalentClass`) in the DBpedia ontology definition. If one had exploited this equivalence, it would have been possible to merge this two

tables, too. This idea is further elaborated in the future work section 8.2.

7.3.3. Semantic Merging Experiment

For this experiments, we disable the use of semantics in the merging phase: No merging is done based on labels or label hierarchies, and semantic checks in the subset-superset merging are also disabled. This configuration emulates the initial process described in section 3.3, where merging was done before labeling so that no semantic information was available at that point in time. As stated there, this version of the process led so severe quality problems with the distribution of subjects to structures, as instances of a base concept were added to structures about its sub-concepts.

	without semantics	**with semantics**
coverage	88%	89%
# tables	37	37
# labeled	35	35
type	84%	78%
ontology	11%	16%
foreign key	0	0%
none	5%	5%
duplicates (# tables)	Article (10), Collection (2), Episode (2), Produce (3), Website (3)	Article (4), Company (3), Product (2), Website (5)

Table 7.6.: Semantic merging experiment: Statistics for the Web-crawled dataset

For the Web-crawled dataset, shown in table 7.6, this has only marginal effects on schema size and label sources. The distribution of label sources changes slightly towards type-based values if no semantics are used during merging. More duplicate labels exist, these would have been merged if semantics had been enabled. As the label `Article` appears even 10 times within the 37 tables, the goal of *discriminating* table names is clearly not met by this algorithm variant. The equal number of tables is coincidence.

	without semantics	**with semantics**
coverage	79%	79%
# tables	161	140
# labeled	161	140
type	100%	100%
ontology	0%	0%
foreign key	0%	0%
none	0%	0%
duplicates (# tables)	AdministrativeRegion (2), Aircraft (2), AmericanFootballPlayer (2), Athlete (3), BasketballPlayer (3), Building (2), City (7), Company (4), IceHockeyPlayer (2), Lake (3), OfficeHolder (4), Person (3), RadioStation (2), River (3), School (3), Settlement (13), Stadium (2), Station (4), TennisPlayer (2), TennisTournament (2), Weapon (2)	Person (2)

Table 7.7.: Semantic merging experiment: Statistics for the DBpedia dataset

When looking at the DBpedia dataset in table 7.7, we see dramatic effects. Without using semantics, the schema has about 20 tables more, among which many duplicates are found. One table name (`Settlement`), is used 13 times among the 161 tables of the schema. We obtained these results in an early stage of algorithm development, when we did not use any semantics for merging. In fact, at that time, we merged before labeling, hence zero semantic information was available for merging. Then, merging was only based on subset-superset relations and incident relationships. As elaborated in section 3.3, this leads to general concepts being merged into sub-concepts. As almost all labels in this dataset are based on the DBpedia ontology, we can compute the average hierarchy level of all labels, to flesh out that the labels are *too specific* when semantic merging is disabled. The DBpedia ontology forms a hierarchical tree, with `Thing` being the root, level 0. The average hierarchy level with enabled semantic merging is 2.96, without semantics it is 3.23. For example, the level 3 label `SportsLeague` that is present in the semantic version, is replaced by the level 4 labels `BasketballLeague` and `SoccerLeague` in the non-semantic version. Other examples for too specific labels are `SkiArea` (instead of `Place`) and `Plant` (instead of `Eukaryote`). These tables contain instances of the super-concepts (e.g., some animals in the `Plant` table, and are therefore considered overly specific. Hence, the changes to our overall process, perform merging after labeling to have semantic information available, leads to significantly better results. Not only is the label quality significantly improved and instances of a concept are no longer labeled with a sub-concept's name, but also the schema size is reduced without losing coverage.

7.4. Survey

In addition to the technical experiments above, we also presented the tables and their names to 19 persons. We evaluated two different datasets using the survey, Web-crawled data and DBpedia data. The slice of Web-crawled data we used was transformed into 113 tables using our algorithms, out of which 107 had a name candidate. The DBpedia data (we used the dataset used in the DBPSB benchmark[21] plus the Wikedia Pagelinks dataset from DBpedia) consists of roughly 450 million triples and is transformed to 298 tables. Out of them, 287 tables have a name candidate. The DBpedia dataset used for this survey is more complete than the one used in the experiments above. The afore-mentioned experiments use a smaller dataset that misses e.g., labels in foreign languages, to keep the resulting schema smaller and better analyzable. In general, the datasets are however comparable.

The survey consists of two parts. In the first part, tables are shown to the user. For each table, the number of tuples, up to eight columns and ten sample tuples (rows) are shown. We then ask the users to provide one or more names for this table. In the second part, we present the top 3 table name suggestions provided by our algorithms, and ask the users to rate these suggestions one a 5-point Likert scale (bad/poor/fair/good/excellent). Each table was evaluated by at least three different participants. By letting the users provide their ideal names first, we avoid bias introduced by seeing our suggestions. We also shuffled the list of candidates we present to the users, so that the candidate preferred by our algorithms is not always on position one. The survey instructions and an example sheet showing some tables can be found in appendix D.

	Web-crawled dataset	**DBpedia dataset**
∅ rating top 3 candidates	3.6	3.8
∅ rating table names	4.1	4.6

Table 7.8.: Survey results on a 5-point Likert scale (1=bad, 5=excellent)

The first result we obtain is the average label rating, as shown in table 7.8. We find that the DBpedia labels get a slightly better rating (3.8) than the labels of the Web-crawled dataset (3.6). We assume that

[21] DBpedia SPARQL Benchmark `http://aksw.org/Projects/DBPSB.html`

the well-maintained DBpedia dataset with its suiting ontology produces better fitting labels than the many different small ontologies used in the "dirty" Web-crawled dataset. However, both average ratings show that the overall quality of labels is quite good. When only looking at the best label among the top 3 candidates, that is also the final name chosen, we obtain average ratings of 4.1 for the Web-crawled and 4.6 for the DBpedia dataset. As these ratings are significantly better than the top 3 average rating, we find that the order of candidates defined by our algorithms is reasonable.

In addition, also new labels where suggested by the users. When looking at these labels, we find that most human-provided labels fit the ones we generated. Where they differentiate, we find different types and reasons:

More detailed labels Labels suggested by participants are more detailed and more specific, e.g., "novel" instead of "book" and "tech article" instead of "article". Other examples try to avoid homonyms, e.g. "fashion model" instead of "model".

More common words Where algorithm-wise generated labels have complex names, participants often suggested simpler terms. For example, they would name a table "shop" rather than "location of sales or service provisioning".

Deducing names Where our algorithms fail to generate a hypernym that describes the class, participants do not: they suggest e.g., "athlete" instead of "swimmer", "handball player". Another remarkable example is a table with the algorithm-wise created labels "image" and "license", that is names "copyright of images" by the participant. All these deducing example take place without the participants' knowledge of the computer-generated labels.

List of concepts Sometimes, the participants cannot think of a hypernym, and give a list of concepts instead. For example, they created the table names "plants or animals" and "prison or correctional institution".

Name for whole table instead of single tuples Participants tend do give table names in plural. Sometimes, they even give names for a *collection* of tuples rather than the tuples themselves, e.g., "news feed" instead of "article", and "anatomy" instead of "body part".

Adding indirection Participants do not always choose the name of the concept, but add a layer of indirection. For example, the RDF description of a book is not "book" for them, but "bookInfo". Other possibilities for expressing this indirection are "summary of ..." and "advertisement for ...". This indirection level was especially used in the Web-crawled dataset. We assume that it has to do with the subject URIs that look more "official" and familiar in the DBpedia dataset.

Instance vs. type problem Some concepts describe individuals (e.g., "person"), whereas others do describe sub-concepts (e.g., "animal" that does not contain famous animals but species). For example, a table listing automobile types get both "car" and "automobile type" as suggestions.

Awareness of hierarchies Participants seem to be aware of the concept of hierarchy within table names. If they provide multiple name suggestions, they usually form a hierarchy, e.g., "wrestler", "athlete", "person", "name".

Shifting the focus Participants are not interested in the same details as those provided by our generated labels. For example, the table name "Australian rules football player" gets high scores, but user-provided names skip the details: "football player". For another table, the suggested label "US athlete" shifts the focus to the nationality rather than the sports discipline, as the computer-generated label "basketball player" does.

To conclude, these results clearly show that we achieve a good overall label quality, with only few and non-systematic deviations. For example, labels are neither too specific nor too general. Sometimes, the labels based on type property values or ontology class names get low scores because the ontology/type-based names do not contain the right set of information (e.g., missing context, too specific information that the users are

not interested in). We got comments that the spelling/typography of labels needs to be improved, for example "EnglishWebsite" instead of "_website__en_gb". Implementing heuristics to create meaningful camel-case labels out of the existing ones would lead to a significantly better user experience without much effort. But in an overall view, users are satisfied with our labels, with slightly better results for the considered-cleaner DBpedia dataset.

7.5. Discussion

Having the experiments and survey results, available, we can now conclude whether our algorithms fulfill the overall goals of *MonetDB/RDF*, and where the limits of our approach are.

7.5.1. Fulfillment of Criteria

In addition to survey and experiments results, our implementation is also judged based on the criteria we established in section 3.4. Our experiments show that we are able to create a small schema (e.g., 140 tables for DBpedia dataset) that covers the aspired 80% of the triples. But, our experiments have also shown that achieving this balanced result required fine-tuning of the frequency threshold, the only parameter set by users: For the Web-crawled dataset, it is possible to achieve a schema covering the vast majority of triples using a reasonably high frequency threshold. For DBpedia, one has to set the frequency threshold low. Because of the complex class hierarchy in the DBpedia ontology, where classes contain many properties, there are orders of magnitude more possible property sets, hence many more initial CS's, with therefore lower number of subjects in it. Although it is still possible to use our algorithms with datasets based on such complex ontologies, it is more difficult to define the frequency thresholds for those datasets. Setting the frequency threshold parameter automatically would therefore not only improve the trade-off between schema size and data coverage, but also fulfill another goal, requiring as few decisions as possible from users. Some ideas on how to set the frequency threshold are discussed as a possible future work task in section 8.2. Staying at the "few user input only" goal, the only remaining user input would then be the ontologies. Shipping ontologies with *MonetDB/RDF* is difficult due to licensing issues, but a basic labeling (using incident link names) is possible even without ontologies. However, our experiments show that adding semantic information to the structuring and labeling process significantly improves the resulting schema, e.g., by reducing the number of tables with duplicated names by merging them.

Another important evaluation criterion evolving from the list of goals and criteria listed in section 3.4 is the aspired *level of detail* of both structures and labels. By choosing labels that are as specific as possible, and avoiding merging different concepts together, we achieve good results in this task. For example, our experiment with disabled semantic merging shows that overly specific tables such as `SoccerLeague` and `BasketballLeague` are merged when semantics are enabled for the merging phase. When analyzing the user-created labels in our survey, we found only few tables names that are found too generic by our users (e.g., "tech article" instead of "article"). We therefore conclude that our algorithms find the right level of detail for creating structures and labels.

Regarding the label quality, we find very good results through our survey. The criteria for label names we established in section 3.4, descriptiveness, uniqueness, and being a headword, are also fulfilled. Most labels consist of one noun, although few table names would be replaced by more common headwords (e.g., "shop" rather than "location of sales or service provisioning"). When semantic merging is enabled, only very few duplicate table names exist. These should be merged, for example using the techniques described in the discussion of the data sources experiments.

7.5.2. Limits of the Algorithms

One issue we came across is missing *versioning* in ontologies and RDF data. Although being relatively stable, ontologies do slightly change over time. For example, the class `ProductOfServices-SomeInstancesPlaceholder` of the GoodRelations ontology is replaced by the class `SomeItems`. As our algorithms currently do not leverage implicit version information via `owl:equivalentClass` markers, we treat both classes differently. As RDF data does not get updated to newer versions of the ontology automatically in non-maintained datasets such as Web crawls, both old and new version are used in parallel, and this is not reflected in the RDF data itself. The above-mentioned example of GoodRelations classes is taken from the Web-crawled dataset, where both classes are contained. Leveraging this implicit version information is further elaborated in section 8.2.

Another nice-to-have feature is better integration of hierarchies into the final relational schema. We currently leverage semantic hierarchy information for labeling and merging, but do not include them to the final schema. Although the relational model does not contain hierarchies natively, there are several options on how to include hierarchical information and therefore allow for e.g., also getting results from the `Athlete` table when the `Person` table is queried.

As we found several data errors in DBpedia during our implementation and evaluation (cf. section 4.7.2), we conclude that we are probably the first to merge type property information and ontology class information. This approach – only a side product of *MonetDB/RDF* – could further improve the quality of ontology-based datasets such as DBpedia. Section 8.2 further discusses this idea on data error detection and handling.

These and more suggestions for future extensions and improvements to *MonetDB/RDF* can be found in the future work section 8.2.

8. Conclusions and Future Work

Section 8.1 of this chapter summarizes the major points of this thesis. At last, is section 8.2, improvements to the prototype described in this thesis are listed.

8.1. Conclusions

In this thesis, we have presented *MonetDB/RDF*, an RDF store build over the RDBMS *MonetDB*. We claim that although RDF data contains no schema information (schema-last approach), it has an *emergent* schema that can be detected and exploited using the techniques described in this thesis. In contrast to previous attempts to store RDF data in *property tables* [8, 18, 34, 35] within RDBMS, we propose full transformation into relational data without human effort. Our experiments show that in fact the vast majority of RDF triples can be represented using a relatively small set of relational tables, both on well-maintained datasets such as DBpedia and considered-dirty Web-crawled RDF data.

Our *self-organizing* RDF store prototype therefore has to detect the underlying, emergent schema of an RDF dataset, create a relational schema from it, and transform all RDF data into this new structure. Using the *Characteristic Sets* approach by Neumann and Moerkotte [20] [20], *MonetDB/RDF* groups data according to the information (set of properties) available per subject. This approach is then advanced by taking relationships between these groups into account, and by extracting *concept information* from the dataset as well as from additional external semantic information. As our discovery of several inconsistencies in the popular DBpedia dataset show, we seem to be the first to try this approach of assigning concept information from different data sources to RDF data. Leveraging the available concept information, two refinements are possible: First, as the lack of self-descriptiveness of datasets is overcome by exploiting concept information, it is now possible to automatically assign human understandable names to the structures found so far. Secondly, by leveraging the concept and hierarchy information of structures, we can merge the structures to create a small, dense relational schema. The final step is now to spill the RDF triples into the relational structures. To do so, foreign key relationships have to be defined by leveraging the relationship information gathered during the transformation process. Furthermore, the data has to be transformed to SQL data types and the relational schema has to be normalized by adding additional tables for many-to-many relationships and non-atomic (multivalued) elements. The whole transformation process takes place while the RDF data is bulk-loaded into the database.

Our prototype *MonetDB/RDF* has an optimized physical storage that will be complemented by optimized operators [23] for efficient querying. Providing both an SQL and an (yet to be finished) SPARQL interface, *MonetDB/RDF* has the flexibility needed to provide as much support as possible for querying large, probably unknown, RDF datasets. When used via the SQL interface, all optimizations built into *MonetDB* can be leveraged. In addition, the large variety of SQL-based additional third-party tools can be used, for example full-text search engines or schema visualization tools. Our survey shows that the generated table names get an average score of 4.6 on a 5-point Likert scale (1 = bad, 5 = excellent). The labeled tables leads to an improved comprehensibility, and hence improved ability to pose queries against previously unknown datasets.

Being the first approach that fully transforms RDF data into relational data, we claim that semantic information should not only be used when creating RDF data or formulating SPARQL queries, but the inclusion of semantic information can also improve storage and query performance of RDF stores.

8.2. Future Work

Although already achieving notable results in creating a dense schema and generating human understandable names, we still have many improvements and enhancements to *MonetDB/RDF* in mind. The suggested future improvements can be divided into four categories: Some ideas aim to improve the algorithms described in this thesis. These suggestions are summarized in section 8.2.1. The tasks described in section 8.2.2 add new features to the module. Section 8.2.4 lists ideas for further data analysis and evaluation. Finally, for production readiness of this *MonetDB/RDF* prototype, several changes are necessary. These are listed in section 8.2.3.

8.2.1. Algorithmic Improvements

The following suggestions aim to change the algorithms to improve the quality of both structures and labels.

Usage of Type Property Values for Structuring For some predicates, the *object values* provide useful insight for the structuring process. We call these predicates *type properties*, as they are usually named *type* (e.g., `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`). We found that often subjects with the same property set do have different values for these type values, hence giving a *semantic reason* for *not* putting these subjects into the same CS. For example, structuring the DBpedia dataset led to a characteristic set with properties `type`, `name`, and `country`, that contained subjects from many different concepts, for example villages, persons, and legal cases.

As the basic mapping of subjects to CS's takes place in the exploration step described in section 3.6, that is the best phase to enrich the CS's with add type property value information. The values of type properties are added to the set of predicates that characterizes a CS. Adding type property values changes the hash value of the predicate set, hence only subjects with the same value in the type property are added to the same CS. This leads to a higher number of CS's, but they might be merged again in section 5.1. Merging ensures that the final schema will not be flooded with many CS's with similar property sets but different type property values.

This type property value information can be leveraged by the labeling phase: CS's that contain a type property could get that value as name. However, it has to be ensured that the other data sources, especially ontologies, are not obsoleted, because type property values do not always provide the best CS names. In fact, type property values tend to be on the wrong level of detail, either too generic or to specific, as discussed in section 4.6.

Order of Candidates from Different Ontologies Usually, datasets contain triples from a large variety of ontologies. Figure 8.1 shows the distribution of ontologies in the Web-crawled dataset.

Therefore, a CS might have properties from multiple ontologies. At present, when computing ontology-based candidates, each ontology that appears within a CS generates candidates, and these candidates are then merged into a candidate list. To improve the quality of this candidate lists, ontologies that cover only few properties of the CS should be left out. The first reason is that these ontologies cannot *represent* the CS content as they are only responsible for a little part of the content (triple-wise). Another reason is that similarity scores (cf.

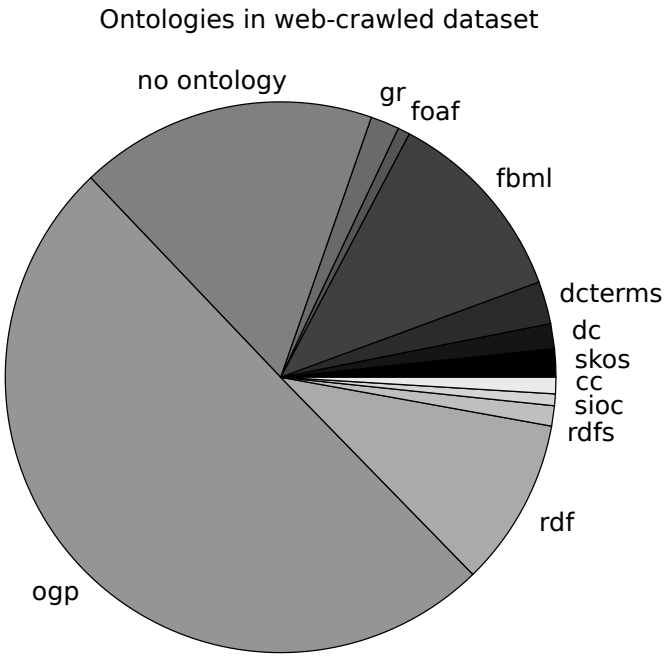


Figure 8.1.: Ontologies within the Web-crawled dataset

section 4.7.2) between a CS and such an ontology rely are not meaningful because they rely on few properties only. We suggest a threshold of e.g. 20% of the properties.

Subject URIs as Additional Data Source The last part of subject URIs (as well as the value of a `label` property if available) usually contain a good description or name of the subject. One could try to infer the common concept of these descriptions and use this common concept as CS name. Venetis et al. [31] use a database that contains hyponym/hypernym information to do exactly this.

No Dummy Table Names If no label candidate is available, a dummy name is assigned to a CS/table as shown in section 4.9. It might however be better to assign a name that was not chosen as candidate because it did not exceed a threshold, than not choosing any name. To implement this feature, one has to keep the non-candidates of at least one data source and choose one if no other name is available. Every name is better than `DUMMY`.

Transformation of URIs At the moment, syntactical differences in URIs are not recognized. For example, if an ontology prefix sometimes starts with `http` and sometimes with `https`, we do not group these variances together. Another example are URL redirects such as `ogp.me`[22] being an abbreviation for `opengraphprotocol.org`

Besides these syntactical differences, equivalences can also be defined on a semantic level, using the `owl:sameAs` or `owl:equivalentClass` properties. `owl:sameAs` defines equivalences between

[22]Facebook Open Graph Protocol `http://ogp.me/`

instances and can hence be used to merge two subjects into one. In contrast, `owl:equivalentClass` defines class-level equivalence, hence is relevant for e.g., table names.

Exploiting and Loading More Information from Ontologies As mentioned above, using the `owl:sameAs` property when loading ontologies would probably improve the overall quality of structures and labels. The same holds for some more properties in ontology definitions, e.g., `owl:equivalentClass` and `rdfs:subPropertyOf`. The first one is used to define equivalence between different classes, for example when a class is renamed. By using `owl:equivalentClass`, the old, deprecated class can be marked as equivalent to the new class, hence introducing backward-compatibility. The second one, `rdfs:subPropertyOf`, defines a hierarchy of properties that we do not use at the moment. It could be a good idea to collapse property hierarchies to ensure that we do not miss relationships between properties but still keep the schema simple.

There are even more properties in ontology definitions one could look at, such as `owl:disjointWith` that defines that a subject cannot be an instance of two disjoint classes. This information could be used to ensure that the concepts introduced by the two disjoint classes are kept separate structure-wise in any case.

Including Hierarchy Information into Relational Schema We currently do not indicate hierarchies in the relational schema. If a super-concept table and a sub-concept table exist, there is no hint in the relational schema that they are related and share properties. Several methods exist to represent a hierarchy in relational data. These techniques are necessary for e.g., *object-relational mapping* and could be used in *MonetDB/RDF*, too.

Spelling of Table Names As mentioned in section 7.4, camel-case labels are preferred by our survey participants. Currently, the spelling is taken over from the different ontologies the labels come from, hence the labels do use different notations. A consistent camel-case syntax is a low-hanging fruit when it comes to improving the user experience with our SQL interface.

Column Order Venetis et al. [31] benefit from the intuitive order of human-made tables with the subject column on the left. To achieve a human-made look for our SQL schema, we should also care about the order of the columns. Of course, the subject column that contains the URIs per subject should be the left-most columns. Other primary key candidate columns should follow. Afterwards, the more NULL values a column contains, the righter it should be placed in the table. Another approach would be using the tf-idf values of columns rather than the number of NULL values in it. A third idea is exploiting the order of predicates in the ontology. This idea assumes that the order of properties in the class definition was carefully chosen, just as Matono and Kojima [18] argue about the order of triples in a dataset. No matter which idea is chosen, it will be better than the order at present that is defined by when a property was first found while loading the dataset.

Detecting and Dealing with Data Errors Web-crawled datasets, but also maintained datasets such as DBpedia, contain data errors. For example, the current version of DBpedia classifies the topic "Online_backup_service" as a music genre. If the type information is used, the subject will be grouped together with other music genres. On the other hand, if the property set of this subject is compared to ontology classes, it does not contain properties that are typical for music genres. Therefore, by combining our two data sources *property types* and *ontologies*, we are able to detect data errors. These erroneous subjects can then be put into the PSO triple store. This error checking task would have to take place on an instance level, not on a CS level, and therefore consumes huge amounts of resources for large datasets. Therefore,

before including it into the *MonetDB/RDF* code, the performance of this task has to be evaluated. However, this kind of analysis is interesting for dataset maintainers, too, and could therefore be tackled in the future.

Changing Parameter set by Users Currently, the only parameter presented to the users is the frequency threshold for initial CS detection. This parameter is not easy to understand because it does not directly influence the minimal size of the resulting tables, or any other interesting measure. Instead of letting users set this parameter, it would be better to let them define the percentage of data that should be covered by the relational schema, or a maximum number of tables in the resulting schema, or other useful thresholds. In opposition the currently used parameter, these new suggestions cannot be set accurately within the algorithms. Instead, the algorithms will have to be adjusted to meet the required threshold as accurate as possible.

8.2.2. New Features

The following features were out of scope for this thesis. However, they should be implemented as part of the *MonetDB/RDF* project and are tightly connected to the work described in this thesis.

Zooming and Exploring a Schema Although we try to create a reasonable small schema, it might still be too big to be explored by simply looking at it. A possibility to show only few, important tables first and then allow for *zooming into* the schema would simplify the exploration of big schemas. To provide a *first overview*, one can simply select the biggest tables (and maybe also the important dimension tables). Other possibilities would be scoring the importance of tables by an iterative measures such as a page-rank-like criterion. For *zooming*, things get more complicated. If the user wants to zoom into a specific table, a given number of tables (including the chosen one) should be shown can be reached from the chosen table by using *foreign keys* and that cover *as many data as possible*. This problem is known as *(Rooted) Node-Weight Connected Subgraph Problem with Budget Constraint (B-RMWCS)* and is NP-hard [2]. We therefore propose using heuristics to compute a set of tables to be shown when zooming into a table.

Searching a Schema To quickly find a certain table in a big schema, a keyword search for table names and property names would be helpful. Answering keyword search queries on a static set of data with a list of relevant entities is referred to as *ad-hoc object retrieval* [25]. The order of the results is determined using a *weighting function* that needs to be adjusted to fit the requirements. The well-known weighting function BM25 [26] and its refined version BM25F [27] define the relevance of a document on the basis of how often the keywords appear within the document in comparison to how often they appear in other documents (*tf-idf*). Search frameworks have to be adjusted to query schema metadata only and to retrieve a table/view. Besides names and properties, one could store additional table name candidates and use them for the keyword search.

Updating Datasets Currently, *MonetDB/RDF* does not support updates to the loaded dataset (defined as non-goal in section 1.2.2). Neither may triples be changed, nor may new triples be added to the dataset. However, in practice, one wants to add more RDF data to the dataset even if it already has been transformed into a relational schema. The easiest way to achieve this would be adding new triples to the PSO store, hence no structuring/merging or labeling algorithms would have to be changed. This approach has two major disadvantages: *i)* The triple store grows and grows and the schema will eventually degenerate into an ordinary triple store, and *ii)* new concepts that are added to the dataset are not recognized and therefore not added to the relational schema. It is therefore reasonable to load the new triples into the PSO table first to have them available immediately, but to run the whole structuring, labeling, and merging process as soon there

is a sufficient amount of new triples. A requirement we expect when integration this update mechanism is the re-use of the old relational schema. If the relational schema would be allowed to change totally after an upgrade, all applications that rely on that schema would have to be rewritten. The update procedure should therefore keep the old schema and enhance it, for example by adding new columns to tables, add foreign key constraints, or add new tables.

8.2.3. Production Readiness and Integration into *MonetDB*

The prototype designed as part of this thesis lacks some essential configuration possibilities before non-experts can operate it.

Fine-Grained Labels per Row As introduced in section 4.9.2, we store a label per subject that can provide more fine-grained information about tuples than the table name. Until now, these labels are not shown to the user. They should be made available upon user request.

Loading Ontologies At present, ontologies that the users want to be available in *MonetDB/RDF* have to be transformed to a simple CSV format. The transformations necessary to get this format are described in appendix B. To allow for a simpler upload of ontologies, we need to provide an interface that also checks the formal correctness of the given files. Furthermore, we need to output the list of currently loaded ontologies. We need to provide instructions on how to transform ontologies, and maybe even create a syntax specification or kind of *meta-ontology* for this purpose.

Loading Ontology Prefixes and Type Properties Two lists of ontology prefixes and type properties are used within *MonetDB/RDF*. These lists have been created to contain values for the most popular ontologies, but do not cover all possible ontologies. It is therefore necessary to allow for enhancements to these lists. The list containing *ontology URI prefixes* is used for URI shortening. If the prefix of a property is found in this list, it is easy to extract the non-prefix part. For properties whose ontologies are not in the list, the heuristic described by Neumayer et al. [21] is used to shorten the property URI. Including more ontology prefixes into the list can therefore lead to better abbreviated table and property names. The list of *type properties* is used for extracting candidates from type property values. If type properties of a used ontology are not on the list, their values will not be considered for table names. Having an incomplete type properties list will therefore lead to significantly worse label suggestions by the best label source.

JDBC Interface SQL tools that provide a graphical overview of the schema are helpful for exploring the schema, as stated above. These tools use a “get all tables” command offered by JDBC to collect the metadata needed to display the relational schema. A general JDBC interface is already available within *MonetDB*. One could adjust this interface to allow for zooming as mentioned afore-head. If zoomed in, the JDBC “get all tables” should return only a subset of tables and foreign key relationships.

8.2.4. Analysis and Evaluation

Further improvements can be identified if more evaluation and analysis of datasets and results takes place.

Extended Survey At a later point in time, an additional, extended survey should be taken. The first idea for this survey is asking for label ratings in different dimensions (level of detail, syntax/notion of label, ...) to get a more detailed view of the label quality. Another idea is to let users rank label candidates and compare their rankings with the one made by our algorithms. Systematic deviations could then be analyzed to improve the label assignment algorithm (cf. section 4.9) that defines the candidate order. One cause for bad label quality and too many NULL values in a CS is the mixture of two different concepts into one CS. An interesting survey would therefore be to ask users to cross out columns that do not belong to a CS. Another idea is to let users categorize tuples, by pointing out in which table (by name) they would fit best. This hand-made mapping can then be compared to the table the subjects were put in by our algorithms. This extended survey could also be send to more and heterogeneous participants by using an online crowdsourcing platform.

Analysis of Outlying Data We currently drop irregular data at different positions in out algorithms. If the outlying data still has some structure in it, one could find better data structures than the PSO triple store to store outliers in. Other implications might be changed thresholds because too much – or too few – data is dropped.

A. Hierarchies in Ontologies

During the work described in this thesis we analyzed seven ontologies. All of them have a hierarchy in their list of classes. Some have a single hierarchy with a top class that covers all classes, others have multiple hierarchies to represent multiple base concepts, and others have only few hierarchical structures to describe some specialized subclasses only. Some ontologies include classes from foreign ontologies in their hierarchy, e.g. as base concept they derive their classes from.

DBpedia All classes are arranged into one hierarchy with `owl:Thing` being the top class. This behavior is suggested by the W3C Recommendation on OWL 2 [19]: "*owl:Thing* represents the set of all individuals."

Eurostat[23] Only few classes exist, and these are all member of a two-level hierarchy with `e:Region` being the top class.

Friend of a Friend (FOAF) Four hierarchies exist, with the top classes `foaf:Agent`, `foaf:Document`, `foaf:LabelProperty`, and `foaf:Project`.

GoodRelations Most classes in this ontology do not belong to a hierarchy, but there are small hierarchies for e.g., price specifications and payment methods.

Lehigh University Benchmark (LUBM)[24] Multiple hierarchies exist within this ontology.

Semantic Web for Research Communities (SWRC)[25] This ontologies assigns its classes to multiple ontologies.

DublinCore[26] Only few classes belong to small, flat hierarchies.

[23] Eurostat `http://eurostat.linked-statistics.org/`
[24] LUBM `http://swat.cse.lehigh.edu/projects/lubm/`
[25] SWRC `http://ontoware.org/swrc/`
[26] DublinCore `http://dublincore.org/schemas/rdfs/` (dcterms, dctype, dcam, dcelements)

B. Transforming and Loading Ontologies

Ontologies are crucial within *MonetDB/RDF*, as they are an important semantic data source used for both structuring/merging and labeling RDF data. Two types of information taken from ontologies are required within *MonetDB/RDF*. First, the hierarchy of classes. Second, the properties per class. At present, this information is loaded using a simple CSV format, as shown in figure B.1.

```
class URI | class label | parent URI or NULL

class URI | property URI
```

Figure B.1.: CSV format for subclass-superclass information as well as classes and their attributes

Most ontologies are available in RDF/XML format, that can easiest by transferred to the above-mentioned CSV format by querying it with SPARQL. For example, the queries to transform the DBpedia ontology are shown in figures B.2 and B.3. As presented in section 8.2, we are planning to replace the simple CSV format with a simple *meta-ontology*.

```
SPARQL
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT ?s ?label ?p FROM <http://dbpedia.org/ontology>
WHERE {
    ?s rdf:type owl:Class.
    ?s rdfs:label ?label.
    FILTER(langMatches(lang(?label), "en")).
    OPTIONAL ?s rdfs:subClassOf* ?p
};
```

Figure B.2.: SPARQL query to get subclass-superclass information from the DBpedia ontology

Figure B.2 gets all classes and their English labels as well as – if available – their parents. The transitive `rdfs:subClassOf*` ensures that also grandparents etc. are stored for each class.

```
SPARQL
[...]
SELECT DISTINCT ?sc ?p FROM <http://dbpedia.org/ontology>
WHERE {
    ?s rdf:type owl:Class.
    ?p rdfs:domain ?s.
    ?sc rdfs:subClassOf* ?s
} GROUP BY ?s;
```

Figure B.3.: SPARQL query to extract classes and their properties from the DBpedia ontology

Figure B.3 extracts all classes and their properties. Properties are attached to classes using `rdfs:domain`. Properties are inherited. Hence, if a property is attached to a superclass, it is also available in its subclasses. By using the transitive `rdfs:subClassOf*`, this constraint is reflected.

C. Type Properties Available in *MonetDB/RDF*

We include 18 type properties in *MonetDB/RDF*. These type properties are used for detecting type information for labeling, as elaborated in section 4.6. The third one, `rdf:type`, is the most popular one.

```
char* typeAttributes[] = {
    "<http://ogp.me/ns#type>",
    "<https://ogp.me/ns#type>",
    "<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>",
    "<http://purl.org/dc/elements/1.1/type>",
    "<http://mixi-platform.com/ns#type>",
    "<http://ogp.me/ns/fb#type>",
    "<http://opengraph.org/schema/type>",
    "<http://opengraphprotocol.org/schema/type>",
    "<http://purl.org/dc/terms/type>",
    "<http://purl.org/goodrelations/v1#typeOfGood>",
    "<http://search.yahoo.com/searchmonkey/media/type>",
    "<https://opengraphprotocol.org/schema/type>",
    "<https://search.yahoo.com/searchmonkey/media/type>",
    "<http://www.w3.org/1999/xhtmltype>",
    "<http://dbpedia.org/ontology/longtype>",
    "<http://dbpedia.org/ontology/type>",
    "<http://dbpedia.org/ontology/typeOfElectrification>",
    "<http://dbpedia.org/property/type>"
};
```

Figure C.1.: List of type properties available in *MonetDB/RDF*

D. Survey

The motivation for our questionnaire and its results are discussed in section 7.4. Two example tables presented in the survey are shown in figure D.3. The labels suggested by our algorithms are shown in figure D.1. The users were asked to rate these suggestions on a 5-point Likert scale (bad/poor/fair/good/excellent). Along with the survey sheets, we sent instructions on how to fill in the survey, cited in figure D.2.

Table	Name	Rating
Table 171, 678 tuples	class	
	rc_Fish	
Table 182, 10347 tuples	AmericanFootballPlayer	

Figure D.1.: Name suggestions
Up to three name suggestions in random order

Thank you very much for joining our survey on evaluating the quality of labeling tables extracted from RDF datasets. Please read the description before starting the survey.
This survey consists of two parts. In the first part, you will see a set of tables with their sample data (i.e., 10 tuples for each table). For the convenience of the representation, each table will be represented with a limited number of columns (i.e., up to 8). The detailed representation of each table is as follows.

Table number, number of tuples
Columns which are not shown with sample data
Truncated set of columns with 10 sample tuples

Columns marked with "*" are multi-values column. The values in this column are separated by ";". "[Column]->[TableName]" indicates that the column is a foreign key pointing to the table [TableName].
Please suggest (one or more) names for each table that you think describe the table and its content best. Please write the suggested names at the end of each table. Please finish this part of the survey before starting with the second part.
In the second part, you will see a set of names suggested by our algorithm for each table. Note that the candidates are provided in a random order that does not reflect the priority assigned to each name by our algorithm. By comparing each suggestion with the table and the name(s) you choose, please rate their quality using the following ratings.

1=bad
2=poor
3=fair
4=good
5=excellent

Thank you very much!

Figure D.2.: Instructions

Table 171, 678 tuples

Additional columns: subject*, depiction, thumbnail, hasPhotoCollection, abstract*, comment*, sameAs*

Subject	type*	label*	wasDerivedFrom	wikiPageID	wikiPageRedirects	wikiPageWikiLink*	isPrimaryTopicOf	wikiPageRevisionID
org/resource/Barred_gudgeon>/	rc_Fish;	Bostrychus zon	<Barred_gudgeo	12604548	NULL	Endemism;Fish	<Barred_gudgeo	540704899
esource/Chromis_brevirostris>/	rc_Fish;	Chromis brevirо	<Chromis_brevi	31774660	<Shortsnout_Ch	Shortsnout_Chr	<Chromis_brevi	429091758
.org/resource/Barbatula_tigris>/	rc_Fish;	Barbatula tigris;	<Barbatula_tigri	32874151	<Oxynoemache	Oxynoemacheil	<Barbatula_tigri	531644849
dia.org/resource/Carp_bream>/	rc_Animal;rc_Fi	Carp bream;	<Carp_bream?o	29627505	<Common_brea	Common_bream	<Carp_bream>	397167324
source/Squalius_cephaloides>/	rc_Fish;	Squalius cephal	<Squalius_ceph	12185304	<European_chu	European_chub;	<Squalius_ceph	459973033
oplus_sp._nov._'Dridri_mena'>/	rc_Fish;	Paretroplus sp.	<Paretroplus_sp	12620437	<Damba>	Damba;	<Paretroplus_sp	439310179
dia.org/resource/Crying_izak>/	rc_Fish;	Crying izak;	<Crying_izak?o	29128506	<Crying_Izak>	Crying_Izak;	<Crying_izak>	409870637
urce/Sumireyakko_venustus>/	rc_Fish;	Sumireyakko ve	<Sumireyakko_	26696883	<Centropyge_ve	Centropyge_ver	<Sumireyakko_	351992421
a.org/resource/Redside_Dace>/	rc_Fish;	Redside Dace;	<Redside_Dace	35873154	<Redside_dace	Redside_dace;	<Redside_Dace	493373854
e/Oncorhynchus_formosanus>/	rc_Fish;	Oncorhynchus f	<Oncorhynchus	27756732	<Oncorhynchus	Oncorhynchus_	<Oncorhynchus	368880794

Table 182, 10347 tuples

Additional columns: undraftedYear, debutTeam->SoccerClub, formerTeam*, status, activeYearsEndYear, activeYearsStartYear, weight, team*, height, deathPlace*, deatl

Subject	type*	label*	draftRound	draftPick	draftYear	number	position	Person_weight
rg/resource/Roderick_Rogers>/	rc_FootballPlay	Roderick Roger	NULL	NULL	NULL	NULL	Safety (America	84.8232
org/resource/Reggie_Rhodes>/	rc_FootballPlay	Reggie Rhodes;	NULL	NULL	NULL	NULL	Defensive linem	136.08
Hamilton_(American_football)>/	Person;Person;	Justin Hamilton	7	222	2006	NULL	Safety	100.6992
dia.org/resource/Ray_Zellars>/	rc_FootballPlay	Ray Zellars;	2	44	1995	34	Running back	105.6888
il_Smith_(American_football)>/	rc_Athlete;rc_Fo	Neil Smith (Ame	1	2	1988	9091	Defensive end	NULL
Williams_(offensive_lineman)>/	Person;Person;	Keith Williams (	6	196	2011	68	Offensive Guard	149.688
org/resource/Antoine_Bethea>/	rc_Athlete;rc_Fo	Antoine Bethea;	6	207	2006	41	Free safety	92.988
.org/resource/Mike_L._Jones>/	Person;Person;	Mike L. Jones;	2	14	1990	NULL	Tight end / Qual	106.596
rg/resource/Desmond_Bryant>/	Person;Person;	Desmond Bryan	NULL	NULL	NULL	92	NULL	141.0696
cob_Bell_(American_football)>/	rc_Athlete;rc_Fo	Jacob Bell (Am	5	138	2004	606362	NULL	137.8944

Figure D.3.: Table description

Table name and size, Columns that are not chosen for in-depth representation, subject and up to eight columns

Bibliography

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proc. of the 33rd VLDB*, pages 411–422, 2007.

[2] Eduardo Álvarez Miranda, Ivana Ljubić, and Petra Mutzel. The Rooted Maximum Node-Weight Connected Subgraph Problem. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 300–315. 2013.

[3] Michele Banko, Michael J. Cafarella, Stephen Soderland, Matt Broadhead, and Oren Etzioni. Open Information Extraction from the Web. In *Proc. of the 20th ICJAI*, pages 2670–2676, 2007.

[4] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 / STD 66, 2005.

[5] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an Efficient RDF Store over a Relational Database. In *Proc. of the 2013 SIGMOD*, pages 121–132, 2013.

[6] Andreas Brodt, Oliver Schiller, and Bernhard Mitschang. Efficient Resource Attribute Retrieval in RDF Triple Stores. In *Proc. of the 20th CIKM*, pages 1445–1454, 2011.

[7] Sören Brunk and Philipp Heim. tfacet – Hierarchical Faceted Exploration of RDF Data. `http://www.visualdataweb.org/tfacet.php`, accessed 2014-05-20.

[8] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proc. of the 31st VLDB*, pages 1216–1227, 2005.

[9] Filippo Geraci, Marco Pellegrini, Marco Maggini, and Fabrizio Sebastiani. Cluster Generation and Cluster Labelling for Web Snippets: A Fast and Accurate Hierarchical Solution. In *String Processing and Information Retrieval*, pages 25–36. 2006.

[10] Sunil Goyal and Rupert Westenthaler. RDF Gravity (RDF Graph Visualization Tool). `http://semweb.salzburgresearch.at/apps/rdf-gravity/`, accessed 2014-05-20.

[11] Philipp Heim. gfacet – Graph-based Faceted Exploration of RDF Data. `http://www.visualdataweb.org/gfacet.php`, accessed 2014-05-20.

[12] Martin Hepp. Goodrelations: An ontology for describing products and services offers on the web. In *EKAW*, pages 329–346, 2008.

[13] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[14] Martin Kavalec and Vojtěch Svaték. A Study on Automated Relation Labelling in Ontology Learning. In *Ontology Learning from Text: Methods, Evaluation and Applications*, pages 44–58. 2005.

[15] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`, 1999.

[16] Justin J. Levandoski and Mohamad F. Mokbel. RDF Data-Centric Storage. In *Proc. ICWS*, pages 911–918, 2009.

[17] Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct. pages 1648–1653, 2009.

[18] Akiyoshi Matono and Isao Kojima. Paragraph Tables: A Storage Scheme Based on RDF Document Structure. In *DEXA (2)*, pages 231–247, 2012.

[19] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C Recommendation `http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/`, 2012.

[20] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *Proc. of the 27th ICDE*, pages 984–994, 2011.

[21] Robert Neumayer, Krisztian Balog, and Kjetil Nørvåg. When Simple is (more than) Good Enough: Effective Semantic Search with (almost) no Semantics. In *Advances in Information Retrieval*, pages 540–543, 2012.

[22] Zuzana Nevěřilová. Visual browser. `http://nlp.fi.muni.cz/projekty/visualbrowser/`, accessed 2014-05-20.

[23] Minh-Duc Pham. Self-organizing Structured RDF in MonetDB. In *Proc. of ICDE/PhD Symposium 2013*, 2013.

[24] Minh-Duc Pham, Linnea Passing, and Peter Boncz. Discovering the Emergent Schema of RDF Data. submitted for publishing.

[25] Jeffrey Pound, Peter Mika, and Hugo Zaragoza. Ad-hoc Object Retrieval in the Web of Data. In *Proc. of the 19th WWW*, pages 771–780, 2010.

[26] Stephen Robertson and Hugo Zaragoza. The Probabilistic Relevance Framework; BM25 and Beyond. *Found. Trends Inf. Retr.*, pages 333–389, 2009.

[27] Stephen Robertson, Hugo Zaragoza, and Michael Taylor. Simple BM25 Extension to Multiple Weighted Fields. In *Proceedings of the thirteenth CIKM*, pages 42–49, 2004.

[28] Gerard Salton and Michael J McGill. Introduction to Modern Information Retrieval. 1983.

[29] Toby Segaran, Colin Evans, Jamie Taylor, Segaran Toby, Evans Colin, and Taylor Jamie. *Programming the Semantic Web*. 1st edition, 2009.

[30] Pucktada Treeratpituk and Jamie Callan. Automatically Labeling Hierarchical Clusters. In *Proc. dg.o*, pages 167–176, 2006.

[31] Petros Venetis, Alon Halevy, Jayant Madhavan, Marius Paşca, Warren Shen, Fei Wu, Gengxin Miao, and Chung Wu. Recovering Semantics of Tables on the Web. In *Proc. of the 37th VLDB*, pages 528–538, 2011.

[32] World Wide Web Consortium (W3C). W3C Semantic Web Activity. `http://www.w3.org/2001/sw/`, accessed 2014-05-20.

[33] Qinglei Wang, Yanan Qian, Ruihua Song, Zhicheng Dou, Fan Zhang, Tetsuya Sakai, and Qinghua Zheng. Mining subtopics from text fragments for a web query. *Information Retrieval*, pages 1–20, 2013.

[34] Yan Wang, Xiaoyong Du, Jiaheng Lu, and Xiaofang Wang. Flextable: Using a Dynamic Relation Model to Store RDF Data. In *Database Systems for Advanced Applications*, pages 580–594. 2010.

[35] Kevin Wilkinson. Jena Property Table Implementation. In *Proc. ISWC/SSWS*, pages 54–68, 2006.