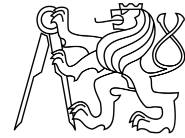
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Distributed Conversion of RDF Data to the Relational Model
Student:	Bc. David P íhoda
Supervisor:	Ing. Ivo Lašek, Ph.D.
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2017/18

Instructions

Become familiar with the RDF model for data exchange. Explore and compare current approaches and tools for the conversion of RDF data to the relational model.

Design and implement a tool for converting RDF data to the relational model. The tool will be able to run distributively in a cluster as well as locally on a single machine. Demonstrate the tool's ability to convert big datasets and load the results to a commonly used RDBMS (e.g. , PostgreSQL). Demonstrate the tool's ability to load a variety of different sources (preferably from the group of Bio2RDF datasets: http://bio2rdf.org).

The thesis will include:
* Survey of existing approaches and tools.
* Theoretical analysis.
* Ready to use conversion tool based on the Apache Spark framework.
* Java library ready to use in different Java projects.
* Use case demonstration and examples.
* Functional and performance testing.

References

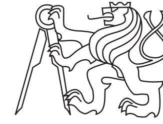
Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague January 27, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of theoretical computer science



Master's thesis

Distributed Conversion of RDF Data to the Relational Model

Bc. David Příhoda

Supervisor: Ing. Ivo Lašek, Ph.D.

9th May 2017

Acknowledgements

I would like to thank my supervisor Ivo Lašek for advice and discussions during the preparation of this thesis. I would also like to thank my friends and family for the great support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 9th May 2017 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2017 David Příhoda. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Příhoda, David. *Distributed Conversion of RDF Data to the Relational Model.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017. Also available from: ⟨`http://davidprihoda.com/rdf2x`⟩.

Abstrakt

Ve formátu RDF je ukládán rostoucí objem hodnotných informací. Relační databáze však stále přinášejí výhody z hlediska výkonu a množství podporovaných nástrojů. Představujeme RDF2X, nástroj pro automatický distribuovaný převod RDF dat do relačního modelu. Poskytujeme srovnání souvisejících přístupů, analyzujeme měření převodu 8.4 miliard RDF trojic a ilustrujeme přínos našeho nástroje na dvou případových studiích.

Klíčová slova Linked Data, RDF, RDB, Relační model, Bio2RDF

Abstract

The Resource Description Framework (RDF) stores a growing volume of valuable information. However, relational databases still provide advantages in terms of performance, familiarity and the number of supported tools. We present RDF2X, a tool for automatic distributed conversion of RDF datasets to the relational model. We provide a comparison of related approaches, report on the conversion of 8.4 billion RDF triples and demonstrate the contribution of our tool on case studies from two different domains.

Keywords Linked Data, RDF, RDB, Relational model, Bio2RDF

Contents

Introduction **1**
Thesis structure 1

1 Background **3**
1.1 RDF: Resource Description Framework 3
1.2 Relational databases 5
1.3 Apache Spark 8

2 Related work **11**
2.1 Converting relational data to RDF 11
2.2 Using RDBMS for RDF storage 14
2.3 Converting RDF to the relational model 16

3 Design **23**
3.1 Requirements 23
3.2 Architecture 24
3.3 Conversion process 25

4 Implementation **29**
4.1 Tools and libraries 29
4.2 Implemented features 30
4.3 RDF2X Library 36

5 Bio2RDF Conversion Statistics **39**
5.1 Introduction 39
5.2 Explored datasets 40
5.3 Data preparation 40
5.4 Conversion 40
5.5 Content statistics 42
5.6 Conclusion 43

6 Case study I: Visualizing clinical trials **45**
6.1 Introduction 45
6.2 Data preparation 46
6.3 Conversion 46
6.4 Schema visualization 47
6.5 Data visualization 47
6.6 Conclusion 52

7 Case study II: Querying Wikidata with SQL **55**
7.1 Introduction 55
7.2 Data preparation 56
7.3 Wikibase data model 56
7.4 Conversion 58
7.5 Produced schema 58
7.6 SQL experiments 58
7.7 Conclusion 66

Conclusion **67**
Future work 67

Bibliography **69**

A Contents of CD **75**

List of Figures

1.1 Visualization of a simple RDF graph 3
1.2 Turtle RDF serialization example 4
1.3 SPARQL query example 6
1.4 SQL query example 7
1.5 Spark cluster architecture 9

2.1 Direct mapping example 12
2.2 R2RML mapping example 13
2.3 Vertical table schema example 15
2.4 Horizontal table schema example 15
2.5 Property table schema example 16
2.6 Star/snowflake schema production based on TPTS approach . . . 19
2.7 Number of SQL queries produced by RDF2RDB 20

3.1 RDF2X architecture 24
3.2 RDF2X conversion process 25

4.1 Conversion input RDF input file in Turtle format 31
4.2 Produced entity tables 32
4.3 Produced Entity-Attribute-Value table 33
4.4 Produced relation tables 34
4.5 Schema of metadata tables 35

5.1 RDF2X conversion time of selected Bio2RDF datasets. 41
5.2 Duration of RDF2X conversion phases. 41
5.3 Influence of minNonNullFraction on table sparsity 42

6.1 Graph visualization of ClinicalTrials.gov entity relationships. . . . 47
6.2 Data Source definition in Tableau. 48
6.3 Number of clinical studies per start year. 48
6.4 Distribution of clinical study duration. 48

6.5 Number of clinical studies by country 49
6.6 Percentage of studies by minimum and maximum required age. . . 49
6.7 Percentage of studies by two eligibility criteria 50
6.8 Most frequent clinical study condition labels and terms. 50
6.9 Number of studies per year for the most frequently studied conditions. 51
6.10 Frequency of drug labels and intervention terms 52
6.11 Number of studies per year for the most frequent intervention terms. 52
6.12 Top 3 most frequent interventions for the top most frequently studied conditions 53

7.1 Example of Wikidata RDF statements 57
7.2 Schema of selected Wikidata entity tables. 59

List of Tables

1.1 Relational database example 6

2.1 Relational data input for RDB to RDF conversion 12
2.2 Comparison of RDF to RDB conversion tools 22

5.1 Size of selected Bio2RDF datasets 42

Introduction

This thesis presents the design and development of RDF2X, a tool for automatic distributed conversion of RDF datasets to the relational model.

Both RDF and the relational model have their specific advantages and disadvantages. When automatic data conversion between different representations is possible, we can freely decide which representation is best fit for our use case and exploit its specific properties. We believe that our conversion tool is the missing piece of the interoperability puzzle between Linked Data and traditional formats, allowing for new opportunities in RDF usage and therefore contributing to its expansion.

RDF data is often created from relational databases and other sources. An integral part of converting data to RDF is mapping it to a common ontology, connecting corresponding entities and properties from different sources. Using our tool, such integrated datasets can be converted back to the relational model. This creates a new range of possibilities for using RDF as a stepping stone in relational database data integration.

Thesis structure

In the first chapter, we provide an introduction to the problem domain. In the second chapter, we summarize and compare related approaches. In the third chapter, we propose requirements and design of the RDF2X tool. In the fourth chapter, we describe the implemented features. In the fifth chapter, we report on the conversion of datasets from the Bio2RDF project[1]. In chapter six, we provide a high-level demonstration of our tool by converting and visualizing the ClinicalTrials.gov RDF dataset[2]. Finally, in chapter seven, we provide a low-level demonstration by converting a subdomain of Wikidata[3] and exploring the result with SQL queries.

[1] http://bio2rdf.org

[2] http://download.bio2rdf.org/release/3/release.html

[3] http://wikidata.org

Chapter 1

Background

In this chapter, we provide an introduction to the problem domain: the Resource Description Framework, relational databases and the Apache Spark computing platform.

1.1 RDF: Resource Description Framework

The Resource Description Framework (RDF) is a framework for representing information in the Web [1]. Since its publication in 2004, RDF has become a widely used standard for the integration of heterogeneous data sources, introducing repositories in various domains including life sciences [2][3], government [4], finance [5] and general knowledge [6][7][8].

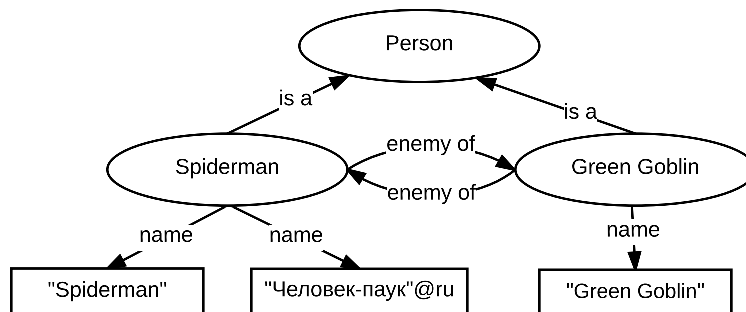


Figure 1.1: Visualization of a simple RDF graph, inspired by Example 1 in [9]. Ovals represent resources, rectangles represent literal values, arrows represent relationships.

RDF describes resources and their relationships in the form of a graph. An example of a RDF graph can be found on figure 1.1. Each graph is constructed by a collection of RDF triples, each triple consists of a subject, a predicate and an object:

- the **subject** is an *IRI* or a *blank node*
- the **predicate** is an *IRI*
- the **object** is an *IRI*, a *literal* or a *blank node*

IRIs are globally unique strings that identify resources and properties. Blank nodes are resource identifiers valid only in the context of a single document. Literals are values such as strings, numbers, and dates. Each triple can therefore express either a literal property of a resource (when the object is a literal) or a relationship between two resources (when the object is an IRI or a blank node). In both cases, the type of the relationship is specified by the predicate.

1.1.1 Serialization formats

The RDF concept itself defines only an abstract syntax, not a specific serialization format. There are multiple popular formats that are commonly used to represent RDF statements.

Turtle Turtle is a textual representation of RDF written in a compact and natural text form, with abbreviations for common usage patterns and datatypes [9]. An example of Turtle serialization of the graph on figure 1.1 can be found on figure 1.2.

```
@base <http://example.org/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rel: <http://www.perceive.net/schemas/relationship/> .

<#green-goblin>
    rel:enemyOf <#spiderman> ;
    a foaf:Person ;
    foaf:name "Green Goblin" .

<#spiderman>
    rel:enemyOf <#green-goblin> ;
    a foaf:Person ;
    foaf:name "Spiderman", "Человек-паук"@ru .
```

Figure 1.2: Turtle RDF serialization representing the graph on figure 1.1.

N-Quads and N-Triples N-Quads and N-Triples are line-based, plain text representations of RDF [10][11]. Their advantage over Turtle is that since they store one statement per line, they can be read one line at a time without storing intermediate information such as abbreviations. This also implies that they can be parsed in parallel without the need for any communication.

1.1.2 RDF Schema

To introduce structure into RDF documents, RDF Schema[12] was designed. It introduces a vocabulary that makes it possible to describe special types of resources called *classes* and *properties*.

Classes define groups of RDF resources. Classes themselves are RDF resources, instances of `rdfs:Class`. To state that a resource is an instance of a class, the `rdf:type` property may be used. To state that all instances of class `A` are also instances of class `B`, an `A rdfs:subClassOf B` statement can be used.

Properties define relationships between RDF resources. Properties themselves are RDF resources, instances of the `rdf:Property` class. To state that all pairs of instances related by property `A` are also related by property `B`, an `A rdfs:subPropertyOf B` statement can be used.

To further describe and augment the meaning of *classes*, *properties* and their relationships, Web Ontology Language (OWL) [13] can be used. OWL enables creation of *ontologies*, formal definitions of a common set of terms that are used to describe and represent a domain [14]. For example, `owl:equivalentClass` states that two classes are equivalent, `owl:sameAs` states that two resources represent the same individual, enabling integration of data and schema from different sources.

1.1.3 SPARQL

SPARQL is a query language for RDF, introducing capabilities for graph pattern matching, aggregation, subqueries and other functionality [15]. Many Linked Data sources such as Wikidata[1], DBpedia[2] or Bio2RDF[3] maintain a public SPARQL endpoint that provides immediate access to billions of RDF triples. An example SPARQL query can be seen on figure 1.3. We demonstrate more complex SPARQL queries and their SQL equivalents on a subdomain of Wikidata in chapter 7.

1.2 Relational databases

Relational databases are databases storing data in the relational model [16]. Data in a relational database is organized into tuples, grouped into relations:

- **Tuple** is an ordered set of attribute values. Each attribute is defined by a name and a type. In database terms, tuples are called *rows*, attributes are called *columns*.
- **Relation** is a set of tuples. In database terms, relations are called *tables*.

[1] `http://query.wikidata.org`
[2] `https://dbpedia.org/sparql`
[3] `http://bio2rdf.org/sparql`

```
PREFIX example: <http://example.org/#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rel: <http://www.perceive.net/schemas/relationship/>

SELECT ?person ?personName
WHERE {
    ?person rel:enemyOf example:spiderman .
    ?person foaf:name ?personName .
}

# Result
?person               ?personName
example:green-goblin  "Green Goblin"
```

Figure 1.3: SPARQL query example on RDF graph on figure 1.1.

Relations can be connected using *joins*, operations that join tuples of two relations based on a condition on one or more attributes of each relation. An example of a relational database with two relations can be seen on figure 1.1. Systems that maintain a relational database are called relational database management systems (RDBMS).

Person

ID	Name	Name_ru
1	Spiderman	Человек-паук
2	Green Goblin	NULL

Person_Enemy_Person

From_ID	To_ID
1	2
2	1

Table 1.1: Simple example of a relational database with two tables representing the same information as the graph on figure 1.1.

Entity–relationship model The Entity–relationship (ER) model describes schema of a relational database in terms of *entities* and *relationships* [17]. Entities are types of instances with specific properties, each entity type is stored in one table, rows representing one of its instances. Relationships are associations between instances, each relationship type is stored in one table, rows representing one relationship between two instances.

Entity-Attribute-Value model The Entity-Attribute-Value (EAV) model is a data model used to avoid sparse tables created by the ER model when instances of an entity define values for a different subset of attributes. Instead of storing attributes as columns in the entity table, an additional table is created, storing attribute values as rows. Each row in an EAV table stores four fields: a numeric identifier of the instance, an attribute name, an attribute type and finally a value, most commonly cast to string. The disadvantage of the EAV model is the increased number of joins needed to select entity attributes and the need for casting attribute values to the correct type.

1.2.1 SQL

SQL is a query language for relational databases, introducing capabilities for selecting table rows, joining tables, aggregation, data manipulation and other functionality. Each RDBMS implements an SQL execution engine that creates, optimizes and executes a query plan, producing the desired result in an efficient fashion. An example SQL query can be seen on figure 1.4. We demonstrate more complex SQL queries and their SPARQL equivalents on a subdomain of Wikidata in chapter 7.

```
SELECT Name FROM Person INNER JOIN Person_Person ON
Person.ID = Person_Enemy_Person.From_ID
AND Person_Enemy_Person.To_ID = 1
```

Figure 1.4: SQL query example producing an equivalent result as 1.3.

1.2.2 Benefits

In this section, we summarize the benefits of relational databases, demonstrating the potential motivation of our future users.

Widespread support Since their introduction in the 1970s, RDBMS have secured a strong and stable position as the most used database management system [18]. Based on the number of mentions of different DBMS engines on the internet, RDBMS take up more than 80% of the DBMS market, compared to RDF stores which cover only 0.3% [18]. Relational databases are used in diverse production environments including web applications [19], data warehousing, OLAP and predictive analytics [20]. Many database administrator tools for RDBMS are available [21], allowing for straightforward development and support. With minimal differences in SQL dialects, users can easily switch between different RDBMS without the need for major updates.

Query speed As we show in section 2.2, many RDF stores use a RDBMS as a database engine for performance reasons. When an application is using such a RDF store, the added layer of SPARQL abstraction enables for richer semantics and graph queries. However, when the application is designed to work directly with the relational database, the knowledge of the structure, datatypes, indexes and volume of data allow for direct optimization with predictable query times.

1.2.3 Disadvantages

In this section, we summarize the disadvantages of relational databases when compared to RDF, presenting some of the challenges of RDF data conversion to the relational model.

Schema limits As opposed to RDF, data in relational databases is *schema-first* in the sense that schema has to be defined in order to store the data. In RDF, schema is defined rather as a layer on top of the data, enabling more flexibility, complex semantics and the ability to easily integrate data from different sources.

Data representation Data in relational databases is stored in tables, each table with a defined set of columns. The number of cells in a table is the product of the number of its columns and its rows. When instances of an entity specify different subsets of attributes, the produced table can be very sparse, requiring an order of magnitude more memory than the RDF representation.

RDBMS-specific limits Each RDBMS has its own specific properties in terms of the maximum allowed number of columns and tables, maximum lengths of identifiers and other limits. When creating a relational schema and loading the data, these limits need to be considered.

1.3 Apache Spark

Apache Spark[1] is a general-purpose cluster computing system. It enables distributed processing of data using a set of basic operations. Spark builds on the fault-tolerant approach of Apache Hadoop MapReduce, outperforming it more than 10x in many types of tasks [22]. Spark can be used through APIs for Java, Scala, Python and R.

1.3.1 Architecture

A diagram of Spark architecture can be seen on figure 1.5. An application running on a Spark cluster is coordinated by a *driver program*, connecting to *executors* on worker nodes of the cluster using a *SparkContext*. SparkContext can connect to several types of cluster managers such as Spark's own cluster manager, Mesos or YARN, which allocate resources across applications.

1.3.2 Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) are read-only collections of objects partitioned across nodes of a Spark cluster. Operations can be performed on RDDs, resulting in local computation by each executor and sometimes in communication between nodes. There are two types of operations on Spark datasets – *actions* and *transformations*.

[1] `http://spark.apache.org/`

[1] `http://spark.apache.org/docs/latest/cluster-overview.html`

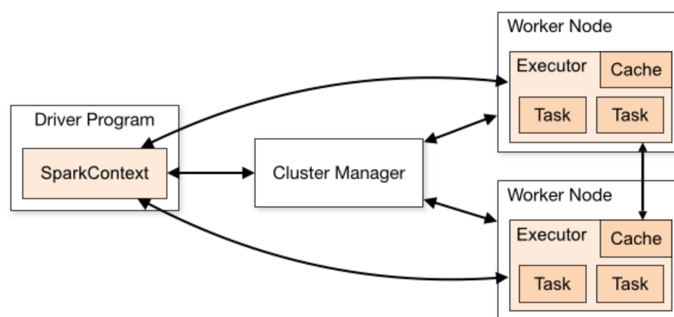


Figure 1.5: Spark cluster architecture[1].

Actions Actions return a value to the driver program after running a computation on the dataset. Actions include:

- `count()` which returns the number of elements.
- `collect()` which returns all elements as an array at the driver program
- `reduce(f)` which aggregates the elements into one value using a function `f`. The function performs a commutative and associative operation, reducing two values into one. This operation is repeated on pairs of elements until a single value is remaining.

Transformations Transformations create a new dataset from an existing one. They are not computed right away. Instead, they are chained and finally computed all at once when an action is performed to return a result to the driver program. Transformations include:

- `map(f)` which returns a new dataset by passing each element through a function `f`.
- `filter(f)` which returns a new dataset with only those elements for which a function `f` returns `true`.
- `reduceByKey(f)` which can be performed on `(key, value)` elements, performing a `reduce(f)` operation separately for each key.
- `join(dataset)` which can be performed on `(key, value)` elements of two datasets, joining elements with equal keys.

Input and output RDDs can be read from multiple storage systems including relational databases, ElasticSearch or HDFS (Hadoop distributed filesystem). HDFS is a distributed file system designed to reliably store very large files across machines in a large cluster [23]. It enables fast distribution of workload by storing each file as a sequence of blocks, which are replicated for fault tolerance on multiple nodes of the cluster.

Chapter **2**

Related work

In this chapter, we summarize related work, focusing on three categories – converting relational data to RDF, using RDBMS for RDF storage and converting RDF to the relational model.

2.1 Converting relational data to RDF

Due to their popularity, relational databases are one of the biggest potential sources to enrich the Web of data. Moreover, mapping different relational databases to RDF using a common ontology enables data integration. Therefore, there has been a substantial amount of effort put into the research and development of tools and mapping languages for the conversion of relational data to RDF. This initiative is therefore parallel to our efforts, in the opposite direction. We can safely say that converting data in this direction is well researched and technologically mature. A recent comprehensive survey of existing tools was published by Michel et al. [24], providing their classification based on mapping description and mapping implementation.

2.1.1 Mapping description

The first conversion step is defining mappings from the relational representation to an RDF representation. This includes generating IRIs from IDs, detecting relations from foreign key constraints, adding `rdf:type` and more complex mappings to existing OWL ontologies. The various approaches were studied by the W3C RDB2RDF Working Group [25]. Their work resulted in two recommendations published in 2012, "A Direct Mapping of Relational Data to RDF" [26] and "R2RML: RDB to RDF Mapping Language" [25].

People

PK		*FK Address(ID)*
ID	**fname**	**addr**
7	Bob	18
8	Sue	NULL

Addresses

PK		
ID	**city**	**state**
18	Cambridge	MA

Table 2.1: Simple example of input relational data for mapping to RDF [26]. PK and FK are primary and foreign key constraints.

2.1.1.1 Direct Mapping

The Direct Mapping [26] defines a simple transformation, automatically mapping relational concepts to a newly generated OWL ontology. Tables become classes, table rows become sets of triples with a common subject. All resulting IRIs begin with a common base prefix. Direct mapping can be used when no suitable ontology exists, or when the goal is to rapidly make a data source available. It is also frequently used as a starting point to more complex domain-specific mappings [24]. A direct mapping of table 2.1 produces the following RDF output [26]:

```
@base <http://foo.example/DB/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<People/ID=7> rdf:type <People> .
<People/ID=7> <People#ID> 7 .
<People/ID=7> <People#fname> "Bob" .
<People/ID=7> <People#addr> 18 .
<People/ID=7> <People#ref-addr> <Addresses/ID=18> .
<People/ID=8> rdf:type <People> .
<People/ID=8> <People#ID> 8 .
<People/ID=8> <People#fname> "Sue" .

<Addresses/ID=18> rdf:type <Addresses> .
<Addresses/ID=18> <Addresses#ID> 18 .
<Addresses/ID=18> <Addresses#city> "Cambridge" .
<Addresses/ID=18> <Addresses#state> "MA" .
```

Figure 2.1: Direct mapping output of table 2.1. The mapping generates IRIs for resources based on their primary keys and detects the 1:n relationship based on the foreign key constraint.

2.1.1.2 R2RML

R2RML [25] is a generic language to describe a set of mappings that translate data from a relational database into RDF. It supports direct mapping as a default setting and provides a vocabulary for specifying custom transformations into existing ontologies. An R2RML mapping defines how to convert a row in a logical table to a number of RDF triples. A set of rules expressed in

RDF statements enables generation of type statements, properties and relationships. The source of a mapping can be defined by a table, a view or an SQL query, enabling the use of aggregation and SQL functions. An example of an R2RML mapping of table 2.1 and the resulting triples can be seen on figure 2.2.

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ex: <http://example.org/ns#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<#AddressTableView> rr:sqlQuery """
    SELECT Addresses.ID, city, state,
           (SELECT COUNT(*) FROM People WHERE People.addr=Addresses.ID)
           AS inhabitants
    FROM Addresses; """. # count number of city inhabitants

<#TriplesMap1>
    rr:logicalTable <#AddressTableView>;
    rr:subjectMap [ # create an IRI from the ID, define type (1)
        rr:template "http://data.example.org/address/{ID}";
        rr:class ex:Address; ];
    rr:predicateObjectMap [ # map the city name column to a statement (2)
        rr:predicate ex:city;
        rr:objectMap [ rr:column "city" ]; ];
    rr:predicateObjectMap [
        rr:predicate ex:inhabitants; # map the number of inhabitants to a statement (3)
        rr:objectMap [ rr:column "inhabitants" ]; ].

# Result of mapping

<http://data.example.org/address/18> rdf:type ex:Address . # (1)
<http://data.example.org/address/18> ex:city "Cambridge" . # (2)
<http://data.example.org/address/18> ex:inhabitants 1 . # (3)
```

Figure 2.2: R2RML mapping of table 2.1 and its output. An aggregated result is used to compute the number of inhabitants on each address.

Currently, R2RML is a widely adopted standard implemented in several production tools such as Morph-RDB[1], DB2Triple[2], Ontop[3], RDOTE[4], Ultrawrap[5], Virtuoso[6] and Oracle 12c[7].

2.1.2 Mapping implementation

Mapping implementation describes whether the database tuples are translated virtually (on demand) or by materialization (producing a whole RDF dataset).

[1]https://github.com/oeg-upm/morph-rdb
[2]https://github.com/antidot/db2triples
[3]http://ontop.inf.unibz.it/
[4]https://sourceforge.net/projects/rdote/
[5]https://capsenta.com/ultrawrap/
[6]https://virtuoso.openlinksw.com
[7]https://www.oracle.com/database/

Virtual converters provide a SPARQL endpoint or follow the Linked Data paradigm to publish resource information via HTTP. Each request to these endpoints is mapped to a corresponding SQL query, returning a converted result. Similar to the RDBMS-based RDF stores described in section 2.2, virtual converters use a RDBMS as a backend, converting SPARQL queries to SQL queries. The difference between these approaches is that virtual converters use an external relational database that can also be used by other applications in other contexts, RDF stores merely use an internal RDBMS to store the data in their own representation. The advantage of virtual converters is that they enable real-time data integration without the need for a continuously updated centralized warehouse that would replicate all the relational data. From the tools mentioned earlier, virtual conversion is supported by Morph-RDB, Ontop, Ultrawrap, Virtuoso and Oracle 12c.

Data materialisation is a static conversion of the whole relational database into a RDF dataset. This approach may be used to provide periodic RDF dumps that are published directly or loaded into any triple store that publishes the data via a SPARQL endpoint. This is often referred to as the Extract-Transform-Load (ETL) approach [24]. Materialization can be performed using built-in functionality or using a SPARQL endpoint with a query that selects all triples.

2.2 Using RDBMS for RDF storage

There are many approaches to storing RDF datasets, each having its benefits and disadvantages in the complexity of implementation, flexibility, storage efficiency and performance. This variety in RDF Stores resulted in multiple surveys comparing their architecture and their features [27] [28] [29].

Many RDF stores actually use RDBMS to store and query data, their contribution is therefore related to our efforts. However, the underlying motivation is very different, since they employ a database schema that is optimized exclusively for performance, not for readability. The different approaches are compared in various surveys [30] [31] [32] focusing mainly on the employed database schema.

2.2.1 Vertical table schema

The earliest implementations such as Sesame [33] employ the *vertical* (*triple*) table schema, storing all triples in a single table with the corresponding three columns for subject, predicate and object [27]. An example can be found on figure 2.3. This implementation requires many self-joins when querying – one per SPARQL triple pattern, resulting in high query times even for simple queries. To reduce the impact, extensive indexing can be used, creating an index for multiple permutations of subject, predicate and object columns [32], each addressing a different type of SPARQL query.

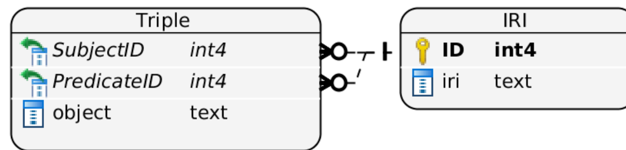


Figure 2.3: Vertical table schema example. IRIs can be stored directly or replaced by a numeric identifier joined on an IRI table.

2.2.2 Horizontal table schema

To save space and increase query performance, other architectures make use of regularities in data by grouping related information together [30]. The first such category is the *horizontal* (*binary*) table store, creating a table for each property. An example can be found on figure 2.4. Since the value of a single property is often of the same datatype, column-based compression can significantly reduce storage requirements.

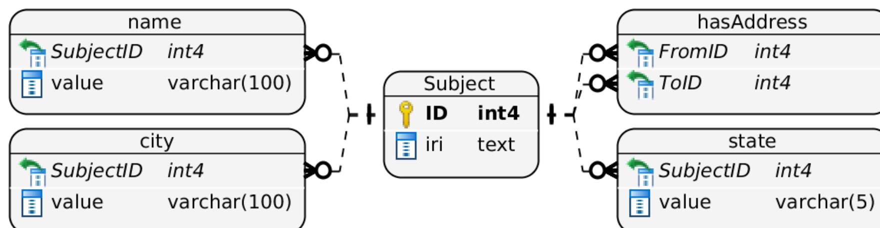


Figure 2.4: Horizontal table schema example. Triples not conforming to the schema are stored in a fallback vertical triple table.

2.2.3 Property table schema

The second category making use of regularities in data is the *property* (*n-ary*) table store which represents resources and their multiple properties as single rows, just as the traditional relational model. Approaches using this schema are therefore most related to our work. An example can be found on figure 2.5. Since each resource can have different properties, storing all resources in a single table would result in many *null* values, resources are therefore divided into different tables based on similarity. The challenge of implementing a property table store is considerably higher that of the other approaches since changes in data (and therefore in resource properties) have to be propagated to the schema.

This approach was demonstrated in Jena [34], creating a table for each `rdf:type`, making use of the fact that resources of the same type often use the same set of properties. Another example is 3XL [35], a PostgreSQL-based store using a relational model based on OWL ontology.

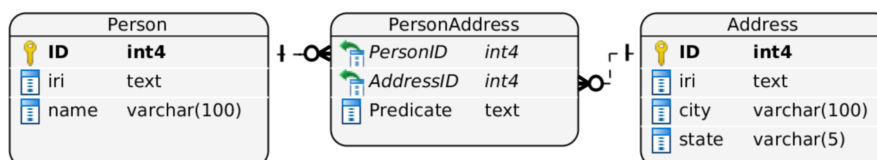


Figure 2.5: Property table schema example. Triples not conforming to the schema are stored in a fallback vertical triple table.

A recent work on relational RDF stores was introduced by Pham et al. [36] who define *emergent schemas*, a method of grouping RDF resources to relational tables based directly on their set of properties – *characteristic sets*. The goal of this approach is to reduce table sparsity (number of *null* values) compared to grouping resources by type. The schema generation process is automatic and configurable by several parameters including maximum number of tables and minimum number of rows of a table. Statements that cannot be expressed in the resulting schema are stored in a triple table. Based on analyzing several public Linked Data datasets (including noisy data such as WebData Commons and DBpedia), the authors conclude that the schema explains more than 95% of triples. Their method was implemented into MonetDB[1] column-store database and into the existing RDBMS backend of Virtuoso, resulting in 3-10x increase in query performance [37].

2.3 Converting RDF to the relational model

As opposed to converting relational data to RDF, there is not much research published on conversion in the opposite direction. While there is a large body of research on converting OWL to the relational schema, it does not cover the conversion of the data itself, which is often implemented exclusively for a given ontology or dataset. In this section we review both the methodologies for ontology to relational schema conversion and the specific and generic tools for converting RDF datasets to the relational model.

2.3.1 Ontology conversion

Since the publication of OWL [13] in 2004 and OWL 2 [38] in 2009 many public ontologies were created for different application domains. For example, Friend of a Friend (FOAF)[2] is an ontology that describes people and social networks, the Semanticscience Integrated Ontology (SIO) [3] is designed for biomedical knowledge integration and is used in projects such as Bio2RDF

[1] `https://www.monetdb.org/`
[2] `http://xmlns.com/foaf/spec/`
[3] `http://sio.semanticscience.org`

[39] and PubChemRDF [3], whereas DBpedia [1] and YAGO [2] ontologies are designed to represent general knowledge.

When these ontologies are used in the production environment, the described data can be stored in different types of databases, especially relational databases, making use of their robustness, performance, maturity, availability and reliability [40]. Therefore, many methods were designed to translate the ontologies to their relational schema counterparts, using their common semantics where possible and representing the remaining concepts using metadata and application logic [41]. A study comparing the benefits and disadvantages of ontologies and relational schemas was published by Martinez-Cruz et al. [42], providing references to various mapping approaches. A recent survey of some of the approaches was published by Humaira et al. [40].

One of the first approaches was proposed by El-Ghalayini et al. [43] who describe converting DAML+OIL (predecessor of OWL) ontologies to the conceptual model. Classes are mapped to tables (incorporating subclass information as entity generalization/specialization), datatype properties are mapped to columns and object properties are mapped to relations. Another early approach is proposed by Gali et al. [44] who describe an automated mapping of OWL ontologies to relational tables based on methods to store XML documents.

Based on the work of [44], Vysniauskas et al. propose a mapping that preserves OWL concepts not natively supported by RDMBS by using meta tables [45]. The authors further extend their research and introduce a mapping algorithm that preserves full OWL2 semantics [41], demonstrating usage on SQL equivalents of sample SPARQL queries. An implementation of their algorithm is published [3] in the form of a plugin for Protégé[4], a popular ontology modeling tool. The plugin enables exporting an OWL ontology to DDL queries that create tables and constraints as well as DML queries that insert existing instances present in the ontology.

A different approach is taken by Teswanich et al. [46] who present a straightforward overview of the requirements for a RDF to RDB mapping system. Although no conversion algorithm is introduced, the presented mapping served as reference for multiple future implementations [47][48][49][50], including ours. The requirements can be summarized as follows:

- Instances of `rdfs:Class` are mapped to tables.
- All resources are mapped to rows in each of their `rdf:type` class table and its superclass tables, identified by a unique numeric primary key in each table.

[1] `http://mappings.dbpedia.org/server/ontology/classes/`
[2] `http://www.yago-knowledge.org/`
[3] OWL2ToRDB, `https://protegewiki.stanford.edu/wiki/OWL2ToRDB`
[4] `http://protege.stanford.edu/`

- Subclass tables specified with `rdfs:subClassOf` property contain its superclass ID column linked to the superclass table with a foreign key.
- Instances of `rdf:Property` whose `rdfs:range` is a `rdfs:Class` are mapped to relations between the `rdfs:domain` class and the `rdfs:range` class.
- Instances of `rdf:Property` whose `rdfs:range` is a `rdfs:Literal` are mapped to columns in their `rdfs:domain` class table.

2.3.2 RDF data conversion

This section introduces existing approaches to automatic RDF data conversion to the relational model, referencing scientific literature and publicly available tools.

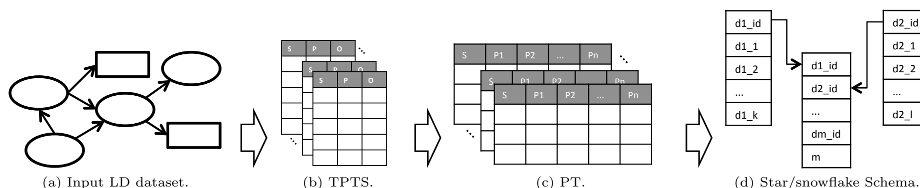
Since SPARQL produces naturally tabular results, a naive conversion approach is to design a number of SPARQL queries, each producing contents of an entity table, or to design a single query that focuses on one `rdf:type` and aggregates all related information into a single table. One of the latter approaches is described by Mynarz et al. [51] who create a single aggregated CSV table for the purpose of knowledge discovery and predictive modeling. Another approach is presented by Allocca et al. [50] who introduce a tool for converting RDF datasets back to their CSV source using the original RML[1] mapping that was used to generate the set of RDF triples.

A virtual conversion approach was explored by Ramanujam et al. [47] who introduce R2D, a tool that enables SQL access to a triple store. This is achieved using a custom JDBC driver that converts SQL queries into their SPARQL equivalents. A relational schema is extracted automatically from RDF data, mapping RDF concepts to their relational counterparts – cardinality of relationships is detected to produce either one-to-many or many-to-many relations, blank nodes are included by mapping their properties to columns of resources that reference them. In their next work [52] the authors add support for RDF reification, a concept of providing additional information about RDF statements. Their goal is to reuse existing relational tools on RDF data, which is demonstrated on a data visualization platform. Unfortunately, 1) due to virtual conversion the tool does not provide any performance improvements over triple stores, 2) the relational schema mapping extraction was reported only on data of 60K triples, taking 250 seconds (with linear extrapolation, processing only 1 million triples per hour), 3) no public software was released.

Though not originally intended as converters, RDF stores with a RDBMS backend described in section 2.2 can be used for RDF conversion. One of these is 3XL [35], a PostgreSQL-based triple store that uses a relational schema based on an OWL ontology. The database is populated in bulk by converting an RDF dataset into a set of CSV files which are loaded using the COPY

[1] An extension of R2RML mapping language, `http://rml.io`

command. The tool uses PostgreSQL inheritance to express `rdfs:subClassOf` relationships and array columns for relations and multivalued properties. The 3XL store is available in a public GitHub repository[1]. In our experiments with the tool, we encountered two issues, 1) a complete OWL ontology describing all classes, property ranges and property domains is required in order to create the corresponding tables and columns, which is not available in many public RDF datasets (including Wikidata and Bio2RDF), 2) table and column names are formatted from IRI suffixes without sanitization, resulting in errors when illegal characters are present.



(a) Input LD dataset. (b) TPTS. (c) PT. (d) Star/snowflake Schema.

Figure 2.6: Star/snowflake schema production based on TPTS approach introduced in [53]. Firstly, the RDF dataset is loaded to type-partitioned triple tables in a relational database, each table storing triples of subjects of one `rdf:type`. Secondly, each type-partitioned table is converted into a property table (described in section 2.2.3) directly in the database by generated SQL queries. Finally, properties are classified as measures or dimensions and the star/snowflake schema is extracted.

In recent years, Semantic Web technologies were exploited for Online Analytical Processing (OLAP), an approach to compactly storing multi-dimensional data for fast visualization and analytics. OLAP systems produce a virtual *OLAP cube* that stores pre-computed values (facts) in multiple dimensions. In most cases, OLAP cubes are stored in relational databases in the form of star or snowflake schemas [20]. A survey of OLAP approaches based on Semantic Web technologies was published by Abelló et al. [54]. A semi-automatic method of generating mappings from RDF to star/snowflake schemas is introduced by Nebot et al. [55]. A fully automated method is introduced by Inoue et al. [56], extracting all properties and relationships from a RDF dataset. The star/snowflake schema is produced and populated automatically by a three-step process called Type-Partitioned Triple Store (TPTS) described in Figure 2.6. In their next work the authors propose SPOOL [53], a framework that produces OLAP cubes using SPARQL without the need to download the whole RDF dataset. The framework is demonstrated on two public SPARQL endpoints – CIA World FactBook and DBPedia. Unfortunately, neither of the tools was released publicly.

An approach to converting RDF-archived relational databases back to their relational form was presented by Stefanova et al. [49]. The input for their

[1] `https://github.com/xiufengliu/3XL`

conversion method is a data archive file and a schema archive file, both in RDF format, produced from a relational database by a proprietary archiving software. Two approaches are proposed, the naive Insert Attribute Value (IAV) approach and the Triple Bulk Load (TBL) approach. The IAV approach produces SQL INSERT and UPDATE statements for each triple. The TBL approach first sorts the dataset in subject-predicate-object order, making it possible to collect all information for one resource at a time in one pass over the data. The resources are saved in CSV files and loaded into the database with a bulk load command. The authors measure the performance of the two approaches on datasets containing up to 200 million triples, concluding that the TBL approach can achieve 18-25 times better performance than IAV, being able to reconstruct 36 million triples per hour on a single machine.

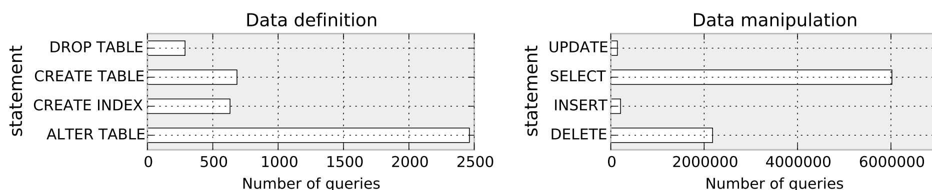


Figure 2.7: Number of SQL queries produced by the RDF2RDB conversion tool in our experiment with the first 100 thousand statements from the Bio2RDF ClinicalTrials.gov dataset (roughly 0.1%). More than 850 thousand queries were produced in total, averaging at more than 85 queries per RDF statement. The conversion process took 170 minutes, converting 300 times less statements per minute compared to our tool running on the same machine.

Most related to our approach is RDF2RDB[57]. To the best of our knowledge, it is the only publicly available tool that is able to convert RDF datasets into entity and relation tables and persist them in a database of choice. RDF2RDB conversion is fully automated, extracting the relational schema from RDFS statements as well as from the relationships present in the data. The tool also handles name collisions (adding numeric suffixes), multi-valued properties (storing them in separate tables) and properties with multiple datatypes (adding a datatype suffix). The differences between our approaches are 1) RDF2RDB is able to apply entailment information including `rdfs:subPropertyOf` or `owl:equivalentClass` statements, our tool currently only applies `rdfs:subClassOf` statements 2) RDF2RDB converts one RDF triple at a time, executing multiple SQL queries and persisting the result on the fly, while our method first collects the whole relational schema and then populates the database in batches of rows 3) thanks to the incremental approach, RDF2RDB is able to add additional RDF triples into an existing database produced by the tool, while our approach is only able to convert the whole dataset at once 4) due to the incremental approach, RDF2RDB produces many SQL queries for a single RDF statement as shown

on figure 2.7, whereas our approach produces a single INSERT query for multiple RDF statements describing a single resource, resulting in an order of magnitude higher performance.

2.3.3 Comparison

To summarize, we provide a classification of RDF-to-RDB conversion approaches described above. Only those approaches that provided enough information to be classified are included. The result can be seen in table 2.2. Approaches are classified using the following dimensions:

- **RDF Source** – How the RDF statements are obtained. Either RDF file or SPARQL endpoint.
- **Processing** – Where the RDF data conversion takes place. Approaches include loading the whole dataset to memory, executing SPARQL queries on a triple store, loading the dataset to a relational database and transforming it using SQL queries, or performing one or more passes over a file on disk.
- **Entity definition** – How entity tables are defined. Approaches include using OWL ontologies, `rdf:type` statements or grouping instances with similar sets of properties.
- **Property definition** – How properties and relations of entity tables are defined. Approaches include using OWL ontologies, `rdf:Property` statements or discovering used properties in RDF data.
- **Data insertion** – How the database is populated with data. Approaches include persisting updates to the database one triple at a time, performing batched INSERT queries or CSV bulk load.
- **Increments** – Whether new RDF statements can be applied to a database produced earlier. Some approaches enable inserting new rows to an existing schema, some also support propagating changes to the schema itself.
- **Published software** – Whether any software was released.

	RDF Source	Processing	Entity definition	Property definition	Data insertion	Increments	Published software
OWL2ToRDB[41]	file	in memory	owl:Class	OWL	SQL script file	data only	yes[1]
Teswanich et al.[46]	file	?	rdf:type	RDFS + discovered	-	-	-
Mynarz et al.[51]	SPARQL	in triple store	single table	discovered	CSV file	-	-
Allocca et al.[50]	file	in memory	single table	reverse RML	CSV file	-	yes[2]
R2D [47]	SPARQL	realtime*	rdf:type + similarity	RDFS + discovered	not needed	data and schema	-
3XL [35]	file	one pass	rdf:type	OWL	CSV bulk load	-	yes[3]
Emergent[58]	file	one pass in SPO order**	similarity	discovered	native bulk load	?	yes[4]
SPOOL TPTT[56]	file	in relational database	rdf:type	discovered	?	-	-
SPOOL SPARQL[53]	SPARQL	in triple store	rdf:type	discovered	?	-	-
SAQ[49]	file	one pass in SPO order**	archived schema	archived schema	CSV bulk load	-	-
rdf2rdb.py[57]	file	one pass	rdf:type	discovered	one triple at a time	data and schema	yes[5]
RDF2X	file	multiple passes, distributed	rdf:type	discovered	batch insert, CSV bulk load	-	yes[6]

* SQL converted to SPARQL, result processed in triple store
** Subject-predicate-object order
[1] Protégé plugin OWL2ToRDB `https://protegewiki.stanford.edu/wiki/OWL2ToRDB`
[2] Conversion tool RML2CSV `https://bitbucket.org/carloallocca/rml2csv`
[3] Triple store 3XL `https://github.com/xiufengliu/3XL`
[4] Experimental RDF branch of MonetDB `https://dev.monetdb.org/hg/MonetDB/shortlog/rdf`
[5] Conversion tool RDF2RDB `https://github.com/michaelbrunnbauer/rdf2rdb`
[6] Conversion tool RDF2X `http://davidprihoda.com/rdf2x/`

Table 2.2: Comparison of RDF to RDB conversion tools.

Chapter 3

Design

In this chapter, we introduce functional requirements for our conversion tool, propose and justify its architecture and describe all steps in a proposed conversion process.

3.1 Requirements

Based on research of related work summarized in the previous chapter, we assembled a list of requirements for a functional conversion tool from RDF to the relational model:

- **Common input formats** – Common RDF serialization formats are supported, including N-Quads, N-Triples and Turtle.
- **Memory efficiency** – Memory requirements are defined only by the size of the schema, not the size of the data
- **Database request efficiency** – The output database is not queried more than necessary
- **Entity tables** – Entity tables store resources of the same `rdf:type`
- **Numeric IDs** – A unique numeric identifier is assigned to each resource
- **Efficient properties** – Properties are saved as entity columns or in another efficient fashion
- **Datatype preservation** – Literal values are stored in columns of corresponding data type
- **Multiple values** – All values of multivalued properties are preserved
- **Meaningful names** – Tables and columns are assigned unique human-readable names, each property (predicate) has the same name in all entity tables
- **Relation tables** – Many-to-many relations of resources are stored as rows in relation tables, referencing the resources by their IDs

- **Automation** – The whole conversion process is performed without the need for user intervention
- **Increments** – Applying additional statements to a database produced earlier is possible
- **Semantic extensions** – OWL and RDFS statements defining relationships between schema elements (classes, properties) are mapped to corresponding concepts in the relational model where possible

The RDF2X conversion tool will satisfy all except two of these requirements. The **Increments** requirement will not be satisfied, all statements have to be converted at once. The **Semantic extensions** requirement will only be partially satisfied, only the `rdfs:subClassOf` property will be applied to the schema. However, the architecture in general makes it possible to implement both of these requirements in future versions after overcoming minor engineering challenges.

3.2 Architecture

In this section, we propose and justify the architecture of RDF2X. An overview can be seen on figure 3.1

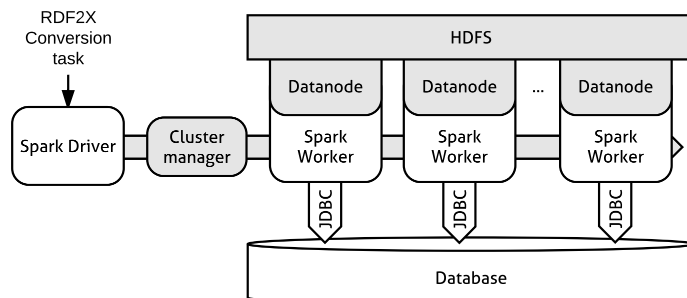


Figure 3.1: RDF2X architecture. A dataset is converted by specifying a conversion task using configuration parameters and submitting the configured RDF2X application to a Spark cluster. Next, tasks are created by the driver program and distributed to the worker nodes using a cluster manager. Finally, data is persisted to the database from all nodes at once through JDBC connections.

We decided to build our tool on the Apache Spark distributed computing platform. Firstly, this will allow us to process the data in parallel on multiple machines using a simple set of operations. Secondly, it will provide us with ready-to-use distributed functions for parsing RDF datasets and persisting the converted data to a database through JDBC. Using Spark, conversion can also be executed locally on one machine, the only disadvantage is the unnecessary overhead for very small datasets.

We decided to obtain RDF statements from RDF dump files instead of querying a SPARQL endpoint. This will allow the user to simply convert a downloaded dataset without having to load it to a triple store. Also, it enables conversion of multiple datasets from different sources into one database. One downside of this approach is that we miss the opportunity to delegate work to a triple store with existing datasets. However, this might often be desirable, since we do not want to put additional workload on production databases.

To load the data to the database, we decided to use two approaches. First, we use the native Spark SQL functionality of writing data using batched INSERT statements. Next, we will implement a CSV bulk load approach that will insert the data using PostgreSQL CopyManager.

3.3 Conversion process

This section provides an overview of the RDF2X conversion process, describing the steps needed to transform RDF triples into entity and relation tables in a database. A diagram of the conversion process can be seen on figure 3.2.

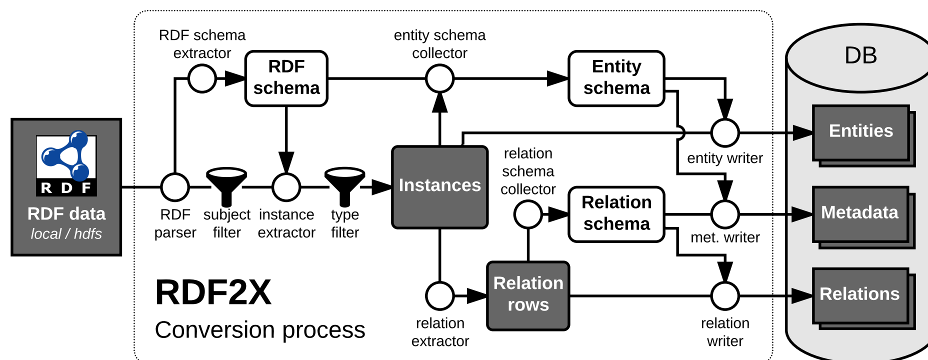


Figure 3.2: RDF2X conversion process. RDF data, cached instances, cached relation rows and the resulting database tables are stored on disk (dark grey boxes). Schema information is stored in memory (white boxes). Circles represent different conversion elements that process data using Apache Spark.

3.3.1 Collecting RDF schema

In the first step, a *RDF schema* storing some necessary information about the dataset needs to be extracted. First, we collect a list of all distinct predicate IRIs (representing different types of relations and properties, e.g. `foaf:name`) and object IRIs used in `rdf:type` statements (representing types of resources, e.g. `foaf:Person`). These IRIs are mapped to memory efficient numeric identifiers which are used in the next phases instead of the string values.

Next, we gather all `rdfs:subClassOf` statements and store them in a graph structure. Finally, we gather all `rdfs:label` statements for given predicates and types.

3.3.2 Extracting instances

In the second step, we reduce the RDF statements by subject IRI, merging all information about each single subject into an *Instance*, a structure storing every assigned `rdf:type`, properties and outgoing relationships of a RDF resource. The input RDF statements can be filtered by a user-specified set of subject IRIs and a depth parameter that enables related instances to be included. Additionally, instances can be filtered by a user-specified set of allowed types. Next, each instance is assigned a globally unique numeric ID. Finally, instances can be persisted to disk or to memory to avoid recomputing them from the RDF dataset for each of the next phases.

3.3.3 Creating entity schema

In the third step, we use the extracted instances to create an *entity schema* that describes the table structure of each `rdf:type`. First, we generate safe and unique string identifiers for all the used types and properties from labels and IRI suffixes. Properties that are used by instances of a type enough to satisfy a user-defined threshold are stored as its columns, the rest (including multivalued properties) is stored in a separate Entity-Attribute-Value table (described in section 1.2). Finally, columns that are already present in a superclass table can be removed.

3.3.4 Extracting relation rows

In the fourth step, rows of relation tables are extracted from the cached instances. A row in a relation table consists of three values - ID of the source instance, ID of the target instance and a predicate representing the relationship type.

3.3.5 Creating relation schema

In the fifth step, we use the extracted relation rows to create a *relation schema* that describes the table structure of each many-to-many relationship. Different schemas of relation tables are implemented, as described in section 4.2.4.

3.3.6 Writing output

Finally, we use the extracted schema and data to create and populate the database. The process consists of three steps – persisting metadata tables

storing information about the schema, persisting entity tables along with the EAV table, persisting relation tables and finally creating keys and indexes.

Chapter **4**

Implementation

In this chapter, we introduce the used tools and libraries, describe the features implemented in our tool and provide conversion results of a simple RDF dataset.

4.1 Tools and libraries

In this section, we describe the libraries we used to implement the RDF2X conversion tool. The tool itself was written in Java 8, making use of its lambda functions, streams and other functionality.

Apache Spark 1.6.2 Apache Spark[1] is described in section 1.3. There are currently two maintained branches of Spark – Spark 1.6.x and Spark 2.x. The 2.x branch introduces performance improvements and usability improvements of the API and other advantages. The 1.6.x branch is maintained for backwards compatibility. We decided for the 1.6.x branch because it was deployed on our existing development cluster. However, if an update to the 2.x version is desirable, it can be implemented with minor updates.

Apache Jena Elephas Apache Jena[2] is a Java framework for building Semantic Web and Linked Data applications. We use a module of Jena called Jena Elephas[3] for distributed parsing of RDF datasets into Spark RDDs.

JGraphT JGraphT[4] is a Java graph library that provides mathematical graph-theory objects and algorithms. We use JGraphT for storing and querying a graph of `rdfs:subClassOf` statements.

[1] `http://spark.apache.org/`
[2] `https://jena.apache.org/`
[3] `https://jena.apache.org/documentation/hadoop/`
[4] `http://jgrapht.org/`

JCommander JCommander[1] is a small Java framework for parsing command line parameters. We use JCommander to parse all configuration parameters for the conversion job.

Mockito & JUnit Mockito and JUnit are Java libraries that enable unit testing. We use both of the libraries to create unit tests for all elements of the conversion process.

Project Lombok Project Lombok[2] is a Java framework that enables boilerplate code generation using annotations. We use Project Lombok to inject a logger and to generate constructors, getters and setters.

4.2 Implemented features

In this section, we provide a summary of all implemented features that influence the conversion process.

4.2.1 Input parsing

RDF triples are read from one or more RDF dump files using Apache Jena Elephas. Files in N-Triples and N-Quads format can be read in parallel, each executor reading one part of the input. This is because each line contains a full statement, resources are referenced by full IRIs. Other formats such as Turtle can only be read on the master node and redistributed to the executors. This is because Turtle is not a line-based format, splitting it would require translating prefixed IRIs into full IRIs and other preprocessing.

4.2.2 Entity tables

An entity table in the relational database stores instances (RDF resources) of the same `rdf:type` (class).

Instance IDs Instances are uniquely identified by their IRI and also by an ID, a globally unique numeric identifier. A primary key constraint is added to the ID column, an index is added to the IRI column to enable fast lookup.

Instance types An instance with multiple types is saved to each type's table, represented with the same ID and IRI in each one. Instances without a type are ignored.

[1] `http://jcommander.org/`
[2] `https://projectlombok.org/`

```
@base <http://example.org/resource/> .
@prefix example: <http://example.org/ns/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<#Czechia>
    a example:Country;
    example:name "Czech republic", "Czechia", "Česká republika"@cs;
    example:neighborOf <#Germany>;
    example:population 10553843.

<#Germany>
    a example:Country;
    example:name "Germany", "Deutschland"@de;
    example:neighborOf <#Czechia>.

<#Vltava>
    a example:River;
    example:name "Vltava", "Vltava"@cs;
    example:basinCountry <#Czechia>.

example:Country rdfs:subClassOf example:Location.
```

Figure 4.1: Example of a RDF input file in Turtle format. Results of conversion are presented below.

Class hierarchy Class hierarchy expressed with `rdfs:subClassOf` statements is considered – each instance is stored in its explicitly specified type's tables as well as in the tables of all superclass types. Foreign key constraints are created on the ID column from each child table to its parent table.

Entity names Entity table names are formatted from the suffix of their type IRI or from their `rdfs:label`. Names are sanitized to keep only allowed characters (numbers, letters and underscores). Numeric suffixes are added to handle non-unique or reserved names.

4.2.3 Literal properties

Literal properties specify literal values such as integers, strings or dates. As opposed to the relational model, RDF properties are first-class citizens defined independently of entity types. When converted, properties can be saved either as columns in one or more entity tables, or as multiple rows in the EAV table, depending on configurable criteria.

Columns Properties that are common enough for instances of a specific type are stored directly in its entity table as columns. Formally, for each property p (of a specific predicate IRI, datatype and language) and entity type e, we compute $nonNullFraction(e, p)$, the percentage of instances of type e that contain a non-null value of the property p. Property p satisfying the $nonNullFraction(e, p) >= minNonNullFraction$ threshold for entity e

is saved as its column, with *null* values where no value is specified. If the threshold is not satisfied, values of property p in instances of type e are saved in the EAV table. This mechanism is used to control table sparsity. An explicit limit on the number of table columns can also be given to deal with RDBMS-specific limits. When persisting an entity table, its columns are sorted either alphabetically or by $nonNullFraction$, most frequent columns first.

river

id	iri	name	name_cs
2	#Vltava	Vltava	Vltava

country

id	iri
1	#Czechia
3	#Germany

location

id	iri	name*	name_cs	population	name_de
1	#Czechia	Czechia	Česká rep...	10553843	null
3	#Germany	Germany	null	null	Deutschland

(a) minNonNullFraction = 0.0

river

id	iri	name	name_cs
2	#Vltava	Vltava	Vltava

country

id	iri
1	#Czechia
3	#Germany

location

id	iri	name*
1	#Czechia	Czechia
3	#Germany	Germany

(b) minNonNullFraction = 1.0

Figure 4.2: Entity tables produced from example 4.1 with two different *minNonNullFraction* thresholds. In case (a) all properties become columns. In case (b) only properties with 100% filled-in values for a given entity become columns. In both cases, columns that are already present in table *location* are removed from the subclass table *country*. Column *name* in the *location* table stores one of multiple values.

Property datatype Values of properties can have multiple datatypes. In RDF2X, values are first converted to native Java objects and then to relational database datatypes. Currently, RDF2X supports the following datatypes: boolean, string, float, double, integer and long.

Multi-valued properties Multi-valued properties specify multiple different values of the same datatype and language for a single instance, creating a one-to-many cardinality relationship. If a property p is multivalued in at least one instance of an entity type e, all values of the property in instances of e are saved to the EAV table. If a multivalued property is also saved in e as a column, the cell will contain a randomly chosen value.

Column names Column names are formatted from the suffix of their property predicate IRI or from their `rdfs:label`. Names are sanitized to keep only allowed characters (numbers, letters and underscores). A language suffix is added if specified for the given property. Datatype suffixes are added if multiple datatypes of the same property are present in the table. Numeric suffixes are added to handle non-unique or reserved names. Properties are named all at once to guarantee that each property will be saved under the same column name in different entity tables.

_attributes

	id	predicate	datatype	language	value
1	#Czechia	ex:name	STRING	*null*	Czechia
2	#Czechia	ex:name	STRING	*null*	Czech republic
3	#Germany	ex:name	STRING	*null*	Germany
4	#Czechia	ex:name	STRING	cs	Česká republika
5	#Germany	ex:name	STRING	de	Deutschland
6	#Czechia	ex:population	INTEGER	*null*	10553843

Figure 4.3: Entity-Attribute-Value table produced from example 4.1 with $minNonNullFraction = 1$. Rows 1–3 store a multivalued property from the *location* table. Rows 4–6 store properties with $nonNullFraction < 1$.

Class hierarchy Class inheritance expressed with `rdfs:subClassOf` statements is considered – columns that are already present in a superclass table are removed from the subclass tables, marking their location in the column metadata table.

Property hierarchy and constraints Property inheritance expressed with `rdfs:subPropertyOf` statements is not considered. In the future, an additional metadata table could be introduced to store this information. Property constraints expressed with `rdfs:range` (representing the possible types of the object – target) and `rdfs:domain` (representing the possible types of the subject – source) are not used. Instead, the occurring types are extracted from the data.

4.2.4 Relation tables

A relation expresses an existing property between two instances. All relations are considered to have many-to-many cardinality. Relations are converted to rows in relation tables, linking the two instances using their numeric IDs. Multiple strategies of creating relation tables were implemented, as seen on Figure 4.4.

Single table strategy Store all relations in a single table. This is possible because all instances have globally unique IDs.

Type strategy Create a relation table for each pair of entity tables that contain related instances. When the related instances have multiple types, a relation table can be created for all pairs of types or only for the root types (types that do not have a specified superclass). Another approach would be to use `rdfs:domain` and `rdfs:range` statements to infer pairs of types for which we should create relation tables – this feature is not yet implemented in our tool, however, it will be considered for a future version.

_relations

id_from	id_to	predicate
#Vltava	#Czechia	ex:basinCountry
#Czechia	#Germany	ex:neighborOf
#Germany	#Czechia	ex:neighborOf

↑ Single table ↑

river_location

river_id_from	location_id_to	predicate
#Vltava	#Czechia	ex:basinCountry

location_location

location_id_from	location_id_to	predicate
#Czechia	#Germany	ex:neighborOf
#Germany	#Czechia	ex:neighborOf

↑ Types (root only) ↑

river_country

river_id	country_id	predicate
#Vltava	#Czechia	ex:basinCountry

river_location

river_id	location_id	predicate
#Vltava	#Czechia	ex:basinCountry

location_country

location_id	country_id	predicate
#Czechia	#Germany	ex:neighborOf
#Germany	#Czechia	ex:neighborOf

country_location

country_id	location_id	predicate
#Czechia	#Germany	ex:neighborOf
#Germany	#Czechia	ex:neighborOf

+ country_country, location_location

↑ Types (all) ↑

neighborof

id_from	id_to
#Czechia	#Germany
#Germany	#Czechia

basincountry

id_from	id_to
#Vltava	#Czechia

↑ Predicates ↑

location_neighborof_location

location_id_from	location_id_to
#Czechia	#Germany
#Germany	#Czechia

river_basincountry_location

river_id_from	location_id_to
#Vltava	#Czechia

↑ Type+Predicate (root only) ↑

Figure 4.4: Relation tables produced from example 4.1 with different strategies. Instance IDs and predicate indexes were replaced with IRI suffixes for readability. The corresponding schema is created automatically based on requested strategy.

Predicate strategy Create a relation table for each relational property (predicate), storing all relations of the same type between any instances.

Type+Predicate strategy Create a relation table for each relational property between two entity tables. This approach most closely resembles the traditional relational model. The pairs of related types can be filtered same as in the **Type** strategy.

4.2.5 Filtering

The tool enables filtering the input RDF dataset to convert only a subset of the data. The dataset can be filtered in two ways.

Filtering by resource IRI The input dataset of triples can be filtered directly by a set of resource IRIs. Additionally, a *depth* parameter can be used to include resources related to the specified resources in *depth* steps. Related resources can be found either exclusively in the direction of the relations (adding resources that are referenced by the existing set) or in both directions (also adding resources that are *referencing* the existing set). This is accomplished by maintaining an in-memory set of IRIs. For each of *depth* steps we perform one pass over the dataset to find resources that are related to our in-memory set.

Filtering by `rdf:type` The dataset can also be filtered by a set of allowed `rdf:types`. Since we do not know which statements will represent the reques-

ted types, all the triples need to be aggregated to instances before filtering can occur.

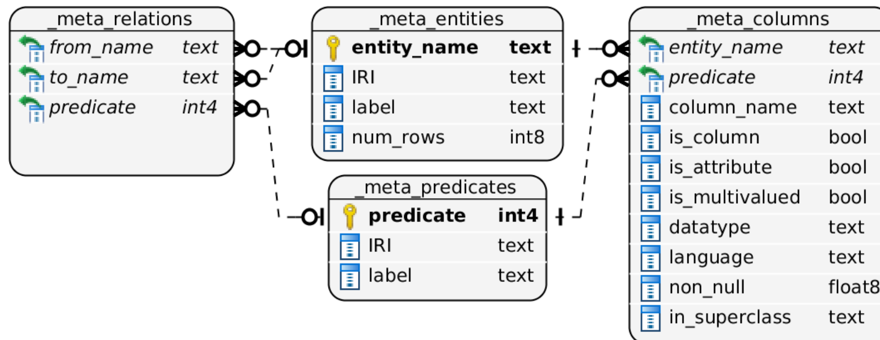


Figure 4.5: Schema of metadata tables.

4.2.6 Metadata

Along with the RDF data, four metadata tables are stored, describing the entities, properties and relations. Schema of the metadata tables can be seen on figure 4.5.

4.2.7 Output formats

Finally, the converted data is written to one of supported outputs:

- **Database via JDBC** Creates the corresponding schema, populates it with data and adds primary keys, foreign keys and indexes. When using PostgreSQL, data can be loaded via CSV COPY bulk load. Other drivers are handled by native Spark JDBC functionality, producing bulk INSERT statements. Finally, primary keys, foreign keys and indexes are added.
- **CSV / JSON** Creates one folder for each entity and relation table. Provided by native Spark functionality and the Spark CSV[1] extension.
- **ElasticSearch**[2] Writes rows of entity and relation tables as documents of corresponding type in an ElasticSearch index. Provided by Elasticsearch Spark [3] extension.

4.2.8 RDF2X Flavors

Each RDF dataset may require custom adjustments to the conversion process to produce desired output. However, making direct modifications to the core

[1]`https://github.com/databricks/spark-csv`
[2]`https://www.elastic.co/products/elasticsearch`
[3]`https://www.elastic.co/products/hadoop`

of the conversion tool would introduce undesired complexity. Therefore, we introduced *RDF2X Flavors*, a system based on a Java interface that is called at several important points in the conversion process. A custom *Flavor* can be created by implementing the interface and performing custom modifications where needed. This system is demonstrated on the *Wikidata Flavor* introduced in chapter 7.

4.3 RDF2X Library

RDF2X also serves as a Java library, providing general processing capabilities for each part of the conversion process – parsing and filtering RDF data, aggregating RDF statements into instances, extracting entity and relation schemas and persisting to several outputs.

Usage of our library can be demonstrated on this instance aggregation example in Java. First, we parse a folder of RDF files, then we select statements about a specified resource and resources related to it in one or two steps, and finally we aggregate the statements into Instance objects and print the result:

```
// parse RDF file into a Spark dataset
QuadParser parser = new ElephasQuadParser(sparkContext);
JavaRDD<Quad> quads = parser
        .parseQuads("/path/to/input/folder");

// collect RDF schema describing the dataset
RdfSchemaCollector rsc = new RdfSchemaCollector(sparkContext);
RdfSchema schema = rsc.collectSchema(quads);

// filter by statements related to a specific subject
QuadFilterConfig filterConfig = new QuadFilterConfig()
        .setRelatedDepth(2)
        .addResources("http://example.com/Prague");
quads = new QuadFilter(filterConfig).filter(quads);

// aggregate the RDF statements into instances
new InstanceAggregator(schema)
        .aggregateInstances(quads)
        .collect()
        .forEach(instance -> {
            System.out.println(instance);
        });
```

The library can also be used to run the whole conversion job programmatically from Java or Scala code, persisting the output to database, disk or a collection of Spark DataFrames:

```
ConvertConfig config = new ConvertConfig()
        .setInputFile("/path/to/input.nq")
        .setOutputConfig(
            OutputConfig.toJSON("/output/folder")
        );
ConvertJob job = new ConvertJob(config, sparkContext);
job.run();
```

Documentation of the RDF2X library can be found on our website[1].

[1] http://davidprihoda.com/rdf2x/

Chapter **5**

Bio2RDF Conversion Statistics

In this chapter, we report on our conversion experiments of RDF datasets provided by the Bio2RDF[1] project. We provide statistics on the conversion performance on a small Spark cluster, on the size of converted datasets compared to the resulting relational representation and on table sparsity based on configurable criteria.

5.1 Introduction

In the past decade there has been a major increase in the number of public life science data sources, such as PubMed, providing citations for biomedical literature, or ClinicalTrials.gov, a database of clinical studies of human participants conducted around the world. These data sources are available in various formats, using different schemas and namespaces. Following the Linked Data initiative, there has been significant effort to integrate these data sources, providing a single interface and interlinking corresponding concepts. One of the successful leaders in these efforts is Bio2RDF, an open-source project that provides a consistent RDF representation of open life science datasets [2].

The main contribution of Bio2RDF is a library of scripts that convert data into RDF in a process called RDFization. Each script is written specifically for one of the supported life science data sources available in various formats such as HTML, relational database or XML. An essential component of the RDFization process is the Bio2RDF API for generating validated IRIs. Each resource in a dataset is assigned a unique IRI on the bio2rdf.org domain and is connected to equivalent resources in other datasets. This enables querying for concepts across multiple data sources. Bio2RDF data can be accessed through the SPARQL endpoint[2] or through downloadable RDF datasets [3].

[1] `http://bio2rdf.org/`
[2] `http://bio2rdf.org/sparql/`
[3] `http://download.bio2rdf.org/`

5.2 Explored datasets

In our experiments we focused on the following datasets with descriptions provided at the Bio2RDF website:

- **PubMed**[1] is a service of the U.S. National Library of Medicine that includes citations from MEDLINE and other life science journals for biomedical articles.
- **ChEMBL**[2] is a database of bioactive compounds, their quantitative properties and bioactivities.
- **CTD**[3] (Comparative Toxicogenomics Database) includes manually curated data describing cross-species chemical-gene/protein interactions and chemical/gene-disease relationships.
- **ClinicalTrials.gov**[4] is a registry and results database of publicly and privately supported clinical studies of human participants conducted around the world.
- **GOA**[5] provides high-quality Gene Ontology annotations to proteins in the UniProt Knowledgebase and International Protein Index.
- **DrugBank**[6] is a bioinformatics and chemoinformatics resource that combines detailed drug data with comprehensive drug target information.

5.3 Data preparation

The datasets were downloaded directly from the Bio2RDF dataset repository[7]. All datasets were provided as one or more gzipped N-Quads files, except ChEMBL, which was provided in Turtle format. As we explained in section 4.2.1, Turtle files cannot be split and have to be read by a single thread. Therefore, in order to be able to test full power of our cluster, we converted the ChEMBL Turtle files into N-Quads format using rdf2rdf[8]. Finally, the datasets were loaded to HDFS.

5.4 Conversion

The datasets were converted using RDF2X on a small Spark cluster of 5 executor nodes, each executor was assigned 8GB of memory and two cores.

[1] `https://www.ncbi.nlm.nih.gov/pubmed/`
[2] `https://www.ebi.ac.uk/chembl/`
[3] `http://ctdbase.org/`
[4] `https://clinicaltrials.gov/`
[5] `http://www.geneontology.org/`
[6] `https://www.drugbank.ca/`
[7] `http://download.bio2rdf.org/`
[8] `https://github.com/knakk/rdf2rdf`

The result was persisted in a PostgreSQL database. For each dataset (except PubMed) we compared two methods of data insertion – *batched inserts*, using Spark JDBC writer, and *PostgreSQL COPY*, using our custom extension exploiting the PostgreSQL CopyManager. Results can be seen on figure 5.1. On average, the COPY method reduces the conversion time by 35%. In the next paragraphs, we will therefore report on conversion using the COPY method.

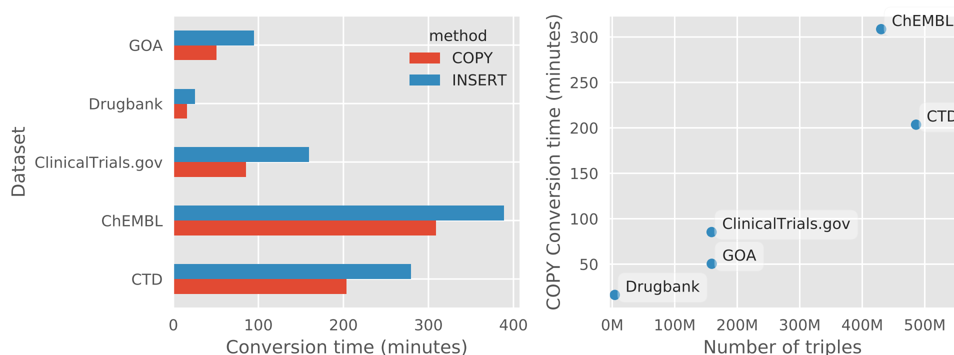


Figure 5.1: RDF2X conversion time of selected Bio2RDF datasets.

The conversion process can be divided into seven phases, each producing one or more Spark jobs. We measured the duration of each phase by inspecting the application logs, the result can be seen on figure 5.2.

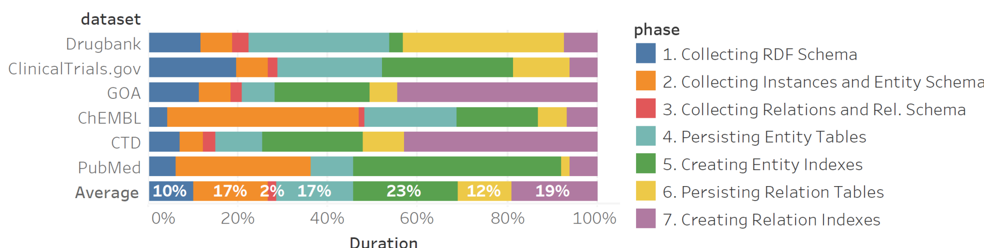


Figure 5.2: Duration of each conversion phase as percentage of total conversion time. Workload of phases 1–3 is handled by the Spark cluster, workload of phase 4 and 6 is handled by both the Spark cluster (filtering Spark DataFrames, preparing statements) and the database (executing statements), workload of phases 5 and 7 is handled exclusively by the database.

More than 8.4 billion triples were converted in total, processing 92 million triples per hour on average. Since not enough total memory was available to process the data directly in memory, the intermediate results had to be cached on disk. This suggests that there is a large potential for improvement if the user has access to more memory and processing power.

5.5 Content statistics

Finally, we compare the original and the converted representations of the RDF datasets. A summary can be seen in table 5.1. On average, size of the resulting database is 68% the size of the uncompressed RDF dataset in N-Quads format.

					Database						
		RDF			*Total*	*Entity tables*			*Relation tables*		
Dataset	**Conversion Time**	**Triples**	**Gzipped**	**Raw N-Quads**	**Size**	**Rows**	**Table size**	**Index size**	**Rows**	**Table size**	**Index size**
Drugbank	16 min	4,215,954	44.9 MB	1.0 GB	1.2 GB	886,602	359.8 MB	98.9 MB	5,513,564	298.8 MB	397.3 MB
ClinicalTrials	1 h 32 min	159,001,344	1.8 GB	46.2 GB	29.9 GB	42,598,182	15.9 GB	4.7 GB	71,646,514	4.1 GB	5.3 GB
GOA	50 min	159,251,919	767.9 MB	31.6 GB	16.1 GB	12,511,020	3.5 GB	1.1 GB	96,052,043	5.0 GB	6.5 GB
ChEMBL	5 h 9 min	430,271,708	2.7 GB	66.2 GB	29.9 GB	52,416,746	9.4 GB	6.2 GB	120,016,262	6.3 GB	8.1 GB
CTD	3 h 37 min	486,080,314	3.3 GB	97.5 GB	83.0 GB	87,983,063	21.9 GB	7.9 GB	421,665,772	23.2 GB	30.0 GB
PubMed	80 h 26 min	7,221,791,590	77.3 GB	1.6 TB	809.2 GB	2,472,787,243	454.9 GB	244.2 GB	763,818,337	48.1 GB	62.1 GB
Total	**91 h 52 min**	**8,460,612,829**	**86.0 GB**	**1.9 TB**	**969.3 GB**	**2,669,182,856**	**506.0 GB**	**264.2 GB**	**1,478,712,492**	**86.9 GB**	**112.3 GB**

Table 5.1: Size of selected Bio2RDF datasets.

Properties that are common enough for a given entity type are saved as columns, the rest is saved in the Entity-Attribute-Value table. This is controlled by the $minNonNullFraction$ configuration parameter, which defines what percentage of cells has to be non-null.

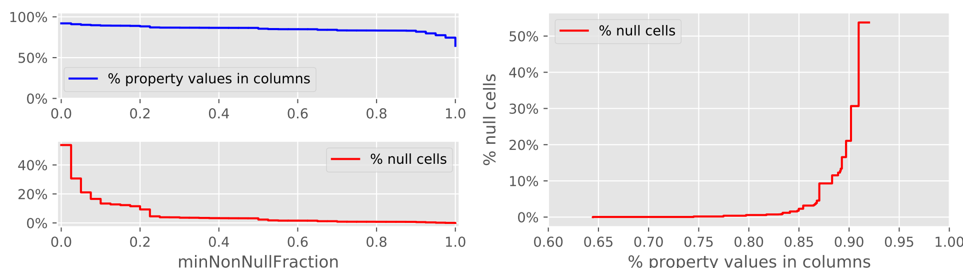


Figure 5.3: Influence of $minNonNullFraction$ on table sparsity. The left chart demonstrates how increasing the threshold slowly reduces the ratio of property values saved in columns while sharply reducing the ratio of null-valued cells. The right chart shows the direct influence between the number of property values in columns and the percentage of null cells.

To measure the influence of this parameter on table sparsity, we converted all datasets with $minNonNullFraction$ set to zero. This saves all properties in columns (except multivalued properties, which are saved both in the EAV table and in columns, saving one arbitrary value). The percentage of non-null cells of each column is saved in the metadata table, we can therefore easily determine how would table sparsity change with the parameter set to a higher value. The result can be seen on figure 5.3, averaging results from all datasets. When 0% non-null cells are requested, 92% of all values are saved in columns

(the remaining 8% are multivalued), while producing sparse tables with 54% null-valued cells. When 100% non-null cells are requested, 64% of values are saved in columns, while all cells are non-null.

5.6 Conclusion

In this chapter, we demonstrated the RDF2X conversion tool on public datasets provided by the Bio2RDF project. More than 8.4 billion triples were converted. The conversion took approximately 92 hours on a Spark cluster of 5 executors with 8GB memory and two cores, converting 92 million triples per hour on average. Finally, we demonstrated the influence of the $minNonNullFraction$ threshold on table sparsity.

Chapter 6

Case study I: Visualizing clinical trials

In this chapter, we provide a high-level demonstration of our tool by converting the ClinicalTrials.gov RDF dataset provided by the Bio2RDF project and visualizing the converted database using Tableau visualization platform. We guide our potential user through the whole process: downloading the RDF dataset, setting up the RDF2X conversion tool, running the conversion job and finally visualizing the schema and the data.

6.1 Introduction

ClinicalTrials.gov[1] is a database of publicly and privately supported clinical studies of human participants conducted around the world. In this case study, we use its RDF representation created by the Bio2RDF[2] initiative from publicly provided XML files. In this case study, we will focus on the following topics, as described by the ClinicalTrials.gov Glossary[3]:

- **Clinical study** A research study using human subjects to evaluate biomedical or health-related outcomes.
- **Eligibility criteria** The key standards that people who want to participate in a clinical study must meet or the characteristics they must have.
- **Condition** The disease, disorder, syndrome, illness, injury or other health-related issue that is being studied.
- **Intervention** A process or action that is the focus of a clinical study. Interventions include drugs, medical devices, procedures, vaccines, and

[1] `https://clinicaltrials.gov`
[2] `http://bio2rdf.org`
[3] `https://clinicaltrials.gov/ct2/about-studies/glossary`

other products that are either investigational or already available.

- **Term** An entity that unifies names of drugs, conditions and other concepts related to a clinical study.
- **Country** Country where a clinical study takes place.
- **Location** Site where a clinical study takes place.

6.2 Data preparation

The ClinicalTrials.gov RDF dataset can be downloaded from the official Bio2RDF repository, we will use the latest version from Release 4[1] of December 2015. The data is provided as a single gzipped N-Quads file, which can be read in parallel by the Jena parser. Therefore, no further preprocessing is needed.

6.3 Conversion

We will use Maven to run RDF2X locally from source. First, the tool can be downloaded from our website[2]. Then, we need to install required dependencies: Java Development Kit 8[3], Maven[4] and Spark 1.6.2[5].

Configuration parameters are passed as program arguments. To get a schema supported by visualization tools, we will use the *Type predicates* relation schema strategy, which will produce a separate relation table for each pair of related entity tables and the type of relationship (predicate). Secondly, we enable instance repartitioning by type, this will bring instances of the same type together in the same partition. When converting on one machine, this produces no communication overhead and makes persisting substantially faster. We start the conversion with the following command:

```
mvn exec:java -Dexec.args="convert \
--input.file /path/to/input/ \
--output.target DB \
--instances.repartitionByType true \
--db.url jdbc:postgresql://localhost:5432/db \
--db.user user \
--db.password password \
--relations.schema TypePredicates"
```

[1] http://download.bio2rdf.org/release/4/

[2] http://davidprihoda.com/rdf2x

[3] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

[4] https://maven.apache.org/

[5] http://spark.apache.org/releases/spark-release-1-6-2.html

The conversion process took 9 hours and 10 minutes on a laptop with a 2GHz 4-core processor and 8GB RAM. The data was loaded to a PostgreSQL database running on the same machine.

6.4 Schema visualization

To get an overview of the converted data, we can first look at the schema of our created database. In total, 63 entity tables and 491 relation tables were created. Therefore, a traditional UML schema would be too complex to present here. Instead, we can visualize the schema using Gephi[1] graph visualization platform. The nodes (entities) and edges (relations) of our graph can be extracted easily from the entity and relation metadata tables. The result can be seen on figure 6.1.

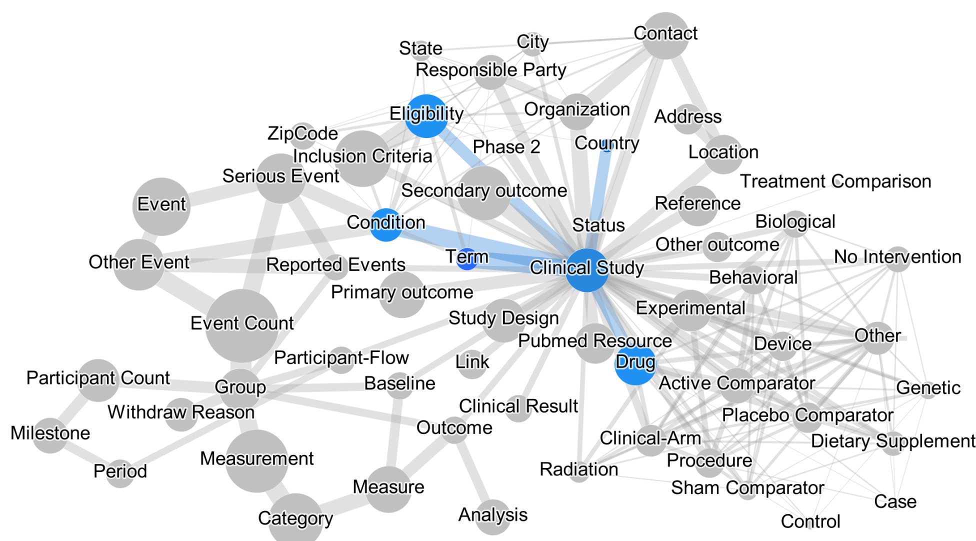


Figure 6.1: Graph visualization of ClinicalTrials.gov entity relationships. Nodes represent entities, edges represent many-to-many relationships, node sizes and edge weights correspond approximately to numbers of rows. Entities and relationships that will be visualized in the next sections are highlighted in blue.

6.5 Data visualization

To visualize the data, we will use Tableau[2], a data visualization platform that will allow us to connect directly to the database and create visualizations simply by dragging and dropping table fields.

[1] https://gephi.org
[2] https://www.tableau.com/

6.5.1 First look

First, we will look at basic information about clinical studies. We create a *Data Source* by providing our database credentials and selecting relevant tables that we will be focusing on. Thanks to the foreign keys, Tableau will join our tables automatically. The result can be seen on figure 6.2.

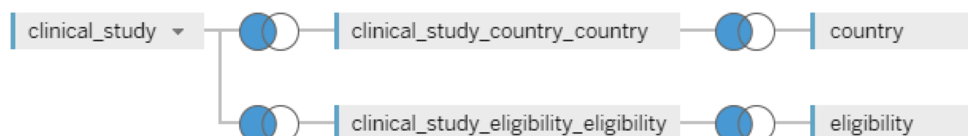


Figure 6.2: Definition of our first Data Source in Tableau with three entities: *Clinical Study*, *Country* and *Eligibility*.

Each study has a *Start Date* field, we can therefore plot the number of studies per year. We will also include the *Is FDA regulated* field which determines whether the study is regulated by the U.S. Food and Drug Administration. The result can be seen on figure 6.3. We can see that most recorded studies started in the past two decades and that roughly one third is FDA regulated.

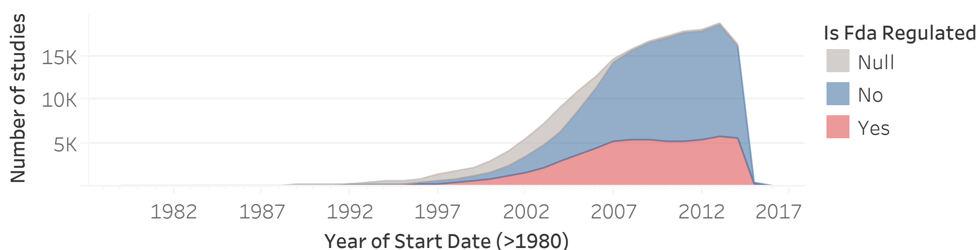


Figure 6.3: Number of clinical studies per start year.

Most clinical studies also define a *Completion Date*, we can therefore look at their duration. The distribution of study durations in months can be seen on figure 6.4. The distinct peaks are located at 12-month intervals, since it is apparently common that a study is scheduled in terms of years.

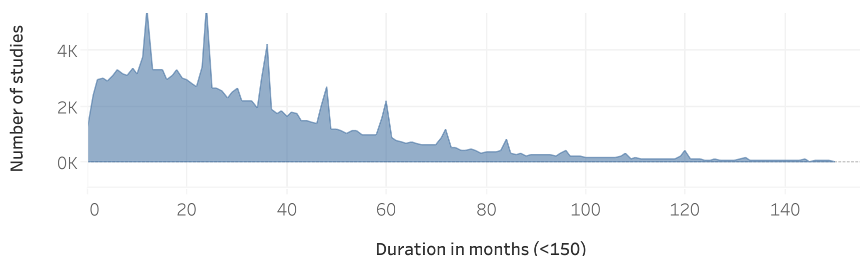


Figure 6.4: Distribution of clinical study duration.

Next, we can look at countries where the clinical studies were conducted. The result is presented on figure 6.5, we can see that almost half of all studies were conducted in the United States.

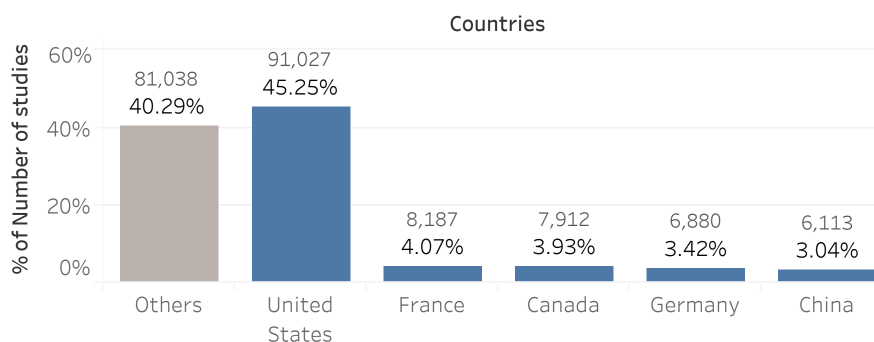


Figure 6.5: Number of clinical studies by country.

6.5.2 Eligibility criteria

Next, we will focus on four eligibility criteria for participation – *Minimum Age*, *Maximum Age*, *Gender* and *Healthy Volunteers*. The last field indicates whether the study allows people who do not have the condition or related conditions or symptoms to participate in that study.

The minimum and maximum age criteria can be visualized together using a *highlight table*, the result can be seen on figure 6.6. The most frequent combination is minimum age between 10 and 20 years and no limit on maximum age. By further inspection using a custom SQL query, we see that 34.2% of all eligibility criteria define a minimum of 18 years and no maximum age.

	Maximum Age (bin)												
Minim..	None	0-10	10-20	20-30	30-40	40-50	50-60	60-70	70-80	80-90	90-100	100+	Total
80-90	0.0%									0.0%	0.0%		0.0%
70-80	0.3%								0.0%	0.0%	0.0%		0.3%
60-70	1.2%							0.0%	0.1%	0.2%	0.1%	0.0%	1.7%
50-60	1.2%						0.0%	0.1%	0.4%	0.4%	0.2%	0.0%	2.3%
40-50	1.1%					0.0%	0.1%	0.4%	0.7%	0.6%	0.1%		3.0%
30-40	0.5%				0.0%	0.1%	0.2%	0.4%	0.6%	0.4%	0.1%		2.1%
20-30	2.3%			0.0%	0.3%	0.8%	0.7%	1.2%	1.1%	0.9%	0.2%	0.0%	7.5%
10-20	36.5%		1.0%	0.6%	1.1%	3.3%	3.8%	8.1%	6.8%	4.6%	1.2%	0.0%	67.2%
0-10	0.7%	0.4%	2.4%	0.3%	0.1%	0.1%	0.1%	0.1%	0.1%	0.1%	0.0%		4.4%
None	8.6%	0.5%	1.3%	0.5%	0.1%	0.1%	0.1%	0.1%	0.2%	0.1%	0.0%		11.6%
Total	52.4%	0.9%	4.6%	1.4%	1.6%	4.4%	4.9%	10.5%	9.9%	7.4%	2.0%	0.0%	100.0%

Figure 6.6: Percentage of studies by minimum and maximum required age. Age limits are grouped to bins of ten years.

Next, we can look at which gender is requested and whether healthy volunteers are accepted. The result can again be visualized using a highlight table, the result can be seen on figure 6.7. In most studies, only volunteers with related conditions or symptoms are accepted. Most studies accept any gender.

	Gender				
Healthy Volunteers	Null	Both	Female	Male	Total
Null	0.3%	0.8%	0.1%	0.0%	1.2%
No		66.4%	6.3%	2.8%	75.5%
Yes		17.7%	3.2%	2.4%	23.3%
Total	0.3%	84.9%	9.6%	5.2%	100.0%

Figure 6.7: Percentage of studies by two eligibility criteria – gender and whether healthy volunteers are accepted

6.5.3 Conditions

Next, we examine conditions, the focus of each clinical study. The conditions of each clinical study are defined in two ways - by specific free-text labels and by one or more unified terms. The free-text labels are stored in the *Condition* entity table, most labels are rather specific and referenced only by one clinical study. The unified terms are stored in the *Term* entity table.

An overview of the most used condition labels and terms can be seen on figure 6.8. We see that there is a significant overlap between the two entities – some condition labels such as HIV Infections or Hypertension have corresponding term identifiers. However, some common condition labels such as Healthy, Obese, Breast Cancer or Prostate Cancer do not have corresponding terms. Therefore, we decided to focus on the label from the *Condition* entity in the next visualizations to determine the most studied conditions.

	Condition									
Condition Term	Healthy	Breast Cancer	HIV Infec tions	Obesity	Hyperte..	Asthma	Pain	Prostate Cancer	Schizop..	Depress..
Null	5,113	3,405		2,102				1,879		
HIV Infections	10		2,983	6	6	3	7		2	18
Hypertension	11	1	6	151	2,329	20	4	1	2	21
Asthma	20	1	3	29	20	2,190	1			10
Pain	6	42	7	8	4	1	1,918	19	2	46
Schizophrenia	6		2	17	2		2		1,757	27
Depression	10	30	18	16	21	10	46	4	27	1,735
Diabetes Mellitus, Type 2	26			41	30	3	2		2	4
Coronary Artery Disease	4	1	2	9	27	8	2		1	9
Leukemia		31	2				13	5		5

Figure 6.8: Number of studies for the combination of top 10 most frequent condition labels and condition terms.

We can examine the development of the most frequent conditions by plotting number of their studies by year. The result can be seen on figure 6.9. For example, we see that the number of clinical studies of HIV Infections apparently dropped substantially in the last decade.

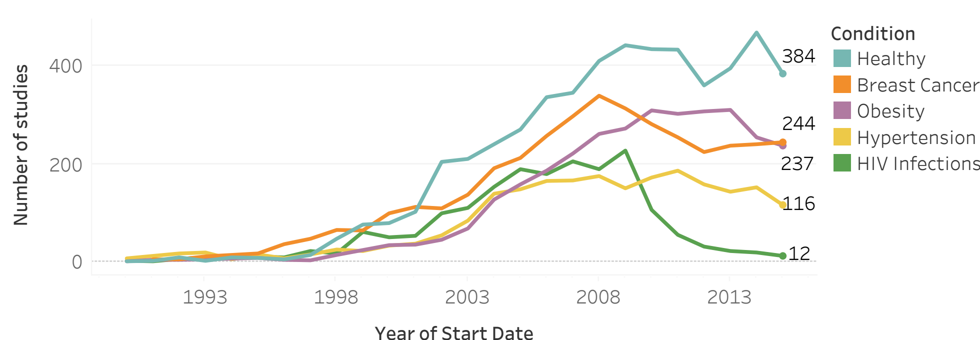


Figure 6.9: Number of studies per year for the most frequent condition labels.

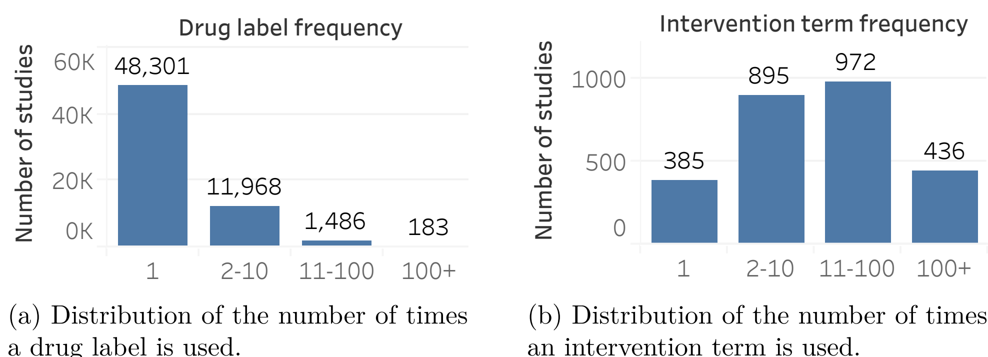
6.5.4 Interventions

Finally, we will focus on clinical study interventions. Just like conditions, interventions are defined in two ways - by specific free-text labels and by one or more unified terms. The free text labels are saved in multiple entities based on intervention type, for example *Drug*, *Procedure* and *Biological*. The unified terms are stored in the *Term* entity table. To see the distribution of the number of times a label or a term is referenced, we defined a custom SQL datasource using the following SQL queries:

```
SELECT drug.title_en, COUNT(*)
FROM clinical_study_intervention_drug
INNER JOIN drug ON drug_id=drug.id
GROUP BY drug.title_en

SELECT term.title_en, COUNT(*)
FROM clinical_study_intervention_browse_term
INNER JOIN term ON term_id=term.id
GROUP BY term.id, term.title_en
```

Plot of the distribution can be seen on figure 6.10. We see that the vast majority of drug labels are only used once. This is due to the fact that drug labels often contain the dosage and other specific information. Only 183 distinct drug labels are referenced by more than 100 studies. On the other hand, most intervention terms are reused, 436 of them by more than 100 studies. Therefore, we decided to focus on the intervention terms. For example, we can plot the number of studies per year for the top five most frequent ones, the result can be seen on figure 6.11. All five of the interventions



(a) Distribution of the number of times a drug label is used.

(b) Distribution of the number of times an intervention term is used.

Figure 6.10: Frequency of drug labels and intervention terms.

are chemotherapy drugs approved for medical use by the FDA. The most recent drug is Bevacizumab, approved in 2004[1], explaining the visible increase in the following years.

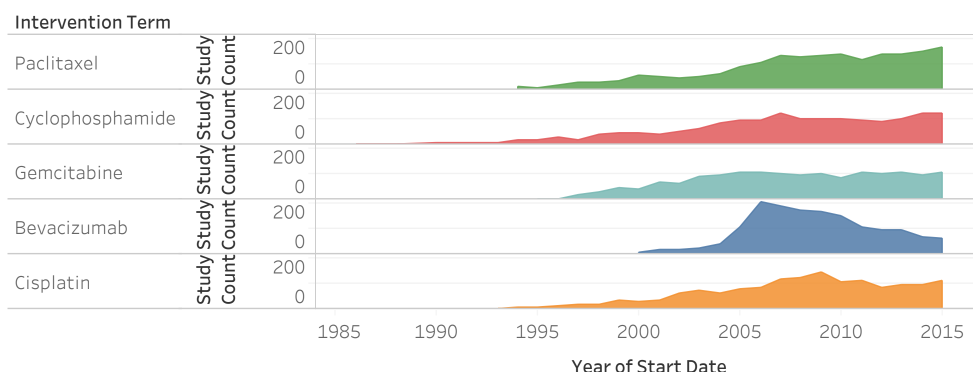


Figure 6.11: Number of clinical studies per year for the five most frequent intervention terms.

Finally, we can look at the most frequent interventions for each one of the most studied conditions by looking at the number of times they are referenced together by the same study. The result can be seen on figure 6.12.

6.6 Conclusion

In this chapter, we provided a high-level demonstration of our tool by converting the ClinicalTrials.gov RDF dataset provided by the Bio2RDF project. We visualized the converted database using Tableau visualization platform, demonstrating one of the possible benefits of automatic conversion of RDF data to the relational model.

[1] `https://www.drugs.com/monograph/bevacizumab.html`

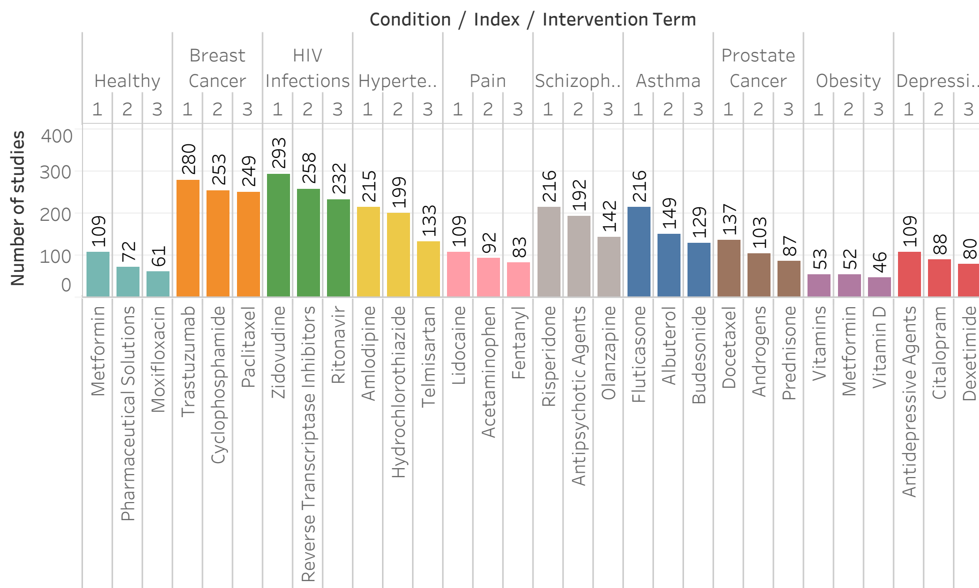


Figure 6.12: Top 3 most frequent interventions for the top most frequently studied conditions

Chapter 7

Case study II: Querying Wikidata with SQL

In this chapter, we provide a low-level demonstration of our tool by converting a music subdomain of Wikidata into the relational model. The resulting database is explored using multiple SQL queries compared with their SPARQL equivalents. We guide our potential user through the whole process: downloading and preprocessing the RDF dataset, configuring the RDF2X conversion tool, running the conversion job on an existing cluster and finally exploring the data using SQL queries.

7.1 Introduction

In attempt to represent general human knowledge, several initiatives have emerged, extracting structured information from public sources such as Wikipedia and providing the result in RDF.

The first representative is DBpedia, introduced in 2007 [6]. It is a community effort to periodically and automatically extract structured information from Wikipedia. The project remains active and growing, currently representing 9.5 billion statements about millions of entities[1]. DBpedia can be accessed through a SPARQL endpoint and through periodic RDF dumps.

Parallel to DBpedia, Freebase was introduced in 2007 [7]. It was based on initial data harvested from Wikipedia, MusicBrainz and other sources and was subsequently managed separately by public crowdsourcing. In 2010 it was acquired by Google. In 2016 it was officially shut down in favor of two of its successors – Google Knowledge Graph and Wikidata [59].

Wikidata was founded directly by Wikimedia Foundation in 2012 [8]. In contrast to DBpedia, which periodically extracts information from Wikipedia, the approach of Wikidata is opposite – creating a structured collaboratively

[1] `http://wiki.dbpedia.org/dbpedia-version-2016-04`

edited knowledge base of information that can be included in Wikipedia pages and other applications. Currently, Wikidata stores billions of statements about more than 26 million entities[1]. Wikidata can be accessed through a SPARQL endpoint and through database dumps in RDF and JSON.

7.2 Data preparation

The current version of the RDF dataset can be downloaded from the Wikimedia Downloads[2] site updated several times per week. Until May 2017, the dataset was provided in gzipped Turtle format, which can only be parsed in one thread. This can be solved by the rdf2rdf[3] converter, first decompressing the Turtle file and then converting it to the N-Triples format:

```
# Decompress the Turtle file
gunzip wikidata-20170417-all-BETA.ttl.gz

# Convert it to the N-Triples format
rdf2rdf -in=wikidata-20170417-all-BETA.ttl -out=wikidata.nt

# Optionally, compress it again to reduce disk I/O of our job
gzip wikidata.nt
```

Luckily, since April 2017, the dump is published in N-Triples format, this preprocessing step is therefore not needed anymore.

7.3 Wikibase data model

The Wikidata RDF dump is based on the Wikibase data model. A resource in Wikibase is either an *Item* or a *Property*. The model does not make an explicit distinction between resources that represent classes and resources that represent instances. A resource can represent both of these concepts at the same time. An example is given on figure 7.1.

Wikidata does not follow the convention of human-readable IRIs. Rather, each resource is assigned a numeric suffix. Therefore, it is not desirable to use the IRIs for name formatting. Instead, we will use the RDF2X feature of generating entity and property names from their RDFS labels.

Property values of Wikidata entities can either be assigned directly or through a *Statement* node that stores the value as well as qualifiers that specify additional information. For example, a statement about a city's population might include a qualifier specifying the point in time. Statement nodes are assigned *ranks* that determine which statement should be prioritized. In this

[1]https://www.wikidata.org/wiki/Wikidata:Statistics
[2]https://dumps.wikimedia.org/wikidatawiki/entities/
[3]https://github.com/knakk/rdf2rdf

case study, we focus only on so-called *truthy* statements. These statements have the best rank for a given property and are assigned directly as seen on figure 7.1. For semantic reasons, different IRI prefixes are used for the same property in different contexts. For example, to refer to the property as an entity, the *entity* prefix `wd` is used. When used as a predicate in a truthy statement, the *direct statement* prefix `wdt` is used. This is an issue for generating property names from labels, since the RDFS label is only assigned to the entity IRI.

```
@prefix wd: <http://www.wikidata.org/entity/> .
@prefix wdt: <http://www.wikidata.org/prop/direct/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

# Assigning labels
wd:Q5 rdfs:label "human"@en .
wd:Q254 rdfs:label "Wolfgang Amadeus Mozart"@en .
wd:Q5994 rdfs:label "piano"@en .
wd:Q52954 rdfs:label "keyboard instrument"@en .
wd:P1303 rdfs:label "instrument"@en .
# Mozart is an instance of Human
wd:Q254 wdt:P31 wd:Q5 .
# Mozart plays the piano
wd:Q254 wdt:P1303 wd:Q5994 .
# The piano is a subclass of keyboard instruments
wd:Q5994 wdt:P279 wd:Q52954 .
```

Figure 7.1: Example of Wikidata RDF statements. An item is assigned a type using the custom *instance of* statement. Wikidata does not define any distinction between classes or instances, each item can be used in both contexts, as demonstrated by the *Piano* entity, which is a subclass while it is also referenced by another instance in a many-to-many relationship.

To handle the presence of multiple IRIs for the same entity and other specific Wikidata concepts, we implemented a custom RDF2X Flavor. Firstly, it replicates RDFS labels to all the different IRI prefixes of a property. Secondly, it selects the following default settings:

- `formatting.useLabels` is enabled.
- `relations.schema` is set to `Predicates` to create one relation table for each property.
- `rdf.subclassPredicate` is set to the *subclass of* property `wdt:P279`.
- `rdf.typePredicate` is set to the *instance of* property `wdt:P31` and also to the *subclass of* property to include subclasses as rows in the table of their superclass. This is not semantically correct, but usually provides meaningful results. For example, it includes the piano entity in the music instruments table.

7.4 Conversion

We will submit the conversion job to our existing cluster managed by Apache YARN. Therefore, our dependencies will be installed automatically, we only need the RDF2X executable JAR file that can be downloaded from our website[1]. We will use the following configuration parameters:

- `flavor` selects the implemented `Wikidata` flavor
- `input.acceptedLanguage` is set to `'en'` to allow only English texts
- `filter.type` is set to IRIs of entities whose instances we want to persist: band, country, human, music genre, musical instrument and song.
- `entities.minColumnNonNullRatio` is set to `0.1` so that only properties specified for at least 10% instances of a given type are stored in columns, the rest will be stored in the Entity-Attribute-Value table
- `rdf.cacheFile` provides a location for caching the collected RDF schema file. This is useful if we want to run the conversion multiple times on different filtered types without having to recollect the schema.

The conversion took approximately 2 hours and 20 minutes on a Spark cluster of five executors with 8GB memory and two cores. When running again with a cached RDF schema, the conversion time was 50 minutes shorter. With each conversion, 2.3 billion triples were processed.

7.5 Produced schema

The resulting database contains 3.8 million entity table rows, vast majority of them stored in the `human` table (3.4 million). It is not yet possible to filter properties during conversion, therefore, all relation tables were created for all 246 occurring relational properties. We removed the unwanted tables manually, reducing the schema to its final form visible on figure 7.2.

7.6 SQL experiments

Now, we can explore the created database. For each topic, we first provide a SPARQL query that can be executed at the Wikidata SPARQL endpoint[2] to produce the desired result. Next, we introduce an equivalent SQL query, demonstrating how Linked Data concepts are naturally translated to the relational model.

[1] `http://davidprihoda.com/rdf2x`
[2] `https://query.wikidata.org/`

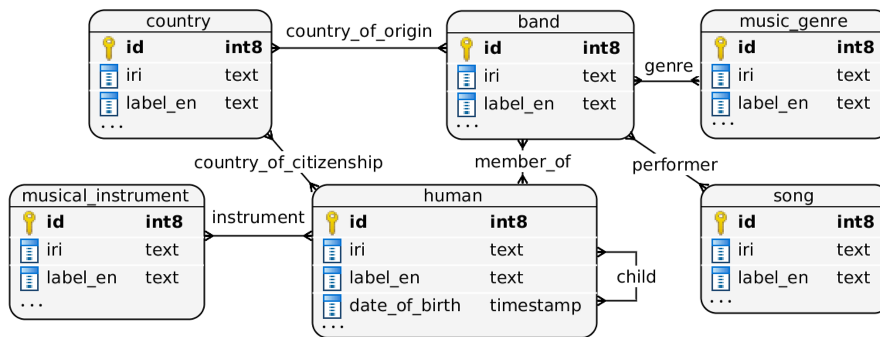


Figure 7.2: Schema of selected entity tables. Relations between tables are illustrative, since `Predicate` relation tables are not restricted to two specific entity types. Rather, each one stores all relations of a given property type.

7.6.1 Songs by The Beatles

First, let's consider a simple example of gathering all songs by The Beatles, sorted by date of creation. This can be achieved with the following SPARQL query:

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT (year(?date) as ?year) ?songLabel
WHERE
{
  ?song wdt:P31 wd:Q7366 .   # is a song
  ?song wdt:P175 wd:Q1299 .  # song is performed by The Beatles
  ?song wdt:P577 ?date .     # song has publication date
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en" }
}
ORDER BY ?date
```

year	title
1953	No Other Love
1959	Till There Was You
1961	Please Mr. Postman
1962	Love Me Do
1963	Ask Me Why

A corresponding SQL query can be designed simply by selecting from the `song` table and joining with the `band` table through the `performer` relation:

```
SELECT
  date_part('year',song.publication_date) as year,
  song.label_en as title
FROM band
  INNER JOIN performer ON band.id = performer.id_to
  INNER JOIN song ON performer.id_from = song.id
  WHERE band.label_en = 'The Beatles'
ORDER BY song.publication_date;
```

In a real-world application, we would reference the Beatles instance by its ID, the join could therefore be performed directly on the `performer` table.

7.6.2 Members of The Beatles

Next, we can demonstrate the aggregation functionality of SPARQL by fetching all instruments played by members of The Beatles. We will group the result by the person's name and use the `GROUP_CONCAT` aggregate to concatenate names of all instruments into a single list:

```
SELECT
  ?beatleLabel
  ?born
  (GROUP_CONCAT(?instrumentLabel; separator=', ') AS ?instruments)
WHERE
{
  ?beatle wdt:P463 wd:Q1299 .       # member of The Beatles
  ?beatle wdt:P569 ?born .          # beatle has date of birth
  ?beatle wdt:P1303 ?instrument .   # beatle plays an instrument
  ?instrument rdfs:label ?instrumentLabel . # request label explicitly
  FILTER((LANG(?instrumentLabel)) = "en") . # to be able to aggregate

  SERVICE wikibase:label { bd:serviceParam wikibase:language "en" }
}
GROUP BY ?beatleLabel ?born
```

beatle	born	instruments
Ringo Starr	1940-07-07	drum, drum kit, percussion instrument
John Lennon	1940-10-09	harmonica, bass guitar, piano, guitar
Pete Best	1941-11-24	drum kit
Sir Paul McCartney	1942-06-18	bass guitar, piano, guitar
George Harrison	1943-02-25	violin, guitar

We can achieve the same result using SQL by selecting from the `human` table, joining with the `band` table through the `member_of` relation and with the `musical_instrument` table through the `instrument` relation. The instruments can be concatenated using the `string_agg` function:

```
SELECT
  human.name_en AS beatle,
  human.date_of_birth AS born,
  string_agg(musical_instrument.name_en, ', ') as instruments
FROM band
  INNER JOIN member_of ON member_of.id_to = band.id
  INNER JOIN human ON member_of.id_from = human.id
  INNER JOIN instrument ON instrument.id_from = human.id
  INNER JOIN musical_instrument
    ON instrument.id_to = musical_instrument.id
WHERE band.name_en = 'The Beatles'
GROUP BY human.name_en, born;
```

7.6.3 Relatives of The Beatles

Next, let's look at the Wikidata concept of property inheritance. This can be demonstrated on fetching all relatives of members of The Beatles. The Wikidata property *relative* (`wd:P1038`) has multiple subproperties such as *mother*, *father* and *child*, which are used for direct family members. To get our result, first, we define all subproperties using the *subproperty of* property chained with the `*` path operator that defines a path of zero or more occurrences. Then, we collect all values for the defined properties:

```
SELECT
  ?beatleLabel
  ?familyPropertyLabel
  (GROUP_CONCAT(DISTINCT ?personLabel; separator=', ') AS ?relatives)
WHERE
{
  # get the direct property for all subproperties of relative
  ?familyProperty wdt:P1647* wd:P1038 .
  ?familyProperty wikibase:directClaim ?familyClaim .
  # member of The Beatles
  ?beatle wdt:P463 wd:Q1299 .
  # beatle is related to person
  ?beatle ?familyClaim ?person .
  # request labels
  ?person rdfs:label ?personLabel .
  FILTER((LANG(?personLabel)) = "en") .
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en" }
}
GROUP BY ?beatleLabel ?familyPropertyLabel
ORDER BY ?beatleLabel
```

beatle	relative	names
George Harrison	child	Dhani Harrison
George Harrison	spouse	Pattie Boyd, Olivia Harrison
John Lennon	child	Sean Lennon, Julian Lennon
John Lennon	father	Alfred Lennon
John Lennon	mother	Julia Lennon
John Lennon	spouse	Yoko Ono, Cynthia Lennon
Paul McCartney	child	Mary McCartney, Stella McCartney, Beatrice McCartney, James McCartney, Heather McCartney
Paul McCartney	father	Jim McCartney
Paul McCartney	mother	Mary McCartney
Paul McCartney	sibling	Mike McGear
Paul McCartney	spouse	Linda McCartney, Heather Mills
Ringo Starr	child	Lee Starkey
Ringo Starr	spouse	Barbara Bach

The concept of property inheritance is not natively supported by relational databases. However, we can achieve the same result by explicitly including all the requested relation tables, merged using the `UNION` operator:

```
SELECT
  human.name_en,
  rel as relative,
  string_agg(person.name_en, ', ') as names
FROM band
  INNER JOIN member_of ON member_of.id_to = band.id
  INNER JOIN human ON member_of.id_from = human.id
  INNER JOIN (
      (SELECT 'relative' as rel, * FROM relative) UNION
      (SELECT 'father' as rel, * FROM father) UNION
      (SELECT 'mother' as rel, * FROM mother) UNION
      (SELECT 'spouse' as rel, * FROM spouse) UNION
      (SELECT 'child' as rel, * FROM child) UNION
      (SELECT 'sibling' as rel, * FROM sibling)
    ) relatives
    ON relatives.id_from = human.id
  LEFT JOIN human person ON relatives.id_to = person.id
WHERE band.name_en = 'The Beatles'
GROUP BY human.name_en, rel
ORDER BY human.name_en, rel;
```

This suggests a possible improvement of our conversion tool. For relation schema created using the *Predicate* or *Type predicate* strategy, instances of a given property could be persisted in all of its parent relation tables, just like instances of a given type are added to all of its superclass entity tables.

7.6.4 Bands from the Czech Republic

Next, let's combine the musical domain with provided geographical information. For example, using the following SPARQL query, we can fetch music genres of the oldest Czech bands that have their Wikidata entry:

```
SELECT
  ?bandLabel
  (year(?inception) AS ?year)
  (GROUP_CONCAT(?genreLabel; separator=', ') AS ?genres)
WHERE
{
  ?band wdt:P31 wd:Q215380 .       # is a band
  ?band wdt:P495 wd:Q213 .         # band is from Czech republic
  ?band wdt:P571 ?inception .      # band has inception year
  ?band wdt:P136 ?genre .          # band has genre
  ?genre rdfs:label ?genreLabel .  # request labels
  FILTER((LANG(?genreLabel)) = "en") .
  SERVICE wikibase:label { bd:serviceParam wikibase:language "en" }
}
GROUP BY ?bandLabel ?inception
ORDER BY ?year
```

name	inception	genres
Hradišťan	1950	contemporary folk music, folk music
Etc...	1974	rock music
Tichá dohoda	1986	alternative rock
Malignant Tumour	1991	heavy metal music
Peneři strýčka Homeboye	1993	rapping, hip hop

The same result can be achieved using SQL by selecting from the `band` table, joining with the `country` table through the `country_of_origin` relation and with the `music_genre` table through the `genre` relation.

```
SELECT
  band.label_en as name,
  date_part('year',band.inception) as inception,
  string_agg(music_genre.label_en, ', ') as genres
FROM country
  INNER JOIN country_of_origin ON country.id = country_of_origin.id_to
  INNER JOIN band ON country_of_origin.id_from = band.id
  INNER JOIN genre ON band.id = genre.id_from
  INNER JOIN music_genre ON music_genre.id = genre.id_to
  WHERE country.label_en = 'Czech Republic'
    AND band.label_en IS NOT NULL
GROUP BY band.label_en, band.inception
ORDER BY band.inception;
```

7.6.5 Number of bands by country

Next, let's focus on countries in general and generate a simple statistic of the number of bands that have their Wikidata entry per country. This can be achieved using the following SPARQL query:

```
SELECT ?countryLabel (COUNT(?band) AS ?bands)
WHERE
{
  ?band wdt:P31 wd:Q215380 .  # is a band
  ?band wdt:P495 ?country .   # band is from a country

  SERVICE wikibase:label { bd:serviceParam wikibase:language "en" }
}
GROUP BY ?countryLabel
ORDER BY DESC(?bands)
```

name	bands
United States of America	14898
Canada	1318
Netherlands	1241
United Kingdom	991
Italy	933

The same result can be achieved using SQL by counting the number of `country_of_origin` relations between the `band` table and the `country` table:

```
SELECT
  country.label_en as name,
  COUNT(*) as bands
FROM country
  INNER JOIN country_of_origin ON country.id = country_of_origin.id_to
  INNER JOIN band ON country_of_origin.id_from = band.id
GROUP BY country.label_en
ORDER BY 2 DESC;
```

7.6.6 Number of bands per capita

Finally, let's extend the previous query and calculate the number of bands per million inhabitants. The population of each country is readily available, the result can be achieved using the following SPARQL query:

```
SELECT
  ?countryLabel
  ?population
  (COUNT(?band) AS ?bands)
  (COUNT(?band) * 1000000 / ?population AS ?perMillion)
WHERE
{
  ?band wdt:P31 wd:Q215380 .        # is a band
  ?band wdt:P495 ?country .         # band is from a country
  ?country wdt:P1082 ?population .  # country has population

  SERVICE wikibase:label { bd:serviceParam wikibase:language "en" }
}
GROUP BY ?countryLabel ?population
ORDER BY DESC(?perMillion)
```

name	population	bands	per million
Estonia	1313271	199	151
Faroe Islands	49291	7	142
Finland	5501043	603	109
Sweden	9954420	805	80
Netherlands	17000000	1241	73

To achieve this result in SQL, first we need to check where the population property is stored. Apparently, less than 10% of countries specify their population, the property value is therefore stored as a row in the Entity-Attribute-Value table. First, we use this SQL query to find the numeric identifier of our property:

```
SELECT predicate FROM _meta_predicates WHERE label = 'population';
```

Finally, we can achieve the desired result by selecting from the `country` table, joining with the `band` table through the `country_of_origin` relation and with the `_attributes` EAV table directly on the country's primary key:

```
SELECT
  country.label_en as name,
  _attributes.value::int as population,
  COUNT(*) as bands,
  COUNT(*)*1000000/_attributes.value::int as "per million"
FROM country
  INNER JOIN country_of_origin ON country.id = country_of_origin.id_to
  INNER JOIN band ON country_of_origin.id_from = band.id
  INNER JOIN _attributes ON _attributes.id = country.id
    AND _attributes.predicate = 4165 AND _attributes.datatype='INTEGER'
GROUP BY country.label_en, country.description_en, _attributes.value
ORDER BY 4 DESC;
```

7.7 Conclusion

In this chapter, we provided a low-level demonstration of our tool by converting a music subdomain of Wikidata into the relational model. We explored multiple SPARQL queries and their SQL equivalents in the produced schema, demonstrating the natural representation of Linked Data concepts in the relational model created automatically by our conversion tool.

Conclusion

In this thesis, we introduced RDF2X, a tool for automatic distributed conversion of RDF datasets to the relational model. We provided a comparison of related work in three categories – converting relational data to RDF, using RDBMS for RDF storage and converting RDF to the relational model. We reported on the conversion of 8.4 billion RDF triples from six datasets provided by the Bio2RDF project. Our tool converted 92 million triples per hour on average on a Spark cluster of 5 executors with 8GB memory and two cores. Finally, we demonstrated the contribution of our tool on two case studies – visualizing clinical trials from the ClinicalTrials.gov Bio2RDF dataset and querying a music subdomain of Wikidata with SQL.

Future work

The RDF2X tool introduced in this thesis provides all the necessary features to automatically convert a RDF dataset to the relational model. However, there is still a large potential for improvement. First of all, performance could be improved by switching to the Spark 2.x branch, merging some phases of the conversion process into one pass over the data (namely the RDF schema collection) and by implementing the CSV bulk load functionality for other RDBMS apart from PostgreSQL (SQL Server BULK INSERT, Oracle SQL*Loader, etc.). Second of all, support for more RDF concepts can be implemented, such as storing blank nodes and instances without a `rdf:type`, applying RDFS and OWL statements such as `owl:equivalentClass` or `owl:sameAs`. Thirdly, appending additional RDF statements to an existing database could be implemented, inserting new instances and relations, updating existing ones and even modifying the schema. Finally, a graphical user interface could be implemented to provide a more user-friendly experience when running locally on one machine.

Bibliography

[1] Consortium, W. W. W. RDF 1.1 concepts and abstract syntax. 2014. Available from: https://www.w3.org/TR/rdf11-concepts/

[2] Belleau, F.; Nolin, M.-A.; et al. Bio2RDF: towards a mashup to build bioinformatics knowledge systems. *Journal of biomedical informatics*, volume 41, no. 5, 2008: pp. 706–716.

[3] Fu, G.; Batchelor, C.; et al. PubChemRDF: towards the semantic annotation of PubChem compound and substance databases. *Journal of cheminformatics*, volume 7, no. 1, 2015: p. 34.

[4] Ding, L.; DiFranzo, D.; et al. Data-gov Wiki: Towards Linking Government Data. In *AAAI Spring Symposium: Linked data meets artificial intelligence*, volume 10, 2010, pp. 1–1.

[5] Capadisli, S. World Bank Linked Data. Available from: http://worldbank.270a.info/

[6] Auer, S.; Bizer, C.; et al. Dbpedia: A nucleus for a web of open data. In *The semantic web*, Springer, 2007, pp. 722–735.

[7] Bollacker, K.; Evans, C.; et al. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, AcM, 2008, pp. 1247–1250.

[8] Vrandečić, D.; Krötzsch, M. Wikidata: A Free Collaborative Knowledgebase. *Commun. ACM*, volume 57, no. 10, Sept. 2014: pp. 78–85, ISSN 0001-0782, doi:10.1145/2629489. Available from: http://doi.acm.org/10.1145/2629489

[9] Beckett, D.; Berners-Lee, T.; et al. RDF 1.1 Turtle. *World Wide Web Consortium*, 2014. Available from: https://www.w3.org/TR/2014/REC-turtle-20140225/

[10] Beckett, D. RDF 1.1 N-Triples. *W3C Recommendation*, 2014. Available from: `https://www.w3.org/TR/n-triples/`

[11] Beckett, D. RDF 1.1 N-Quads. *W3C Recommendation*, 2014. Available from: `https://www.w3.org/TR/n-quads/`

[12] Brickley, D.; Guha, R. V. RDF vocabulary description language 1.0: RDF schema. 2004.

[13] McGuinness, D. L.; Van Harmelen, F.; et al. OWL web ontology language overview. *W3C recommendation*, volume 10, no. 10, 2004: p. 2004.

[14] Heflin, J. OWL Web Ontology Language-Use Cases and Requirements. *W3C Recommendation*, volume 10, 2004: p. 12.

[15] Harris, S.; Seaborne, A.; et al. SPARQL 1.1 query language. *W3C recommendation*, volume 21, no. 10, 2013.

[16] Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM*, volume 13, no. 6, 1970: pp. 377–387.

[17] Chen, P. P.-S. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, volume 1, no. 1, 1976: pp. 9–36.

[18] DB-Engines.com, s. I. G. DBMS popularity broken down by database model. Available from: `http://db-engines.com/en/ranking_categories`

[19] Wikipedia. Comparison of web frameworks — Wikipedia, The Free Encyclopedia. 2017, [Online; accessed 30-March-2017]. Available from: `https://en.wikipedia.org/w/index.php?title=Comparison_of_web_frameworks&oldid=771443121`

[20] Kimball, R.; Ross, M. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[21] Wikipedia. Comparison of database tools — Wikipedia, The Free Encyclopedia. 2017, [Online; accessed 30-March-2017]. Available from: `https://en.wikipedia.org/w/index.php?title=Comparison_of_database_tools&oldid=767635457`

[22] Zaharia, M.; Chowdhury, M.; et al. Spark: Cluster Computing with Working Sets. *HotCloud*, volume 10, no. 10-10, 2010: p. 95.

[23] Borthakur, D.; et al. HDFS architecture guide. *Hadoop Apache Project*, volume 53, 2008.

[24] Michel, F.; Montagnat, J.; et al. *A survey of RDB to RDF translation approaches and tools.* Dissertation thesis, I3S, 2014.

[25] Halpin, H.; Herman, I. RDB2RDF Working Group Charter. 2011. Available from: `https://www.w3.org/2001/sw/rdb2rdf/`

[26] Arenas, M.; Bertails, A.; et al. A direct mapping of relational data to RDF. 2012. Available from: `https://www.w3.org/TR/rdb-direct-mapping/`

[27] Faye, D. C.; Curé, O.; et al. A survey of RDF storage approaches. *Arima Journal*, volume 15, 2012: pp. 11–35.

[28] Nitta, K.; Savnik, I. Survey of RDF storage managers. In *Proceedings of the Sixth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA'14)*, 2014, pp. 148–153.

[29] Özsu, M. T. A survey of RDF data management systems. *Frontiers of Computer Science*, volume 10, no. 3, 2016: pp. 418–432.

[30] Sakr, S.; Al-Naymat, G. Relational processing of RDF queries: a survey. *ACM SIGMOD Record*, volume 38, no. 4, 2010: pp. 23–28.

[31] Jalali, V.; Zhou, M.; et al. A study of RDB-based RDF data management techniques. In *International Conference on Web-Age Information Management*, Springer, 2011, pp. 366–378.

[32] Albahli, S.; Melton, A. RDF Data Management: A survey of RDBMS-Based Approaches. In *WIMS*, 2016, p. 31.

[33] Broekstra, J.; Kampman, A.; et al. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, Springer, 2002, pp. 54–68.

[34] Wilkinson, K.; Wilkinson, K. Jena property table implementation. 2006.

[35] Liu, X.; Thomsen, C.; et al. 3XL: An Efficient DBMS-Based Triple-Store. *2012 23rd International Workshop on Database and Expert Systems Applications*, volume 00, 2012: pp. 284–288, ISSN 1529-4188, doi: doi.ieeecomputersociety.org/10.1109/DEXA.2012.7.

[36] Pham, M.-D.; Passing, L.; et al. Deriving an emergent relational schema from rdf data. In *Proceedings of the 24th International Conference on World Wide Web*, ACM, 2015, pp. 864–874.

[37] Pham, M.-D.; Boncz, P. Exploiting Emergent Schemas to make RDF systems more efficient. In *International Semantic Web Conference*, Springer, 2016, pp. 463–479.

[38] Consortium, W. W. W.; et al. OWL 2 web ontology language document overview. 2012.

[39] Callahan, A.; Cruz-Toledo, J.; et al. Bio2RDF release 2: improved coverage, interoperability and provenance of life science linked data. In *Extended Semantic Web Conference*, Springer, 2013, pp. 200–212.

[40] Humaira, A. A Survey on Automatic Mapping of Ontology to Relational Database Schema. 2015.

[41] Vyšniauskas, E.; Nemuraitė, L.; et al. Preserving semantics of OWL 2 ontologies in relational databases using hybrid approach. *Information Technology And Control*, volume 41, no. 2, 2012: pp. 103–115.

[42] Martinez-Cruz, C.; Blanco, I. J.; et al. Ontologies versus relational databases: are they so different? A comparison. *Artificial Intelligence Review*, 2012: pp. 1–20.

[43] El-Ghalayini, H.; Odeh, M.; et al. Reverse engineering ontology to conceptual data models. *arXiv preprint cs/0412036*, 2004.

[44] Gali, A.; Chen, C. X.; et al. From ontology to relational databases. In *International Conference on Conceptual Modeling*, Springer, 2004, pp. 278–289.

[45] Vysniauskas, E.; Nemuraite, L. Transforming ontology representation from OWL to relational database. *Information technology and control*, 2006.

[46] Teswanich, W.; Chittayasothorn, S. A transformation from rdf documents and schemas to relational databases. In *Communications, Computers and Signal Processing, 2007. PacRim 2007. IEEE Pacific Rim Conference on*, IEEE, 2007, pp. 38–41.

[47] Ramanujam, S.; Gupta, A.; et al. R2D: Extracting relational structure from RDF stores. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology-Volume 01*, IEEE Computer Society, 2009, pp. 361–366.

[48] Bagui, S.; Bouressa, J. Mapping RDF and RDF-Schema to the Entity Relationship Model. *Journal of Emerging Trends in Computing and Information Sciences*, volume 5, no. 12, 2014.

[49] Stefanova, S.; Risch, T. Scalable reconstruction of RDF-archived relational databases. In *Proceedings of the Fifth Workshop on Semantic Web Information Management*, ACM, 2013, p. 5.

[50] Allocca, C.; Gougousis, A. A Preliminary Investigation of Reversing RML: From an RDF dataset to its Column-Based data source. *Biodiversity data journal*, , no. 3, 2015.

[51] Mynarz, J.; Svátek, V. Towards a Benchmark for LOD-Enhanced Knowledge Discovery from Structured Data. In *KNOW@ LOD*, 2013, pp. 41–48.

[52] Ramanujam, S.; Gupta, A.; et al. Relationalization of provenance data in complex RDF reification nodes. *Electronic Commerce Research*, volume 10, no. 3-4, 2010: pp. 389–421.

[53] Komamizu, T.; Amagasa, T.; et al. SPOOL: a SPARQL-based ETL framework for OLAP over linked data. In *Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services*, ACM, 2015, p. 49.

[54] Abelló, A.; Romero, O.; et al. Using semantic web technologies for exploratory OLAP: a survey. *IEEE transactions on knowledge and data engineering*, volume 27, no. 2, 2015: pp. 571–588.

[55] Nebot, V.; Berlanga, R. Building data warehouses with semantic web data. *Decision Support Systems*, volume 52, no. 4, 2012: pp. 853–868.

[56] Inoue, H.; Amagasa, T.; et al. An ETL framework for online analytical processing of linked open data. In *International Conference on Web-Age Information Management*, Springer, 2013, pp. 111–117.

[57] Michael Brunnbauer, n. G. RDF2RDB: Convert RDF data to relational databases. Available from: `https://github.com/michaelbrunnbauer/rdf2rdb`

[58] Pham, M.-D.; Passing, L.; et al. Deriving an emergent relational schema from rdf data. In *Proceedings of the 24th International Conference on World Wide Web*, ACM, 2015, pp. 864–874.

[59] Pellissier Tanon, T.; Vrandečić, D.; et al. From Freebase to Wikidata: The Great Migration. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, ISBN 978-1-4503-4143-1, pp. 1419–1428, doi: 10.1145/2872427.2874809. Available from: `https://doi.org/10.1145/2872427.2874809`

Appendix A

Contents of CD

- readme.txt ...... the file with CD contents description
- rdf2x_usage.pdf ...... usage documentation of RDF2X conversion tool
- docs ...... the Javadoc directory
- src ...... the thesis source code directory
- thesis ...... the thesis text directory
  - thesis.zip ...... the $\LaTeX$ source code files of the thesis
  - thesis.pdf ...... the Diploma thesis in PDF format